ETH Lecture 401-0663-00L Numerical Methods for CSE

Numerical Methods for Computational Science and Engineering

Prof. R. Hiptmair, SAM, ETH Zurich

(with contributions from Prof. P. Arbenz and Dr. V. Gradinaru)

Autumn Term 2016

URL: https://people.math.ethz.ch/~grsam/HS16/NumCSE/NumCSE16.pdf

Always under construction!

SVN revision 91232

The online version will always be a work in progress and subject to change.

(Nevertheless, the structure and the main contents can be expected to be stable)

Do not print!

Main source of information: **Lecture homepage**

Important links:

- Lecture Git repository: https://gitlab.math.ethz.ch/NumCSE/NumCSE.git
  (Clone this repository to get access to most of the C++ codes in the lecture document and homework problems. ➙ Git guide)
- Lecture recording: http://www.video.ethz.ch/lectures/d-math/2016/autumn/401-0663-00L.html
- Tablet notes: http://www.sam.math.ethz.ch/~grsam/HS16/NumCSE/NCSE16_Notes/
- Homework problems: https://people.math.ethz.ch/~grsam/HS16/NumCSE/NCSEProblems.pdf

Contents

Chapter 0

Introduction

0.0.1 Focus of this course

▷ on algorithms (principles, computational cost, scope, and limitations),

▷ on (efficient and stable) implementation in C++ based on the numerical linear algebra EIGEN (a **D**omain **S**pecific **L**anguage embedded into C++)

▷ on numerical experiments (design and interpretation).

(0.0.1) Aspects outside the scope of this course

No emphasis will be put on

- theory and proofs (unless essential for understanding of algorithms).

  ☞ 401-3651-00L Numerical Methods for Elliptic and Parabolic Partial Differential Equations
  401-3652-00L Numerical Methods for Hyperbolic Partial Differential Equations
  (both courses offered in BSc Mathematics)

- hardware aware implementation (cache hierarchies, CPU pipelining, etc.)

  ☞ 263-2300-00L How To Write Fast Numerical Code (Prof. M. Püschel, D-INFK)

- issues of high-performance computing (HPC, shard and distributed memory parallelisation, vectorization)

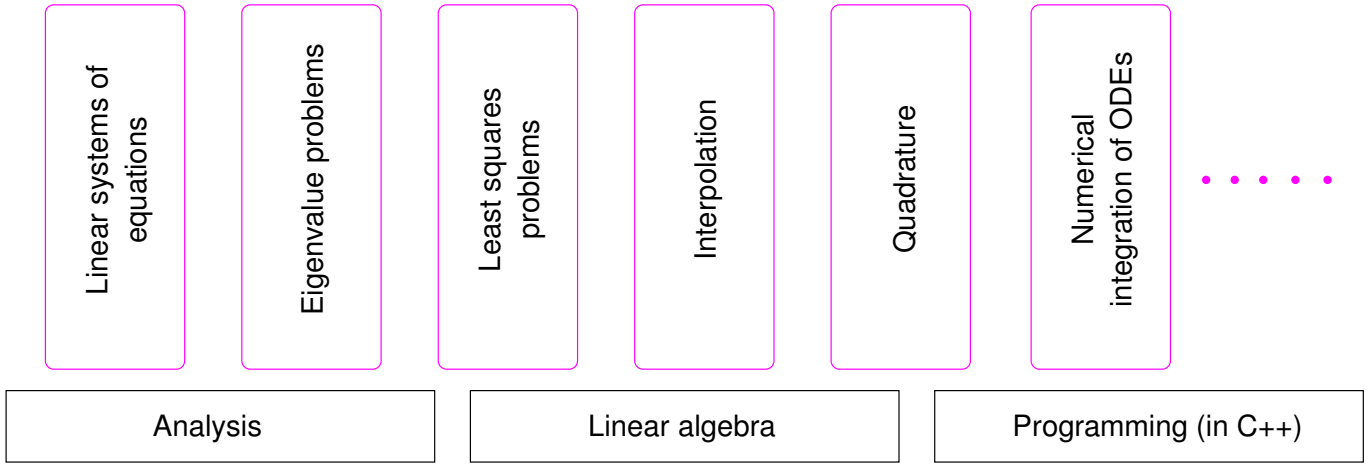
  ☞ 401-0686-10L High Performance Computing for Science and Engineering (HPCSE, Profs. M. Troyer and P. Koumoutsakos)
  263-2800-00L Design of Parallel and High-Performance Computing (Prof. T. Höfler)

However, note that these other courses partly rely on knowledge of elementary numerical methods, which is covered in this course.

Contents

(0.0.2) Prequisites

This course will take for granted basic knowledge of linear algebra, calculus, and programming, that you should have acquired during your first year at ETH.



(0.0.3) Numerical methods: A motley toolbox

This course discusses **elementary numerical methods and techniques**

They are vastly different in terms of ideas, design, analysis, and scope of application. They are the items in a **toolbox**, some only loosely related by the common purpose of being building blocks for codes for numerical simulation.

Do not expect much coherence between the chapters of this course!

Fig. 1

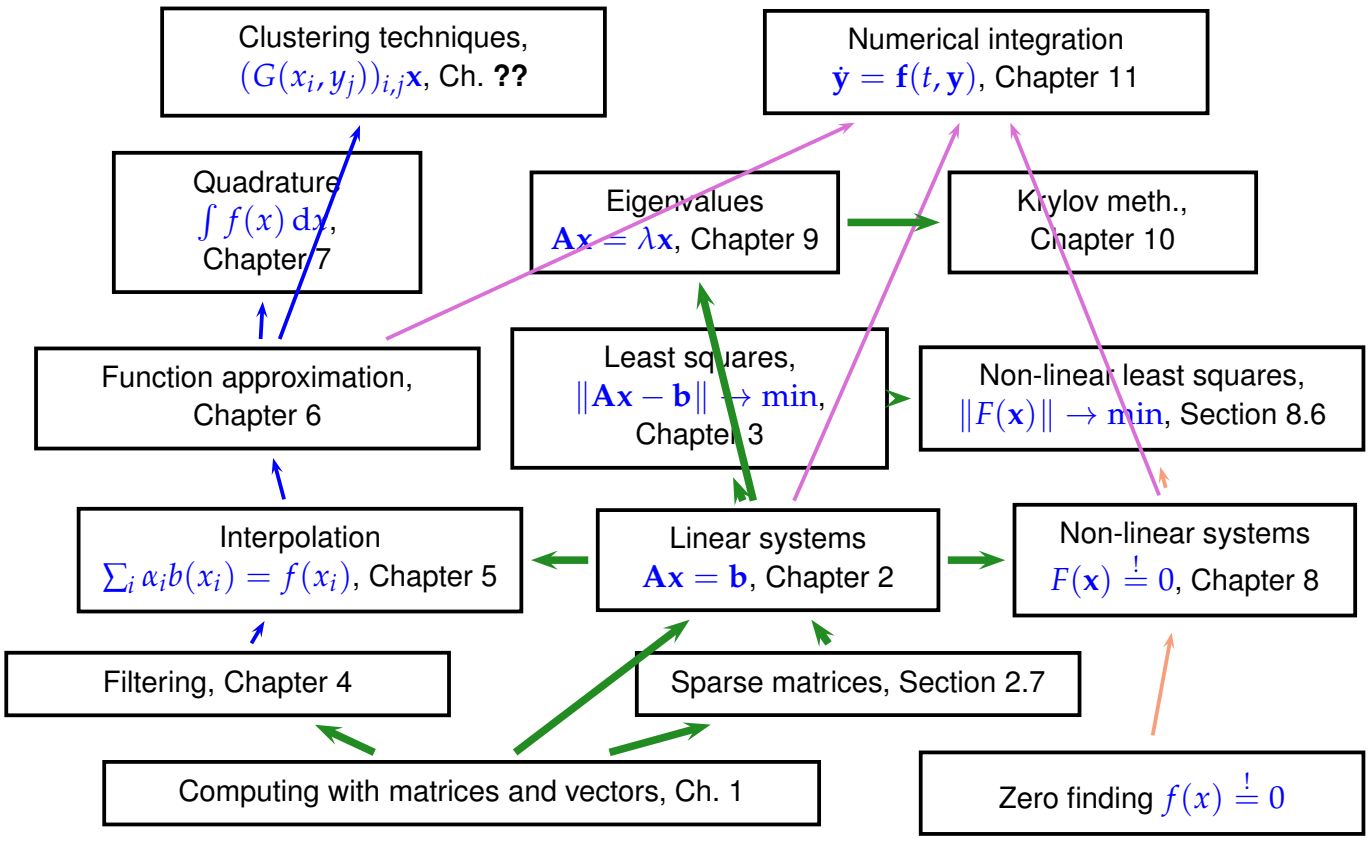
A purpose-oriented notion of "Numerical methods for CSE":

A: "Stop putting a hammer, a level, and duct tape in one box! They have nothing to do with each other!"

B: "I might need any of these tools when fixing something about the house"

(0.0.4) Dependencies of topics

Despite the diverse nature of the individual topics covered in this course, some depend on others for providing essential building blocks. The following directed graph tries to capture these relationships. The arrows have to be read as "uses results or algorithms of".



Any one-semester course "Numerical methods for CSE" will cover only selected chapters and sections of this document. Only topics addressed in **class** or in **homework problems** will be relevant for the final exam!

(0.0.5) Relevance of this course

I am a student of computer science. After the exam, may I safely forget everything I have learned in this mandatory "numerical methods" course? **No**, because it is highly likely that other courses or projects will rely on the contents of this course:

- singular value decomposition, least squares ⇒ Computational statistics, machine learning
- function approximation, numerical quadrature, numerical integration ⇒ Numerical methods for PDEs
- interpolation, least squares ⇒ Computer graphics
- eigensolvers, sparse linear systems ⇒ Graph theoretic algorithms
- numerical integration ⇒ Computer animation

and many more applications of fundamental numerical methods ....

Hardly anyone will need everything covered in this course, but *most of you will need something*.

0.0.2 Goals

- Knowledge of the fundamental algorithms in numerical mathematics
- Knowledge of the essential terms in numerical mathematics and the techniques used for the analysis of numerical algorithms
- Ability to choose the appropriate numerical method for concrete problems
- Ability to interpret numerical results
- Ability to implement numerical algorithms efficiently in C++ using numerical libraries

Indispensable:	Learning by doing (➔ exercises)

0.0.3 To avoid misunderstandings . . .

(0.0.6) "Lecture notes"

These course materials are neither a textbook nor comprehensive lecture notes.
They are meant to be supplemented by explanations given in class.

Some pieces of advice:

- the lecture material is not designed to be self-contained, but is to be studied beside attending the course or watching the course videos,
- this document is not meant for mere reading, but for working with,
- turn pages all the time and follow the numerous cross-references,
- study the relevant section of the course material when doing homework problems,
- study referenced literature to refresh prerequisite knowledge and for alternative presentation of the material (from a different angle, maybe), but be careful about not getting confused or distracted by information overload.

(0.0.7) Comprehension is a process . . .

- The course is difficult and demanding (*ie.* ETH level)
- Do **not** expect to understand everything in class. The average student will
  - understand about one third of the material when attending the lectures,

- understand another third when making a *serious effort* to solve the homework problems,
- hopefully understand the remaining third when studying for the examination after the end of the course.

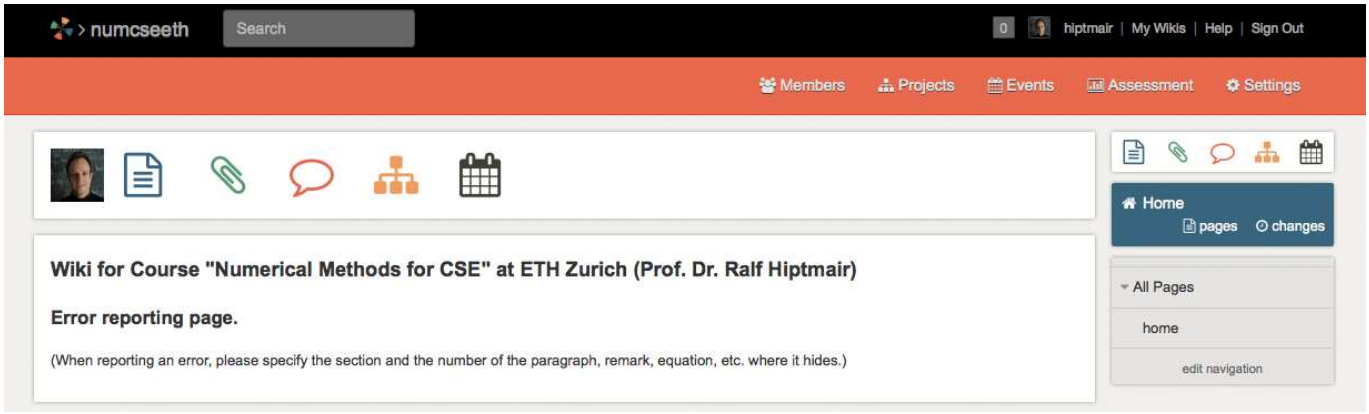
Perseverance will be rewarded!

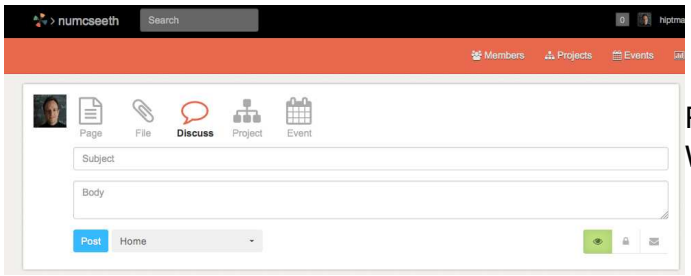
0.0.4 Reporting errors

As the documents will always be in a state of flux, they will inevitably and invariably teem with small errors, mainly typos and omissions.

Please report errors in the l,ecture material through the **Course Wiki**!

Join information will be sent to you by email.





Please point out errors by leaving a comment in the Wiki ("Discuss" menu item).

When reporting an error, please specify the section and the number of the paragraph, remark, equation, etc. where it hides. You need not give a page number.

0.0.5 Literature

Parts of the following textbooks may be used as supplementary reading for this course. References to relevant sections will be provided in the course material.

Studying extra literature is not important for following this course!

- [?] U. ASCHER AND C. GREIF, *A First Course in Numerical Methods*, SIAM, Philadelphia, 2011.

  Comprehensive introduction to numerical methods with an algorithmic focus based on MATLAB. (Target audience: students of engineering subjects)

- [?] W. DAHMEN AND A. REUSKEN, *Numerik für Ingenieure und Naturwissenschaftler*, Springer, Heidelberg, 2006.

  Good reference for large parts of this course; provides a lot of simple examples and lucid explanations, but also rigorous mathematical treatment.
  (Target audience: undergraduate students in science and engineering)
  Available for download at PDF

- [?] M. HANKE-BOURGEOIS, *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*, Mathematische Leitfäden, B.G. Teubner, Stuttgart, 2002.

  Gives detailed description and mathematical analysis of algorithms and relies on MATLAB. Profound treatment of theory way beyond the scope of this course. (Target audience: undergraduates in mathematics)

- [?] A. QUARTERONI, R. SACCO, AND F. SALERI, *Numerical mathematics*, vol. 37 of Texts in Applied Mathematics, Springer, New York, 2000.

  Classical introductory numerical analysis text with many examples and detailed discussion of algorithms. (Target audience: undergraduates in mathematics and engineering)
  Can be obtained from website.

- [?] P. DEUFLHARD AND A. HOHMANN, *Numerische Mathematik. Eine algorithmisch orientierte Einführung*, DeGruyter, Berlin, 1 ed., 1991.

  Modern discussion of numerical methods with profound treatment of theoretical aspects (Target audience: undergraduate students in mathematics).

- [?]: W.. GANDER, M.J. GANDER, AND F. KWOK, *Scientific Computing*, Text in Computational Science and Engineering, springer, 2014.

  Comprehensive treatment of elementary numerical methods with an algorithmic focus.

D-INFK maintains a webpage with links to some of these books.

Essential prerequisite for this course is a solid knowledge in linear algebra and calculus. Familiarity with the topics covered in the first semester courses is taken for granted, see

- [?] K. NIPP AND D. STOFFER, *Lineare Algebra*, vdf Hochschulverlag, Zürich, 5 ed., 2002.
- [?] M. GUTKNECHT, *Lineare algebra*, lecture notes, SAM, ETH Zürich, 2009, available online.
- [?] M. STRUWE, *Analysis für Informatiker*. Lecture notes, ETH Zürich, 2009, available online.

0.1 Specific information

0.1.1 Assistants and exercise classes

Lecturer:	Prof. Ralf Hiptmair	HG G 58.2,	☎ 044 632 3404,	hiptmair@sam.math.ethz.ch
Assistants:	Daniele Casati,	HG E 62.2,	☎ 044 632 ????,	daniele.casati@sam.math.ethz.ch
	Filippo Leonardi,	HG J 45,	☎ 044 633 9379,	filippo.leonardi@sam.math.ethz.ch
	Daniel Hupp,			huppd@inf.ethz.ch
	Marija Kranjčević,			marija.kranjcevic@inf.ethz.ch
	Heinekamp Sebastian,			heinekas@student.ethz.ch
	Hillebrand Fabian,			hifabian@student.ethz.ch
	Schaffner Yannick,			schyanni@student.ethz.ch
	Thomas Graf,			thomas.graf@student.ethz.ch
	Schwarz Fabian,			fschwarz@student.ethz.ch
	Accaputo Giuseppe,			accaputg@student.ethz.ch
	Baumann Christian,			chbaumann@student.ethz.ch
	Romero Francisco,			francisco.romero@sam.math.ethz.ch
	Dabrowski Alexander,			alexander.dabrowski@sam.math.ethz.ch
	Luca Mondada,			lmondada@student.ethz.ch
	Varghese Alexander ,			xandeepv@student.ethz.ch
	Xandeep,			

Though the assistants email addresses are provided above, their use should be restricted to cases of emergency:

> In general refrain from sending email messages to the lecturer or the assistants. They will not be answered!
>
> Questions should be asked in class (in public or during the break in private), in the tutorials, or in the study center hours.

Classes:	Thu, 08.15-10.00 (HG F 1), Fri, 13.15-16.00 (HG F 1)
Tutorials:	Mon, 10.15-12.00 (CLA E 4, LFW E 11, LFW E13, ML H 41.1, ML J 34.1)
	Mon, 13.15-15.00 (CLA E 4, HG E 33.3, HG E 33.5, HG G 26.5, LEE D 105)
Study center:	Mon, 18.00-20.00 (HG E 41)

Before the first tutorial you will receive a link where you can register to a tutorial class. Keep in mind that one tutorial will be held in German and one will be reserved for CSE students.

0.1.2 Study center

The tutorials classes for this course will be supplemented by the option to do supervised work in the ETH "flexible lecture hall" (study center) HG E 41 ▷.

Several *assistants will be present* to explain and discuss homework problems both from the previous and the current problem sheet. They are also supposed to answer questions about the course.

The study center session is also a good opportunity to *do the homework in a group*. In case you are stalled you may call an assistant and ask for advice.

0.1.3 Assignments

A steady and persistent effort spent on homework problems is essential for success in this course.

You should expect to spend 4-6 hours per week on trying to solve the homework problems. Since many involve small coding projects, the time it will take an individual student to arrive at a solution is hard to predict.

(0.1.1) Homeworks and tutors' corrections

- The weekly assignments will be a few problems from the **NCSE Problem Collection** available online as PDF. The particular problems to be solved will be communicated on **Friday** every week.

  Please note that this problem collection is being compiled during this semester. Thus, make sure that you obtain the most current version every week.
- Some or all of the problems of an assignment sheet will be discussed in the tutorial classes on Monday 10 days after the problems have been assigned.
- A few problems on each sheet will be marked as core problems. Every participant of the course is strongly advised to try and solve *at least* the core problems.
- If you want your tutor to examine your solution of the current problem sheet, please put it into the plexiglass trays in front of HG G 53/54 by the Thursday after the publication. You should submit your codes using the online submission interface. This is voluntary, but feedback on your performance on homework problems can be important.
- Please clearly mark the homework problems that you want your tutor to inspect.
- You are encouraged to hand-in incomplete and wrong solutions, you can receive valuable feedback even on incomplete attempts.

(0.1.2) Git code repository

C++ codes for both the classroom and homework problems are made available through a git repository also accessible through Gitlab (Link):

The Gitlab toplevel page gives a short introduction into the repository for the course and provides a link to online sources of information about Git.

Ddownload is possible via Git or as a zip archive. Which method you choose is up to you, but it should be noted that updating via git is more convenient.

➢ Shell command to download the git repository:

```
> git clone https://gitlab.math.ethz.ch/NumCSE/NumCSE.git
```

Updating the repository to fetch upstream changes is then possible by executing `> git pull` inside the NumCSE folder.

Note that by default participants of the course will have read access only. However, if you want to *contribute corrections and enhancements* of lecture or homework codes your are invited to submit a merge request. Beforehand you have to inform your tutor so that a personal Gitlab account can be set up for you.

The Zip-archive download link is here.

For instructions on how to compile assignments or lecture codes see the README file.

0.1.4 Information on examinations

(0.1.3) Examinations during the teaching period

From the ETH course directory:

> Computer based examination involving coding problems beside theoretical questions. Parts of the lecture documents and other materials will be made available online during the examination. A 30-minute **mid-term exam** and a 30-minute **end term exam** will be held during the teaching period on dates specified in the beginning of the semester. Points earned in these exams will be taken into account through a **bonus of up to 20%** of the total points in the final session exam.

Both will be closed book examinations on paper.

Dates:

- Mid-term: **Friday, Nov 4, 2016, 13:15**
- End-term: **Friday, Dec 23, 2016, 13:15**
- Make-up term exam: **Friday, Jan 13, 2017, 9:15**
- Repetition term exam: **Friday, May 12, 2017, 16:15**

The term exams are regarded as **central elements** and as such are graded on a **pass/fail** basis.

Admission to the main exam is conditional on passing at least one term exam

Only students who could not take part in one of the term exams for cogent reasons like illness (doctor's certificate required!) may take part in the make-up term exam. Please contact Daniele Casati (daniele.casati@sam.math.ethz.ch) by email, if you think that you are eligible for the make-up term exam, and attach all required documentation. You will be informed, whether you are admitted.

Only students who have failed both term exams can take part in the repetition term exam in spring next year. This is their only chance to be admitted to the main exam in Summer 2017.

(0.1.4) Main examination during exam session

- Three-hour written examination involving coding problems to be done at the computer on

  Thursday January 26, 2017, 9:00 - 12:00, HG G 1

- Dry-run for computer based examination:

  TBA, registration via course website

- Subjects of examination:
  - All topics, which have been addressed in class or in a homework problem (including the homework problems not labelled as "core problems")
- Lecture documents will be available as PDF during the examination. The corresponding final version of the lecture documents will be made available on TBA
- You may bring a **summary of up to 10 pages A4** in your own handwriting. No printouts and copies are allowed.
- The exam questions will be asked in English.

(0.1.5) Repeating the main exam

- Everybody who passed at least one of the term exams, the make-up term exam, or the repetition term exam for last year's course and wants to repeat the main exam, will be allowed to do so.
- Bonus points earned in term exams in last year's course can be taken into account for this course's main exam.

- If you are going to repeat the main exam, but also want to earn a bonus through this year's term exams, please declare this intention before the mid-term exam.

0.2 Programming in C++11

C++11 is the *current* ANSI/ISO standard for the programming language C++. On the one hand, it offers a wealth of features and possibilities. On the other hand, this can be confusing and even be prone to inconsistencies. A major cause of inconsistent design is the requirement with backward compatibility with the C programming language and the earlier standard C++98.

However, C++ has become the main language in computational science and engineering and high performance computing. Therefore this course relies on C++ to discuss the implementation of numerical methods.

In fact C++ is a blend of different programming paradigms:

- an object oriented core providing classes, inheritance, and runtime polymorphism,
- a powerful template mechanism for parametric types and partial specialization, enabling *template meta-programming* and compile-time polymorphism,
- a collection of abstract data containers and basic algorithms provided by the Standard Template Libary (STL).

Supplementary reading. A popular book for learning C++ that has been upgraded to include the latest C++11 standard is [**?**].

The book [**?**] gives a comprehensive presentation of the new features of C++11 compared to earlier versions of C++.

There are plenty of online reference pages for C++, for instance `http://en.cppreference.com` and `http://www.cplusplus.com/`.

(0.2.1) Building, compiling, and debugging

- We use the command line build tool CMAKE, see web page.
- The compilers supporting all features of C++ needed for this course, are **clang** and **GCC**. Both are open source projects and free. CMAKE will automatically select a suitable compiler on your system (Linux or Mac OS X).
- A command line tool for debugging is **lldb**, see short introduction by Till Ehrengruber, student of CSE@ETH.

The following sections highlight a few particular aspects of C++11 that may be important for code development in this course.

0.2.1 Function Arguments and Overloading

(0.2.2) Function overloading

Argument types are an integral part of a function declaration in C++. Hence the following functions are different

```
int* f(int);      // use this in the case of a single numeric argument
double f(int *); // use only, if pointer to a integer is given
void f(const MyClass &);  // use when called for a MyClass object
```

and the compiler selects the function to be used depending on the type of the arguments following rather sophisticated rules, refer to overload resolution rules. Complications arise, because implicit type conversions have to be taken into account. In case of ambiguity a compile-time error will be triggered. Functions cannot be distinguished by return type!

For member functions (methods) of classes an additional distinction can be introduced by the **const** specifier:

```
struct MyClass {
  double f(double);       // use for a mutable object of type MyClass
  double f(double) const; // use this version for a constant object
  ...
};
```

The second version of the method `f` is invoked for *constant objects* of type **MyClass**.

(0.2.3) Operator overloading

In C++ unary and binary operators like `=`, `==`, `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`, `%`, `&&`, `||`, etc. are regarded as functions with a fixed number of arguments (one or two). For built-in numeric and logic types they are defined already. They can be extended to any other type, for instance

```
MyClass operator +(const MyClass &,const MyClass &);
MyClass operator +(const MyClass &,double);
MyClass operator +(const MyClass &);   // unary + !
```

The same selection rules as for function overloading apply. Of course, operators can also be introduced as class member functions.

C++ gives complete freedom to overload operators. However, the semantics of the new operators should be close to the customary use of the operator.

(0.2.4) Passing arguments by value and by reference

Consider a generic function declared as follows:

```
void f(MyClass x); // Argument x passed by value.
```

When `f` is invoked, a *temporary copy* of the argument is created through the copy constructor or the move constructor of **MyClass**. The new temporary object is a *local variable* inside the function body.

When a function is declared as follows

```
void f(MyClass &x); // Argument x passed by reference.
```

then the argument is passed to the scope of the function and can be changed inside the function. *No copies* are created. If one wants to avoid the creation of temporary objects, which may be costly, but also wants to indicate that the argument will not be modified inside `f`, then the declaration should read

```
void f(const MyClass &x); // Argument x passed by constant referene.
```

New in C++11 is move semantics, enabled in the following definition

```
void f(const MyClass &&x); // Optional shallow copy
```

In this case, if the scope of the object passed as the argument is merely the function or **std::move()** tags it as disposable, the move constructor of **MyClass** is invoked, which will usually do a *shallow copy* only. Refer to Code 0.2.22 for an example.

0.2.2 Templates

(0.2.5) Function templates

The template mechanism supports parameterization of definitions of classes and functions by type. An example of a function templates is

```
template <typename ScalarType,typename VectorType>
VectorType saxpy(ScalarType alpha,const VectorType &x,const
   VectorType &y)
  { return(alpha*x+y); }
```

Depending on the concrete type of the arguments the compiler will instantiate particular versions of this function, for instance `saxpy<float,double>`, when `alpha` is of type `float` and both `x` and `y` are of type `double`. In this case the return type will be float.

For the above example the compiler will be able to deduce the types `ScalarType` and `VectorType` from the arguments. The programmer can also specify the types directly through the `< >`-syntax as in

```
saxpy<double,double>(a,x,y);
```

If an instantiation for all arguments of type `double` is desired. In case, the arguments do not supply enough information about the type parameters, specifying (some of) them through `< >` is mandatory.

(0.2.6) Class templates

A class template defines a class depending on one or more type parameters, for instance

```
template <typename T>
class MyClsTempl {
public:
  using parm_t = typename T::value_t; // T-dependent type
  MyClsTempl(void);          // Default constructor
  MyClsTempl(const T&);      // Constructor with an argument
  template <typename U>
  T memfn(const T&,const U&) const; // Templated member function
private:
  T *ptr;   // Data member, T-pointer
};
```

Types **MyClsTempl<T>** for a concrete choice of **T** are instantiated when a corresponding object is declared, for instance via

```
double x = 3.14;
MyClass myobj; // Default construction of an object
MyClsTempl<double> tinstd; // Instantiation for T = double
MyClsTempl<MyClass> mytinst(myobj); // Instantiation for T = MyClass
MyClass ret = mytinst.memfn(myobj,x); // Instantiation of member
   function for U = double, automatic type deduction
```

The types spawned by a template for different parameter types have nothing to do with each other.

Requirements on parameter types

The parameter types for a template have to provide all type definitions, member functions, operators, and data to make possible the instantiation ("compilation") of the class of function template.

0.2.3 Function Objects and Lambda Functions

A function object is an object of a type that provides an overloaded "function call" **operator** `()`. Function objects can be implemented in two different ways:

(I) through special classes like the following that realizes a function $\mathbb{R} \mapsto \mathbb{R}$

```
class MyFun {
public:
  ...
  double operator (double x) const; // Evaluation operator
  ...
};
```

The evaluation operator can take more than one argument and need not be declared `const`.

(II) through lambda functions, an "anonymous function" defined as

```
[<capture list>] (<arguments>) -> <return type> { body; }
```

where <capture list> is a list of variables from the local scope to be passed to the lambda function; an `&` indicates passing by reference,
<arguments> is a comma separated list of function arguments complete with types,
<return type> is an *optional* return type; often the compiler will be able to deduce the return type from the definition of the function.

Function classes should be used, when the function is needed in different places, whereas lambda functions for short functions intended for single use.

C++11 code 0.2.8: Demonstration of use of lambda function

```
int main() {
  // initialize a vector from an initializer list
  std::vector<double> v({1.2,2.3,3.4,4.5,5.6,6.7,7.8});
  // A vector of the same length
  std::vector<double> w(v.size());
  // Do cumulative summation of v and store result in w
  double sum = 0;
  std::transform(v.begin(),v.end(),w.begin(),
                 [&sum] (double x) { sum += x; return sum;});
  cout << "sum = " << sum << ", w = [ ";
  for(auto x: w) cout << x << ' '; cout << ']' << endl;
  return(0);
};
```

In this code the lambda function captures the local variable `sum` by reference, which enables the lambda function to change its value in the surrounding scope.

(0.2.9) Function type wrappers

The special class `std::function` provides types for general polymorphic function wrappers.

```
std::function<return type(arg types)>
```

C++11 code 0.2.10: Use of `std::function`

```
double binop(double arg1,double arg2) { return (arg1/arg2); }

void stdfunctiontest(void) {
  // Vector of objects of a particular signature
  std::vector<std::function<double(double,double)>> fnvec;
  // Store reference to a regular function
  fnvec.push_back(binop);
  // Store are lambda function
  fnvec.push_back([] (double x,double y) -> double { return y/x; });
  for (auto fn : fnvec) { std::cout << fn(3,2) << std::endl; }
}
```

In this example an object of type `std::function<double(double,double)>` can hold a regular function taking two **double** arguments and returning another **double** or a lambda function with the same signature. Guess the output of `stdfunctiontest`!

0.2.4 Multiple Return Values

In MATLAB it is customary to return several variables from a function call:

```
function [x,y,z] = f(a,b)
```

In C++ this is possible by using the tuple utility. For instance, the following function computes the mimimal and maximal element of a vector and also returns its cumulative sum. It returns all these values.

C++11 code 0.2.11: Function with multiple return values

```
template<typename T>
std::tuple<T,T,std::vector<T>> extcumsum(const std::vector<T> &v) {
  // Local summation variable captured by reference by lambda function
  T sum{};
  // temporary vector for returning cumulative sum
  std::vector<T> w{};
  // cumulative summation
  std::transform(v.cbegin(),v.cend(),back_inserter(w),
                 [&sum] (T x) { sum += x; return(sum); });
  return(std::tuple<T,T,std::vector<T>>
    (*std::min_element(v.cbegin(),v.cend()),
     *std::max_element(v.cbegin(),v.cend()),w));
}
```

This code snippet shows how to extract the individual components of the tuple returned by the previous function.

C++11 code 0.2.12: Calling a function with multiple return values

```
int main () {
  // initialize a vector from an initializer list
  std::vector<double> v({1.2,2.3,3.4,4.5,5.6,6.7,7.8});
  // Variables for return values
  double minv,maxv;  // Extremal elements
  std::vector<double> cs; // Cumulative sums
  std::tie(minv,maxv,cs) = extcumsum(v);
  cout << "min = " << minv << ", max = " << maxv << endl;
  cout << "cs = [ "; for(double x: cs) cout << x << ' '; cout << "]"
     << endl;
  return(0);
}
```

Be careful: many temporary objects might be created! A demonstration of this hidden cost is given in Exp. 0.2.39.

0.2.5 A Vector Class

Since C++ is an object oriented programming language, datatypes defined by classes play a pivotal role in every C++ program. Here, we demonstrate the main ingredients of a class definition and other important facilities of C++ for the class **MyVector** meant for objects representing vectors from $\mathbb{R}^n$. The codes can be found in ↪ GITLAB.

C++11 class 0.2.13: Definition of a simple vector class MyVector

```
namespace myvec {
class MyVector {
public:
  using value_t = double;
  // Constructor creating constant vector, also default constructor
  explicit MyVector(std::size_t n = 0,double val = 0.0);
  // Constructor:  initialization from an STL container
  template <typename Container>  MyVector(const Container &v);
  // Constructor:  initialization from an STL iterator range
  template <typename Iterator> MyVector(Iterator first,Iterator last);
  // Copy constructor, computational cost O(n)
  MyVector(const MyVector &mv);
  // Move constructor, computational cost O(1)
  MyVector(MyVector &&mv);
  // Copy assignment operator, computational cost O(n)
  MyVector &operator = (const MyVector &mv);
  // Move assignment operator, computational cost O(1)
  MyVector &operator = (MyVector &&mv);
  // Destructor
  virtual ~MyVector(void);
  // Type conversion to STL vector
  operator std::vector<double> () const;

  // Returns length of vector
  std::size_t length(void) const { return n; }
  // Access operators:  rvalue & lvalue, with range check
  double operator [] (std::size_t i) const;
  double &operator [] (std::size_t i);
  // Comparison operators
  bool operator == (const MyVector &mv) const;
  bool operator != (const MyVector &mv) const;
  // Transformation of a vector by a function R -> R
  template <typename Functor>
  MyVector &transform(Functor &&f);

  // Overloaded arithmetic operators
  // In place vector addition:  x += y;
  MyVector &operator +=(const MyVector &mv);
  // In place vector subtraction:  x-= y;
  MyVector &operator -=(const MyVector &mv);
  // In place scalar multiplication:  x *= a;
  MyVector &operator *=(double alpha);
```

```
43   // In place scalar division:  x /= a;
44   MyVector &operator /=(double alpha);
45   // Vector addition
46   MyVector operator + (MyVector mv) const;
47   // Vector subtraction
48   MyVector operator - (const MyVector &mv) const;
49   // Scalar multiplication from right and left:  x = a*y; x = y*a
50   MyVector operator * (double alpha) const;
51   friend MyVector operator * (double alpha,const MyVector &);
52   // Scalar divsion:  x = y/a;
53   MyVector operator / (double alpha) const;
54
55   // Euclidean norm
56   double norm(void) const;
57   // Euclidean inner product
58   double operator *(const MyVector &) const;
59   // Output operator
60   friend std::ostream &
61   operator << (std::ostream &,const MyVector &mv);
62
63   static bool dbg; // Flag for verbose output
64 private:
65   std::size_t n; // Length of vector
66   double *data;  // data array (standard C array)
67 };
68 }
```

Note the use of a public static data member `dbg` in Line 63 that can be used to control debugging output by setting `MyVector::dbg = true` or `MyVector::dbg = false`.

Remark 0.2.14 (Contiguous arrays in C++)

The class **MyVector** uses a C-style array and dynamic memory management with **new** and **delete** to store the vector components. This is for demonstration purposes only and not recommended.

Arrays in C++

In C++ use the STL container **std::vector<T>** for storing data in contiguous memory locations.

(0.2.16) Member and friend functions of MyVector

C++11 code 0.2.17: Constructor for constant vector, also default constructor, see Line 28

```
1 MyVector::MyVector(std::size_t _n,double _a):n(_n),data(nullptr) {
2   if (dbg) cout << "{Constructor MyVector(" << _n
3                 << ") called" << '}' << endl;
```

```
4   if (n > 0) data = new double [_n];
5   for (std::size_t l=0;l<n;++l) data[l] = _a;
6 }
```

This constructor can also serve as default constructor (a constructor that can be invoked without any argument), because defaults are supplied for all its arguments.

The following two constructors initialize a vector from sequential containers according to the conventions of the STL.

C++11 code 0.2.18: Templated constructors copying vector entries from an STL container

```
1 template <typename Container>
2 MyVector::MyVector(const Container &v):n(v.size()),data(nullptr) {
3   if (dbg) cout << "{MyVector(length " << n
4                 << ") constructed from container" << '}' << endl;
5   if (n > 0) {
6     double *tmp = (data = new double [n]);
7     for(auto i: v) *tmp++ = i;  // foreach loop
8   }
9 }
```

Note the use of the new C++11 facility of a "foreach loop" iterating through a container in Line 7.

C++11 code 0.2.19: Constructor initializing vector from STL iterator range

```
1 template <typename Iterator>
2 MyVector::MyVector(Iterator first,Iterator last):n(0),data(nullptr) {
3   n = std::distance(first,last);
4   if (dbg) cout << "{MyVector(length " << n
5                 << ") constructed from range" << '}' << endl;
6   if (n > 0) {
7     data = new double [n];
8     std::copy(first,last,data);
9   }
10 }
```

The use of these constructors is demonstrated in the following code

C++11 code 0.2.20: Initialization of a MyVector object from an STL vector

```
1 int main() {
2   myvec::MyVector::dbg = true;
3   std::vector<int> ivec = { 1,2,3,5,7,11,13 }; // initializer list
4   myvec::MyVector v1(ivec.cbegin(),ivec.cend());
5   myvec::MyVector v2(ivec);
6   myvec::MyVector vr(ivec.crbegin(),ivec.crend());
7   cout << "v1 = " << v1 << endl;
8   cout << "v2 = " << v2 << endl;
9   cout << "vr = " << vr << endl;
```

```
10   return(0);
11 }
```

The following output is produced:

```
{MyVector(length 7) constructed from range}
{MyVector(length 7) constructed from container}
{MyVector(length 7) constructed from range}
v1 = [ 1,2,3,5,7,11,13 ]
v2 = [ 1,2,3,5,7,11,13 ]
vr = [ 13,11,7,5,3,2,1 ]
{Destructor for MyVector(length = 7)}
{Destructor for MyVector(length = 7)}
{Destructor for MyVector(length = 7)}
```

The copy constructor listed next relies on the *STL algorithm* **std::**`copy` to copy the elements of an existing object into a newly created object. This takes n operations.

C++11 code 0.2.21: Copy constructor

```
1 MyVector::MyVector(const MyVector &mv):n(mv.n),data(nullptr) {
2   if (dbg) cout << "{Copy construction of MyVector(length "
3                 << n << ")" << '}' << endl;
4   if (n > 0) {
5     data = new double [n];
6     std::copy_n(mv.data,n,data);
7   }
8 }
```

An important new feature of C++11 is move semantics which helps avoid expensive copy operations. The following implementation just performs a shallow copy of pointers and, thus, for large n is much cheaper than a call to the copy constructor from Code 0.2.21. The source vector is left in an empty vector state.

C++11 code 0.2.22: Move constructor

```
1 MyVector::MyVector(MyVector &&mv):n(mv.n),data(mv.data) {
2   if (dbg) cout << "{Move construction of MyVector(length "
3                 << n << ")" << '}' << endl;
4   mv.data = nullptr; mv.n = 0; // Reset victim of data theft
5 }
```

The following code demonstrates the use of **std::move()** to mark a vector object as disposable and allow the compiler the use of the move constructor. The code also uses left multiplication with a scalar, see Code 0.2.35.

C++11 code 0.2.23: Invocation of copy and move constructors

```
1 int main() {
2   myvec::MyVector::dbg = true;
3   myvec::MyVector v1(std::vector<double>(
```

```
    {1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9}));
  myvec::MyVector v2(2.0*v1); // Scalar multiplication
  myvec::MyVector v3(std::move(v1));
  cout << "v1 = " << v1 << endl;
  cout << "v2 = " << v2 << endl;
  cout << "v3 = " << v3 << endl;
  return (0);
}
```

This code produces the following output. We observe that `v1` is empty after its data have been "stolen" by `v2`.

```
{MyVector(length 8) constructed from container}
{operator a*, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator *=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{Destructor for MyVector(length = 0)}
{Move construction of MyVector(length 8)}
v1 = [ ]
v2 = [ 2.4,4.6,6.8,9,11.2,13.4,15.6,17.8 ]
v3 = [ 1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9 ]
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 0)}
```

We observe that the object `v1` is reset after having been moved to `v3`.

Use `std::move` only for special purposes like above and only if an object has a move constructor. Otherwise a 'move' will trigger a plain copy operation. In particular, do not use `std::move` on objects at the end of their scope, e.g., within `return` statements.

The next operator effects copy assignment of an rvalue **MyVector** object to an lvalue **MyVector**. This involves $O(n)$ operations.

C++11 code 0.2.24: Copy assignment operator

```
MyVector &MyVector::operator = (const MyVector &mv) {
  if (dbg) cout << "{Copy assignment of MyVector(length "
                << n << "<-" << mv.n << ")"<< '}' << endl;
  if (this == &mv) return(*this);
  if (n != mv.n) {
    n = mv.n;
    if (data != nullptr) delete [] data;
    if (n > 0) data = new double [n]; else data = nullptr;
  }
  if (n > 0) std::copy_n(mv.data,n,data);
  return(*this);
}
```

The move semantics is realized by an assignment operator relying on shallow copying.

C++11 code 0.2.25: Move assignment operator

```
MyVector &MyVector::operator = (MyVector &&mv) {
  if (dbg) cout << "{Move assignment of MyVector(length "
               << n << "<-" << mv.n << ")"<< '}' << endl;
   if (data != nullptr) delete [] data;
   n = mv.n; data = mv.data;
   mv.n = 0; mv.data = nullptr;
   return(*this);
}
```

The destructor releases memory allocated by **new** during construction or assignment.

C++11 code 0.2.26: Destructor: releases allocated memory

```
MyVector::~MyVector(void) {
  if (dbg) cout << "{Destructor for MyVector(length = "
               << n << ")" << '}' << endl;
  if (data != nullptr) delete [] data;
}
```

The **operator** keyword is also use to define implicit type conversions.

C++11 code 0.2.27: Type conversion operator: copies contents of vector into STL vector

```
MyVector::operator std::vector<double> () const {
  if (dbg) cout << "{Conversion to std::vector, length = " << n <<
     '}' << endl;
  return std::vector<double>(data,data+n);
}
```

The bracket operator `[]` can be used to fetch and set vector components. Note that index range checking is performed; an *exception* is thrown for invalid indices. The following code also gives an example of operator overloading as discussed in § 0.2.3.

C++11 code 0.2.28: rvalue and lvalue access operators

```
double MyVector::operator [] (std::size_t i) const {
  if (i >= n) throw(std::logic_error("[] out of range"));
  return data[i];
}

double &MyVector::operator [] (std::size_t i) {
  if (i >= n) throw(std::logic_error("[] out of range"));
  return data[i];
}
```

Componentwise direct comparison of vectors. Can be dangerous in numerical codes,*cf.* Rem. 1.5.36.

C++11 code 0.2.29: Comparison operators

```
bool MyVector::operator == (const MyVector &mv) const
{
  if (dbg) cout << "{Comparison ==: " << n << " <-> " << mv.n << '}'
     << endl;
  if (n != mv.n) return(false);
  else {
    for(std::size_t l=0;l<n;++l)
      if (data[l] != mv.data[l]) return(false);
  }
  return(true);
}

bool MyVector::operator != (const MyVector &mv) const {
  return !(*this == mv);
}
```

The `transform` method applies a function to every vector component and overwrites it with the value returned by the function. The function is passed as an object of a type providing a `()`-operator that accepts a single argument convertible to `double` and returns a value convertible to `double`.

C++11 code 0.2.30: Transformation of a vector through a functor `double` → `double`

```
template <typename Functor>
MyVector &MyVector::transform(Functor &&f) {
  for(std::size_t l=0;l<n;++l) data[l] = f(data[l]);
  return(*this);
}
```

The following code demonstrates the use of the `transform` method in combination with

1. a function object of the following type

C++11 code 0.2.31:

```
struct SimpleFunction {
  SimpleFunction(double _a = 1.0):cnt(0),a(_a) {}
  double operator () (double x) { cnt++; return(x+a); }
  int cnt;          // internal counter
  const double a;  // increment value
};
```

2. a lambda function defined directly inside the call to `transform`.

C++11 code 0.2.32:

```
int main() {
  myvec::MyVector::dbg = false;
  double a = 2.0; // increment
```

```
  int cnt = 0;    // external counter used by lambda function
  myvec::MyVector mv(std::vector<double>(
    {1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9}));
  mv.transform([a,&cnt] (double x) { cnt++; return(x+a); });
  cout << cnt << " operations, mv transformed = " << mv << endl;
  SimpleFunction trf(a); mv.transform(trf);
  cout << trf.cnt << " operations, mv transformed = " << mv << endl;
  mv.transform(SimpleFunction(-4.0));
  cout << "Final vector = " << mv << endl;
  return(0);
}
```

The output is

```
8 operations, mv transformed = [ 3.2,4.3,5.4,6.5,7.6,8.7,9.8,10.9 ]
8 operations, mv transformed = [ 5.2,6.3,7.4,8.5,9.6,10.7,11.8,12.9 ]
Final vector = [ 1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9 ]
```

Operator overloading provides the "natural" vector operations in $\mathbb{R}^n$ both in place and with a new vector created for the result.

C++11 code 0.2.33: In place arithmetic operations (one argumnt)

```
MyVector &MyVector::operator +=(const MyVector &mv) {
  if (dbg) cout << "{operator +=, MyVector of length "
               << n << '}' << endl;
  if (n != mv.n) throw(std::logic_error("+=: vector size mismatch"));
  for(std::size_t l=0;l<n;++l) data[l] += mv.data[l];
  return(*this);
}

MyVector &MyVector::operator -=(const MyVector &mv) {
  if (dbg) cout << "{operator -=, MyVector of length "
               << n << '}' << endl;
  if (n != mv.n) throw(std::logic_error("-=: vector size mismatch"));
  for(std::size_t l=0;l<n;++l) data[l] -= mv.data[l];
  return(*this);
}

MyVector &MyVector::operator *=(double alpha) {
  if (dbg) cout << "{operator *=, MyVector of length "
               << n << '}' << endl;
  for(std::size_t l=0;l<n;++l) data[l] *= alpha;
  return(*this);
}

MyVector &MyVector::operator /=(double alpha) {
  if (dbg) cout << "{operator *=, MyVector of length "
               << n << '}' << endl;
  for(std::size_t l=0;l<n;++l) data[l] /= alpha;
  return(*this);
```

```
29 }
```

C++11 code 0.2.34: Binary arithmetic operators (two arguments)

```
1  MyVector MyVector::operator + (MyVector mv) const {
2    if (dbg) cout << "{operator +, MyVector of length "
3                  << n << '}' << endl;
4    if (n != mv.n) throw(std::logic_error("+: vector size mismatch"));
5    mv += *this;
6    return(mv);
7  }
8
9  MyVector MyVector::operator - (const MyVector &mv) const {
10   if (dbg) cout << "{operator +, MyVector of length "
11                 << n << '}' << endl;
12   if (n != mv.n) throw(std::logic_error("+: vector size mismatch"));
13   MyVector tmp(*this); tmp -= mv;
14   return(tmp);
15 }
16
17 MyVector MyVector::operator * (double alpha) const {
18   if (dbg) cout << "{operator *a, MyVector of length "
19                 << n << '}' << endl;
20   MyVector tmp(*this); tmp *= alpha;
21   return(tmp);
22 }
23
24 MyVector MyVector::operator / (double alpha) const {
25   if (dbg) cout << "{operator /, MyVector of length " << n << '}' <<
       endl;
26   MyVector tmp(*this); tmp /= alpha;
27   return(tmp);
28 }
```

C++11 code 0.2.35: Non-member function for left multiplication with a scalar

```
1  MyVector operator * (double alpha,const MyVector &mv) {
2    if (MyVector::dbg) cout << "{operator a*, MyVector of length "
3                            << mv.n << '}' << endl;
4    MyVector tmp(mv); tmp *= alpha;
5    return(tmp);
6  }
```

C++11 code 0.2.36: Euclidean norm

```
1  double MyVector::norm(void) const {
2    if (dbg) cout << "{norm: MyVector of length " << n << '}' << endl;
3    double s = 0;
```

```
4   for(std::size_t l=0;l<n;++l) s += (data[l]*data[l]);
5   return(std::sqrt(s));
6 }
```

Adopting the notation in some linear algebra texts, the operator $*$ has been chosen to designate the Euclidean inner product:

C++11 code 0.2.37: Euclidean inner product

```
1 double MyVector::operator *(const MyVector &mv) const {
2   if (dbg) cout << "{dot *, MyVector of length " << n << '}' << endl;
3   if (n != mv.n) throw(std::logic_error("dot: vector size mismatch"));
4   double s = 0;
5   for(std::size_t l=0;l<n;++l) s += (data[l]*mv.data[l]);
6   return(s);
7 }
```

At least for debugging purposes every reasonably complex class should be equipped with output functionality.

C++11 code 0.2.38: Non-member function output operator

```
1 std::ostream &operator << (std::ostream &o,const MyVector &mv) {
2   o << "[ ";
3   for(std::size_t l=0;l<mv.n;++l)
4     o << mv.data[l] << (l==mv.n-1?' ':',');
5   return(o << "]");
6 }
```

Experiment 0.2.39 ("Behind the scenes" of MyVector arithmetic)

The following code highlights the use of operator overloading to obtain readable and compact expressions for vector arithmetic.

C++11 code 0.2.40:

```
1 int main() {
2   myvec::MyVector::dbg = true;
3   myvec::MyVector
      x(std::vector<double>({1.2,2.3,3.4,4.5,5.6,6.7,7.8,8.9}));
4   myvec::MyVector
      y(std::vector<double>({2.1,3.2,4.3,5.4,6.5,7.6,8.7,9.8}));
5   auto z = x+(x*y)*x+2.0*y/(x-y).norm();
6 }
```

We run the code and trace calls. This is printed to the console:

```
{MyVector(length 8) constructed from container}
{MyVector(length 8) constructed from container}
{dot *, MyVector of length 8}
{operator a*, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator *=, MyVector of length 8}
{operator +, MyVector of length 8}
{operator +=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{operator a*, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator *=, MyVector of length 8}
{operator +, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator -=, MyVector of length 8}
{norm: MyVector of length 8}
{operator /, MyVector of length 8}
{Copy construction of MyVector(length 8)}
{operator *=, MyVector of length 8}
{operator +, MyVector of length 8}
{operator +=, MyVector of length 8}
{Move construction of MyVector(length 8)}
{Destructor for MyVector(length = 0)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 0)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 8)}
{Destructor for MyVector(length = 8)}
```

Several temporary objects are created and destroyed and quite a few *copy operations* take place. The situation would be worse unless move semantics was available; if we had not supplied a move constructor, a few more copy operations would have been triggered. Even worse, the frequent copying of data runs a high risk of cache misses. This is certainly not an efficient way to do elementary vector operations though it looks elegant at first glance.

Example 0.2.41 (Gram-Schmidt orthonormalization based on MyVector implementation)

Gram-Schmidt orthonormalization has been taught in linear algebra and its theory will be revisited in § 1.5.1. Here we use this simple algorithm from linear algebra to demonstrate the use of the vector class **MyVector** defined in Code 0.2.13.

The templated function `gramschmidt` takes a sequence of vectors stored in a **std::vector** object. The actual vector type is passed as a template parameter. It has to supply `length` and `norm` member functions as well as in place arithmetic operations `-=`, `/` and `=`. Note the use of the highlighted methods of the **std::vector** class.

C++11 code 0.2.42: templated function for Gram-Schmidt orthonormalization

```
1 template <typename Vec>
```

```
2  std::vector<Vec> gramschmidt(const std::vector<Vec> &A,double
       eps=1E-14) {
3    const int k = A.size();   // no. of vectors to orthogonalize
4    const int n = A[0].length(); // length of vectors
5    cout << "gramschmidt orthogonalization for " << k << ' ' << n <<
       "-vectors" << endl;
6    std::vector<Vec> Q({A[0]/A[0].norm()}); // output vectors
7    for(int j=1;(j<k) && (j<n);++j) {
8      Q.push_back(A[j]);
9      for(int l=0;l<j;++l) Q.back() -= (A[j]*Q[l])*Q[l];
10     if (Q.back().norm() < eps*A[j].norm()) { // premature termination ?
11       Q.pop_back(); break;
12     }
13     Q.back() /= Q.back().norm(); // normalization
14   }
15   return (Q); // return at end of local scope
16 }
```

This driver program calls a function that initializes a sequence of vectors and then orthonormalizes them by means of the Gram-Schmidt algorithm. Eventually orthonormality of the computed vectors is tested. Please pay attention to

- the use of auto to avoid cumbersome type declarations,
- the for loops following the "foreach" syntax.
- automatic indirect template type deduction for the templated function `gramschmidt` from its argument. In Line 6 the function gramschmidt<MyVector> is instantiated.

C++11 code 0.2.43: Driver code for Gram-Schmidt orthonormalization

```
1  int main() {
2    myvec::MyVector::dbg = false;
3    const int n = 7; const int k = 7;
4    auto A(initvectors(n,k,[] (int i,int j)
5           { return std::min(i+1,j+1); }));
6    auto Q(gramschmidt(A)); // instantiate template for MyVector
7    cout << "Set of vectors to be orthonormalized:" << endl;
8    for (const auto &a: A) { cout << a << endl; }
9    cout << "Output of Gram-Schmidt orthonormalization: " << endl;
10   for (const auto &q: Q) { cout << q << endl; }
11   cout << "Testing orthogonality:" << endl;
12   for (const auto &qi: Q) {
13     for (const auto &qj: Q)
14       cout << std::setprecision(3) << std::setw(9) << qi*qj << ' ';
15     cout << endl; }
16   return(0);
17 }
```

This initialization function takes a functor argument as discussed in Section 0.2.3.

C++11 code 0.2.44: Initialization of a set of vectors through a functor with two arguments

```
template<typename Functor>
  std::vector<myvec::MyVector>
    initvectors(std::size_t n,std::size_t k,Functor &&f) {
  std::vector<MyVector> A{};
  for(int j=0;j<k;++j) {
    A.push_back(MyVector(n));
    for(int i=0;i<n;++i)
      (A.back())[i] = f(i,j);
    }
  return(A);
}
```

0.3 Creating Plots with MATHGL

0.3.1 MATHGL Documentation (by J. Gacon)

MATHGL is a huge open-source plotting library for scientific graphics. It can be used for many programming languages, in particular also for C++. Mainly we will use the **Figure** library (implemented by J. Gacon, student of CSE@ETH) introduced below (Section 0.3.4).

However for some special plots using MATHGL can be necessary. A full documentation can be found at http://mathgl.sourceforge.net/doc_en/index.html.

First of all note that all MATHGL plot commands do not take `std::vector`s or EIGEN vectors as arguments but only `mglData`.
NOTE: The Figure environment takes care of all this conversion and formatting!

From `std::vector<double>` or `Eigen::VectorXd` a `mglData` can be initialized as follows:

C++ code 0.3.1: Vector to `mglData`

```
  std::vector<double> v;
  Eigen::VectorXd w;
  // ...  initialize data...

  // the constructor takes a pointer to the first element
  // and the size of the vector
  mglData vd(v.data(), v.size()),
          wd(w.data(), w.size());
```

For `Eigen::RowVectorXd` we first must do some rearrangements of the vector, as the `data()` method returns a pointer to the column major data.

C++ code 0.3.2: `Eigen::RowVectorXd` to `mglData`

```
Eigen::RowVectorXd v;
// ... initialize data...

// Store data in a temporary array
std::vector<double> tmp(v.cols());
for (long i = 0; i < r.cols(); ++i) tmp[i] = v(i);
mglData vd(tmp.data(), tmp.size());
```

For matrices the constructor is slightly different:

C++ code 0.3.3: Matrices to `mglData`

```
Eigen::MatrixXd A;
// ... initialize data ...

// The column major data layout fits MATHGL conventions
mglData Ad(A.rows(), A.cols(), A.data());
```

0.3.2 MATHGL Installation

If you're using Linux the easiest way to install MATHGL is via the command line: `apt-get install mathgl` (Ubuntu/Debian) or `dnf install mathgl` (Fedora).

You can also use CMake to install it.

1. Install CMake (`apt-get install cmake` or from https://cmake.org/download/)
2. Download the MATHGL *source* from
   http://mathgl.sourceforge.net/doc_en/Download.html#Download
3. In the `mathgl-2.3.*` directory do:
   `mkdir build && cd build`
   `cmake ..`
   `make` (this step might take a while)
   `(sudo) make install`

0.3.3 Corresponding Plotting functions of MATLAB and MATHGL

MATHGL offers a subset of MATLAB's large array of plotting functions. The following tables list corresponding commands/methods:

Plotting in 1-D

MATLAB	MATHGL

`axis([0,5,-2,2])`
Default: autofit (`axis auto`)
	`gr.Ranges(0,5,-2,2)`
	Default: `x=-1:1, y=-1:1`
	Workaround:
	`gr.Ranges(x.Minimal(), x.Maximal(),`
	`y.Minimal(), y.Maximal())`

`axis([0,5,-inf,inf])`	`gr.Range('x',0,5)`
`axis([-inf,inf,-2,2])`	`gr.Range('y',-2,2)`
`xlabel('x-axis')`	`gr.Label('x', "x-axis")`
`ylabel('y-axis')`	`gr.Label('y', "y-axis")`

`legend('sin(x)', 'x^2')`	`gr.AddLegend("sin(x)","b")`
	`gr.AddLegend("\\x^2", "g")`
	`gr.Legend()`

`legend('exp(x)')` `legend('boxoff')`	`gr.AddLegend("exp(x)","b")` `gr.Legend(1,1,"")`
`legend('x','Location',` `'northwest')`	`gr.AddLegend("x","b")` `gr.Legend(0,1)`
`legend('cos(x)',` `'Orientation','horizontal')`	`gr.AddLegend("cos(x)","b")` `gr.Legend("#-")`
$(0,1)$ $(0.5,1)$ $(1,1)$ $(0,0.5)$ $(0.5,0.5)$ $(1,0.5)$ $(0,0)$ $(0.5,0)$ $(1,0)$	Legend alignment in MathGL: Values larger than 1 will give position outside of the graph. Default is (1,1).
`plot(y)`	`gr.Plot(y)`
`plot(t,y)`	`gr.Plot(t,y)`

`plot(t0,y0,t1,y1)`
 	
`plot(t,y,'b+')`
	`gr.Plot(t0,y0)`
	`gr.Plot(t1,y1)`
	`gr.Plot(t,y,"b+")`

`print('myfig','-depsc')`
`print('myfig','-dpng')`
	`gr.WriteEPS("myfig.eps")`
	`gr.WritePNG("myfig.png")`
	(compile w/ flag `-lpng`)

`title('Plot title')`	`gr.Title("Plot title")` (title high above plot) `gr.Subplot(1,1,0,"<_")` `gr.Title("Plot title")` } (title directly above plot)

Plotting in 2-D

MATLAB	MATHGL
`colorbar`	`gr.Colorbar()`
`mesh(Z)`	`gr.Mesh(Z)`
`mesh(X,Y,Z)`	`gr.Mesh(X,Y,Z)`
`surface(Z)`	`gr.Surf(Z)`
`surface(X,Y,Z)`	`gr.Surf(X,Y,Z)`
`pcolor(Z)`	`gr.Tile(Z)`
`pcolor(X,Y,Z)`	`gr.Tile(X,Y,Z)`
`plot3(X,Y,Z)`	`gr.Plot(X,Y,Z)`

Additionaly, you have to add `gr.Rotate(50,60)` before the plot command for MathGL to create a 3-D box, otherwise the result is 2-D.

0.3.4 The Figure Class

The Figure library is an interface to MATHGL. By taking care of formatting and the layout it allows a very simple, fast and easy use of the powerful plot library.

This library depends on MATHGL (and optionally on EIGEN), so the installation requires a working version of these dependencies.

0.3.4.1 Introductory example

This short example code will show, how the Figure class can be used.

C++11 code 0.3.4: [
A first code using MATHGL]

```
#include <vector>
#include <Eigen/Dense>
#include <figure/figure.hpp>

int main() {
  std::vector<double> x(10), y(10);
  for (int i = 0; i < 10; ++i){
    x[i] = i; y[i] = std::exp(-0.2*i)*std::cos(i);
  }
  Eigen::VectorXd u = Eigen::VectorXd::LinSpaced(500, 0, 9),
                  v = ( u.array().cos()*(-0.2*u).array().exp()
                     ).matrix();
  mgl::Figure fig;
  fig.plot(x, y, " +r").label("Sample Data");
  fig.plot(u, v, "b").label("Function");
  fig.legend();
  fig.save("plot.eps");
  return 0;
}
```

The figure beside displays the graphical output stored in EPS format in the file `plot.eps`. ▷

Note the use of the methods `label` and `legend` to create the legend.

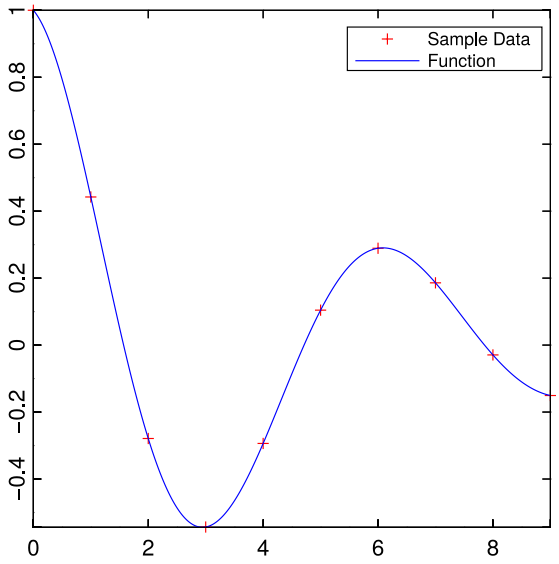


Fig. 3

0.3.4.2 `Figure` Methods

(0.3.5) The `grid` method

This method prints gridlines in the background of a plot.

Definition:

```
void grid( const bool& on = true,
           const std::string& gridType = "-",
           const std::string& gridCol = "h" )
```

Restrictions: None.

Examples:

```
1  mgl::Figure fig;
2  fig.plot(x, y);
3  fig.grid(); // set grid
4  fig.save("plot.eps");
5
6  mgl::Figure fig;
7  fig.plot(x, y);
8  fig.grid(true, "!", "h"); // grey (-> h) fine (-> !)  mesh
9  fig.save("plot.eps");
```

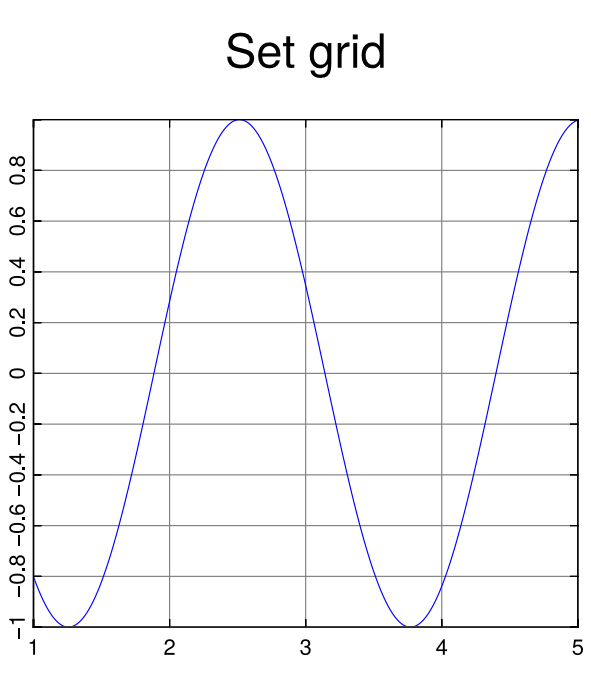


Fig. 4

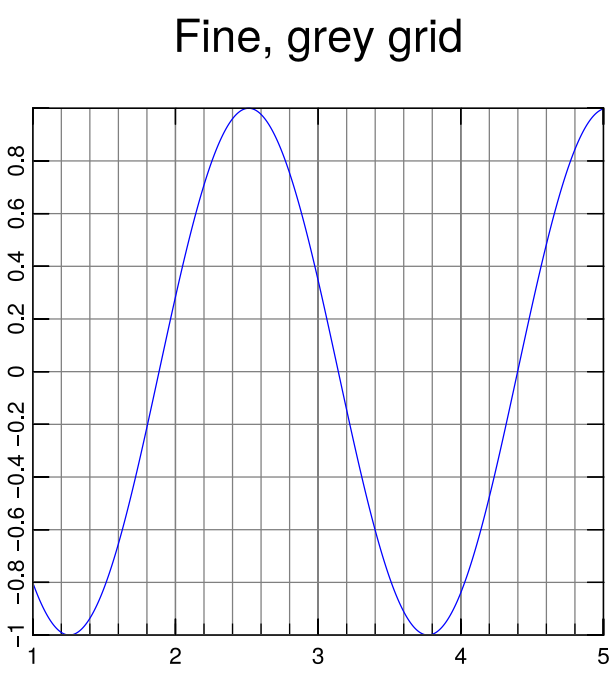


Fig. 5

(0.3.6) The `xlabel` method

This method adds a label text to the x-axis.

Definition:

```
void xlabel( const std::string& label,
             const double& pos = 0 )
```

Restrictions: None.

Examples:

```
1  mgl::Figure fig;
2  fig.plot(x, y, "g+"); // 'g+' equals matlab '+-g'
3  fig.xlabel("Linear x axis");
4  fig.save("plot.eps");
5
6  mgl::Figure fig;
7  fig.xlabel("Logarithmix x axis"); // no restricitons on call order
8  fig.setlog(true, true);
9  fig.plot(x, y, "g+");
10 fig.save("plot.eps");
```

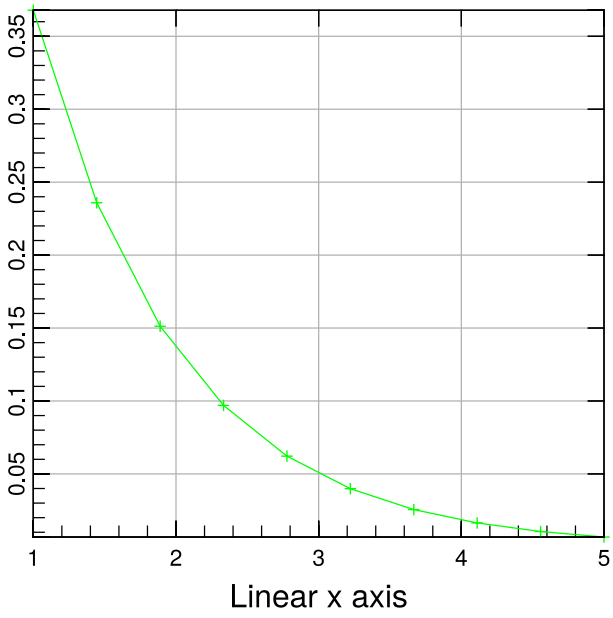


Fig. 6

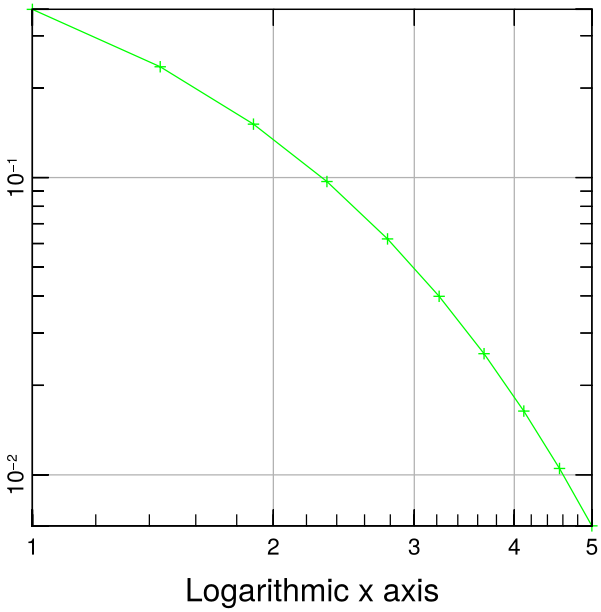


Fig. 7

(0.3.7) The `ylabel` method

This command adds text to the y-axis.

Definition:

```
void ylabel( const std::string& label,
             const double& pos = 0 )
```

Restrictions: None.

Examples: See `xlabel`.

(0.3.8) The `legend` method

This method adds a legend to a plot. Legend entries have to be defined by the `label` method given after the `plot` command.

Definition:

```
void legend( const double& xPos = 1,
             const double& yPos = 1 )
```

Restrictions: None.

Examples:

```
1 mgl::Figure fig;
2 fig.plot(x0, y0).label("My Function");
3 fig.legend(); // 'activate' legend
4 fig.save("plot");
5
```

```
6   mgl::Figure fig;
7   fig.plot(x0, y0).label("My Function");
8   fig.legend(0.5, 0.25); // set position to (0.5, 0.25)
9   fig.save("plot");
10
11  mgl::Figure fig;
12  fig.plot(x0, y0).label("My Function");
13  fig.save("plot"); // legend won't appear as legend() hasn't been called
```

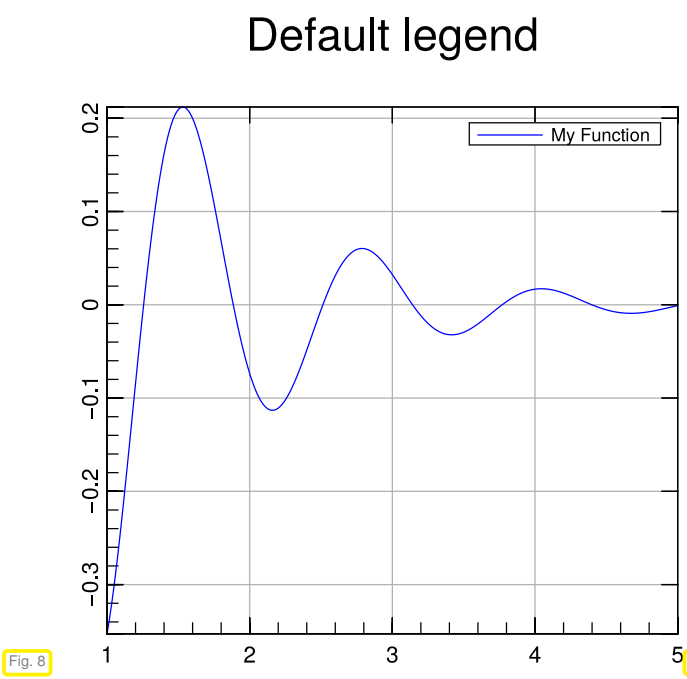


Fig. 8

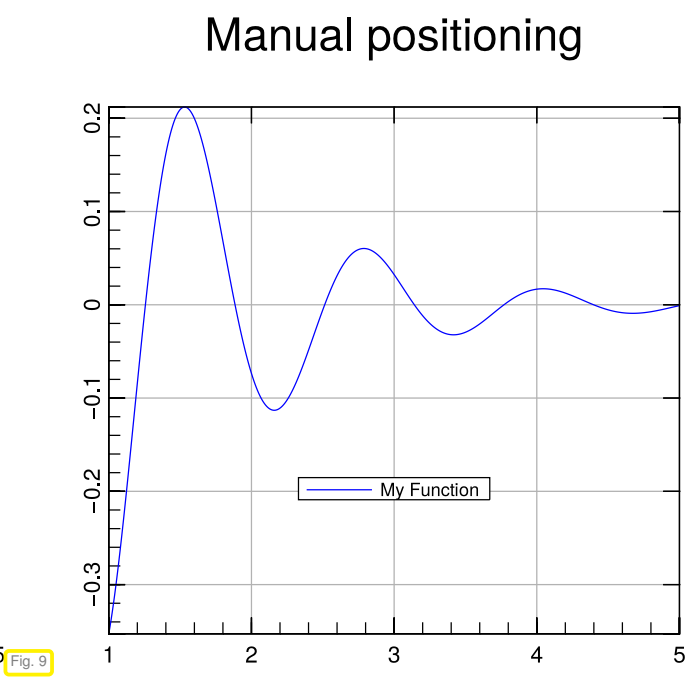


Fig. 9

(0.3.9) The `setlog` method

Selecting axis scaling for a plot, either linear or logarithmic.

Definition:

```
void setlog( const bool& logx = false,
             const bool& logy = false,
             const bool& logz = false )
```

Restrictions: All plots will use the latest `setlog` options or default if none have been set.

Examples:

```
1   mgl::Figure fig;
2   fig.setlog(true, false); // -> semilogx
3   fig.plot(x0, y0);
4   fig.setlog(false, true); // -> semilogy
5   fig.plot(x1, y1);
6   fig.setlog(true, true); // -> loglog
7   fig.plot(x2, y2);
8   fig.save("plot.eps"); // ATTENTION: all plots will use loglog-scale
9
10  mgl::Figure fig;
```

```
11  fig.plot(x, y);
12  fig.save("plot.eps"); // -> default (= linear) scaling
```

Multiple setlog calls

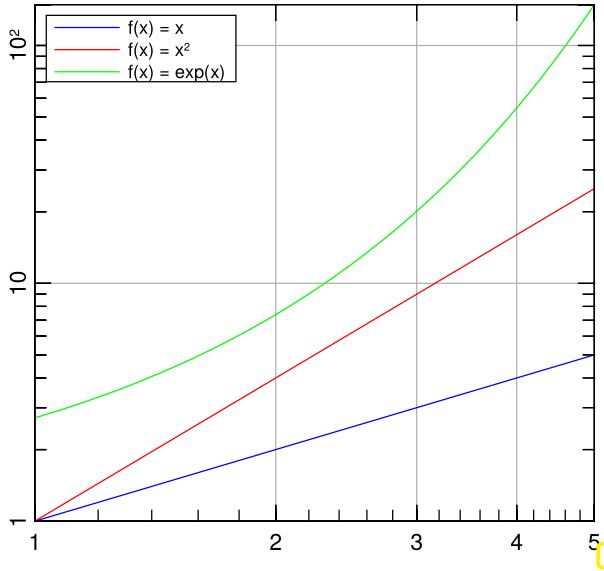


Fig. 10

Default scaling

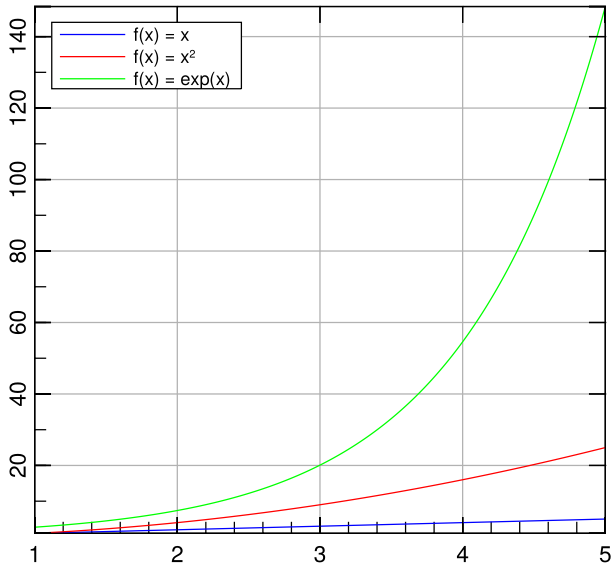


Fig. 11

(0.3.10) The `plot` method

The standard method for plotting function graphs.

Definition:

```
template <typename yVector>
void plot( const yVector& y,
           const std::string& style = "" )

template <typename xVector, typename yVector>
void plot( const xVector& x,
           const yVector& y,
           const std::string& style = "" )
```

Restrictions: `xVector` and `yVector` must have a `size()` method, which returns the size of the vector and a `data()` method, which returns a pointer to the first element in the vector.
Furthermore `x` and `y` must have same length.

Examples:

```
1  mgl::Figure fig;
2  fig.plot(x, y, "g;"); // green and dashed linestyle
3  fig.save("data.eps");
4
5  mgl::Figure fig;
6  fig.plot(x, y); // OK - style is optional
7  fig.save("data.eps");
8
9  mgl::Figure fig;
```

```
10  fig.plot(x, y, " *r", "Data w/ red dots"); // ' *r' equals matlab 'r*'
11  fig.save("data.eps");
```

Plot w/ dashed style

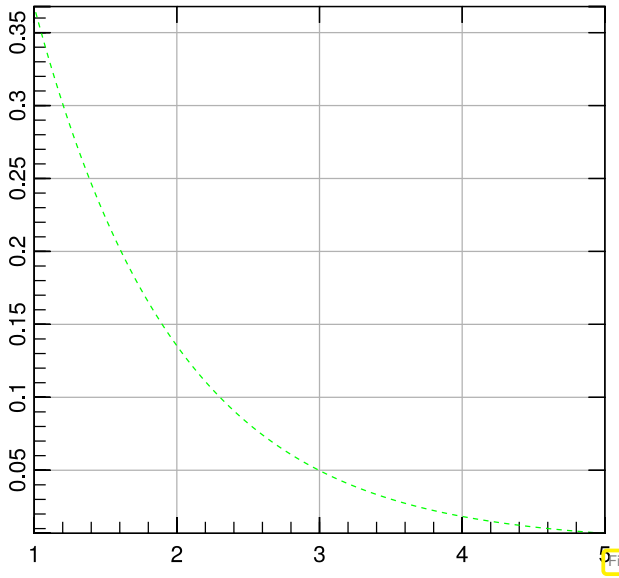


Fig. 12

Default style

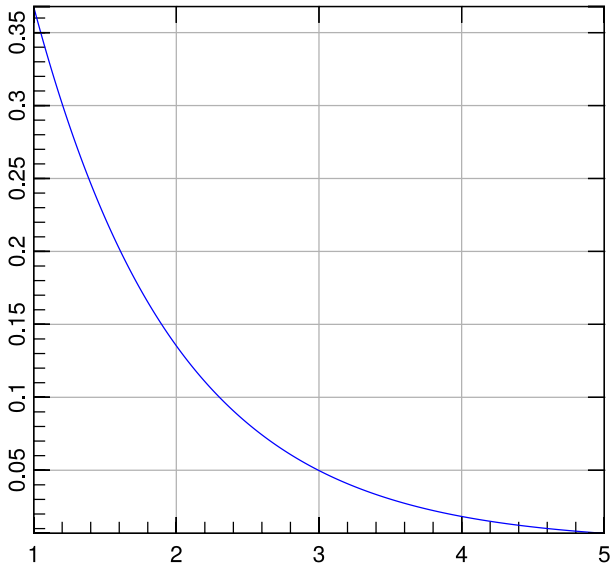


Fig. 13

Data plotting

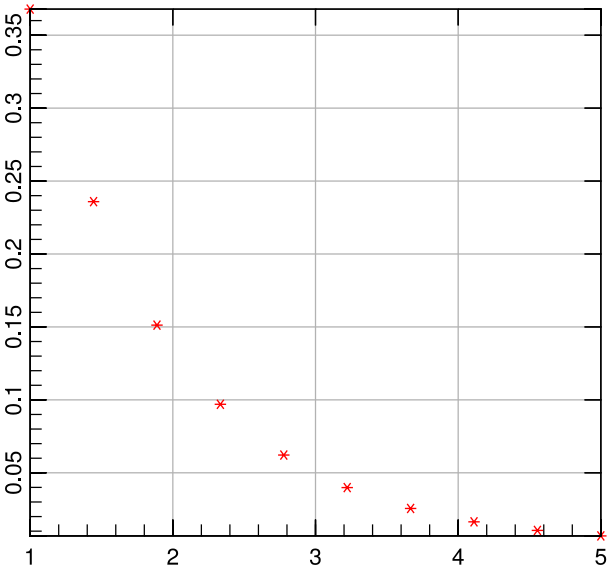


Fig. 14

See below for the various colors and linestyles that can be chosen for plots.

(0.3.11) The `plot3` method

This method can visualize curves in 3D space.

Definition:

```
template <typename xVector, typename yVector, typename zVector>
void plot3( const xVector& x,
            const yVector& y,
            const zVector& z,
            const std::string& style = "" )
```

Restrictions: Same restrictions as in `plot` for two vectors, extended to `zVector`.

Examples:

```
mgl::Figure fig;
fig.plot3(x, y, z);
fig.save("trajectories.eps");
```

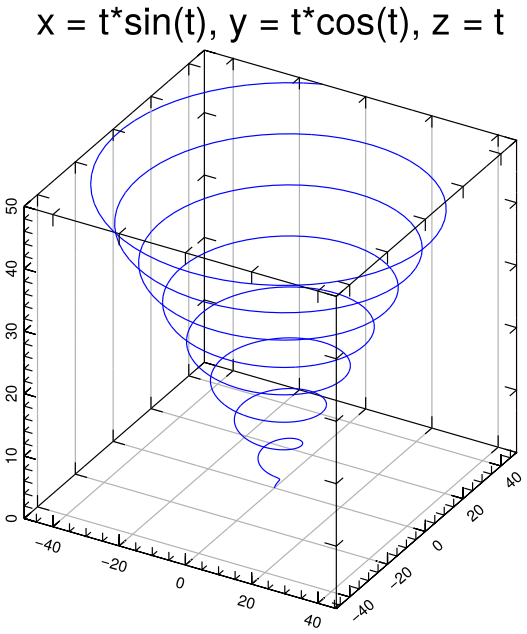


Fig. 15

This is the default color and line style.

As in the case of the `plot` method different colors and linestyles can be selected through the optional string argument `style`, see § 0.3.18.

(0.3.12) The `fplot` method

This command can be used to plot a function of x passed a expression in a string.

Definition:

```
void fplot( const std::string& function,
            const std::string& style = "" )
```

Restrictions: None.

Examples:

```
mgl::Figure fig;
fig.fplot("(3*x^2 - 4.5/x)*exp(-x/1.3)");
fig.fplot("5*sin(5*x)*exp(-x)", "r").label("5sin(5x)*e^{-x}");
fig.ranges(0.5, 5, -5, 5); // be sure to set ranges for fplot!
fig.save("plot.eps");

mgl::Figure fig;
fig.plot(x, y, "b").label("Benchmark");
fig.fplot("x^2", "k;").label("O(x^2)");
// here we don't set the ranges as it uses the range given by the
// x,y data and we use fplot to draw a reference line O(x^2)
fig.save("runtimes.eps");
```

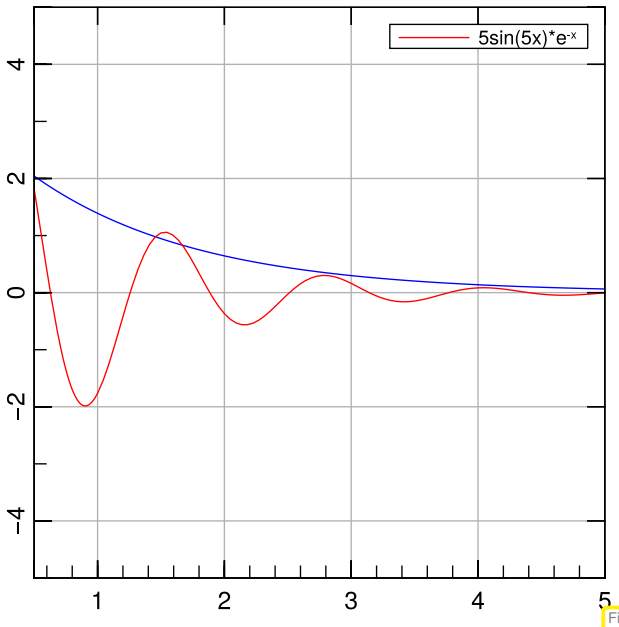


Fig. 16

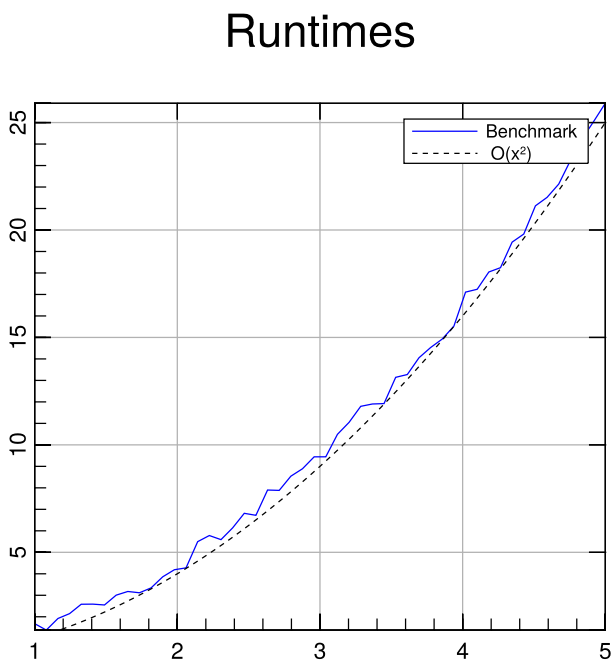


Fig. 17

(0.3.13) The `ranges` method

The method sets the axis ranges for a 2D plot.

Definition:

```
void ranges( const double& xMin,
             const double& xMax,
             const double& yMin,
             const double& yMax )
```

Restrictions: xMin $<$ xMax, yMin $<$ yMax and ranges must be $>$ 0 for axis in logarithmic scale.

Examples:

```
1  mgl::Figure fig;
2  fig.ranges(-1,1,-1,1);
3  fig.plot(x, y, "b");
4
5  mgl::Figure fig;
6  fig.plot(x, y, "b");
7  fig.ranges(0, 2.3, 4, 5); // ranges can be called before or after 'plot'
8
9  mgl::Figure fig;
10 fig.ranges(-1, 1, 0, 5);
11 fig.setlog(true, true); // will run but MathGL will throw a warning
12 fig.plot(x, y, "b");
```

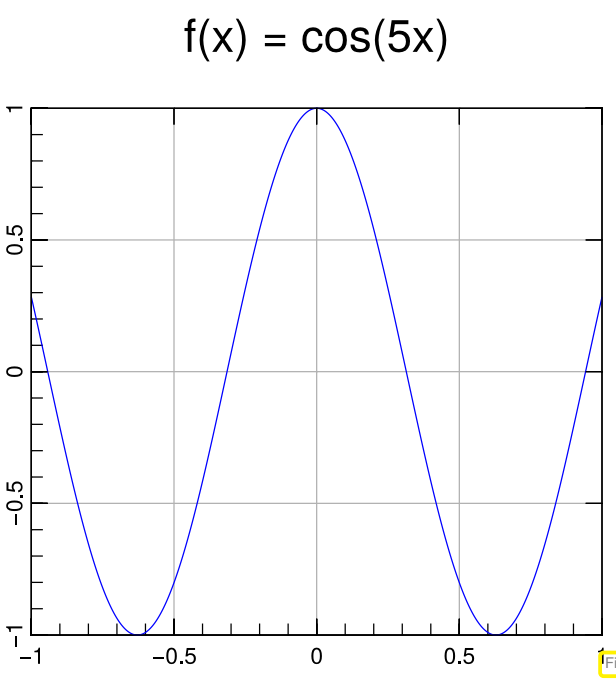


Fig. 18

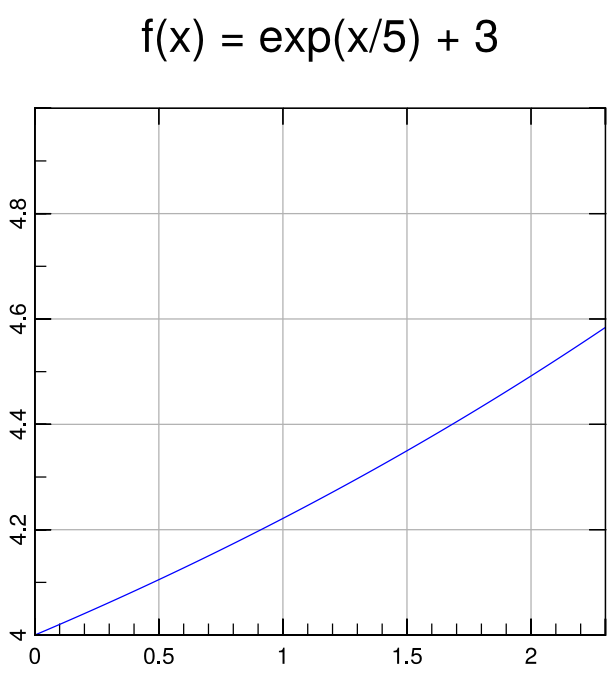


Fig. 19

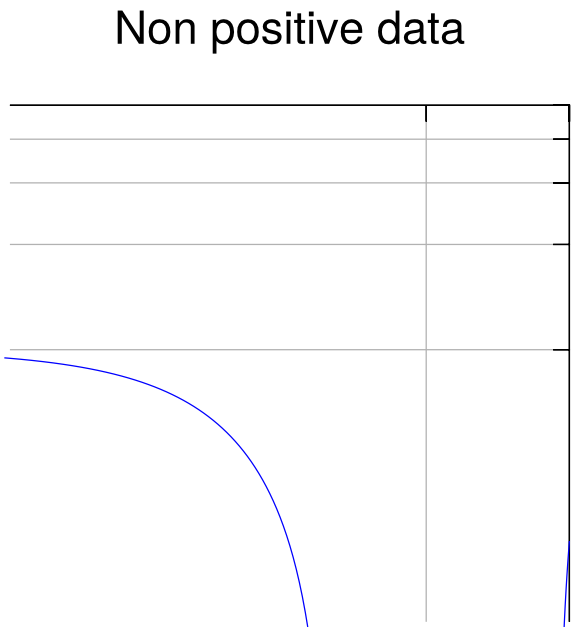


Fig. 20

Illegal ranges for logarithmic scale yield a mangled output.

(0.3.14) The `save` method

Saves the graphics currently stored in a figure object to file. The default format is EPS.

Definition:

```
void save( const std::string& file )
```

Restrictions: Supported file formats: `.eps` and `.png`.

Examples:

```
1 mgl::Figure fig;
2 fig.save("plot.eps"); // OK
3
```

```
4  mgl::Figure fig;
5  fig.save("plot"); // OK - will be saved as plot.eps
6
7  mgl::Figure fig;
8  fig.save("plot.png"); // OK - but needs -lpng flag!
```

(0.3.15) The `spy` method

This method renders the sparsity pattern of a matrix.

Definition:

```
template <typename Matrix>
MglPlot& spy(const Matrix& A, const std::string& style = "b");

template <typename Scalar>
MglPlot& spy(const Eigen::SparseMatrix<Scalar>& A, const
   std::string& style = "b");
```

Restrictions: None.
Examples:

```
1  mgl::Figure fig;
2  fig.spy(A); // e.g:  A = Eigen::MatrixXd, or Eigen::SparseMatrix<double>
```

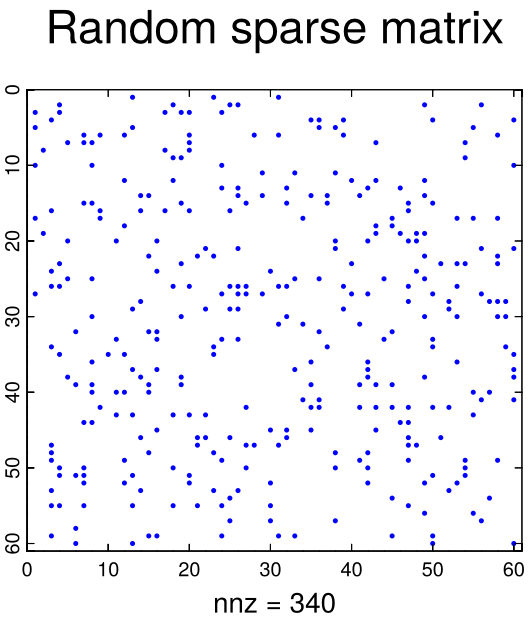


Fig. 21

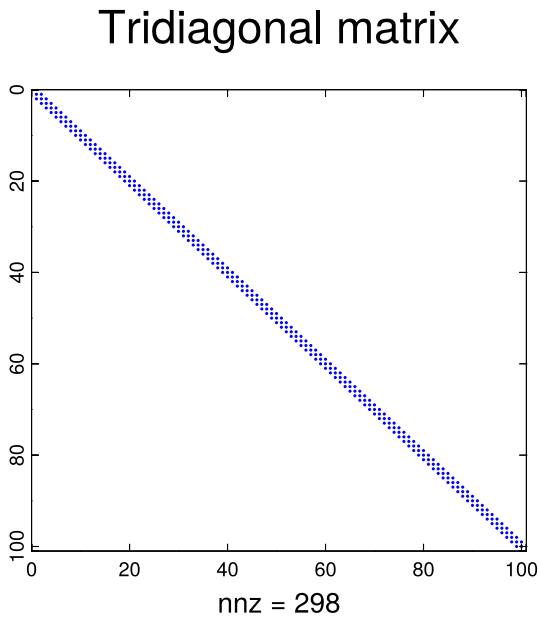


Fig. 22

(0.3.16) The `triplot` method

This method allows to plot triangulations.

Definition:

```
template <typename Scalar, typename xVector, typename yVector>
MglPlot& triplot(const Eigen::Matrix<Scalar, -1, -1, Eigen::RowMajor>& T,
   const xVector& x, const yVector& y, std::string style = "");

template <typename Scalar, typename xVector, typename yVector>
MglPlot& triplot(const Eigen::Matrix<Scalar, -1, -1, Eigen::ColMajor>& T,
   const xVector& x, const yVector& y, std::string style = "");
```

Restrictions: The vectors `x` and `y` must have the same dimensions and the matrix `T` must be of dimension $N \times 3$, with N being the number of edges.

Examples:

```
1  mgl::Figure fig;
2  fig.triplot(T, x, y, "b?"); // '?'  enumerates all vertices of the mesh
3
4  mgl::Figure fig;
5  fig.triplot(T, x, y, "bF"); // 'F' will yield a solid background
6  fig.triplot(T, x, y, "k"); // draw black mesh on top
```

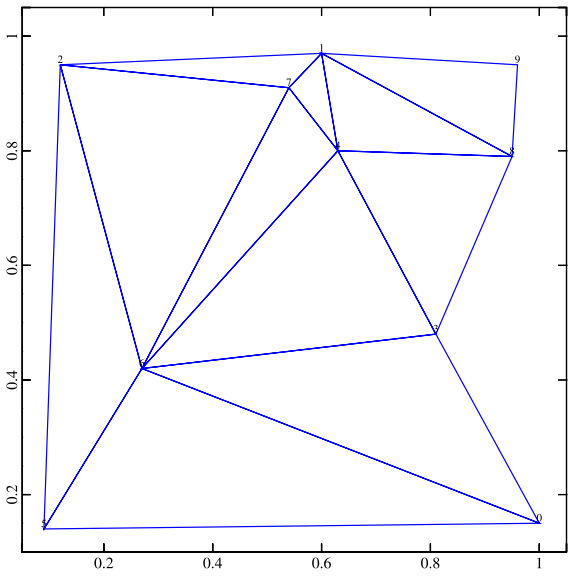


Fig. 23

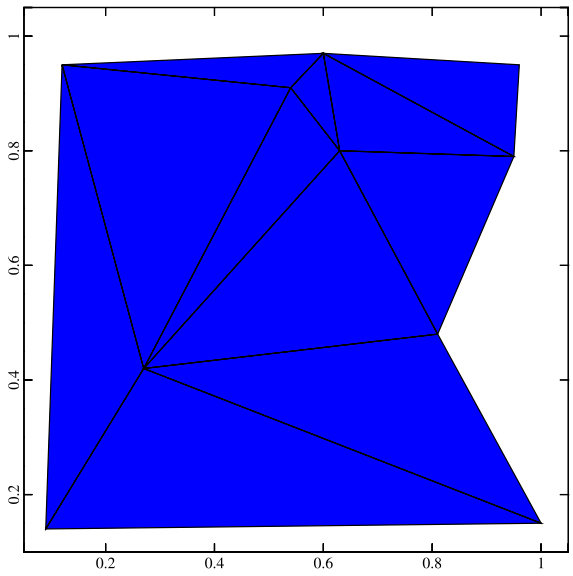


Fig. 24

(0.3.17) The `title` method

Add a title line to any plot.

Definition:

```
void title( const std::string& text )
```

Restrictions: None.

(0.3.18) The string style specifiers

Linecolors[a]:

blue	b
green	g
red	r
cyan	c
magenta	m
yellow	y
gray	h
green-blue	l
sky-blue	n
orange	q
green-yellow	e
blue-violet	u
purple	p

[a] Upper-case letters will give a darker version of the lower-case version.

Linestyles:

none	
solid	-
dashed	;
small dashed	=
long dashed	|
dotted	:
dash-dotted	j
small dash-dotted	i

None is used as follows: " r*" gives red stars w/o any lines

Linemarkers:

+	+
o	o
◇	d
·	.
△	^
▽	v
◁	<
▷	>
⊙	#.
⊞	#+
⊠	#x

Chapter 1

Computing with Matrices and Vectors

(1.0.1) Prerequisite knowledge for Chapter 1

This chapter heavily relies on concepts and techniques from linear algebra as taught in the 1st semester introductory course. Knowledge of the following topics from linear algebra will be taken for granted and they should be refreshed in case of gaps:

- Operations involving matrices and vectors [**?**, Ch. 2]
- Computations with block-structured matrices
- Linear systems of equations: existence and uniqueness of solutions [**?**, Sects. 1.2, 3.3]
- Gaussian elimination [**?**, Ch. 2]
- LU-decomposition and its connection with Gaussian elimination [**?**, Sect. 2.4]

(1.0.2) Levels of operations in simulation codes

The lowest level of real arithmetic available on computers are the elementary operations "$+$", "$-$", "$*$", "$\backslash$", "^", usually implemented in hardware. The next level comprises computations on finite arrays of real numbers, the elementary linear algebra operations (BLAS). On top of them we build complex algorithms involving iterations and approximations.

Complex iterative/recursive/approximative algorithms
Linear algebra operations on arrays (BLAS)
Elementary operations in $\mathbb{R}$

Hardly ever anyone will contemplate implementing elementary operations on binary data formats; similarly, well tested and optimised code libraries should be used for all elementary linear algebra operations in simulation codes. This chapter will introduce you to such libraries and how to use them smartly.

Contents

1.1 Fundamentals

1.1.1 Notations

The notations in this course try to adhere to established conventions. Since these may not be universal, idiosyncrasies cannot be avoided completely. Notations in textbooks may be different, beware!

✎ notation for generic field of numbers: $\mathbb{K}$

In this course, $\mathbb{K}$ will designate either $\mathbb{R}$ (real numbers) or $\mathbb{C}$ (complex numbers); complex arithmetic [**?**, Sect. 2.5] plays a crucial role in many applications, for instance in signal processing.

(1.1.1) Notations for vectors

✦ Vectors = are n-tuples ($n \in \mathbb{N}$) with components $\in \mathbb{K}$.

vector = one-dimensional array (of real/complex numbers)

✎ vectors will usually be denoted by small **bold** symbols: $\mathbf{a}, \mathbf{b}, \ldots, \mathbf{x}, \mathbf{y}, \mathbf{z}$

✦ Default in this lecture: vectors $=$ column vectors

$$\underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{K}^n}_{\text{column vector}} \qquad \underbrace{\begin{bmatrix} x_1 \cdots x_n \end{bmatrix} \in \mathbb{K}^{1,n}}_{\text{row vector}}$$

$\mathbb{K}^n \triangleq$ vector space of *column vectors* with n components in $\mathbb{K}$.

Unless stated otherwise, in mathematical formulas vector components are indexed from 1 !

✎ notation for column vectors: **bold** small roman letters, e.g. $\mathbf{x}, \mathbf{y}, \mathbf{z}$

✦ Transposing: $\begin{cases} \text{column vector} & \mapsto & \text{row vector} \\ \text{row vector} & \mapsto & \text{column vector} \end{cases}$.

$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}^\top = \begin{bmatrix} x_1 \cdots x_n \end{bmatrix} \quad , \quad \begin{bmatrix} x_1 \cdots x_n \end{bmatrix}^\top = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

✎ Notation for row vectors: $\mathbf{x}^\top, \mathbf{y}^\top, \mathbf{z}^\top$

✦ Addressing vector components:

✎ two notations: $\mathbf{x} = [x_1 \dots x_n]^\top \rightarrow x_i, \quad i = 1, \dots, n$
$\mathbf{x} \in \mathbb{K}^n \rightarrow (\mathbf{x})_i, \quad i = 1, \dots, n$

✦ Selecting sub-vectors:

✎ notation: $\mathbf{x} = [x_1 \dots x_n]^\top \succ (\mathbf{x})_{k:l} = (x_k, \dots, x_l)^\top, 1 \le k \le l \le n$

✦ j-th unit vector: $\mathbf{e}_j = [0, \dots, 1, \dots, 0]^\top, \quad (\mathbf{e}_j)_i = \delta_{ij}, i, j = 1, \dots, n.$

✎ notation: Kronecker symbol $\delta_{ij} := 1$, if $i = j$, $\delta_{ij} := 0$, if $i \neq j$.

(1.1.2) Notations and notions for matrices

✦ Matrices = two-dimensional arrays of real/complex numbers

$$\mathbf{A} := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix} \in \mathbb{K}^{n,m} \quad , \quad n, m \in \mathbb{N} \, .$$

vector space of $n \times m$-matrices: ($n \mathrel{\hat{=}}$ number of rows, $m \mathrel{\hat{=}}$ number of columns)

✎ notation: **bold** CAPITAL roman letters, e.g., $\mathbf{A}, \mathbf{S}, \mathbf{Y}$

$\mathbb{K}^{n,1} \leftrightarrow$ column vectors, $\mathbb{K}^{1,n} \leftrightarrow$ row vectors

✦ Writing a matrix as a tuple of its columns or rows

$$\mathbf{c}_i \in \mathbb{K}^n, \quad i = 1, \dots, m \quad \triangleright \quad \mathbf{A} = [\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_m] \in \mathbb{K}^{n,m} \, ,$$

$$\mathbf{r}_i \in \mathbb{K}^m, \quad i = 1, \dots, n \quad \triangleright \quad \mathbf{A} = \begin{bmatrix} \mathbf{r}_1^\top \\ \vdots \\ \mathbf{r}_n^\top \end{bmatrix} \in \mathbb{K}^{n,m} \, .$$

✦ Addressing matrix entries & sub-matrices (✎ notations):

$$\mathbf{A} := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}$$

→ entry $(\mathbf{A})_{i,j} = a_{ij}$, $1 \le i \le n, 1 \le j \le m$,
→ i-th row, $1 \le i \le n$: $a_{i,:} = (\mathbf{A})_{i,:}$,
→ j-th column, $1 \le j \le m$: $a_{:,j} = (\mathbf{A})_{:,j}$,
→ matrix block (sub-matrix) $(a_{ij})_{\substack{i=k,\dots,l \\ j=r,\dots,s}} = (\mathbf{A})_{k:l,r:s}$, $1 \le k \le l \le n$, $1 \le r \le s \le m$.

The colon (:) range notation is inspired by MATLAB's matrix addressing conventions, see Section 1.2.1. $(\mathbf{A})_{k:l,r:s}$ is a matrix of size $(l-k+1)\times(s-r+1)$.

✦ Transposed matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}^\top := \begin{bmatrix} a_{11} & \dots & a_{n1} \\ \vdots & & \vdots \\ a_{1m} & \dots & a_{mn} \end{bmatrix} \in \mathbb{K}^{m,n} .$$

✦ Adjoint matrix (Hermitian transposed):

$$\mathbf{A}^{\mathrm{H}} := \begin{bmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{bmatrix}^{\mathrm{H}} := \begin{bmatrix} \bar{a}_{11} & \dots & \bar{a}_{n1} \\ \vdots & & \vdots \\ \bar{a}_{1m} & \dots & \bar{a}_{mn} \end{bmatrix} \in \mathbb{K}^{m,n} .$$

✎ notation: $\bar{a}_{ij} = \mathfrak{Re}(a_{ij}) - i\mathfrak{Im}(a_{ij})$ complex conjugate of a_{ij}.

1.1.2 Classes of matrices

Most matrices occurring in mathematical modelling have a special structure. This section presents a few of these. More will come up throughout the remainder of this chapter; see also [**?**, Sect. 4.3].

(1.1.3) Special matrices

Terminology and notations for a few very special matrices:

$$\text{Identity matrix:} \quad \mathbf{I} = \begin{bmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{bmatrix} \in \mathbb{K}^{n,n} ,$$

$$\text{Zero matrix:} \quad \mathbf{O} = \begin{bmatrix} 0 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 0 \end{bmatrix} \in \mathbb{K}^{n,m} ,$$

$$\text{Diagonal matrix:} \quad \mathbf{D} = \begin{bmatrix} d_1 & & 0 \\ & \ddots & \\ 0 & & d_n \end{bmatrix} \in \mathbb{K}^{n,n} , \quad d_j \in \mathbb{K} , \quad j = 1,\dots,n .$$

The creation of special matrices can usually be done by special commands or functions in the various languages or libraries dedicated to numerical linear algebra, see § 1.2.5, § 1.2.13.

(1.1.4) Diagonal and triangular matrices

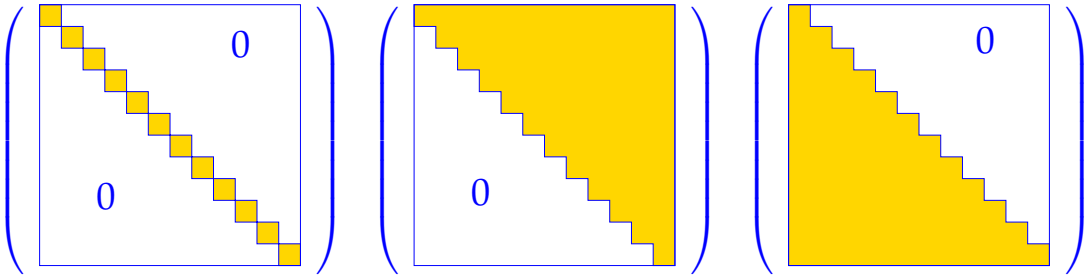
A little terminology to quickly refer to matrices whose non-zero entries occupy special locations:

Definition 1.1.5. Types of matrices

A matrix $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{m,n}$ is

- diagonal matrix, if $a_{ij} = 0$ for $i \neq j$,
- upper triangular matrix if $a_{ij} = 0$ for $i > j$,
- lower triangular matrix if $a_{ij} = 0$ for $i < j$.

A triangular matrix is normalized, if $a_{ii} = 1$, $i = 1, \ldots, \min\{m, n\}$.



diagonal matrix upper triangular lower triangular

(1.1.6) Symmetric matrices

Definition 1.1.7. Hermitian/symmetric matrices

A matrix $\mathbf{M} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is Hermitian, if $\mathbf{M}^{\mathrm{H}} = \mathbf{M}$. If $\mathbb{K} = \mathbb{R}$, the matrix is called symmetric.

Definition 1.1.8. Symmetric positive definite (s.p.d.) matrices → [?, Def. 3.31], [?, Def. 1.22]

$\mathbf{M} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is symmetric (Hermitian) positive definite (s.p.d.), if

$$\mathbf{M} = \mathbf{M}^{\mathrm{H}} \quad \text{and} \quad \forall \mathbf{x} \in \mathbb{K}^n: \ \mathbf{x}^{\mathrm{H}}\mathbf{M}\mathbf{x} > 0 \quad \Leftrightarrow \quad \mathbf{x} \neq 0 \, .$$

If $\mathbf{x}^{\mathrm{H}}\mathbf{M}\mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{K}^n$ ▷ $\mathbf{M}$ positive semi-definite.

Lemma 1.1.9. Necessary conditions for s.p.d. → [?, Satz 3.33], [?, Prop. 1.18]

For a symmetric/Hermitian positive definite matrix $\mathbf{M} = \mathbf{M}^{\mathrm{H}} \in \mathbb{K}^{n,n}$ holds true:

1. $m_{ii} > 0, i = 1, \ldots, n$,
2. $m_{ii}m_{jj} - |m_{ij}|^2 > 0 \quad \forall 1 \leq i < j \leq n$,
3. *all eigenvalues of $\mathbf{M}$ are positive.* *(← also sufficient for symmetric/Hermitian $\mathbf{M}$)*

Remark 1.1.10 (S.p.d. Hessians)

Recall from analysis: in an isolated local minimum x^* of a C^2-function $f : \mathbb{R}^n \mapsto \mathbb{R}$ ➤ Hessian $\mathrm{D}^2 f(x^*)$ s.p.d. (see Def. 8.4.11 for the definition of the Hessian)

To compute the minimum of a C^2-function iteratively by means of Newton's method (→ Sect. 8.4) a linear

system of equations with the s.p.d. Hessian as system matrix has to be solved in each step.

The solutions of many equations in science and engineering boils down to finding the minimum of some (energy, entropy, etc.) function, which accounts for the prominent role of s.p.d. linear systems in applications.

?! *Review question(s) 1.1.11.* (Special matrices)

- We consider two matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,m}$, both with at most $N \in \mathbb{N}$ non-zero entries. What is the maximal number of non-zero entries of $\mathbf{A} + \mathbf{B}$?
- A matrix $\mathbf{A} \in \mathbb{R}^{n,m}$ enjoys the following property (banded matrix):

$$i \in \{1,\ldots,n\},\ j \in \{1,\ldots,m\},\ i-j \notin \{-B_-,\ldots,B_+\} \quad \Rightarrow \quad (\mathbf{A})_{ij} = 0\,,$$

  for given $B_-, B_+ \in \mathbb{N}_0$. What is the maximal number of non-zero entries of $\mathbf{A}$.

1.2 Software and Libraries

Whenever algorithms involve matrices and vectors (in the sense of linear algebra) it is advisable to rely on suitable code libraries or numerical programming environments.

1.2.1 MATLAB

MATLAB ("matrix laboratory") is a commercial (sold by MathWorks Corp.)

- full fledged high level programming language designed for numerical algoritms (domain specific language: DSL)
- integrated development environment (IDE) offering editor, debugger, profiler, tracing facilities,
- rather comprehensive collection of numerical libraries.

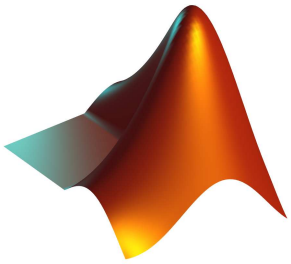


Fig. 25

Many textbooks, for instance [**?**] and [**?**] rely on MATLAB to demonstrate the actual implementation of numerical algorithms. So did earlier versions of this course. The current version has dumped MATLAB and, hence, this section **can be skipped** safely.

In its basic form MATLAB is an interpreted scripting language without strict type-binding. This, together with its uniform IDE across many platforms, makes it a very popular tool for *rapid prototyping* and *testing* in CSE.

Plenty of resources are available for MATLAB's users, among them

- MATLAB documentation accessible through the Help menu ot through this link,
- MATLAB's help facility through the commands **help** <**function**> or `doc` <**function**>,
- A concise MATLAB primer, one of many available online, see also here

- MATLAB-Einführung (in German) by P. Arbenz.

"In MATLAB everything is a matrix"

(Fundamental "data type" in MATLAB = matrix of complex numbers)

➢ In MATLAB vectors are represented as $n \times 1$-matrices (column vectors) or $1 \times n$-matrices (row vectors).

Note: The treatment of vectors as special matrices is consistent with the basic operations from matrix calculus.

(1.2.1) Fetching the dimensions of a matrix

☞ `v = size(A)` yields a row vector `v` of length 2 with `v(1)` containing the number of rows and `v(1)` containing the number of columns of the matrix `A`.

☞ **numel**`(A)` returns the total number of entries of `A`; if `A` is a (row or column) vector, we get the length of A.

(1.2.2) Access to matrix and vector components in MATLAB

Access (rvalue & lvalue) to components of a vector and entries of a matrix in MATLAB is possible through the `()-operator`:

☞ `r = v(i)`: retrieve `i`-th entry of vector `v`. `i` must be an integer and smaller or equal **numel**`(v)`.

☞ `r = A(i,j)`: get matrix entry $(\mathbf{A})_{i,j}$ for two (valid) integer indices `i` and `j`.

☞ `r = A(end-1,end-2)`: get matrix entry $(\mathbf{A})_{n-1,m-2}$ of an $n \times m$-matrix `A`.

! In case the matrix `A` is too small to contain an entry $(\mathbf{A})_{i,j}$, write access to `A(i,j)` will automatically trigger a dynamic adjustment of the matrix size to hold the accessed entry. The other new entries are filled with zeros.

```
% Caution: matrices are dynamically
   expanded when
% out of range entries are accessed
M = [1,2,3;4,5,6]; M(4,6) = 1.0; M,
```

Output: `M=`

1	2	3	0	0	0
4	5	6	0	0	0
0	0	0	0	0	0
0	0	0	0	0	1

➢ Danger of accidental exhaustion of memory!

(1.2.3) Access to submatrices in MATLAB

For any two (row or column) vectors `I`, J of positive integers `A(I,J)` selects the submatrix

$$\left[(\mathbf{A})_{i,j}\right]_{\substack{i\in \mathtt{I}\\ j\in \mathtt{J}}} \in \mathbb{K}^{\sharp \mathtt{I},\sharp \mathtt{J}} .$$

`A(I,J)` can be used as both r-value and l-value; in the former case the maximal components of `I` and `J` have to be smaller or equal the corresponding matrix dimensions, lest MATLAB issue the error message

`Index exceeds matrix dimensions`. In the latter case, the size of the matrix is grown, if needed, see § 1.2.2.

(1.2.4) Initialization of matrices in MATLAB by concatenation

Inside square brackets `[ ]` the following two matrix construction operators can be used:

- `,`-operator $\hat{=}$ adding another matrix to the right (horizontal concatenation)
- `;`-operator $\hat{=}$ adding another matrix at the bottom (vertical concatenation)

(The `,`-operator binds more strongly than the `;`-operator!)

! Matrices joined by the `,`-operator must have the same number of rows.
Matrices concatenated vertically must have the same number of columns

▷ Filling a small matrix:
```
A = [1,2;3,4;5,6];
A = [[1;3;5],[2;4;6]];
```
$\rightarrow 3 \times 2$ matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$.

▷ Initialization of vectors in MATLAB:

column vectors	`x = [1;2;3];`
row vectors	`y = [1,2,3];`

▷ Building a matrix from blocks:

```
% MATLAB script demonstrating the construction
   of a matrix from blocks
A = [1,2;3,4]; B = [5,6;7,8];
C = [A,B;-B,A], % use concatenation
```

Output: `C=`

```
   1   2   5   6
   3   4   7   8
  -5  -6   1   2
  -7  -8   3   4
```

(1.2.5) Special matrices in MATLAB

[Special matrices in MATLAB → § 1.1.3]

- ☞ $n \times n$ identity matrix: `I = eye(n);`
- ☞ $m \times n$ zero matrix: `O = zeros(m,n);`
- ☞ $m \times n$ random matrix with entries equidistributed in $[0,1]$: `R = rand(m,n);`
- ☞ $n \times n$ diagonal matrix with components of n-vector `d` (both row or column vectors are possible) on its diagonal: `D = diag(d);`

(1.2.6) Initialization of equispaced vectors ("loop index vectors")

In MATLAB `v = (a:s:b)`, where `a,b,s` are real numbers, creates a *row vector* as initialised by the following code

```
if ((b >= a) && (s > 0))
  v = [a]; while (v(end)+s <= b), v = [v,v(end)+s]; end
```

```
elseif ((b <= a) && (s < 0))
  v = [a]; while (v(end)+s >= b), v = [v,v(end)+s]; end
else v = [];
end
```

Examples:

```
>> v = (3:-0.5:-0.3)
v =  3.0000 2.5000 2.0000 1.5000 1.0000  0.5000 0
>> v = (1:2.5:-13)
v = Empty matrix: 1-by-0
```

These vectors can be used to program loops in MATLAB

```
for i = (a:s:b)
% Do something with the loop variable i
end
```

In general we could also pass a matrix as "loop index vector". In this case the loop variable with run through the *columns* of the matrix

```
% MATLAB loop over columns of a matrix
M = [1,2,3;4,5,6];
for i = M; i, end
```

Output:

```
i =  1    i =  2    i =  3
     4         5         6
```

(1.2.7) Special structural operations on matrices in MATLAB

- `A'` ≙ Hermitian transpose of a matrix `A`, transposing without complex conjugation done by `transpose(A)`.
- **triu** `(A)` and **tril** `(A)` return the upper and lower triangular parts of a matrix `A` as r-value (copy): If $\mathbf{A} \in \mathbb{K}^{m,n}$

$$(\texttt{triu}(\mathbf{A}))_{i,j} = \begin{cases} (\mathbf{A})_{i,j} & \text{, if } i \leq j \text{ ,} \\ 0 & \text{else.} \end{cases} \quad (\texttt{tril}(\mathbf{A}))_{i,j} = \begin{cases} (\mathbf{A})_{i,j} & \text{, if } i \geq j \text{ ,} \\ 0 & \text{else.} \end{cases}$$

- **diag** `(A)` for a *matrix* $\mathbf{A} \in \mathbb{K}^{m,n}$, $\min\{m,n\} \geq 2$ returns the column vector $[(\mathbf{A})_{i,i}]_{i=1,\min\{m,n\}} \in \mathbb{K}^{\min\{m,n\}}$.

1.2.2 PYTHON

PYTHON is a widely used general-purpose and open source programming language. Together with the packages like NUMPY and MATPLOTLIB it delivers similar functionality like MATLAB for free. For interactive computing IPYTHON can be used. All those packages belong to the SCIPY ecosystem.

PYTHON features a good documentation and several scientific distributions are available (e.g. Anaconda, Enthought) which contain the most important packages. On most Linux-distributions the SCIPY ecosystem is also available in the software repository, as well as many other packages including for example the Spyder IDE delivered with Anaconda.

A good introduction tutorial to numerical PYTHON are the SCIPY-lectures. The full documentation of NUMPY and SCIPY can be found here. For former MATLAB-users there's also a guide. The scripts in this lecture notes follow the official PYTHON style guide.

Note that in PYTHON we have to import the numerical packages explicitly before use. This is normally done at the beginning of the file with lines like `import numpy as np` and `from matplotlib import pyplot as plt`. Those import statements are often skipped in this lecture notes to focus on the actual computations. But you can always assume the import statements as given here, e.g. `np.ravel(A)` is a call to a NUMPY function and `plt.loglog(x, y)` is a call to a MATPLOTLIB pyplot function.

PYTHON is not used in the current version of the lecture. Nevertheless a few PYTHON codes are supplied in order to convey similarities and differences to implementations in MATLAB and C++.

(1.2.8) Matrices and Vectors in PYTHON

The basic numeric data type in PYTHON are NUMPY's n-dimensional arrays. Vectors are normally implemented as 1D arrays and no distinction is made between row and column vectors. Matrices are represented as 2D arrays.

- ☞ `v = np.array([1, 2, 3])` creates a 1D array with the three elements 1, 2 and 3.
- ☞ `A = np.array([[1, 2], [3, 4]]` creates a 2D array.
- ☞ `A.shape` gives the n-dimensional size of an array.
- ☞ `A.size` gives the total number of entries in an array.

Note: There's also a matrix class in NUMPY with different semantics but its use is officially discouraged and it might even be removed in future release.

(1.2.9) Manipulating arrays in PYTHON

There are many possibilities listed in the documentation how to create, index and manipulate arrays.

An important difference to MATLAB is, that all arithmetic operations are normally performed element-wise, e.g. `A * B` is not the matrix-matrix product but element-wise multiplication (in MATLAB: `A.*A`). Also `A * v` does a broadcasted element-wise product. For the matrix product one has to use `np.dot(A, B)` or `A.dot(B)` explicitly.

1.2.3 EIGEN

Currently, the most widely used programming language for the development of new simulation software in scientific and industrial high-performance computing is C++. In this course we are going to use and discuss EIGEN as an example for a C++ library for numerical linear algebra ("embedded" domain specific language: DSL).

EIGEN is a header-only C++ template library designed to enable easy, natural and efficient numerical linear algebra: it provides data structures and a wide range of operations for matrices and vectors, see

below. EIGEN also implements many more fundamental algorithms (see the documentation page or the discussion below).

EIGEN relies on expression templates to allow the efficient evaluation of complex expressions involving matrices and vectors. Refer to the example given in the EIGEN documentation for details.

➥ Link to an "EIGEN Cheat Sheet" (quick reference relating to MATLAB commands)

(1.2.10) Compilation of codes using EIGEN

Compiling and linking on Mac OS X 10.10:

```
clang -D_HAS_CPP0X -std=c++11 -Wall -g \
     -Wno-deprecated-register -DEIGEN3_ACTIVATED \
     -I/opt/local/include -I/usr/local/include/eigen3  \
     -o main.cpp.o -c main.cpp
/usr/bin/c++ -std=c++11 -Wall -g -Wno-deprecated-register \
-DEIGEN3_ACTIVATED -Wl,-search_paths_first \
-Wl,-headerpad_max_install_names main.cpp.o \
-o executable /opt/local/lib/libboost_program_options-mt.dylib
```

Of course, different compilers may be used on different platforms. In all cases, basic EIGEN functionality can be used without linking with a special library. Usually the generation of such elaborate calls of the compiler is left to a build system like CMAKE.

(1.2.11) Matrix and vector data types in EIGEN

A generic matrix data type is given by the templated class

```
Matrix<typename Scalar,
       int RowsAtCompileTime, int ColsAtCompileTime>
```

Here **Scalar** is the underlying scalar type of the matrix entries, which must support the usual operations '+','-','*','/', and '+=', '*=', '¯', etc. Usually the scalar type will be either `double`, `float`, or `complex<>`. The cardinal template arguments `RowsAtCompileTime` and `ColsAtCompileTime` can pass a fixed size of the matrix, if it is known at compile time. There is a specialization selected by the template argument `Eigen::Dynamic` supporting variable size "dynamic" matrices.

C++11-code 1.2.12: Vector type and their use in EIGEN

```
#include <Eigen/Dense>

template<typename Scalar>
void eigenTypeDemo(unsigned int dim)
{
  // General dynamic (variable size) matrices
  using dynMat_t =
     Eigen::Matrix<Scalar,Eigen::Dynamic,Eigen::Dynamic>;
  // Dynamic (variable size) column vectors
  using dynColVec_t = Eigen::Matrix<Scalar,Eigen::Dynamic,1>;
  // Dynamic (variable size) row vectors
```

```
11   using dynRowVec_t = Eigen::Matrix<Scalar,1,Eigen::Dynamic>;
12   using index_t = typename dynMat_t::Index;
13   using entry_t = typename dynMat_t::Scalar;
14
15   // Declare vectors of size 'dim'; not yet initialized
16   dynColVec_t colvec(dim);
17   dynRowVec_t rowvec(dim);
18   // Initialisation through component access
19   for(index_t i=0; i< colvec.size(); ++i) colvec(i) = (Scalar)i;
20   for(index_t i=0; i< rowvec.size(); ++i) rowvec(i) = (Scalar)1/(i+1);
21   colvec[0] = (Scalar)3.14; rowvec[dim-1] = (Scalar)2.718;
22   // Form tensor product, a matrix, see Section 1.3.1
23   dynMat_t vecprod = colvec*rowvec;
24   const int nrows = vecprod.rows();
25   const int ncols = vecprod.cols();
26 }
```

Note that in Line 23 we could have relied on automatic type deduction via **auto** `vectprod = ...`. However, often it is safer to forgo this option and specify the type directly.

The following convenience data types are provided by EIGEN, see documentation:

- **MatrixXd** $\hat{=}$ generic variable size matrix with **d**ouble precision entries
- **VectorXd**, **RowVectorXd** $\hat{=}$ dynamic column and row vectors (**=** dynamic matrices with one dimension equal to 1)
- **MatrixNd** with $N = 2,3,4$ for small fixed size square $N \times N$-matrices (type `double`)
- **VectorNd** with $N = 2,3,4$ for small column vectors with fixed length N.

The `d` in the type name may be replaced with `i` (for `int`), `f` (for `float`), and `cd` (for `complex<double>`) to select another basic scalar type.

All matrix type feature the methods `cols()`, `rows()`, and `size()` telling the number of columns, rows, and total number of entries.

Access to individual matrix entries and vector components, both as Rvalue and Lvalue, is possible through the `()`-operator taking two arguments of type `index_t`. If only one argument is supplied, the matrix is accessed as a linear array according to its memory layout. For vectors, that is, matrices where one dimension is fixed to 1, the `[]`-operator can replace `()` with one argument, see Line 21 of Code 1.2.12.

(1.2.13) Initialization of *dense* matrices in EIGEN

The entry access operator `(int i,int j)` allows the most direct setting of matrix entries; there is hardly any runtime penalty.

Of course, in EIGEN dedicated functions take care of the initialization of the special matrices introduced in **??**:

```
Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n,n);
Eigen::MatrixXd O = Eigen::MatrixXd::Zero(n,m);
```

```
Eigen::MatrixXd D = d_vector.asDiagonal();
```

C++11-code 1.2.14: Initializing special matrices in EIGEN

```
#include <Eigen/Dense>
// Just allocate space for matrix, no initialisation
Eigen::MatrixXd A(rows,cols);
// Zero matrix.  Similar to matlab command zeros(rows, cols);
Eigen::MatrixXd B = MatrixXd::Zero(rows, cols);
// Ones matrix.  Similar to matlab command ones(rows, cols);
Eigen::MatrixXd C = MatrixXd::Ones(rows, cols);
// Matrix with all entries same as value.
Eigen::MatrixXd D = MatrixXd::Constant(rows, cols, value);
// Random matrix, entries uniformly distributed in [0,1]
Eigen::MatrixXd E = MatrixXd::Random(rows, cols);
// (Generalized) identity matrix, 1 on main diagonal
Eigen::MatrixXd I = MatrixXd::Identity(rows,cols);
std::cout << "size of A = (" << A.rows() << ',' << A.cols() << ')' <<
   std::endl;
```

A versatile way to initialize a matrix relies on a combination of the operators « and `,`, which allows the construction of a matrix from blocks:

```
MatrixXd mat3(6,6);
mat3 <<
    MatrixXd::Constant(4,2,1.5), // top row, first block
    MatrixXd::Constant(4,3,3.5), // top row, second block
    MatrixXd::Constant(4,1,7.5), // top row, third block
    MatrixXd::Constant(2,4,2.5), // bottom row, left block
    MatrixXd::Constant(2,2,4.5); // bottom row, right block
```

The matrix is filled top to bottom left to right, block dimensions have to match (like in MATLAB).

(1.2.15) Access to submatrices in EIGEN (→ documentation)

The method `block(int i,int j,int p,int q)` returns a reference to the submatrix with upper left corner at position (i,j) and size $p \times q$.

The methods `row(int i)` and `col(int j)` provide a reference to the corresponding row and column of the matrix. Even more specialised access methods are

`topLeftCorner(p,q)`, `bottomLeftCorner(p,q)`,
`topRightCorner(p,q)`, `bottomRightCorner(p,q)`,
`topRows(q)`, `bottomRows(q)`,
`leftCols(p)`, and `rightCols(q)`,

with obvious purposes.

C++11 code 1.2.16: Demonstration code for access to matrix blocks in EIGEN ➙ GITLAB

```
2  template<typename MatType>
3  void blockAccess(Eigen::MatrixBase<MatType> &M)
4  {
5    using index_t = typename Eigen::MatrixBase<MatType>::Index;
6    using entry_t = typename Eigen::MatrixBase<MatType>::Scalar;
7    const index_t nrows(M.rows()); // No. of rows
8    const index_t ncols(M.cols()); // No. of columns
9
10   cout << "Matrix M = " << endl << M << endl; // Print matrix
11   // Block size half the size of the matrix
12   index_t p = nrows/2,q = ncols/2;
13   // Output submatrix with left upper entry at position (i,i)
14   for(index_t i=0; i < min(p,q); i++)
15     cout << "Block (" << i << ',' << i << ',' << p << ',' << q
16          << ") = " << M.block(i,i,p,q) << endl;
17   // l-value access: modify sub-matrix by adding a constant
18   M.block(1,1,p,q) += Eigen::MatrixBase<MatType>::Constant(p,q,1.0);
19   cout << "M = " << endl << M << endl;
20   // r-value access: extract sub-matrix
21   MatrixXd B = M.block(1,1,p,q);
22   cout << "Isolated modified block = " << endl << B << endl;
23   // Special sub-matrices
24   cout << p << " top rows of m = " << M.topRows(p) << endl;
25   cout << p << " bottom rows of m = " << M.bottomRows(p) << endl;
26   cout << q << " left cols of m = " << M.leftCols(q) << endl;
27   cout << q << " right cols of m = " << M.rightCols(p) << endl;
28   // r-value access to upper triangular part
29   const MatrixXd T = M.template triangularView<Upper>(); //
30   cout << "Upper triangular part = " << endl << T << endl;
31   // l-value access to upper triangular part
32   M.template triangularView<Lower>() *= -1.5; //
33   cout << "Matrix M = " << endl << M << endl;
34 }
```

EIGEN offers views for access to triangular parts of a matrix, see Line 29 and Line 32, according to

`M.triangularView<XX>()`

where `XX` can stand for one of the following: `Upper`, `Lower`, `StrictlyUpper`, `StrictlyLower`, `UnitUpper`, `UnitLower`, see documentation.

For column and row vectors references to sub-vectors can be obtained by the methods `head(int length)`, `tail(int length)`, and `segment(int pos, int length)`.

Note: Unless the preprocessor switch `NDEBUG` is set, EIGEN performs range checks on all indices.

(1.2.17) Componentwise operations in EIGEN

Since operators like MATLAB's `.*` are not available, EIGEN uses the Array concept to furnish entry-wise operations on matrices. An EIGEN-Array contains the same data as a matrix, supports the same methods for initialisation and access, but replaces the operators of matrix arithmetic with entry-wise actions. Matrices and arrays can be converted into each other by the `array()` and `matrix()` methods, see documentation for details.

C++11 code 1.2.18: Using Array in EIGEN ↪ GITLAB

```
2  void matArray(int nrows,int ncols){
3    Eigen::MatrixXd m1(nrows,ncols),m2(nrows,ncols);
4    for(int i = 0; i < m1.rows(); i++)
5      for(int j=0; j < m1.cols(); j++) {
6        m1(i,j) = (double)(i+1)/(j+1);
7        m2(i,j) = (double)(j+1)/(i+1);
8      }
9    // Entry-wise product, not a matrix product
10   Eigen::MatrixXd m3 = (m1.array() * m2.array()).matrix();
11   // Explicit entry-wise operations on matrices are possible
12   Eigen::MatrixXd m4(m1.cwiseProduct(m2));
13   // Entry-wise logarithm
14   cout << "Log(m1) = " << endl << log(m1.array()) << endl;
15   // Entry-wise boolean expression, true cases counted
16   cout << (m1.array() > 3).count() << " entries of m1 > 3" << endl;
17 }
```

The application of a functor (→ Section 0.2.3) to all entries of a matrix can also been done via the `unaryExpr()` method of a matrix:

```
// Apply a lambda function to all entries of a matrix
auto fnct = [](double x) { return (x+1.0/x); };
cout << "f(m1) = " << endl << m1.unaryExpr(fnct) << endl;
```

Remark 1.2.19 (EIGEN in use)

- ☞ EIGEN is used as one of the base libraries for the Robot Operating System (ROS), an open source project with strong ETH participation.
- ☞ The geometry processing library libigl uses EIGEN as its basic linear algebra engine. At ETH it is being used and developed at ETH Zurich, at the Interactive Geometry Lab and Advanced Technologies Lab.

1.2.4 (Dense) Matrix storage formats

All numerical libraries store the entries of a (generic = *dense*) matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ in a linear array of length mn (or longer). Accessing entries entails suitable index computations.

Two natural options for "vectorisation" of a matrix: *row major, column major*

$$\mathbf{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Row major (C-arrays, bitmaps, Python):

A_arr	1	2	3	4	5	6	7	8	9

Column major (Fortran, MATLAB, EIGEN):

A_arr	1	4	7	2	5	8	3	6	9

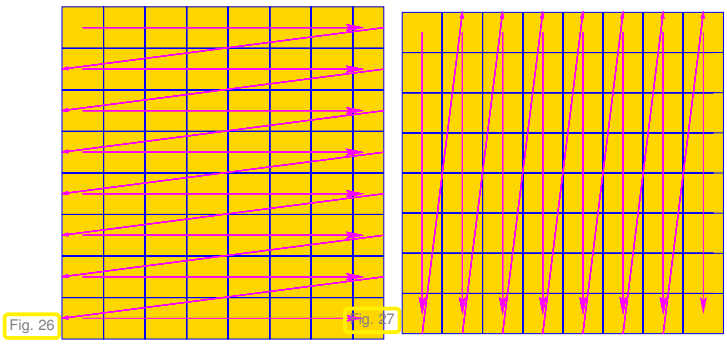
Access to entry $(\mathbf{A})_{ij}$ of $\mathbf{A} \in \mathbb{K}^{n,m}$, $i = 1,\ldots,n$, $j = 1,\ldots,m$:

row major:

$$(\mathbf{A})_{ij} \leftrightarrow \texttt{A\_arr(m*(i-1)+(j-1))}$$

column major:

$$(\mathbf{A})_{ij} \leftrightarrow \texttt{A\_arr(n*(j-1)+(i-1))}$$



Example 1.2.20 (Accessing matrix data as a vector)

Both in MATLAB and EIGEN the single index access operator relies on the linear data layout: In MATLAB

```
A = [1 2 3;4 5 6;7 8 9]; A(:)',
```

produces the terminal output

```
    1     4     7     2     5     8     3     6     9
```

which clearly reveals the *column major* storage format.

In PYTHON the default data layout is row major, but it can be explicitly set. Further, array transposition does not change any data, but only the memory order and array shape.

PYTHON-code 1.2.21: Storage order in PYTHON

```
# array creation
A = np.array([[1, 2], [3, 4]])  # default (row major) storage
B = np.array([[1, 2], [3, 4]], order='F')  # column major storage

# show internal storage
np.ravel(A, 'K')  # array elements as stored in memory:  [1, 2, 3, 4]
np.ravel(B, 'K')  # array elements as stored in memory:  [1, 3, 2, 4]

# nothing happens to the data on transpose, just the storage order changes
np.ravel(A.T, 'K')  # array elements as stored in memory:  [1, 2, 3, 4]
np.ravel(B.T, 'K')  # array elements as stored in memory:  [1, 3, 2, 4]

# storage order can be accessed by checking the array's flags
A.flags['C_CONTIGUOUS']  # True
B.flags['F_CONTIGUOUS']  # True
A.T.flags['F_CONTIGUOUS']  # True
B.T.flags['C_CONTIGUOUS']  # True
```

In EIGEN the data layout can be controlled by a template argument; default is column major.

C++11 code 1.2.22: Single index access of matrix entries in EIGEN ⇝ GITLAB

```
2  void storageOrder(int nrows=6,int ncols=7)
3  {
4    cout << "Different matrix storage layouts in Eigen" << endl;
5    // Template parameter ColMajor selects column major data layout
6    Matrix<double,Dynamic,Dynamic,ColMajor> mcm(nrows,ncols);
7    // Template parameter RowMajor selects row major data layout
8    Matrix<double,Dynamic,Dynamic,RowMajor> mrm(nrows,ncols);
9    // Direct initialization; lazy option:  use int as index type
10   for (int l=1,i=0; i<nrows; i++)
11     for (int j=0; j<ncols; j++,l++)
12       mcm(i,j) = mrm(i,j) = l;
13
14   cout << "Matrix mrm = " << endl << mrm << endl;
15   cout << "mcm linear = ";
16   for (int l=0;l<mcm.size(); l++) cout << mcm(l) << ',';
17   cout << endl;
18
19   cout << "mrm linear = ";
20   for (int l=0;l<mrm.size(); l++) cout << mrm(l) << ',';
21   cout << endl;
22 }
```

The function call `storageOrder(3,3)`, *cf.* Code 1.2.22 yields the output

```
1 Different matrix storage layouts in Eigen
2 Matrix mrm =
3 1 2 3
4 4 5 6
5 7 8 9
6 mcm linear = 1,4,7,2,5,8,3,6,9,
7 mrm linear = 1,2,3,4,5,6,7,8,9,
```

Remark 1.2.23 (Vectorisation of a matrix)

Mapping a column-major matrix to a column vector with the same number of entries is called vectorization or linearization in numerical linear algebra, in symbols

$$\mathrm{vec} : \mathbb{K}^{n,m} \to \mathbb{K}^{n\cdot m} \quad , \quad \mathrm{vec}(\mathbf{A}) = \begin{bmatrix} (\mathbf{A})_{:,1} \\ (\mathbf{A})_{:,2} \\ \vdots \\ (\mathbf{A})_{:,m} \end{bmatrix} . \tag{1.2.24}$$

Remark 1.2.25 (MATLAB command reshape)

matlab offers the built-in command **reshape** for changing the dimensions of a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$:

```
B = reshape(k,l,A);     % error, in case kl ≠ mn
```

This command will create an $k \times l$-matrix by just reinterpreting the linear array of entries of `A` as data for a matrix with `k` rows and `l` columns. Regardless of the size and entries of the matrices the following test will always produce a `equal = true` result

```
if (()prod(size(A)) ~= (k*l)), error('Size mismatch'); end
B = reshape(A,k,l);
equal = (B(:) == A(:));
```

Remark 1.2.26 (NUMPY function reshape)

NUMPY offers the function `np.reshape` for changing the dimensions of a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$:

```
# read elements of A in row major order (default)
B = np.reshape(A, (k, l))   # error, in case kl ≠ mn
B = np.reshape(A, (k, l), order='C')   # same as above
# read elements of A in column major order
B = np.reshape(A, (k, l), order='F')
# read elements of A as stored in memory
B = np.reshape(A, (k, l), order='A')
```

This command will create an $k \times l$-array by reinterpreting the array of entries of `A` as data for an array with `k` rows and `l` columns. The order in which the elements of `A` are be read can be set by the `order` argument to row major (default, `'C'`), column major (`'F'`) or `A`'s internal storage order, i.e. row major if `A` is row major or column major if `A` is column major (`'A'`).

Remark 1.2.27 (Reshaping matrices in EIGEN)

If you need a reshaped view of a matrix' data in EIGEN you can obtain it via the raw data vector belonging to the matrix. Then use this information to create a matrix view by means of `Map` → documentation.

C++11 code 1.2.28: Demonstration on how reshape a matrix in EIGEN ↪ GITLAB

```
2  template<typename MatType>
3  void reshapetest(MatType &M)
4  {
5    using index_t = typename MatType::Index;
6    using entry_t = typename MatType::Scalar;
7    const index_t nsize(M.size());
8
9   // reshaping possible only for matrices with non-prime dimensions
10   if ((nsize %2) == 0) {
```

```
    entry_t *Mdat = M.data(); // raw data array for M
    // Reinterpretation of data of M
    Map<Eigen::Matrix<entry_t,Dynamic,Dynamic>> R(Mdat,2,nsize/2);
    // (Deep) copy data of M into matrix of different size
    Eigen::Matrix<entry_t,Dynamic,Dynamic> S =
      Map<Eigen::Matrix<entry_t,Dynamic,Dynamic>>(Mdat,2,nsize/2);

    cout << "Matrix M = " << endl << M << endl;
    cout << "reshaped to " << R.rows() << 'x' << R.cols()
      << " = " << endl << R << endl;
    // Modifying R affects M, because they share the data space !
    R *= -1.5;
    cout << "Scaled (!) matrix M = " << endl << M << endl;
    // Matrix S is not affected, because of deep copy
    cout << "Matrix S = " << endl << S << endl;
  }
}
```

This function has to be called with a mutable (l-value) matrix type object. A sample output is printed next:

```
Matrix M =
 0 -1 -2 -3 -4 -5 -6
 1  0 -1 -2 -3 -4 -5
 2  1  0 -1 -2 -3 -4
 3  2  1  0 -1 -2 -3
 4  3  2  1  0 -1 -2
 5  4  3  2  1  0 -1
reshaped to 2x21 =
 0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1 -6 -4 -2
 1  3  5  0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1
Scaled (!) matrix M =
   -0  1.5    3  4.5    6  7.5    9
-1.5   -0  1.5    3  4.5    6  7.5
  -3 -1.5   -0  1.5    3  4.5    6
-4.5   -3 -1.5   -0  1.5    3  4.5
  -6 -4.5   -3 -1.5   -0  1.5    3
-7.5   -6 -4.5   -3 -1.5   -0  1.5
Matrix S =
 0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1 -6 -4 -2
 1  3  5  0  2  4 -1  1  3 -2  0  2 -3 -1  1 -4 -2  0 -5 -3 -1
```

Experiment 1.2.29 (Impact of matrix data access patterns on runtime)

Modern CPU feature several levels of memories (registers, L1 cache, L2 cache, ..., main memory) of different latency, bandwidth, and size. Frequently accessing memory locations with widely different addresses results in many cache misses and will considerably slow down the CPU.

Two loops in MATLAB:

```
A = randn(n,n);
for j = 1:n-1,
  A(:,j+1) = A(:,j+1) - A(:,j);
end
```

column oriented access

```
A = randn(n,n);
for i = 1:n-1,
  A(i+1,:) = A(i+1,:) - A(i,:);
end
```

row oriented access

MATLAB-code 1.2.30: Timing for row and column oriented matrix access in MATLAB

```
% Timing for row/column operations on matrices
% We conduct K runs in order to reduce the risk of skewed maesurements
% due to OS activity during MATLAB run.
K = 3; res = [];
for n=2.^(4:13)
  A = randn(n,n);

  t1 = realmax;
  for k=1:K,  tic;
    for j = 1:n-1,  A(:,j+1) = A(:,j+1) - A(:,j); end;
    t1 = min(toc,t1);
  end
  t2 = realmax;
  for k=1:K,  tic;
    for i = 1:n-1,  A(i+1,:) = A(i+1,:) - A(i,:); end;
    t2 = min(toc,t2);
  end
  res = [res; n, t1 , t2];
end

% Plot runtimes versus matrix sizes
figure; plot(res(:,1),res(:,2),'r+', res(:,1),res(:,3),'m*');
xlabel('{\bf n}','fontsize',14);
ylabel('{\bf runtime [s]}','fontsize',14);
legend('A(:,j+1) = A(:,j+1) - A(:,j)','A(i+1,:) = A(i+1,:) -
   A(i,:)',...
       'location','northwest');
print -depsc2 '../PICTURES/accessrtlin.eps';

figure; loglog(res(:,1),res(:,2),'r+', res(:,1),res(:,3),'m*');
xlabel('{\bf n}','fontsize',14);
ylabel('{\bf runtime [s]}','fontsize',14);
legend('A(:,j+1) = A(:,j+1) - A(:,j)','A(i+1,:) = A(i+1,:) -
   A(i,:)',...
       'location','northwest');
print -depsc2 '../PICTURES/accessrtlog.eps';
```

C++11 code 1.2.31: Timing for row and column oriented matrix access for EIGEN ↪ GITLAB

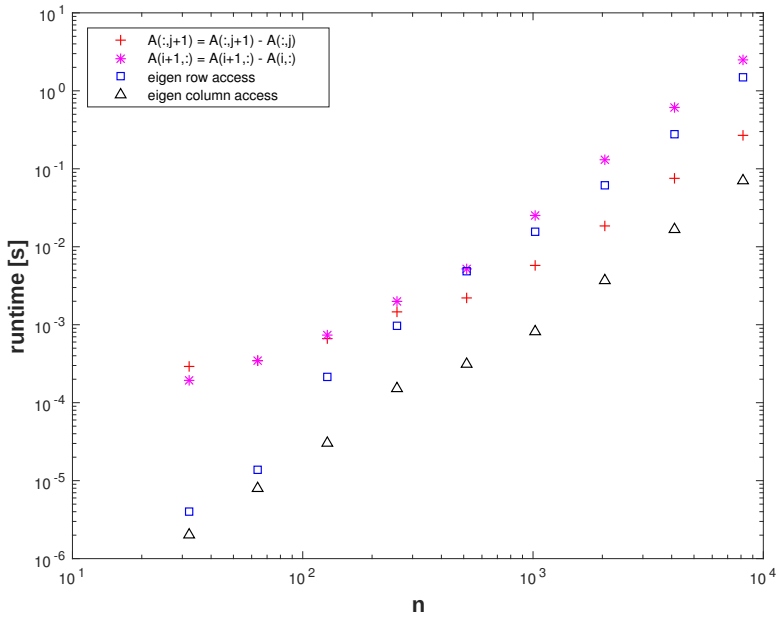
```
void rowcolaccesstiming(void)
{
  const int K = 3; // Number of repetitions
```

```
5   const int N_min = 5; // Smalles matrix size 32
6   const int N_max = 13; // Scan until matrix size of 8192
7   unsigned long n = (1L << N_min);
8   Eigen::MatrixXd times(N_max-N_min+1,3);
9
10  for(int l=N_min; l<= N_max; l++, n*=2) {
11    Eigen::MatrixXd A = Eigen::MatrixXd::Random(n,n);
12    double t1 = 1000.0;
13    for(int k=0;k<K;k++) {
14      auto tic =  high_resolution_clock::now();
15      for(int j=0; j < n-1; j++) A.row(j+1) -= A.row(j); // row access
16      auto toc = high_resolution_clock::now();
17      double t =
            (double)duration_cast<microseconds>(toc-tic).count()/1E6;
18      t1 = std::min(t1,t);
19    }
20    double t2 = 1000.0;
21    for(int k=0;k<K;k++) {
22      auto tic =  high_resolution_clock::now();
23      for(int j=0; j < n-1; j++) A.col(j+1) -= A.col(j); //column
            access
24      auto toc = high_resolution_clock::now();
25      double t =
            (double)duration_cast<microseconds>(toc-tic).count()/1E6;
26      t2 = std::min(t2,t);
27    }
28    times(l-N_min,0) = n;      times(l-N_min,1) = t1;
          times(l-N_min,2) = t2;
29  }
30  std::cout << times << std::endl;
31  }
```

PYTHON-code 1.2.32: Timing for row and column oriented matrix access in PYTHON

```
1  import numpy as np
2  import timeit
3  from matplotlib import pyplot as plt
4
5  def col_wise(A):
6      for j in range(A.shape[1] - 1):
7          A[:, j + 1] -= A[:, j]
8
9  def row_wise(A):
10     for i in range(A.shape[0] - 1):
11         A[i + 1, :] -= A[i, :]
12
13 # Timing for row/column-wise operations on matrix, we conduct k runs in
       order
14 # to reduce risk of skewed measurements due to OS activity during run.
15
```

```
16 k = 3
17 res = []
18 for n in 2**np.mgrid[4:14]:
19     A = np.random.normal(size=(n, n))
20
21     t1 = min(timeit.repeat(lambda: col_wise(A), repeat=k,
           number=1))
22     t2 = min(timeit.repeat(lambda: row_wise(A), repeat=k,
           number=1))
23
24     res.append((n, t1, t2))
25
26 # plot runtime versus matrix sizes
27 ns, t1s, t2s = np.transpose(res)
28
29 plt.figure()
30 plt.plot(ns, t1s, '+', label='A[:, j + 1] -= A[:, j]')
31 plt.plot(ns, t2s, 'o', label='A[i + 1, :] -= A[i, :]')
32 plt.xlabel(r'$n$')
33 plt.ylabel(r'runtime [s]')
34 plt.legend(loc='upper left')
35 plt.savefig('../PYTHON_PICTURES/accessrtlin.eps')
36
37 plt.figure()
38 plt.loglog(ns, t1s, '+', label='A[:, j + 1] -= A[:, j]')
39 plt.loglog(ns, t2s, 'o', label='A[i + 1, :] -= A[i, :]')
40 plt.xlabel(r'$n$')
41 plt.ylabel(r'runtime [s]')
42 plt.legend(loc='upper left')
43 plt.savefig('../PYTHON_PICTURES/accessrtlog.eps')
44
45 plt.show()
```



Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3, -DNDEBUG

The compiler flags `-O3` and `-DNDEBUG` are essential. The C++ code would be significantly slower if the default compiler options were used!

For both MATLAB and EIGEN codes we observe a glaring discrepancy of CPU time required for accessing entries of a matrix in rowwise or columnwise fashion. This reflects the impact of features of the unterlying hardware architecture, like cache size and memory bandwidth:

Interpretation of timings: Since matrices in MATLAB are stored column major all the matrix elements in a column occupy contiguous memory locations, which will all reside in the cache together. Hence, column oriented access will mainly operate on data in the cache even for large matrices. Conversely, row oriented access addresses matrix entries that are stored in distant memory locations, which incurs frequent cash misses (cache thrashing).

The impact of hardware architecture on the performance of algorithms will **not** be taken into account in this course, because hardware features tend to be both intricate and ephemeral. However, for modern high performance computing it is essential to adapt implementations to the hardware on which the code is supposed to run.

1.3 Basic linear algebra operations

First we refresh the basic rules of vector and matrix calculus. Then we will learn about a very old programming interface for simple dense linear algebra operations.

1.3.1 Elementary matrix-vector calculus

What you should know from linear algebra [**?**, Sect. 2.2]:

- vector space operations in matrix space $\mathbb{K}^{m,n}$ (addition, multiplication with scalars)
- dot product: $\mathbf{x},\mathbf{y} \in \mathbb{K}^n$, $n \in \mathbb{N}$: $\mathbf{x}\cdot\mathbf{y} := \mathbf{x}^H\mathbf{y} = \sum_{i=1}^{n} \bar{x}_i y_i \in \mathbb{K}$

  (in EIGEN: `x.dot(y)` or `x.adjoint()*y`, `x`,`y` $\hat{=}$ column vectors)

- tensor product: $\mathbf{x} \in \mathbb{K}^m, \mathbf{y} \in \mathbb{K}^n, n \in \mathbb{N}$: $\mathbf{x}\mathbf{y}^H = \left(x_i \bar{y}_j\right)_{\substack{i=1,\ldots,m \\ j=1,\ldots,n}} \in \mathbb{K}^{m,n}$
  (in EIGEN: `x*y.adjoint()`, `x,y` $\hat{=}$ column vectors)
- All are special cases of the matrix product:

$$\mathbf{A} \in \mathbb{K}^{m,n}, \quad \mathbf{B} \in \mathbb{K}^{n,k}: \qquad \mathbf{AB} = \left[\sum_{j=1}^{n} a_{ij} b_{jl}\right]_{\substack{i=1,\ldots,m \\ l=1,\ldots,k}} \in \mathbb{R}^{m,k}. \tag{1.3.1}$$

Recall from linear algebra basic properties of the matrix product: for all $\mathbb{K}$-matrices $\mathbf{A}, \mathbf{B}, \mathbf{C}$ (of suitable sizes), $\alpha, \beta \in \mathbb{K}$

associative:	$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$,
bi-linear:	$(\alpha\mathbf{A} + \beta\mathbf{B})\mathbf{C} = \alpha(\mathbf{AC}) + \beta(\mathbf{BC})$, $\mathbf{C}(\alpha\mathbf{A} + \beta\mathbf{B}) = \alpha(\mathbf{CA}) + \beta(\mathbf{CB})$,
non-commutative:	$\mathbf{AB} \neq \mathbf{BA}$ in general.

(1.3.2) Visualisation of (special) matrix products

Dependency of an entry of a product matrix:

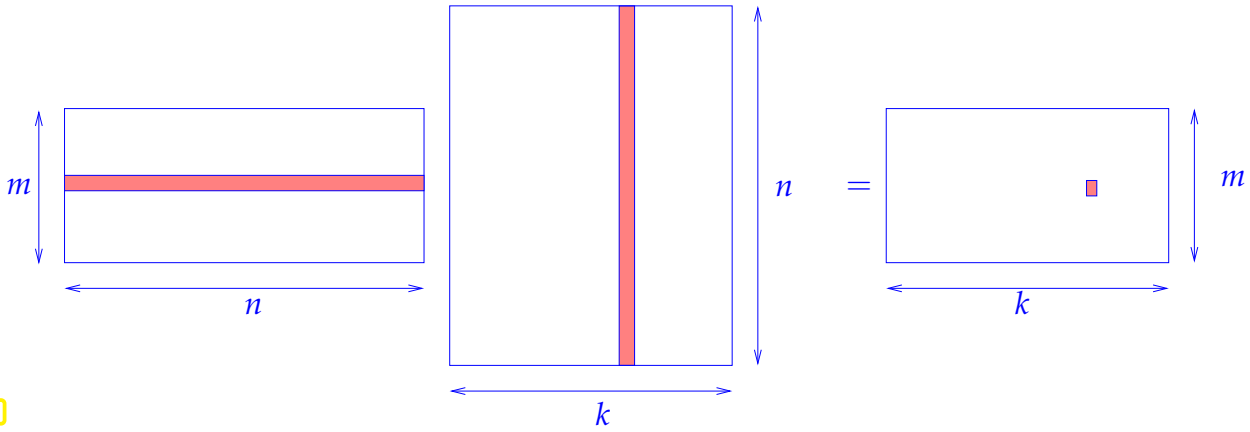
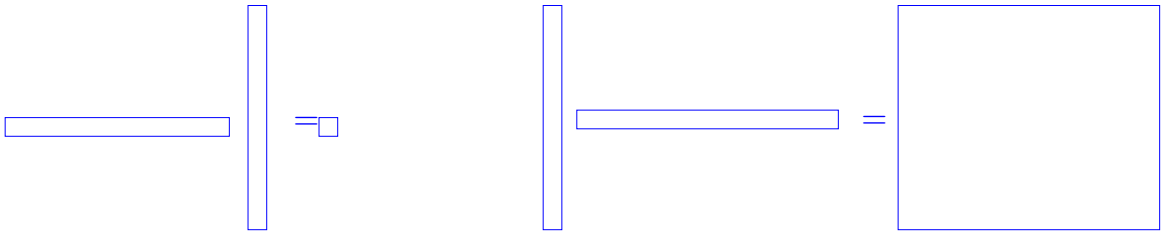


Fig. 29



dot product tensor product

Remark 1.3.3 (Row-wise & column-wise view of matrix product)

To understand what is going on when forming a matrix product, it is often useful to decompose it into matrix×vector operations in one of the following two ways:

$\mathbf{A} \in \mathbb{K}^{m,n}$, $\mathbf{B} \in \mathbb{K}^{n,k}$:

$$\mathbf{AB} = \begin{bmatrix} \mathbf{A}(\mathbf{B})_{:,1} & \ldots & \mathbf{A}(\mathbf{B})_{:,k} \end{bmatrix} \quad , \quad \mathbf{AB} = \begin{bmatrix} (\mathbf{A})_{1,:}\mathbf{B} \\ \vdots \\ (\mathbf{A})_{m,:}\mathbf{B} \end{bmatrix} . \tag{1.3.4}$$

matrix assembled from columns — matrix assembled from rows

For notations refer to Sect. 1.1.1.

Remark 1.3.5 (Understanding the structure of product matrices)

A "mental image" of matrix multiplication is useful for telling special properties of product matrices.

For instance, zero blocks of the product matrix can be predicted easily in the following situations using the idea explained in Rem. 1.3.3 (try to understand how):

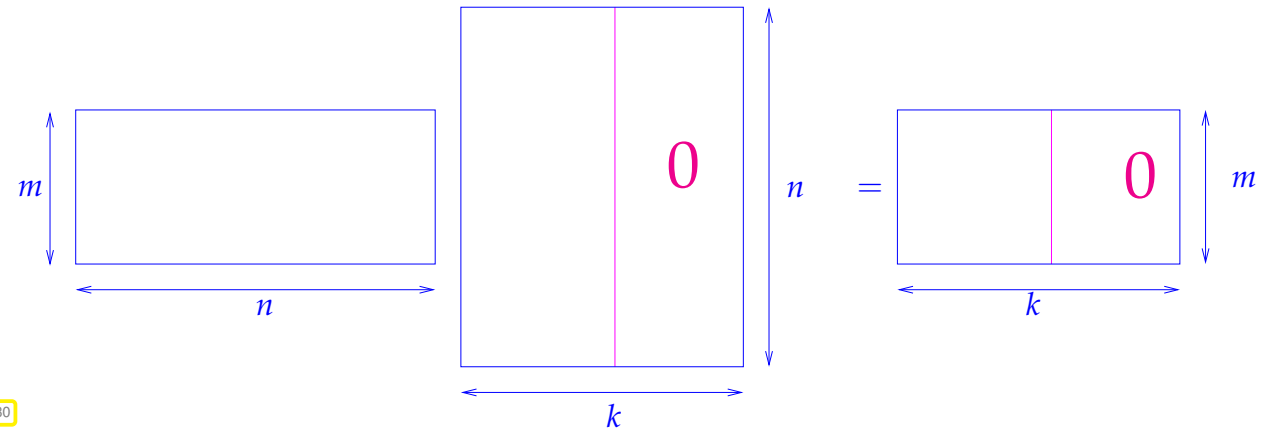


Fig. 30

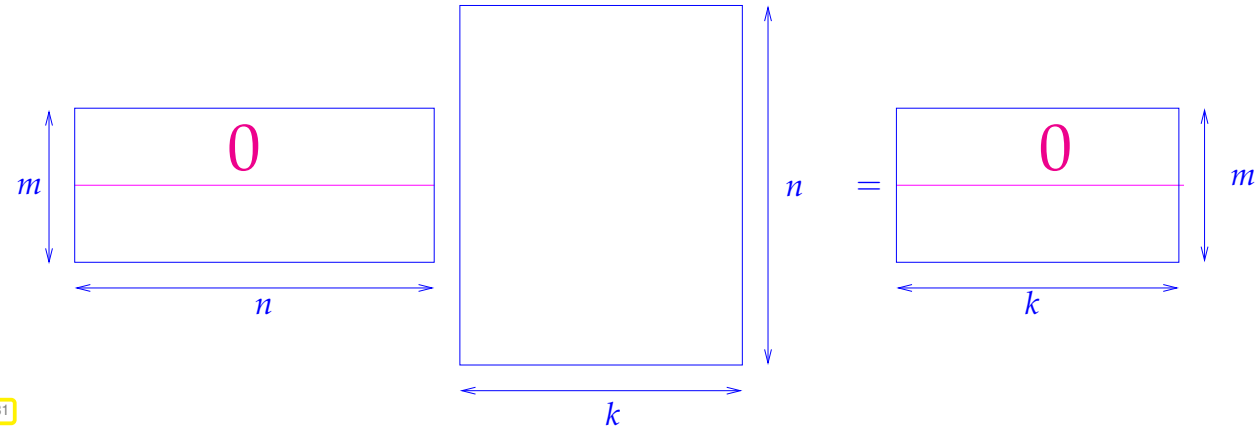


Fig. 31

A clear understanding of matrix multiplication enables you to "see", which parts of a matrix factor matter in a product:

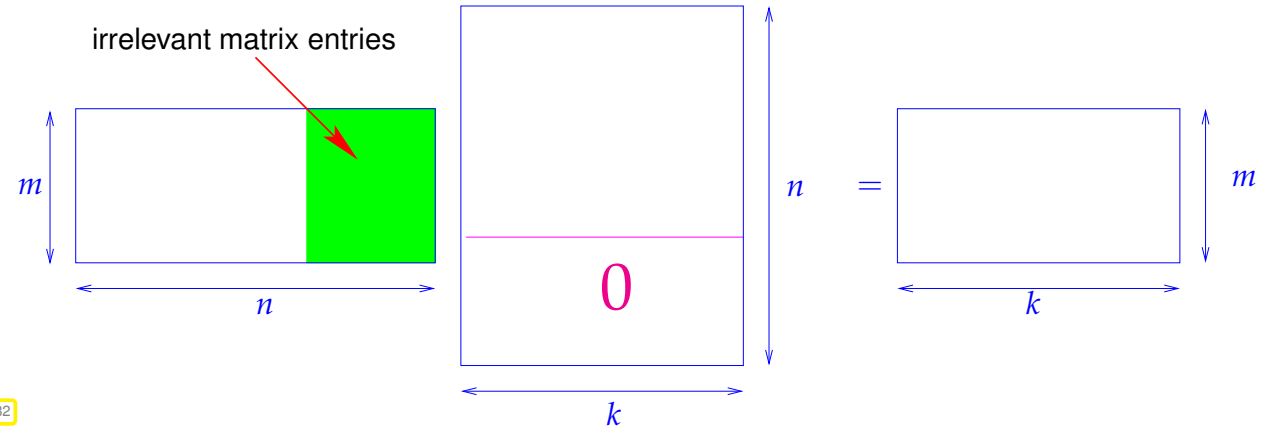
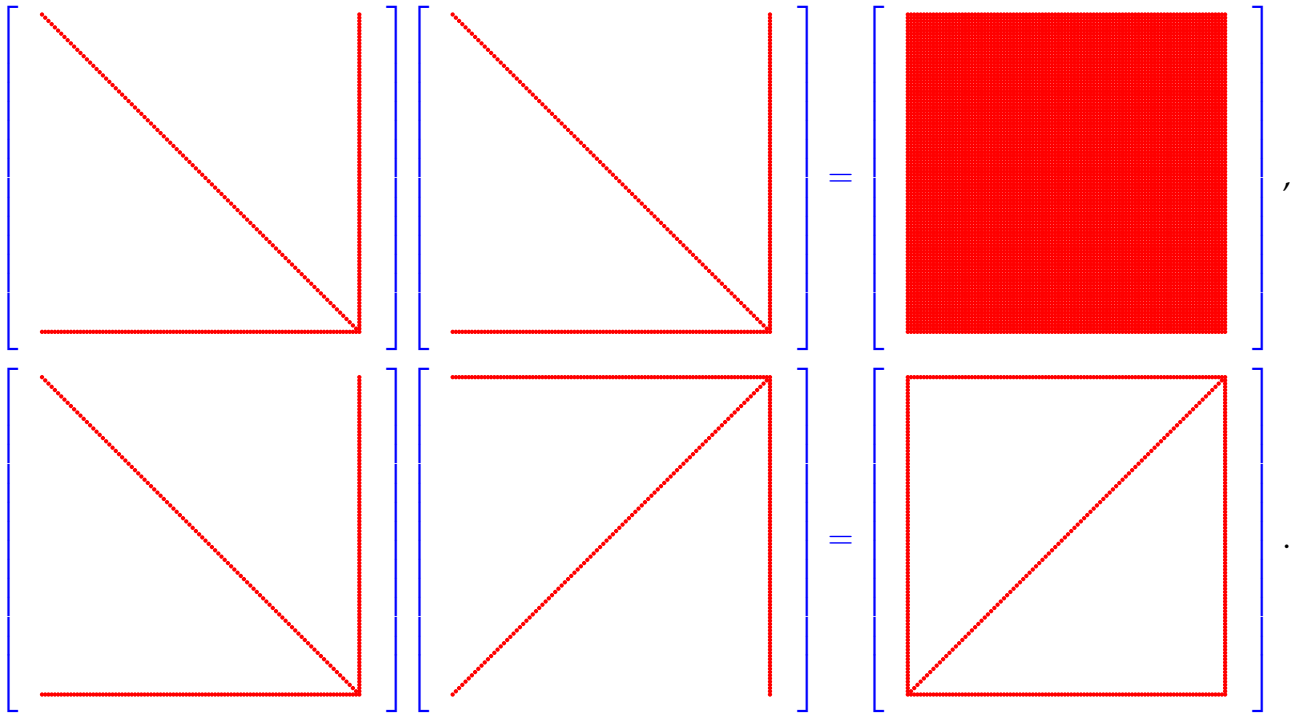


Fig. 32

"Seeing" the structure/pattern of a matrix product:



These nice renderings of the so-called patterns of matrices, that is, the distribution of their non-zero entries have been created by a special EIGEN/**Figure**-command for visualizing the structure of a matrix: `fig.spy(M)`

C++11 code 1.3.6: Visualizing the structure of matrices in EIGEN ↬ GITLAB

```
2  #include <Eigen/Dense>
3
4  #include <figure/figure.hpp>
5
6  using namespace Eigen;
7
8  int main() {
9      int n = 100;
10     MatrixXd A(n,n), B(n,n); A.setZero(); B.setZero();
11     A.diagonal() = VectorXd::LinSpaced(n,1,n);
12     A.col(n-1) = VectorXd::LinSpaced(n,1,n);
13     A.row(n-1) = RowVectorXd::LinSpaced(n,1,n);
```

```
14      B = A.colwise().reverse();
15      MatrixXd C = A*A, D = A*B;
16      mgl::Figure fig1, fig2, fig3, fig4;
17      fig1.spy(A);    fig1.save("Aspy_cpp");
18      fig2.spy(B);    fig2.save("Bspy_cpp");
19      fig3.spy(C);    fig3.save("Cspy_cpp");
20      fig4.spy(D);    fig4.save("Dspy_cpp");
21      return 0;
22  }
```

This code also demonstrates the use of `diagonal()`, `col()`, `row()` for L-value access to parts of a matrix.

PYTHON/MATPLOTLIB-command for visualizing the structure of a matrix: `plt.spy(M)`

PYTHON-code 1.3.7: Visualizing the structure of matrices in PYTHON

```
1 n = 100
2 A = np.diag(np.mgrid[:n])
3 A[:, -1] = A[-1, :] = np.mgrid[:n]
4 plt.spy(A)
5 plt.spy(A[::-1, :])
6 plt.spy(np.dot(A, A))
7 plt.spy(np.dot(A, B))
```

Remark 1.3.8 (Multiplying triangular matrices)

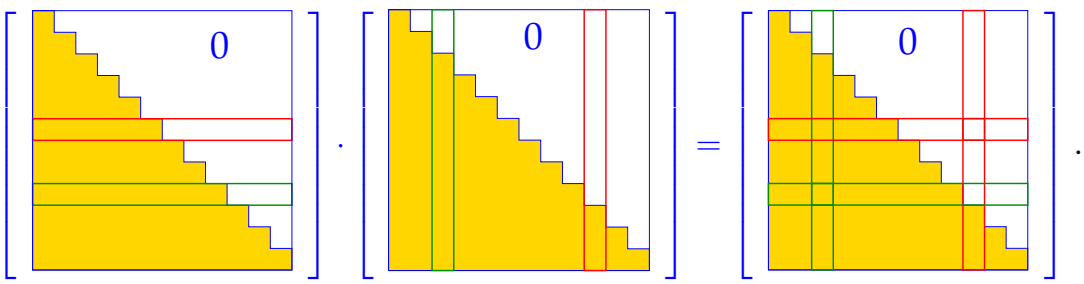
The following result is useful when dealing with matrix decompositions that often involve triangular matrices.

Lemma 1.3.9. Group of regular diagonal/triangular matrices

$$\mathbf{A},\mathbf{B} \begin{Bmatrix} \text{diagonal} \\ \text{upper triangular} \\ \text{lower triangular} \end{Bmatrix} \Rightarrow \mathbf{AB} \text{ and } \mathbf{A}^{-1} \begin{Bmatrix} \text{diagonal} \\ \text{upper triangular} \\ \text{lower triangular} \end{Bmatrix}.$$

(assumes that $\mathbf{A}$ is regular)

"Proof by visualization" → Rem. 1.3.5



Experiment 1.3.10 (Scaling a matrix)

Scaling = multiplication with diagonal matrices (with non-zero diagonal entries):

It is important to know the different effect of multiplying with a diagonal matrix from left or right:

- multiplication with diagonal matrix *from left* ➤ row scaling

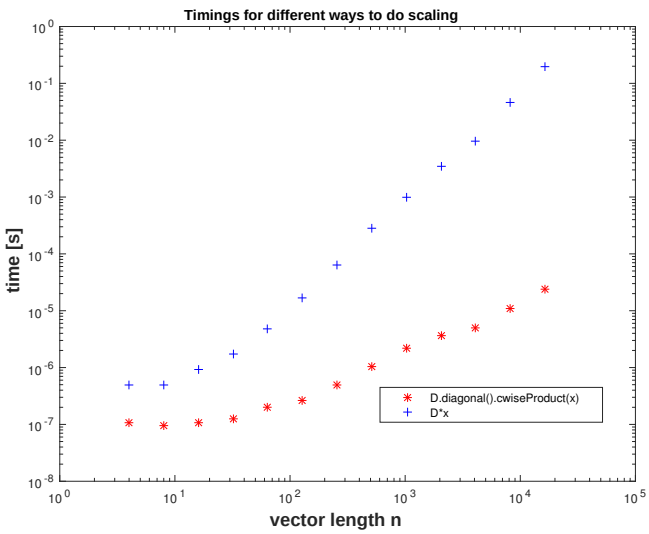
$$\begin{bmatrix} d_1 & 0 & & 0 \\ 0 & d_2 & & 0 \\ & & \ddots & \\ 0 & 0 & & d_n \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & & a_{2m} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} = \begin{bmatrix} d_1a_{11} & d_1a_{12} & \dots & d_1a_{1m} \\ d_2a_{21} & d_2a_{22} & \dots & d_2a_{2m} \\ \vdots & & & \vdots \\ d_na_{n1} & d_na_{n2} & \dots & d_na_{nm} \end{bmatrix} = \begin{bmatrix} d_1(\mathbf{A})_{1,:} \\ \vdots \\ d_n(\mathbf{A})_{n,:} \end{bmatrix}.$$

- multiplication with diagonal matrix *from right* ➤ column scaling

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & & a_{2m} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} \begin{bmatrix} d_1 & 0 & & 0 \\ 0 & d_2 & & 0 \\ & & \ddots & \\ 0 & 0 & & d_m \end{bmatrix} = \begin{bmatrix} d_1a_{11} & d_2a_{12} & \dots & d_ma_{1m} \\ d_1a_{21} & d_2a_{22} & \dots & d_ma_{2m} \\ \vdots & & & \vdots \\ d_1a_{n1} & d_2a_{n2} & \dots & d_ma_{nm} \end{bmatrix}$$
$$= \begin{bmatrix} d_1(\mathbf{A})_{:,1} & \dots & d_m(\mathbf{A})_{:,m} \end{bmatrix}.$$

Multiplication with a scaling matrix $\mathbf{D} := \operatorname{diag}(d_1,\dots,d_n) \in \mathbb{R}^{n,n}$ in EIGEN can be realised in three ways, see Code 1.3.11, **??** 9-**??** 1 **??** 13, and **??** 15.

The code will be slowed down massively in cas a temporary dense matrix is created inadvertentl Notice that EIGEN's expression templates avoid th pointless effort, see **??** 15.



C++11 code 1.3.11: Timing multiplication with scaling matrix in EIGEN ⇝ GITLAB

```
2  int nruns = 3, minExp = 2, maxExp = 14;
3  MatrixXd tms(maxExp-minExp+1,4);
4  for(int i = 0; i <= maxExp-minExp; ++i){
5    Timer tbad, tgood, topt; // timer class
6    int n = std::pow(2, minExp + i);
7    VectorXd d = VectorXd::Random(n,1), x = VectorXd::Random(n,1),
       y(n);
8    for(int j = 0; j < nruns; ++j) {
9      MatrixXd D = d.asDiagonal(); //
```

```
10        // matrix vector multiplication
11        tbad.start(); y = D*x; tbad.stop(); //
12        // componentwise multiplication
13        tgood.start(); y= d.cwiseProduct(x); tgood.stop(); //
14        // matrix multiplication optimized by Eigen
15        topt.start(); y = d.asDiagonal()*x; topt.stop(); //
16      }
17      tms(i,0)=n;
18      tms(i,1)=tgood.min(); tms(i,2)=tbad.min(); tms(i,3)= topt.min();
19    }
```

PYTHON-code 1.3.12: Timing multiplication with scaling matrix in PYTHON

```
import numpy as np
from matplotlib import pyplot as plt
import timeit

# script for timing a smart and foolish way to carrry out
# multiplication with a scaling matrix

nruns = 3
res = []
for n in 2**np.mgrid[2:15]:
    d = np.random.uniform(size=n)
    x = np.random.uniform(size=n)

    tbad = min(timeit.repeat(lambda: np.dot(np.diag(d), x),
                             repeat=nruns, number=1))
    tgood = min(timeit.repeat(lambda: d * x, repeat=nruns,
                              number=1))

    res.append((n, tbad, tgood))

ns, tbads, tgoods = np.transpose(res)
plt.figure()
plt.loglog(ns, tbads, '+', label='using np.diag')
plt.loglog(ns, tgoods, 'o', label='using *')
plt.legend(loc='best')
plt.title('Timing for different ways to do scaling')
plt.savefig('../PYTHON_PICTURES/scaletiming.eps')
plt.show()
```

Hardly surprising, the component-wise multiplication of the two vectors is way faster than the intermittent initialisation of a diagonal matrix (main populated by zeros) and the computation of a matrix×vector product. Nevertheless, such blunders keep on haunting numerical codes. Do not rely solely on EIGEN optimizations!

Remark 1.3.13 (Row and column transformations)

Simple operations on rows/columns of matrices, *cf.* what was done in Exp. 1.2.29, can often be expressed as multiplication with special matrices: For instance, given $\mathbf{A} \in \mathbb{K}^{n,m}$ we obtain $\mathbf{B}$ by adding row $(\mathbf{A})_{j,:}$ to row $(\mathbf{A})_{j+1,:}$, $1 \leq j < n$.

Realisation through matrix product ⟹

$$\mathbf{B} = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & 1 & 1 & & \\ & & & & \ddots & \\ & & & & & 1 \end{bmatrix} \mathbf{A} \,.$$

The matrix multiplying $\mathbf{A}$ from the left is a specimen of a transformation matrix, a matrix that coincides with the identity matrix $\mathbf{I}$ except for a single off-diagonal entry.

left-multiplication right-multiplication	with transformation matrices ↣	row transformations column transformations

row/column transformations will play a central role in Sect. 2.3

Remark 1.3.14 (Matrix algebra)

A vector space $(V, \mathbb{K}, +, \cdot)$, where V is additionally equipped with a bi-linear and associative "multiplication" is called an algebra. Hence, the vector space of square matrices $\mathbb{K}^{n,n}$ with matrix multiplication is an algebra with *unit element* $\mathbf{I}$.

(1.3.15) Block matrix product

Given matrix dimensions $M, N, K \in \mathbb{N}$ block sizes $1 \leq n < N$ ($n' := N - n$), $1 \leq m < M$ ($m' := M - m$), $1 \leq k < K$ ($k' := K - k$) we start from the following matrices:

$$\begin{matrix} \mathbf{A}_{11} \in \mathbb{K}^{m,n} & \mathbf{A}_{12} \in \mathbb{K}^{m,n'} \\ \mathbf{A}_{21} \in \mathbb{K}^{m',n} & \mathbf{A}_{22} \in \mathbb{K}^{m',n'} \end{matrix} \quad , \quad \begin{matrix} \mathbf{B}_{11} \in \mathbb{K}^{n,k} & \mathbf{B}_{12} \in \mathbb{K}^{n,k'} \\ \mathbf{B}_{21} \in \mathbb{K}^{n',k} & \mathbf{B}_{22} \in \mathbb{K}^{n',k'} \end{matrix} .$$

This matrices serve as sub-matrices or matrix blocks and are assembled into larger matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \in \mathbb{K}^{M,N} \quad , \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} \in \mathbb{K}^{N,K} .$$

It turns out that the matrix product $\mathbf{AB}$ can be computed by the same formula as the product of simple 2×2-matrices:

⟹

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix} . \tag{1.3.16}$$

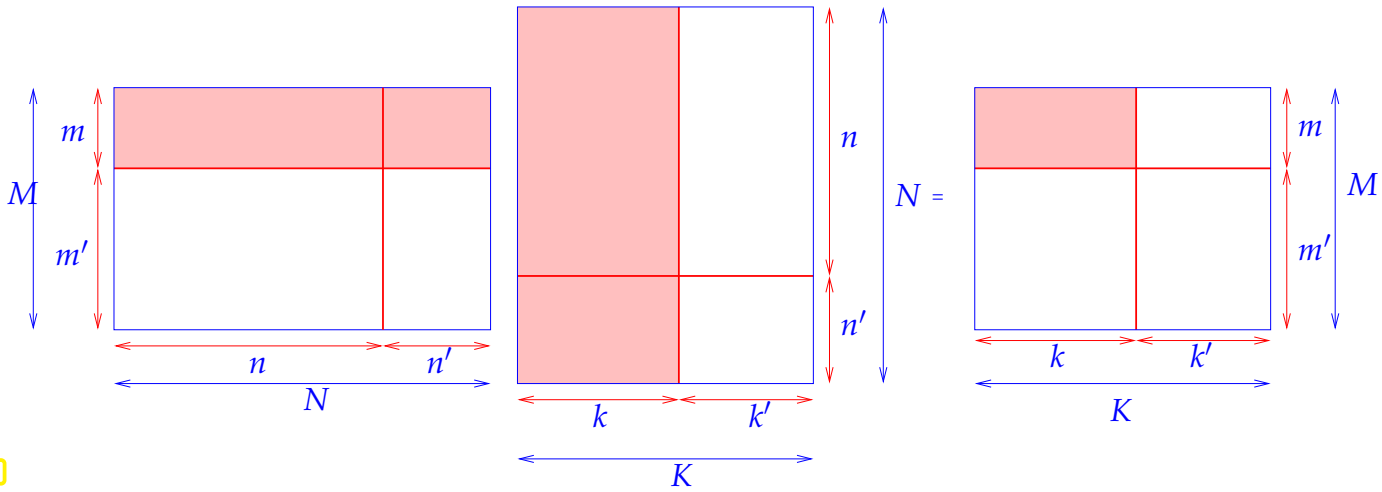


Fig. 34

Bottom line: one can compute with block-structured matrices in *almost* (∗) the same ways as with matrices with real/complex entries, see [**?**, Sect. 1.3.3].

(∗): you must not use the commutativity of multiplication (because matrix multiplication is not commutative).

1.3.2 BLAS – Basic Linear Algebra Subprograms

BLAS (Basic Linear Algebra Subprograms) is a specification (API) that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication. They are the de facto low-level routines for linear algebra libraries (Wikipedia).

The BLAS API is standardised by the BLAS technical forum and, due to its history dating back to the 70s, follows conventions of FORTRAN 77, see the Quick Reference Guide for examples. However, wrappers for other programming languages are available. CPU manufacturers and/or developers of operating systems usually supply highly optimised implementations:

- OpenBLAS: open source implementation with some general optimisations, available under BSD license.
- ATLAS (Automatically Tuned Linear Algebra Software): open source BLAS implementation with auto-tuning capabilities. Comes with C and FORTRAN interfaces and is included in Linux distributions.
- Intel MKL (Math Kernel Library): commercial highly optimised BLAS implemetation available for all Intel CPUs. Used by most proprietory simulation software and also MATLAB.

Experiment 1.3.17 (Multiplying matrices in MATLAB)

MATLAB-code 1.3.18: Timing different implementations of matrix multiplication in MATLAB

```
% MATLAB script for timing different implementations of matrix
    multiplications
nruns = 3; times = [];
for n=2.^(2:10) % n = size of matrices
  fprintf('matrix size n = %d\n',n);
  A = rand(n,n); B = rand(n,n); C = zeros(n,n);
```

```
  t1 = realmax;
  % loop based implementation (no BLAS)
  for l=1:nruns
    tic;
    for i=1:n, for j=1:n
        for k=1:n, C(i,j) = C(i,j) + A(i,k)*B(k,j); end
      end, end
    t1 = min(t1,toc);
    end
  t2 = realmax;
  % dot product based implementation (BLAS level 1)
  for l=1:nruns
    tic;
    for i=1:n
      for j=1:n, C(i,j) = dot(A(i,:),B(:,j));  end
    end
    t2 = min(t2,toc);
  end
  t3 = realmax;
  % matrix-vector based implementation (BLAS level 2)
  for l=1:nruns
    tic;
    for j=1:n, C(:,j) = A*B(:,j); end
    t3 = min(t3,toc);
  end
  t4 = realmax;
  % BLAS level 3 matrix multiplication
  for l=1:nruns
    tic; C = A*B; t4 = min(t4,toc);
  end
  times = [ times; n t1 t2 t3 t4];
end

figure('name','mmtiming');
loglog(times(:,1),times(:,2),'r+-',...
     times(:,1),times(:,3),'m*-',...
     times(:,1),times(:,4),'b^-',...
     times(:,1),times(:,5),'kp-');
title('Timings: Different implementations of matrix
   multiplication');
xlabel('matrix size n','fontsize',14);
ylabel('time [s]','fontsize',14);
legend('loop implementation','dot product implementation',...
       'matrix-vector implementation','BLAS gemm (MATLAB *)',...
       'location','northwest');

print -depsc2 '../PICTURES/mvtiming.eps';
```

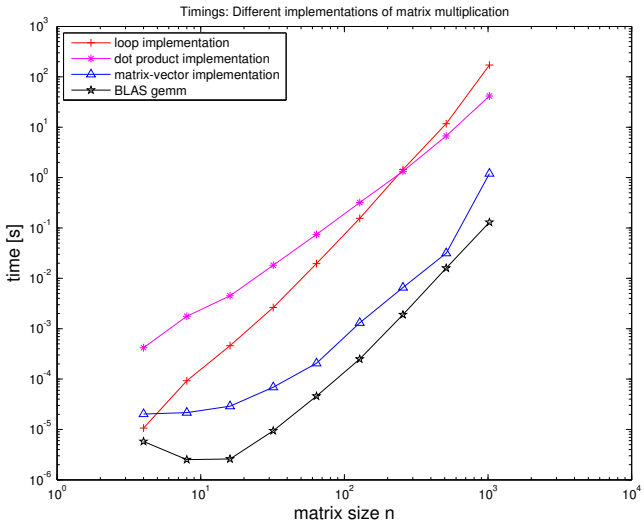


Fig. 35

Platform:

- Mac OS X 10.6
- Intel Core 7, 2.66 GHz
- L2 256 kB, L3 4 MB, Mem 4 GB
- MATLAB 7.10.0 (R 2010a)

In MATLAB we can achieve a tremendous gain in execution speed by relying on compact matrix/vector operations that invoke efficient BLAS routine.

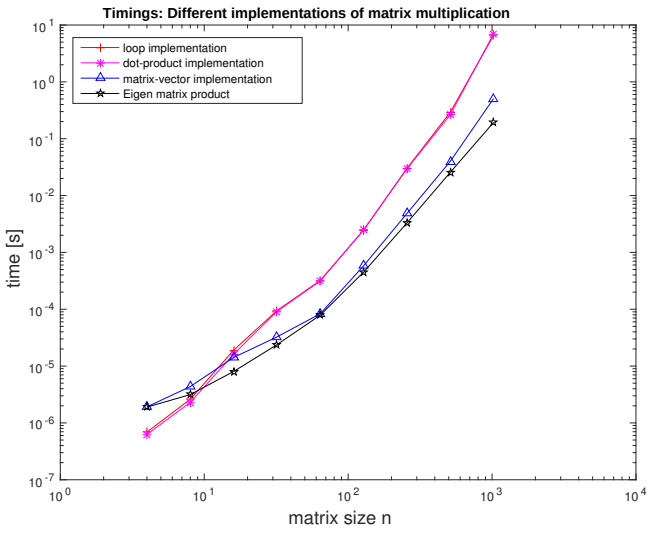
Advise: avoid loops in MATLAB and replace them with vectorised operations.

To some extent the same applies to EIGEN code, a corresponding timing script is given here:

C++11 code 1.3.19: Timing different implementations of matrix multiplication in EIGEN
➙ GITLAB

```
//!  script for timing different implementations of matrix
   multiplications
//!  no BLAS is used in Eigen!
void mmtiming(){
  int nruns = 3, minExp = 2, maxExp = 10;
  MatrixXd timings(maxExp-minExp+1,5);
  for(int p = 0; p <= maxExp-minExp; ++p){
    Timer t1, t2, t3, t4;    // timer class
    int n = std::pow(2, minExp + p);
    MatrixXd A = MatrixXd::Random(n,n);
    MatrixXd B = MatrixXd::Random(n,n);
    MatrixXd C = MatrixXd::Zero(n,n);
    for(int q = 0; q < nruns; ++q){
      // Loop based implementation no template magic
      t1.start();
      for(int i = 0; i < n; ++i)
        for(int j = 0; j < n; ++j)
          for(int k = 0; k < n; ++k)
            C(i,j) += A(i,k)*B(k,j);
      t1.stop();
      // dot product based implementation little template magic
      t2.start();
      for(int i = 0; i < n; ++i)
        for(int j = 0; j < n; ++j)
          C(i,j) = A.row(i).dot( B.col(j) );
      t2.stop();
      // matrix-vector based implementation middle template magic
      t3.start();
      for(int j = 0; j < n; ++j)
        C.col(j) = A * B.col(j);
      t3.stop();
      // Eigen matrix multiplication template magic optimized
      t4.start();
```

```
34            C = A * B;
35            t4.stop();
36        }
37        timings(p,0)=n; timings(p,1)=t1.min(); timings(p,2)=t2.min();
38        timings(p,3)=t3.min(); timings(p,4)=t4.min();
39    }
40    std::cout << std::scientific << std::setprecision(3) << timings <<
          std::endl;
41    //Plotting
42    mgl::Figure fig;
43    fig.setFontSize(4);
44    fig.setlog(true, true);
45    fig.plot(timings.col(0),timings.col(1), " +r-").label("loop
          implementation");
46    fig.plot(timings.col(0),timings.col(2)," *m-").label("dot-product
          implementation");
47    fig.plot(timings.col(0),timings.col(3)," ^b-").label("matrix-vector
          implementation");
48    fig.plot(timings.col(0),timings.col(4)," ok-").label("Eigen matrix
          product");
49    fig.xlabel("matrix size n"); fig.ylabel("time [s]");
50    fig.legend(0.05,0.95); fig.save("mmtiming");
51 }
```



Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3

In EIGEN we can achieve some gain in execution speed by relying on compact matrix/vector operations that invoke efficient EIGEN routine. Notice that loops are not as punished as in MATLAB!

The same applies to PYTHON code, a corresponding timing script is given here:

PYTHON-code 1.3.20: Timing different implementations of matrix multiplication in PYTHON

```
1 # script for timing different implementations of matrix multiplications
2 import numpy as np
3 from matplotlib import pyplot as plt
4 import timeit
5
6 def mm_loop_based(A, B, C):
7     m, n = A.shape
8     _, p = B.shape
```

```
    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]
    return C

def mm_blas1(A, B, C):
    m, n = A.shape
    _, p = B.shape
    for i in range(m):
        for j in range(p):
            C[i, j] = np.dot(A[i, :], B[:, j])
    return C

def mm_blas2(A, B, C):
    m, n = A.shape
    _, p = B.shape
    for i in range(m):
        C[i, :] = np.dot(A[i, :], B)
    return C

def mm_blas3(A, B, C):
    C = np.dot(A, B)
    return C

def main():
    nruns = 3
    res = []
    for n in 2**np.mgrid[2:11]:
        print('matrix size n = {}'.format(n))
        A = np.random.uniform(size=(n, n))
        B = np.random.uniform(size=(n, n))
        C = np.random.uniform(size=(n, n))

        tloop = min(timeit.repeat(lambda: mm_loop_based(A, B, C),
                                  repeat=nruns, number=1))
        tblas1 = min(timeit.repeat(lambda: mm_blas1(A, B, C),
                                   repeat=nruns, number=1))
        tblas2 = min(timeit.repeat(lambda: mm_blas2(A, B, C),
                                   repeat=nruns, number=1))
        tblas3 = min(timeit.repeat(lambda: mm_blas3(A, B, C),
                                   repeat=nruns, number=1))
        res.append((n, tloop, tblas1, tblas2, tblas3))

    ns, tloops, tblas1s, tblas2s, tblas3s = np.transpose(res)
    plt.figure()
    plt.loglog(ns, tloops, 'o', label='loop implementation')
    plt.loglog(ns, tblas1s, '+', label='dot product
       implementation')
    plt.loglog(ns, tblas2s, '*', label='matrix-vector
```

```
        implementation')
    plt.loglog(ns, tblas3s, '^', label='BLAS gemm (np.dot)')
    plt.legend(loc='upper left')
    plt.savefig('../PYTHON_PICTURES/mvtiming.eps')
    plt.show()

if __name__ == '__main__':
    main()
```

BLAS routines are grouped into "levels" according to the amount of data and computation involved (asymptotic complexity, see Section 1.4.1 and [**?**, Sect. 1.1.12]):

- **Level 1**: vector operations such as scalar products and vector norms.
  asymptotic complexity $O(n)$, (with $n \hat{=}$ vector length),
  e.g.: dot product: $\rho = \mathbf{x}^\top \mathbf{y}$
- **Level 2**: vector-matrix operations such as matrix-vector multiplications.
  asymptotic complexity $O(mn)$,(with $(m,n) \hat{=}$ matrix size),
  e.g.: matrix×vector multiplication: $\mathbf{y} = \alpha \mathbf{A}\mathbf{x} + \beta \mathbf{y}$
- **Level 3**: matrix-matrix operations such as matrix additions or multiplications.
  asymptotic complexity often $O(nmk)$,(with $(n,m,k) \hat{=}$ matrix sizes),
  e.g.: matrix product: $\mathbf{C} = \mathbf{A}\mathbf{B}$

Syntax of BLAS calls:
The functions have been implemented for different types, and are distinguished by the first letter of the function name. E.g. *sdot* is the dot product implementation for single precision and *ddot* for double precision.

✦ **BLAS LEVEL 1**: vector operations, asymptotic complexity $O(n)$, $n \hat{=}$ vector length

- dot product $\rho = \mathbf{x}^\top \mathbf{y}$

  xDOT(N,X,INCX,Y,INCY)

  - x $\in \{S, D\}$, scalar type: S $\hat{=}$ type `float`, D $\hat{=}$ type `double`
  - $N \hat{=}$ length of vector (modulo stride INCX)
  - $X \hat{=}$ vector $\mathbf{x}$: array of type x
  - `INCX` $\hat{=}$ stride for traversing vector X
  - $Y \hat{=}$ vector $\mathbf{y}$: array of type x
  - `INCY` $\hat{=}$ stride for traversing vector Y
- vector operations $\mathbf{y} = \alpha \mathbf{x} + \mathbf{y}$

  xAXPY(N,ALPHA,X,INCX,Y,INCY)

  - x $\in \{S, D, C, Z\}$, S $\hat{=}$ type `float`, D $\hat{=}$ type `double`, C $\hat{=}$ type `complex`
  - $N \hat{=}$ length of vector (modulo stride INCX)

- ALPHA $\hat{=}$ scalar α
- $X \hat{=}$ vector $\mathbf{x}$: array of type x
- INCX $\hat{=}$ stride for traversing vector X
- $Y \hat{=}$ vector $\mathbf{y}$: array of type x
- INCY $\hat{=}$ stride for traversing vector Y

✦ **BLAS LEVEL 2**: matrix-vector operations, asymptotic complexity $O(mn)$, $(m,n) \hat{=}$ matrix size

- matrix×vector multiplication $\mathbf{y} = \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$

```
xGEMV(TRANS,M,N,ALPHA,A,LDA,X,
INCX,BETA,Y,INCY)
```

  - x $\in \{S, D, C, Z\}$, scalar type: S $\hat{=}$ type `float`, D $\hat{=}$ type `double`, C $\hat{=}$ type `complex`
  - $M, N \hat{=}$ size of matrix $\mathbf{A}$
  - ALPHA $\hat{=}$ scalar parameter α
  - A $\hat{=}$ matrix $\mathbf{A}$ stored in *linear array* of length $M \cdot N$ (column major arrangement)

$$(\mathbf{A})_{i,j} = \mathrm{A}[N * (j-1) + i] \,.$$

  - LDA $\hat{=}$ "leading dimension" of $\mathbf{A} \in \mathbb{K}^{n,m}$, that is, the number n of rows.
  - $X \hat{=}$ vector $\mathbf{x}$: array of type x
  - INCX $\hat{=}$ stride for traversing vector X
  - BETA $\hat{=}$ scalar paramter β
  - $Y \hat{=}$ vector $\mathbf{y}$: array of type x
  - INCY $\hat{=}$ stride for traversing vector Y

- **BLAS LEVEL 3**: matrix-matrix operations, asymptotic complexity $O(mnk)$, $(m,n,k) \hat{=}$ matrix sizes

  - matrix×matrix multiplication $\mathbf{C} = \alpha\mathbf{A}\mathbf{B} + \beta\mathbf{C}$

```
xGEMM(TRANSA,TRANSB,M,N,K,
      ALPHA,A,LDA,X,B,LDB,
      BETA,C,LDC)
```

    (☞ meaning of arguments as above)

Remark 1.3.21 (BLAS calling conventions)

The BLAS calling syntax seems queer in light of modern object oriented programming paradigms, but it is a legacy of FORTRAN77, which was (and partly still is) the programming language, in which the BLAS

routines were coded.

It is a very common situation in scientific computing that one has to rely on old codes and libraries implemented in an old-fashioned style.

Example 1.3.22 (Calling BLAS routines from C/C++)

When calling BLAS library functions from C, all arguments have to be passed by reference (as pointers), in order to comply with the argument passing mechanism of FORTRAN77, which is the model followed by BLAS.

C++-code 1.3.23: BLAS-based SAXPY operation in C++

```
#define daxpy_  daxpy
#include <iostream>

// Definition of the required BLAS function.  This is usually done
// in a header file like blas.h that is included in the EIGEN3
// distribution
extern "C" {
  int daxpy_(const int* n, const double* da, const double* dx,
             const int* incx, double* dy, const int* incy);
}

using namespace std;

int main(){
  const int    n     = 5; // length of vector
  const int    incx  = 1; // stride
  const int    incy  = 1; // stride
  double alpha = 2.5;     // scaling factor

  // Allocated raw arrays of doubles
  double* x  = new double [n];
  double* y  = new double [n];

  for (size_t i=0; i<n; i++){
    x[i] = 3.1415 * i;
    y[i] = 1.0 / (double)(i+1);
  }

  cout << "x=["; for (size_t i=0; i<n; i++) cout << x[i] << ' ';
  cout << "]" << endl;
  cout << "y=["; for (size_t i=0; i<n; i++) cout << y[i] << ' ';
  cout << "]" << endl;

  // Call the BLAS library function passing pointers to all arguments
  // (Necessary when calling FORTRAN routines from C
  daxpy_(&n, &alpha, x, &incx, y, &incy);

```

```
38   cout << "y = " << alpha << " * x + y = ]";
39   for (int i=0; i<n; i++) cout << y[i] << ' '; cout << "]" << endl;
40   return(0);
41 }
```

When using EIGEN in a mode that includes an external BLAS library, all this calls are wrapped into EIGEN methods.

Example 1.3.24 (Using Intel Math Kernel Library (Intel MKL) from EIGEN)

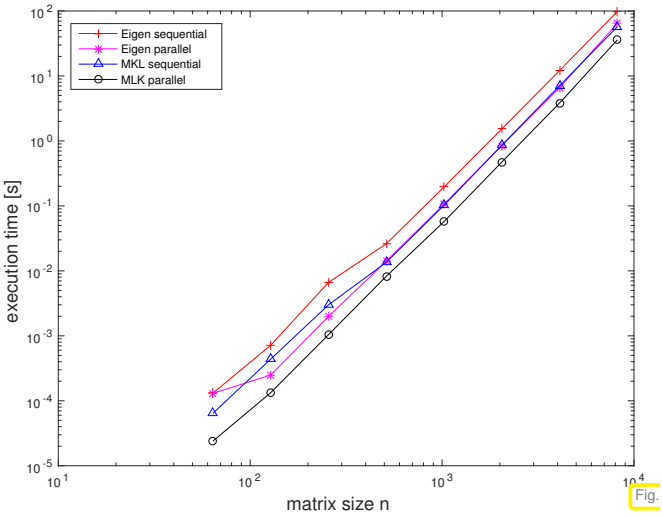
The Intel Math Kernel Library is a highly optimized math library for Intel processors and can be called directly from EIGEN (Using Intel® Math Kernel Library from Eigen) using the correct compiler flags.

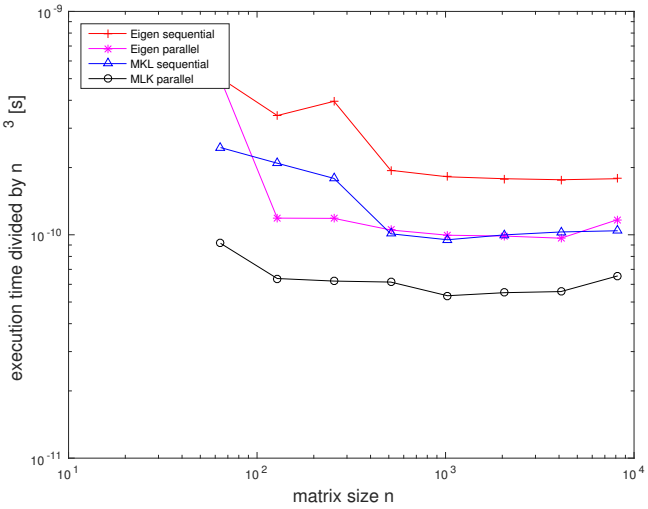
C++-code 1.3.25: Timing of matrix multiplication in EIGEN for MKL comparison ↪ GITLAB

```
2  //!  script for timing different implementations of matrix
       multiplications
3  void mmeigenmkl(){
4    int nruns = 3, minExp = 6, maxExp = 13;
5    MatrixXd timings(maxExp-minExp+1,2);
6    for(int p = 0; p <= maxExp-minExp; ++p){
7      Timer t1; // timer class
8      int n = std::pow(2, minExp + p);
9      MatrixXd A = MatrixXd::Random(n,n);
10     MatrixXd B = MatrixXd::Random(n,n);
11     MatrixXd C = MatrixXd::Zero(n,n);
12     for(int q = 0; q < nruns; ++q){
13           t1.start();
14           C = A * B;
15           t1.stop();
16     }
17     timings(p,0)=n; timings(p,1)=t1.min();
18   }
19   std::cout << std::scientific << std::setprecision(3) << timings <<
       std::endl;
20 }
```

Timing results:

n	EIGEN sequential [s]	EIGEN parallel [s]	MKL sequential [s]	MKL parallel [s]
64	1.318e-04	1.304e-04	6.442e-05	2.401e-05
128	7.168e-04	2.490e-04	4.386e-04	1.336e-04
256	6.641e-03	1.987e-03	3.000e-03	1.041e-03
512	2.609e-02	1.410e-02	1.356e-02	8.243e-03
1024	1.952e-01	1.069e-01	1.020e-01	5.728e-02
2048	1.531e+00	8.477e-01	8.581e-01	4.729e-01
4096	1.212e+01	6.635e+00	7.075e+00	3.827e+00
8192	9.801e+01	6.426e+01	5.731e+01	3.598e+01





Timing environment:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3

1.4 Computational effort

Large scale numerical computations require immense resources and execution time of numerical codes often becomes a central concern. Therefore, much emphasis has to be put on

1. designing algorithms that produce a desired result with (nearly) minimal computational effort (defined precisely below),
2. exploit possibilities for parallel and vectorised execution,
3. organising algorithms in order to make them fit memory hierarchies,
4. implementing codes that make optimal use of hardware resources and capabilities,

While Item 2–Item 4 are out of the scope of this course and will be treated in more advanced lectures, Item 1 will be a recurring theme.

The following definition encapsulates what is regarded as a measure for the "cost" of an algorithm in computational mathematics.

Definition 1.4.1. Computational effort

The computational effort required by a numerical code amounts to the number of elementary operations (additions,subtractions,multiplications,divisions,square roots) executed in a run.

(1.4.2) What computational effort does not tell us

Fifty years ago counting elementary operations provided good predictions of runtimes, but nowadays this is no longer true.

"Computational effort $\not\sim$ runtime"

The computational effort involved in a run of a numerical code is only loosely related to overall execution time on modern computers.

This is conspicuous in Exp. 1.2.29, where algorithms incurring exactly the same computational effort took different times to execute.

The reason is that on today's computers a key bottleneck for fast execution is latency and bandwidth of memory, *cf.* the discussion at the end of Exp. 1.2.29 and [**?**]. Thus, concepts like I/O-complexity [**?**, **?**] might be more appropriate for gauging the efficiency of a code, because they take into account the pattern of memory access.

1.4.1 (Asymptotic) complexity

The concept of computational effort from Def. 1.4.1 is still useful in a particular context:

Definition 1.4.4. (Asymptotic) complexity

The asymptotic complexity of an algorithm characterises the worst-case dependence of its computational effort on one or more problem size parameter(s) when these tend to ∞.

- *Problem size parameters* in numerical linear algebra usually are the lengths and dimensions of the vectors and matrices that an algorithm takes as inputs.
- *Worst case* indicates that the maximum effort over a set of admissible data is taken into account.

When dealing with asymptotic complexities a mathematical formalism comes handy:

Definition 1.4.5. Landau symbol [?, p. 7]

We write $F(n) = O(G(n))$ for two functions $F, G : \mathbb{N} \to \mathbb{R}$, if there exists a constant $C > 0$ and $n_* \in \mathbb{N}$ such that

$$F(n) \leq C\, G(n) \quad \forall n \geq n_* \,.$$

More generally, $F(n_1, \ldots, n_k) = O(G(n_1, \ldots, n_k))$ for two functions $F, G : \mathbb{N}^k \to \mathbb{R}$ implies the existence of a constant $C > 0$ and a threshold value $n_* \in \mathbb{N}$ such that

$$F(n_1, \ldots, n_k) \leq C G(n_1, \ldots, n_k) \quad \forall n_1, \ldots, n_k \in \mathbb{N} \,, \quad n_\ell \geq n_*,\ \ell = 1, \ldots, k \,.$$

Remark 1.4.6 (Meaningful "O-bounds" for complexity)

Of course, the definition of the Landau symbol leaves ample freedom for stating meaningless bounds; an algorithm that runs with linear complexity $O(n)$ can be correctly labelled as possessing $O(\exp(n))$ complexity.

Yet, whenever the Landau notation is used to describe asymptotic complexities, the bounds have to be sharp in the sense that no function with slower asymptotic growth will be possible inside the O. To make this precise we stipulate the following.

Sharpness of a complexity bound

Whenever the asymptotic complexity of an algorithm is stated as $O(n^\alpha \log^\beta n \exp(\gamma n^\delta))$ with non-negative parameters $\alpha, \beta, \gamma, \delta \geq 0$ in terms of the problem size parameter n, we take for granted that choosing a smaller value for any of the parameters will no longer yield a valid (or provable) asymptotic bound.

In particular

- complexity $O(n)$ means that the complexity is not $O(n^\alpha)$ for any $\alpha < 1$,
- complexity $O(\exp(n))$ excludes asymptotic complexity $O(n^p)$ for any $p \in \mathbb{R}$.

Terminology: If the asymptotic complexity of an algorithm is $O(n^p)$ with $p = 1, 2, 3$ we say that it is of "linear", "quadratic", and "cubic" complexity, respectively.

Remark 1.4.8 (Relevance of asymptotic complexity)

§ 1.4.2 warned us that computational effort and, thus, asymptotic complexity, of an algorithm for a concrete problem on a particular platform may not have much to do with the actual runtime (the blame goes to memory hierarchies, internal pipelining, vectorisation, etc.).

Then, why do we pay so much attention to asymptotic complexity in this course?

To a certain extent, the asymptotic complexity allows to predict the *dependence of the runtime* of a particular implementation of an algorithm *on the problem size* (for large problems).

For instance, an algorithm with asymptotic complexity $O(n^2)$ is likely to take $4\times$ as much time when the problem size is doubled.

(1.4.9) Concluding polynomial complexity from runtime measurements

Available: "Measured runtimes" $t_i = t_i(n_i)$ for different values $n_1, n_2, \ldots, n_N$, $n_i \in \mathbb{N}$, of the problem size parameter

Conjectured: power law dependence $t_i \approx C n_i^\alpha$ (also "algebraic dependence"), $\alpha \in \mathbb{R}$

How can we glean evidence that supports or refutes our conjecture from the data? Look at the data in doubly logarithmic scale!

$$t_i = C n_i^\alpha \quad \Rightarrow \quad \log(t_i) \approx \log C + \alpha \log(n_i)\,, \quad i = 1, \ldots, N\,.$$

If the conjecture holds true, then the points (n_i, t_i) will approximately lie on a *straight line* with slope α in a doubly logarithmic plot (which can be created in MATLAB by the **loglog** plotting command and in EIGEN with the **Figure**-command `fig.setlog(true, true);`).

➢ quick "visual test" of conjectured asymptotic complexity

More rigorous: Perform linear regression on $(\log n_i, \log t_i)$, $i = 1, \ldots, N$ ($\rightarrow$ Chapter 3)

1.4.2 Cost of basic operations

Performing elementary BLAS-type operations through simple (nested) loops, we arrive at the following obvious complexity bounds:

operation	description	#mul/div	#add/sub	asymp. complexity
dot product	$(\mathbf{x} \in \mathbb{R}^n, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}^H\mathbf{y}$	n	$n-1$	$O(n)$
tensor product	$(\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n) \mapsto \mathbf{x}\mathbf{y}^H$	nm	0	$O(mn)$
matrix product$^{(*)}$	$(\mathbf{A} \in \mathbb{R}^{m,n}, \mathbf{B} \in \mathbb{R}^{n,k}) \mapsto \mathbf{AB}$	mnk	$mk(n-1)$	$O(mnk)$

Remark 1.4.10 ("Fast" matrix multiplication)

$(*)$: The $O(mnk)$ complexity bound applies to "straightforward" matrix multiplication according to (1.3.1).

For $m = n = k$ there are (sophisticated) variants with better asymptotic complexity, e.g., the divide-and-conquer Strassen algorithm [**?**] with asymptotic complexity $O(n^{\log_2 7})$:

Start from $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{n,n}$ with $n = 2\ell$, $\ell \in \mathbb{N}$. The idea relies on the block matrix product (1.3.16) with $\mathbf{A}_{ij}, \mathbf{B}_{ij} \in \mathbb{K}^{\ell,\ell}$, $i, j \in \{1, 2\}$. Let $\mathbf{C} := \mathbf{AB}$ be partitioned accordingly: $\mathbf{C} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{22} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix}$. Then tedious elementary computations reveal

$$\begin{aligned} \mathbf{C}_{11} &= \mathbf{Q}_0 + \mathbf{Q}_3 - \mathbf{Q}_4 + \mathbf{Q}_6 \,, \\ \mathbf{C}_{21} &= \mathbf{Q}_1 + \mathbf{Q}_3 \,, \\ \mathbf{C}_{12} &= \mathbf{Q}_2 + \mathbf{Q}_4 \,, \\ \mathbf{C}_{22} &= \mathbf{Q}_0 + \mathbf{Q}_2 - \mathbf{Q}_1 + \mathbf{Q}_5 \,, \end{aligned}$$

where the $\mathbf{Q}_k \in \mathbb{K}^{\ell,\ell}$, $k = 1, \ldots, 7$ are obtained from

$$\begin{aligned} \mathbf{Q}_0 &= (\mathbf{A}_{11} + \mathbf{A}_{22}) * (\mathbf{B}_{11} + \mathbf{B}_{22}) \,, \\ \mathbf{Q}_1 &= (\mathbf{A}_{21} + \mathbf{A}_{22}) * \mathbf{B}_{11} \,, \\ \mathbf{Q}_2 &= \mathbf{A}_{11} * (\mathbf{B}_{12} - \mathbf{B}_{22}) \,, \\ \mathbf{Q}_3 &= \mathbf{A}_{22} * (-\mathbf{B}_{11} + \mathbf{B}_{21}) \,, \\ \mathbf{Q}_4 &= (\mathbf{A}_{11} + \mathbf{A}_{12}) * \mathbf{B}_{22} \,, \\ \mathbf{Q}_5 &= (-\mathbf{A}_{11} + \mathbf{A}_{21}) * (\mathbf{B}_{11} + \mathbf{B}_{12}) \,, \\ \mathbf{Q}_6 &= (\mathbf{A}_{12} - \mathbf{A}_{22}) * (\mathbf{B}_{21} + \mathbf{B}_{22}) \,. \end{aligned}$$

Beside a considerable number of matrix additions (computational effort $O(n^2)$) it takes only **7** multiplications of matrices of size $n/2$ to compute $\mathbf{C}$! Strassen's algorithm boils down to the *recursive application* of these formulas for $n = 2^k$, $k \in \mathbb{N}$.

A refined algorithm of this type can achieve complexity $O(n^{2.36})$, see [**?**].

1.4.3 Reducing complexity in numerical linear algebra: Some tricks

In computations involving matrices and vectors complexity of algoritms can often be reduced by performing the operations in a particular order:

Example 1.4.11 (Efficient associative matrix multiplication)

We consider the multiplication with a rank-1-matrix. Matrices with rank 1 can always be obtained as the tensor product of two vectors, that is, the matrix product of a column vector and a row vector. Given $\mathbf{a} \in \mathbb{K}^m$, $\mathbf{b} \in \mathbb{K}^n$, $\mathbf{x} \in \mathbb{K}^n$ we may compute the vector $\mathbf{y} = \mathbf{a}\mathbf{b}^\top\mathbf{x}$ in two ways:

$$\mathbf{y} = \left(\mathbf{a}\mathbf{b}^\top\right)\mathbf{x}\,. \qquad (1.4.12) \qquad\qquad \mathbf{y} = \mathbf{a}\left(\mathbf{b}^\top\mathbf{x}\right)\,. \qquad (1.4.13)$$

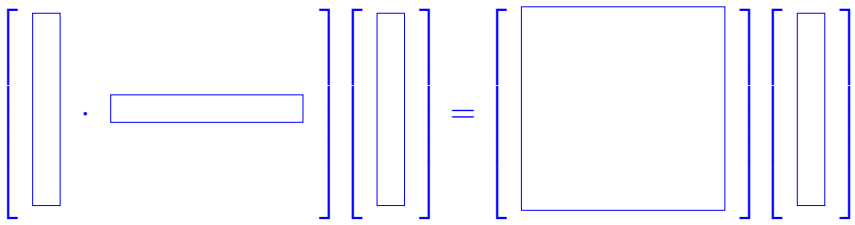
```
T = (a*b.transpose())*x;
```

```
t = a*b.dot(x);
```

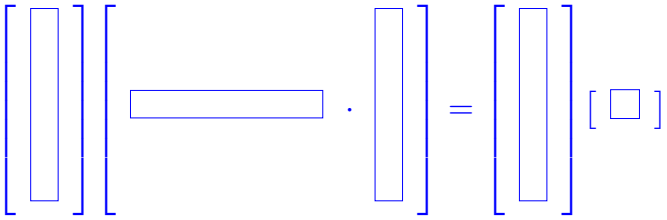
➤ complexity $O(mn)$

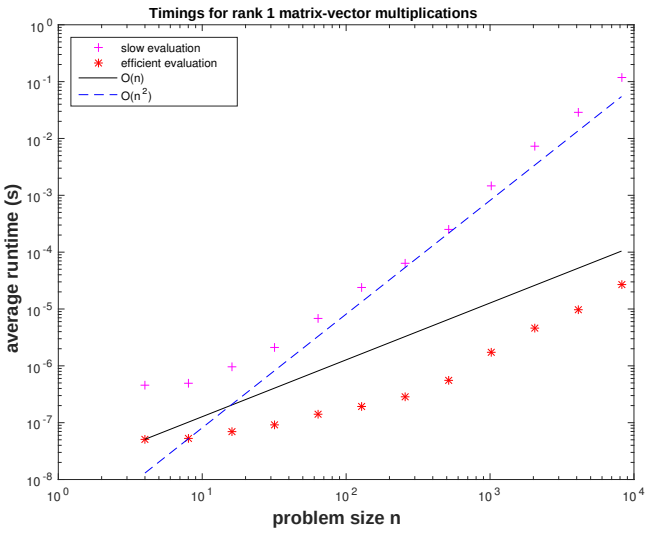
➤ complexity $O(n+m)$ ("linear complexity")

Visualization of evaluation according to (1.4.12):



Visualization of evaluation according to (1.4.13):





◁ average runtimes for efficient/inefficient matrix×vector multiplication with rank-1 matrices , see § 1.4.9 for the rationale behind choosing a *doubly logarithmic plot*.

Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB,
- 8 GB main memory
- gcc 4.8.4, -O3

C++11 code 1.4.14: EIGEN code for Ex. 1.4.11 ↪ GITLAB

```
//!  This function compares the runtimes for the multiplication
//!  of a vector with a rank-1 matrix ab^T, a,b ∈ R^n
//!  using different associative evaluations.
//!  Runtime measurements consider minimal time for
//!  several (nruns) runs
MatrixXd dottenstiming() {
  const int nruns = 3, minExp = 2, maxExp = 13;
  // Matrix for storing recorded runtimes
  MatrixXd timings(maxExp-minExp+1,3);
  for(int i = 0; i <= maxExp-minExp; ++i){
    Timer tfool, tsmart; // Timer objects
    const int n = std::pow(2, minExp + i);
    VectorXd a = VectorXd::LinSpaced(n,1,n);
    VectorXd b = VectorXd::LinSpaced(n,1,n).reverse();
    VectorXd x = VectorXd::Random(n,1), y(n);
    for(int j = 0; j < nruns; ++j){
      // Grossly wasteful evaluation
      tfool.start(); y = (a*b.transpose())*x;      tfool.stop();
      // Efficient implementation
      tsmart.start(); y = a * b.dot(x); tsmart.stop();
    }
    timings(i,0)=n;
    timings(i,1)=tsmart.min(); timings(i,2)=tfool.min();
  }
  return timings;
}
```

Complexity can sometimes be reduced by reusing intermediate results.

Example 1.4.15 (Hidden summation)

The asymptotic complexity of the EIGEN code

```
Eigen::MatrixXd AB = A*B.transpose();
y = AB.triangularView<Eigen::Upper>()*x;
```

when supplied with two low-rank matrices $\mathbf{A},\mathbf{B} \in \mathbb{K}^{n,p}$, $p \ll n$, in terms of $n \to \infty$ obviously is $O(n^2)$, because an intermediate $n \times n$-matrix $\mathbf{A}\mathbf{B}^T$ is built.

First, consider the case of a tensor product (= rank-1) matrix, that is, $p = 1$, $\mathbf{A} \leftrightarrow \mathbf{a} = [a_1,\ldots,a_n]^\top \in \mathbb{K}^n$,

$\mathbf{B} \leftrightarrow \mathbf{b} = [b_1, \ldots, b_n] \in \mathbb{K}^n$. Then

$$\mathbf{y} = \operatorname{triu}(\mathbf{a}\mathbf{b}^T)\mathbf{x} = \begin{bmatrix} a_1b_1 & a_1b_2 & \ldots & & \ldots & a_1b_n \\ 0 & a_2b_2 & a_2b_3 & \ldots & \ldots & a_2b_n \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \ldots & & \ldots & 0 & a_nb_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \\ \vdots \\ x_n \end{bmatrix}$$

$$= \begin{bmatrix} a_1 & & \\ & \ddots & \\ & & a_n \end{bmatrix} \underbrace{\begin{bmatrix} 1 & 1 & \ldots & & \ldots & 1 \\ 0 & 1 & 1 & \ldots & \ldots & 1 \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ 0 & \ldots & & \ldots & 0 & 1 \end{bmatrix}}_{\mathbf{T}} \begin{bmatrix} b_1 & & \\ & \ddots & \\ & & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ \\ \\ \vdots \\ x_n \end{bmatrix}.$$

Thus, the core problem is the fast multiplication of a vector with an upper triangular matrix $\mathbf{T}$ described in EIGEN syntax by `Eigen::MatrixXd::Ones(n,n).triangularView<Eigen::Upper>()`. Note that multiplication of a vector $\mathbf{x}$ with $\mathbf{T}$ yields a vector of partial sums of components of $\mathbf{x}$ starting from last component. This can be achieved by invoking the special C++ command `std::partial_sum` in the numeric header (documentation). We also observe that

$$\mathbf{A}\mathbf{B}^T = \sum_{\ell=1}^{p} (\mathbf{A})_{:,\ell} ((\mathbf{B})_{:,\ell})^\top ,$$

so that that the computations for the special case $p = 1$ discussed above can simply be reused p times!

C++11 code 1.4.16: Efficient multiplication with the upper diagonal part of a rank-p-matrix in EIGEN ⇝ GITLAB

```
2  //!  Computation of y = triu(AB^T)x
3  //!  Efficient implementation with backward cumulative sum
4  //!  (partial_sum)
5  template<class Vec, class Mat>
6  void lrtrimulteff(const Mat& A, const Mat& B, const Vec& x, Vec& y){
7    const int n = A.rows(), p = A.cols();
8    assert( n == B.rows() && p == B.cols()); // size missmatch
9    for(int l = 0; l < p; ++l){
10     Vec tmp = (B.col(l).array() * x.array()).matrix().reverse();
11     std::partial_sum(tmp.data(), tmp.data()+n, tmp.data());
12     y += (A.col(l).array() * tmp.reverse().array()).matrix();
13   }
14 }
```

This code enjoys the obvious complexity of $O(pn)$ for $p, n \to \infty$, $p < n$. The code offers an example of a function templated with its argument types, see § 0.2.5. The types **Vec** and **Mat** must fit the concept of EIGEN vectors/matrices.

The next concept from linear algebra is important in the context of computing with multi-dimensional arrays.

Definition 1.4.17. Kronecker product

The Kronecker product $\mathbf{A} \otimes \mathbf{B}$ of two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{l,k}$, $m,n,l,k \in \mathbb{N}$, is the $(ml) \times (nk)$-matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{11}\mathbf{B} & (\mathbf{A})_{1,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{ml,nk} .$$

Example 1.4.18 (Multiplication of Kronecker product with vector)

The function $(\mathbf{A} \otimes \mathbf{B})\mathbf{x}$ when invoked with two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{l,k}$ and a vector $\mathbf{x} \in \mathbb{K}^{nk}$, will suffer an asymptotic complexity of $O(m \cdot n \cdot l \cdot k)$, determined by the size of the intermediate dense matrix $\mathbf{A} \otimes \mathbf{B} \in \mathbb{K}^{ml,nk}$.

Using the partitioning of the vector $\mathbf{x}$ into n equally long sub-vectors

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}^1 \\ \mathbf{x}^2 \\ \vdots \\ \mathbf{x}^n \end{bmatrix} \quad , \quad \mathbf{x}^j \in \mathbb{K}^k ,$$

we find the representation

$$(\mathbf{A} \otimes \mathbf{B})\mathbf{x} = \begin{bmatrix} (\mathbf{A})_{1,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{1,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{1,n}\mathbf{B}\mathbf{x}^n \\ (\mathbf{A})_{2,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{2,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{2,n}\mathbf{B}\mathbf{x}^n \\ \vdots \\ \vdots \\ (\mathbf{A})_{m,1}\mathbf{B}\mathbf{x}^1 + (\mathbf{A})_{m,2}\mathbf{B}\mathbf{x}^2 + \cdots + (\mathbf{A})_{m,n}\mathbf{B}\mathbf{x}^n \end{bmatrix} .$$

The idea is to form the products $\mathbf{B}\mathbf{x}^j$, $j = 1, \dots, n$, once, and then combine them linearly with coefficients given by the entries in the rows of $\mathbf{A}$:

C++11 code 1.4.19: Efficient multiplication of Kronecker product with vector in EIGEN ➙ GITLAB

```
2 //!  @brief Multiplication of Kronecker product with vector y = (A⊗B)x.
     Elegant way using reshape
3 //!  WARNING: using Matrix::Map we assume the matrix is in ColMajor
     format, *beware* you may incur bugs if matrix is in RowMajor isntead
4 //!  @param[in] A Matrix m × n
5 //!  @param[in] B Matrix l × k
6 //!  @param[in] x Vector of dim nk
7 //!  @param[out] y Vector y = kron(A,B)*x of dim ml
8 template <class Matrix, class Vector>
```

```
9  void kronmultv(const Matrix &A, const Matrix &B, const Vector &x,
       Vector &y){
10   unsigned int m = A.rows(); unsigned int n = A.cols();
11   unsigned int l = B.rows(); unsigned int k = B.cols();
12   // 1st matrix mult.  computes the products Bx^j
13   // 2nd matrix mult.  combines them linearly with the coefficients
         of A
14   Matrix t = B * Matrix::Map(x.data(),k,n) * A.transpose(); //
15   y = Matrix::Map(t.data(), m*l, 1);
16 }
```

Recall the reshaping of a matrix in EIGEN in order to understand this code: Rem. 1.2.27.

The asymptotic complexity of this code is determined by the two matrix multiplications in Line 14. This yields the asymptotic complexity $O(lkn + mnl)$ for $l, k, m, n \to \infty$.

PYTHON-code 1.4.20: Efficient multiplication of Kronecker product with vector in PYTHON

```
1 def kronmultv(A, B, x):
2     n, k = A.shape[1], B.shape[1]
3     assert x.size == n * k, 'size mismatch'
4     xx = np.reshape(x, (n, k))
5     Z = np.dot(xx, B.T)
6     yy = np.dot(A, Z)
7     return np.ravel(yy)
```

Note that different reshaping is used in the PYTHON code due to the default row major storage order.

1.5 Machine Arithmetic

1.5.1 Experiment: Loss of orthogonality

(1.5.1) Gram-Schmidt orthogonalisation

From linear algebra [**?**, Sect. 4.4] or Ex. 0.2.41 we recall the fundamental algorithm of Gram-Schmidt orthogonalisation of an ordered finite set $\{\mathbf{a}^1, \ldots, \mathbf{a}^k\}$, $k \in \mathbb{N}$, of vectors $\mathbf{a}^\ell \in \mathbb{K}^n$:

Input: $\{\mathbf{a}^1,\ldots,\mathbf{a}^k\} \subset \mathbb{K}^n$

1: $\mathbf{q}^1 := \frac{\mathbf{a}^1}{\|\mathbf{a}^1\|_2}$ *% 1st output vector*
2: **for** $j = 2,\ldots,k$ **do**
{ *% Orthogonal projection*
3: $\mathbf{q}^j := \mathbf{a}^j$
4: **for** $\ell = 1,2,\ldots,j-1$ **do** (GS)
5: { $\mathbf{q}^j \leftarrow \mathbf{q}^j - \mathbf{a}^j \cdot \mathbf{q}^\ell \mathbf{q}^\ell$ }
6: **if** ($\mathbf{q}^j = \mathbf{0}$) **then** STOP
7: **else** { $\mathbf{q}^j \leftarrow \frac{\mathbf{q}^j}{\|\mathbf{q}^j\|_2}$ }
8: }

Output: $\{\mathbf{q}^1,\ldots,\mathbf{q}^j\}$

In linear algebra we have learnt that, if it does not STOP prematurely, this algorithm will compute orthonormal vectors $\mathbf{q}^1,\ldots,\mathbf{q}^k$ satisfying

$$\mathrm{Span}\{\mathbf{q}^1,\ldots,\mathbf{q}^\ell\} = \mathrm{Span}\{\mathbf{a}^1,\ldots,\mathbf{a}^\ell\}\,, \tag{1.5.2}$$

for all $\ell \in \{1,\ldots,k\}$.

More precisely, if $\mathbf{a}^1,\ldots,\mathbf{a}^\ell$, $\ell \le k$, are linearly independent, then the Gram-Schmidt algorithm will not terminate before the $\ell+1$-th step.

✎ Notation: $\|\cdot\|_2 \hat{=}$ Euclidean norm of a vector $\in \mathbb{K}^n$

The following code implements the Gram-Schmidt orthonormalization of a set of vectors passed as the columns of a matrix $\mathbf{A} \in \mathbb{R}^{n,k}$.

C++11 code 1.5.3: Gram-Schmidt orthogonalisation in EIGEN ↪ GITLAB

```
2  template <class Matrix>
3  Matrix gramschmidt( const Matrix & A ) {
4      Matrix Q = A;
5      // First vector just gets normalized, Line 1 of (GS)
6      Q.col(0).normalize();
7      for(unsigned int j = 1; j < A.cols(); ++j) {
8          // Replace inner loop over each previous vector in Q with fast
9          // matrix-vector multiplication (Lines 4, 5 of (GS))
10         Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).transpose() *
               A.col(j));//
11         // Normalize vector, if possible.
12         // Otherwise colums of A must have been linearly dependent
13         if( Q.col(j).norm() <= 10e-9 * A.col(j).norm() ) { //
14             std::cerr << "Gram-Schmidt failed: A has lin. dep
                   columns." << std::endl;
15             break;
16         } else { Q.col(j).normalize(); } // Line 7 of (GS)
17     }
18     return Q;
19 }
```

We will soon learn the rationale behind the odd test in Line 13..

PYTHON-code 1.5.4: Gram-Schmidt orthogonalisation in PYTHON

```
1  def gramschmidt(A):
2      _, k = A.shape
3      Q = A[:, [0]] / np.linalg.norm(A[:, 0])
4      for j in range(1, k):
5          q = A[:, j] - np.dot(Q, np.dot(Q.T, A[:, j]))
```

```
        nq = np.linalg.norm(q)
        if nq < 1e-9 * np.linalg.norm(A[:, j]):
            break
        Q = np.column_stack([Q, q / nq])
    return Q
```

Note the different loop range due to the zero-based indexing in PYTHON.

Experiment 1.5.5 (Unstable Gram-Schmidt orthonormalization)

If $\{\mathbf{a}^1,\ldots,\mathbf{a}^k\}$ are linearly independent we expect the output vectors $\mathbf{q}^1,\ldots,\mathbf{q}^k$ to be orthonormal:

$$(\mathbf{q}^\ell)^\top \mathbf{q}^m = \delta_{\ell,m}\,, \quad \ell,m \in \{1,\ldots,k\}\,. \tag{1.5.6}$$

This property can be easily tested numerically, for instance by computing $\mathbf{Q}^\top\mathbf{Q}$ for a matrix $\mathbf{Q} = [\mathbf{a}^1,\ldots,\mathbf{q}^k] \in \mathbb{R}^{n,k}$.

C++11 code 1.5.7: Wrong result from Gram-Schmidt orthogonalisation EIGEN ⇝ GITLAB

```
void gsroundoff(MatrixXd& A){
  // Gram-Schmidt orthogonalization of columns of A, see Code 1.5.3
  MatrixXd Q = gramschmidt(A);
  // Test orthonormality of columns of Q, which should be an
  // orthogonal matrix according to theory
  cout << setprecision(4) << fixed << "I = "
       << endl << Q.transpose()*Q << endl;
  // EIGEN's stable internal Gram-Schmidt orthogonalization by
  // QR-decomposition, see Rem. 1.5.9 below
  HouseholderQR<MatrixXd> qr(A.rows(),A.cols()); //
  qr.compute(A); MatrixXd Q1 = qr.householderQ(); //
  // Test orthonormality
  cout << "I1 = " << endl << Q1.transpose()*Q1 << endl;
  // Check orthonormality and span property (1.5.2)
  MatrixXd R1 = qr.matrixQR().triangularView<Upper>();
  cout << scientific << "A-Q1*R1 = " << endl << A-Q1*R1 << endl;
}
```

We test the orthonormality of the output vectors of Gram-Schmidt orthogonalization for a special matrix $\mathbf{A} \in \mathbb{R}^{10,10}$, a so-called Hilbert matrix, defined by $(\mathbf{A})_{i,j} = (i+j-1)^{-1}$. Then Code 1.5.7 produces the follwing output:

```
I =
   1.0000    0.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
   0.0000    1.0000   -0.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
  -0.0000   -0.0000    1.0000    0.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
   0.0000    0.0000    0.0000    1.0000   -0.0000    0.0000   -0.0000   -0.0000   -0.0000   -0.0000
  -0.0000   -0.0000   -0.0000   -0.0000    1.0000    0.0000   -0.0008   -0.0007   -0.0007   -0.0006
   0.0000    0.0000    0.0000    0.0000    0.0000    1.0000   -0.0540   -0.0430   -0.0360   -0.0289
  -0.0000   -0.0000   -0.0000   -0.0000   -0.0008   -0.0540    1.0000    0.9999    0.9998    0.9996
  -0.0000   -0.0000   -0.0000   -0.0000   -0.0007   -0.0430    0.9999    1.0000    1.0000    0.9999
  -0.0000   -0.0000   -0.0000   -0.0000   -0.0007   -0.0360    0.9998    1.0000    1.0000    1.0000
  -0.0000   -0.0000   -0.0000   -0.0000   -0.0006   -0.0289    0.9996    0.9999    1.0000    1.0000
```

Obviously, the vectors produced by the function `gramschmidt` fail to be orthonormal, contrary to the predictions of rigorous results from linear algebra!

However, Line 11, Line 12 of Code 1.5.7 demonstrate another way to orthonormalize the columns of a matrix using EIGEN's built-in class template **HouseholderQR**:

```
I1 =
 1.0000 -0.0000  0.0000 -0.0000 -0.0000 -0.0000 -0.0000 -0.0000 -0.0000  0.0000
-0.0000  1.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000 -0.0000  0.0000 -0.0000
 0.0000 -0.0000  1.0000 -0.0000 -0.0000  0.0000  0.0000  0.0000  0.0000  0.0000
-0.0000  0.0000 -0.0000  1.0000  0.0000 -0.0000 -0.0000  0.0000  0.0000  0.0000
-0.0000  0.0000 -0.0000  0.0000  1.0000 -0.0000  0.0000 -0.0000 -0.0000  0.0000
-0.0000 -0.0000  0.0000 -0.0000 -0.0000  1.0000 -0.0000 -0.0000  0.0000 -0.0000
-0.0000 -0.0000  0.0000 -0.0000  0.0000 -0.0000  1.0000  0.0000  0.0000 -0.0000
-0.0000 -0.0000  0.0000  0.0000 -0.0000 -0.0000  0.0000  1.0000 -0.0000  0.0000
-0.0000  0.0000  0.0000  0.0000 -0.0000  0.0000  0.0000 -0.0000  1.0000 -0.0000
 0.0000 -0.0000  0.0000  0.0000  0.0000 -0.0000 -0.0000  0.0000 -0.0000  1.0000
```

Now we observe apparently perfect orthogonality (1.5.6) of the columns of the matrix `Q1` in Code 1.5.7. There is another algorithm that reliably yields the theoretical output of Gram-Schmidt orthogonalization.

"Computers cannot compute"

Computers cannot compute "properly" in $\mathbb{R}$: numerical computations may not respect the laws of analysis and linear algebra!

This introduces an important new aspect in the study of numerical algorithms.

Remark 1.5.9 (Stable orthonormalization by QR-decomposition)

In Code 1.5.7 we saw the use of the EIGEN class `HousholderQR<MatrixType>` for the purpose of Gram-Schmidt orthogonalisation. The underlying theory and algorithms will be explained later in Section 3.3.3. There we will have the following insight:

➢ Up to signs the columns of the matrix $\mathbf{Q}$ available from the QR-decomposition of $\mathbf{A}$ are the same vectors as produced by the Gram-Schmidt orthogonalisation of the columns of $\mathbf{A}$.

! Code 1.5.7 demonstrates a case where a desired result can be obtained by two *algebraically equivalent* computations, that is, they yield the same result in a mathematical sense. Yet, when implemented on a computer, the results can be *vastly different*. One algorithm may produce junk ("unstable algorithm"), whereas the other lives up to the expectations ("stable algorithm")

Supplement to Exp. 1.5.5: despite its ability to produce orthonormal vectors, we get as output for `D=A-Q1*R1` in Code 1.5.7:

```
D =
 2.2204e-16  3.3307e-16  3.3307e-16  1.9429e-16  1.9429e-16  5.5511e-17  1.3878e-16  6.9389e-17  8.3267e-17  9.7145e-17
 0.0000e+00  1.1102e-16  8.3267e-17  5.5511e-17  0.0000e+00  5.5511e-17 -2.7756e-17  0.0000e+00  0.0000e+00  4.1633e-17
-5.5511e-17  5.5511e-17  2.7756e-17  5.5511e-17  0.0000e+00  0.0000e+00  0.0000e+00 -1.3878e-17  1.3878e-17  1.3878e-17
 0.0000e+00  5.5511e-17  2.7756e-17  2.7756e-17  0.0000e+00  1.3878e-17 -1.3878e-17  0.0000e+00  1.3878e-17  2.7756e-17
 0.0000e+00  2.7756e-17  0.0000e+00  1.3878e-17  1.3878e-17  1.3878e-17  0.0000e+00  1.3878e-17  1.3878e-17  4.1633e-17
-2.7756e-17  2.7756e-17  1.3878e-17  4.1633e-17  2.7756e-17  1.3878e-17  0.0000e+00 -1.3878e-17  2.7756e-17  2.7756e-17
 0.0000e+00  2.7756e-17  0.0000e+00  2.7756e-17  2.7756e-17  1.3878e-17  0.0000e+00  1.3878e-17  2.7756e-17  2.0817e-17
 0.0000e+00  2.7756e-17  2.7756e-17  1.3878e-17  1.3878e-17  1.3878e-17  0.0000e+00  1.3878e-17  2.0817e-17  2.7756e-17
 1.3878e-17  1.3878e-17  1.3878e-17  2.7756e-17  1.3878e-17  0.0000e+00 -1.3878e-17  6.9389e-18 -6.9389e-18  1.3878e-17
 0.0000e+00  2.7756e-17  1.3878e-17  1.3878e-17  1.3878e-17  0.0000e+00  0.0000e+00  0.0000e+00  1.3878e-17  1.3878e-17
```

➥ The computed QR-decomposition apparently fails to meet the exact algebraic requirements stipulated by Thm. 3.3.9. However, note the tiny size of the "defect".

Remark 1.5.10 (QR-decomposition in EIGEN)

The two different QR-decompositions (3.3.3.1) and (3.3.3.1) of a matrix $\mathbf{A} \in \mathbb{K}^{n,k}$, $n,k \in \mathbb{N}$, can be computed in EIGEN as follows:

```
HouseholderQR<MatrixXd> qr(A);
// Full QR-decomposition (3.3.3.1)
Q = qr.householderQ();
// Economical QR-decomposition (3.3.3.1)
thinQ = qr.householderQ() *
   MatrixXd::Identity(A.rows(),std::min(A.rows(), A.cols()));
```

The returned matrices `Q` and `R` correspond to the QR-factors $\mathbf{Q}$ and $\mathbf{R}$ as defined above. See the discusssion, whether the "economical" decomposition is truly economical: some expression template magic.

Remark 1.5.11 (QR-decomposition in PYTHON)

The two different QR-decomposition (3.3.3.1) and (3.3.3.1) of a matrix $\mathbf{A} \in \mathbb{K}^{n,k}$, $n,k \in \mathbb{N}$, can be computed in PYTHON as follows:

```
1 Q, R = np.linalg.qr(A, mode='reduced')   # Economical (3.3.3.1)
2 Q, R = np.linalg.qr(A, mode='complete')  # Full (3.3.3.1)
```

The returned matrices `Q` and `R` correspond to the QR-factors $\mathbf{Q}$ and $\mathbf{R}$ as defined above.

1.5.2 Machine Numbers

(1.5.12) The finite and discrete set of machine numbers

The reason, why computers must fail to execute exact computations with real numbers is clear:

Computer = finite automaton ➢ can handle only *finitely many* numbers, not $\mathbb{R}$

machine numbers, set $\mathbb{M}$

Essential property: $\mathbb{M}$ is a finite, discrete subset of $\mathbb{R}$ (its numbers separated by gaps)

The set of machine numbers $\mathbb{M}$ cannot be closed under elementary arithmetic operations $+,-,\cdot,/$, that is, when adding, multiplying, etc., two machine numbers the result may not belong to $\mathbb{M}$.

The results of elementary operations with operands in $\mathbb{M}$ have to be mapped back to $\mathbb{M}$, an operation called rounding.

➤ roundoff errors (*ger.:* Rundungsfehler) are inevitable

The impact of roundoff means that mathematical identities may not carry over to the computational realm. As we have seen above in Exp. 1.5.5

Computers cannot compute "properly" !

numerical computations $\neq$ **analysis linear algebra**

This introduces a *new* and *important* aspect in the study of numerical algorithms!

(1.5.13) Internal representation of machine numbers

Now we give a brief sketch of the internal structure of machine numbers $\in \mathbb{M}$. The main insight will be that

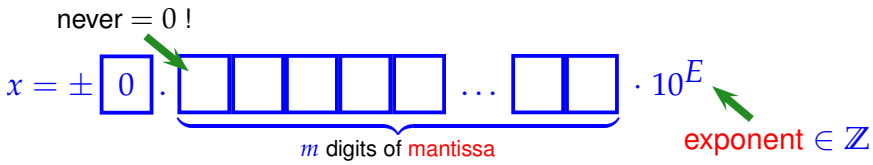
"Computers use floating point numbers (scientific notation)"

Example 1.5.14 (Decimal floating point numbers)

Some 3-digit normalized decimal floating point numbers:

valid: $0.723 \cdot 10^2$, $0.100 \cdot 10^{-20}$, $-0.801 \cdot 10^5$
invalid: $0.033 \cdot 10^2$, $1.333 \cdot 10^{-4}$, $-0.002 \cdot 10^3$

General form of an m-digit normalized decimal floating point number:



Of course, computers are restricted to a *finite range* of exponents:

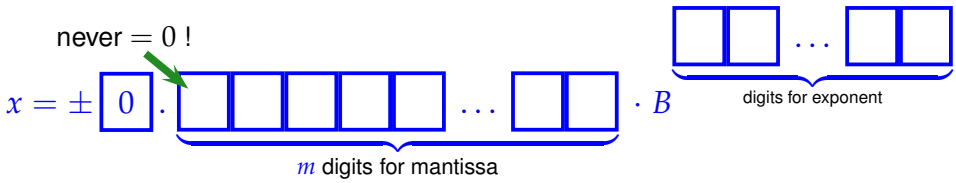
Definition 1.5.15. Machine numbers/floating point numbers → [?, Sect. 2.1]

Given
- ☞ basis $B \in \mathbb{N} \setminus \{1\}$,
- ☞ exponent range $\{e_{\min}, \ldots, e_{\max}\}$, $e_{\min}, e_{\max} \in \mathbb{Z}$, $e_{\min} < e_{\max}$,
- ☞ number $m \in \mathbb{N}$ of digits (for mantissa),

the corresponding set of machine numbers is

$$\mathbb{M} := \{d \cdot B^E : d = i \cdot B^{-m}, i = B^{m-1}, \ldots, B^m - 1, E \in \{e_{\min}, \ldots, e_{\max}\}\}$$

machine number $\in \mathbb{M}$:



Remark 1.5.16 (Extremal numbers in $\mathbb{M}$)

Clearly, there is a largest element of $\mathbb{M}$ and two that are closest to zero. These are mainly determined by the range for the exponent E, *cf.* Def. 1.5.15.

Largest machine number (in modulus) : $x_{\max} = \max|\mathbb{M}| = (1 - B^{-m}) \cdot B^{e_{\max}}$
Smallest machine number (in modulus) : $x_{\min} = \min|\mathbb{M}| = B^{-1} \cdot B^{e_{\min}}$

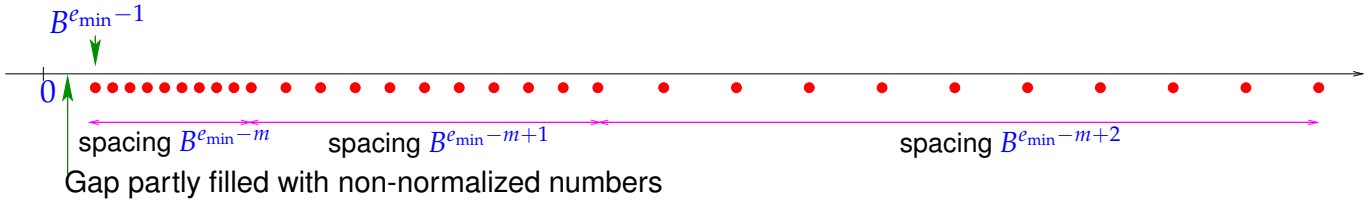
In C++ these extremal machine numbers are accessible through the

```
std::numeric_limits<double>::max()
```
and `std::numeric_limits<double>::min()`

functions. Other properties of arithmetic types can be queried accordingly from the numeric_limits header.

Remark 1.5.17 (Distribution of machine numbers)

From Def. 1.5.15 it is clear that there are equi-spaced sections of $\mathbb{M}$ and that the gaps between machine numbers are bigger for larger numbers:



Non-normalized numbers violate the lower bound for the mantissa i in Def. 1.5.15.

(1.5.18) IEEE standard 754 for machine numbers → [?], [?, Sect. 2.4], → link

No surprise: for modern computers $B = 2$ (binary system), the other parameters of the universally implemented machine number system are

single precision : $m = 24^*, E \in \{-125, \ldots, 128\}$ ➢ 4 bytes

double precision : $m = 53^*, E \in \{-1021, \ldots, 1024\}$ ➢ 8 bytes

$*$: including bit indicating sign

The standardisation of machine numbers is important, because it ensures that the same numerical algorithm, executed on different computers will nevertheless produce the same result.

Remark 1.5.19 (Special cases in IEEE standard)

```
double x = exp(1000), y = 3/x, z = x*sin(M_PI), w = x*log(1);
cout << x << endl << y << endl << z << endl << w << endl;
```

Output:

```
1 inf
2 0
3 inf
4 -nan
```

$E = e_{\max}$, $M \neq 0$ $\hat{=}$ NaN = *Not a number* $\rightarrow$ exception
$E = e_{\max}$, $M = 0$ $\hat{=}$ Inf = *Infinity* $\rightarrow$ overflow
$E = 0$ $\hat{=}$ Non-normalized numbers $\rightarrow$ underflow
$E = 0$, $M = 0$ $\hat{=}$ number 0

(1.5.20) Characteristic parameters of IEEE floating point numbers (double precision)

☞ C++ does not always fulfill the requirements of the IEEE 754 standard and it needs to be checked with `std::numeric_limits<T>::is_iec559`.

C++11-code 1.5.21: Querying characteristics of `double` numbers ⇢ GITLAB

```
2  #include <limits>
3  #include <iostream>
4  #include <iomanip>
5
6  using namespace std;
7
8  int main(){
9      cout << std::numeric_limits<double>::is_iec559 << endl
10     << std::defaultfloat << numeric_limits<double>::min() << endl
11     << std::hexfloat << numeric_limits<double>::min() << endl
12     << std::defaultfloat << numeric_limits<double>::max() << endl
13     << std::hexfloat << numeric_limits<double>::max() << endl;
14 }
```

Output:

```
true
2.22507e-308
0010000000000000
1.79769e+308
7feffffffffffff
```

1.5.3 Roundoff errors

Experiment 1.5.22 (Input errors and roundoff errors)

The following computations would always result in 0, if done in exact arithmetic.

C++11-code 1.5.23: Demonstration of roundoff errors ↪ GITLAB

```
#include <iostream>
int main(){
      std::cout.precision(15);
      double a = 4.0/3.0, b = a-1, c = 3*b, e = 1-c;
      std::cout << e << std::endl;
      a = 1012.0/113.0; b = a-9; c = 113*b; e = 5+c;
      std::cout << e << std::endl;
      a = 83810206.0/6789.0; b = a-12345; c = 6789*b; e = c-1;
      std::cout << e << std::endl;
}
```

Output:

```
2.22044604925031e-16
6.75015598972095e-14
-1.60798663273454e-09
```

Can you devise a similar calculation, whose result is even farther off zero? Apparently the rounding that inevitably accompanies arithmetic operations in $\mathbb{M}$ can lead to results that are far away from the true result.

For the discussion of errors introduced by rounding we need important notions.

Definition 1.5.24. Absolute and relative error → [?, Sect. 1.2]

Let $\widetilde{x} \in \mathbb{K}$ be an approximation of $x \in \mathbb{K}$. Then its absolute error is given by

$$\epsilon_{\text{abs}} := |x - \widetilde{x}| ,$$

and its relative error is defined as

$$\epsilon_{\text{rel}} := \frac{|x - \widetilde{x}|}{|x|} .$$

Remark 1.5.25 (Relative error and number of correct digits)

The number of correct (significant, valid) digits of an approximation $\widetilde{x}$ of $x \in \mathbb{K}$ is defined through the relative error:

$$\text{If } \epsilon_{\text{rel}} := \frac{|x-\widetilde{x}|}{|x|} \leq 10^{-\ell}, \text{ then } \widetilde{x} \text{ has } \ell \text{ correct digits, } \ell \in \mathbb{N}_0$$

(1.5.26) Floating point operations

We may think of the elementary binary operations $+, -, *, /$ in $\mathbb{M}$ comprising two steps:

❶ Compute the exact result of the operation.

❷ Perform rounding of the result of ❶ to map it back to $\mathbb{M}$.

Definition 1.5.27. Correct rounding

Correct rounding ("rounding up") is given by the function

$$\text{rd} : \begin{cases} \mathbb{R} & \to & \mathbb{M} \\ x & \mapsto & \max \operatorname{argmin}_{\widetilde{x}\in\mathbb{M}} |x - \widetilde{x}| \, . \end{cases}$$

(Recall that $\operatorname{argmin}_x F(x)$ is the set of arguments of a real valued function F that makes it attain its (global) minimum.)

Of course, ❶ above is not possible in a strict sense, but the effect of both steps can be realised and yields a floating point realization of $\star \in \{+, -, \cdot, /\}$.

✎ Notation: write $\widetilde{\star}$ for the floating point realization of $\star \in \{+, -, \cdot, /\}$:

Then ❶ and ❷ may be summed up into

$$\text{For } \star \in \{+, -, \cdot, /\}: \quad x \,\widetilde{\star}\, y := \text{rd}(x \star y) \, .$$

Remark 1.5.28 (Breakdown of associativity)

As a consequence of rounding addition $\widetilde{+}$ and multiplication $\widetilde{*}$ as implemented on computers fail to be associative. They will usually be commutative, though this is not guaranteed.

(1.5.29) Estimating roundoff errors → **[?, p. 23]**

Let us denote by EPS the largest relative error (→ Def. 1.5.24) incurred through rounding:

$$\text{EPS} := \max_{x\in I\setminus\{0\}} \frac{|\,\text{rd}(x) - x|}{|x|} \, , \tag{1.5.30}$$

where $I = [\min|\mathbb{M}|, \max|\mathbb{M}|$ is the range of positive machine numbers.

For machine numbers according to Def. 1.5.15 EPS can be computed from the defining parameters B (base) and m (length of mantissa) [**?**, p. 24]:

$$\text{EPS} = \tfrac{1}{2}B^{1-m} \,. \tag{1.5.31}$$

However, when studying roundoff errors, we do not want to delve into the intricacies of the internal representation of machine numbers. This can be avoided by just using a single bound for the relative error due to rounding, and, thus, also for the relative error potentially suffered in each elementary operation.

Assumption 1.5.32. "Axiom" of roundoff analysis

There is a small positive number EPS, the machine precision, such that for the elementary arithmetic operations $\star \in \{+,-,\cdot,/\}$ and "hard-wired" functions* $f \in \{\exp, \sin, \cos, \log, \ldots\}$ holds

$$x \,\tilde{\star}\, y = (x \star y)(1+\delta) \quad , \quad \tilde{f}(x) = f(x)(1+\delta) \quad \forall x, y \in \mathbb{M} \,,$$

with $|\delta| < \text{EPS}$.

*: this is an ideal, which may not be accomplished even by modern CPUs.

➤ relative roundoff errors of elementary steps in a program bounded by machine precision **!**

Example 1.5.33 (Machine precision for IEEE standard)

C++ tells the machine precision as following:

C++11-code 1.5.34: Finding out EPS in C++ ➙ GITLAB

```
#include <iostream>
#include <limits> // get various properties of arithmetic types
int main() {
  std::cout.precision(15);
  std::cout << std::numeric_limits<double>::epsilon() << std::endl;
}
```

Output:

```
2.22044604925031e-16
```

Knowing the machine precision can be important for checking the validity of computations or coding termination conditions for iterative approximations.

Experiment 1.5.35 (Adding EPS to 1)

```
cout.precision(25);
double eps =
  numeric_limits<double>::epsilon();
cout << fixed << 1.0 + 0.5*eps << endl
     << 1.0 - 0.5*eps << endl
     << (1.0 + 2/eps) - 2/eps << endl;
```

Output:

```
1.0000000000000000000000000
0.9999999999999998889776975
0.0000000000000000000000000
```

In fact, the following "definition" of EPS is sometimes used:

EPS is the smallest positive number $\in \mathbb{M}$ for which $1\widetilde{+}\mathrm{EPS} \neq 1$ (in $\mathbb{M}$):

We find that $1\widetilde{+}\mathrm{EPS} = 1$ actually complies with the "axiom" of roundoff error analysis, Ass. 1.5.32:

$$1 = (1+\mathrm{EPS})(1+\delta) \quad \Rightarrow \quad |\delta| = \left|\frac{\mathrm{EPS}}{1+\mathrm{EPS}}\right| < \mathrm{EPS}\,,$$

$$\frac{2}{\mathrm{EPS}} = (1+\frac{2}{\mathrm{EPS}})(1+\delta) \quad \Rightarrow \quad |\delta| = \left|\frac{\mathrm{EPS}}{2+\mathrm{EPS}}\right| < \mathrm{EPS}\,.$$

Do we have to worry about these tiny roundoff errors **?**

YES (→ Exp. 1.5.5):

- accumulation of roundoff errors
- amplification of roundoff errors

Remark 1.5.36 (Testing equality with zero)

Since results of numerical computations are almost always polluted by roundoff errors: Tests like `if (x == 0)` are pointless and even dangerous, if x contains the result of a numerical computation.

Remedy: Test `if (abs(x) < eps*s) ...`,
$s \triangleq$ positive number, compared to which $|x|$ should be small.

We saw a first example of this practise in Code 1.5.3, Line 13.

Remark 1.5.37 (Overflow and underflow)

overflow) $\triangleq$ |result of an elementary operation| $> \max\{\mathbb{M}\}$
$\triangleq$ IEEE standard $\Rightarrow$ `Inf`
underflow $\triangleq$ $0 <$ |result of an elementary operation| $< \min\{|\mathbb{M}\setminus\{0\}|\}$
$\triangleq$ IEEE standard $\Rightarrow$ use non-normalized numbers (!)

The Axiom of roundoff analysis Ass. 1.5.32 does *not hold* once non-normalized numbers are encountered:

C++11-code 1.5.38: Demonstration of over-/underflow ↪ GITLAB

```
#include <iostream>
#define _USE_MATH_DEFINES
#include <cmath>
```

```
#include <limits>
using namespace std;
int main(){
      cout.precision(15);
      double min = numeric_limits<double>::min();
      double res1 = M_PI*min/123456789101112;
      double res2 = res1*123456789101112/min;
      cout << res1 << endl << res2 << endl;
}
```

Output:

```
5.68175492717434e-322
3.15248510554597
```

Try to avoid underflow and overflow

A simple example teaching how to avoid overflow during the computation of the norm of a 2D vector [**?**, Ex. 2.9]:

$$r = \sqrt{x^2 + y^2}$$

straightforward evaluation: overflow, when $|x| > \sqrt{\max|\mathbb{M}|}$ or $|y| > \sqrt{\max|\mathbb{M}|}$.

$$r = \begin{cases} |x|\sqrt{1+(y/x)^2} & \text{, if } |x| \geq |y| \text{ ,} \\ |y|\sqrt{1+(x/y)^2} & \text{, if } |y| > |x| \text{ .} \end{cases}$$

➢ no overflow!

1.5.4 Cancellation

In general, predicting the impact of roundoff errors on the result of a multi-stage computation is very difficult, if possible at all. However, there is a constellation that is particularly prone to dangerous amplification of roundoff and still can be detected easily.

Example 1.5.39 (Computing the zeros of a quadratic polynomial)

The following simple EIGEN code computes the real roots of a quadratic polynomial $p(\xi) = \xi^2 + \alpha\xi + \beta$ by the discriminant formula

$$p(\xi_1) = p(\xi_2) = 0\,, \quad \xi_{1/2} = \frac{1}{2}\left(\alpha \pm \sqrt{D}\right) \text{, if} \quad D := \alpha^2 - 4\beta \geq 0\,. \tag{1.5.40}$$

C++11-code 1.5.41: Discriminant formula for the real roots of $p(\xi) = \xi^2 + \alpha\xi + \beta$ ↪ GITLAB

```
//!  C++ function computing the zeros of a quadratic polynomial
//!  ξ → ξ^2 + αξ + β by means of the familiar discriminant
//!  formula ξ_{1,2} = ½(−α ± √(α^2 − 4β)).  However
//!  this implementation is vulnerable to round-off!  The zeros are
//!  returned in a column vector
Vector2d zerosquadpol(double alpha, double beta){
  Vector2d z;
```

```
  double D = std::pow(alpha,2) -4*beta; // discriminant
  if(D < 0) throw "no real zeros";
  else{
    // The famous discriminant formula
    double wD = std::sqrt(D);
    z << (-alpha-wD)/2, (-alpha+wD)/2; //
  }
  return z;
}
```

This formula is applied to the quadratic polynomial $p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})$ after its coefficients α, β have been computed from γ, which will have introduced *small* relative roundoff errors (of size EPS).

C++11-code 1.5.42: Testing the accuracy of computed roots of a quadratic polynomial
➷ GITLAB

```
//!  Eigen Function for testing the computation of the zeros of a
    parabola
void compzeros(){
  int n = 100;
  MatrixXd res(n,4);
  VectorXd gamma = VectorXd::LinSpaced(n, 2,992);
  for(int i = 0; i < n; ++i){
    double alpha = -(gamma(i) + 1./gamma(i));
    double beta = 1.;
    Vector2d z1 =  zerosquadpol(alpha, beta);
    Vector2d z2 = zerosquadpolstab(alpha, beta);
    double ztrue = 1./gamma(i), z2true = gamma(i);
    res(i,0) = gamma(i);
    res(i,1) = std::abs((z1(0)-ztrue)/ztrue);
    res(i,2) = std::abs((z2(0)-ztrue)/ztrue);
    res(i,3) = std::abs((z1(1)-z2true)/z2true);
  }
  // Graphical output of relative error of roots computed by unstable
     implementation
  mgl::Figure fig1;
  fig1.setFontSize(3);
  fig1.title("Roots of a parabola computed in an unstable manner");
  fig1.plot(res.col(0),res.col(1), " +r").label("small root");
  fig1.plot(res.col(0),res.col(3), " *b").label("large root");
  fig1.xlabel("\\gamma");
  fig1.ylabel("relative errors in \\xi_1, \\xi_2");
  fig1.legend(0.05,0.95);
  fig1.save("zqperrinstab");
  // Graphical output of relative errors (comparison), small roots
  mgl::Figure fig2;
  fig2.title("Roundoff in the computation of zeros of a parabola");
  fig2.plot(res.col(0), res.col(1), " +r").label("unstable");
  fig2.plot(res.col(0), res.col(3), " *m").label("stable");
  fig2.xlabel("\\gamma");
```

```
34    fig2.ylabel("relative errors in \\xi_1");
35    fig2.legend(0.05,0.95);
36    fig2.save("zqperr");
37  }
```

Observation:

Roundoff incurred during the computation of α and β leads to "wrong" roots.
For large γ the computed small root may be fairly inaccurate as regards its *relative error*, which can be several orders of magnitude larger than machine precision EPS.

The large root always enjoys a small relative error about the size of EPS.

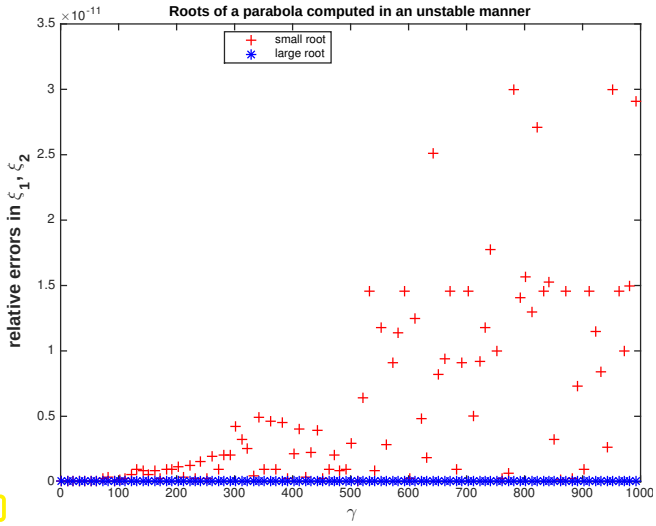


Fig. 40

In order to understand why the small root is much more severely affected by roundoff, note that its computation involves the subtraction of two large numbers, if γ is large. This is the typical situation, in which cancellation occurs.

(1.5.43) Visualisation of cancellation effect

We look at the *exact* subtraction of two almost equal positive numbers both of which have small relative errors (red boxes) with respect to some desired exact value (indicated by blue boxes). The result of the subtraction will be small, but the errors may add up during the subtraction, ultimately constituting a large fraction of the result.

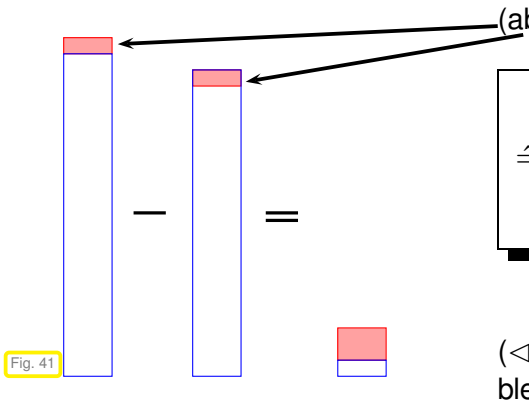


Fig. 41

Cancellation

$\hat{=}$ Subtraction of almost equal numbers

(➤ extreme *amplification* of *relative* errors)

(◁ Roundoff error introduced by subtraction itself is negligible.)

Example 1.5.44 (Cancellation in decimal system)

We consider two positive numbers x, y of about the same size afflicted with relative errors $\approx 10^{-7}$. This means that their seventh decimal digits are perturbed, here indicated by $*$. When we subtract the two numbers the perturbed digits are shifted to the left, resulting in a possible relative error of $\approx 10^{-3}$:

$$\begin{array}{rcll} x & = & 0.123467* & \leftarrow \text{7th digit perturbed} \\ y & = & 0.123456* & \leftarrow \text{7th digit perturbed} \\ \hline x-y & = & 0.000011* = 0.11*000 \cdot 10^{-4} & \leftarrow \text{3rd digit perturbed} \end{array}$$

padded zeroes

Again, this example demonstrates that cancellation wreaks havoc through error amplification, not through the roundoff error due to the subtraction.

Example 1.5.45 (Cancellation when evaluating difference quotients → [?, Sect. 8.2.6], [?, Ex. 1.3])

From analysis we know that the derivative of a differentiable function $f : I \subset \mathbb{R} \to \mathbb{R}$ at a point $x \in I$ is the limit of a difference quotient

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} .$$

This suggests the following approximation of the derivative by a difference quotient with small but finite $h > 0$

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{for} \quad |h| \ll 1 .$$

Results from analysis tell us that the approximation error should tend to zero for $h \to 0$. More precise quantitative information is provided by the Taylor formula for a twice continuously differentiable function [**?**, p. 5]

$$f(x+h) = f(x) + f'(x)h + \tfrac{1}{2} f''(\xi) h^2 \quad \text{for some} \quad \xi = \xi(x,h) \in [\min\{x, x+h\}, \max\{x, x+h\}] , \tag{1.5.46}$$

from which we infer

$$\frac{f(x+h) - f(x)}{h} - f'(x) = \tfrac{1}{2} h f''(\xi) \quad \text{for some} \quad \xi = \xi(x,h) \in [\min\{x, x+h\}, \max\{x, x+h\}] . \tag{1.5.47}$$

We investigate the approximation of the derivative by difference quotients for $f = \exp$, $x = 0$, and different values of $h > 0$:

C++11-code 1.5.48: Difference quotient approximation of the derivative of exp ↪ GITLAB

```
//!  Difference quotient approximation
//!  of the derivative of exp
void diffq(){
  double h = 0.1, x = 0.0;
  for(int i = 1; i <= 16; ++i){
    double df = (exp(x+h)-exp(x))/h;
    cout << setprecision(14) << fixed;
    cout  << setw(5) << -i
       << setw(20) << df-1 << endl;

    h /= 10;
  }
}
```

Measured relative errors ▷

$\log_{10}(h)$	relative error
-1	0.05170918075648
-2	0.00501670841679
-3	0.00050016670838
-4	0.00005000166714
-5	0.00000500000696
-6	0.00000049996218
-7	0.00000004943368
-8	-0.00000000607747
-9	0.00000008274037
-10	0.00000008274037
-11	0.00000008274037
-12	0.00008890058234
-13	-0.00079927783736
-14	-0.00079927783736
-15	0.11022302462516
-16	-1.00000000000000

We observe an initial decrease of the relative approximation error followed by a steep increase when h drops below 10^{-8}.

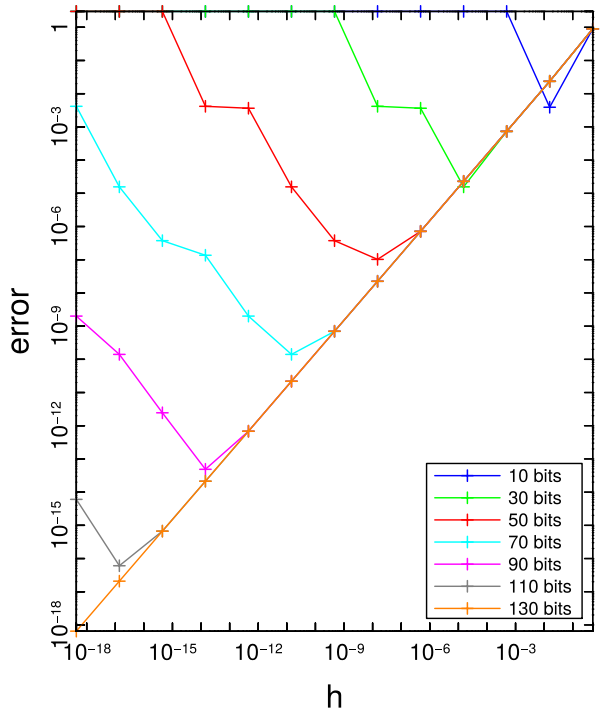


Fig. 42

That the observed errors are really due to round-off errors is confirmed by the following numerical results reported besides, using a variable precision floating point module of EIGEN, the MPFRC++ Support module.

C++11-code 1.5.49: Evaluation of difference quotients with variable precision ↪ GITLAB

```
// This module provides support for multi precision floating point
   numbers via the MPFR C++ library which itself is built upon
   MPFR/GMP.
#include <unsupported/Eigen/MPRealSupport>
//!  Numerical differentiation of exponential function with extended
   precision arithmetic
```

```
5  //!  Uses the unsupported EIGEN MPFRC++ Support module
6  void numericaldifferentiation(){
7       // declare numeric type
8       typedef mpfr::mpreal numeric_t;
9       // bit number that should be evaluated
10      int n = 7, k = 13;
11      VectorXi bits(n); bits << 10,30,50,70,90,110,130;
12      MatrixXd experr(13, bits.size()+1);
13      for(int i = 0; i < n; ++i){
14           // set precision to bits(i) bits (double has 53 bits)
15           numeric_t::set_default_prec( bits(i) );
16           numeric_t x = "1.1";// Evaluation point in extended precision
17           for(int j=0;j<k;++j) {
18                numeric_t h = mpfr::pow("2", -1-5*j);// width of
                     difference quotient in extended precision
19                // compute (absolute) error
20                experr(j,i+1) = mpfr::abs(( mpfr::exp(x+h) -
                     mpfr::exp(x) ) / h - mpfr::exp(x)).toDouble();
21                experr(j,0) = h.toDouble();
22           }
23      }
24      // Plotting
25      // ...
```

Line 16: The literal `"1.1"` in instead of `1.1` prevents the conversion to double.

Obvious culprit: cancellation when computing the numerator of the difference quotient for small $|h|$ leads to a strong amplification of inevitable errors introduced by the evaluation of the transcendent exponential function.

We witness the competition of two opposite effects: Smaller h results in a better approximation of the derivative by the difference quotient, but the impact of cancellation is the stronger the smaller $|h|$.

$$\left.\begin{array}{l}\text{Approximation error} \quad f'(x) - \dfrac{f(x+h)-f(x)}{h} \to 0 \\ \text{Impact of roundoff} \to \infty\end{array}\right\} \quad \text{as } h \to 0\,.$$

In order to provide a rigorous underpinning for our conjecture, in this example we embark on our first roundoff error analysis merely based on the "Axiom of roundoff analysis" Ass. 1.5.32: As in the computational example above we study the approximation of $f'(x) = e^x$ for $f = \exp$, $x \in \mathbb{R}$.

$$\begin{aligned} \texttt{df} &= \frac{e^{x+h}\boxed{(1+\delta_1)} - e^x\boxed{(1+\delta_2)}}{h} \\ &= e^x\left(\frac{e^h-1}{h} + \frac{\delta_1 e^h - \delta_2}{h}\right) \\ \Rightarrow \quad |\texttt{df}| &\le e^x\left(\frac{e^h-1}{h} + \text{eps}\,\frac{1+e^h}{h}\right) \end{aligned}$$

correction factors take into account roundoff:
(→ "'axiom of roundoff analysis", Ass. 1.5.32)

$$|\delta_1|, |\delta_2| \le \text{eps}\,.$$

$1+O(h)$ $\quad O(h^{-1})$ for $h \to 0$

(Note that the estimate for the term $(e^h - 1)/h$ is a particular case of (1.5.47).)

▶ relative error: $\left|\frac{e^x - \mathtt{df}}{e^x}\right| \approx h + \frac{2\mathtt{eps}}{h} \to \min$ for $h = \sqrt{2\,\mathtt{eps}}$.

In double precision: $\sqrt{2\mathtt{eps}} = 2.107342425544702 \cdot 10^{-8}$

Remark 1.5.50 (Cancellation during the computation of relative errors)

In the numerical experiment of Ex. 1.5.45 we computed the relative error of the result by subtraction, see Code 1.5.48. Of course, massive cancellation will occur! Do we have to worry?

In this case cancellation can be tolerated, because we are *interested only* in the magnitude of the relative error. Even if it was affected by a large relative error, this information is still not compromised.

For example, if the relative error has the exact value 10^{-8}, but can be computed only with a huge relative error of 10%, then the perturbed value would still be in the range $[0.9 \cdot 10^{-8}, 1.1 \cdot 10^{-8}]$. Therefore it will still have the correct magnitude and still permit us to conclude the number of valid digits correctly.

Remark 1.5.51 (Cancellation in Gram-Schmidt orthogonalisation of Exp. 1.5.5)

The matrix created by the MATLAB command `A = hilb(10)`, the so-called Hilbert matrix, has columns that are almost linearly dependent.

Cancellation when computing orthogonal projection of vector $\mathbf{a}$ onto space spanned by vector $\mathbf{b}$ ▷

$$\mathbf{p} = \mathbf{a} - \frac{\mathbf{a}\cdot\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}\mathbf{b} .$$

If $\mathbf{a}$, $\mathbf{b}$ point in almost the same direction, $\|\mathbf{p}\| \ll \|\mathbf{a}\|, \|\mathbf{b}\|$, so that a "tiny" vector $\mathbf{p}$ is obtained by subtracting two "long" vectors, which implies cancellation.

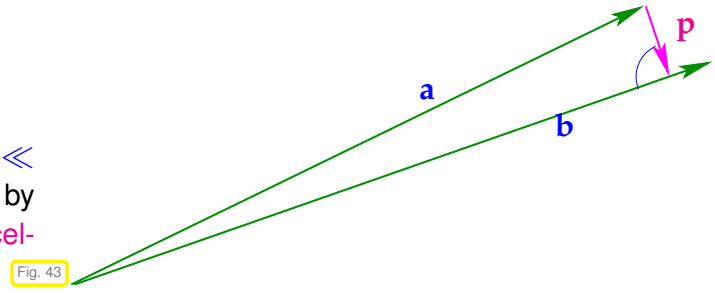


Fig. 43

This can happen in Line 10 of Code 1.5.3.

Example 1.5.52 (Cancellation: roundoff error analysis)

We consider a simple arithmetic expression written in two ways:

$$a^2 - b^2 = (a+b)(a-b) , \quad a, b \in \mathbb{R}; .$$

We evaluate this term by means of two algebraically equivalent algorithms for the input data $a = 1.3$, $b = 1.2$ in 2-digit decimal arithmetic with standard rounding. ("Algebraically equivalent" means that two algorithms will produce the same results in the absence of roundoff errors.

Algorithm A	Algorithm B
$x := a \tilde{\cdot} a = 1.7$ (rounded)	$x := a \tilde{+} b = 2.5$ (exact)
$y := b \tilde{\cdot} b = 1.4$ (rounded)	$y := a \tilde{-} b = 0.1$ (exact)
$x \tilde{-} y = 0.30$ (exact)	$x * y = 0.25$ (exact)

Algorithm B produces the exact result, whereas Algorithm A fails to do so. Is this pure coincidence or an indication of the superiority of algorithm B? This question can be answered by roundoff error analysis. We demonstrate the approach for the two algorithms A & B and general input $a, b, \in \mathbb{R}$.

Roundoff error analysis heavily relies on Ass. 1.5.32 and dropping terms of "higher order" in the machine precision, that is terms that behave like $O(\mathrm{EPS}^q)$, $q > 1$. It involves introducing the relative roundoff error for every elementary operation through a factor $(1+\delta)$, $|\delta| \leq \mathrm{EPS}$.

Algorithm A:

$$x = a^2(1+\delta_1)\,, y = b^2(1+\delta_2)$$
$$\tilde{f} = (a^2(1+\delta_1) - b^2(1+\delta_2))(1+\delta_3) = f + a^2\delta_1 - b^2\delta_2 + (a^2-b^2)\delta_3 + O(\mathrm{EPS}^2)$$

$$\frac{|\tilde{f}-f|}{|f|} \leq \mathrm{EPS}\frac{a^2+b^2+|a^2-b^2|}{|a^2-b^2|} + O(\mathrm{EPS}^2) = \mathrm{EPS}\left(1+\frac{|a^2+b^2|}{|a^2-b^2|}\right) + O(\mathrm{EPS}^2)\,. \tag{1.5.53}$$

will be neglected

For $a \approx b$ the relative error of the result of Algorithm A will be much larger than the machine precision EPS. This reflects cancellation in the last subtraction step.

Algorithm B:

$$x = (a+b)(1+\delta_1)\,, y = (a-b)(1+\delta_2)$$
$$\tilde{f} = (a+b)(a-b)(1+\delta_1)(1+\delta_2)(1+\delta_3) = f + (a^2-b^2)(\delta_1+\delta_2+\delta_3) + O(\mathrm{EPS}^2)$$

$$\frac{|\tilde{f}-f|}{|f|} \leq |\delta_1+\delta_2+\delta_3| + O(\mathrm{EPS}^2) \leq 3\mathrm{EPS} + O(\mathrm{EPS}^2)\,. \tag{1.5.54}$$

Relative error of the result of Algorithm B is always $\approx$ EPS **!**

In this example we see a general guideline at work:

> If inevitable, subtractions prone to cancellation should be done as early as possible.

The reason is that input data and and initial intermediate results are usually not as much tainted by roundoff errors as numbers computed after many steps.

(1.5.55) Avoiding disastrous cancellation

The following examples demonstrate a few fundamental techniques for steering clear of cancellation by using alternative formulas that yield the same value (in exact arithmetic), but do not entail subtracting two numbers of almost equal size.

Example 1.5.56 (Stable discriminant formula → Ex. 1.5.39, [?, Ex. 2.10])

If ξ_1 and ξ_2 are the two roots of the quadratic polynomial $p(\xi) = \xi^2 + \alpha\xi + \beta$, then $\xi_1 \cdot \xi_2 = \beta$ (Vieta's formula). Thus once we have computed a root, we can obtain the other by simple division.

Idea:

❶ Depending on the sign of α compute "stable root" without cancellation.
❷ Compute other root from Vieta's formula (avoiding subtraction)

C++11-code 1.5.57: Stable computation of real root of a quadratic polynomial ↪ GITLAB

```
2  //!  C++ function computing the zeros of a quadratic polynomial
3  //!  ξ → ξ^2 + αξ + β by means of the familiar discriminant
4  //!  formula ξ_{1,2} = 1/2(−α ± sqrt(α^2 − 4β)).
5  //!  This is a stable implementation based on Vieta's theorem.
6  //!  The zeros are returned in a column vector
7  VectorXd zerosquadpolstab(double alpha, double beta){
8    Vector2d z(2);
9    double D = std::pow(alpha,2) −4*beta; // discriminant
10   if(D < 0) throw "no real zeros";
11   else{
12     double wD = std::sqrt(D);
13     // Use discriminant formula only for zero far away from 0
14     // in order to avoid cancellation. For the other zero
15     // use Vieta's formula.
16     if(alpha >= 0){
17       double t = 0.5*(−alpha−wD); //
18       z << t, beta/t;
19     }
20     else{
21       double t = 0.5*(−alpha+wD); //
22       z << beta/t, t;
23     }
24   }
25   return z;
26 }
```

➥ Invariably, we add numbers with the same sign in Line 17 and Line 21.

Numerical experiment based on the driver code Code 1.5.42.

Observation:

The new code can also compute the small root of the polynomial $p(\xi) = (\xi - \gamma)(\xi - \frac{1}{\gamma})$ (expanded in monomials) with a relative error $\approx$ EPS.

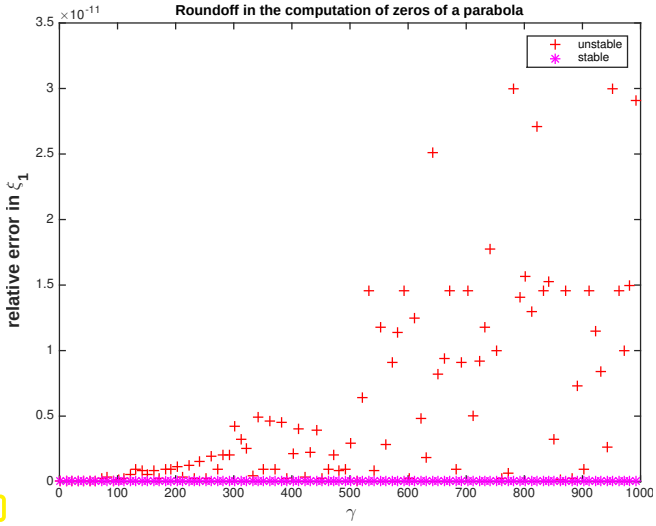


Fig. 44

Example 1.5.58 (Exploiting trigonometric identities to avoid cancellation)

The task is to evaluate the integral

$$\int_0^x \sin t \, dt = \underbrace{1 - \cos x}_{I} = \underbrace{2\sin^2(x/2)}_{II} \quad \text{for} \quad 0 < x \ll 1 \,, \tag{1.5.59}$$

and this can be done by the two different formulas I and II.

Relative error of expression I (`1-cos(x)`) with respect to equivalent expression II (`2*sin(x/2)^{2}`) ▷

Expression I is affected by cancellation for $|x| \ll 1$, since then $\cos x \approx 1$, whereas expression II can be evaluated with a relative error $\approx$ EPS for all x.

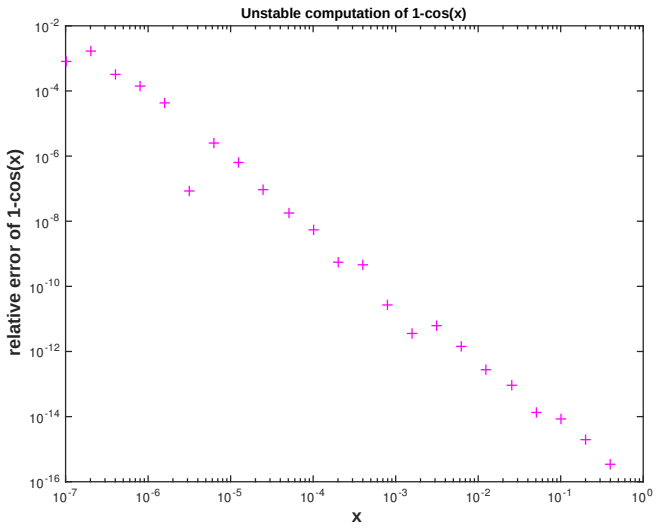


Fig.

Analytic manipulations offer ample opportunity to rewrite expressions in equivalent form immune to cancellation.

Example 1.5.60 (Switching to equivalent formulas to avoid cancellation)

Now we see an example of a computation allegedly dating back to Archimedes, who tried to approximate the area of a circle by the areas of inscribed regular polygons.

Approximation of circle by regular n-gon, $n \in \mathbb{N}$ ▷

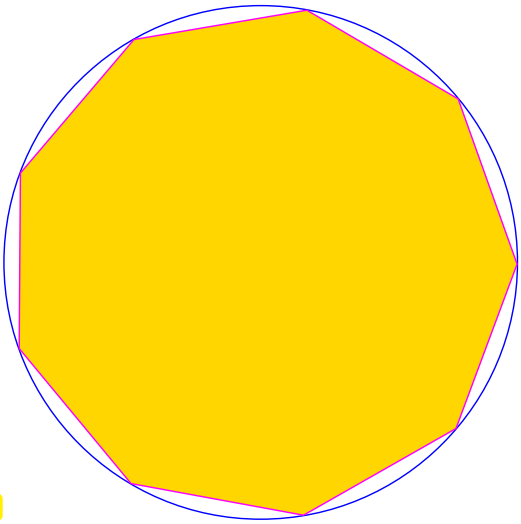


Fig. 46

Area of n-gon:

$$A_n = n \cos\frac{\alpha_n}{2} \sin\frac{\alpha_n}{2} = \frac{n}{2}\sin\alpha_n = \frac{n}{2}\sin\left(\frac{2\pi}{n}\right) .$$

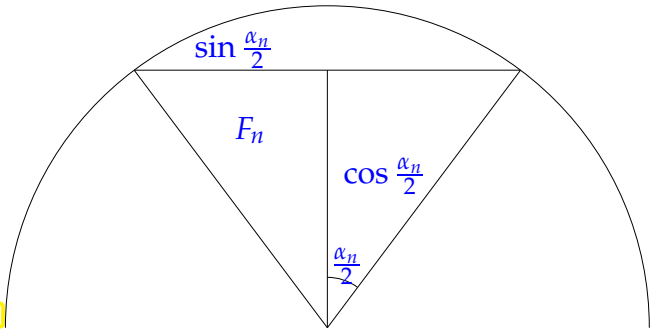


Fig. 47

Recursion formula for A_n derived from

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\cos\alpha_n}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}} ,$$

Initial approximation: $A_6 = \frac{3}{2}\sqrt{3}$.

C++11-code 1.5.61: Tentative computation of circumference of regular polygon ⇝ GITLAB

```
2  //!  Approximation of Pi by approximating the circumference of a
3  //!  regular polygon
4  MatrixXd ApproxPlinstable(double tol = 1e-8, int maxIt = 50){
5    double s=sqrt(3)/2.; double An=3.*s;// initialization (hexagon case)
6    unsigned int n = 6, it = 0;
7    MatrixXd res(maxIt,4); // matrix for storing results
8    res(it,0) = n; res(it,1) = An;
9    res(it,2) = An - M_PI; res(it,3)=s;
10   while( it < maxIt && s > tol ){// terminate when s is 'small enough'
11     s = sqrt((1.- sqrt(1.-s*s))/2.);// recursion for area
12     n *= 2; An = n/2.*s;      // new estimate for circumference
13     ++it;
14     res(it,0) =n; res(it,1) =An;// store results and (absolute) error
15     res(it,2) = An - M_PI; res(it,3)=s;
16   }
17   return res.topRows(it);
18 }
```

The approximation deteriorates after applying the recursion formula many times:

n	A_n	$A_n - \pi$	$\sin \alpha_n$
6	2.598076211353316	-0.543516442236477	0.866025403784439
12	3.000000000000000	-0.141592653589794	0.500000000000000
24	3.105828541230250	-0.035764112359543	0.258819045102521
48	3.132628613281237	-0.008964040308556	0.130526192220052
96	3.139350203046872	-0.002242450542921	0.065403129230143
192	3.141031950890530	-0.000560702699263	0.032719082821776
384	3.141452472285344	-0.000140181304449	0.016361731626486
768	3.141557607911622	-0.000035045678171	0.008181139603937
1536	3.141583892148936	-0.000008761440857	0.004090604026236
3072	3.141590463236762	-0.000002190353031	0.002045306291170
6144	3.141592106043048	-0.000000547546745	0.001022653680353
12288	3.141592516588155	-0.000000137001638	0.000511326906997
24576	3.141592618640789	-0.000000034949004	0.000255663461803
49152	3.141592645321216	-0.000000008268577	0.000127831731987
98304	3.141592645321216	-0.000000008268577	0.000063915865994
196608	3.141592645321216	-0.000000008268577	0.000031957932997
393216	3.141592645321216	-0.000000008268577	0.000015978966498
786432	3.141593669849427	0.000001016259634	0.000007989485855
1572864	3.141592303811738	-0.000000349778055	0.000003994741190
3145728	3.141608696224804	0.000016042635011	0.000001997381017
6291456	3.141586839655041	-0.000005813934752	0.000000998683561
12582912	3.141674265021758	0.000081611431964	0.000000499355676
25165824	3.141674265021758	0.000081611431964	0.000000249677838
50331648	3.143072740170040	0.001480086580246	0.000000124894489
100663296	3.159806164941135	0.018213511351342	0.000000062779708
201326592	3.181980515339464	0.040387861749671	0.000000031610136
402653184	3.354101966249685	0.212509312659892	0.000000016660005
805306368	4.242640687119286	1.101048033529493	0.000000010536712
1610612736	6.000000000000000	2.858407346410207	0.000000007450581

Where does cancellation occur in Line 11 of Code 1.5.61? Since $s \ll 1$, computing $1 - s$ will not trigger cancellation. However, the subtraction $1 - \sqrt{1 - s^2}$ will, because $\sqrt{1 - s^2} \approx 1$ for $s \ll 1$:

$$\sin\frac{\alpha_n}{2} = \sqrt{\frac{1-\cos\alpha_n}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}}$$

For $\alpha_n \ll 1$: $\sqrt{1-\sin^2\alpha_n} \approx 1$

Cancellation here!

We arrive at an *equivalent* formula not vulnerable to cancellation essentially using the identity $(a+b)(a-b) = a^2 - b^2$ in order to eliminate the difference of square roots in the numerator.

$$\begin{aligned}\sin\frac{\alpha_n}{2} &= \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}} = \sqrt{\frac{1-\sqrt{1-\sin^2\alpha_n}}{2}\cdot\frac{1+\sqrt{1-\sin^2\alpha_n}}{1+\sqrt{1-\sin^2\alpha_n}}} \\ &= \sqrt{\frac{1-(1-\sin^2\alpha_n)}{2(1+\sqrt{1-\sin^2\alpha_n})}} = \frac{\sin\alpha_n}{\sqrt{2\left(1+\sqrt{1-\sin^2\alpha_n}\right)}}\,.\end{aligned}$$

C++11-code 1.5.62: Stable recursion for area of regular n-gon ⇝ GITLAB

```
2 //!  Approximation of Pi by approximating the circumference of a
3 //!  regular polygon
```

```
4  MatrixXd apprpistable(double tol = 1e-8, int maxIt = 50){
5    double s=sqrt(3)/2.; double An=3.*s;// initialization (hexagon case)
6    unsigned int n = 6, it = 0;
7    MatrixXd res(maxIt,4); // matrix for storing results
8    res(it,0) = n; res(it,1) = An;
9    res(it,2) = An - M_PI; res(it,3)=s;
10   while( it < maxIt && s > tol ){// terminate when s is 'small enough'
11     s = s/sqrt(2*(1+sqrt((1+s)*(1-s))));// Stable recursion without
         cancellation
12     n *= 2; An = n/2.*s;     // new estimate for circumference
13     ++it;
14     res(it,0) =n; res(it,1) =An;// store results and (absolute) error
15     res(it,2) = An - M_PI; res(it,3)=s;
16   }
17   return res.topRows(it);
18 }
```

Using the stable recursion, we observe better approximation for polygons with more corners:

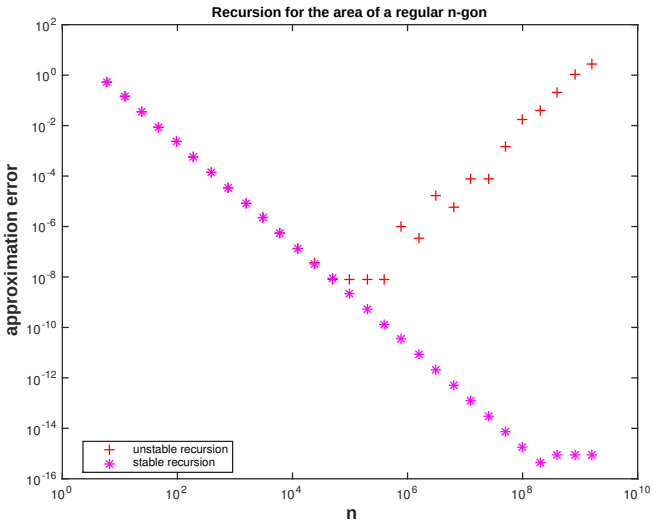
n	A_n	$A_n - \pi$	$\sin \alpha_n$
6	2.598076211353316	-0.543516442236477	0.866025403784439
12	3.000000000000000	-0.141592653589793	0.500000000000000
24	3.105828541230249	-0.035764112359544	0.258819045102521
48	3.132628613281238	-0.008964040308555	0.130526192220052
96	3.139350203046867	-0.002242450542926	0.065403129230143
192	3.141031950890509	-0.000560702699284	0.032719082821776
384	3.141452472285462	-0.000140181304332	0.016361731626487
768	3.141557607911857	-0.000035045677936	0.008181139603937
1536	3.141583892148318	-0.000008761441475	0.004090604026235
3072	3.141590463228050	-0.000002190361744	0.002045306291164
6144	3.141592105999271	-0.000000547590522	0.001022653680338
12288	3.141592516692156	-0.000000136897637	0.000511326907014
24576	3.141592619365383	-0.000000034224410	0.000255663461862
49152	3.141592645033690	-0.000000008556103	0.000127831731976
98304	3.141592651450766	-0.000000002139027	0.000063915866118
196608	3.141592653055036	-0.000000000534757	0.000031957933076
393216	3.141592653456104	-0.000000000133690	0.000015978966540
786432	3.141592653556371	-0.000000000033422	0.000007989483270
1572864	3.141592653581438	-0.000000000008355	0.000003994741635
3145728	3.141592653587705	-0.000000000002089	0.000001997370818
6291456	3.141592653589271	-0.000000000000522	0.000000998685409
12582912	3.141592653589663	-0.000000000000130	0.000000499342704
25165824	3.141592653589761	-0.000000000000032	0.000000249671352
50331648	3.141592653589786	-0.000000000000008	0.000000124835676
100663296	3.141592653589791	-0.000000000000002	0.000000062417838
201326592	3.141592653589794	0.000000000000000	0.000000031208919
402653184	3.141592653589794	0.000000000000001	0.000000015604460
805306368	3.141592653589794	0.000000000000001	0.000000007802230
1610612736	3.141592653589794	0.000000000000001	0.000000003901115

Plot of errors for approximations of π as computed by the two algebraically equivalent recursion formulas▷

Observation, *cf.* Ex. 1.5.45

Amplified roundoff errors due to cancellation supersedes approximation error for $n \geq 10^5$.

Roundoff errors merely of magnitude EPS in the case of stable recursion



Example 1.5.63 (Summation of exponential series)

In principle, the function value $\exp(x)$ can be approximated up to any accuracy by summing sufficiently many terms of the globally convergent exponential series.

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$
$$= 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ldots .$$

C++11-code 1.5.64: Summation of exponential series ↪ GITLAB

```
double expeval(double x,
               double tol=1e-8){
  // Initialization
  double y = 1.0, term = 1.0;
  long int k = 1;
  // Termination criterion
  while(abs(term) > tol*y) {
    term *= x/k;    // next summand
    y += term; // Summation
    ++k;
  }
  return y;
}
```

Results for $\texttt{tol} = 10^{-8}$, $\widetilde{\exp}$ designates the approximate value for $\exp(x)$ returned by the function from Code 1.5.64. Rightmost column lists relative errors, which tells us the number of valid digits in the approximate result.

x	Approximation $\widetilde{\exp}(x)$	$\exp(x)$	$\frac{\lvert\exp(x)-\widetilde{\exp}(x)\rvert}{\exp(x)}$
-20	6.1475618242e-09	2.0611536224e-09	1.982583033727893
-18	1.5983720359e-08	1.5229979745e-08	0.049490585500089
-16	1.1247503300e-07	1.1253517472e-07	0.000534425951530
-14	8.3154417874e-07	8.3152871910e-07	0.000018591829627
-12	6.1442105142e-06	6.1442123533e-06	0.000000299321453
-10	4.5399929604e-05	4.5399929762e-05	0.000000003501044
-8	3.3546262812e-04	3.3546262790e-04	0.000000000662004
-6	2.4787521758e-03	2.4787521767e-03	0.000000000332519
-4	1.8315638879e-02	1.8315638889e-02	0.000000000530724
-2	1.3533528320e-01	1.3533528324e-01	0.000000000273603
0	1.0000000000e+00	1.0000000000e+00	0.000000000000000
2	7.3890560954e+00	7.3890560989e+00	0.000000000479969
4	5.4598149928e+01	5.4598150033e+01	0.000000001923058
6	4.0342879295e+02	4.0342879349e+02	0.000000001344248
8	2.9809579808e+03	2.9809579870e+03	0.000000002102584
10	2.2026465748e+04	2.2026465795e+04	0.000000002143799
12	1.6275479114e+05	1.6275479142e+05	0.000000001723845
14	1.2026042798e+06	1.2026042842e+06	0.000000003634135
16	8.8861105010e+06	8.8861105205e+06	0.000000002197990
18	6.5659968911e+07	6.5659969137e+07	0.000000003450972
20	4.8516519307e+08	4.8516519541e+08	0.000000004828737

Observation:

Large relative approximation errors for $x \ll 0$.

For $x \ll 0$ we have $\exp(x)| \ll 1$, but this value is computed by summing large numbers of opposite sign.

Terms summed up for $x = -20$ ▷

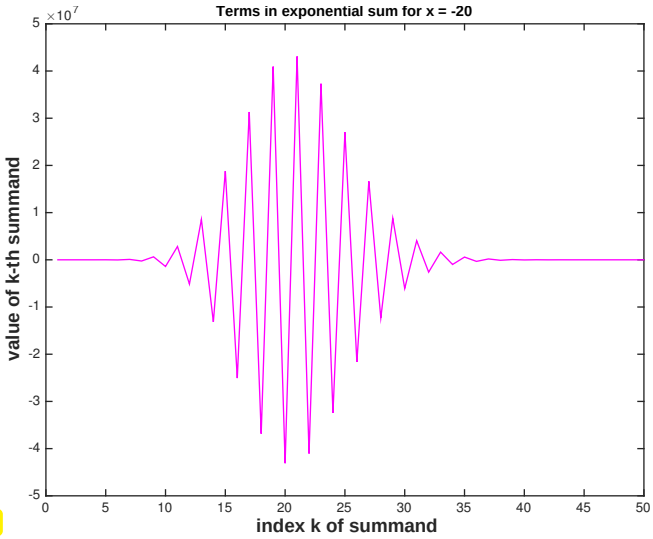


Fig. 49

Remedy: Cancellation can be avoided by using identity

$$\exp(x) = \frac{1}{\exp(-x)} \quad \text{, if} \quad x < 0\,.$$

Example 1.5.65 (Combat cancellation by approximation)

In a computer code we have to provide a routine for the evaluation of

$$\int_0^1 e^{at}\,\mathrm{d}t = \frac{\exp(a)-1}{a} \quad \text{for any} \quad a > 0\,, \tag{1.5.66}$$

cf. the discussion of cancellation in the context of numerical differentiation in Ex. 1.5.45. There we

observed massive cancellation.

Recall the Taylor expansion formula in one dimension for a function that is $m+1$ times continuously differentiable in a neighborhood of x [**?**, Satz 5.5.1]

$$f(x+h) = \sum_{k=0}^{m} \frac{1}{k!} f^{(k)}(x) h^k + R_m(x,h) \;, \quad R_m(x,h) = \frac{1}{(m+1)!} f^{(m+1)}(\xi) h^{m+1} \;,$$

for some $\xi \in [\min\{x, x+h\}, \max\{x, x+h\}]$, and for all sufficiently small $|h|$. Here $R(x,h)$ is called the remainder term and $f^{(k)}$ denotes the k derivative of f.

Cancellation in (1.5.66) can be avoided by replacing $\exp(a)$, $a > 0$, with a suitable Taylor expansion around $a = 0$ and then dividing by a:

$$\frac{\exp(a)-1}{a} = \sum_{k=0}^{m} \frac{1}{(k+1)!} a^k + R_m(a) \;, \quad R_m(a) = \frac{1}{(m+1)!} \exp(\xi) a^m \text{ for some } 0 \le \xi \le a \;.$$

For a similar discussion see [**?**, Ex. 2.12].

Issue: How to choose the number m of terms to be retained in the Taylor expansion? We have to pick m large enough such that the relative approximation error remains below a prescribed threshold `tol`. To *estimate* the relative approximation error, we use the expression for the remainder together with the simple estimate $(\exp(a)-1)/a > 1$ for all $a > 0$:

$$\text{rel. err.} = \frac{(e^a - 1)/a - \sum_{k=0}^{m} \frac{1}{(k+1)!} a^k}{(e^a-1)/a} \le \frac{1}{(m+1)!} \exp(\xi) a^m \le \frac{1}{(m+1)!} \exp(a) a^m \;.$$

For $a = 10^{-3}$ we get

m	1	2	3	4	5
	1.0010e-03	5.0050e-07	1.6683e-10	4.1708e-14	8.3417e-18

Hence, keeping $m = 3$ terms is enough for achieving about 10 valid digits.

Relative error of unstable formula `(exp(a)-1.0)/a` and relative error, when using a Taylor expansion approximation for small a ▷

```
if (abs(a) < 1E-3)
  v = 1.0 + (1.0/2 + 1.0/6*a)*a;
else
  v = (exp(a)-1.0)/a;
end
```

Error computed by comparison with MATLAB's built-in function `expm1` that provides a stable implementation of $\exp(x) - 1$.

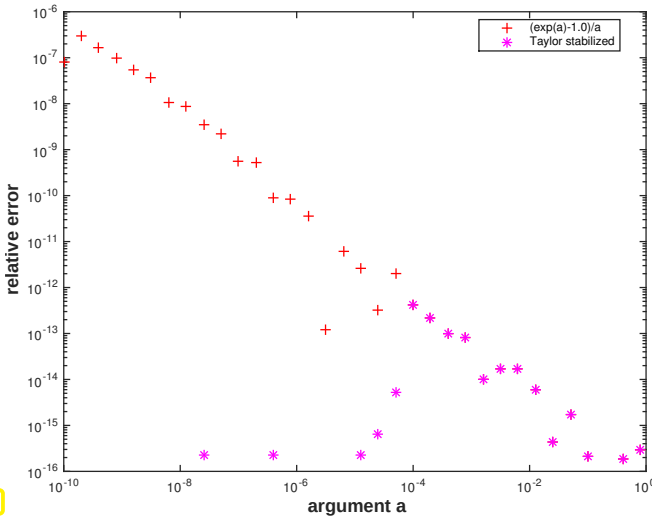


Fig. 50

1.5.5 Numerical stability

We have seen that a particular "problem" can be tackled by different "algorithms", which produce different results due to roundoff errors. This section will clarify what distinguishes a "good" algorithm from a rather abstract point of view.

(1.5.67) The "problem"

A mathematical notion of "problem":

- data space X, usually $X \subset \mathbb{R}^n$
- result space Y, usually $Y \subset \mathbb{R}^m$
- mapping (problem function) $F : X \mapsto Y$

A problem is a well defined *function* that assigns to each datum a result.

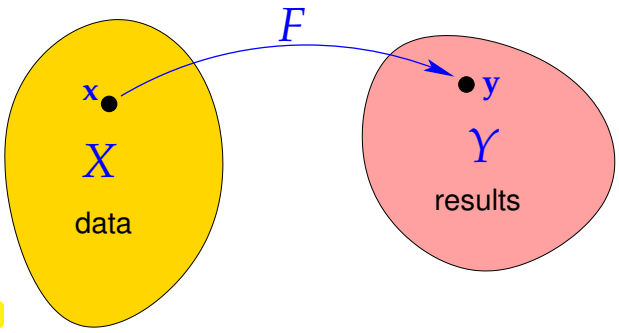


Fig. 51

Note: In this course, both the data space X and the result space Y will always be subsets of finite dimensional vector spaces.

Example 1.5.68 (The "matrix×vector-multiplication problem")

We consider the "problem" of computing the product $\mathbf{Ax}$ for a given matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ and a given vector $\mathbf{x} \in \mathbb{K}^n$.

➢
- Data space $X = \mathbb{K}^{m,n} \times \mathbb{K}^n$ (input is a matrix and a vector)
- Result space $Y = \mathbb{R}^m$ (space of column vectors)
- Problem function $F : X \to Y$, $F(\mathbf{a}, \mathbf{x}) := \mathbf{Ax}$

(1.5.69) Norms on spaces of vectors and matrices

Norms provide tools for measuring errors. Recall from linear algebra and calculus [**?**, Sect. 4.3], [**?**, Sect. 6.1]:

Definition 1.5.70. Norm

X = vector space over field $\mathbb{K}$, $\mathbb{K} = \mathbb{C}, \mathbb{R}$. A map $\|\cdot\| : X \mapsto \mathbb{R}_0^+$ is a norm on X, if it satisfies

(i) $\forall \mathbf{x} \in X$: $\mathbf{x} \neq 0 \Leftrightarrow \|\mathbf{x}\| > 0$ (definite),
(ii) $\|\lambda \mathbf{x}\| = |\lambda| \|\mathbf{x}\| \quad \forall \mathbf{x} \in X, \lambda \in \mathbb{K}$ (homogeneous),
(iii) $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in X$ (triangle inequality).

Examples: (for vector space $\mathbb{K}^n$, vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T \in \mathbb{K}^n$)

name	:	definition	EIGEN function
Euclidean norm	:	$\|\mathbf{x}\|_2 := \sqrt{\lvert x_1\rvert^2 + \cdots + \lvert x_n\rvert^2}$	`x.norm()`
1-norm	:	$\|\mathbf{x}\|_1 := \lvert x_1\rvert + \cdots + \lvert x_n\rvert$	`x.lpNorm<1>()`
∞-norm, max norm	:	$\|\mathbf{x}\|_\infty := \max\{\lvert x_1\rvert, \ldots, \lvert x_n\rvert\}$	`x.lpNorm<Eigen::Infinity>()`

Remark 1.5.71 (Inequalities between vector norms)

All norms on the vector space $\mathbb{K}^n$, $n \in \mathbb{N}$, are equivalent in the sense that for arbitrary two norms $\|\cdot\|_1$ and $\|\cdot\|_2$ we can always find a constant $C > 0$ such that

$$\|\mathbf{v}\|_1 \leq C \|\mathbf{v}\|_2 \quad \forall \mathbf{v} \in \mathbb{K}^n \,. \tag{1.5.72}$$

Of course, the constant C will usually depend on n and the norms under consideration.

For the vector norms introduced above, explicit expressions for the constants "C" are available: for all $\mathbf{x} \in \mathbb{K}^n$

$$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n}\|\mathbf{x}\|_2 , \tag{1.5.73}$$
$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n}\|\mathbf{x}\|_\infty , \tag{1.5.74}$$
$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n\|\mathbf{x}\|_\infty . \tag{1.5.75}$$

The matrix space $\mathbb{K}^{m,n}$ is a vector space, of course, and can also be equipped with various norms. Of particular importance are norms *induced by vector norms* on $\mathbb{K}^n$ and $\mathbb{K}^m$.

Definition 1.5.76. Matrix norm

Given vector norms $\|\cdot\|_1$ and $\|\cdot\|_2$ on $\mathbb{K}^n$ and $\mathbb{K}^m$, respectively, the associated matrix norm is defined by

$$\mathbf{M} \in \mathbb{R}^{m,n}: \quad \|\mathbf{M}\| := \sup_{\mathbf{x}\in\mathbb{R}^n\setminus\{0\}} \frac{\|\mathbf{M}\mathbf{x}\|_2}{\|\mathbf{x}\|_1} .$$

By virtue of definition the matrix norms enjoy an important property, they are sub-multiplicative:

$$\forall \mathbf{A} \in \mathbb{K}^{n,m}, \mathbf{B} \in \mathbb{K}^{m,k}: \quad \|\mathbf{A}\mathbf{B}\| \leq \|\mathbf{A}\|\|\mathbf{B}\| . \tag{1.5.77}$$

✎ notations for matrix norms for *quadratic matrices* associated with standard vector norms:

$$\|\mathbf{x}\|_2 \to \|\mathbf{M}\|_2 , \quad \|\mathbf{x}\|_1 \to \|\mathbf{M}\|_1 , \quad \|\mathbf{x}\|_\infty \to \|\mathbf{M}\|_\infty$$

Example 1.5.78 (Matrix norm associated with ∞-norm and 1-norm)

Rather simple formulas are available for the matrix norms induced by the vector norms $\|\cdot\|_\infty$ and $\|\cdot\|_1$

$$\text{e.g. for } \mathbf{M} \in \mathbb{K}^{2,2}: \quad \begin{aligned} \|\mathbf{M}\mathbf{x}\|_\infty &= \max\{|m_{11}x_1 + m_{12}x_2|, |m_{21}x_1 + m_{22}x_2|\} \\ &\leq \max\{|m_{11}| + |m_{12}|, |m_{21}| + |m_{22}|\}\, \|x\|_\infty , \\ \|\mathbf{M}\mathbf{x}\|_1 &= |m_{11}x_1 + m_{12}x_2| + |m_{21}x_1 + m_{22}x_2| \\ &\leq \max\{|m_{11}| + |m_{21}|, |m_{12}| + |m_{22}|\}(|x_1| + |x_2|) . \end{aligned}$$

For general $\mathbf{M} \in \mathbb{K}^{m,n}$

➢ matrix norm $\leftrightarrow \|\cdot\|_\infty$ = row sum norm $\|\mathbf{M}\|_\infty := \max_{i=1,\ldots,m} \sum_{j=1}^{n} |m_{ij}|$, (1.5.79)

➢ matrix norm $\leftrightarrow \|\cdot\|_1$ = column sum norm $\|\mathbf{M}\|_1 := \max_{j=1,\ldots,n} \sum_{i=1}^{m} |m_{ij}|$. (1.5.80)

Sometimes special formulas for the Euclidean matrix norm come handy [**?**, Sect. 2.3.3]:

Lemma 1.5.81. Formula for Euclidean norm of a Hermitian matrix

$$\mathbf{A} \in \mathbb{K}^{n,n}, \mathbf{A} = \mathbf{A}^H \quad \Rightarrow \quad \|\mathbf{A}\|_2 = \max_{\mathbf{x}\neq 0} \frac{|\mathbf{x}^H \mathbf{A}\mathbf{x}|}{\|\mathbf{x}\|_2^2} .$$

Proof. Recall from linear algebra: Hermitian matrices (a special class of normal matrices) enjoy unitary similarity to diagonal matrices:

$$\exists \mathbf{U} \in \mathbb{K}^{n,n}, \text{ diagonal } \mathbf{D} \in \mathbb{R}^{n,n}: \quad \mathbf{U}^{-1} = \mathbf{U}^H \quad \text{and} \quad \mathbf{A} = \mathbf{U}^H \mathbf{D} \mathbf{U} .$$

Since multiplication with an unitary matrix preserves the 2-norm of a vector, we conclude

$$\|\mathbf{A}\|_2 = \left\|\mathbf{U}^H \mathbf{D} \mathbf{U}\right\|_2 = \|\mathbf{D}\|_2 = \max_{i=1,\ldots,i} |d_i| , \quad \mathbf{D} = \operatorname{diag}(d_1, \ldots, d_n) .$$

On the other hand, for the same reason:

$$\max_{\|\mathbf{x}\|_2=1} \mathbf{x}^H \mathbf{A}\mathbf{x} = \max_{\|\mathbf{x}\|_2=1} (\mathbf{U}\mathbf{x})^H \mathbf{D} (\mathbf{U}\mathbf{x}) = \max_{\|\mathbf{y}\|_2=1} \mathbf{y}^H \mathbf{D}\mathbf{y} = \max_{i=1,\ldots,i} |d_i| .$$

Hence, both expressions in the statement of the lemma agree with the largest modulus of eigenvalues of $\mathbf{A}$.

□

Corollary 1.5.82. Euclidean matrix norm and eigenvalues

For $\mathbf{A} \in \mathbb{K}^{m,n}$ the Euclidean matrix norm $\|\mathbf{A}\|_2$ is the square root of the largest (in modulus) eigenvalue of $\mathbf{A}^H\mathbf{A}$.

For a normal matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ (that is, $\mathbf{A}$ satisfies $\mathbf{A}^H\mathbf{A} = \mathbf{A}\mathbf{A}^H$) the Euclidean matrix norm agrees with the modulus of the largest eigenvalue.

(1.5.83) (Numerical) algorithm

When we talk about an "algorithm" we have in mind a concrete code function in MATLAB or C++; the only way to describe an algorithm is through a piece of code. We assume that this function defines another mapping $\widetilde{F} : X \to Y$ on the data space of the problem. Of course, we can only feed data to the MATLAB/C++-function, if they can be represented in the set $\mathbb{M}$ of machine numbers. Hence, implicit in the definition of $\widetilde{F}$ is the assumption that input data are subject to *rounding* before passing them to the code function proper.

Problem	Algorithm
$F : X \subset \mathbb{R}^n \to Y \subset \mathbb{R}^m$	$\widetilde{F} : X \to \widetilde{Y} \subset \mathbb{M}$

(1.5.84) Stable algorithm → **[?, Sect. 1.3]**

[Stable algorithm]

- We study a problem (→ § 1.5.67) $F : X \to Y$ on data space X into result space Y.
- We assume that both X and Y are equipped with norms $\|\cdot\|_X$ and $\|\cdot\|_Y$, respectively (→ Def. 1.5.70).
- We consider a concrete algorithm $\widetilde{F} : X \to Y$ according to § 1.5.83.

We write $w(\mathbf{x})$, $\mathbf{x} \in X$, for the computational effort (→ Def. 1.4.1) required by the algorithm for input $\mathbf{x}$.

Definition 1.5.85. Stable algorithm

An algorithm $\widetilde{F}$ for solving a problem $F : X \mapsto Y$ is numerically stable if for all $\mathbf{x} \in X$ its result $\widetilde{F}(\mathbf{x})$ (possibly affected by roundoff) is the exact result for "slightly perturbed" data:

$$\exists C \approx 1: \quad \forall \mathbf{x} \in X: \quad \exists \widetilde{\mathbf{x}} \in X: \quad \|\mathbf{x} - \widetilde{\mathbf{x}}\|_X \leq C w(\mathbf{x}) \,\mathrm{EPS}\, \|\mathbf{x}\|_X \quad \wedge \quad \widetilde{F}(\mathbf{x}) = F(\widetilde{\mathbf{x}}) \,.$$

Here EPS should be read as machine precision according to the "Axiom" of roundoff analysis Ass. 1.5.32.

Illustration of Def. 1.5.85
($\mathbf{y} \mathrel{\widehat{=}}$ exact result for exact data $\mathbf{x}$)

Terminology:
Def. 1.5.85 introduces stability in the sense of backward error analysis

▷

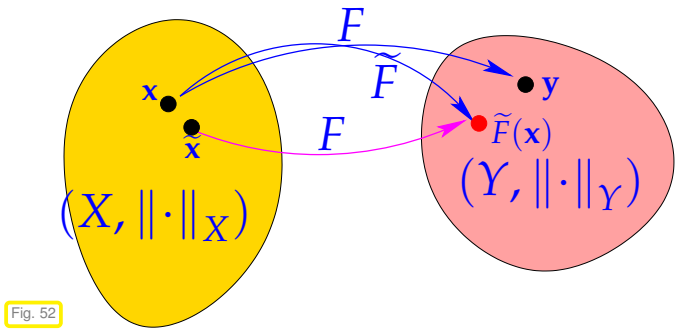


Fig. 52

Sloppily speaking, the impact of roundoff $(*)$ on a *stable algorithm* is of the same order of magnitude as the effect of the inevitable perturbations due to rounding the input data.

➢ For stable algorithms roundoff errors are "harmless".

$(*)$ In some cases the definition of $\widetilde{F}$ will also involve some approximations as in Ex. 1.5.65. Then the above statement also includes approximation errors.

Example 1.5.86 (Testing stability of matrix×vector multiplication)

Assume you are given a black box implementation of a function

```
VectorXd mvmult(const MatrixX &A,const VectorXd &x)
```

that purports to provide a stable implementation of $\mathbf{Ax}$ for $\mathbf{A} \in \mathbb{K}^{m,n}$, $\mathbf{x} \in \mathbb{K}^n$, *cf.* Ex. 1.5.68. How can we verify this claim for particular data. Both, $\mathbb{K}^{m,n}$ and $\mathbb{K}^n$ are equipped with the Euclidean norm.

The task is, given $\mathbf{y} \in \mathbb{K}^n$ as returned by the function, to find conditions on $\mathbf{y}$ that ensure the existence of a $\widetilde{\mathbf{A}} \in \mathbb{K}^{m,n}$ such that

$$\widetilde{\mathbf{A}}\mathbf{x} = \mathbf{y} \quad \text{and} \quad \left\|\widetilde{\mathbf{A}} - \mathbf{A}\right\|_2 \leq Cmn \,\mathrm{EPS}\, \|\mathbf{A}\|_2 \,, \tag{1.5.87}$$

for a small constant ≈ 1.

In fact we can choose (easy computation)

$$\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{z}\mathbf{x}^T \quad , \quad \mathbf{z} := \frac{\mathbf{y} - \mathbf{A}\mathbf{x}}{\|\mathbf{x}\|_2^2} \in \mathbb{K}^m \, ,$$

and we find

$$\left\|\widetilde{\mathbf{A}} - \mathbf{A}\right\|_2 = \left\|\mathbf{z}\mathbf{x}^T\right\|_2 = \sup_{\mathbf{w}\in\mathbb{K}^n\setminus\{\mathbf{0}\}} \frac{\mathbf{x}\cdot\mathbf{w}\|\mathbf{z}\|_2}{\|\mathbf{w}\|_2} \leq \|\mathbf{x}\|_2\|\mathbf{z}\|_2 = \frac{\|\mathbf{y}-\mathbf{A}\mathbf{x}\|_2}{\|\mathbf{x}\|_2} \, .$$

Hence, in principle stability of an algorithm for computing $\mathbf{A}\mathbf{x}$ is confirmed, if for every $\mathbf{x} \in X$ the computed result $\mathbf{y} = \mathbf{y}(\mathbf{x})$ satisfies

$$\|\mathbf{y} - \mathbf{A}\mathbf{x}\|_2 \leq C\, mn\, \mathrm{EPS}\, \|\mathbf{x}\|_2\|\mathbf{A}\|_2 \, ,$$

with a small constant $C > 0$ independent of data and problem size.

Remark 1.5.88 (Numerical stability and sensitive dependence on data)

A problem shows sensitive dependence on the data, if small perturbations of input data lead to large perturbations of the output. Such problems are also called ill-conditioned. For such problems stability of an algorithm is easily accomplished.

Fig. 53

Example: The problem is the prediction of the position of the billard ball after ten bounces given the initial position, velcity, and spin.

It is well known, that tiny changes of the initial conditions can shift the final location of the ball to virtually any point on the table: the billard problem is chaotic.

Hence, a stable algorithm for its solution may just output a *fixed* or *random* position without even using the initial conditions!

Learning Outcomes

Principal take-home knowledge and skills from this chapter:

- Knowledge about the syntax of fundamental operations on matrices and vectors in EIGEN.
- Understanding of the concepts of computational effort/cost and asymptotic complexity in numerics.
- Awareness of the asymptotic complexity of basic linear algebra operations
- Ability to determine the (asymptotic) computational effort for a concrete (numerical linear algebra) algorithm.
- Ability to manipulate simple expressions involving matrices and vectors in order to reduce the computational cost for their evaluation.

Chapter 2

Direct Methods for Linear Systems of Equations

Contents

2.1 Preface

(2.1.1) The problem: solving a linear system

Given : square matrix $\mathbf{A} \in \mathbb{K}^{n,n}$, vector $\mathbf{b} \in \mathbb{K}^n$, $n \in \mathbb{N}$

Sought : solution vector $\mathbf{x} \in \mathbb{K}^n$: $\boxed{\mathbf{A}\mathbf{x} = \mathbf{b}}$ $\leftarrow$ (square) linear system of equations (LSE)

(Formal problem mapping $(\mathbf{A}, \mathbf{b}) \mapsto \mathbf{A}^{-1}\mathbf{b}$)

(Terminology: $\mathbf{A} \triangleq$ system matrix/coefficient matrix, $\mathbf{b} \triangleq$ right hand side vector)

Linear systems with rectangular system matrices $\mathbf{A} \in \mathbb{K}^{m,n}$, called "overdetermined" for $m > n$, and

"underdetermined" for $m < n$ will be treated in Chapter 3.

(2.1.2) LSE: key components of mathematical models in many fields

Linear systems of equations are ubiquitous in computational science: they are encountered

- with discrete linear models in network theory (see Ex. 2.1.3), control, statistics;
- in the case of *discretized* boundary value problems for ordinary and partial differential equations (→ course "Numerical methods for partial differential equations", 4th semester);
- as a result of linearization (e.g, "Newton's method" → Sect. 8.4).

Example 2.1.3 (Nodal analysis of (linear) electric circuit [?, Sect. 4.7.1])

Now we study a very important application of numerical simulation, where (large, sparse) linear systems of equations play a central role: Numerical circuit analysis. We begin with *linear circuits* in the *frequency domain*, which are directly modelled by complex linear systems of equations. Later we tackle circuits with non-linear elements, see Ex. 8.0.1, and, finally, will learn about numerical methods for computing the transient (time-dependent) behavior of circuits, see Ex. 11.1.13.

Modeling of simple linear circuits takes only elementary physical laws as covered in any introductory course of physics (or even in secondary school physics). There is no sophisticated physics or mathematics involved.

Node (*ger.:* Knoten) ≙ junction of wires

☞ *number* nodes $1, \ldots, n$

I_{kj}: current from node $k \to$ node j, $I_{kj} = -I_{jk}$

Kirchhoff current law (KCL) : sum of node currents $= 0$:

$$\forall k \in \{1, \ldots, n\}: \quad \sum_{j=1}^{n} I_{kj} = 0 \,. \qquad (2.1.4)$$

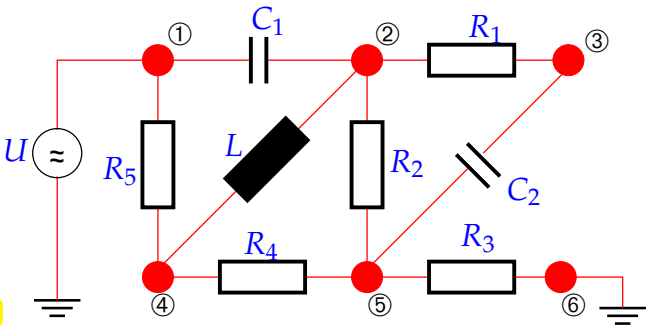


Fig. 54

Unknowns: nodal potentials U_k, $k = 1, \ldots, n$.
(some may be known: grounded nodes: ⑥ in Fig. 54, voltage sources: ① in Fig. 54)

Constitutive relations for circuit elements: (in *frequency domain* with angular frequency $\omega > 0$):

- Ohmic resistor: $I = \dfrac{U}{R}$, $[R] = 1\mathrm{VA}^{-1}$
- capacitor: $I = \imath\omega C U$, capacitance $[C] = 1\mathrm{AsV}^{-1}$
- coil/inductor : $I = \dfrac{U}{\imath\omega L}$, inductance $[L] = 1\mathrm{VsA}^{-1}$

➤ $$I_{kj} = \begin{cases} R^{-1}(U_k - U_j) \,, \\ \imath\omega C(U_k - U_j) \,, \\ -\imath\omega^{-1}L^{-1}(U_k - U_j) \,. \end{cases}$$

✎ notation: $\imath$ ≙ imaginary unit "$\imath := \sqrt{-1}$", $\imath = \exp(\imath\pi/2)$

Here we face the special case of a linear circuit: all relationships between branch currents and voltages are of the form

$$I_{kj} = \alpha_{kj}(U_k - U_j) \quad \text{with} \quad \alpha_{kj} \in \mathbb{C} \,. \tag{2.1.5}$$

The concrete value of α_{kj} is determined by the circuit element connecting node k and node j.

These constitutive relations are derived by assuming a harmonic time-dependence of all quantities, which is termed circuit analysis in the frequency domain (AC-mode).

$$\text{voltage:} \quad u(t) = \mathrm{Re}\{U\exp(\imath\omega t)\} \quad , \quad \text{current:} \quad i(t) = \mathrm{Re}\{I\exp(\imath\omega t)\} \,. \tag{2.1.6}$$

Here $U, I \in \mathbb{C}$ are called complex amplitudes. This implies for temporal derivatives (denoted by a dot):

$$\frac{du}{dt}(t) = \mathrm{Re}\{\imath\omega U\exp(\imath\omega t)\} \quad , \quad \frac{di}{dt}(t) = \mathrm{Re}\{\imath\omega I\exp(\imath\omega t)\} \,. \tag{2.1.7}$$

For a capacitor the total charge is proportional to the applied voltage:

$$q(t) = Cu(t) \overset{i(t)=\dot{q}(t)}{\Rightarrow} i(t) = C\dot{u}(t) \,.$$

For a coil the voltage is proportional to the rate of change of current: $u(t) = L\dot{i}(t)$. Combined with (2.1.6) and (2.1.7) this leads to the above constitutive relations.

Constitutive relations + (2.1.4) ► linear system of equations:

$$\begin{aligned}
&② : \quad \imath\omega C_1(U_2 - U_1) + R_1^{-1}(U_2 - U_3) - \imath\omega^{-1}L^{-1}(U_2 - U_4) + R_2^{-1}(U_2 - U_5) = 0 \,,\\
&③ : \quad R_1^{-1}(U_3 - U_2) + \imath\omega C_2(U_3 - U_5) = 0 \,,\\
&④ : \quad R_5^{-1}(U_4 - U_1) - \imath\omega^{-1}L^{-1}(U_4 - U_2) + R_4^{-1}(U_4 - U_5) = 0 \,,\\
&⑤ : \quad R_2^{-1}(U_5 - U_2) + \imath\omega C_2(U_5 - U_3) + R_4^{-1}(U_5 - U_4) + R_3^{-1}(U_5 - U_6) = 0 \,,\\
&\qquad U_1 = U \quad , \quad U_6 = 0 \,.
\end{aligned}$$

No equations for nodes ① and ⑥, because these nodes are connected to the "outside world" so that the Kirchhoff current law (2.1.4) does not hold (from a local perspective). This is fitting, because the voltages in these nodes are known anyway.

▼

$$\begin{pmatrix}
\imath\omega C_1 + \frac{1}{R_1} - \frac{i}{\omega L} + \frac{1}{R_2} & -\frac{1}{R_1} & \frac{i}{\omega L} & -\frac{1}{R_2} \\
-\frac{1}{R_1} & \frac{1}{R_1} + \imath\omega C_2 & 0 & -\imath\omega C_2 \\
\frac{i}{\omega L} & 0 & \frac{1}{R_5} - \frac{i}{\omega L} + \frac{1}{R_4} & -\frac{1}{R_4} \\
-\frac{1}{R_2} & -\imath\omega C_2 & -\frac{1}{R_4} & \frac{1}{R_2} + \imath\omega C_2 + \frac{1}{R_4} + R_3^{-1}
\end{pmatrix}
\begin{pmatrix} U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix}
=
\begin{pmatrix} \imath\omega C_1 U \\ 0 \\ \frac{1}{R_5} U \\ 0 \end{pmatrix}$$

This is a linear system of equations with *complex* coefficients: $\mathbf{A} \in \mathbb{C}^{4,4}$, $\mathbf{b} \in \mathbb{C}^4$. For the algorithms to be discussed below this does not matter, because they work alike for real and complex numbers.

2.2 Theory: Linear systems of equations

2.2.1 Existence and uniqueness of solutions

Known from linear algebra [**?**, Sect. 1.2], [**?**, Sect. 1.3]:

Definition 2.2.1. Invertible matrix → [?, Sect. 2.3]

$$\mathbf{A} \in \mathbb{K}^{n,n} \quad \begin{matrix}\text{invertible /}\\ \text{regular}\end{matrix} \quad :\Leftrightarrow \quad \exists_1 \mathbf{B} \in \mathbb{K}^{n,n}: \quad \mathbf{AB} = \mathbf{BA} = \mathbf{I}\,.$$

$\mathbf{B} \hat{=}$ inverse of $\mathbf{A}$, (✎ notation $\mathbf{B} = \mathbf{A}^{-1}$)

New, recall a few concepts from linear algebra needed to state criteria for the invertibility of a matrix.

Definition 2.2.2. Image space and kernel of a matrix

Given $\mathbf{A} \in \mathbb{K}^{m,n}$, the range/image (space) of $\mathbf{A}$ is the subspace of $\mathbb{K}^m$ spanned by the columns of $\mathbf{A}$

$$\mathcal{R}(\mathbf{A}) := \{\mathbf{Ax},\ \mathbf{x} \in \mathbb{K}^n\} \subset \mathbb{R}^m\,.$$

The kernel/nullspace of $\mathbf{A}$ is

$$\mathcal{N}(\mathbf{A}) := \{\mathbf{z} \in \mathbb{R}^n :\ \mathbf{Az} = \mathbf{0}\}\,.$$

Definition 2.2.3. Rank of a matrix → [?, Sect. 2.4], [?, Sect. 1.5]

The rank of a matrix $\mathbf{M} \in \mathbb{K}^{m,n}$, denoted by $\operatorname{rank}(\mathbf{M})$, is the maximal number of linearly independent rows/columns of $\mathbf{M}$.
Equivalently, $\operatorname{rank}(\mathbf{A}) = \dim \mathcal{R}(\mathbf{A})$.

Theorem 2.2.4. Criteria for invertibility of matrix → [?, Sect. 2.3 & Cor. 3.8]

A square matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ is invertible/regular if one of the following equivalent conditions is satisfied:

1. $\exists \mathbf{B} \in \mathbb{K}^{n,n}$: $\mathbf{BA} = \mathbf{AB} = \mathbf{I}$,
2. $\mathbf{x} \mapsto \mathbf{Ax}$ *defines an endomorphism of* $\mathbb{K}^n$,
3. *the columns of* $\mathbf{A}$ *are linearly independent (full column rank),*
4. *the rows of* $\mathbf{A}$ *are linearly independent (full row rank),*
5. $\det \mathbf{A} \neq 0$ *(non-vanishing determinant),*
6. $\operatorname{rank}(\mathbf{A}) = n$ *(full rank).*

(2.2.5) Solution of a LSE as a "problem"

Linear algebra give us a *formal* way to denote solution of LSE:

$$\mathbf{A} \in \mathbb{K}^{n,n} \text{ regular} \quad \& \quad \mathbf{A}\mathbf{x} = \mathbf{b} \quad \Rightarrow \quad \mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \,.$$

inverse matrix

Now recall our notion of "problem" from § 1.5.67 as a function F mapping data in a data space X to a result in a result space Y. Concretely, for $n \times n$ linear systems of equations:

$$F : \begin{cases} X := \mathbb{K}^{n,n}_* \times \mathbb{K}^n & \to \quad Y := \mathbb{K}^n \\ (\mathbf{A}, \mathbf{b}) & \mapsto \quad \mathbf{A}^{-1}\mathbf{b} \end{cases}$$

✎ notation: (open) set of regular matrices $\subset \mathbb{K}^{n,n}$:

$$\mathbb{K}^{n,n}_* := \{\mathbf{A} \in \mathbb{K}^{n,n} : \mathbf{A} \text{ regular/invertible} \to \text{Def. 2.2.1}\} \,.$$

Remark 2.2.6 (The inverse matrix and solution of a LSE)

EIGEN: inverse of a matrix $\mathbf{A}$ available through `A.inverse()`

Always avoid computing the inverse of a matrix (which can almost always be avoided)!
In particular, never ever even contemplate using `x = A.inverse()*b` to solve the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, *cf.* Exp. 2.4.13. The next sections present a sound way to do this.

2.2.2 Sensitivity of linear systems

The Sensitivity of a problem (for given data) gauges
the impact of small perturbations of the data on the result.

Before we examine sensitivity for linear systems of equations, we look at the simpler problem of matrix×vector multiplication.

Example 2.2.7 (Sensitivity of linear mappings)

For a *fixed* given regular $\mathbf{A} \in \mathbb{K}^{n,n}$ we study the problem map

$$F : \mathbb{K}^n \to \mathbb{K}^n \quad , \quad \mathbf{x} \mapsto \mathbf{A}\mathbf{x} \,,$$

that is, now we consider only the vector $\mathbf{x}$ as data.

Goal: Estimate relative perturbations in $F(\mathbf{x})$ due to relative perturbations in $\mathbf{x}$.

We assume that $\mathbb{K}^n$ is equipped with *some* vector norm ($\to$ Def. 1.5.70) and we use the induced matrix norm ($\to$ Def. 1.5.76) on $\mathbb{K}^{n,n}$. Using linearity and the elementary estimate $\|\mathbf{M}\mathbf{x}\| \le \|\mathbf{M}\|\|\mathbf{x}\|$, which is a direct consequence of the definition of an induced matrix norm, we obtain

$$\begin{aligned} \mathbf{A}\mathbf{x} = \mathbf{y} \quad &\Rightarrow \quad \|\mathbf{x}\| \le \left\|\mathbf{A}^{-1}\right\| \|\mathbf{y}\| \\ \mathbf{A}(\mathbf{x} + \Delta\mathbf{x}) = \mathbf{y} + \Delta\mathbf{y} \quad &\Rightarrow \quad \mathbf{A}\Delta\mathbf{x} = \Delta\mathbf{y} \quad \Rightarrow \quad \|\Delta\mathbf{y}\| \le \|\mathbf{A}\|\|\Delta\mathbf{x}\| \end{aligned}$$

$$\Rightarrow \quad \frac{\|\Delta \mathbf{y}\|}{\|\mathbf{y}\|} \leq \frac{\|\mathbf{A}\|\|\Delta \mathbf{x}\|}{\|\mathbf{A}^{-1}\|^{-1}\|\mathbf{x}\|} = \|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\|\frac{\|\Delta \mathbf{x}\|}{\|\mathbf{x}\|} . \tag{2.2.8}$$

relative perturbation in result — relative perturbation in data

We have found that the quantity $\|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\|$ bounds amplification of relative errors in the argument vector in a matrix×vector-multiplication with the matrix $\mathbf{A}$.

Now we study the sensitivity of the problem of finding the solution of a linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$ regular, $\mathbf{b} \in \mathbb{R}$, see § 2.1.1. We write $\widetilde{\mathbf{x}}$ for the solution of the perturbed linear system.

Question: To what extent do perturbations in the data $\mathbf{A}$, $\mathbf{b}$ cause a

$$\text{(normwise) relative error:} \qquad \epsilon_r := \frac{\|\mathbf{x} - \widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} .$$

($\|\cdot\| \mathrel{\hat{=}}$ suitable vector norm, e.g., maximum norm $\|\cdot\|_\infty$)

Perturbed linear system:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \leftrightarrow \quad (\mathbf{A} + \Delta\mathbf{A})\widetilde{\mathbf{x}} = \mathbf{b} + \Delta\mathbf{b} \quad \blacktriangleright \quad (\mathbf{A} + \Delta\mathbf{A})(\widetilde{\mathbf{x}} - \mathbf{x}) = \Delta\mathbf{b} - \Delta\mathbf{A}\mathbf{x} . \tag{2.2.9}$$

Theorem 2.2.10. Conditioning of LSEs → **[?, Thm. 3.1]**

If $\mathbf{A}$ *regular,* $\|\Delta\mathbf{A}\| < \left\|\mathbf{A}^{-1}\right\|^{-1}$ *and* (2.2.9)*, then*

(i) $\mathbf{A} + \Delta\mathbf{A}$ *is regular/invertible,*

(ii)

$$\frac{\|\mathbf{x} - \widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\left\|\mathbf{A}^{-1}\right\|\|\mathbf{A}\|}{1 - \left\|\mathbf{A}^{-1}\right\|\|\mathbf{A}\|\|\Delta\mathbf{A}\|/\|\mathbf{A}\|}\left(\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|}\right) .$$

relative error of data — *relative perturbations*

The proof is based on the following fundamental result:

Lemma 2.2.11 (Perturbation lemma)**.** → *[?, Thm. 1.5]*

$$\mathbf{B} \in \mathbb{R}^{n,n},\ \|\mathbf{B}\| < 1 \quad \Rightarrow \quad \mathbf{I} + \mathbf{B} \text{ regular} \quad \wedge \quad \left\|(\mathbf{I} + \mathbf{B})^{-1}\right\| \leq \frac{1}{1 - \|\mathbf{B}\|}.$$

Proof. △-inequality ➢ $\|(\mathbf{I} + \mathbf{B})\mathbf{x}\| \geq (1 - \|\mathbf{B}\|)\|\mathbf{x}\|, \forall \mathbf{x} \in \mathbb{R}^n$ ➢ $\mathbf{I} + \mathbf{B}$ regular.

$$\blacktriangleright \left\|(\mathbf{I} + \mathbf{B})^{-1}\right\| = \sup_{\mathbf{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\|(\mathbf{I} + \mathbf{B})^{-1}\mathbf{x}\|}{\|\mathbf{x}\|} = \sup_{\mathbf{y} \in \mathbb{R}^n \setminus \{0\}} \frac{\|\mathbf{y}\|}{\|(\mathbf{I} + \mathbf{B})\mathbf{y}\|} \leq \frac{1}{1 - \|\mathbf{B}\|}$$

Proof (of Thm. 2.2.10) Lemma 2.2.11 ➢ $\left\|(\mathbf{A} + \Delta\mathbf{A})^{-1}\right\| \leq \frac{\left\|\mathbf{A}^{-1}\right\|}{1 - \|\mathbf{A}^{-1}\Delta\mathbf{A}\|}$ & (2.2.9)

$$\Rightarrow \|\Delta\mathbf{x}\| \leq \frac{\left\|\mathbf{A}^{-1}\right\|}{1 - \|\mathbf{A}^{-1}\Delta\mathbf{A}\|}(\|\Delta\mathbf{b}\| + \|\Delta\mathbf{A}\mathbf{x}\|) \leq \frac{\left\|\mathbf{A}^{-1}\right\|\|\mathbf{A}\|}{1 - \left\|\mathbf{A}^{-1}\right\|\|\Delta\mathbf{A}\|}\left(\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{A}\|\|\mathbf{x}\|} + \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|}\right)\|\mathbf{x}\| .$$

Note that the term $\|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\|$ occurs frequently. Therefore it has been given a special name:

Definition 2.2.12. Condition (number) of a matrix

Condition (number) of a matrix $\mathbf{A} \in \mathbb{R}^{n,n}$: $\quad \operatorname{cond}(\mathbf{A}) := \left\|\mathbf{A}^{-1}\right\|\|\mathbf{A}\|$

Note: $\operatorname{cond}(\mathbf{A})$ depends on the matrix norm $\|\cdot\|$!

Rewriting estimate of Thm. 2.2.10 with $\Delta\mathbf{b} = 0$,

$$\epsilon_r := \frac{\|\mathbf{x} - \widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\operatorname{cond}(\mathbf{A})\delta_A}{1 - \operatorname{cond}(\mathbf{A})\delta_A}\,, \quad \delta_A := \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|}\,. \tag{2.2.13}$$

From (2.2.13) we conclude important messsages of $\operatorname{cond}(\mathbf{A})$:

- If $\operatorname{cond}(\mathbf{A}) \gg 1$, *small perturbations* in $\mathbf{A}$ can lead to *large relative errors* in the solution of the LSE.
- If $\operatorname{cond}(\mathbf{A}) \gg 1$, a stable algorithm ($\rightarrow$ Def. 1.5.85) can produce solutions with large relative error !

Recall Thm. 2.2.10: for regular $\mathbf{A} \in \mathbb{K}^{n,n}$, small $\Delta\mathbf{A}$, generic vector/matrix norm $\|\cdot\|$

$$\begin{array}{c}\mathbf{A}\mathbf{x} = \mathbf{b} \\ (\mathbf{A} + \Delta\mathbf{A})\widetilde{\mathbf{x}} = \mathbf{b} + \Delta\mathbf{b}\end{array} \quad \Rightarrow \quad \frac{\|\mathbf{x} - \widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \leq \frac{\operatorname{cond}(\mathbf{A})}{1 - \operatorname{cond}(\mathbf{A})\|\Delta\mathbf{A}\|/\|\mathbf{A}\|}\left(\frac{\|\Delta\mathbf{b}\|}{\|\mathbf{b}\|} + \frac{\|\Delta\mathbf{A}\|}{\|\mathbf{A}\|}\right)\,. \tag{2.2.14}$$

$\operatorname{cond}(\mathbf{A}) \gg 1$ ➢ small relative changes of data $\mathbf{A}, \mathbf{b}$ *may* effect huge relative changes in solution.

$\operatorname{cond}(\mathbf{A})$ indicates sensitivity of "LSE problem" $(\mathbf{A}, \mathbf{b}) \mapsto \mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$
(as "amplification factor" of (worst-case) relative perturbations in the data $\mathbf{A}, \mathbf{b}$).

Terminology:

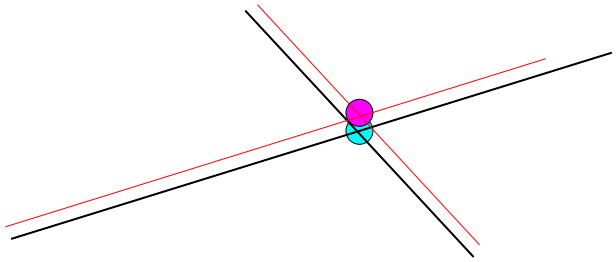
Small changes of data $\Rightarrow$ small perturbations of result : well-conditioned problem
Small changes of data $\Rightarrow$ large perturbations of result : ill-conditioned problem

Note: sensitivity gauge depends on the chosen norm **!**

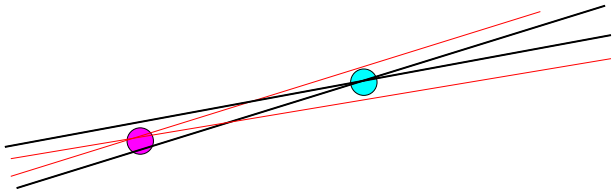
Example 2.2.15 (Intersection of lines in 2D)

Solving a 2×2 linear system of equations amounts to finding the intersection of two lines in the coordinate plane. This relationship allows a geometric view of "sensitivity of a linear system":

In distance metric (Euclidean vector norm):



nearly orthogonal intersection: well-conditioned



glancing intersection: ill-conditioned

Hessian normal form of line #i, $i = 1,2$:

$$L_i = \{\mathbf{x} \in \mathbb{R}^2 : \mathbf{x}^T \mathbf{n}_i = d_i\} , \quad \mathbf{n}_i \in \mathbb{R}^2, d_i \in \mathbb{R} .$$

LSE for finding intersection: $\underbrace{\begin{bmatrix} \mathbf{n}_1^T \\ \mathbf{n}_2^T \end{bmatrix}}_{=:\mathbf{A}} \mathbf{x} = \underbrace{\begin{bmatrix} d_1 \\ d_2 \end{bmatrix}}_{=:\mathbf{b}}$,

$\mathbf{n}_i \hat{=}$ (unit) direction vectors, $d_i \hat{=}$ distance to origin.

The following code investigates condition numbers for the matrix $\begin{bmatrix} 1 & \cos\varphi \\ 0 & \sin\varphi \end{bmatrix}$ that can arise when computing the intersection of two lines enclosing the angle φ.

C++11-code 2.2.16: condition numbers of 2×2 matrices ↪ GITLAB

```
2  VectorXd phi = VectorXd::LinSpaced(50,M_PI/200,M_PI/2);
3  MatrixXd res(phi.size(), 3);
4  Matrix2d A; A(0,0) = 1; A(1,0) = 0;
5  for(int i = 0; i < phi.size(); ++i){
6      A(0,1) = std::cos(phi(i)); A(1,1) = std::sin(phi(i));
7      // L2 condition number is the quotient of the maximal
8      // and minimal singular value of A
9      JacobiSVD<MatrixXd> svd(A);
10     double C2 = svd.singularValues()(0) / //
11     svd.singularValues()(svd.singularValues().size()-1);
12     // L-infinity condition number
13     double Cinf=A.inverse().cwiseAbs().rowwise().sum().maxCoeff()*
           A.cwiseAbs().rowwise().sum().maxCoeff(); //
14     res(i,0) = phi(i); res(i,1) = C2; res(i,2) = Cinf;
15 }
16 // Plot
17 // ...
```

In Line 10 we compute the condition number of $\mathbf{A}$ with respect to the Euclidean vector norm using special EIGEN built-in functions.

Line 13 evaluated the condition number of a matrix for the maximum norm, recall Ex. 1.5.78.

We clearly observe a blow-up of $\mathrm{cond}(\mathbf{A})$ (with respect to the Euclidean vector norms) as the angle enclosed by the two lines shrinks.

This corresponds to a large sensitivity of the location of the intersection point in the case of glancing incidence.

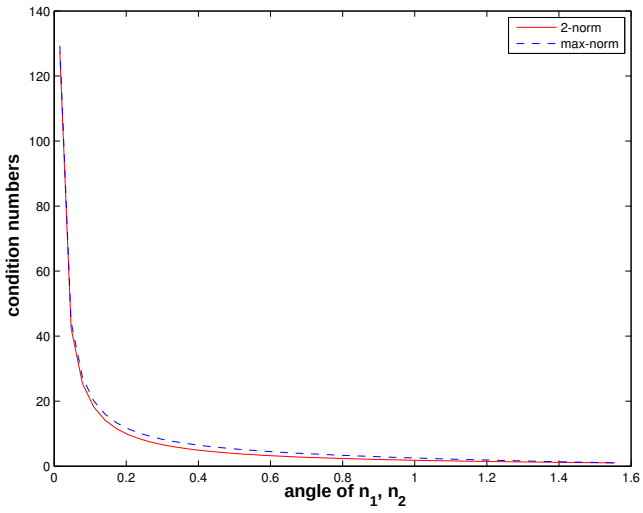


Fig. 55

Heuristics for predicting large $\mathrm{cond}(\mathbf{A})$

$\mathrm{cond}(\mathbf{A}) \gg 1 \quad \leftrightarrow \quad$ columns/rows of $\mathbf{A}$ "almost linearly dependent"

2.3 Gaussian Ellimination

2.3.1 Basic algorithm

! Exceptional feature of linear systems of equations (LSE):
☞ "exact" solution computable with finitely many elementary operations

Algorithm: Gaussian elimination ($\rightarrow$ secondary school, linear algebra,)

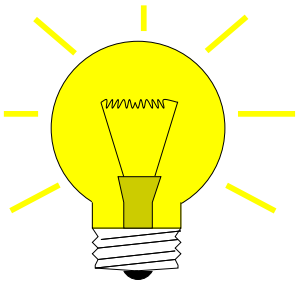
Familiarity with the algorithm of Gaussian elimination for a square linear system of equations will be taken for granted.

Supplementary reading. In case you cannot remember the main facts about Gaussian elimination, very detailed accounts and examples can be found in

- M. Gutknecht's lecture notes [**?**, Ch. 1],
- the textbook by Nipp & Stoffer [**?**, Ch. 1],
- the numerical analysis text by Quarteroni et al. [**?**, Sects. 3.2 & 3.3],
- the textbook by Ascher & Greif [**?**, Sect. 5.1],

and, to some extend, below, see Ex. 2.3.1.

Wikipedia: Although the method is named after mathematician **Carl Friedrich Gauss**, the earliest presentation of it can be found in the important Chinese mathematical text *Jiuzhang suanshu* or The Nine Chapters on the Mathematical Art, dated approximately 150 B.C., and commented on by **Liu Hui** in the 3rd century.



Idea: transformation to "simpler", but equivalent LSE by means of successive (invertible) *row transformations*

Ex. 1.3.13: row transformations $\leftrightarrow$ left-multiplication with transformation matrix

Obviously, left multiplication with a regular matrix does not affect the solution of an LSE: for any *regular* $\mathbf{T} \in \mathbb{K}^{n,n}$

$$\mathbf{Ax} = \mathbf{b} \quad \Rightarrow \quad \mathbf{A}'\mathbf{x} = \mathbf{b}' \quad \text{, if} \quad \mathbf{A}' = \mathbf{TA}, \mathbf{b}' = \mathbf{Tb} \; .$$

So we may try to convert the linear system of equations to a form that can be solved more easily by multiplying with regular matrices from left, which boils down to applying row transformations. A suitable target format is a diagonal linear system of equations, for which all equations are completely decoupled. This is the gist of Gaussian elimination.

Example 2.3.1 (Gaussian elimination)

① (Forward) elimination:

$$\begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \\ -3 \end{pmatrix} \quad \longleftrightarrow \quad \begin{array}{rcrcrcr} x_1 & + & x_2 & & & = & 4 \\ 2x_1 & + & x_2 & - & x_3 & = & 1 \\ 3x_1 & - & x_2 & - & x_3 & = & -3 \end{array} \; .$$

$$\left[\begin{array}{ccc} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{array}\right] \left[\begin{array}{c} 4 \\ 1 \\ -3 \end{array}\right] \rightarrow \left[\begin{array}{ccc} \mathbf{1} & 1 & 0 \\ 0 & -1 & -1 \\ 3 & -1 & -1 \end{array}\right] \left[\begin{array}{c} 4 \\ -7 \\ -3 \end{array}\right] \rightarrow \left[\begin{array}{ccc} \mathbf{1} & 1 & 0 \\ 0 & -1 & -1 \\ 0 & -4 & -1 \end{array}\right] \left[\begin{array}{c} 4 \\ -7 \\ -15 \end{array}\right]$$

$$\rightarrow \underbrace{\left[\begin{array}{ccc} 1 & 1 & 0 \\ 0 & \mathbf{-1} & -1 \\ 0 & 0 & 3 \end{array}\right]}_{=\mathbf{U}} \left[\begin{array}{c} 4 \\ -7 \\ 13 \end{array}\right]$$

■ = pivot row, pivot element **bold**.

◤ transformation of LSE to upper triangular form

② Solve by back substitution: back substitution = Rücksubstitution

$$\begin{array}{rcrcrcr} x_1 & + & x_2 & & & = & 4 \\ & - & x_2 & - & x_3 & = & -7 \\ & & & & 3x_3 & = & 13 \end{array} \quad \Rightarrow \quad \begin{array}{rcl} x_3 & = & \frac{13}{3} \\ x_2 & = & 7 - \frac{13}{3} = \frac{8}{3} \\ x_1 & = & 4 - \frac{8}{3} = \frac{4}{3} \; . \end{array}$$

More detailed examples: [**?**, Sect. 1.1], [**?**, Sect. 1.1].

More general:

$$\begin{array}{ccccccccc} a_{11}\,x_1 & + & a_{12}\,x_2 & + & \cdots & + & a_{1n}\,x_n & = & b_1 \\ a_{21}\,x_1 & + & a_{22}\,x_2 & + & \cdots & + & a_{2n}\,x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots & \vdots & \vdots \\ \vdots & & \vdots & & \vdots & & \vdots & \vdots & \vdots \\ a_{n1}\,x_1 & + & a_{n2}\,x_2 & + & \cdots & + & a_{nn}\,x_n & = & b_n \end{array}$$

- i-th row - $l_{i1}\cdot$ 1st row (pivot row), $l_{i1} := a_{i1}/a_{11}$, $i = 2,\ldots,n$

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ & & a_{22}^{(1)}x_2 & + & \cdots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\ & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & a_{n2}^{(1)}x_2 & + & \cdots & + & a_{nn}^{(1)}x_n & = & b_n^{(1)} \end{array}$$

with

$$\begin{aligned} a_{ij}^{(1)} &= a_{ij} - a_{1j}\,l_{i1}, \qquad i,j = 2,\ldots,n\,, \\ b_i^{(1)} &= b_i - b_1\,l_{i1}, \qquad i = 2,\ldots,n\,. \end{aligned}$$

- i-th row - $l_{i1}\cdot$ 2nd row (pivot row), $l_{i2} := a_{i2}^{(1)}/a_{22}^{(1)}$, $i = 3,\ldots,n$.

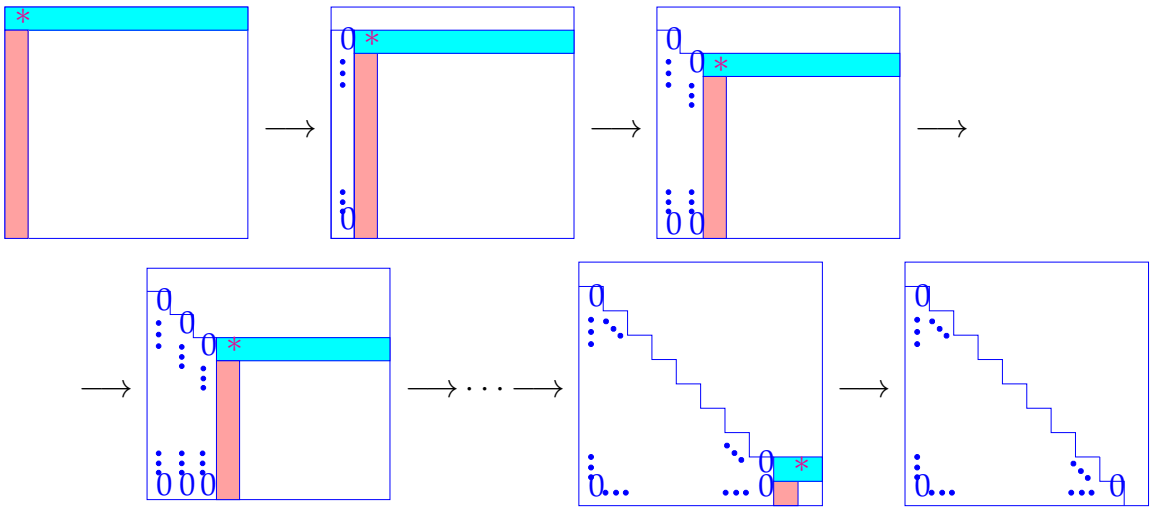
$$\begin{array}{ccccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ & & a_{22}^{(1)}x_2 & + & a_{23}^{(1)}x_3 & + & \cdots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\ & & & & a_{33}^{(2)}x_3 & + & \cdots & + & a_{3n}^{(2)}x_n & = & b_3^{(2)} \\ & & & & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & & & a_{n3}^{(2)}x_3 & + & \cdots & + & a_{nn}^{(2)}x_n & = & b_n^{(2)} \end{array}$$

➤ After $n-1$ steps: linear systems of equations in upper triangular form

$$\begin{array}{ccccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ & & a_{22}^{(1)}x_2 & + & a_{23}^{(1)}x_3 & + & \cdots & + & a_{2n}^{(1)}x_n & = & b_2^{(1)} \\ & & & & a_{33}^{(2)}x_3 & + & \cdots & + & a_{3n}^{(2)}x_n & = & b_3^{(2)} \\ & & & & & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ & & & & & & \ddots & \ddots & \vdots & \vdots & \vdots \\ & & & & & & & & a_{nn}^{(n-1)}x_n & = & b_n^{(n-1)} \end{array}$$

Terminology: $a_{11}, a_{22}^{(1)}, a_{33}^{(2)}, \ldots, a_{n-1,n-1}^{(n-2)}$ = pivots/pivot elements

Graphical depiction:



$*$ $\hat{=}$ pivot (necessarily $\neq 0$ ➙ **here: assumption**),  = pivot row

In k-th step (starting from $\mathbf{A} \in \mathbb{K}^{n,n}$, $1 \le k < n$, pivot row $\mathbf{a}_{k.}^{\top}$):

transformation: $\mathbf{A}\mathbf{x} = \mathbf{b}$ ➤ $\mathbf{A}'\mathbf{x} = \mathbf{b}'$.

with

$$a'_{ij} := \begin{cases} a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj} & \text{for } k < i, j \le n \,, \\ 0 & \text{for } k < i \le n, j = k \,, \\ a_{ij} & \text{else,} \end{cases} \quad b'_i := \begin{cases} b_i - \frac{a_{ik}}{a_{kk}} b_k & \text{for } k < i \le n \,, \\ b_i & \text{else.} \end{cases} \tag{2.3.2}$$

multipliers l_{ik}

(2.3.3) Gaussian elimination: algorithm

Here we give a direct EIGEN implementation of Gaussian elimination for LSE $\mathbf{Ax} = \mathbf{b}$ (grossly inefficient!).

C++11 code 2.3.4: Solving LSE $\mathbf{Ax} = \mathbf{b}$ with Gaussian elimination ↪ GITLAB

```
2  //!  Gauss elimination without pivoting, x = A\b
3  //!  A must be an n×n-matrix, b an n-vector
4  //!  The result is returned in x
5  void gausselimsolve(const MatrixXd &A, const VectorXd& b,
6                      VectorXd& x) {
7    int n = A.rows();
8    MatrixXd Ab(n,n+1); // Augmented matrix [A,b]
9    Ab << A, b; //
10   // Forward elimination (cf. step ① in Ex. 2.3.1)
11   for(int i = 0; i < n-1; ++i) {
12     double pivot = Ab(i,i);
13     for(int k = i+1; k < n; ++k) {
14       double fac = Ab(k,i)/pivot;
15       Ab.block(k,i+1,1,n-i)-= fac * Ab.block(i,i+1,1,n-i); //
16     }
17   }
18   // Back substitution (cf. step ② in Ex. 2.3.1)
19   Ab(n-1,n) = Ab(n-1,n) / Ab(n-1,n-1);
20   for(int i = n-2; i >= 0; --i) {
21     for(int l = i+1; l < n; ++l) Ab(i,n) -= Ab(l,n)*Ab(i,l);
22     Ab(i,n) /= Ab(i,i);
23   }
24   x = Ab.rightCols(1); //
25 }
```

Line 9: right hand side vector set as last column of matrix, facilitates simultaneous row transformations of matrix and r.h.s.

Variable `fac` ≙ multiplier

Line 24: extract solution from last column of transformed matrix.

(2.3.5) Computational effort of Gaussian elimination

Forward elimination: three nested loops (note: compact vector operation in line 15 involves another loop

from $i+1$ to m)

Back substitution: two nested loops

computational cost ($\leftrightarrow$ number of elementary operations) of Gaussian elimination [**?**, Sect. 1.3]:

$$\begin{aligned} \text{elimination :} \quad & \sum_{i=1}^{n-1}(n-i)(2(n-i)+3) = n(n-1)(\tfrac{2}{3}n+\tfrac{7}{6}) \text{ Ops. ,} \\ \text{back substitution :} \quad & \sum_{i=1}^{n} 2(n-i)+1 = n^2 \text{ Ops. .} \end{aligned} \tag{2.3.6}$$

asymptotic complexity ($\rightarrow$ Sect. 1.4) of Gaussian elimination (without pivoting) for generic LSE $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$ = $\frac{2}{3}n^3 + O(n^2) = O(n^3)$

Experiment 2.3.7 (Runtime of Gaussian elimination)

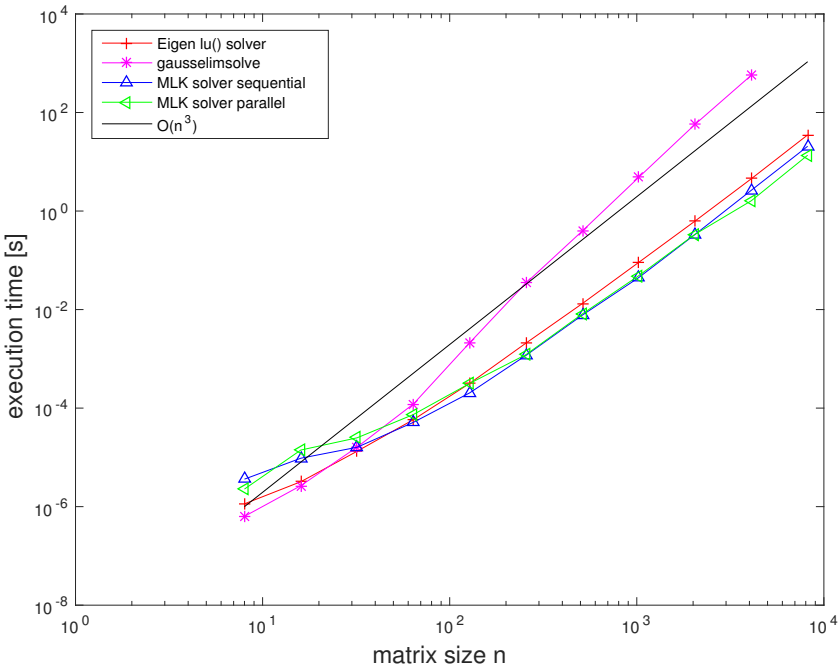
C++11 code 2.3.8: Measuring runtimes of Code 2.3.4 vs. EIGEN lu()-operator vs. MKL
➙ GITLAB

```
//!  Eigen code for timing numerical solution of linear systems
MatrixXd gausstiming(){
  std::vector<int> n = {8,16,32,64,128,256,512,1024,2048,4096,8192};
  int nruns = 3;
  MatrixXd times(n.size(),3);
  for(int i = 0; i < n.size(); ++i){
    Timer t1, t2;  // timer class
    MatrixXd A = MatrixXd::Random(n[i],n[i]) +
      n[i]*MatrixXd::Identity(n[i],n[i]);
    VectorXd b = VectorXd::Random(n[i]);
    VectorXd x(n[i]);
    for(int j = 0; j < nruns; ++j){
      t1.start();  x = A.lu().solve(b);  t1.stop();// Eigen
        implementation
      #ifndef EIGEN_USE_MKL_ALL   // only test own algorithm without
        MKL
      if(n[i] <= 4096)       // Prevent long runs
        t2.start();    gausselimsolve(A,b,x);   t2.stop();     //
          own gauss elimination
      #endif
    }
    times(i,0) = n[i]; times(i,1) = t1.min(); times(i,2) = t2.min();
  }
  return times;
}
```

Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3

EIGEN is about two orders of magnitude faster than a direct implementation, MKL is even faster.



n	Code 2.3.4 [s]	EIGEN lu() [s]	MKL sequential [s]	MKL parallel [s]
8	6.340e-07	1.140e-06	3.615e-06	2.273e-06
16	2.662e-06	3.203e-06	9.603e-06	1.408e-05
32	1.617e-05	1.331e-05	1.603e-05	2.495e-05
64	1.214e-04	5.836e-05	5.142e-05	7.416e-05
128	2.126e-03	3.180e-04	2.041e-04	3.176e-04
256	3.464e-02	2.093e-03	1.178e-03	1.221e-03
512	3.954e-01	1.326e-02	7.724e-03	8.175e-03
1024	4.822e+00	9.073e-02	4.457e-02	4.864e-02
2048	5.741e+01	6.260e-01	3.347e-01	3.378e-01
4096	5.727e+02	4.531e+00	2.644e+00	1.619e+00
8192	-	3.510e+01	2.064e+01	1.360e+01

Never implement Gaussian elimination yourself **!**

use numerical libraries (LAPACK/MKL) or EIGEN **!**

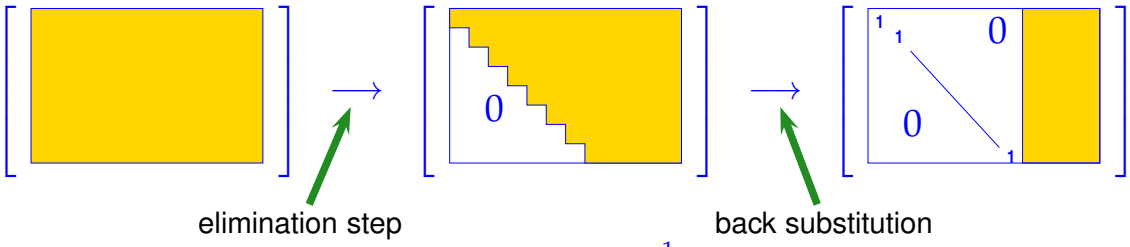
A concise list of libraries for numerical linear algebra and related problems can be found here.

Remark 2.3.9 (Gaussian elimination for non-square matrices)

In Code 2.3.4: the right hand side vector $\mathbf{b}$ was first appended to matrix $\mathbf{A}$ as rightmost column, and then forward elimination and back substitution were carried out on the resulting matrix.

➢ Gaussian elimination for $\mathbf{A} \in \mathbb{K}^{n,n+1}$!

"fat matrix": $\mathbf{A} \in \mathbb{K}^{n,m}$, $m>n$:



Recall Code 2.3.4 ($m = n+1$): the solution vector $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ was recovered as the rightmost column of the augmented matrix $(\mathbf{A}, \mathbf{b})$ after forward elimination and back substitution. In the above cartoon it would be contained in the yellow part of the matrix on the right.

Simultaneous solving of LSE with multiple right hand sides

Given regular $\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{B} \in \mathbb{K}^{n,k}$, seek $\mathbf{X} \in \mathbb{K}^{n,k}$ such that

$$\mathbf{AX} = \mathbf{B} \quad \Leftrightarrow \quad \mathbf{X} = \mathbf{A}^{-1}\mathbf{B}$$

In EIGEN the following code accomplishes this:

```
MatrixXd X = A.lu().solve(B);
```

asymptotic complexity: $O(n^2(n+k))$

C++11 code 2.3.10: Gaussian elimination with multiple r.h.s. → Code 2.3.4 ⇢ GITLAB

```
//!  Gauss elimination without pivoting, X = A^{-1}B
//!  A must be an n×n-matrix, B an n×m-matrix
//!  Result is returned in matrix X
void gausselimsolvemult(const MatrixXd &A, const MatrixXd& B,
                        MatrixXd& X) {
  int n = A.rows(), m = B.cols();
  MatrixXd AB(n, n+m); // Augmented matrix [A,B]
  AB << A, B;
  // Forward elimination, do not forget the B part of the Matrix
  for(int i = 0; i < n-1; ++i){
    double pivot = AB(i,i);
    for(int k = i+1; k < n; ++k){
      double fac = AB(k,i)/pivot;
      AB.block(k,i+1,1,m+n-i-1) -= fac * AB.block(i,i+1,1,m+n-i-1);
    }
  }
  // Back substitution
  AB.block(n-1, n,1, m) /= AB(n-1,n-1);
  for(int i = n-2; i >= 0; --i) {
    for(int l = i+1; l < n; ++l) {
      AB.block(i,n,1,m) -= AB.block(l,n,1,m)*AB(i,l);
    }
    AB.block(i,n,1,m) /= AB(i,i);
  }
  X = AB.rightCols(m);
}
```

Next two remarks: For understanding or analyzing special variants of Gaussian elimination, it is useful to be aware of

- the effects of elimination steps on the level of *matrix blocks*, *cf.* Rem. 1.3.15,
- and of the *recursive nature* of Gaussian elimination.

Remark 2.3.11 (Gaussian elimination via rank-1 modifications)

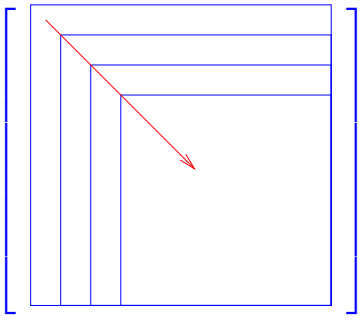
Block perspective (first step of Gaussian elimination with pivot $\alpha \neq 0$), *cf.* (2.3.2):

$$\mathbf{A} := \left[\begin{array}{c|c} \alpha & \mathbf{c}^\top \\ \hline \mathbf{d} & \mathbf{C} \end{array}\right] \rightarrow \mathbf{A}' := \left[\begin{array}{c|c} \alpha & \mathbf{c}^\top \\ \hline \mathbf{0} & \mathbf{C}' := \mathbf{C} - \frac{\mathbf{d}\mathbf{c}^\top}{\alpha} \end{array}\right]. \tag{2.3.12}$$

rank-1 modification of $\mathbf{C}$

Adding a tensor product of two vectors to a matrix is called a rank-1 modification of that matrix.

(2.3.12) suggests a *recursive* variant of Gaussian elimination:



r.h.s. $\mathbf{b} \sim$ A(:,end) ➤

C++11 code 2.3.13: GE by rank-1 modification ↬ GITLAB

```
2  //!  in-situ Gaussian elimination, no pivoting
3  //!  right hand side in rightmost column of A
4  //!  back substitution is not done in this code!
5  void blockgs(MatrixXd &A){
6    int n = A.rows();
7    for(int i = 1; i < n; ++i){
8      // rank-1 modification of C
9      A.bottomRightCorner(n-i,n-i+1) -=
           A.col(i-1).tail(n-i) *
           A.row(i-1).tail(n-i+1) / A(i-1,i-1);
10     A.col(i-1).tail(n-i).setZero();// set d = 0
11   }
12 }
```

In this code the Gaussian elimination is carried out in situ: the matrix A is replaced with the transformed matrices during elimination. If the matrix is not needed later this offers maximum efficiency. Notice that the recursive call is omitted!

Remark 2.3.14 (Block Gaussian elimination)

Recall "principle" from Ex. 1.3.15: deal with block matrices ("matrices of matrices") like regular matrices (except for commutativity of multiplication!).

Given: regular matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ with sub-matrices
$\mathbf{A}_{11} := (\mathbf{A})_{1:k,1:k}$, $\mathbf{A}_{22} = (\mathbf{A})_{k+1:n,k+1,n}$, $\mathbf{A}_{12} = (\mathbf{A})_{1:k,k+1,n}$, $\mathbf{A}_{21} := (\mathbf{A})_{k+1:n,1:k}$, $k < n$,
right hand side vector $\mathbf{b} \in \mathbb{K}^n$, $\mathbf{b}_1 = (\mathbf{b})_{1:k}$, $\mathbf{b}_2 = (\mathbf{b})_{k+1:n}$

$$\left[\begin{array}{cc|c} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{b}_1 \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{b}_2 \end{array}\right] \xrightarrow{❶} \left[\begin{array}{cc|c} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{b}_1 \\ 0 & \mathbf{A}_{22}-\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12} & \mathbf{b}_2-\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1 \end{array}\right]$$
$$\xrightarrow{❷} \left[\begin{array}{cc|c} \mathbf{I} & 0 & \mathbf{A}_{11}^{-1}(\mathbf{b}_1-\mathbf{A}_{12}\mathbf{S}^{-1}\mathbf{b}_S) \\ 0 & \mathbf{I} & \mathbf{S}^{-1}\mathbf{b}_S \end{array}\right];,$$

where $\mathbf{S} := \mathbf{A}_{22}-\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ (Schur complement, see Rem. 2.3.34), $\mathbf{b}_S := \mathbf{b}_2-\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1$

❶: elimination step, ❷: backsubstitution step *Assumption*: (sub-)matrices regular, if required.

We can read off the solution of the block-partitioned linear system from the above Gaussian elimination:

$$\begin{bmatrix}\mathbf{A}_{11} & \mathbf{A}_{12}\\ \mathbf{A}_{21} & \mathbf{A}_{22}\end{bmatrix}\begin{bmatrix}\mathbf{x}_1\\ \mathbf{x}_2\end{bmatrix} = \begin{bmatrix}\mathbf{b}_1\\ \mathbf{b}_2\end{bmatrix} \quad \Rightarrow \quad \begin{array}{l}\mathbf{x}_2 = \mathbf{S}^{-1}\mathbf{b}_S\,,\\ \mathbf{x}_1 = \mathbf{A}_{11}^{-1}(\mathbf{b}_1-\mathbf{A}_{12}\mathbf{S}^{-1}\mathbf{b}_S)\,.\end{array} \tag{2.3.15}$$

2.3.2 LU-Decomposition

A matrix factorization (*ger.* Matrixzerlegung) expresses a general matrix $\mathbf{A}$ as product of two *special* (factor) matrices. Requirements for these special matrices define the matrix factorization.

Mathematical issue: existence & uniqueness
Numerical issue: algorithm for computing factor matrices

Matrix factorizations

- ☞ often capture the essence of algorithms in compact form (here: Gaussian elimination),
- ☞ are important building blocks for complex algorithms,
- ☞ are key theoretical tools for algorithm analysis.

In this section: forward elimination step of Gaussian elimination will be related to a special matrix factorization, the so-called LU-decomposition or LU-factorization.

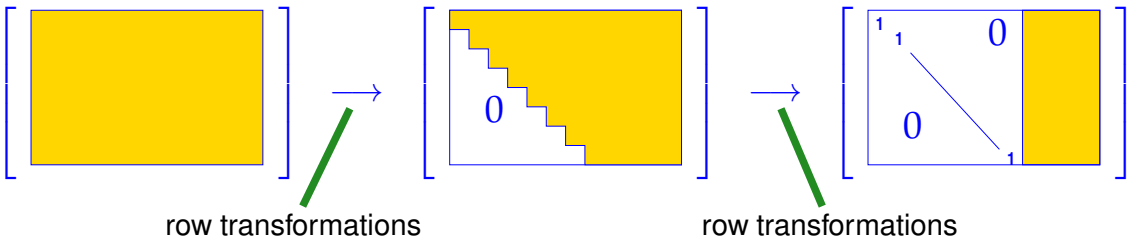
Supplementary reading. The LU-factorization should be well known from the introductory linear algebra course. In case you need to refresh your knowledge, please consult one of the following:

- textbook by Nipp & Stoffer [**?**, Sect. 2.4],
- book by M. Hanke-Bourgeois [**?**, II.4],
- linear algebra lecture notes by M. Gutknecht [**?**, Sect. 3.1],
- textbook by Quarteroni et al. [**?**, Sect.3.3.1],
- Sect. 3.5 of the book by Dahmen & Reusken,

- Sect. 5.1 of the textbook by Ascher & Greif [**?**].

See also (2.3.16) below.

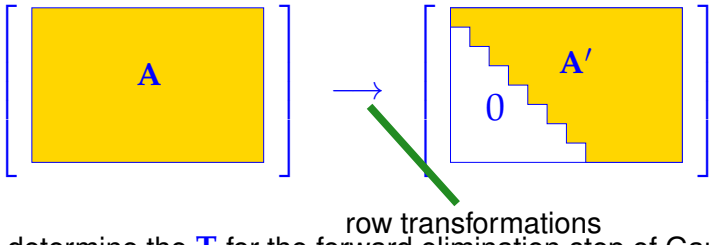
The gist of Gaussian elimination:



Here: row transformation = adding a multiple of a matrix row to another row, or
multiplying a row with a non-zero scalar (number)
swapping two rows
(more special than row transformations discussed in Ex. 1.3.13)

Note: these row transformations preserve regularity of a matrix (why ?)
▷ suitable for transforming linear systems of equations
(solution will not be affected)

Ex. 1.3.13: row transformations can be realized by multiplication *from left* with suitable transformation matrices. When multiplying these transformation matrices we can emulate the effect to successive row transformations through left multiplication with a matrix $\mathbf{T}$:



$$\Leftrightarrow \quad \mathbf{TA} = \mathbf{A}' \,.$$

Now we want to determine the $\mathbf{T}$ for the forward elimination step of Gaussian elimination.

Example 2.3.16 (Gaussian elimination and LU-factorization → [?, Sect. 2.4], [?, II.4], [?, Sect. 3.1])

LSE from Ex. 2.3.1: consider (forward) Gaussian elimination:

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} \quad \longleftrightarrow \quad \begin{array}{rcrcrcr} x_1 & + & x_2 & & & = & 4 \\ 2x_1 & + & x_2 & - & x_3 & = & 1 \\ 3x_1 & - & x_2 & - & x_3 & = & -3 \end{array} .$$

$$\begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} \rightarrowtail \begin{bmatrix} 1 & & \\ 2 & 1 & \\ & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ -7 \\ -3 \end{bmatrix} \rightarrowtail$$

$$\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 3 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} \\ 0 & -1 & -1 \\ 0 & -4 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ -7 \\ -15 \end{bmatrix} \rightarrow \underbrace{\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 3 & \mathbf{4} & 1 \end{bmatrix}}_{=L} \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 0 & \mathbf{-1} & \mathbf{-1} \\ 0 & 0 & 3 \end{bmatrix}}_{=U} \begin{bmatrix} 4 \\ -7 \\ 13 \end{bmatrix}$$

= pivot row, pivot element **bold**, *negative* multipliers red

Details: link between Gaussian elimination and matrix factorization → Ex. 2.3.16
(row transformation = multiplication with elimination matrix)

$$a_1 \neq 0 \quad \blacktriangleright \quad \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ -\frac{a_2}{a_1} & 1 & & & 0 \\ -\frac{a_3}{a_1} & & \ddots & & \\ \vdots & & & & \\ -\frac{a_n}{a_1} & 0 & & & 1 \end{bmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} a_1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} . \tag{2.3.17}$$

$\blacktriangleright$ $n-1$ steps of Gaussian elimination: ➤ matrix factorization (→ Ex. 2.3.1)
(non-zero pivot elements assumed)

$$\mathbf{A} = \mathbf{L}_1 \cdot \cdots \cdot \mathbf{L}_{n-1}\mathbf{U} \quad \text{with} \quad \begin{array}{l} \text{elimination matrices } \mathbf{L}_i,\ i = 1,\ldots,n-1\ , \\ \text{upper triangular matrix } \mathbf{U} \in \mathbb{R}^{n,n}\ . \end{array}$$

$$\begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ l_2 & 1 & & & 0 \\ l_3 & & \ddots & & \\ \vdots & & & & \\ l_n & 0 & & & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & & & 0 \\ 0 & h_3 & 1 & & \\ \vdots & \vdots & & \ddots & \\ 0 & h_n & 0 & & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & \cdots & 0 \\ l_2 & 1 & & & 0 \\ l_3 & h_3 & 1 & & \\ \vdots & \vdots & & \ddots & \\ l_n & h_n & 0 & & 1 \end{bmatrix}$$

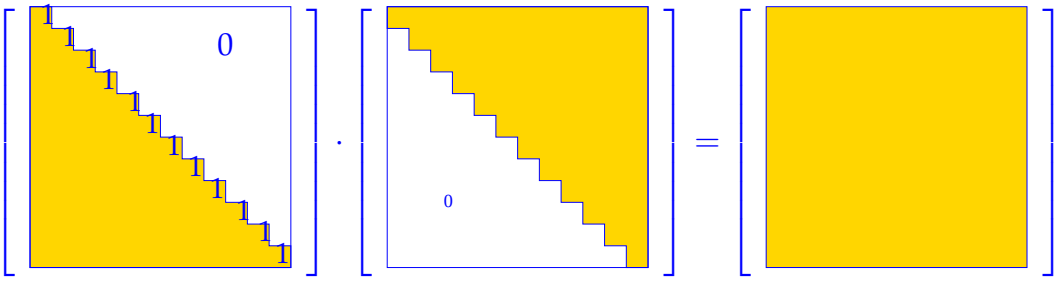
$\blacktriangleright$ $\mathbf{L}_1 \cdot \cdots \cdot \mathbf{L}_{n-1}$ are normalized lower triangular matrices
(entries = multipliers $-\frac{a_{ik}}{a_{kk}}$ from (2.3.2) → Ex. 2.3.1)

The (forward) Gaussian elimination (without pivoting), for $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$, if possible, is *algebraically equivalent* to an LU-factorization/LU-decomposition $\mathbf{A} = \mathbf{LU}$ of $\mathbf{A}$ into a normalized lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$, [**?**, Thm. 3.2.1], [**?**, Thm. 2.10], [**?**, Sect. 3.1].

Algebraically equivalent ≙ when carrying out the forward elimination in situ as in Code 2.3.4 and storing the multipliers in a lower triangular matrix as in Ex. 2.3.16, then the latter will contain the $\mathbf{L}$-factor and the original matrix will be replaced with the $\mathbf{U}$-factor.

Definition 2.3.18.

LU-decomposition/LU-factorization Given a square matrix $\mathbf{A} \in \mathbb{K}^{n,n}$, an upper triangular matrix $\mathbf{U} \in \mathbb{K}^{n,n}$ and a normalized lower triangular matrix (→ Def. 1.1.5) form an LU-decomposition/LU-factorization of $\mathbf{A}$, if $\mathbf{A} = \mathbf{LU}$.



Lemma 2.3.19. Existence of LU-decomposition

The LU-decomposition of $\mathbf{A} \in \mathbb{K}^{n,n}$ exists, if all submatrices $(\mathbf{A})_{1:k,1:k}$, $1 \leq k \leq n$, are regular.

Proof. by block matrix perspective ($\rightarrow$ Rem. 1.3.15) and induction w.r.t. n:

$n = 1$: assertion trivial

$n-1 \rightarrow n$: Induction hypothesis ensures existence of normalized lower triangular matrix $\widetilde{\mathbf{L}}$ and regular upper triangular matrix $\widetilde{\mathbf{U}}$ such that $\widetilde{\mathbf{A}} = \widetilde{\mathbf{L}}\widetilde{\mathbf{U}}$, where $\widetilde{\mathbf{A}}$ is the upper left $(n-1) \times (n-1)$ block of $\mathbf{A}$:

$$\left[\begin{array}{c|c} \widetilde{\mathbf{A}} & \mathbf{b} \\ \hline \mathbf{a}^\top & \alpha \end{array}\right] = \left[\begin{array}{c|c} \widetilde{\mathbf{L}} & 0 \\ \hline \mathbf{x}^\top & 1 \end{array}\right] \left[\begin{array}{c|c} \widetilde{\mathbf{U}} & \mathbf{y} \\ \hline 0 & \xi \end{array}\right] =: \mathbf{LU} \,.$$

Then solve

❶ $\widetilde{\mathbf{L}}\mathbf{y} = \mathbf{b}$ $\rightarrow$ provides $\mathbf{y} \in \mathbb{K}^n$,

❷ $\mathbf{x}^\top \widetilde{\mathbf{U}} = \mathbf{a}^\top$ $\rightarrow$ provides $\mathbf{x} \in \mathbb{K}^n$,

❸ $\mathbf{x}^\top \mathbf{y} + \xi = \alpha$ $\rightarrow$ provides $\xi \in \mathbb{K}$.

Regularity of $\mathbf{A}$ involves $\xi \neq 0$ (why?) so that $\mathbf{U}$ will be regular, too.

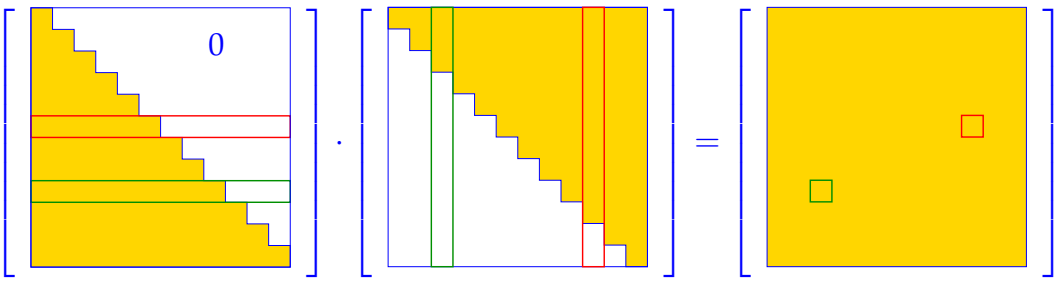
(2.3.20) Uniqueness of LU-decomposition

Regular upper triangular matrices and normalized lower triangular matrices form *matrix groups* ($\rightarrow$ Lemma 1.3.9). Their only common element is the identity matrix.

$$\mathbf{L}_1\mathbf{U}_1 = \mathbf{L}_2\mathbf{U}_2 \quad \Rightarrow \quad \mathbf{L}_2^{-1}\mathbf{L}_1 = \mathbf{U}_2\mathbf{U}_1^{-1} = \mathbf{I} \,.$$

(2.3.21) Basic algorithm for computing LU-decomposition

A direct way to determine the factor matrices of the LU-decomposition [**?**, Sect. 3.1], [**?**, Sect. 3.3.3]: We study the entries of the product of a normalized lower triangular and an upper triangular matrix, see Def. 1.1.5:



Taking into account the zero entries known a priori, we arrive at

$$\mathbf{LU} = \mathbf{A} \quad \Rightarrow \quad a_{ik} = \sum_{j=1}^{\min\{i,k\}} l_{ij}u_{jk} = \begin{cases} \sum_{j=1}^{i-1} l_{ij}u_{jk} + 1 \cdot u_{ik} & \text{, if } i \leq k \text{ ,} \\ \sum_{j=1}^{k-1} l_{ij}u_{jk} + l_{ik}u_{kk} & \text{, if } i > k \text{ .} \end{cases} \tag{2.3.22}$$

This reveals how to compute the entries of $\mathbf{L}$ and $\mathbf{U}$ sequentially. We start with the top row of $\mathbf{U}$, which agrees with that of $\mathbf{A}$, and then work our way towards to bottom right corner:

➤
- row by row computation of $\mathbf{U}$
- column by column computation of $\mathbf{L}$

Entries of $\mathbf{A}$ can be replaced with those of $\mathbf{L}$, $\mathbf{U}$!
(so-called in situ/in place computation)

(Crout's algorithm, [**?**, Alg. 3.1])

$\hat{=}$ rows of $\mathbf{U}$

$\hat{=}$ columns of $\mathbf{L}$

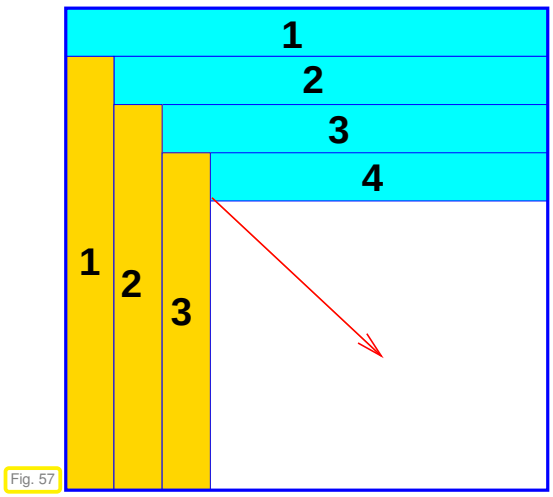


Fig. 57

The following code follows this sequential computation scheme:

C++11 code 2.3.23: LU-factorization ↬ GITLAB

```
//! Algorithm of Crout: LU-factorization of A ∈ K^{n,n}
void lufak(const MatrixXd& A, MatrixXd& L, MatrixXd& U){
  int n = A.rows(); assert( n == A.cols());
  L = MatrixXd::Identity(n,n);
  U = MatrixXd::Zero(n,n);
  for(int k = 0; k < n; ++k){
    // Compute row of U
    for(int j = k; j < n; ++j)
      U(k,j) = A(k,j) - (L.block(k,0,1,k) * U.block(0,j,k,1))(0,0);
    // Compute column of L
    for(int i = k+1; i < n; ++i)
      L(i,k) = (A(i,k) - (L.block(i,0,1,k) *
        U.block(0,k,k,1))(0,0))/U(k,k);
  }
}
```

It is instructive to compare this code with a simple implementation of the matrix product of a normalized lower triangular and an upper triangular matrix. From this perspective the LU-factorization looks like the "inversion" of matrix multiplication:

C++11 code 2.3.24: matrix multiplication $\mathbf{L} \cdot \mathbf{U}$ ↬ GITLAB

```
//! Multiplication of normalized lower/upper triangular matrices
void lumult(const MatrixXd& L, const MatrixXd& U, MatrixXd& A){
  int n = L.rows();
  assert( n == L.cols() && n == U.cols() && n == U.rows() );
```

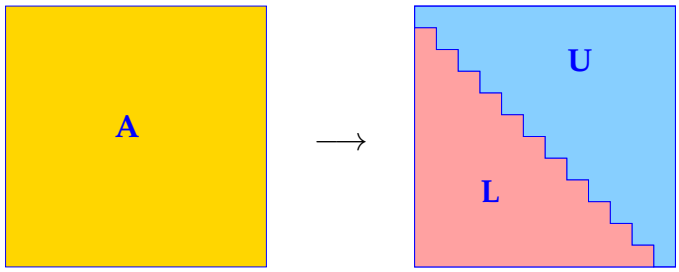
```
6     A = MatrixXd::Zero(n,n);
7     for(int k = 0; k < n; ++k){
8       for(int j = k; j < n; ++j)
9         A(k,j) = U(k,j) + (L.block(k,0,1,k) * U.block(0,j,k,1))(0,0);
10      for(int i = k+1; i < n; ++i)
11        A(i,k) = (L.block(i,0,1,k) * U.block(0,k,k,1))(0,0) +
              L(i,k)*U(k,k);
12    }
13  }
```

Observe: Solving for entries `L(i,k)` of $\mathbf{L}$ and `U(k,j)` of $\mathbf{U}$ in the multiplication of an upper triangular and normalized lower triangular matrix (Code 2.3.24) yields the algorithm for LU-factorization (Code 2.3.23).

The computational cost of LU-factorization is immediate from Code 2.3.23:

asymptotic complexity of LU-factorization of $\mathbf{A} \in \mathbb{R}^{n,n}$ = $\frac{1}{3}n^3 + O(n^2) = O(n^3)$ (2.3.25)

Remark 2.3.26 (In-situ LU-decomposition)



Replace entries of $\mathbf{A}$ with entries of $\mathbf{L}$ (strict lower triangle) and $\mathbf{U}$ (upper triangle).

Remark 2.3.27 (Recursive LU-factorization)

Recall: recursive view of Gaussian elimination → Rem. 2.3.11

In light of the close relationship between Gaussian elimination and LU-factorization there will also be a recursive version of LU-factorization.

Recursive in situ (in place) LU-decomposition of $\mathbf{A} \in \mathbb{R}^{n,n}$ (without pivoting):

$\mathbf{L}$,$\mathbf{U}$ stored in place of $\mathbf{A}$:

C++11 code 2.3.28: Recursive LU-factorization ↪ GITLAB

```
2  //!  in situ recursive LU-factorization
```

```
3  MatrixXd lurec(const MatrixXd &A){
4      int n = A.rows();
5      MatrixXd result(n,n);
6      if(n > 1){
7          VectorXd fac = A.col(0).tail(n-1) / A(0,0);//
8          result.bottomRightCorner(n-1,n-1) = lurec(
               A.bottomRightCorner(n-1,n-1)    - fac *
               A.row(0).tail(n-1) );//
9          result.row(0) = A.row(0); result.col(0).tail(n-1) = fac;
10         return result;
11     }
12     return A;
13 }
```

Refer to (2.3.12) to understand `lurec`: the rank-1 modification of the lower $(n-1)\times(n-1)$-block of the matrix is done in lines Line 7-Line 8 of the code.

C++11 code 2.3.29: Driver for recursive LU-factorization ⇝ GITLAB

```
2  //!  post-processing:  extract L and U
3  void lurecdriver(const MatrixXd &A, MatrixXd &L, MatrixXd &U){
4      MatrixXd A_dec = lurec(A);
5      // post-processing:
6      //extract L and U
7      U = A_dec.triangularView<Upper>();
8      L.setIdentity();
9      L += A_dec.triangularView<StrictlyLower>();
10 }
```

(2.3.30) Using LU-factorization to solve a linear system of equations

Solving an $n\times n$ linear system of equations by LU-factorization:

$\mathbf{Ax}=\mathbf{b}$:
① LU-decomposition $\mathbf{A}=\mathbf{LU}$, #elementary operations $\frac{1}{3}n(n-1)(n+1)$
② forward substitution, solve $\mathbf{Lz}=\mathbf{b}$, #elementary operations $\frac{1}{2}n(n-1)$
③ backward substitution, solve $\mathbf{Ux}=\mathbf{z}$, #elementary operations $\frac{1}{2}n(n+1)$

➢ asymptotic complexity: (in leading order) the same as for Gaussian elimination

However, the perspective of LU-factorization reveals that the solution of linear systems of equations can be split into two separate phases with different asymptotic complexity in terms of the number n of unknowns:

setup phase (factorization) Cost: $O(n^3)$	+	**elimination phase** (forward/backward substition) Cost: $O(n^2)$

Remark 2.3.31 (Rationale for using LU-decomposition in algorithms)

Gauss elimination and LU-factorization for the solution of a linear system of equations (→ § 2.3.30) are equivalent and only differ in the ordering of the steps.

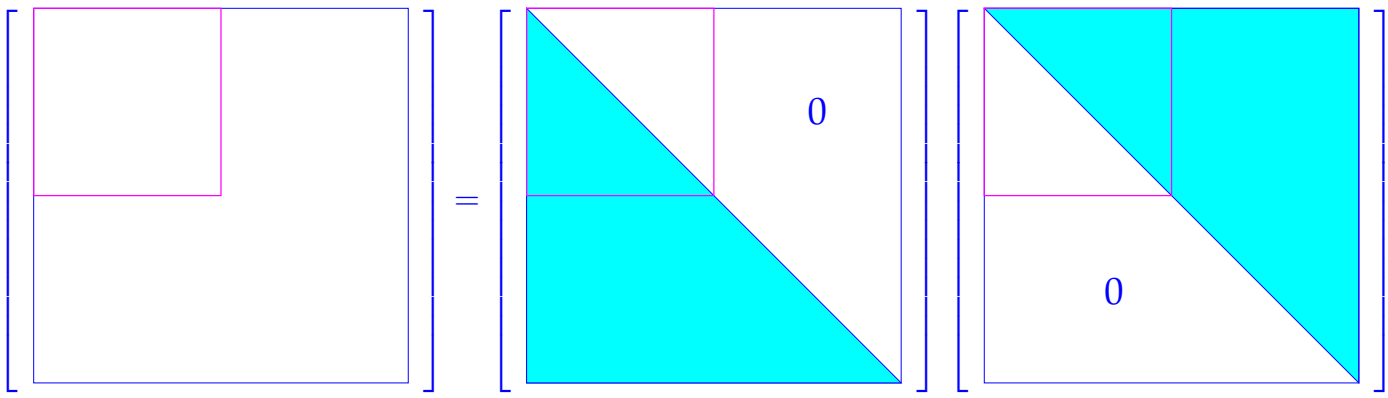
Then, why is it important to know about LU-factorization?

Because in the case of LU-factorization the expensive forward elimination and the less expensive (forward/backward) substitutions are separated, which sometimes can be exploited to reduce computational cost, as highlighted in Rem. 2.5.10 below.

Remark 2.3.32 ("'Partial LU-decompositions" of principal minors)

Principal minor $\hat{=}$ left upper block of a matrix

The following "visual rule" help identify the structure of the LU-factors of a matrix.



(2.3.33)

The left-upper blocks of both $\mathbf{L}$ and $\mathbf{U}$ in the LU-factorization of $\mathbf{A}$ depend only on the corresponding left-upper block of $\mathbf{A}$!

Remark 2.3.34 (Block LU-factorization)

In the spirit of Rem. 1.3.15: block perspective of LU-factorization.

Natural in light of the close connection between matrix multiplication and matrix factorization, *cf.* the relationship between matrix factorization and matrix multiplication found in § 2.3.21:

Block matrix multiplication (1.3.16) $\cong$ block LU-decomposition:

With $\mathbf{A}_{11} \in \mathbb{K}^{n,n}$ regular, $\mathbf{A}_{12} \in \mathbb{K}^{n,m}$, $\mathbf{A}_{21} \in \mathbb{K}^{m,n}$, $\mathbf{A}_{22} \in \mathbb{K}^{m,m}$:

$$\underbrace{\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{A}_{21}\mathbf{A}_{11}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ 0 & \mathbf{S} \end{bmatrix}}_{\text{block LU-factorization}} , \quad \text{Schur complement} \quad \boxed{\mathbf{S} := \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}} . \tag{2.3.35}$$

→ block Gaussian elimination, see Rem. 2.3.14.

2.3.3 Pivoting

Known from linear algebra [**?**, Sect. 1.1]:

$$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} = \begin{bmatrix}b_1\\ b_2\end{bmatrix} \qquad \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} = \begin{bmatrix}b_2\\ b_1\end{bmatrix}$$

breakdown of Gaussian elimination pivot element $= 0$ — Gaussian elimination feasible

Idea (in linear algebra): Avoid zero pivot elements by swapping rows

Example 2.3.36 (Pivoting and numerical stability → [?, Example 3.2.3])

```
MatrixXd A(2,2);
A <<      5.0e-17,  1.0,
          1.0,      1.0;
VectorXd b(2), x2(2);
b << 1.0, 2.0;
VectorXd x1 = A.fullPivLu().solve(b);
gausselimsolve(A,b, x2);// see Code 2.3.10
MatrixXd L(2,2), U(2,2);
lufak(A, L, U);// see Code 2.3.23
VectorXd z = L.lu().solve(b);
VectorXd x3 = U.lu().solve(z);
cout << "x1 = \n" << x1 << "\nx2 = \n" << x2 <<
   "\nx3 = \n" << x3 << std::endl;
```

Output:

```
x1 =
1
1
x2 =
0
1
x3 =
0
1
```

$$\mathbf{A} = \begin{bmatrix}\epsilon & 1\\ 1 & 1\end{bmatrix} \quad , \quad \mathbf{b} = \begin{bmatrix}1\\ 2\end{bmatrix} \quad \Rightarrow \quad \mathbf{x} = \begin{bmatrix}\frac{1}{1-\epsilon}\\ \frac{1-2\epsilon}{1-\epsilon}\end{bmatrix} \approx \begin{bmatrix}1\\ 1\end{bmatrix} \quad \text{for } |\epsilon| \ll 1\,.$$

What is wrong with EIGEN? Needed: insight into roundoff errors, which we already have → Section 1.5.3

Armed with knowledge about the behavior of machine numbers and roundoff errors we can now understand what is going on in Ex. 2.3.36

Straightforward LU-factorization: if $\epsilon \leq \frac{1}{2}\mathrm{EPS}$, $\mathrm{EPS} \hat{=}$ machine precision,

$$\blacktriangleright \quad \mathbf{L} = \begin{bmatrix}1 & 0\\ \epsilon^{-1} & 1\end{bmatrix} \quad , \quad \mathbf{U} = \begin{bmatrix}\epsilon & 1\\ 0 & 1-\epsilon^{-1}\end{bmatrix} \overset{(*)}{=} \widetilde{\mathbf{U}} := \begin{bmatrix}\epsilon & 1\\ 0 & -\epsilon^{-1}\end{bmatrix} \quad \text{in } \mathbb{M}\,! \qquad (2.3.37)$$

$(*)$: because $1 \widetilde{+} 2/\mathrm{EPS} = 2/\mathrm{EPS}$, see Exp. 1.5.35.

Solution of $\mathbf{L}\widetilde{\mathbf{U}}\mathbf{x} = \mathbf{b}$: $\mathbf{x} = \begin{bmatrix} 2\epsilon \\ 1-2\epsilon \end{bmatrix}$ (meaningless result !)

LU-factorization after swapping rows:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix} \Rightarrow \mathbf{L} = \begin{bmatrix} 1 & 0 \\ \epsilon & 1 \end{bmatrix}, \quad \mathbf{U} = \begin{bmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{bmatrix} = \widetilde{\mathbf{U}} := \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \text{ in } \mathbb{M}. \tag{2.3.38}$$

Solution of $\mathbf{L}\widetilde{\mathbf{U}}\mathbf{x} = \mathbf{b}$: $\mathbf{x} = \begin{bmatrix} 1+2\epsilon \\ 1-2\epsilon \end{bmatrix}$ (sufficiently accurate result !)

no row swapping, → (2.3.37): $\mathbf{L}\widetilde{\mathbf{U}} = \mathbf{A} + \mathbf{E}$ with $\mathbf{E} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ unstable !

after row swapping, → (2.3.38): $\mathbf{L}\widetilde{\mathbf{U}} = \widetilde{\mathbf{A}} + \mathbf{E}$ with $\mathbf{E} = \begin{bmatrix} 0 & 0 \\ 0 & \epsilon \end{bmatrix}$ stable !

Introduction to the notion of stability → Section 1.5.5, Def. 1.5.85, see also [**?**, Sect. 2.3].

Suitable pivoting essential for controlling impact of roundoff errors
on Gaussian elimination (→ Section 1.5.5, [**?**, Sect. 2.5])

Example 2.3.39 (Gaussian elimination with pivoting for 3×3-matrix)

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & -3 & 2 \\ 1 & 24 & 0 \end{bmatrix} \overset{❶}{\to} \begin{bmatrix} 2 & -3 & 2 \\ 1 & 2 & 2 \\ 1 & 24 & 0 \end{bmatrix} \overset{❷}{\to} \begin{bmatrix} 2 & -3 & 2 \\ 0 & 3.5 & 1 \\ 0 & 25.5 & -1 \end{bmatrix} \overset{❸}{\to} \begin{bmatrix} 2 & -7 & 2 \\ 0 & 25.5 & -1 \\ 0 & 3.5 & 1 \end{bmatrix} \overset{❹}{\to} \begin{bmatrix} 2 & -7 & 2 \\ 0 & 25.5 & -1 \\ 0 & 0 & 1.373 \end{bmatrix}$$

❶: swap rows 1 & 2.
❷: elimination with top row as pivot row
❸: swap rows 2 & 3
❹: elimination with 2nd row as pivot row

(2.3.40) Algorithm: Gaussian elimination with partial pivoting

C++11 code 2.3.41: Gaussian elimination with pivoting: extension of Code 2.3.4 ↪ GITLAB

```
2  //!  Solving an LSE Ax = b by Gaussian elimination with partial
       pivoting
3  //!  A must be an n×n-matrix, b an n-vector
4  void gepiv(const MatrixXd &A, const VectorXd& b, VectorXd& x){
5       int n = A.rows();
6       MatrixXd Ab(n,n+1);
7       Ab << A, b;     //
8       // Forward elimination by rank-1 modification, see Rem. 2.3.11
9       for(int k = 0; k < n-1; ++k){
10           int j; double p; // p = relativly largest pivot, j = pivot
                row index
```

```
11          p = ( Ab.col(k).tail(n-k).cwiseAbs().cwiseQuotient(
                Ab.block(k,k,n-k,n-k).cwiseAbs().rowwise().maxCoeff() )
                ).maxCoeff(&j);//
12          if( p < std::numeric_limits<double>::epsilon() *
                Ab.block(k,k,n-k,n-k).norm() )
13              throw std::logic_error("nearly singular");//
14          Ab.row(k).tail(n-k+1).swap(Ab.row(k+j).tail(n-k+1));//
15          Ab.bottomRightCorner(n-k-1,n-k) -= Ab.col(k).tail(n-k-1) *
                Ab.row(k).tail(n-k) / Ab(k,k);//
16      }
17      // Back substitution (same as in Code 2.3.4)
18      Ab(n-1,n) = Ab(n-1,n) / Ab(n-1,n-1);
19      for(int i = n-2; i >= 0; --i){
20          for(int l = i+1; l < n; ++l){
21              Ab(i,n) -= Ab(l,n)*Ab(i,l);
22          }
23          Ab(i,n) /= Ab(i,i);
24      }
25      x = Ab.rightCols(1); //
26  }
```

choice of pivot row index j (Line 11 of code): relatively largest pivot [**?**, Sect. 2.5],

$$j \in \{k,\ldots,n\} \quad \text{such that} \quad \frac{|a_{ji}|}{\max\{|a_{jl}|,\ l=k,\ldots,n\}} \to \max \tag{2.3.42}$$

for $k = j$, $k \in \{i,\ldots,n\}$: partial pivoting

Explanations to Code 2.3.41:

Line 7: Augment matrix $\mathbf{A}$ by right hand side vector $\mathbf{b}$, see comments on Code 2.3.4 for explanations.

Line 11: Select index j for pivot row according to the recipe of partial pivoting, see (2.3.42).

Note: Inefficient implementation above (too many comparisons)! Try to do better!

Line 13: If the pivot element is still very small relative to the norm of the matrix, then we have encountered an entire column that is close to zero. Gaussian elimination may not be possible in a stable fashion for this matrix; warn user and terminate.

Line 14: A way to swap rows of a matrix in EIGEN.

Line 15: Forward elimination by means of rank-1-update, see (2.3.12).

Line 25: As in Code 2.3.4: after back substitution last column of augmented matrix supplies solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

(2.3.43) Algorithm: LU-factorization with pivoting

Recall: close relationship between Gaussian elimination and LU-factorization

➢ LU-factorization with pivoting? Of course, just by rearranging the operations of Gaussian forward elimination with pivoting.

EIGEN-code for in place LU-factorization of $\mathbf{A} \in \mathbb{R}^{n,n}$ with partial pivoting:

C++11 code 2.3.44: LU-factorization with partial pivoting ↬ GITLAB

```
2  void lupiv(MatrixXd &A){//insitu
3      int n = A.rows();
4      for(int k = 0; k < n-1; ++k){
5          int j; double p; // p = relativly largest pivot, j = pivot
               row index
6          p = ( A.col(k).tail(n-k).cwiseAbs().cwiseQuotient(
               A.block(k,k,n-k,n-k).cwiseAbs().rowwise().maxCoeff() )
               ).maxCoeff(&j); //
7          if( p <  std::numeric_limits<double>::epsilon() *
               A.block(k,k,n-k,n-k).norm() ) //
8              throw std::logic_error("nearly singular");
9          A.row(k).tail(n-k-1).swap(A.row(k+j).tail(n-k-1));//
10         VectorXd fac = A.col(k).tail(n-k-1) / A(k,k);//
11         A.bottomRightCorner(n-k-1,n-k-1) -= fac *
               A.row(k).tail(n-k-1);//
12         A.col(k).tail(n-k-1) = fac;//
13     }
14 }
```

Notice that the recursive call is omitted as in 2.3.11
Explanations to Code 2.3.44:

Line 6: Find the *relatively largest* pivot element p and the index j of the corresponding row of the matrix, see (2.3.42)

Line 7: If the pivot element is still very small relative to the norm of the matrix, then we have encountered an entire column that is close to zero. The matrix is (close to) singular and LU-factorization does not exist.

Line 9: Swap the first and the j-th row of the matrix.

Line 10: Initialize the vector of multiplier.

Line 11: Call the routine for the lower right $(n-1) \times (n-1)$-block of the matrix after subtracting suitable multiples of the first row from the other rows, *cf.* Rem. 2.3.11 and Rem. 2.3.27.

Line 12: Reassemble the parts of the LU-factors. The vector of multipliers yields a column of $\mathbf{L}$, see Ex. 2.3.16.

Remark 2.3.45 (Rationale for partial pivoting policy (2.3.42) → **[?, Page 47])**

Why *relatively* largest pivot element in (2.3.42)? scaling invariance desirable

Scale linear system of equations from Ex. 2.3.36:

$$\begin{bmatrix} 2/\epsilon & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 & 2/\epsilon \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2/\epsilon \\ 2 \end{bmatrix} := \widetilde{\mathbf{b}}$$

No row swapping, if absolutely largest pivot element is used:

$$\begin{bmatrix} 2 & 2/\epsilon \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 2/\epsilon \\ 0 & 1-2/\epsilon \end{bmatrix} \doteq \underbrace{\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}}_{\widetilde{\mathbf{L}}} \underbrace{\begin{bmatrix} 2 & 2/\epsilon \\ 0 & -2/\epsilon \end{bmatrix}}_{\widetilde{\mathbf{U}}} \quad \text{in } \mathbb{M} \,.$$

Using the rules of arithmetic in $\mathbb{M}$ ($\rightarrow$ Exp. 1.5.35), we find

$$\widetilde{\mathbf{U}}^{-1}\widetilde{\mathbf{L}}^{-1}\widetilde{\mathbf{b}} \doteq -\frac{1}{2}\begin{bmatrix} -1 & -1 \\ 0 & \epsilon \end{bmatrix} \frac{2}{\epsilon} \begin{bmatrix} 1 \\ -1 \end{bmatrix} \doteq \begin{bmatrix} 0 \\ 1 \end{bmatrix} ,$$

which is not an acceptable result.

Pivoting: Theoretical perspective

Definition 2.3.46. Permutation matrix

An n-permutation, $n \in \mathbb{N}$, is a bijective mapping $\pi : \{1,\ldots,n\} \mapsto \{1,\ldots,n\}$. The corresponding permutation matrix $\mathbf{P}_\pi \in \mathbb{K}^{n,n}$ is defined by

$$(\mathbf{P}_\pi)_{ij} = \begin{cases} 1 & \text{, if } j = \pi(i) \,, \\ 0 & \text{else.} \end{cases}$$

$$\text{permutation} \quad (1,2,3,4) \mapsto (1,3,2,4) \quad \widehat{=} \quad \mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} .$$

Note:

- $\mathbf{P}^\top = \mathbf{P}^{-1}$ for any permutation matrix $\mathbf{P}$ ($\rightarrow$ permutation matrices orthogonal/unitary)
- $\mathbf{P}_\pi\mathbf{A}$ effects π-permutation *of rows* of $\mathbf{A} \in \mathbb{K}^{n,m}$
- $\mathbf{A}\mathbf{P}_\pi$ effects π-permutation *of columns* of $\mathbf{A} \in \mathbb{K}^{m,n}$

Lemma 2.3.47. Existence of LU-factorization with pivoting $\rightarrow$ [?, Thm. 3.25], [?, Thm. 4.4]

For any regular $\mathbf{A} \in \mathbb{K}^{n,n}$ there is a permutation matrix ($\rightarrow$ Def. 2.3.46) $\mathbf{P} \in \mathbb{K}^{n,n}$, a normalized lower triangular matrix $\mathbf{L} \in \mathbb{K}^{n,n}$, and a regular upper triangular matrix $\mathbf{U} \in \mathbb{K}^{n,n}$ ($\rightarrow$ Def. 1.1.5), such that $\mathbf{PA} = \mathbf{LU}$.

Proof. (by induction)

Every regular matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ admits a row permutation encoded by the permutation matrix $\mathbf{P} \in \mathbb{K}^{n,n}$, such that $\mathbf{A}' := (\mathbf{A})_{1:n-1,1:n-1}$ is regular (why ?).

By induction assumption there is a permutation matrix $\mathbf{P}' \in \mathbb{K}^{n-1,n-1}$ such that $\mathbf{P}'\mathbf{A}'$ possesses a LU-factorization $\mathbf{A}' = \mathbf{L}'\mathbf{U}'$. There are $\mathbf{x},\mathbf{y} \in \mathbb{K}^{n-1}$, $\gamma \in \mathbb{K}$ such that

$$\begin{bmatrix} \mathbf{P}' & 0 \\ 0 & 1 \end{bmatrix} \mathbf{PA} = \begin{bmatrix} \mathbf{P}' & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{A}' & \mathbf{x} \\ \mathbf{y}^\top & \gamma \end{bmatrix} = \begin{bmatrix} \mathbf{L}'\mathbf{U}' & \mathbf{x} \\ \mathbf{y}^\top & \gamma \end{bmatrix} = \begin{bmatrix} \mathbf{L}' & 0 \\ \mathbf{c}^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U} & \mathbf{d} \\ 0 & \alpha \end{bmatrix} ,$$

if we choose

$$\mathbf{d} = (\mathbf{L}')^{-1}\mathbf{x} \quad , \quad \mathbf{c} = (\mathbf{u}')^{-T}\mathbf{y} \quad , \quad \alpha = \gamma - \mathbf{c}^{\top}\mathbf{d} \, ,$$

which is always possible. □

Example 2.3.48 (Ex. 2.3.39 cnt'd)

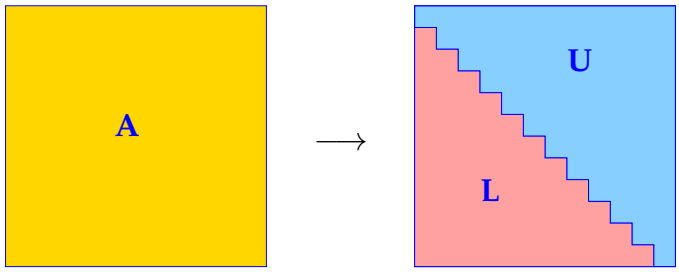
$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & -3 & 2 \\ 1 & 24 & 0 \end{bmatrix} \overset{❶}{\to} \begin{bmatrix} 2 & -3 & 2 \\ 1 & 2 & 2 \\ 1 & 24 & 0 \end{bmatrix} \overset{❷}{\to} \begin{bmatrix} 2 & -3 & 2 \\ 0 & 3.5 & 1 \\ 0 & 25.5 & -1 \end{bmatrix} \overset{❸}{\to} \begin{bmatrix} 2 & -7 & 2 \\ 0 & 25.5 & -1 \\ 0 & 3.5 & 1 \end{bmatrix} \overset{❹}{\to} \begin{bmatrix} 2 & -7 & 2 \\ 0 & 25.5 & -1 \\ 0 & 0 & 1.373 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 2 & -3 & 2 \\ 0 & 25.5 & -1 \\ 0 & 0 & 1.1373 \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0.1373 & 1 \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Two permutations: in step ❶ swap rows #1 and #2, in step ❸ swap rows #2 and #3. Apply these swaps to the identity matrix and you will recover $\mathbf{P}$. See also [**?**, Ex. 3.30].

(2.3.49) LU-decomposition in EIGEN

EIGEN provides various functions for computing the LU-decomposition of a given matrix. They all perform the factorization in-situ → Rem. 2.3.26:



The resulting matrix can be retrieved and used to recover the LU-factors, as demonstrated in the next code snippet.

C++11 code 2.3.50: Performing explicit LU-factorization in EIGEN ↪ GITLAB

```
Eigen::PartialPivLU<MatrixXd> lu(A);
MatrixXd L = MatrixXd::Identity(3,3);
L.triangularView<StrictlyLower>() += lu.matrixLU();
MatrixXd U = lu.matrixLU().triangularView<Upper>();
MatrixXd P = lu.permutationP();
```

Note that for solving a linear system of equations by means of LU-decomposition (the standard algorithm) we never have to extract the LU-factors.

Remark 2.3.51 (Row swapping commutes with forward elimination)

Any kind of pivoting only involves comparisons and row/column permutations, but no arithmetic operations on the matrix entries. This makes the following observation plausible:

The LU-factorization of $\mathbf{A} \in \mathbb{K}^{n,n}$ with partial pivoting by § 2.3.43 is *numerically equivalent* to the LU-factorization of $\mathbf{PA}$ without pivoting ($\rightarrow$ Code in § 2.3.21), when $\mathbf{P}$ is a permutation matrix gathering the row swaps entailed by partial pivoting.

numerically equivalent $\hat{=}$ same result when executed with the same machine arithmetic

The above statement means that whenever we study the impact of roundoff errors on LU-factorization it is safe to consider only the basic version without pivoting, because we can always assume that row swaps have been conducted beforehand.

2.4 Stability of Gaussian Elimination

It will turn out that when investigating the stability of algorithms meant to solve linear systems of equations, a key quantity is the residual.

Definition 2.4.1. Residual

Given an approximate solution $\widetilde{\mathbf{x}} \in \mathbb{K}^n$ of the LSE $\mathbf{Ax} = \mathbf{b}$ ($\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{b} \in \mathbb{K}^n$), its residual is the vector

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\widetilde{\mathbf{x}} .$$

(2.4.2) Probing stability of a direct solver for LSE

Assume that you have downloaded a direct solver for a general (dense) linear system of equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{K}^{n,n}$ regular, $\mathbf{b} \in \mathbb{K}^n$. When given the data $\mathbf{A}$ and $\mathbf{b}$ it returns the perturbed solution $\widetilde{\mathbf{x}}$. How can we tell that $\widetilde{\mathbf{x}}$ is the exact solution of a linear system with slightly perturbed data (in the sense of a tiny relative error of size $\approx$ EPS, EPS the machine precision, see § 1.5.29). That is, how can we tell that $\widetilde{\mathbf{x}}$ is an acceptable solution in the sense of backward error analysis, *cf.* Def. 1.5.85. A similar question was explored in Ex. 1.5.86 for matrix$\times$vector multiplication.

❶ $\mathbf{x} - \widetilde{\mathbf{x}}$ accounted for by perturbation of right hand side:

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ \mathbf{A}\widetilde{\mathbf{x}} &= \mathbf{b} + \Delta\mathbf{b} \end{aligned} \quad \Rightarrow \quad \Delta\mathbf{b} = \mathbf{A}\widetilde{\mathbf{x}} - \mathbf{b} =: -\mathbf{r} \quad \text{(residual, Def. 2.4.1)} .$$

Hence, $\widetilde{\mathbf{x}}$ can be accepted as a solution, if $\dfrac{\|\mathbf{r}\|}{\|\mathbf{b}\|} \leq Cn^3 \cdot \mathrm{EPS}$, for some small constant $C \approx 1$, see Def. 1.5.85. Here, $\|\cdot\|$ can be *any* vector norm on $\mathbb{K}^n$.

❷ $\mathbf{x} - \widetilde{\mathbf{x}}$ accounted for by perturbation of system matrix:

$$\mathbf{Ax} = \mathbf{b} \quad , \quad (\mathbf{A} + \Delta\mathbf{A})\widetilde{\mathbf{x}} = \mathbf{b}$$

$$\left[\text{ try perturbation } \Delta\mathbf{A} = \mathbf{u}\widetilde{\mathbf{x}}^{\mathrm{H}}, \quad \mathbf{u} \in \mathbb{K}^n \right]$$

$$\blacktriangleright \quad \mathbf{u} = \frac{\mathbf{r}}{\|\mathbf{x}\|_2^2} \quad \Rightarrow \quad \Delta\mathbf{A} = \frac{\mathbf{r}\widetilde{\mathbf{x}}^{\mathrm{H}}}{\|\mathbf{x}\|_2^2} .$$

As in Ex. 1.5.86 we find

$$\frac{\|\Delta\mathbf{A}\|_2}{\|\mathbf{A}\|_2} = \frac{\|\mathbf{r}\|}{\|\mathbf{A}\|_2\|\widetilde{\mathbf{x}}\|_2} \leq \frac{\|\mathbf{r}\|_2}{\|\mathbf{A}\widetilde{\mathbf{x}}\|_2} .$$

Thus, $\widetilde{\mathbf{x}}$ is ok in the sense of backward error analysis, if $\dfrac{\|\mathbf{r}\|}{\|\mathbf{A}\widetilde{\mathbf{x}}\|} \leq Cn^3 \cdot \mathrm{EPS}$.

A stable algorithm for solving an LSE yields a residual $\mathbf{r} := \mathbf{b} - \mathbf{A}\widetilde{\mathbf{x}}$ small (in norm) relative to $\mathbf{b}$.

The roundoff error analysis of Gaussian elimination based Ass. 1.5.32 is rather involved. Here we merely summarise the results:

Simplification: equivalence of Gaussian elimination and LU-factorization extends to machine arithmetic, *cf.* Sect. 2.3.2

Lemma 2.4.3. Equivalence of Gaussian elimination and LU-factorization

The following algorithms for solving the LSE $\mathbf{Ax} = \mathbf{b}$ ($\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{b} \in \mathbb{K}^n$) are numerically equivalent:

1. *Gaussian elimination (forward elimination and back substitution) without pivoting, see Algorithm 2.3.3.*
2. *LU-factorization of $\mathbf{A}$ ($\rightarrow$ Code in § 2.3.21) followed by forward and backward substitution, see Algorithm 2.3.30.*

Rem. 2.3.51 $\succ$ sufficient to consider LU-factorization without pivoting

A profound roundoff analysis of Gaussian eliminatin/LU-factorization can be found in [**?**, Sect. 3.3 & 3.5] and [**?**, Sect. 9.3]. A less rigorous, but more lucid discussion is given in [**?**, Lecture 22].

Here we only quote a result due to Wilkinson, [**?**, Thm. 9.5]:

Theorem 2.4.4. Stability of Gaussian elimination with partial pivoting

Let $\mathbf{A} \in \mathbb{R}^{n,n}$ be regular and $\mathbf{A}^{(k)} \in \mathbb{R}^{n,n}$, $k = 1, \ldots, n-1$, denote the intermediate matrix arising in the k-th step of § 2.3.43 (Gaussian elimination with partial pivoting) when carried out with exact arithmetic.
For the approximate solution $\widetilde{\mathbf{x}} \in \mathbb{R}^n$ of the LSE $\mathbf{Ax} = \mathbf{b}$, $\mathbf{b} \in \mathbb{R}^n$, computed as in § 2.3.43 (based on machine arithmetic with machine precision EPS, → Ass. 1.5.32) there is $\Delta\mathbf{A} \in \mathbb{R}^{n,n}$ with

$$\|\Delta\mathbf{A}\|_\infty \leq n^3 \frac{3\mathrm{EPS}}{1-3n\mathrm{EPS}}\,\rho\,\|\mathbf{A}\|_\infty\,, \quad \rho := \frac{\max\limits_{i,j,k} |(\mathbf{A}^{(k)})_{ij}|}{\max\limits_{i,j} |(\mathbf{A})_{ij}|}\,,$$

such that $(\mathbf{A} + \Delta\mathbf{A})\widetilde{\mathbf{x}} = \mathbf{b}$.

ρ "small" ➥ Gaussian elimination with partial pivoting is stable (→ Def. 1.5.85)

If ρ is "small", the computed solution of a LSE can be regarded as the exact solution of a LSE with "slightly perturbed" system matrix (perturbations of size $O(n^3\mathrm{EPS})$).

Bad news: exponential growth $\rho \sim 2^n$ is possible !

Example 2.4.5 (Wilkinson's counterexample)

$$a_{ij} = \begin{cases} 1 & \text{, if } i = j \vee j = n\,, \\ -1 & \text{, if } i > j\,, \\ 0 & \text{else.} \end{cases}\,, \qquad \text{n=10:} \quad \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 \end{bmatrix}$$

Partial pivoting does not trigger row permutations !

➤ $\mathbf{A} = \mathbf{LU}$, $l_{ij} = \begin{cases} 1 & \text{, if } i = j\,, \\ -1 & \text{, if } i > j\,, \\ 0 & \text{else} \end{cases}$ $u_{ij} = \begin{cases} 1 & \text{, if } i = j\,, \\ 2^{i-1} & \text{, if } j = n\,, \\ 0 & \text{else.} \end{cases}$

➤ Exponential blow-up of entries of $\mathbf{U}$!

Blow-up of entries of $\mathbf{U}$!
$\updownarrow (*)$
However, $\mathrm{cond}_2(\mathbf{A})$ is small!

➤ Evidence of Instability of Gaussian elimination!

C++11 code 2.4.6: Gaussian elimination for "Wilkinson system" in EIGEN ↪ GITLAB

```
MatrixXd res(100,2);
for(int n = 10; n <= 100*10; n += 10){
      MatrixXd A(n,n); A.setIdentity();
      A.triangularView<StrictlyLower>().setConstant(-1);
      A.rightCols<1>().setOnes();
      VectorXd x = VectorXd::Constant(n,-1).binaryExpr(
         VectorXd::LinSpaced(n,1,n), [](double x, double y){return
         pow(x,y);});
      double relerr = (A.lu().solve(A*x)-x).norm()/x.norm();
      res(n/10-1,0) = n; res(n/10-1,1) = relerr;
}
// ... different solver(e.g. colPivHouseholderQr()), plotting
```

$(*)$ If $\mathrm{cond}_2(\mathbf{A})$ was huge, then big errors in the solution of a linear system can be caused by small perturbations of either the system matrix or the right hand side vector, see (2.4.12) and the message of Thm. 2.2.10, (2.2.13). In this case, a stable algorithm can obviously produce a grossly "wrong" solution, as was already explained after (2.2.13).
Hence, lack of stability of Gaussian elimination will only become apparent for linear systems with well-conditioned system matrices.

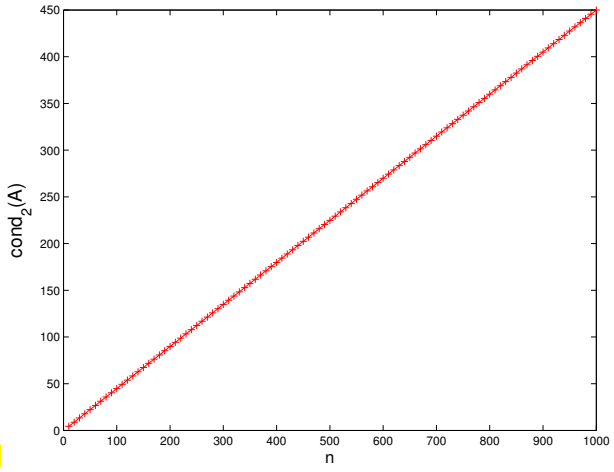


Fig. 58

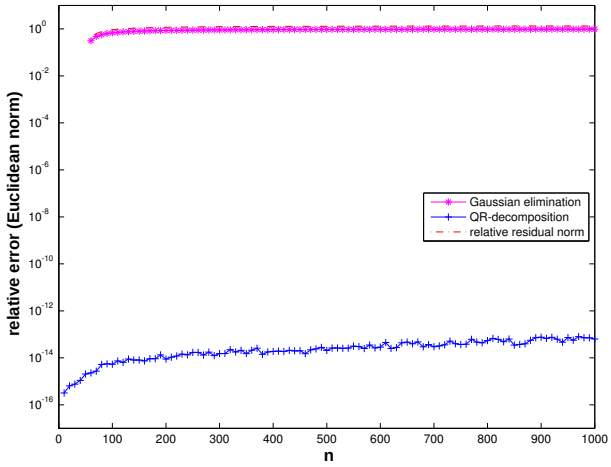


Fig. 59

These observations match Thm. 2.4.4, because in this case we encounter an exponential growth of $\rho = \rho(n)$, see Ex. 2.4.5.

Observation: In practice ρ (almost) always grows only mildly (like $O(\sqrt{n})$) with n

Discussion in [**?**, Lecture 22]: growth factors larger than the order$O(\sqrt{n})$ are exponentially rare in certain relevant classes of *random matrices*.

Example 2.4.7 (Stability by small random perturbations)

C++11 code 2.4.8: Stability by small random perturbations ↪ GITLAB

```
//! Curing Wilkinson's counterexample by random perturbation
```

```
//!  Theory:  Spielman and Teng
MatrixXd res(20,3);
mt19937 gen(42); // seed
std::normal_distribution<> d; // normal distribution, mean = 0.0,
   stddev = 1.0
for(int n = 10; n <= 10*20; n += 10){
      // Build Wilkinson matrix
      MatrixXd A(n,n); A.setIdentity();
      A.triangularView<StrictlyLower>().setConstant(-1);
      A.rightCols<1>().setOnes();
      // imposed solution
      VectorXd x =
         VectorXd::Constant(n,-1).binaryExpr(VectorXd::LinSpaced(n,1,n),
         [](double x, double y){return pow(x,y);});
      double relerr = (A.lu().solve(A*x)-x).norm()/x.norm();
      //Randomly perturbed Wilkinson matrix by matrix with iid
      // N(0,eps) distributed entries
      MatrixXd Ap = A.unaryExpr( [&](double x){ return x +
         numeric_limits<double>::epsilon()*d(gen);} );
      double relerrp = (Ap.lu().solve(Ap*x)-x).norm()/x.norm();
      res(n/10-1,0) = n; res(n/10-1,1) = relerr; res(n/10-1,2) =
         relerrp;
}
// Plotting
```

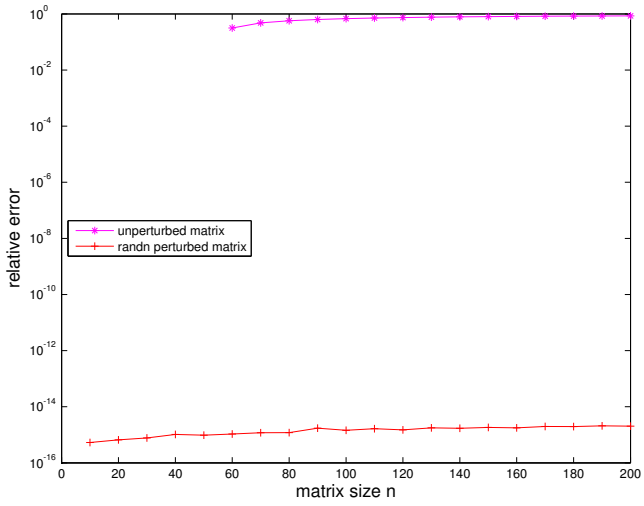


Fig. 60

Recall the statement made above about "improbability" of matrices for which Gaussian elimination with partial pivoting is unstable. This is now matched by the observation that a *tiny random* perturbation of the matrix (almost certainly) cures the problem. This is investigated by the brand-new field of smoothed analysis of numerical algorithms, see [**?**].

Gaussian elimination/LU-factorization with partial pivoting is stable $(*)$
(for all practical purposes) !

$(*)$: stability refers to maximum norm $\|\cdot\|_\infty$.

Experiment 2.4.9 (Conditioning and relative error → Ex. 2.4.10 cnt'd)

In the discussion of numerical stability (→ Def. 1.5.85, Rem. 1.5.88) we have seen that a stable algorithm may produce results with large errors for ill-conditioned problems. The conditioning of the problem of

solving a linear system of equations is determined by the condition number (→ Def. 2.2.12) of the system matrix, see Thm. 2.2.10.

Hence, for an ill-conditioned linear system, whose system matrix is beset with a huge condition number, (stable) Gaussian elimination may return "solutions" with large errors. This will be demonstrated in this experiment.

Numerical experiment with *nearly singular matrix* from Ex. 2.4.10

$$\mathbf{A} = \mathbf{u}\mathbf{v}^T + \epsilon\mathbf{I}\,,$$
$$\mathbf{u} = \tfrac{1}{3}(1,2,3,\ldots,10)^T\,,$$
$$\mathbf{v} = (-1,\tfrac{1}{2},-\tfrac{1}{3},\tfrac{1}{4},\ldots,\tfrac{1}{10})^T$$

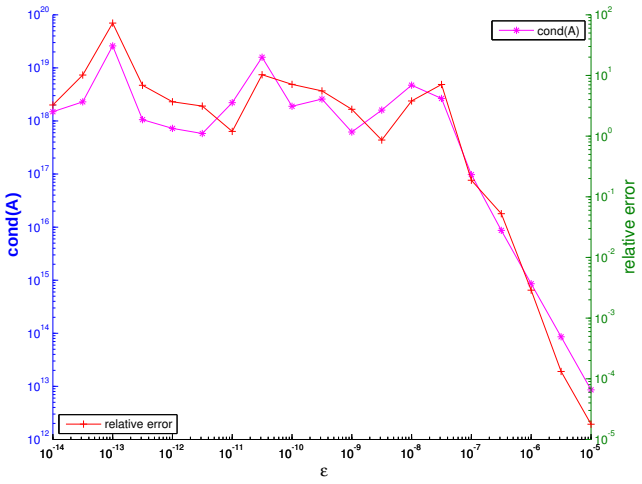


Fig. 61

The practical stability of Gaussian elimination is reflected by the size of a particular vector that can easily be computed after the elimination solver has finished:

In practice *Gaussian elimination/LU-factorization with partial pivoting* produces "relatively small residuals"

Simple consideration as in § 2.4.2:

$$(\mathbf{A}+\Delta\mathbf{A})\widetilde{\mathbf{x}} = \mathbf{b} \quad\Rightarrow\quad \mathbf{r} = \mathbf{b} - \mathbf{A}\widetilde{\mathbf{x}} = \Delta\mathbf{A}\widetilde{\mathbf{x}} \quad\Rightarrow\quad \|\mathbf{r}\| \le \|\Delta\mathbf{A}\|\|\widetilde{\mathbf{x}}\|\,,$$

for any vector norm $\|\cdot\|$. This means that, if a direct solver for an LSE is stable in the sense of backward error analysis, that is, the perturbed solution could be obtained as the exact solution for a slightly relatively perturbed system matrix, then the *residual will be (relatively) small*.

Experiment 2.4.10 (Small residuals by Gaussian elimination)

Numerical experiment with *nearly singular matrix*

$$\mathbf{A} = \mathbf{u}\mathbf{v}^T + \epsilon\mathbf{I}\,,$$

singular rank-1 matrix

with

$$\mathbf{u} = \tfrac{1}{3}(1,2,3,\ldots,10)^T\,,$$
$$\mathbf{v} = (-1,\tfrac{1}{2},-\tfrac{1}{3},\tfrac{1}{4},\ldots,\tfrac{1}{10})^T$$

C++11 code 2.4.11: Small residuals for Gauss elimination ↪ GITLAB

```
int n = 10;
VectorXd u = VectorXd::LinSpaced(n,1,n) / 3.0;
VectorXd v = u.cwiseInverse().array() *
   VectorXd::LinSpaced(n,1,n).unaryExpr( [](double x){return
   pow(-1,x);} ).array();
```

```
5  VectorXd x = VectorXd::Ones(n);
6  double nx = x.lpNorm<Infinity>();
7  VectorXd expo = VectorXd::LinSpaced(19,-5,-14      );
8  Eigen::MatrixXd res(expo.size(),4);
9  for(int i = 0; i <= expo.size(); ++i){
10       double epsilon = std::pow(10, expo(i));
11       MatrixXd A = u*v.transpose() + epsilon * MatrixXd::Identity(n,n);
12       VectorXd b = A * x;
13       double nb = b.lpNorm<Infinity>();
14       VectorXd xt = A.lu().solve(b);
15       VectorXd r = b - A*xt;
16       res(i,0) = epsilon; res(i,1) = (x-xt).lpNorm<Infinity>()/nx;
17       res(i,2) = r.lpNorm<Infinity>()/nb;
18       // L-infinity condition number
19       res(i,3) = A.inverse().cwiseAbs().rowwise().sum().maxCoeff() *
            A.cwiseAbs().rowwise().sum().maxCoeff();
20 }
```

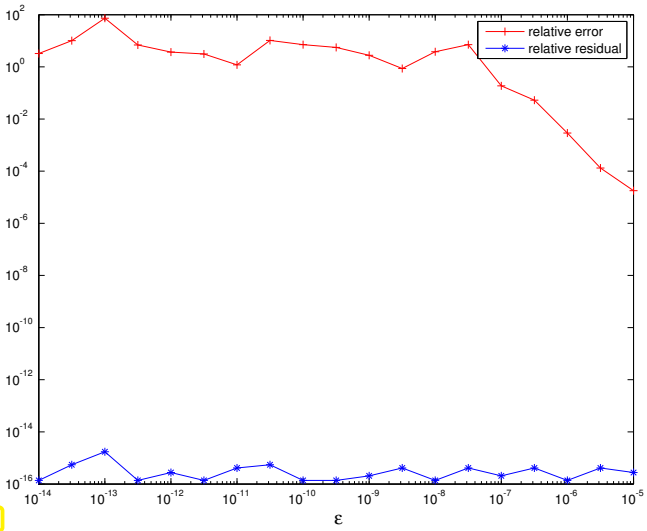


Fig. 62

Observations (w.r.t $\|\cdot\|_\infty$-norm)

- for $\epsilon \ll 1$ large relative error in computed solution $\widetilde{\mathbf{x}}$
- small residuals for any ϵ

How can a *large* relative error be reconciled with a *small* relative residual **?**

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \leftrightarrow \quad \mathbf{A}\widetilde{\mathbf{x}} \approx \mathbf{b}$$

$$\left\{ \begin{array}{lll} \mathbf{A}(\mathbf{x}-\widetilde{\mathbf{x}}) = \mathbf{r} & \Rightarrow & \|\mathbf{x}-\widetilde{\mathbf{x}}\| \le \left\|\mathbf{A}^{-1}\right\|\|\mathbf{r}\| \\ \mathbf{A}\mathbf{x} = \mathbf{b} & \Rightarrow & \|\mathbf{b}\| \le \|\mathbf{A}\|\|\mathbf{x}\| \end{array} \right. \quad \Rightarrow \quad \frac{\|\mathbf{x}-\widetilde{\mathbf{x}}\|}{\|\mathbf{x}\|} \le \|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\|\frac{\|\mathbf{r}\|}{\|\mathbf{b}\|}\,. \tag{2.4.12}$$

➢ If $\mathrm{cond}(\mathbf{A}) := \|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\| \gg 1$, then a small relative residual may not imply a small relative error. Also recall the discussion in Exp. 2.4.9.

Experiment 2.4.13 (Instability of multiplication with inverse)

An important justification for Rem. 2.2.6 is conveyed by this experiment. We again consider the nearly singular matrix from Ex. 2.4.10.

$$\mathbf{A} = \mathbf{u}\mathbf{v}^T + \epsilon\mathbf{I}\,,$$

singular rank-1 matrix

with $\mathbf{u} = \frac{1}{3}(1,2,3,\ldots,10)^T\,,$
$\mathbf{v} = (-1,\frac{1}{2},-\frac{1}{3},\frac{1}{4},\ldots,\frac{1}{10})^T$

C++11 code 2.4.14: Instability of multiplication with inverse ↬ GITLAB

```
int n = 10;
VectorXd u = VectorXd::LinSpaced(n,1,n) / 3.0;
VectorXd v = u.cwiseInverse().array() *
   VectorXd::LinSpaced(n,1,n).unaryExpr( [](double x){return
   pow(-1,x);} ).array();
VectorXd x = VectorXd::Ones(n);
VectorXd expo = VectorXd::LinSpaced(19,-5,-14);
MatrixXd res(expo.size(),4);
for(int i = 0; i < expo.size(); ++i){
      double epsilon = std::pow(10, expo(i));
      MatrixXd A = u*v.transpose() + epsilon * (MatrixXd::Random(n,n)
         + MatrixXd::Ones(n,n))/2;
      VectorXd b = A * x;
      double nb = b.lpNorm<Infinity>();
      VectorXd xt = A.lu().solve(b);      // stable solving
      VectorXd r = b - A*xt;    // residual
      MatrixXd B = A.inverse();
      VectorXd xi = B*b;  // solving via inverse
      VectorXd ri = b - A*xi; // residual
      MatrixXd R = MatrixXd::Identity(n,n) - A*B; // residual
      res(i,0) = epsilon; res(i,1) = (r).lpNorm<Infinity>()/nb;
      res(i,2) = ri.lpNorm<Infinity>()/nb;
      // L-infinity condition number
      res(i,3) = R.lpNorm<Infinity>() / B.lpNorm<Infinity>();
}
```

Computation of the inverse $\mathbf{B} := \texttt{inv}(\mathbf{A})$ is affected by roundoff errors, but does not benefit from favorable compensation of roundoff errors as does Gaussian elimination.

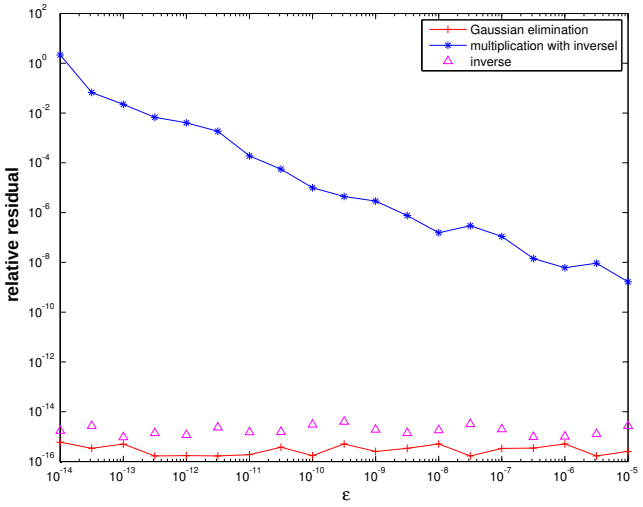


Fig. 63

2.5 Survey: Elimination solvers for linear systems of equations

All direct ($*$) solver algorithms for square linear systems of equations $\mathbf{Ax} = \mathbf{b}$ with given matrix $\mathbf{A} \in \mathbb{K}^{n,n}$, right hand side vector $\mathbf{b} \in \mathbb{K}^n$ and unknown $\mathbf{x} \in \mathbb{K}^n$ rely on variants of Gaussian elimination with pivoting, see Section 2.3.3. Sophisticated, optimised and verified implementations are available in numerical libraries like LAPACK/MKL.

($*$): a direct solver terminates after a predictable finite number of elementary operations for every admissible input.

Never contemplate implementing a general solver for linear systems of equations!

If possible, use algorithms from numerical libraries! ($\rightarrow$ Exp. 2.3.7)

Therefore, familiarity with details of Gaussian elimination is not required, but one must know when and how to use the library functions and one must be able to assess the computational effort they involve.

(2.5.1) Computational effort for direct elimination

We repeat the reasoning of § 2.3.5: Gaussian elimination for a general (dense) matrix invariably involves *three nested loops* of length n, see Code 2.3.4, Code 2.3.41.

Theorem 2.5.2. Cost of Gaussian elimination $\rightarrow$ **§ 2.3.5**

Given a linear system of equations $\mathbf{Ax} = \mathbf{b}$m $\mathbf{A} \in \mathbb{K}^{n,n}$ regular, $\mathbf{b} \in]\mathbb{K}^n$, $n \in \mathbb{N}$, the asymptotic computational effort ($\rightarrow$ Def. 1.4.1) for its direct solution by means of Gaussian elimination in terms of the problem size parameter n is $O(n^3)$ for $n \rightarrow \infty$.

The constant hidden in the Landau symbol can be expected to be rather small (≈ 1) as is clear from (2.3.6).

The cost for solving are substantially lower, if certain properties of the matrix $\mathbf{A}$ are known. This is clear, if $\mathbf{A}$ is diagonal or orthogonal/unitary. It is also true for triangular matrices ($\rightarrow$ Def. 1.1.5), because they can be solved by simple back substitution or forward elimination. We recall the observation made in see § 2.3.30.

Theorem 2.5.3. Cost for solving triangular systems $\rightarrow$ **§ 2.3.5**

In the setting of Thm. 2.5.2, the asymptotic computational effort for solving a triangular linear system of equations is $O(n^2)$ for $n \rightarrow \infty$.

(2.5.4) Direct solution of linear systems of equations in EIGEN

Given: system matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ regular $\leftrightarrow$ A ($n \times n$ EIGEN matrix)
right hand side vectors $\mathbf{B} \in \mathbb{K}^{n,\ell}$ $\leftrightarrow$ B ($n \times \ell$ EIGEN matrix)
(corresponds to multiple right hand sides, *cf.* Code 2.3.10)

linear algebra	EIGEN
$\mathbf{X} = \mathbf{A}^{-1}\mathbf{B} = \left[\mathbf{A}^{-1}(\mathbf{B})_{:,1}, \ldots, \mathbf{A}^{-1}(\mathbf{B})_{:,\ell}\right]$	`X = A.lu().solve(B)`

From detailed information given in § 2.3.30:

$$\text{cost(`X = A.lu().solve(B)`)} \quad = \quad = O(n^3 + ln^2) \quad \text{for } n, l \to \infty$$

Remark 2.5.5 (Communicating special properties of system matrices in EIGEN)

Sometimes, the coefficient matrix of a linear system of equations is known to have certain analytic properties that a direct solver can exploit to perform elimination more efficiently. These properties may even be impossible to detect by an algorithm, because matrix entries that should vanish exactly might have been perturbed due to roundoff.

Thus one needs to pass EIGEN these informations as follows:

```
// A is lower triangular
x = A.triangularView<Eigen::Lower>().solve(b);
// A is upper triangular
x = A.triangularView<Eigen::Upper>().solve(b);
// A is hermitian / self adjoint and positive definite
x = A.selfadjointView<Eigen::Upper>().llt().solve(b);
// A is hermiatin / self adjoint (poitive or negative semidefinite)
x = A.selfadjointView<Eigen::Upper>().ldlt().solve(b);
```

Experiment 2.5.6 (Standard EIGEN lu() operator versus triangularView())

In this numerical experiment we study the gain in efficiency achievable by make the direct solver aware of important matrix properties.

C++11 code 2.5.7: Direct solver applied to a upper triangular matrix ↪ GITLAB

```
//!  Eigen code:  assessing the gain from using special properties
//!  of system matrices in Eigen
MatrixXd timing(){
    std::vector<int> n = {16,32,64,128,256,512,1024,2048,4096,8192};
    int nruns = 3;
    MatrixXd times(n.size(),3);
    for(int i = 0; i < n.size(); ++i){
        Timer t1, t2;  // timer class
        MatrixXd A = VectorXd::LinSpaced(n[i],1,n[i]).asDiagonal();
        A += MatrixXd::Ones(n[i],n[i]).triangularView<Upper>();
        VectorXd b = VectorXd::Random(n[i]);
        VectorXd x1(n[i]), x2(n[i]);
        for(int j = 0; j < nruns; ++j){
            t1.start();    x1 = A.lu().solve(b);    t1.stop();
            t2.start();    x2 =
                A.triangularView<Upper>().solve(b); t2.stop();
```

```
            }
            times(i,0) = n[i]; times(i,1) = t1.min(); times(i,2) =
                t2.min();
        }
        return times;
}
```

Observation: ▷

Being told that only the upper triangular part of the matrix needs to be taken into account, Gaussian elimination reduces to cheap backward elimination, which is much faster than full elimination.

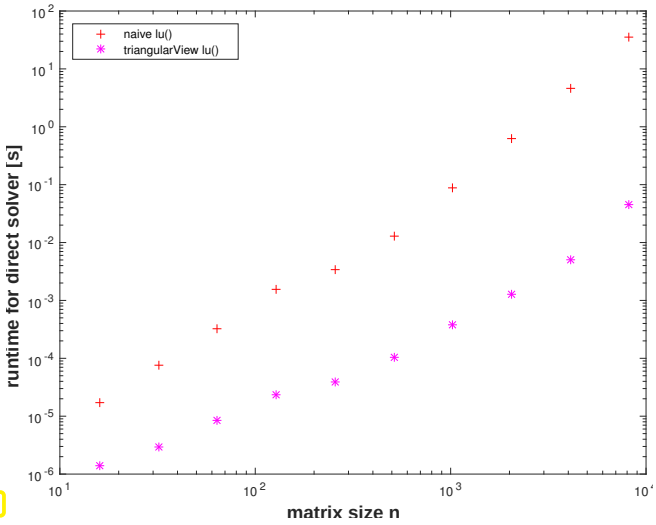


Fig. 64

(2.5.8) Direct solvers for LSE in EIGEN

Invocation of direct solvers in EIGEN is a two stage process:

❶ Request a decomposition (LU,QR,LDLT) of the matrix and store it in a temporary "decomposition object".

❷ Perform backward & forward substitutions by calling the **solve()** method of the decomposition object.

The general format for invoking linear solvers in EIGEN is as follows:

```
Eigen::SolverType<Eigen::MatrixXd> solver(A);
Eigen::VectorXd x = solver.solve(b);
```

This can be reduced to one line, as the solvers can also be used as methods acting on matrices:

```
Eigen::VectorXd x = A.solverType().solve(b);
```

A full list of solvers can be found here, in the EIGEN documentation. The next code demonstrates a few of the available decompositions that can serve as the basis for a linear solver:

C++-code 2.5.9: EIGEN based function solving a LSE ⇝ GITLAB

```
// Gaussian elimination with partial pivoting, Code 2.3.41
void lu_solve(const MatrixXd& A, const VectorXd& b, VectorXd& x) {
  x = A.lu().solve(b); // 'lu()' is short for 'partialPivLu()'
}

// Gaussian elimination with total pivoting
void fullpivlu_solve(const MatrixXd& A, const VectorXd& b, VectorXd&
```

```
    x) {
  x = A.fullPivLu().solve(b); // total pivoting
}

// An elimination solver based on Householder transformations
void qr_solve(const MatrixXd& A, const VectorXd& b, VectorXd& x) {
  Eigen::HouseholderQR<MatrixXd> solver(A);
  x = solver.solve(b);
}

// Use singular value decomposition (SVD)
void svd_solve(const MatrixXd& A, const VectorXd& b, VectorXd& x) {
  x = A.jacobiSvd(Eigen::ComputeThinU | Eigen::ComputeThinV).solve(b);
}
```

The different decompositions trade speed for stability and accuracy: fully pivoted and QR-based decompositions also work for nearly singular matrices, for which the standard LU-factorization may non longer be reliable.

Remark 2.5.10 (Many sequential solutions of LSE)

Both EIGEN and MATLAB provide functions that return decompositions of matrices, here the LU-decomposition (→ Section 2.3.2):

EIGEN : MatrixXd A(n,n); auto ludec = mat.lu();
MATLAB: [L,U] = lu(A)

Based on the *precomputed decompositions*, a linear system of equations with coefficient matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ can be solved with asymptotic computational effort $O(n^2)$, *cf.* § 2.3.30.

The following example illustrates a special situation, in which matrix decompositions can curb computational cost:

C++11 code 2.5.11: Wasteful approach!
➙ GITLAB

```
// Setting:  N ≫ 1,
// large matrix A ∈ K^{n,n}
for(int j = 0; j < N; ++j){
    x = A.lu().solve(b);
    b = some_function(x);
}
```

computational effort $O(Nn^3)$

C++11 code 2.5.12: Smart approach!
➙ GITLAB

```
// Setting:  N ≫ 1,
// large matrix A ∈ K^{n,n}
auto A_lu_dec = A.lu();
for(int j = 0; j < N; ++j){
    x = A_lu_dec.solve(b);
    b = some_function(x);
}
```

computational effort $O(n^3 + Nn^2)$

A concrete example is the so-called inverse power iteration, see Chapter 9, for which a skeleton code is

given next. It computes the iterates

$$\mathbf{x}^* := \mathbf{A}^{-1}\mathbf{x}^{(k)} \;, \quad \mathbf{x}^{(k+1)} := \frac{\mathbf{x}^*}{\|\mathbf{x}^*\|_2} \;, \quad k = 0,1,2,\ldots \;, \tag{2.5.13}$$

C++11-code 2.5.14: Efficient implementation of inverse power method in EIGEN ➙ GITLAB

```
2  template<class VecType, class MatType>
3  VecType invpowit(const Eigen::MatrixBase<MatType> &A,double tol)
4  {
5    using index_t = typename MatType::Index;
6    using scalar_t = typename VecType::Scalar;
7    // Make sure that the function is called with a square matrix
8    const index_t n = A.cols();
9    const index_t m = A.rows();
10   eigen_assert(n == m);
11   // Request LU-decomposition
12   auto A_lu_dec = A.lu();
13   // Initial guess for inverse power iteration
14   VecType xo = VecType::Zero(n);
15   VecType xn = VecType::Random(n);
16   // Normalize vector
17   xn /= xn.norm();
18   // Terminate if relative (normwise) change below threshold
19   while ((xo-xn).norm() > xn.norm()*tol) {
20     xo = xn;
21     xn = A_lu_dec.solve(xo);
22     xn /= xn.norm();
23   }
24   return(xn);
25 }
```

The use of `Eigen::MatrixBase<MatType>` makes it possible to call `invpowit` with an expression argument

```
2  MatrixXd A = MatrixXd::Random(n,n);
3  MatrixXd B = MatrixXd::Random(n,n);
4  VectorXd ev = invpowit<VectorXd>(A+B, tol);
```

This is necessary, because `A+B` will spawn an auxiliary object of a "strange" type determined by the expression template mechanism.

2.6 Exploiting Structure when Solving Linear Systems

By "structure" of a linear system we mean prior knowledge that

- either certain entries of the system matrix vanish,

✦ or the system matrix is generated by a particular formula.

(2.6.1) Triangular linear systems

Triangular linear systems are linear systems of equations whose system matrix is a *triangular matrix* (→ Def. 1.1.5).

Thm. 2.5.3 tells us that (dense) triangular linear systems can be solved by backward/forward elimination with $O(n^2)$ asymptotic computational effort ($n \hat{=}$ number of unknowns) compared to an asymptotic complexity of $O(n^3)$ for solving a generic (dense) linear system of equations (→ Thm. 2.5.2).

This is the simplest case where exploiting special structure of the system matrix leads to faster algorithms for the solution of a special class of linear systems.

(2.6.2) Block elimination

Remember that thanks to the possibility to compute the matrix product in a block-wise fashion (→ § 1.3.15, Gaussian elimination can be conducted on the level of matrix blocks. We recall Rem. 2.3.14 and Rem. 2.3.34.

For $k, \ell \in \mathbb{N}$ consider the block partitioned square $n \times n$, $n := k + \ell$, linear system

$$\begin{bmatrix} \mathbf{A}_{11}\mathbf{A}_{12} \\ \mathbf{A}_{21}\mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}, \quad \begin{array}{l} \mathbf{A}_{11} \in \mathbb{K}^{k,k}, \mathbf{A}_{12} \in \mathbb{K}^{k,\ell}, \mathbf{A}_{21} \in \mathbb{K}^{\ell,k}, \mathbf{A}_{22} \in \mathbb{K}^{\ell,\ell}, \\ \mathbf{x}_1 \in \mathbb{K}^k, \mathbf{x}_2 \in \mathbb{K}^\ell, \mathbf{b}_1 \in \mathbb{K}^k, \mathbf{b}_2 \in \mathbb{K}^\ell. \end{array} \tag{2.6.3}$$

Using block matrix multiplication (applied to the matrix×vector product in (2.6.3)) we find an equivalent way to write the block partitioned linear system of equations:

$$\begin{array}{rcl} \mathbf{A}_{11}\mathbf{x}_1 + \mathbf{A}_{12}\mathbf{x}_2 &=& \mathbf{b}_1, \\ \mathbf{A}_{21}\mathbf{x}_1 + \mathbf{A}_{22}\mathbf{x}_2 &=& \mathbf{b}_2. \end{array} \tag{2.6.4}$$

We assume that $\mathbf{A}_{11}$ is *regular* (invertible) so that we can solve for $\mathbf{x}_1$ from the first equation.

➢quad By elementary algebraic manipulations ("block Gaussian elimination") we find

$$\begin{aligned} \mathbf{x}_1 &= \mathbf{A}_{11}^{-1}(\mathbf{b}_1 - \mathbf{A}_{12}\mathbf{x}_2), \\ \underbrace{(\mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12})}_{\text{Schur complement}}\mathbf{x}_2 &= \mathbf{b}_2 - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{b}_1, \end{aligned}$$

The resulting $\ell \times \ell$ linear system of equations for the unknown vector $\mathbf{x}_2$ is called the Schur complement system for (2.6.3).

Unless $\mathbf{A}$ has a special structure that allows the efficient solution of linear systems with system matrix $\mathbf{A}_{11}$, the Schur complement system is mainly of theoretical interest.

Example 2.6.5 (Linear systems with arrow matrices)

From $n \in \mathbb{N}$, a diagonal matrix $\mathbf{D} \in \mathbb{K}^{n,n}$, $\mathbf{c} \in \mathbb{K}^n$, $\mathbf{b} \in \mathbb{K}^n$, and $\alpha \in \mathbb{K}$, we can build an $(n+1) \times (n+1)$ arrow matrix.

$$\mathbf{A} = \begin{bmatrix} & & \\ & \mathbf{D} & \mathbf{c} \\ & & \\ \mathbf{b}^\top & & \alpha \end{bmatrix} \tag{2.6.6}$$

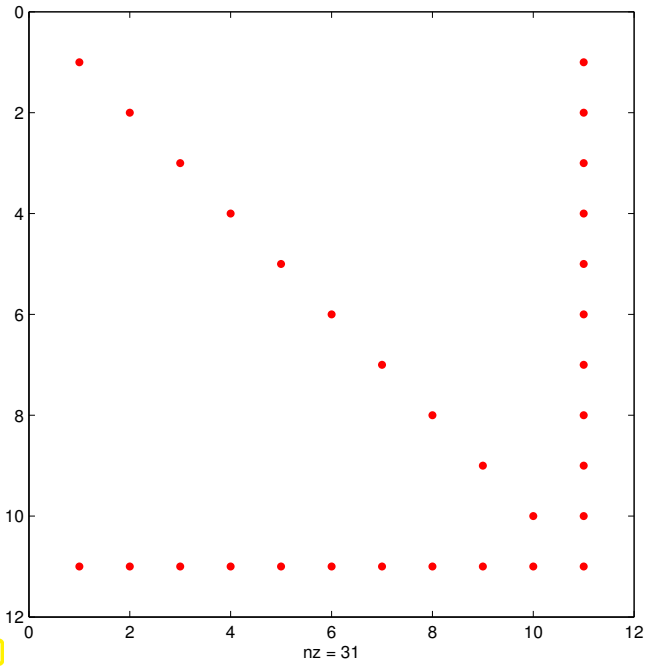


Fig. 65

We can apply the block partitioning (2.6.3) with $k = n$ and $\ell = 1$ to a linear system $\mathbf{Ax} = \mathbf{y}$ with system matrix $\mathbf{A}$ and obtain $\mathbf{A}_{11} = \mathbf{D}$, which can be inverted easily, provided that all diagonal entries of $\mathbf{D}$ are different from zero. In this case

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{D} & \mathbf{c} \\ \mathbf{b}^\top & \alpha \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \xi \end{bmatrix} = \mathbf{y} := \begin{bmatrix} \mathbf{y}_1 \\ \eta \end{bmatrix}, \tag{2.6.7}$$

$$\blacktriangleright \quad \begin{aligned} \xi &= \frac{\eta - \mathbf{b}^T \mathbf{D}^{-1} \mathbf{y}_1}{\alpha - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{c}}, \\ \mathbf{x}_1 &= \mathbf{D}^{-1}(\mathbf{y}_1 - \xi \mathbf{c}) . \end{aligned} \tag{2.6.8}$$

These formulas make sense, if $\mathbf{D}$ is regular *and* $\alpha - \mathbf{b}^\top \mathbf{D}^{-1} \mathbf{c} \neq 0$, which is another condition for the invertibility of $\mathbf{A}$.

Using the formula (2.6.8) we can solve the linear system (2.6.7) with an asymptotic complexity $O(n)$! This superior speed compared to Gaussian elimination applied to the (dense) linear system is evident in runtime measurements.

C++11 code 2.6.9: Dense Gaussian elimination applied to arrow system ↪ GITLAB

```
2  VectorXd arrowsys_slow(const VectorXd &d, const VectorXd &c, const
       VectorXd &b, const double alpha, const VectorXd &y){
3    int n = d.size();
4    MatrixXd A(n + 1,n + 1); A.setZero();
5    A.diagonal().head(n) = d;
6    A.col(n).head(n) = c;
7    A.row(n).head(n) = b;
8    A(n, n) = alpha;
9    return A.lu().solve(y);
10 }
```

Asymptotic complexity $O(n^3)$!
(Due to the serious blunder of accidentally creating a matrix full of zeros, *cf.* Exp. 1.3.10.)

mit samcode-Umgebung

C++11 code 2.6.10: Solving an arrow system according to (2.6.8) ↬ **GITLAB**

```
2  VectorXd arrowsys_fast(const VectorXd &d, const VectorXd &c, const
       VectorXd &b, const double alpha, const VectorXd &y){
3    int n = d.size();
4    VectorXd z = c.array() / d.array();          // z = D^{-1}c
5    VectorXd w = y.head(n).array() / d.array();  // w = D^{-1}y_1
6    double xi = (y(n) - b.dot(w)) / (alpha - b.dot(z));
7    VectorXd x(n+1);
8    x << w - xi*z, xi;
9    return x;
10 }
```

Asymptotic complexity $O(n)$

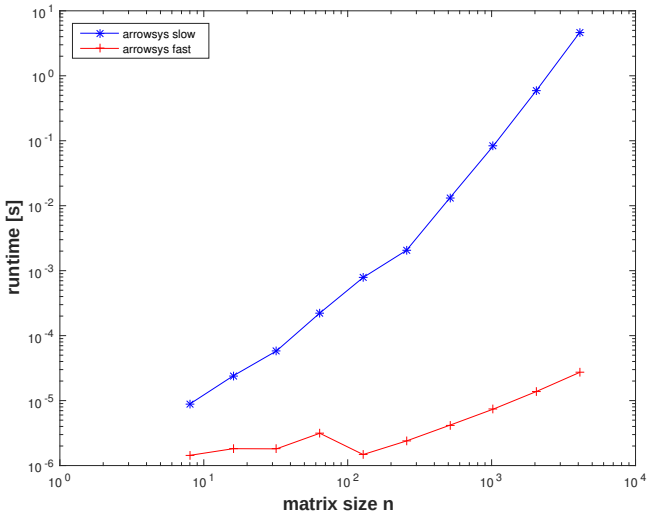
C++11 code 2.6.11: Runtime measurement of Code 2.6.9 vs. Code 2.6.10 vs. sparse techniques ↬ **GITLAB**

```
2  MatrixXd arrowsystiming(){
3    std::vector<int> n = {8,16,32,64,128,256,512,1024,2048,4096};
4    int nruns = 3;
5    MatrixXd times(n.size(),6);
6    for(int i = 0; i < n.size(); ++i){
7      Timer t1, t2, t3, t4;    // timer class
8      double alpha = 2;
9      VectorXd b = VectorXd::Ones(n[i],1);
10     VectorXd c = VectorXd::LinSpaced(n[i],1,n[i]);
11     VectorXd d = -b;
12     VectorXd y = VectorXd::Constant(n[i]+1,-1).binaryExpr(
         VectorXd::LinSpaced(n[i]+1,1,n[i]+1), [](double x, double
         y){return pow(x,y);} ).array();
13     VectorXd x1(n[i]+1), x2(n[i]+1), x3(n[i]+1), x4(n[i]+1);
14     for(int j = 0; j < nruns; ++j){
15       t1.start();    x1 = arrowsys_slow(d,c,b,alpha,y); t2.stop();
16       t2.start();    x2 = arrowsys_fast(d,c,b,alpha,y); t2.stop();
17       t3.start();
18       x3 = arrowsys_sparse<SparseLU<SparseMatrix<double> >
           >(d,c,b,alpha,y);
19       t3.stop();
20       t4.start();
21       x4 = arrowsys_sparse<BiCGSTAB<SparseMatrix<double> >
           >(d,c,b,alpha,y);
22       t4.stop();
23     }
24     times(i,0)=n[i]; times(i,1)=t1.min(); times(i,2)=t2.min();
25     times(i,3)=t3.min();times(i,4)=t4.min();times(i,5)=(x4-x3).norm();
26   }
```

```
27     return times;
28 }
```

(Intel i7-3517U CPU @ 1.90GHz×4, 64-bit, ubuntu 14.04 LTS, gcc 4.8.4, -O3)

No comment! ▷



Remark 2.6.12 (Sacrificing numerical stability for efficiency)

The vector based implementation of the solver of Code 2.6.10 can be vulnerable to roundoff errors, because, upon closer inspection, the algorithm turns out to be equivalent to Gaussian elimination without pivoting, *cf.* Section 2.3.3, Ex. 2.3.36.

Caution: stability at risk in Code 2.6.10

Yet, there are classes of matrices for which Gaussian elimination without pivoting is guaranteed to be stable. For such matrices algorithms like that of Code 2.6.10 are safe.

(2.6.13) Solving LSE with low-rank perturbation of system matrix

Given a regular matrix $\mathbf{A} \in \mathbb{K}^{n,n}$, let us assume that at some point in a code we are in a position to solve any linear system $\mathbf{Ax} = \mathbf{b}$ "fast", because

- either $\mathbf{A}$ has a favorable structure, eg. triangular, see § 2.6.1,
- or an LU-decomposition of $\mathbf{A}$ is already available, see § 2.3.30.

Now, a $\widetilde{\mathbf{A}}$ is obtained by changing a *single entry* of $\mathbf{A}$:

$$\mathbf{A}, \widetilde{\mathbf{A}} \in \mathbb{K}^{n,n}: \quad \widetilde{a}_{ij} = \begin{cases} a_{ij} & \text{, if } (i,j) \neq (i^*,j^*) \text{ ,} \\ z + a_{ij} & \text{, if } (i,j) = (i^*,j^*) \text{ ,} \end{cases} \quad , \quad i^*, j^* \in \{1,\ldots,n\} \text{ .} \tag{2.6.14}$$

$$\widetilde{\mathbf{A}} = \mathbf{A} + z \cdot \mathbf{e}_{i^*} \mathbf{e}_{j^*}^T \text{ .} \tag{2.6.15}$$

(Recall: $\mathbf{e}_i \triangleq i$-th unit vector.) The question is whether we can reuse some of the computations spent on solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ in order to solve $\widetilde{\mathbf{A}}\widetilde{\mathbf{x}} = \mathbf{b}$ with less effort than entailed by a direct Gaussian elimination from scratch.

We may also consider a matrix modification affecting a single row: Changing a single row: given $\mathbf{z} \in \mathbb{K}^n$

$$\mathbf{A}, \widetilde{\mathbf{A}} \in \mathbb{K}^{n,n}: \quad \widetilde{a}_{ij} = \begin{cases} a_{ij} & \text{, if } i \neq i^* \text{ ,} \\ (\mathbf{z})_j + a_{ij} & \text{, if } i = i^* \text{ ,} \end{cases} \quad , \quad i^*, j^* \in \{1, \ldots, n\} \text{ .}$$

$$\Longrightarrow \quad \boxed{\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{e}_{i^*}\mathbf{z}^T} \text{ .} \tag{2.6.16}$$

Both matrix modifications (2.6.14) and (2.6.16) represent rank-1-modifications of $\mathbf{A}$. A generic rank-1-modification reads

$$\mathbf{A} \in \mathbb{K}^{n,n} \quad \mapsto \quad \widetilde{\mathbf{A}} := \mathbf{A} + \boxed{\mathbf{u}\mathbf{v}^H} \text{ , } \mathbf{u}, \mathbf{v} \in \mathbb{K}^n \text{ .} \tag{2.6.17}$$

general rank-1-matrix

Idea: Block elimination of an extended linear system, see § 2.6.2

We consider the block partitioned linear system

$$\begin{bmatrix} \mathbf{A} & \mathbf{u} \\ \mathbf{v}^{\mathrm{H}} & -1 \end{bmatrix} \begin{bmatrix} \widetilde{\mathbf{x}} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix} \text{ .} \tag{2.6.18}$$

The Schur complement system after elimination of ξ reads

$$(\mathbf{A} + \mathbf{u}\mathbf{v}^{\mathrm{H}})\widetilde{\mathbf{x}} = \mathbf{b} \quad \Leftrightarrow \quad \widetilde{\mathbf{A}}\widetilde{\mathbf{x}} = \mathbf{b} \text{ . } \quad ! \tag{2.6.19}$$

Hence, we have solved the modified LSE, once we have found the component $\widetilde{\mathbf{x}}$ of the solution of the extended linear system (2.6.18). We do block elimination again, now getting rid of $\widetilde{\mathbf{x}}$ first, which yields the other Schur complement system

$$(1 + \mathbf{v}^{\mathrm{H}}\mathbf{A}^{-1}\mathbf{u})\xi = \mathbf{v}^{\mathrm{H}}\mathbf{A}^{-1}\mathbf{b} \text{ .} \tag{2.6.20}$$

$$\Longrightarrow \quad \mathbf{A}\widetilde{\mathbf{x}} = \mathbf{b} - \frac{\mathbf{u}\mathbf{v}^{\mathrm{H}}\mathbf{A}^{-1}}{1 + \mathbf{v}^{\mathrm{H}}\mathbf{A}^{-1}\mathbf{u}}\mathbf{b} \text{ .} \tag{2.6.21}$$

The generalization of this formula to rank-k-perturbations if given in the following lemma:

Lemma 2.6.22. Sherman-Morrison-Woodbury formula

For regular $\mathbf{A} \in \mathbb{K}^{n,n}$, and $\mathbf{U}, \mathbf{V} \in \mathbb{K}^{n,k}$, $n, k \in \mathbb{N}$, $k \leq n$, holds

$$(\mathbf{A} + \mathbf{U}\mathbf{V}^H)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^H\mathbf{A}^{-1} \text{ ,}$$

if $\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U}$ *is regular.*

Proof. Straightforward algbra:

$$\left(\mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}(\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})^{-1}\mathbf{V}^H\mathbf{A}^{-1}\right)(\mathbf{A} + \mathbf{U}\mathbf{V}^H) =$$

$$\mathbf{I} - \mathbf{A}^{-1}\mathbf{U}\underbrace{(\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})^{-1}(\mathbf{I} + \mathbf{V}^H\mathbf{A}^{-1}\mathbf{U})}_{=\mathbf{I}}\mathbf{V}^H + \mathbf{A}^{-1}\mathbf{U}\mathbf{V}^H = \mathbf{I}\,.$$

Uniqueness of the inverse settles the case. □

We use this result to solve $\widetilde{\mathbf{A}}\widetilde{\mathbf{x}} = \mathbf{b}$ with $\widetilde{\mathbf{A}}$ from (2.6.17) more efficiently than straightforward elimination could deliver, provided that the LU-factorisation $\mathbf{a} = \mathbf{LU}$ is already known.

Apply Lemma 2.6.22 for $k = 1$:

$$\widetilde{\mathbf{x}} = \mathbf{A}^{-1}\mathbf{b} - \frac{\mathbf{A}^{-1}\mathbf{u}(\mathbf{v}^H(\mathbf{A}^{-1}\mathbf{b}))}{1 + \mathbf{v}^H(\mathbf{A}^{-1}\mathbf{u})}\,. \tag{2.6.23}$$

LU-decomposition of $\mathbf{A}$ is available.

Efficient implementation ! Asymptotic complexity $O(n^2)$ (back substitutions on Line 5-6)

C++11 code 2.6.24: Solving a rank-1 modified LSE ↬ GITLAB

```
2  // Solving rank-1 updated LSE based on (2.6.23)
3  template <class LUDec>
4  VectorXd smw(const LUDec &lu, const MatrixXd &u, const VectorXd &v,
       const VectorXd &b){
5    VectorXd z = lu.solve(b); //
6    VectorXd w = lu.solve(u); //
7    double alpha = 1.0 + v.dot(w);
8    if (std::abs(alpha) < std::numeric_limits<double>::epsilon())
9      throw std::runtime_error("A nearly singular");
10   else return (z - w * v.dot(z) / alpha);
11 }
```

Example 2.6.25 (Resistance to currents map)

Many lineare systems with system matrices that differ in a single entry only have to be solved when we want to determine the dependence of the total impedance of a (linear) circuit from the parameters of a

single component.

Large (linear) electric circuit (modelling → Ex. 2.1.3) ▷

Sought:

Dependence of (certain) branch currents on "continuously varying" resistance R_x

(➢ currents for many different values of R_x)

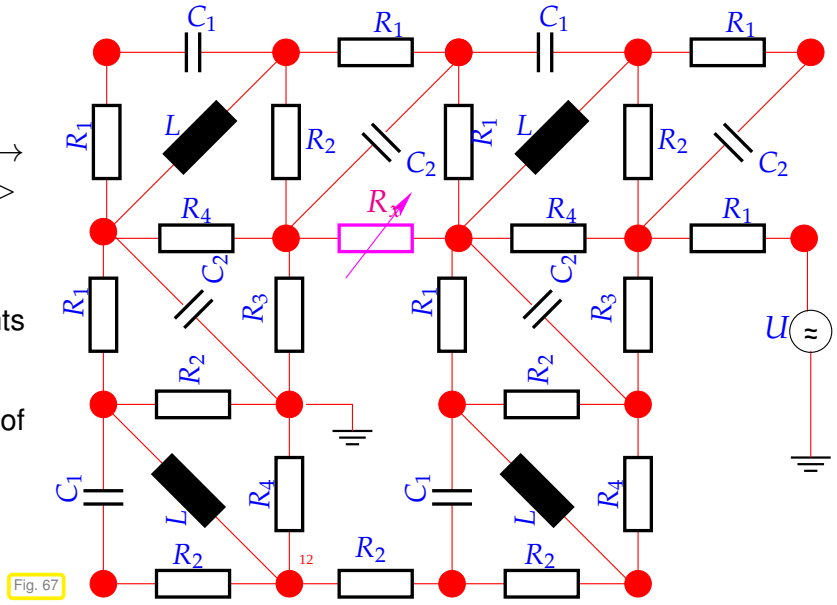


Fig. 67

Only *a few* entries of the nodal analysis matrix $\mathbf{A}$ (→ Ex. 2.1.3) are affected by variation of R_x!
(If R_x connects nodes i & j ⇒ only entries $a_{ii}, a_{jj}, a_{ij}, a_{ji}$ of $\mathbf{A}$ depend on R_x)

2.7 Sparse Linear Systems

A (rather fuzzy) classification of matrices according to their numbers of zeros:

Dense(ly populated) matrices ◆ sparse(ly populated) matrices

Notion 2.7.1. Sparse matrix

$\mathbf{A} \in \mathbb{K}^{m,n}$, $m, n \in \mathbb{N}$, is sparse, if

$$\operatorname{nnz}(\mathbf{A}) := \#\{(i,j) \in \{1,\ldots,m\} \times \{1,\ldots,n\} : a_{ij} \neq 0\} \ll mn \,.$$

Sloppy parlance: matrix sparse :⇔ "almost all" entries $= 0$ /"only a few percent of" entries $\neq 0$

J.H. Wilkinson's informal working definition for a developer of simulation codes:

Notion 2.7.2. Sparse matrix

A matrix with enough zeros that it pays to take advantage of them should be *treated* as sparse.

A more rigorous "mathematical" definition:

Definition 2.7.3. Sparse matrices

Given a strictly increasing sequences $m : \mathbb{N} \mapsto \mathbb{N}$, $n : \mathbb{N} \mapsto \mathbb{N}$, a family $(\mathbf{A}^{(l)})_{l\in\mathbb{N}}$ of matrices with $\mathbf{A}^{(l)} \in \mathbb{K}^{m_l,n_l}$ is sparse (opposite: dense), if

$$\lim_{l\to\infty} \frac{\mathrm{nnz}(\mathbf{A}^{(l)})}{n_l m_l} = 0 \,.$$

Simple example: families of diagonal matrices ($\to$ Def. 1.1.5)

Example 2.7.4 (Sparse LSE in circuit modelling)

See Ex. 2.1.3 for the description of a linear electric circuit by means of a linear system of equations for nodal voltages. For large circuits the system matrices will invariably be huge and sparse.

Modern electric circuits (VLSI chips):
$10^5 - 10^7$ circuit elements

- Each element is connected to only *a few* nodes
- Each node is connected to only *a few* elements

[In the case of a linear circuit]

nodal analysis ➤ sparse circuit matrix
(Impossible to even store as dense matrices)

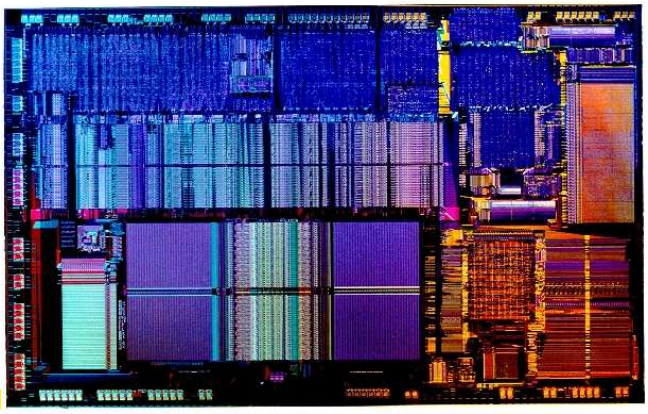


Fig. 68

Remark 2.7.5 (Sparse matrices from the discretization of linear partial differential equations)

Another important context in which sparse matrices usually arise:

☛ spatial discretization of linear boundary value problems for partial differential equations by means of finite element (FE), finite volume (FV), or finite difference (FD) methods ($\to$ 4th semester course "Numerical methods for PDEs").

2.7.1 Sparse matrix storage formats

Sparse matrix storage formats for storing a "sparse matrix" $\mathbf{A} \in \mathbb{K}^{m,n}$ are designed to achieve two objectives:

❶ Amount of memory required is only slightly more than $\mathrm{nnz}(\mathbf{A})$ scalars.

❷ Computational effort for matrix×vector multiplication is proportional to $\mathrm{nnz}(\mathbf{A})$.

In this section we see a few schemes used by numerical libraries.

(2.7.6) Triplet/coordinate list (COO) format

In the case of a sparse matrix $\mathbf{A} \in \mathbb{K}^{m,n}$, this format stores triplets $(i, j, \alpha_{i,j})$, $1 \le i \le m$, $1 \le j \le n$:

```
struct TripletMatrix {
  size_t m,n;          // Number of rows and columns
  vector<size_t> I;    // row indices
  vector<size_t> J;    // column indices
  vector<scalar_t> a; // values associated with index pairs
 };
```

All vectors have the same size $\geq \mathrm{nnz}(\mathbf{a})$.

We write "$\geq$", because *repetitions* of index pairs (i,j) are *allowed*. The matrix entry $(\mathbf{A}))i,j$ is defined to be the sum of all values $\alpha_{i,j}$ associated with the index pair (i,j). The next code clearly demonstrates this summation.

C++-code 2.7.7: Matrix×vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ in triplet format

```
1 void multTriplMatvec(const TripletMatrix &A,
2                       const vector<scalar_t> &x,
3                       vector<scalar_t> &y)
4  for (size_t l=0; l<A.a.size(); l++) {
5    y[A.I[l]] += A.a[l]*x[A.J[l]];
6   }
```

Note that this code assumes that the result vector y has the appropriate length; no index checks are performed.

Code 2.7.7: computational effort is proportional to the number of triplets. (This might be much larger than $\mathrm{nnz}(\mathbf{A})$ in case of many repetitions of triplets.)

(2.7.8) The zoo of sparse matrix formats

Special sparse matrix storage formats store *only* non-zero entries:

- Compressed Row Storage (CRS)
- Compressed Column Storage (CCS) $\rightarrow$ used by MATLAB
- Block Compressed Row Storage (BCRS)
- Compressed Diagonal Storage (CDS)
- Jagged Diagonal Storage (JDS)
- Skyline Storage (SKS)

All of these formats achieve the two objectives stated above. Some have been designed for sparse matrices with additional structure or for seamless cooperation with direct elimination algorithms (JDS,SKS).

Example 2.7.9 (Compressed row-storage (CRS) format)

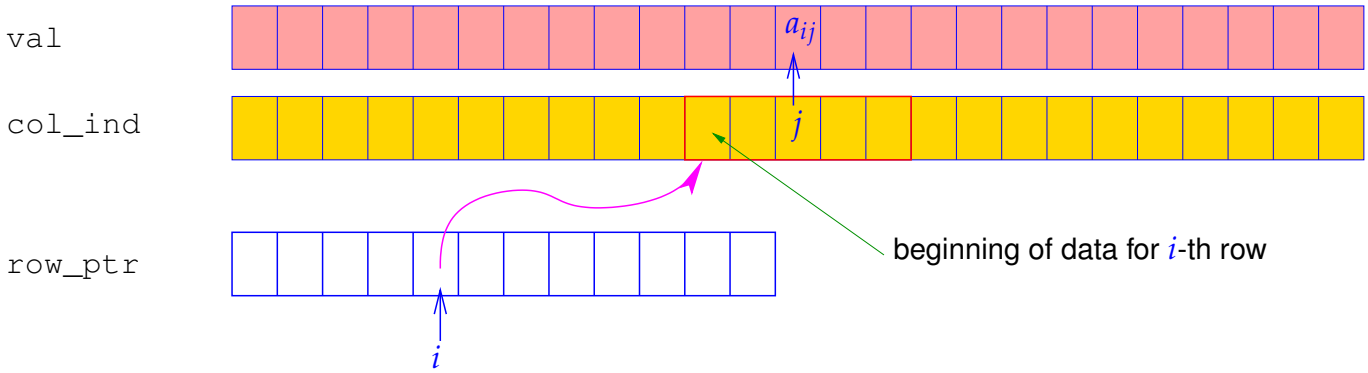
The CRS format for a sparse matrix $\mathbf{A} = (a_{ij}) \in \mathbb{K}^{n,n}$ keeps the data in three *contiguous* arrays:

`vector<scalar_t>`	`val`	size $\mathrm{nnz}(\mathbf{A}) := \#\{(i,j) \in \{1,\ldots,n\}^2, a_{ij} \neq 0\}$
`vector<size_t>`	`col_ind`	size $\mathrm{nnz}(\mathbf{A})$
`vector<size_t>`	`row_ptr`	size $n+1$ & $\texttt{row\_ptr}[n+1] = \mathrm{nnz}(\mathbf{A})+1$ (sentinel value)

As above we write $\mathrm{nnz}(\mathbf{A}) \triangleq$ (**n**umber of **n**on**z**eros) of $\mathbf{A}$

Access to matrix entry $a_{ij} \neq 0$, $1 \leq i,j \leq n$ ("mathematical indexing")

$$\texttt{val}[k] = a_{ij} \quad \Leftrightarrow \quad \begin{cases} \texttt{col\_ind}[k] = j\,, \\ \texttt{row\_ptr}[i] \leq k < \texttt{row\_ptr}[i+1]\,, \end{cases} \qquad 1 \leq k \leq \mathrm{nnz}(\mathbf{A})\,.$$



$$\mathbf{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

`val`-vector:

10	-2	3	9	3	7	8	7	3...9	13	4	2	-1

`col_ind`-array:

1	5	1	2	6	2	3	4	1...5	6	2	5	6

`row_ptr`-array:

1	3	6	9	13	17	20

Variant: diagonal CRS format (matrix diagonal stored in separate array)
The CCS format is equivalent to CRS format for the transposed matrix.

2.7.2 Sparse matrices in MATLAB

Supplementary reading. A detailed discussion of sparse matrix formats and how to work with them efficiently is given in [**?**]. An interesting related article on MATLAB-central can be found here.

(2.7.10) Creating sparse matrices in MATLAB

Matrices have to be created explicitly in sparse format by one of the following commands in order to let MATLAB know that the CCS format is to be used for internal representation.

`A = sparse(m,n);`	create empty $m \times n$ "sparse matrix"
`A = spalloc(m,n,nnz);`	create $m \times n$ sparse matrix & reserve memory
`A = sparse(I,J,a,m,n);`	initialize $m \times n$ sparse matrix from triplets → § 2.7.6
`A = spdiags(B,d,m,n);`	create sparse banded matrix → Section 2.7.6
`A = speye(n);`	sparse identity matrix

☞ Consult the MATLAB documentation for details.

Experiment 2.7.11 (Accessing rows and columns of sparse matrices)

The CCS internal data format used by MATLAB has an impact on the speed of access operations, *cf.* Exp. 1.2.29 for a similar effect.

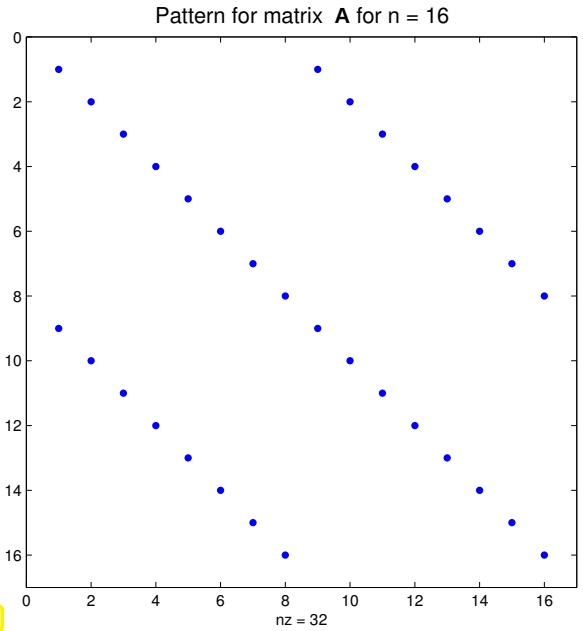


Fig. 69

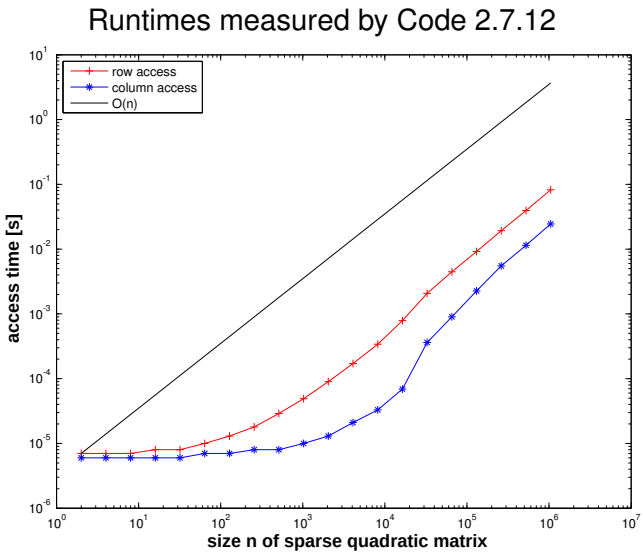


Fig. 70

MATLAB code 2.7.12: timing access to rows/columns of a sparse matrix

```
figure; spy(spdiags(repmat([-1 2 5],16,1),[-8,0,8],16,16)); %
title('Pattern for matrix {\bf A} for n = 16','fontsize',14);
print -depsc2 '../PICTURES/spdiagsmatspy.eps';

t = [];
for i=1:20
  n = 2^i; m = n/2;
  A = spdiags(repmat([-1 2 5],n,1),[-n/2,0,n/2],n,n); %

  t1 = inf; for j=1:5, tic; v = A(m,:)+j; t1 = min(t1,toc); end
  t2 = inf; for j=1:5, tic; v = A(:,m)+j; t2 = min(t2,toc); end
  t = [t; size(A,1), nnz(A), t1, t2 ];
end

figure;
loglog(t(:,1),t(:,3),'r+-', t(:,1),t(:,4),'b*-',...
       t(:,1),t(1,3)*t(:,1)/t(1,1),'k-');
xlabel('{\bf size n of sparse quadratic matrix}','fontsize',14);
ylabel('{\bf access time [s]}','fontsize',14);
legend('row access','column access','O(n)','location','northwest');

print -depsc2 '../PICTURES/sparseaccess.eps';
```

MATLAB uses compressed column storage (CCS), which entails $O(n)$ searches for index j in the index

array when accessing all elements of a matrix row. Conversely, access to a column does not involve any search operations.

Note the use of the MATLAB command `repmat` in Line 1 and Line 8 of the above code. It can be used to build structured matrices. Consult the MATLAB documentation for details.

Experiment 2.7.13 (Efficient Initialization of sparse matrices in MATLAB)

We study different ways to set a few non-zero entries of a sparse matrix. The first code just uses the `()`-operator to set matrix entries.

MATLAB code 2.7.14: Initialization of sparse matrices: entry-wise (I)

```
1 A1 = sparse(n,n);
2 for i=1:n
3     for j=1:n
4         if (abs(i-j) == 1), A1(i,j) = A1(i,j) + 1; end;
5         if (abs(i-j) == round(n/3)), A1(i,j) = A1(i,j) -1; end;
6  end; end
```

The second and third code rely on an intermediate triplet format (→ § 2.7.6) to build the sparse matrix and finally pass this to MATLAB's sparse function.

MATLAB code 2.7.15: Initialization of sparse matrices: triplet based (II)

```
1  dat = [];
2   for i=1:n
3      for j=1:n
4        if (abs(i-j) == 1), dat = [dat; i,j,1.0]; end;
5        if (abs(i-j) == round(n/3)), dat = [dat; i,j,-1.0];
6   end; end; end;
7   A2 = sparse(dat(:,1),dat(:,2),dat(:,3),n,n);
```

MATLAB code 2.7.16: Initialization of sparse matrices: triplet based (III)

```
1  dat = zeros(6*n,3); k = 0;
2    for i=1:n
3      for j=1:n
4        if (abs(i-j) == 1), k=k+1; dat(k,:) = [i,j,1.0];
5        end;
6        if (abs(i-j) == round(n/3))
7            k=k+1; dat(k,:) = [i,j,-1.0];
8        end;
9    end; end;
10   A3 = sparse(dat(1:k,1),dat(1:k,2),dat(1:k,3),n,n);
```

MATLAB code 2.7.17: Initialization of sparse matrices: driver script

```
1 % Driver routine for initialization of sparse matrices
2 K = 3; r = [];
```

```
for n=2.^(8:14)
    t1= 1000; for k=1:K,  fprintf('sparse1, %d, %d\n',n,k); tic;
       sparse1; t1 = min(t1,toc); end
     t2= 1000; for k=1:K,  fprintf('sparse2, %d, %d\n',n,k); tic;
        sparse2; t2 = min(t2,toc); end
    t3= 1000; for k=1:K,  fprintf('sparse3, %d, %d\n',n,k); tic;
       sparse3; t3 = min(t3,toc); end
    r = [r; n, t1 , t2, t3];
end

loglog(r(:,1),r(:,2),'r*',r(:,1),r(:,3),'m+',r(:,1),r(:,4),'b^');
xlabel('{\bf matrix size n}','fontsize',14);
ylabel('{\bf time [s]}','fontsize',14);
legend('Initialization I','Initialization II','Initialization III',...
    'location','northwest');
print -depsc2 '../PICTURES/sparseinit.eps';
```

Output of Code 2.7.17: ▷

- OS: Linux 2.6.16.27-0.9-smp
- CPU: Genuine Intel(R) CPU T2500 2.00GHz
- MATLAB 7.4.0.336 (R2007a)

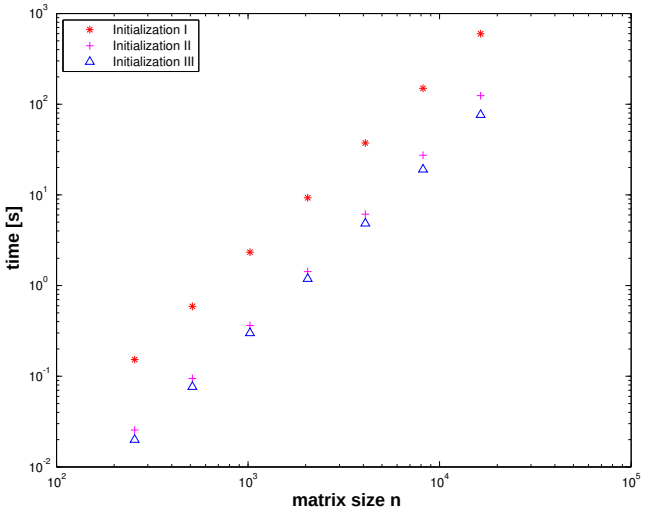


Fig. 71

☛ It is grossly inefficient to initialize a matrix in CCS format (→ Ex. 2.7.9) by setting individual entries one after another, because this usually entails moving large chunks of memory to create space for new non-zero entries.

Instead calls like

```
sparse(dat(1:k,1),dat(1:k,2),dat(1:k,3),n,n);,
```

where the triplet format is defined by (→ § 2.7.6)

$$\texttt{dat}(1:k,1) = i \quad \text{and} \quad \texttt{dat}(1:k,2) = j \quad \Rightarrow \quad a_{ij} = \texttt{dat}(1:k,3)\,,$$

allow MATLAB to allocate memory and initialize the arrays in one sweep.

Experiment 2.7.18 (Multiplication of sparse matrices in MATLAB)

We study a sparse matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ initialized by setting some of its (off-)diagonals with MATLAB's spdiags function:

```
A = spdiags([(1:n)',ones(n,1),(n:-1:1)'],...
   [-floor(n/3),0,floor(n/3)],n,n);
```

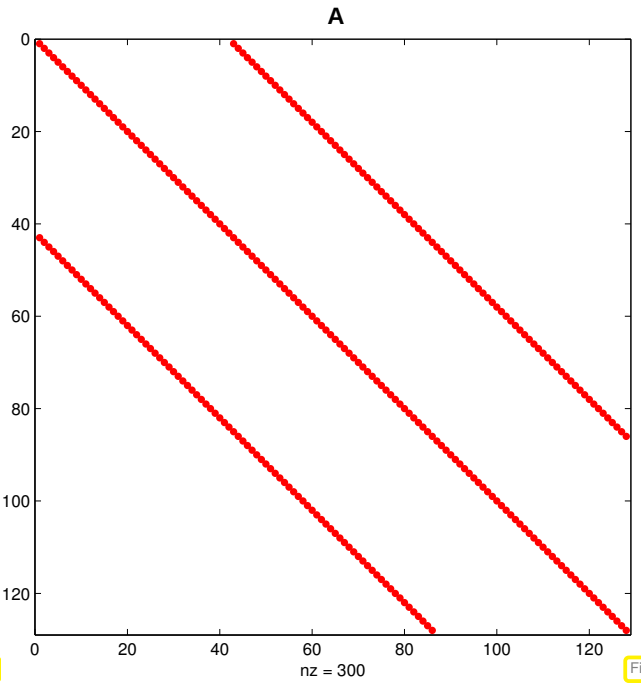


Fig. 72

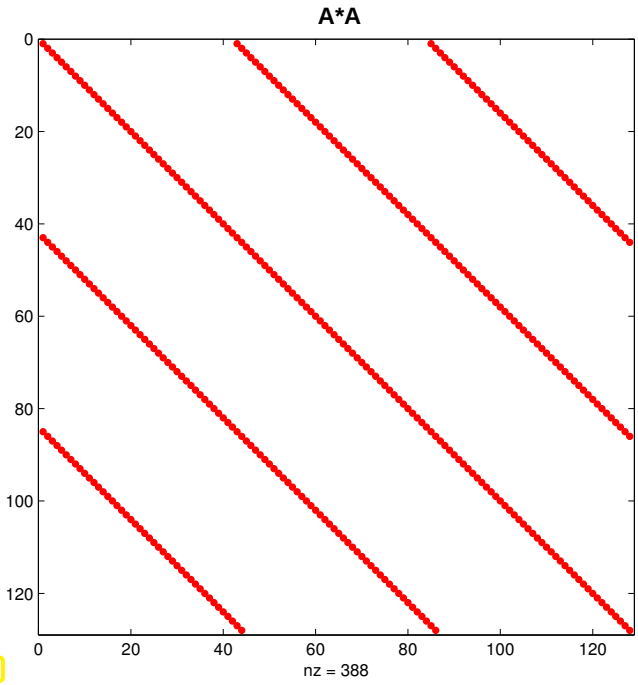


Fig. 73

➢ $\mathbf{A}^2$ is still a sparse matrix (→ Notion 2.7.1)

Runtimes for matrix multiplication `A*A` in MATLAB (`tic/toc` timing) ▷

(same platform as in Ex. 2.7.13)

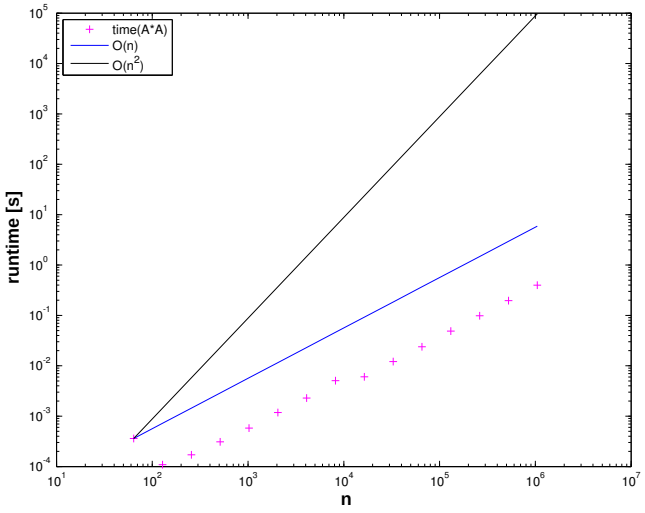


Fig. 74

We observe $O(n)$ asymptotic complexity: "optimal" implementation !

Remark 2.7.19 (Silly MATLAB)

A strange behavior of MATLAB from version R2010a:

MATLAB code 2.7.20: Extracting an entry of a sparse matrix

```
% MATLAB script demonstrating the awkward effects of treating entries
    of
% sparse matrices as sparse matrices themselves.
A = spdiags(ones(100,3),-1:1,100,100);
b = A(1,1),
c = full(b),
whos('b','c');
sum=0; tic; for i=1:1e6, sum = sum + b; end, toc
```

```
sum=0; tic; for i=1:1e6, sum = sum + c; end, toc
```

Output (MATLAB-version 7.12.0.635 (R2011a), MacOS X 10.6, Intel Core i7):

```
» sparseentry
b = (1,1) 1
c = 1

Name Size Bytes Class Attributes

b 1x1 32 double sparse
c 1x1 8 double

Elapsed time is 2.962332 seconds.
Elapsed time is 0.514712 seconds.
```

When extracting a single entry from a sparse matrix, this entry will be stored in sparse format though it is a mere number! This will considerably slow down all operations on that entry.

Change in Indexing for Sparse Matrix Access. Now subscripted reference into a sparse matrix always returns a sparse matrix. In previous versions of MATLAB, using a double scalar to index into a sparse matrix resulted in full scalar output.

2.7.3 Sparse matrices in EIGEN

Eigen can handle sparse matrices in the standard Compressed Row Storage (CRS) and Compressed Column Storage (CCS) format, see Ex. 2.7.9 and the documentation:

```
#include <Eigen/Sparse>
Eigen::SparseMatrix<int, Eigen::ColMajor> Asp(rows,cols); // CCS
   format
Eigen::SparseMatrix<double, Eigen::RowMajor> Bsp(rows,cols); // CRS
   format
```

As already discussed in Exp. 2.7.13, sparse matrices must not be filled by setting entries through index-pair access. As in MATLAB, also for EIGEN the matrix should first be assembled in triplet format, from which a sparse matrix is built. EIGEN offers special facilities for handling triplets.

```
std::vector <Eigen::Triplet <double >   > triplets;
// ..  fill the std::vector triplets ..
Eigen::SparseMatrix<double, Eigen::RowMajor> spMat(rows, cols);
spMat.setFromTriplets(triplets.begin(), triplets.end());
spMat.makeCompressed();
```

The call to `makeCompressed()` removes zero entries that are still kept in the raw sparse matrix format for the sake of efficient operations. For a concrete C++ code dedicated the initialization of a sparse EIGEN matrix from triplets, see Code 2.7.38.

A triplet object can be initialized as demonstrated in the following example:

```
unsigned int row_idx = 2;
unsigned int col_idx = 4;
double value = 2.5;
Eigen::Triplet<double> triplet(row_idx,col_idx,value);
std::cout << '(' << triplet.row() << ',' << triplet.col()
          << ',' << triplet.value() << ')' << std::endl;
```

As shown, a `Triplet` object offers the access member functions `row()`, `col()`, and `value()` to fetch the row index, column index, and scalar value stored in a `Triplet`.

The statement that entry-wise initialization of sparse matrices is not efficient has to be qualified in Eigen. Entries can be set, provided that *enough space* for each row (in `RowMajor` format) is reserved in advance. This done by the `reserve()` method that takes an integer vector of maximal expected numbers of non-zero entries per row:

C++11-code 2.7.21: Accessing entries of a sparse matrix: potentially inefficient!

```
1  unsigned int rows,cols,max_no_nnz_per_row;
2  .....
3  SparseMatrix<double, RowMajor> mat(rows,cols);
4  mat.reserve(RowVectorXi::Constant(cols,max_no_nnz_per_row));
5  // do many (incremental) initializations
6  for ( ) {
7    mat.insert(i,j) = value_ij;
8    mat.coeffRef(i,j) += increment_ij;
9  }
10 mat.makeCompressed();
```

`insert(i,j)` sets an entry of the sparse matrix, which is rather efficient, provided that enough space has be `reserve`d. `coeffRef(i,j)` gives l-value and r-value access to any matrix entry, creating a non-zero entry, if needed: costly!

The usual matrix operations are supported for sparse matrices; addition and subtraction may involve only sparse matrices stored in the *same format*. These operations may incur large hidden costs and have to be used with care!

Example 2.7.22 (Initialization of sparse matrices in Eigen)

We study the runtime behavior of the initialization of a sparse matrix in Eigen parallel to the tests from Exp. 2.7.13. We use the methods described above.

Runtimes (in ms) for the initialization of a banded matrix (with 5 non-zero diagonals, that is, a maximum of 5 non-zero entries per row) using different techniques in Eigen.

Green line: timing for entry-wise initialization with only 4 non-zero entries per row reserved in advance.

(OS: Ubuntu Linux 14.04, CPU: Intel i5@1.80 Ghz, Compiler: g++-4.8.2, -O2)

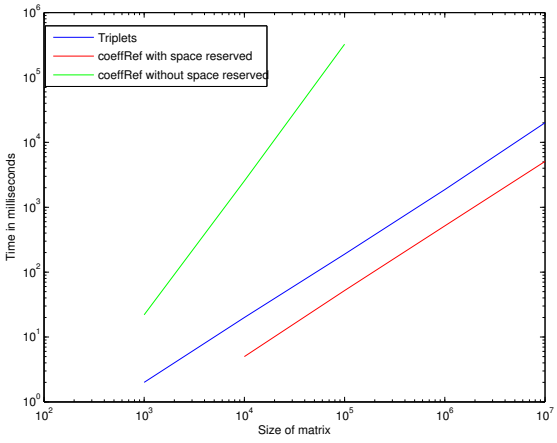


Fig. 75

Observation: insufficient advance allocation of memory massively slows down the set-up of a sparse matrix in the case of direct entry-wise initialization.

Reason: Massive internal copying of data required to created space for "unexpected" entries.

Example 2.7.23 (Smoothing of a triangulation)

This example demonstrates that sparse linear systems of equations naturally arise in the handling of triangulations.

Definition 2.7.24. Planar triangulation

A planar triangulation (mesh) $\mathcal{M}$ consists of a set $\mathcal{N}$ of $N \in \mathbb{N}$ distinct points $\in \mathbb{R}^2$ and a set $\mathcal{T}$ of triangles with vertices in $\mathcal{N}$, such that the following two conditions are satisfied:

1. the interiors of the triangles are mutually disjoint ("no overlap"),
2. for every two *closed* distinct triangles $\in \mathcal{T}$ their intersection satisfies exactly one of the following conditions:
   (a) it is empty
   (b) it is exactly one vertex from $\mathcal{N}$,
   (c) it is a common edge of both triangles

The points in $\mathcal{N}$ are also called the nodes of the mesh, the triangles the cells, and all line segments connecting two nodes and occurring as a side of a triangle form the set of edges. We always assume a

consecutive numbering of the nodes and cells of the triangulation (starting from 1, MATLAB's convention).

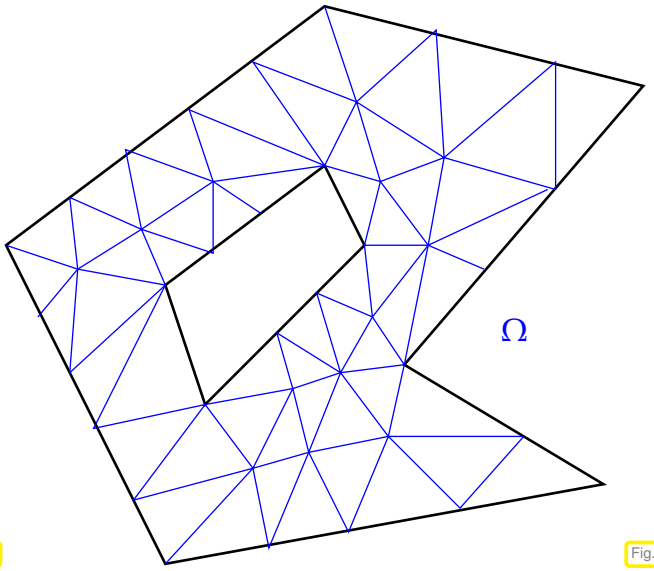


Fig. 76

Valid planar triangulation

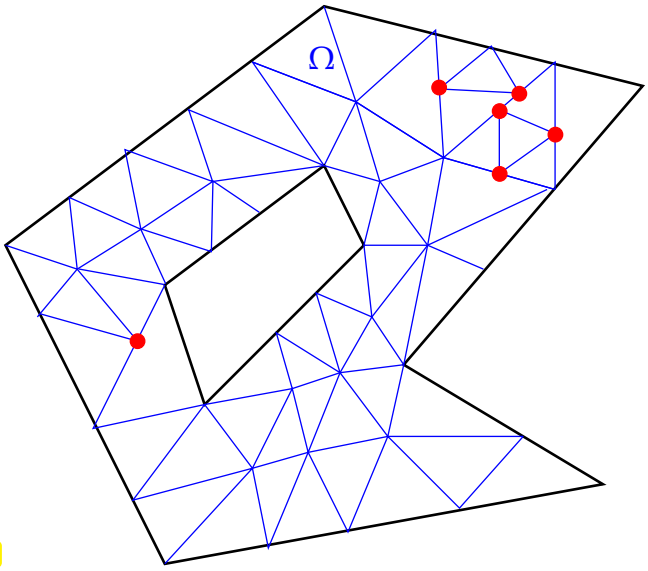
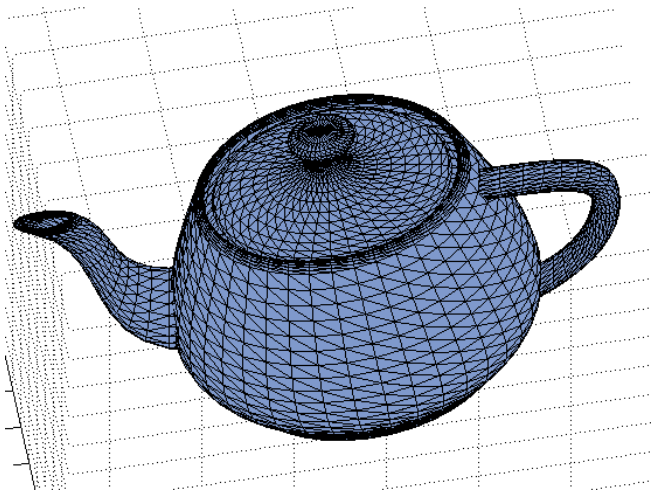
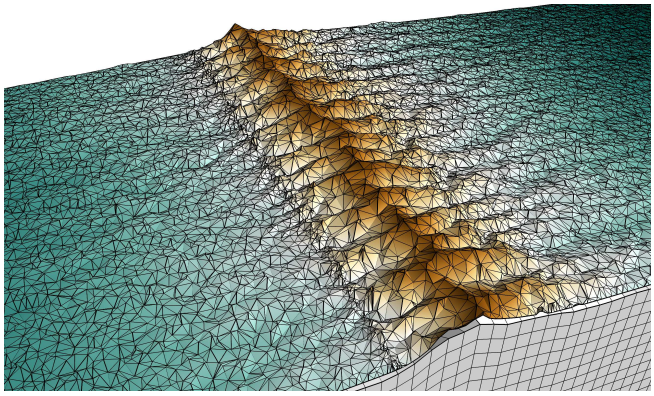


Fig. 77

Mesh with "illegal" hanging nodes

Triangulations are of fundamental importance for computer graphics, landscape models, geodesy, and numerical methods. They need not be planar, but the algorithmic issues remain the same.





MATLAB data structure for describing a triangulation with N nodes and M cells:

- column vector $\mathbf{x} \in \mathbb{R}^N$: x-coordinates of nodes
- column vector $\mathbf{y} \in \mathbb{R}^N$: y-coordinates of nodes
- $M \times 3$-matrix $\mathbf{T}$ whose *rows* contain the index numbers of the vertices of the cells.
  (This matrix is a so-called triangle-node incidence matrix.)

The **Figure** library provides the function `triplot` for drawing planar triangulations:

C++11-code 2.7.25: Initializing and drawing a simple planar triangulations ⇢ GITLAB

```
2  // Demonstration for visualizing a plane triangular mesh
3  // Initialize node coordinates
4  Eigen::VectorXd x(10), y(10);
5  // x and y coordinate of mesh
6  x << 1.0,0.60,0.12,0.81,0.63,0.09,0.27,0.54,0.95,0.96;
7  y << 0.15,0.97,0.95,0.48,0.80,0.14,0.42,0.91,0.79,0.95;
8  // Then specify triangles through the indices of their vertices.
```

```
    These
// indices refer to the ordering of the coordinates as given in the
// vectors x and y.
Eigen::MatrixXi T(11,3);
T << 7, 1, 2,   5, 6, 2,   4, 1, 7,   6, 7, 2,
  6, 4, 7,   6, 5, 0,   3, 6, 0,   8, 4, 3,
  3, 4, 6,   8, 1, 4,   9, 1, 8;
// Call the Figure plotting routine, draw mesh with blue edges
// red vertices and a numbering/ordering
mgl::Figure fig1; fig1.setFontSize(8);
fig1.ranges(0.0, 1.05, 0.0, 1.05);
fig1.triplot(T, x, y, "b?"); // drawing triangulation with numbers
fig1.plot(x, y, " *r"); // mark vertices
fig1.save("meshplot_cpp");
```

Output of Code 2.7.25

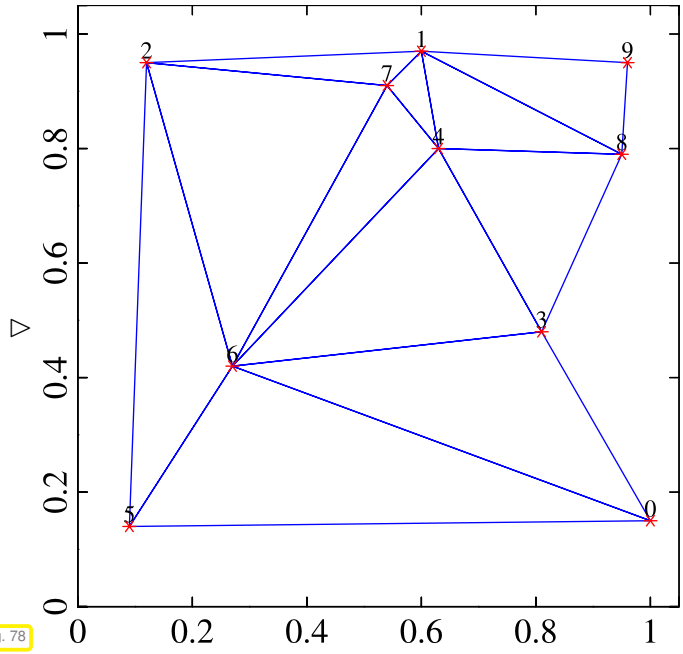


Fig. 78

The cells of a mesh may be rather distorted triangles (with very large and/or small angles), which is usually not desirable. We study an algorithm for smoothing a mesh without changing the planar domain covered by it.

Definition 2.7.26. Boundary edge

Every edge that is adjacent to only one cell is a boundary edge of the triangulation. Nodes that are endpoints of boundary edges are boundary nodes.

✎ Notation: $\Gamma \subset \{1,\ldots,N\} \hat{=}$ set of indices of boundary nodes.

✎ Notation: $\mathbf{p}^i = (p_1^i, p_2^i) \in \mathbb{R}^2 \hat{=}$ coordinate vector of node $\sharp i$, $i = 1,\ldots,N$
($p_1^i \leftrightarrow$ x(i), $p_2^i \leftrightarrow$ y(i) in MATLAB)

We define

$$S(i) := \{j \in \{1,\ldots,N\} : \text{ nodes } i \text{ and } j \text{ are connected by an edge}\}\,, \tag{2.7.27}$$

as the set of node indices of the "neighbours" of the node with index number i.

Definition 2.7.28. Smoothed triangulation

A triangulation is called smoothed, if

$$\mathbf{p}^i = \frac{1}{\sharp S(i)} \sum_{j \in S(i)} \mathbf{p}^j \quad \Leftrightarrow \quad \sharp S(i) p_d^i = \sum_{j \in S(i)} p_d^j,\ d = 1,2\,, \quad \text{for all} \quad i \in \{1,\ldots,N\} \setminus \Gamma\,, \tag{2.7.29}$$

that is, every interior node is located in the center of gravity of its neighbours.

The relations (2.7.29) correspond to the lines of a sparse linear system of equations! In order to state it, we insert the coordinates of all nodes into a column vector $\mathbf{z} \in \mathbb{K}^{2N}$, according to

$$z_i = \begin{cases} p_1^i & \text{, if } 1 \le i \le N\,, \\ p_2^{i-N} & \text{, if } N+1 \le i \le 2N\,. \end{cases} \tag{2.7.30}$$

For the sake of ease of presentation, in the sequel we *assume* (which is not the case in usual triangulation data) that interior nodes have index numbers smaller than that of boundary nodes.

From (2.7.27) we infer that the system matrix $\mathbf{C} \in \mathbb{R}^{2n,2N}$, $n := N - \sharp\Gamma$, of that linear system has the following structure:

$$\mathbf{C} = \begin{bmatrix} \mathbf{A} & \mathbf{O} \\ \mathbf{O} & \mathbf{A} \end{bmatrix}\,, \quad (\mathbf{A})_{i,j} = \begin{cases} \sharp S(i) & \text{, if } i = j\,, \\ -1 & \text{, if } j \in S(i)\,, \\ 0 & \text{else.} \end{cases} \quad \begin{matrix} i \in \{1,\ldots,n\}\,, \\ j \in \{1,\ldots,N\}\,. \end{matrix} \tag{2.7.31}$$

$$\blacktriangleright \quad (2.7.29) \quad \Leftrightarrow \quad \mathbf{C}\mathbf{z} = \mathbf{0}\,. \tag{2.7.32}$$

- ➢ $\mathrm{nnz}(\mathbf{A}) \le$ number of edges of $\mathcal{M}$ + number of interior nodes of $\mathcal{M}$.
- ➢ The matrix $\mathbf{C}$ associated with $\mathcal{M}$ according to (2.7.31) is clearly sparse.
- ➢ The sum of the entries in every row of $\mathbf{C}$ vanishes.

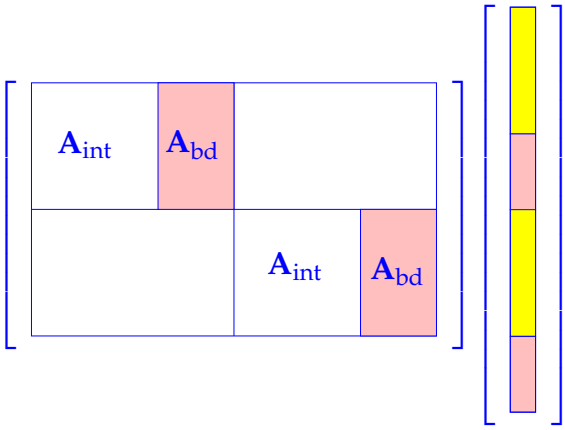
We partition the vector $\mathbf{z}$ into coordinates of nodes in the interior and of nodes on the boundary

$$\mathbf{z}^T = \begin{bmatrix} \mathbf{z}_1^{\mathrm{int}} \\ \mathbf{z}_1^{\mathrm{bd}} \\ \mathbf{z}_2^{\mathrm{int}} \\ \mathbf{z}_2^{\mathrm{bd}} \end{bmatrix} := \left[z_1,\ldots,z_n,z_{n+1},\ldots,z_N,z_{N+1},\ldots,z_{N+n},z_{N+n+1},\ldots,z_{2N}\right]^\top\,.$$

This induces the following block partitioning of the linear system (2.7.32):

$$\begin{bmatrix} \mathbf{A}_{\mathrm{int}} & \mathbf{A}_{\mathrm{bd}} & \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} & \mathbf{A}_{\mathrm{int}} & \mathbf{A}_{\mathrm{bd}} \end{bmatrix} \begin{bmatrix} \mathbf{z}_1^{\mathrm{int}} \\ \mathbf{z}_1^{\mathrm{bd}} \\ \mathbf{z}_2^{\mathrm{int}} \\ \mathbf{z}_2^{\mathrm{bd}} \end{bmatrix} = \mathbf{0}\,, \quad \begin{matrix} \mathbf{A}_{\mathrm{int}} \in \mathbb{R}^{n,n}\,, \\ \mathbf{A}_{\mathrm{bd}} \in \mathbb{R}^{n,N-n}\,. \end{matrix}$$

$$\Updownarrow$$



(2.7.33)

The linear system (2.7.33) holds the key to the algorithmic realization of mesh smoothing; when smoothing the mesh

(i) the node coordinates belonging to interior nodes have to be adjusted to satisfy the equilibrium condition (2.7.29), they are unknowns,

(ii) the coordinates of nodes located on the boundary are fixed, that is, their values are known.

$$\begin{array}{cc} \text{unknown} \quad \mathbf{z}_1^{\text{int}}, \mathbf{z}_2^{\text{int}}, & \text{known} \quad \mathbf{z}_1^{\text{bd}}, \mathbf{z}_2^{\text{bd}} \\ \text{(yellow in (2.7.33))} & \text{(pink in (2.7.33))} \end{array}$$

$$\text{(2.7.32)/(2.7.33)} \quad \Leftrightarrow \quad \mathbf{A}_{\text{int}}\left[\mathbf{z}_1^{\text{int}}\ \mathbf{z}_2^{\text{int}}\right] = -\left[\mathbf{A}_{\text{bd}}\mathbf{z}_1^{\text{bd}}\ \mathbf{A}_{\text{bd}}\mathbf{z}_2^{\text{bd}}\right] . \tag{2.7.34}$$

This is a square linear system with an $n \times n$ system matrix, to be solved for two different right hand side vectors. The matrix $\mathbf{A}_{\text{int}}$ is also known as the matrix of the combinatorial graph Laplacian.

We examine the sparsity pattern of the system matrices $\mathbf{A}_{\text{int}}$ for a sequence of triangulations created by regular refinement.

Definition 2.7.35. Regular refinemnent of a planar triangulation

The planar triangulation with cells obtained by splitting all cells of a planar triangulation $\mathcal{M}$ into four congruent triangles is called the regular refinement of $\mathcal{M}$.

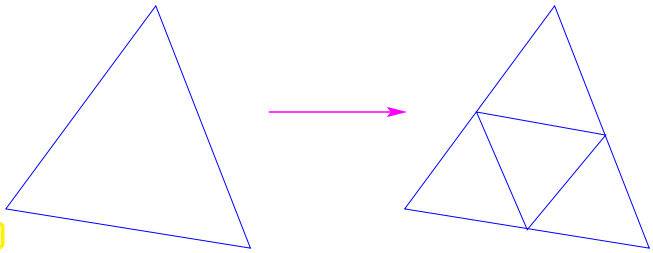
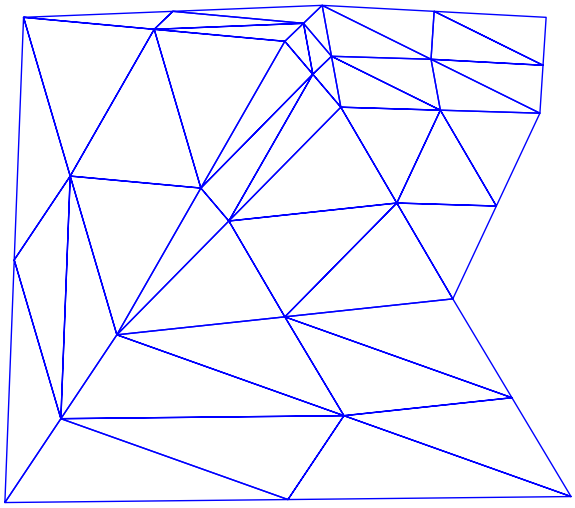


Fig. 79

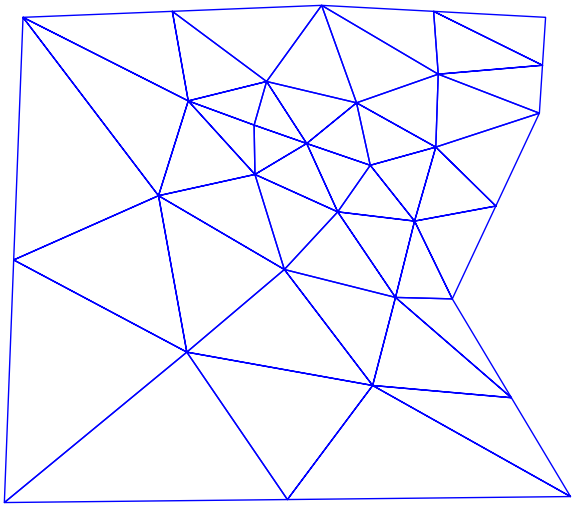
We start from the triangulation of Fig. 78 and in turns perform regular refinement and smoothing (left $\leftrightarrow$

after refinement, right $\leftrightarrow$ after smoothing)

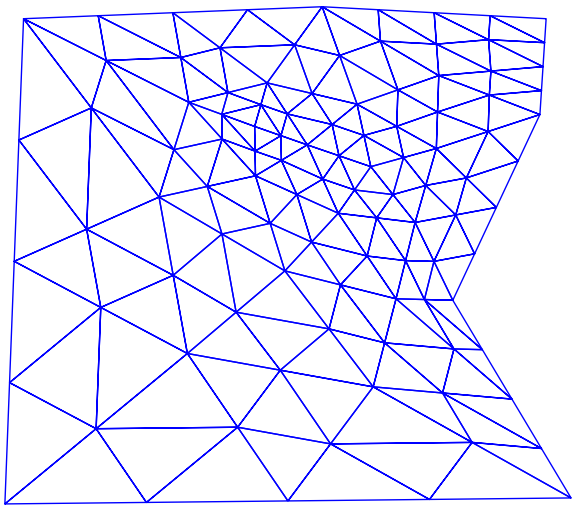
Refined mesh level 1



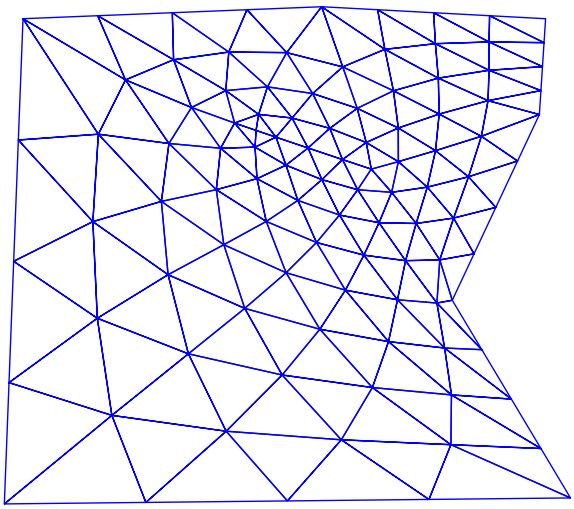
Smoothed mesh level 1



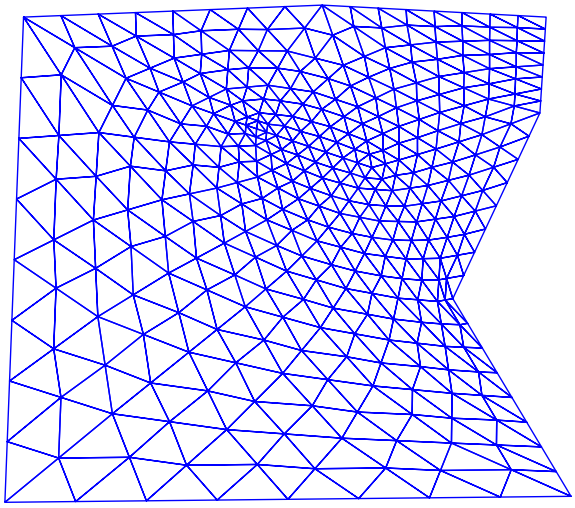
Refined mesh level 2



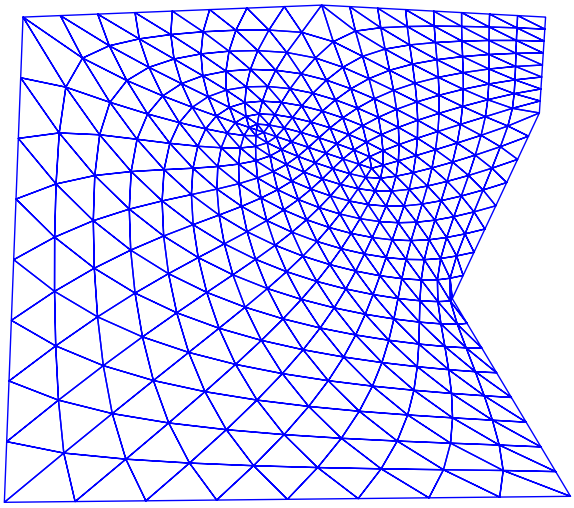
Smoothed mesh level 2



Refined mesh level 3



Smoothed mesh level 3



Below we give spy plots of the system matrices $\mathbf{A}_{\mathrm{int}}$ for the first three triangulations of the sequence:

level 1: 30 points, 73 edges, 44 cells

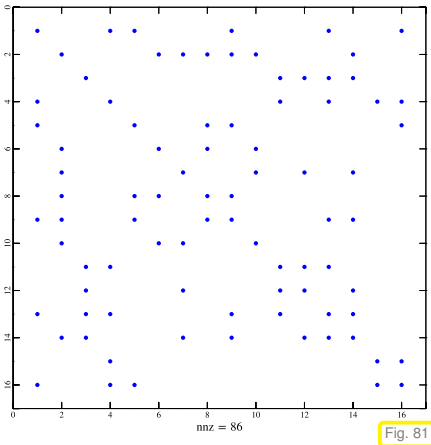


Fig. 80

level 2: 103 points, 278 edges, 176 cells

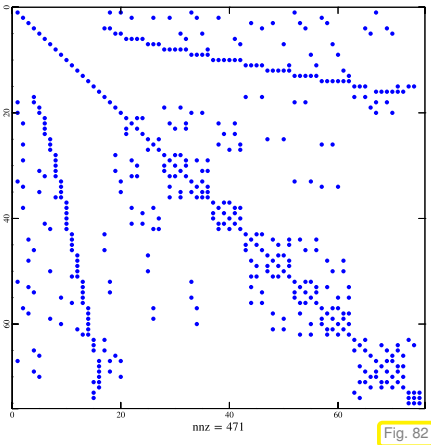


Fig. 81

level 3: 381 points, 1084 edges, 704 cells

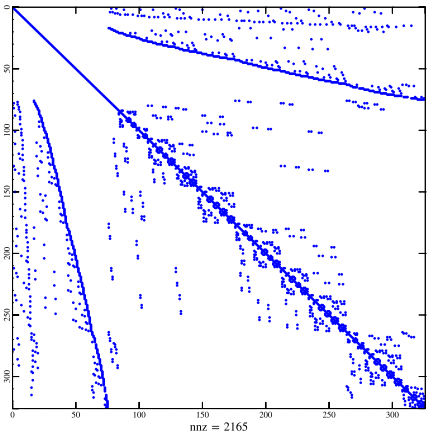


Fig. 82

2.7.4 Direct Solution of Sparse Linear Systems of Equations

Efficient Gaussian elimination for sparse matrices requires sophisticated algorithms that are encapsulated in special types of solvers in EIGEN. Their calling syntax remains unchanged, however:

```
Eigen::SolverType<Eigen::SparseMatrix<double>> solver(A);
Eigen::VectorXd x = solver.solve(b);
```

The standard sparse solver is `SparseLU`.

C++-code 2.7.36: Function for solving a sparse LSE with EIGEN ↪ GITLAB

```
using SparseMatrix = Eigen::SparseMatrix<double>;
// Perform sparse elimination
void sparse_solve(const SparseMatrix& A, const VectorXd& b, VectorXd&
    x) {
  Eigen::SparseLU<SparseMatrix> solver(A);
  x = solver.solve(b);
}
```

(2.7.37) Solving sparse linear systems in EIGEN

The following codes initialize a sparse matrix, then perform an LU-factorization, and, finally, solve a sparse linear system with a random right hand side vector.

C++11-code 2.7.38: Initialisation of sample sparse matrix in EIGEN ↪ GITLAB

```
template <class SpMat>
SpMat initSparseMatrix(size_t n)
{
  using scalar_t = typename SpMat::Scalar;
  vector<Eigen::Triplet<scalar_t>> triplets(5*n);

  for (size_t l=0; l<n; ++l)
    triplets.push_back(Eigen::Triplet<scalar_t>(l,l,5.0));
```

```
10    for (size_t l=1; l<n; ++l) {
11      triplets.push_back(Eigen::Triplet<scalar_t>(l-1,l,1.0));
12      triplets.push_back(Eigen::Triplet<scalar_t>(l,l-1,1.0));
13    }
14    const size_t m = n/2;
15    for (size_t l=0; l<m; ++l) {
16      triplets.push_back(Eigen::Triplet<scalar_t>(l,l+m,1.0));
17      triplets.push_back(Eigen::Triplet<scalar_t>(l+m,l,1.0));
18    }
19    SpMat M(n,n);
20    M.setFromTriplets(triplets.begin(), triplets.end());
21    M.makeCompressed();
22    return M;
23  }
```

C++11-code 2.7.39: Solving a sparse linear system of equations in EIGEN ⇝ GITLAB

```
2   void solveSparseTest(size_t n)
3   {
4     using SpMat = Eigen::SparseMatrix<double, Eigen::RowMajor>;
5
6     const SpMat M = initSparseMatrix<SpMat>(n);
7     cout << "M = " << M.rows() << 'x' << M.cols() << "-matrix with " <<
        M.nonZeros()
8       << "non-zeros" << endl;
9     printTriplets(M);
10
11    const Eigen::VectorXd b = Eigen::VectorXd::Random(n);
12    Eigen::VectorXd x(n);
13
14    Eigen::SparseLU<SpMat> solver; solver.compute(M);
15    if(solver.info() != Eigen::Success) {
16      cerr << "Decomposition failed!" << endl;
17      return;
18    }
19    x = solver.solve(b);
20    if(solver.info() != Eigen::Success) {
21      cerr << "Solver failed!" << endl;
22      return;
23    }
24    cout << "Residual norm = " << (M*x-b).norm() << endl;
25  }
```

The `compute` method of the solver object triggers the actual sparse LU-decomposition. The `solve` method then does forward and backward elimination, *cf.* § 2.3.30. It can be called multiple times, see Rem. 2.5.10.

Experiment 2.7.40 (Sparse elimination for arrow matrix)

In Ex. 2.6.5 we saw that applying the standard lu() solver to a sparse arrow matrix results in an extreme waste of computational resources.

Yet, EIGEN can do much better**!** The main mistake was the creation of a dense matrix instead of storing the arrow matrix in sparse format. There are EIGEN solvers which rely on particular sparse elimination techniques. They still rely of Gaussian elimination with (partial) pivoting (→ Code 2.3.41), but take pains to operate on non-zero entries only. This can greatly boost the speed of the elimination.

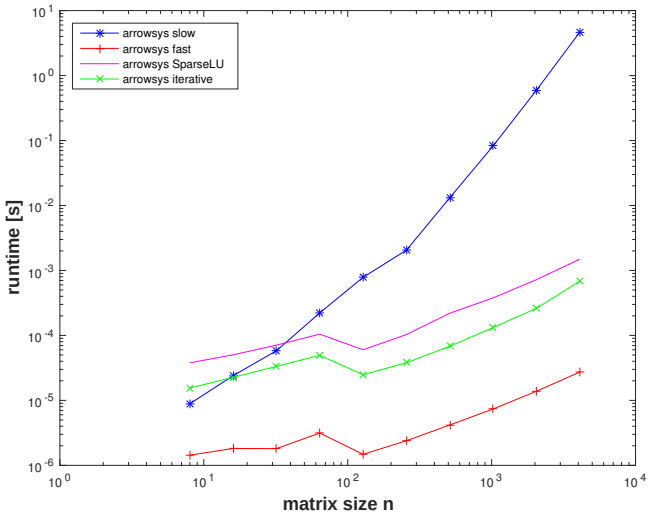
C++11 code 2.7.41: Invoking sparse elimination solver for arrow matrix ➙ GITLAB

```
2  template <class solver_t>
3  VectorXd arrowsys_sparse(const VectorXd &d, const VectorXd &c, const
       VectorXd &b, const double alpha, const VectorXd &y){
4    int n = d.size();
5    SparseMatrix<double> A(n+1, n+1); // default: column-major
6    VectorXi reserveVec = VectorXi::Constant(n+1, 2); // nnz per col
7    reserveVec(n) = n+1;          // last full col
8    A.reserve(reserveVec);
9    for(int j = 0; j < n; ++j){   // initalize along cols for
         efficency
10     A.insert(j,j) = d(j);       // diagonal entries
11     A.insert(n,j) = b(j);       // bottom row entries
12   }
13   for(int i = 0; i < n; ++i){
14     A.insert(i,n) = c(i);       // last col
15   }
16   A.insert(n,n) = alpha;        // bottomRight entry
17   A.makeCompressed();
18   return solver_t(A).solve(y);
19 }
```

Observation:

The sparse elimination solver is several orders of magnitude faster than lu() operating on a dense matrix.

The sparse solver is still slower than Code 2.6.10. The reason is that it is a general algorithm that has to keep track of non-zero entries and has to be prepared to do pivoting.



Experiment 2.7.42 (Timing sparse elimination for the combinatorial graph Laplacian)

We consider a sequence of planar triangulations created by successive regular refinement (→ Def. 2.7.35) of the planar triangulation of Fig. 78, see Ex. 2.7.23. We use different EIGEN and MKL sparse solver for the linear system of equations (2.7.34) associated with each mesh.

Timing results ▷

Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3

We observe an *empirical* asymptotic complexity (→ Def. 1.4.4) of $O(n^{1.5})$, way better than the asymptotic complexity of $O(n^3)$ expected for Gaussian elimination in the case of dense matrices.

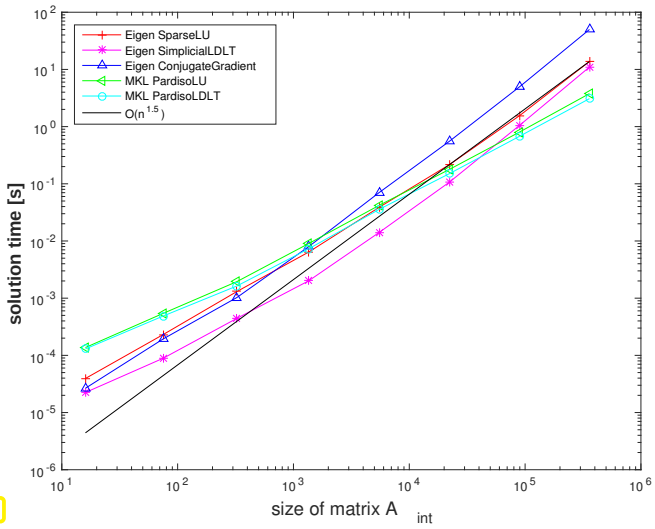


Fig. 84

When solving linear systems of equations directly **dedicated sparse elimination solvers** from *numerical libraries* have to be used!

System matrices are passed to these algorithms in sparse storage formats (→ 2.7.1) to convey information about zero entries.

STOP Never ever even think about implementing a general sparse elimination solver by yourself!

EIGEN sparse solvers:
https://eigen.tuxfamily.org/dox/group__TopicSparseSystems.html

(2.7.43) Implementations of sparse solvers

Widely used implementations of sparse solvers are:

→ SuperLU (http://www.cs.berkeley.edu/~demmel/SuperLU.html),
→ UMFPACK (http://www.cise.ufl.edu/research/sparse/umfpack/), used by MATLAB's \,

→ PARDISO [?] (http://www.pardiso-project.org/), incorporated into MKL

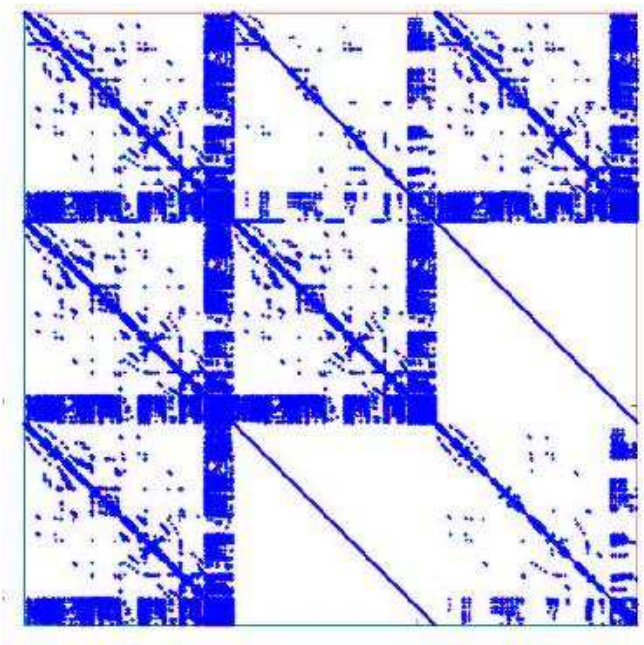


Fig. 85

◁ fill-in (→ Def. 2.7.47) during sparse elimination with PARDISO

PARDISO has been developed by Prof. O. Schenk and his group (formerly University of Basel, now USI Lugano).

C++11-code 2.7.44: Example code demonstrating the use of PARDISO with EIGEN ↬ GITLAB

```
2  void solveSparsePardiso(size_t n){
3    using SpMat = Eigen::SparseMatrix<double>;
4    // Initialize a sparse matrix
5    const SpMat M = initSparseMatrix<SpMat>(n);
6    const Eigen::VectorXd b = Eigen::VectorXd::Random(n);
7    Eigen::VectorXd x(n);
8    // Initalization of the sparse direct solver based on the Pardiso
       library with directly passing the matrix M to the solver
9    // Pardiso is part of the Intel MKL library, see also Ex. 1.3.24
10   Eigen::PardisoLU<SpMat> solver(M);
11   // The checks of Code 2.7.39 are omitted
12   // solve the LSE
13   x = solver.solve(b);
14 }
```

2.7.5 LU-factorization of sparse matrices

In Sect. 2.7.1 we have seen, how sparse matrices can be stored requiring $O(\mathrm{nnz}(\mathbf{A}))$ memory.

In Ex. 2.7.18 we found that (sometimes) matrix multiplication of sparse matrices can also be carried out with optimal complexity, that is, with computational effort proportional to the total number of non-zero entries of all matrices involved.

Does this carry over to the solution of linear systems of equations with sparse system matrices**?** Sec-

tion 2.7.4 says "Yes", when sophisticated library routines are used. In this section, we examine some aspects of Gaussian elimination $\leftrightarrow$ LU-factorisation when applied in a sparse context.

Example 2.7.45 (LU-factorization of sparse matrices)

We examine the following "sparse" matrix with a typical structure and inspect the pattern of the LU-factors returned by EIGEN, see Code 2.7.46.

$$\mathbf{A} = \left(\begin{array}{cccc|ccc} 3 & -1 & & & -1 & & \\ -1 & \ddots & \ddots & & & \ddots & \\ & \ddots & \ddots & -1 & & & \ddots \\ \hline & & -1 & 3 & & & -1 \\ -1 & & & & 3 & -1 & \\ & \ddots & & & -1 & \ddots & \ddots \\ & & \ddots & & & \ddots & \ddots \\ & & & -1 & & & -1 \quad 3 \end{array}\right) \in \mathbb{R}^{n,n}, n \in \mathbb{N}$$

C++11 code 2.7.46: Visualizing LU-factors of a sparse matrix ↬ GITLAB

```
// Build matrix
int n = 100;
RowVectorXd diag_el(5);  diag_el << -1,-1, 3, -1,-1;
VectorXi diag_no(5);     diag_no << -n, -1, 0, 1, n;
MatrixXd B = diag_el.replicate(2*n,1);
B(n-1,1) = 0; B(n,3) = 0; // delete elements
// A custom function from the Utils folder
SparseMatrix<double> A = spdiags(B, diag_no, 2*n, 2*n);
// It is not possible to access the LU-factors in the case of
// EIGEN's LU-decomposition for sparse matrices.
// Therefore we have to resort to the dense version.
auto solver = MatrixXd(A).lu();
MatrixXd L = MatrixXd::Identity(2*n,2*n);
L += solver.matrixLU().triangularView<StrictlyLower>();
MatrixXd U = solver.matrixLU().triangularView<Upper>();
// Plotting
mgl::Figure fig1, fig2, fig3;
fig1.spy(A);   fig1.setFontSize(4);
fig1.title("Sparse matrix");  fig1.save("sparseA_cpp");
fig2.spy(L);   fig2.setFontSize(4);
fig2.title("Sparse matrix: L factor");  fig2.save("sparseL_cpp");
fig3.spy(U);   fig3.setFontSize(4);
fig3.title("Sparse matrix: U factor");  fig3.save("sparseU_cpp");
```

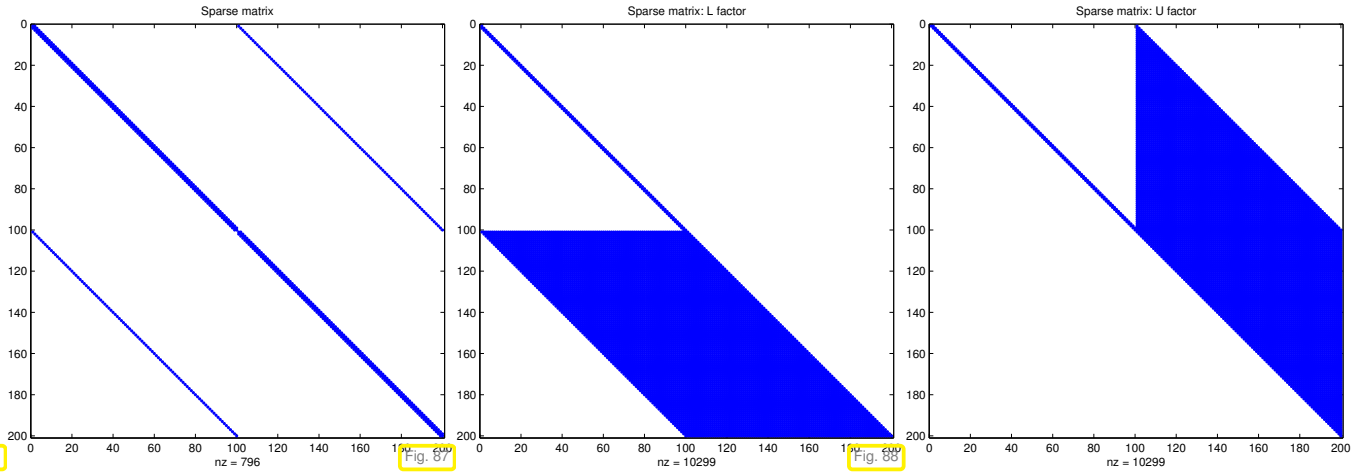


Fig. 86 Fig. 87 Fig. 88

Observation:	$\mathbf{A}$ sparse $\not\Rightarrow$ LU-factors sparse

Of course, in case the LU-factors of a sparse matrix possess many more non-zero entries than the matrix itself, the effort for solving a linear system with direct elimination will increase significantly. This can be quantified by means of the following concept:

Definition 2.7.47. Fill-in

Let $\mathbf{A} = \mathbf{LU}$ be an LU-factorization (→ Sect. 2.3.2) of $\mathbf{A} \in \mathbb{K}^{n,n}$. If $l_{ij} \neq 0$ or $u_{ij} \neq 0$ though $a_{ij} = 0$, then we encounter fill-in at position (i,j).

Example 2.7.48 (Sparse LU-factors)

Ex. 2.7.45 ➢ massive fill-in can occur for sparse matrices

This example demonstrates that fill-in can largely be avoided, if the matrix has favorable structure. In this case a LSE with this particular system matrix $\mathbf{A}$ can be solved efficiently, that is, with a computational effort $O(\mathrm{nnz}(\mathbf{A}))$ by Gaussian elimination.

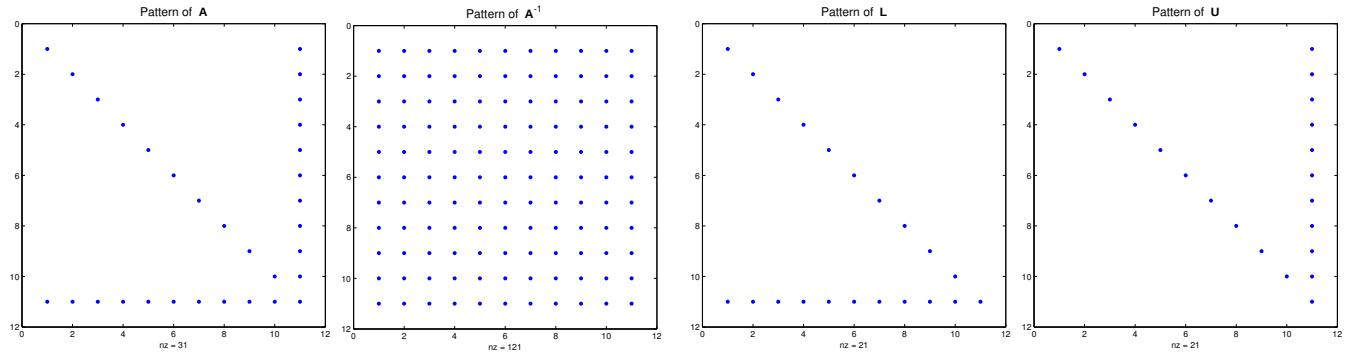
C++11 code 2.7.49: LU-factorization of sparse matrix ↪ GITLAB

```
// Build matrix
MatrixXd A(11,11); A.setIdentity();
A.col(10).setOnes(); A.row(10).setOnes();
// A.reverseInPlace(); // used inEx. 2.7.50
auto solver = A.lu();
MatrixXd L = MatrixXd::Identity(11,11);
L += solver.matrixLU().triangularView<StrictlyLower>();
MatrixXd U = solver.matrixLU().triangularView<Upper>();
MatrixXd Ainv = A.inverse();
// Plotting
mgl::Figure fig1, fig2, fig3, fig4;
fig1.spy(A);   fig1.setFontSize(4);
fig1.title("Pattern of A");   fig1.save("Apat_cpp");
fig2.spy(L);   fig2.setFontSize(4);
```

```
16 fig2.title("Pattern of L");   fig2.save("Lpat_cpp");
17 fig3.spy(U);   fig3.setFontSize(4);
18 fig3.title("Pattern of U");   fig3.save("Upat_cpp");
19 fig4.spy(Ainv);   fig4.setFontSize(4);
20 fig4.title("Pattern of A^{-1}");   fig4.save("Ainvpat_cpp");
```

$\mathbf{A}$ is called an "arrow matrix", see the pattern of non-zero entries below and Ex. 2.6.5.

Recalling Rem. 2.3.32 it is easy to see that the LU-factors of $\mathbf{A}$ will be sparse and that their sparsity patterns will be as depicted below. Observe that despite sparse LU-factors, $\mathbf{A}^{-1}$ will be densely populated.



$\mathbf{L}$, $\mathbf{U}$ sparse $\not\Longrightarrow$ $\mathbf{A}^{-1}$ sparse **!**

Besides stability and efficiency issues, see Exp. 2.4.10, this is another reason why using `x = A.inverse()*y` instead of `y = A.lu().solve(b)` is usually a major blunder.

Example 2.7.50 (LU-decomposition of flipped "arrow matrix")

Recall the discussion in Ex. 2.6.5. Here we look at an arrow matrix in a slightly different form:

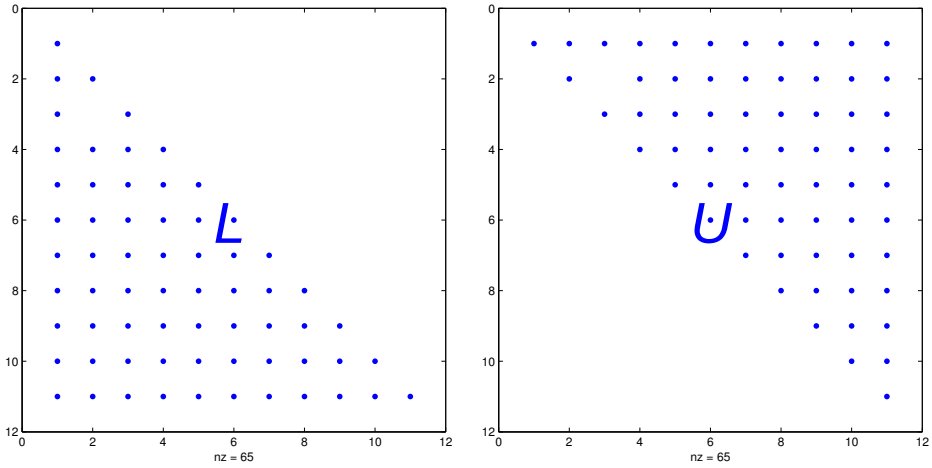
$$\mathbf{M} = \left[\begin{array}{c|c} \alpha & \mathbf{b}^\top \\ \hline \mathbf{c} & \mathbf{D} \end{array}\right] , \quad \begin{array}{l} \alpha \in \mathbb{R} , \\ \mathbf{b}, \mathbf{c} \in \mathbb{R}^{n-1} , \\ \mathbf{D} \in \mathbb{R}^{n-1,n-1} \text{ regular diagonal matrix, } \rightarrow \text{ Def. 1.1.5} \end{array} \tag{2.7.51}$$

Run the algorithm from § 2.3.21 (LU decompisition without pivoting):

- LU-decomposition dense factor matrices with $O(n^2)$ non-zero entries.
- asymptotic computational cost: $O(n^3)$

Output of modified Code 2.7.49:

Obvious fill-in (→ Def. 2.7.47)



Now it comes as a surprise that the arrow matrix $\mathbf{A}$ from Ex. 2.6.5, (2.6.6) has sparse LU-factors!

Arrow matrix (2.6.6) ▷

$$\mathbf{A} = \begin{pmatrix} \mathbf{D} & \mathbf{c} \\ \mathbf{b}^\top & \alpha \end{pmatrix}$$

$$\mathbf{A} = \underbrace{\begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{b}^\top\mathbf{D}^{-1} & 1 \end{pmatrix}}_{=:\mathbf{L}} \cdot \underbrace{\begin{pmatrix} \mathbf{D} & \mathbf{c} \\ 0 & \sigma \end{pmatrix}}_{=:\mathbf{U}}, \quad \sigma := \alpha - \mathbf{b}^\top\mathbf{D}^{-1}\mathbf{c}\,.$$

➢ In this case LU-factorisation is possible without fill-in, cost merely $O(n)$!

Idea: Transform $\mathbf{A}$ into $\mathbf{A}$ by row and column permutations before performing LU-decomposition.

Details: Apply a cyclic permutation of rows/columns:

- 1st row/column $\to$ n-th row/column
- i-th row/column $\to$ $i-1$-th row/column, $i = 2, \ldots, n$

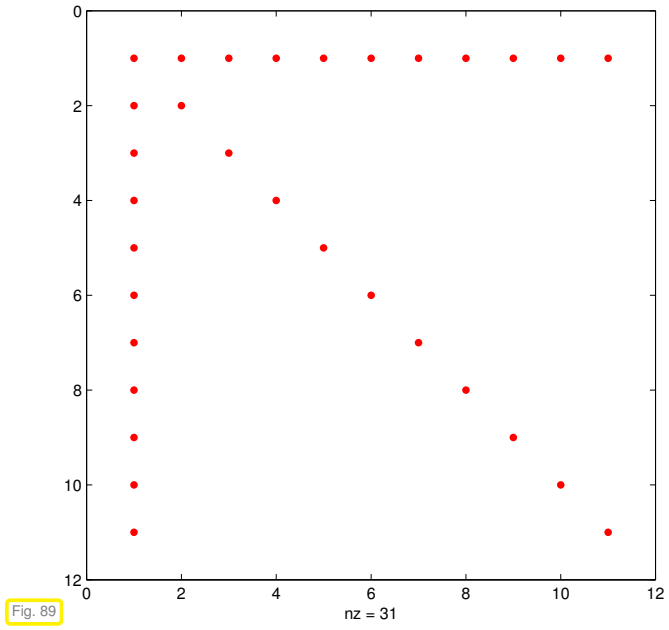


Fig. 89

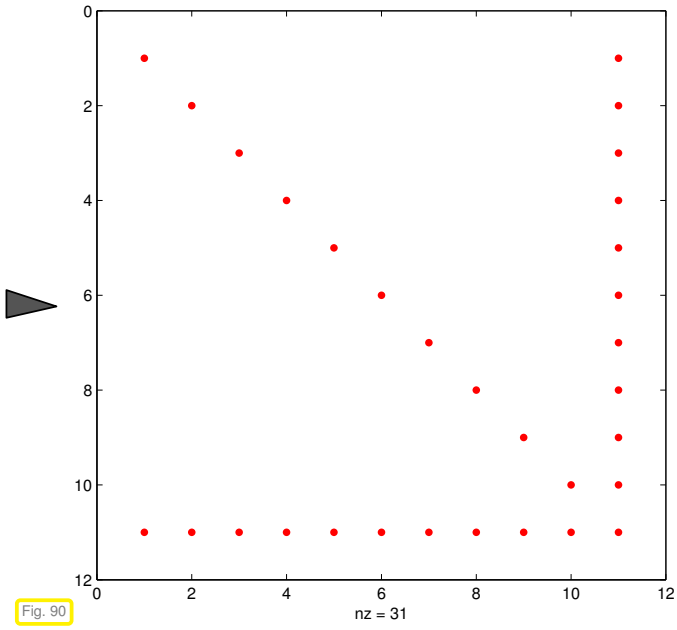


Fig. 90

➢ Then LU-factorization (*without pivoting*) of the resulting matrix requires $O(n)$ operations.

C++11 code 2.7.52: Permuting arrow matrix, see Figs. 89, 90 ↬ GITLAB

```
MatrixXd A(11,11); A.setIdentity();
A.col(0).setOnes(); A.row(0) = RowVectorXd::LinSpaced(11,11,1);
// Permutation matrix (→ Def. 2.3.46) encoding cyclic permutation
MatrixXd P(11,11); P.setZero();
P.topRightCorner(10,10).setIdentity(); P(10,0) = 1;
mgl::Figure fig1, fig2;
fig1.spy(A);    fig1.setFontSize(4);
fig1.save("InvArrowSpy_cpp");
fig2.spy((P*A*P.transpose()).eval()); fig2.setFontSize(4);
fig2.save("ArrowSpy_cpp");
```

Example 2.7.53 (Pivoting destroys sparsity)

In Ex. 2.7.50 we found that permuting a matrix can make it amenable to Gaussian elimination/LU-decomposition with much less fill-in ($\to$ Def. 2.7.47). However, recall from Section 2.3.3 that pivoting, which may be es-

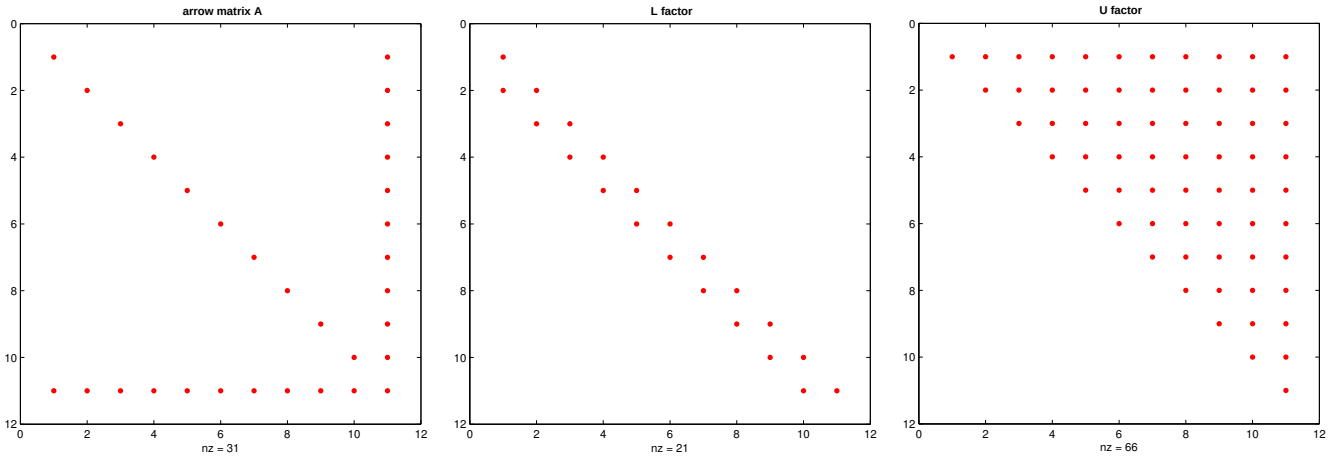
sential for achieving numerical stability, amounts to permuting the rows (or even columns) of the matrix. Thus, we may face the awkward situation that pivoting tries to reverse the very permutation we applied to minimize fill-in! The next example shows that this can happen for an arrow matrix.

C++11 code 2.7.54: fill-in due to pivoting ↪ GITLAB

```
// Study of fill-in with LU-factorization due to pivoting
MatrixXd A(11,11); A.setZero();
A.diagonal() = VectorXd::LinSpaced(11,1,11).cwiseInverse();
A.col(10).setConstant(2); A.row(10).setConstant(2);
auto solver = A.lu();
MatrixXd L = MatrixXd::Identity(11,11);
L += solver.matrixLU().triangularView<StrictlyLower>();
MatrixXd U = solver.matrixLU().triangularView<Upper>();
// Plotting
mgl::Figure fig1, fig2, fig3, fig4;
fig1.spy(A);    fig1.setFontSize(4);
fig1.title("arrow matrix A"); fig1.save("fillinpivotA");
fig2.spy(L);    fig2.setFontSize(4);
fig2.title("L factor");  fig2.save("fillinpivotL");
fig3.spy(U);    fig3.setFontSize(4);
fig3.title("U factor");  fig3.save("fillinpivotU");
std::cout  << A << std::endl;
```

$$\mathbf{A} = \begin{bmatrix} 1 & & & 2 \\ & \frac{1}{2} & & 2 \\ & & \ddots & \vdots \\ & & \frac{1}{10} & 2 \\ 2 & \dots & & 2 \end{bmatrix} \quad \rightarrow \text{ arrow matrix, Ex. 2.7.48}$$

The distributions of non-zero entries of the computed LU-factors ("spy-plots") are as follows:



$\mathbf{A}$ $\mathbf{L}$ $\mathbf{U}$

In this case the solution of a LSE with system matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ of the above type by means of Gaussian elimination with partial pivoting would incur costs of $O(n^3)$.

2.7.6 Banded matrices [?, Sect. 3.7]

Banded matrices are a special class of sparse matrices (→ Notion 2.7.1 with extra structure:

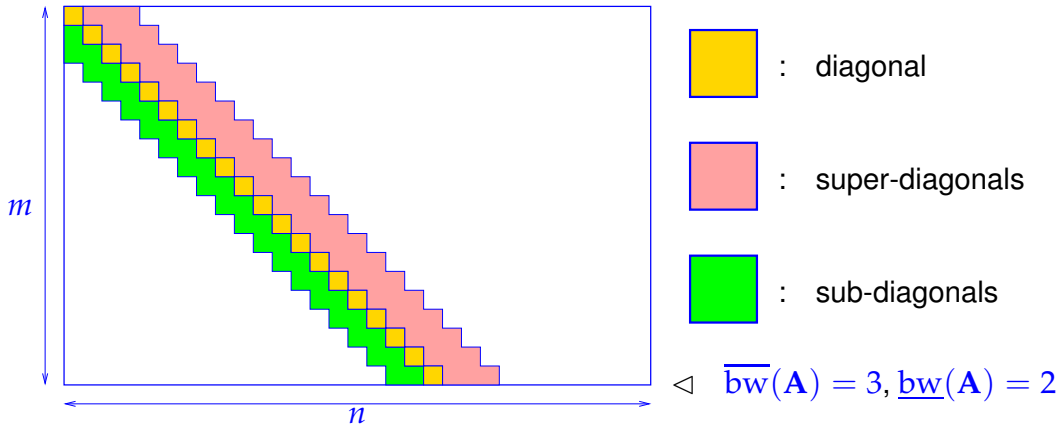
Definition 2.7.55. Bandwidth

For $\mathbf{A} = (a_{ij})_{i,j} \in \mathbb{K}^{m,n}$ we call

$$\overline{\mathrm{bw}}(\mathbf{A}) := \min\{k \in \mathbb{N}: j - i > k \Rightarrow a_{ij} = 0\} \text{ upper bandwidth},$$
$$\underline{\mathrm{bw}}(\mathbf{A}) := \min\{k \in \mathbb{N}: i - j > k \Rightarrow a_{ij} = 0\} \text{ lower bandwidth}.$$

$\mathrm{bw}(\mathbf{A}) := \overline{\mathrm{bw}}(\mathbf{A}) + \underline{m}(\mathbf{A}) + 1$ = bandwidth von $\mathbf{A}$.

- $\mathrm{bw}(\mathbf{A}) = 1 \quad \triangleright \quad \mathbf{A}$ diagonal matrix, → Def. 1.1.5
- $\overline{\mathrm{bw}}(\mathbf{A}) = \underline{\mathrm{bw}}(\mathbf{A}) = 1 \quad \triangleright \quad \mathbf{A}$ tridiagonal matrix
- More general: $\mathbf{A} \in \mathbb{R}^{n,n}$ with $\mathrm{bw}(\mathbf{A}) \ll n \mathrel{\hat{=}}$ banded matrix



◁ $\overline{\mathrm{bw}}(\mathbf{A}) = 3, \underline{\mathrm{bw}}(\mathbf{A}) = 2$

for banded matrix $\mathbf{A} \in \mathbb{K}^{m,n}$: $\mathrm{nnz}(\mathbf{A}) \leq \min\{m,n\}\,\mathrm{bw}(\mathbf{A})$

We now examine a generalization of the concept of a banded matrix that is particularly useful in the context of Gaussian elimination:

Definition 2.7.56. Matrix envelope

For $\mathbf{A} \in \mathbb{K}^{n,n}$ define

row bandwidth $\mathrm{bw}_i^R(\mathbf{A}) := \max\{0, i-j : a_{ij} \neq 0, 1 \leq j \leq n\}, i \in \{1,...,n\}$

column bandwidth $\mathrm{bw}_j^C(\mathbf{A}) := \max\{0, j-i : a_{ij} \neq 0, 1 \leq i \leq n\}, j \in \{1,...,n\}$

envelope $\mathrm{env}(\mathbf{A}) := \left\{ (i,j) \in \{1,\ldots,n\}^2 : \begin{array}{l} i - \mathrm{bw}_i^R(\mathbf{A}) \leq j \leq i\,, \\ j - \mathrm{bw}_j^C(\mathbf{A}) \leq i \leq j \end{array} \right\}$

Example 2.7.57 (Envelope of a matrix)

$$\mathbf{A} = \begin{pmatrix} * & 0 & * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & * & 0 & 0 \\ * & 0 & * & 0 & 0 & 0 & * \\ 0 & 0 & 0 & * & * & 0 & * \\ 0 & * & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & * & * & 0 \\ 0 & 0 & * & * & 0 & 0 & * \end{pmatrix} \quad \begin{aligned} \mathrm{bw}_1^R(A) &= 0 \\ \mathrm{bw}_2^R(A) &= 0 \\ \mathrm{bw}_3^R(A) &= 2 \\ \mathrm{bw}_4^R(A) &= 0 \\ \mathrm{bw}_5^R(A) &= 3 \\ \mathrm{bw}_6^R(A) &= 1 \\ \mathrm{bw}_7^R(A) &= 4 \end{aligned}$$

$\mathrm{env}(A)$ = red entries
$*$ ≙ non-zero matrix entry $a_{ij} \neq 0$

Starting from a "spy-plot", it is easy to find the evelope:

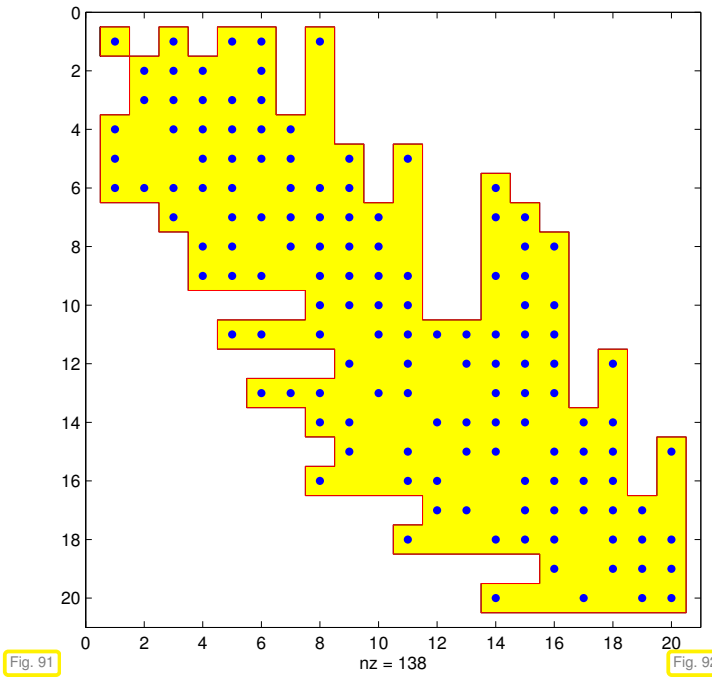


Fig. 91

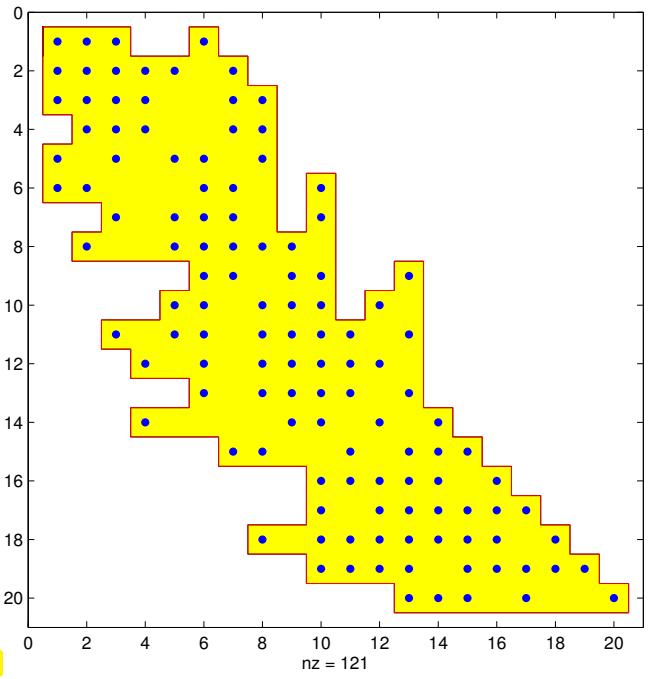


Fig. 92

Note: the envelope of the arrow matrix from Ex. 2.7.48 is just the set of index pairs of its non-zero entries. Hence, the following theorem provides another reason for the sparsity of the LU-factors in that example.

Theorem 2.7.58. Envelope and fill-in → **[?, Sect. 3.9]**

If $\mathbf{A} \in \mathbb{K}^{n,n}$ is regular with LU-factorization $\mathbf{A} = \mathbf{LU}$, then fill-in (→ Def. 2.7.47) is confined to $\mathrm{env}(\mathbf{A})$.

Gaussian elimination *without pivoting*

Proof. (by induction, version I) Examine first step of Gaussian elimination without pivoting, $a_{11} \neq 0$

$$\mathbf{A} = \begin{pmatrix} a_{11} & \mathbf{b}^\top \\ \mathbf{c} & \tilde{\mathbf{A}} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ -\frac{\mathbf{c}}{a_{11}} & \mathbf{I} \end{pmatrix}}_{\mathbf{L}^{(1)}} \underbrace{\begin{pmatrix} a_{11} & \mathbf{b}^\top \\ 0 & \tilde{\mathbf{A}} - \frac{\mathbf{c}\mathbf{b}^\top}{a_{11}} \end{pmatrix}}_{\mathbf{U}^{(1)}}$$

$$\text{If } (i,j) \notin \mathrm{env}(\mathbf{A}) \quad \Rightarrow \quad \begin{cases} c_{i-1} = 0 & \text{, if } i > j\,, \\ b_{j-1} = 0 & \text{, if } i < j\,. \end{cases}$$

$$\Rightarrow \quad \mathrm{env}(\mathbf{L}^{(1)}) \subset \mathrm{env}(\mathbf{A}), \quad \mathrm{env}(\mathbf{U}^{(1)}) \subset \mathrm{env}(\mathbf{A})\,.$$

Moreover, $\mathrm{env}(\tilde{\mathbf{A}} - \frac{\mathbf{c}\mathbf{b}^\top}{a_{11}}) = \mathrm{env}(\mathbf{A}(2:n, 2:n))$ □

Proof. (by induction, version II) Use block-LU-factorization, *cf.* Rem. 2.3.34 and proof of Lemma 2.3.19:

$$\left(\begin{array}{c|c}\widetilde{\mathbf{A}} & \mathbf{b}\\ \hline \mathbf{c}^\top & \alpha\end{array}\right) = \left(\begin{array}{c|c}\widetilde{\mathbf{L}} & 0\\ \hline \mathbf{l}^\top & 1\end{array}\right)\left(\begin{array}{c|c}\widetilde{\mathbf{U}} & \mathbf{u}\\ \hline 0 & \xi\end{array}\right) \quad \Rightarrow \quad \begin{array}{l}\widetilde{\mathbf{U}}^\top\mathbf{l} = \mathbf{c}\,,\\ \widetilde{\mathbf{L}}\mathbf{u} = \mathbf{b}\,.\end{array} \tag{2.7.59}$$

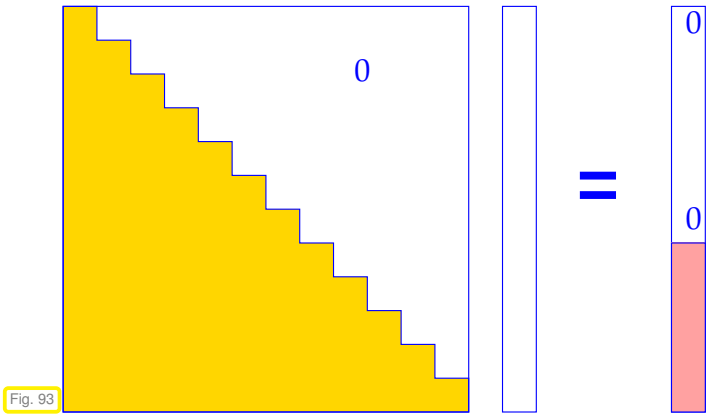


Fig. 93

From Def. 2.7.56:

If $m_n^R(\mathbf{A}) = m$, then $c_1,\ldots,c_{n-m} = 0$ (entries of $\mathbf{c}$ from (2.7.59))

If $m_n^C(\mathbf{A}) = m$, then $b_1,\ldots,b_{n-m} = 0$ (entries of $\mathbf{b}$ from (2.7.59))

$\lhd$ for lower triagular LSE:

If $c_1,\ldots,c_k = 0$ then $l_1,\ldots,l_k = 0$
If $b_1,\ldots,b_k = 0$, then $u_1,\ldots,u_k = 0$

$\Downarrow$

assertion of the theorem $\square$

Thm. 2.7.58 immediately suggests a policy for saving cmputational effort when solving linear system whose system matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ is sparse due to *small envelope*:

$$\sharp\,\mathrm{env}(\mathbf{A}) \ll n^2 :$$

Policy: Confine elimination to envelope!

Details will be given now:

Envelope-aware LU-factorization:

C++11 code 2.7.60: Computing row bandwidths, → Def. 2.7.56 ➙ GITLAB

```
2  //!  computes rowbandwidth numbers m_i^R(A) of A (sparse matrix) according
       to Def. 2.7.56
3  template <class numeric_t>
4  VectorXi rowbandwidth(const SparseMatrix<numeric_t> &A){
5      VectorXi m = VectorXi::Zero(A.rows());
6      for(int k = 0; k < A.outerSize(); ++k)
7          for(typename SparseMatrix<numeric_t>::InnerIterator
               it(A,k); it; ++it)
8            m(it.row()) = std::max<VectorXi::Scalar>(m( it.row()
                ), it.row()-it.col() );
9      return m;
10 }
11 //!  computes row bandwidth numbers m_i^R(A) of A (dense matrix) according
       to Def. 2.7.56
12 template <class Derived>
13 VectorXi rowbandwidth(const MatrixBase<Derived> &A){
14     VectorXi m = VectorXi::Zero(A.rows());
15     for(int i = 1; i < A.rows(); ++i)
16         for(int j = 0; j < i; ++j)
17             if(A(i,j) != 0){
```

```
                    m(i) = i - j;
                    break;
                }
        return m;
}
```

C++11 code 2.7.61: Envelope aware forward substitution ↪ GITLAB

```
//!  evelope aware forward substitution for Lx = y
//!  (L = lower triangular matrix)
//!  argument mr:  row bandwidth vector
VectorXd substenv(const MatrixXd &L, const VectorXd &y, const
   VectorXi &mr){
      int n = L.cols(); VectorXd x(n);
      x(0) = y(0)/L(0,0);
      for(int i = 1; i < n; ++i){
           if( mr(i) > 0){
                double zeta = L.row(i).segment(i-mr(i),mr(i))
                          * x.segment(i-mr(i), mr(i));
                x(i) = (y(i) - zeta)/L(i,i);
           }
           else x(i) = y(i) / L(i,i);
      }
      return x;
}
```

Asymptotic complexity of envelope aware forward substitution, *cf.* § 2.3.30, for $\mathbf{Lx} = \mathbf{y}$, $\mathbf{L} \in \mathbb{K}^{n,n}$ regular lower triangular matrix is

$$O(\#\mathrm{env}(\mathbf{L}))\ !$$

By block LU-factorization ($\rightarrow$ Rem. 2.3.34) we find

$$\left(\begin{array}{c|c} (\mathbf{A})_{1:n-1,1:n-1} & (\mathbf{A})_{1:n-1,n} \\ \hline (\mathbf{A})_{n,1:n-1} & (\mathbf{A})_{n,n} \end{array}\right) = \left(\begin{array}{c|c} \mathbf{L}_1 & 0 \\ \hline \mathbf{l}^\top & 1 \end{array}\right)\left(\begin{array}{c|c} \mathbf{U}_1 & \mathbf{u} \\ \hline 0 & \gamma \end{array}\right), \tag{2.7.62}$$

$$\Rightarrow \quad (\mathbf{A})_{1:n-1,1:n-1} = \mathbf{L}_1\mathbf{U}_1\,,\quad \mathbf{L}_1\mathbf{u} = (\mathbf{A})_{1:n-1,n}\,,\quad \mathbf{U}_1^\top\mathbf{l} = (\mathbf{A})_{n,1:n-1}^\top\,,\quad \mathbf{l}^\top\mathbf{u} + \gamma = (\mathbf{A})_{n,n}\,. \tag{2.7.63}$$

C++11 code 2.7.64: Envelope aware recursive LU-factorization ↪ GITLAB

```
//!  envelope aware recursive LU-factorization
//!  of structurally symmetric matrix
void luenv(const MatrixXd &A, MatrixXd &L, MatrixXd &U){
      int n = A.cols();
      assert(n == A.rows() && "A must be square");
      if(n == 1){ L.setIdentity(); U = A;}
      else{
```

```
9           VectorXi mr = rowbandwidth(A); // = colbandwidth thanks to
               symmetry
10          double gamma;
11          MatrixXd L1(n-1,n-1), U1(n-1,n-1);
12          luenv(A.topLeftCorner(n-1, n-1), L1, U1);
13          VectorXd u = substenv(L1, A.col(n-1).head(n-1), mr);
14          VectorXd l = substenv(U1.transpose(),
               A.row(n-1).head(n-1).transpose(), mr);
15          if(mr(n-1) > 0)
16              gamma = A(n-1,n-1) -
                   l.tail(mr(n-1)).dot(u.tail(mr(n-1)));
17          else
18              gamma = A(n-1,n-1);
19          L.topLeftCorner(n-1,n-1) = L1;      L.col(n-1).setZero();
20          L.row(n-1).head(n-1) = l.transpose();   L(n-1,n-1) = 1;
21          U.topLeftCorner(n-1,n-1) = U1;      U.col(n-1).head(n-1) = u;
22          U.row(n-1).setZero();     U(n-1,n-1) = gamma;
23      }
24  }
```

recursive implementation of envelope aware recursive LU-factorization (no pivoting !)

Assumption:

$$\mathbf{A} \in \mathbb{K}^{n,n} \text{ is}$$
structurally symmetric

Asymptotic complexity ($\mathbf{A} \in \mathbb{K}^{n,n}$)

$$O(n \cdot \#\mathrm{env}(\mathbf{A})) \,.$$

Definition 2.7.65. Structurally symmetric matrix

$\mathbf{A} \in \mathbb{K}^{n,n}$ is structurally symmetric, if

$$(\mathbf{A})_{i,j} \neq 0 \quad \Leftrightarrow \quad (\mathbf{A})_{j,i} \neq 0 \quad \forall i,j \in \{1,\ldots,n\} \,.$$

Since by Thm. 2.7.58 fill-in is confined to the envelope, we need store only the matrix entries a_{ij}, $(i,j) \in \mathrm{env}(\mathbf{A})$ when computing (in situ) LU-factorization of *structurally symmetric*$\mathbf{A} \in \mathbb{K}^{n,n}$

- ➤ Storage required: $n + 2\sum_{i=1}^{n} m_i(\mathbf{A})$ floating point numbers
- ➤ terminology: envelope oriented matrix storage

Example 2.7.66 (Envelope oriented matrix storage)

Linear envelope oriented matrix storage of *symmetric* $\mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n}$:

Two arrays:
`scalar_t * val` size P,
`size_t * dptr` size n

$$P := n + \sum_{i=1}^{n} m_i(A) \,. \qquad (2.7.67)$$

$$\mathbf{A} = \begin{pmatrix} * & 0 & * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & * & 0 & 0 \\ * & 0 & * & 0 & 0 & 0 & * \\ 0 & 0 & 0 & * & * & 0 & * \\ 0 & * & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & * & * & 0 \\ 0 & 0 & * & * & 0 & 0 & * \end{pmatrix}$$

Indexing rule:

$$\texttt{dptr}[j] = k \quad \Updownarrow \quad \texttt{val}[k] = a_{jj}$$

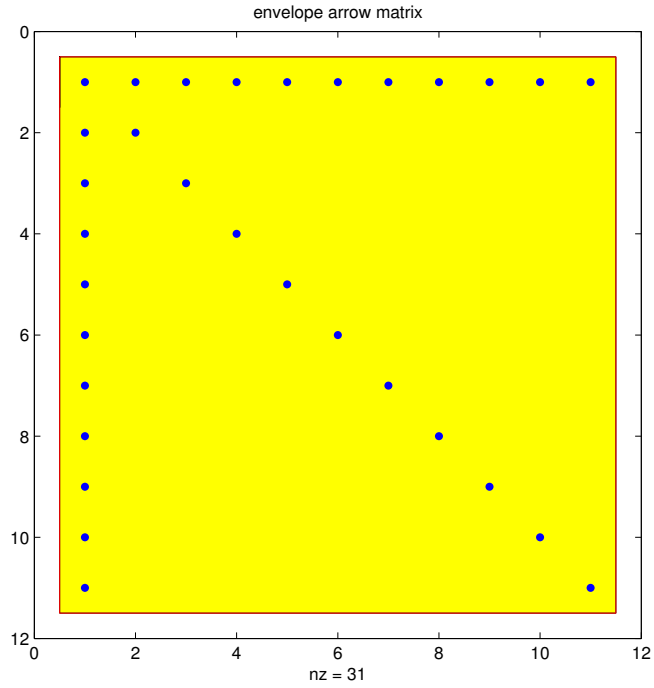
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
val		a_{11}	a_{22}	a_{31}	a_{32}	a_{33}	a_{44}	a_{52}	a_{53}	a_{54}	a_{55}	a_{65}	a_{66}	a_{73}	a_{74}	a_{75}	a_{76}	a_{77}
dptr	0	1	2	5	6	10	12	17										

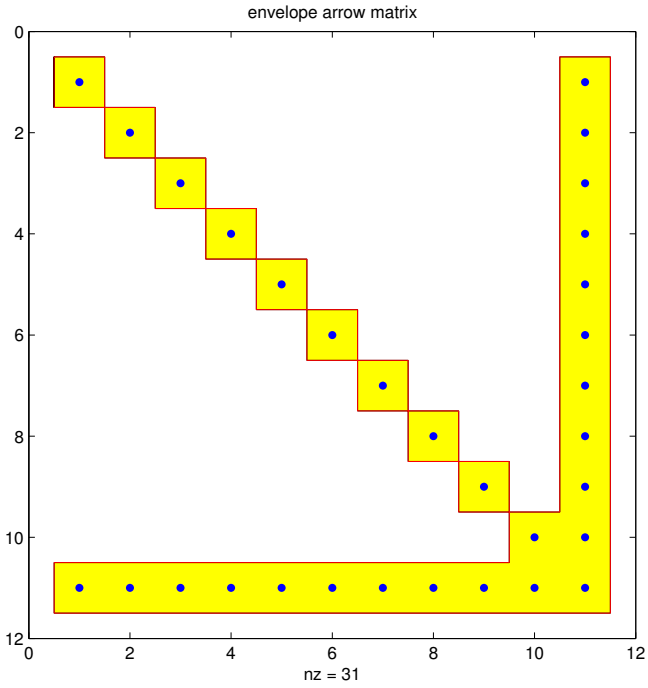
Minimizing bandwidth/envelope:

Goal: Minimize $m_i(\mathbf{A})$,$\mathbf{A} = (a_{ij}) \in \mathbb{R}^{N,N}$, by permuting rows/columns of $\mathbf{A}$

Example 2.7.68 (Reducing bandwidth by row/column permutations)

Recall: cyclic permutation of rows/columns of arrow matrix applied in Ex. 2.7.50. This can be viewed as a drastic shrinking of the envelope:





Another example:Reflection at cross diagonal ➤ reduction of $\#\mathrm{env}(\mathbf{A})$

$$\begin{bmatrix} * & 0 & 0 & * & * & * \\ 0 & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 & 0 \\ * & 0 & 0 & * & * & * \\ * & 0 & 0 & * & * & * \\ * & 0 & 0 & * & * & * \end{bmatrix} \longrightarrow \begin{bmatrix} * & * & * & 0 & 0 & * \\ * & * & * & 0 & 0 & * \\ * & * & * & 0 & 0 & * \\ 0 & 0 & 0 & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & 0 \\ * & * & * & 0 & 0 & * \end{bmatrix}$$

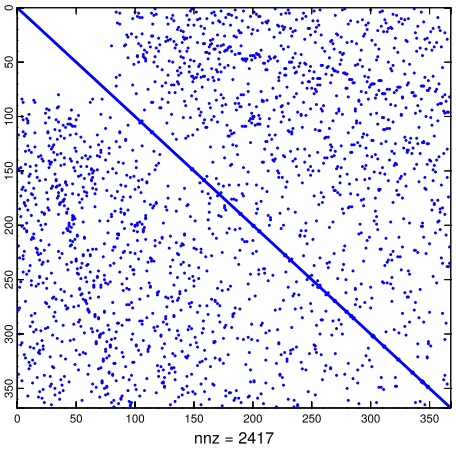
$$i \leftarrow N+1-i$$

$\#\mathrm{env}(\mathbf{A}) = 30$ $\#\mathrm{env}(\mathbf{A}) = 22$

Example 2.7.69 (Reducing fill-in by reordering)

Envelope reducing permutations are at the heart of all modern sparse solvers (→ § 2.7.43). They employ elaborate algorithms for the analysis of matrix graphs, that is, the connections between components of the vector of unknowns defined by non-zero entries of the matrix. For further discussion see [**?**, Sect. 5.7].

EIGEN supplies a few ordering methods for sparse matrices. These methods use permutations to aim for minimal bandwidth/envelop of a given sparse matrix. We study an example with a 347×347 matrix $\mathbf{M}$ originating in the numerical solution of partial differential equations, *cf.* Rem. 2.7.5.



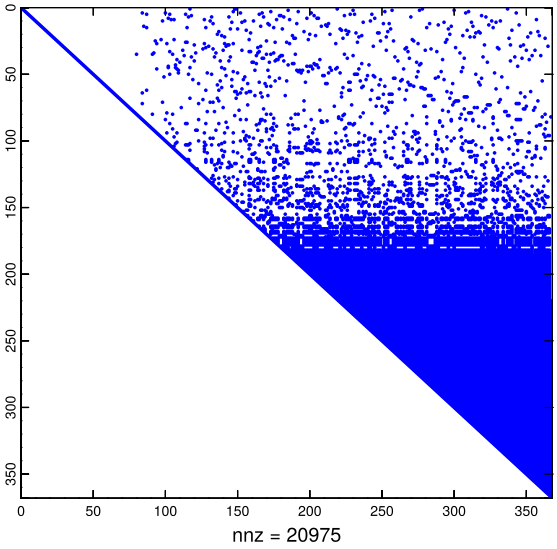
Pattern of $\mathbf{M}$ ➥

(Here: no row swaps from pivoting !)

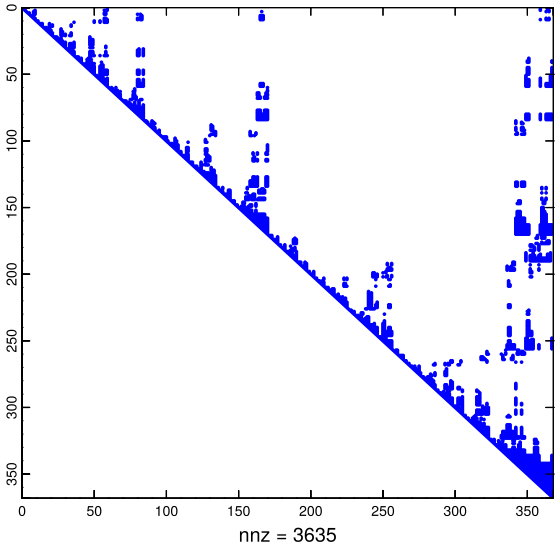
C++11 code 2.7.70: preordering in EIGEN ⇝ GITLAB

```
// L and U cannot be extracted from SparseLU -> LDLT
SimplicialLDLT<SpMat_t, Lower, AMDOrdering<int> > solver1(M);
SimplicialLDLT<SpMat_t, Lower, NaturalOrdering<int> > solver2(M);
MatrixXd U1 = MatrixXd(solver1.matrixU());
MatrixXd U2 = MatrixXd(solver2.matrixU());
// Plotting
mgl::Figure fig1, fig2, fig3;
fig1.spy(M);   fig1.setFontSize(4); fig1.save("Mspy");
fig2.spy(U1);  fig2.setFontSize(4); fig2.save("AMDUSpy");
fig3.spy(U2);  fig3.setFontSize(4); fig3.save("NaturalUSpy");
```

Examine patterns of LU-factors (→ Sect. 2.3.2) after reordering:



no reordering



approximate minimum degree

2.8 Stable Gaussian elimination without pivoting

Recall insights, examples and experimnents:

- Thm. 2.7.58 ➢ special structure of the matrix helps avoid fill-in in Gaussian elimination/LU-factorization *without* pivoting.
- Ex. 2.7.53 ➢ pivoting can trigger huge fill-in that would not occur without it.
- Ex. 2.7.69 ➢ fill-in reducing effect of reordering can be thwarted by later row swapping in the course of pivoting.
- **BUT** pivoting is essential for stability of Gaussian elimination/LU-factorization → Ex. 2.3.36.

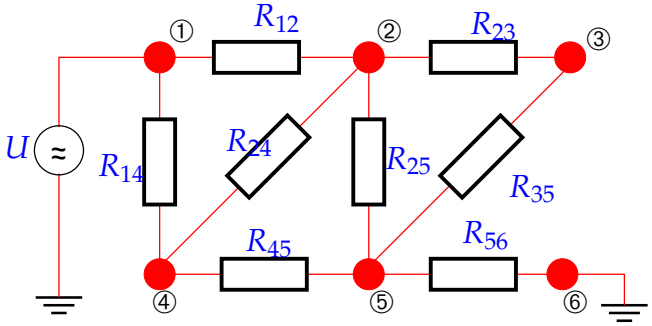
Very desirable: a priori criteria, when Gaussian elimination/LU-factorization remains stable even without pivoting. This can help avoid the extra work for partial pivoting and makes it possible to exploit structure without worrying about stability.

This section will introduce classes of matrices that allow Gaussian elimination without pivoting. Fortunately, linear systems of equations featuring system matrices from these classes are very common in applications.

Example 2.8.1 (Diagonally dominant matrices from nodal analysis → Ex. 2.1.3)

Consider:

electrical circuit entirely composed of Ohmic resistors.



Circuit equations from nodal analysis, see Ex. 2.1.3:

$$\begin{aligned}
② &: R_{12}^{-1}(U_2-U_1)+R_{23}^{-1}(U_2-U_3)+R_{24}^{-1}(U_2-U_4)+R_{25}^{-1}(U_2-U_5) = 0\,,\\
③ &: R_{23}^{-1}(U_3-U_2)+R_{35}^{-1}(U_3-U_5) = 0\,,\\
④ &: R_{14}^{-1}(U_4-U_1)+R_{24}^{-1}(U_4-U_2)+R_{45}^{-1}(U_4-U_5) = 0\,,\\
⑤ &: R_{25}^{-1}(U_5-U_2)+R_{35}^{-1}(U_5-U_3)+R_{45}^{-1}(U_5-U_4)+R_{56}(U_5-U_6) = 0\,,\\
&U_1 = U \quad , \quad U_6 = 0\,.
\end{aligned}$$

$$\begin{pmatrix}
\frac{1}{R_{12}}+\frac{1}{R_{23}}+\frac{1}{R_{24}}+\frac{1}{R_{25}} & -\frac{1}{R_{23}} & -\frac{1}{R_{24}} & -\frac{1}{R_{25}} \\
-\frac{1}{R_{23}} & \frac{1}{R_{23}}+\frac{1}{R_{35}} & 0 & -\frac{1}{R_{35}} \\
-\frac{1}{R_{24}} & 0 & \frac{1}{R_{24}}+\frac{1}{R_{45}} & -\frac{1}{R_{45}} \\
-\frac{1}{R_{25}} & -\frac{1}{R_{35}} & -\frac{1}{R_{45}} & \frac{1}{R_{22}}+\frac{1}{R_{35}}+\frac{1}{R_{45}}+\frac{1}{R_{56}}
\end{pmatrix}
\begin{pmatrix} U_2 \\ U_3 \\ U_4 \\ U_5 \end{pmatrix}
= \begin{pmatrix} \frac{1}{R_{12}} \\ 0 \\ \frac{1}{R_{14}} \\ 0 \end{pmatrix} U$$

➢ Matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ arising from nodal analysis satisfies

- $\mathbf{A} = \mathbf{A}^\top$, $a_{kk} > 0$, $a_{kj} \le 0$ for $k \neq j$, (2.8.2)

- $\sum_{j=1}^{n} a_{kj} \geq 0\,, \quad k = 1,\ldots,n\,,$ (2.8.3)
- $\mathbf{A}$ is regular. (2.8.4)

All these properties are obvious except for the fact that $\mathbf{A}$ is regular.

Proof of (2.8.4): By Thm. 2.2.4 it suffices to show that the nullspace of $\mathbf{A}$ is trivial: $\mathbf{Ax} = 0 \quad \Rightarrow \quad \mathbf{x} = 0$.

Pick $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{Ax} = 0$, and $i \in \{1,\ldots,n\}$ so that

$$|x_i| = \max\{|x_j|,\ j = 1,\ldots,n\}\,.$$

Intermediate goal: show that all entries of $\mathbf{x}$ are the same

$$\mathbf{Ax} = 0 \quad \Rightarrow \quad x_i = \sum_{j\neq i} \frac{a_{ij}}{a_{ii}} x_j \quad \Rightarrow \quad |x_i| \leq \sum_{j\neq i} \frac{|a_{ij}|}{|a_{ii}|} |x_j|\,. \tag{2.8.5}$$

By (2.8.3) and the sign condition from (2.8.2) we conclude

$$\sum_{j\neq i} \frac{|a_{ij}|}{|a_{ii}|} \leq 1\,. \tag{2.8.6}$$

Hence, (2.8.6) combined with the above estimate (2.8.5) that tells us that the maximum is smaller equal than a mean implies $|x_j| = |x_i|$ for all $j = 1,\ldots,n$. Finally, the sign condition $a_{kj} \leq 0$ for $k \neq j$ enforces the same sign of all x_i. Thus, we conclude, w.l.o.g., $x_1 = x_2 = \cdots = x_n$. As

$$\exists i \in \{1,\ldots,n\}\colon \quad \sum_{j=1}^{n} a_{ij} > 0 \quad \text{(strict inequality)}\,,$$

$\mathbf{Ax} = 0$ is only possible for $\mathbf{x} = 0$.

(2.8.7) Diagonally dominant matrices

Definition 2.8.8. Diagonally dominant matrix → **[?, Def. 1.24]**

$\mathbf{A} \in \mathbb{K}^{n,n}$ is diagonally dominant, if

$$\forall k \in \{1,\ldots,n\}\colon \quad \sum_{j\neq k} |a_{kj}| \leq |a_{kk}|\,.$$

The matrix $\mathbf{A}$ is called strictly diagonally dominant, if

$$\forall k \in \{1,\ldots,n\}\colon \quad \sum_{j\neq k} |a_{kj}| < |a_{kk}|\,.$$

Lemma 2.8.9. LU-factorization of diagonally dominant matrices

$\mathbf{A}$ *regular, diagonally dominant with positive diagonal* $\Leftrightarrow$ $\left\{ \begin{array}{c} \mathbf{A} \text{ has LU-factorization} \\ \Updownarrow \\ \text{Gaussian elimination feasible without pivoting}^{(*)} \end{array} \right.$

$(*)$: partial pivoting & diagonally dominant matrices ➢ triggers no row permutations **!**

((2.3.42) will always be satisfied for $j = k$)

➢ Pivoting can be dispensed with without compromising stability.

Proof.(of Lemma 2.8.9)

(2.3.12) $\rightarrow$ induction w.r.t. n:

Clear: partial pivoting in the first step selects a_{11} as pivot element, *cf.* (2.3.42).

After 1st step of elimination:

$$a_{ij}^{(1)} = a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j} , \quad i,j = 2,\ldots,n \quad \Rightarrow \quad a_{ii}^{(1)} > 0 .$$

$$\begin{aligned} |a_{ii}^{(1)}| - \sum_{\substack{j=2 \\ j\neq i}}^{n} |a_{ij}^{(1)}| &= \left| a_{ii} - \frac{a_{i1}}{a_{11}} a_{1i} \right| - \sum_{\substack{j=2 \\ j\neq i}}^{n} \left| a_{ij} - \frac{a_{i1}}{a_{11}} a_{1j} \right| \\ &\geq a_{ii} - \frac{|a_{i1}||a_{1i}|}{a_{11}} - \sum_{\substack{j=2 \\ j\neq i}}^{n} |a_{ij}| - \frac{|a_{i1}|}{a_{11}} \sum_{\substack{j=2 \\ j\neq i}}^{n} |a_{1j}| \\ &\geq a_{ii} - \frac{|a_{i1}||a_{1i}|}{a_{11}} - \sum_{\substack{j=2 \\ j\neq i}}^{n} |a_{ij}| - |a_{i1}| \frac{a_{11} - |a_{1i}|}{a_{11}} \geq a_{ii} - \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \geq 0 . \end{aligned}$$

$\mathbf{A}$ regular, diagonally dominant $\Rightarrow$ partial pivoting according to (2.3.42) selects i-th row in i-th step.

(2.8.10) Gaussian elimination for symmetric positive definite (s.p.d.) matrices

The class of symmetric positive definite (s.p.d.) matrices has been defined in Def. 1.1.8. They permit stable Gaussian elimintation without pivoting:

Theorem 2.8.11. Gaussian elimination for s.p.d. matrices

Every symmetric/Hermitian positive definite matrix ($\rightarrow$ Def. 1.1.8) possesses an LU-decomposition ($\rightarrow$ Sect. 2.3.2).

Equivalent to the assertion of the theorem: Gaussian elimination is feasible *without pivoting*

In fact, this theorem is a corollary of Lemma 2.3.19, because all principal minors of an s.p.d. matrix are s.p.d. themselves.

Sketch of alternative self-contained proof.

Proof by induction: consider first step of elimination

$$\mathbf{A} = \begin{pmatrix} a_{11} & \mathbf{b}^\top \\ \mathbf{b} & \widetilde{\mathbf{A}} \end{pmatrix} \xrightarrow[\text{Gaussian elimination}]{\text{1. step}} \begin{pmatrix} a_{11} & \mathbf{b}^\top \\ 0 & \widetilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}} \end{pmatrix} .$$

➢ to show: $\widetilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}}$ s.p.d. ($\rightarrow$ step of induction argument)

✦ Evident: symmetry of $\widetilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}} \in \mathbb{R}^{n-1,n-1}$

✦ As $\mathbf{A}$ s.p.d. ($\rightarrow$ Def. 1.1.8), for every $\boldsymbol{y} \in \mathbb{R}^{n-1} \setminus \{0\}$

$$0 < \begin{pmatrix} -\frac{\mathbf{b}^\top \mathbf{y}}{a_{11}} \\ \mathbf{y} \end{pmatrix}^\top \begin{pmatrix} a_{11} & \mathbf{b}^\top \\ \mathbf{b} & \widetilde{\mathbf{A}} \end{pmatrix} \begin{pmatrix} -\frac{\mathbf{b}^\top \mathbf{y}}{a_{11}} \\ \mathbf{y} \end{pmatrix} = \boldsymbol{y}^\top (\widetilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}}) \boldsymbol{y} \,.$$

⇚ $\widetilde{\mathbf{A}} - \frac{\mathbf{b}\mathbf{b}^\top}{a_{11}}$ positive definite. □

The proof can also be based on the identities

$$\left(\begin{array}{c|c} (\mathbf{A})_{1:n-1,1:n-1} & (\mathbf{A})_{1:n-1,n} \\ \hline (\mathbf{A})_{n,1:n-1} & (\mathbf{A})_{n,n} \end{array}\right) = \left(\begin{array}{c|c} \mathbf{L}_1 & 0 \\ \hline \mathbf{l}^\top & 1 \end{array}\right) \left(\begin{array}{c|c} \mathbf{U}_1 & \mathbf{u} \\ \hline 0 & \gamma \end{array}\right) , \tag{2.7.62}$$

$$\Rightarrow \quad (\mathbf{A})_{1:n-1,1:n-1} = \mathbf{L}_1\mathbf{U}_1 \,, \quad \mathbf{L}_1\mathbf{u} = (\mathbf{A})_{1:n-1,n} \,, \quad \mathbf{U}_1^\top \mathbf{l} = (\mathbf{A})_{n,1:n-1}^\top \,, \quad \mathbf{l}^\top \mathbf{u} + \gamma = (\mathbf{A})_{n,n} \,,$$

noticing that the principal minor $(\mathbf{A})_{1:n-1,1:n-1}$ is also s.p.d. This allows a simple induction argument.

Note: no pivoting required ($\rightarrow$ Sect. 2.3.3)
(partial pivoting always picks current pivot row)

The next result gives a useful criterion for telling whether a given symmetric/Hermitian matrix is s.p.d.:

Lemma 2.8.12. Diagonal dominance and definiteness

A diagonally dominant Hermitian/symmetric matrix with non-negative diagonal entries is positive semi-*definite.*
A strictly *diagonally dominant Hermitian/symmetric matrix with positive diagonal entries is positive definite.*

Proof. For $\mathbf{A} = \mathbf{A}^{\mathrm{H}}$ diagonally dominant, use inequality between arithmetic and geometric mean (AGM) $ab \le \frac{1}{2}(a^2 + b^2)$:

$$\begin{aligned} \mathbf{x}^{\mathrm{H}}\mathbf{A}\mathbf{x} &= \sum_{i=1}^n \Big(a_{ii}|x_i|^2 + \sum_{i\neq j} a_{ij}\bar{x}_i x_j \Big) \ge \sum_{i=1}^n \Big(a_{ii}|x_i|^2 - \sum_{i\neq j} |a_{ij}||x_i||x_j| \Big) \\ &\overset{\text{AGM}}{\ge} \sum_{i=1}^n a_{ii}|x_i|^2 - \tfrac{1}{2}\sum_{i\neq j} |a_{ij}|(|x_i|^2 + |x_j|^2) \\ &\ge \tfrac{1}{2}\Big(\sum_{i=1}^n \{a_{ii}|x_i|^2 - \sum_{j\neq i} |a_{ij}||x_i|^2\} \Big) + \tfrac{1}{2}\Big(\sum_{j=1}^n \{a_{ii}|x_j|^2 - \sum_{i\neq j} |a_{ij}||x_j|^2\} \Big) \\ &\ge \sum_{i=1}^n |x_i|^2 \Big(a_{ii} - \sum_{j\neq i} |a_{ij}| \Big) \ge 0 \,. \end{aligned}$$

(2.8.13) Cholesky decomposition

Lemma 2.8.14. Cholesky decomposition for s.p.d. matrices → [?, Sect. 3.4], [?, Sect. II.5], [?, Thm. 3.6]

For any s.p.d. $\mathbf{A} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, *there is a unique upper triangular matrix* $\mathbf{R} \in \mathbb{K}^{n,n}$ *with* $r_{ii} > 0$, $i = 1, \ldots, n$, *such that* $\mathbf{A} = \mathbf{R}^{\mathrm{H}}\mathbf{R}$ *(Cholesky decomposition).*

Thm. 2.8.11 $\Rightarrow$ $\mathbf{A} = \mathbf{LU}$ (unique LU-decomposition of $\mathbf{A}$, Lemma 2.3.19)

$$\mathbf{A} = \mathbf{L}\mathbf{D}\widetilde{\mathbf{U}} \quad , \quad \begin{array}{l} \mathbf{D} \mathrel{\widehat{=}} \text{diagonal of } \mathbf{U} \, , \\ \widetilde{\mathbf{U}} \mathrel{\widehat{=}} \textit{normalized} \text{ upper triangular matrix} \rightarrow \text{Def. 1.1.5} \end{array}$$

Due to uniqueness of LU-decomposition

$$\mathbf{A} = \mathbf{A}^{\top} \;\Rightarrow\; \mathbf{U} = \mathbf{D}\mathbf{L}^{\top} \;\Rightarrow\; \boxed{\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^{\top}} \, ,$$

with unique $\mathbf{L}$, $\mathbf{D}$ (diagonal matrix)

$$\mathbf{x}^{\top}\mathbf{A}\mathbf{x} > 0 \quad \forall \mathbf{x} \neq 0 \quad \Rightarrow \quad \mathbf{y}^{\top}\mathbf{D}\mathbf{y} > 0 \quad \forall \mathbf{y} \neq 0 \, .$$

➤ D has positive diagonal ➨ $\mathbf{R} = \sqrt{\mathbf{D}}\mathbf{L}^{\top}$. □

Formulas analogous to (2.3.22)

$$\mathbf{R}^{\mathrm{H}}\mathbf{R} = \mathbf{A} \;\Rightarrow\; a_{ik} = \sum_{j=1}^{\min\{i,k\}} \overline{r}_{ji} r_{jk} = \begin{cases} \sum_{j=1}^{i-1} \overline{r}_{ji} r_{jk} + \overline{r}_{ii} r_{ik} & \text{, if } i < k \, , \\ \sum_{j=1}^{i-1} |r_{ji}|^2 + r_{ii}^2 & \text{, if } i = k \, . \end{cases} \tag{2.8.15}$$

C++11 code 2.8.16: Simple Cholesky factorization ↪ GITLAB

```
//!  simple Cholesky factorization
void cholfac(const MatrixXd &A, MatrixXd &R){
    int n = A.rows();
    R = A;
    for(int k = 0; k < n; ++k){
        for(int j = k+1; j < n; ++j)
            R.row(j).tail(n-j) -= R.row(k).tail(n-j)*R(k,j)/R(k,k);
        R.row(k).tail(n-k) /= std::sqrt(R(k,k));
    }
    R.triangularView<StrictlyLower>().setZero();
}
```

Computational costs (# elementary arithmetic operations) of Cholesky decomposition: $\frac{1}{6}n^3 + O(n^2)$

(➢ "half the costs" of LU-factorization, *cf.* Code in § 2.3.21, but this does not mean "twice as fast" in a concrete implementation, because memory access patterns will have a crucial impact, see Rem. 1.4.8.)

Gains of efficiency hardly justify the use of Cholesky decomposition in modern numerical algorithms. Savings in memory compared to standard LU-factorization (only one factor $\mathbf{R}$ has to be stored) offer a stronger reason to prefer the Cholesky decomposition.

The computation of Cholesky-factorization by means of the algorithm of Code 2.8.16 is numerically stable (→ Def. 1.5.85)!

Reason: recall Thm. 2.4.4: Numerical instability of Gaussian elimination (with any kind of pivoting) manifests itself in massive growth of the entries of intermediate elimination matrices $\mathbf{A}^{(k)}$.

Use the relationship between LU-factorization and Cholesky decomposition, which tells us that we only have to monitor the growth of entries of intermediate upper triangular "Cholesky factorization matrices" $\mathbf{A} = (\mathbf{R}^{(k)})^{\mathrm{H}}\mathbf{R}^{(k)}$.

Consider: Euclidean vector norm/matrix norm (→ Def. 1.5.76) $\|\cdot\|_2$

$$\mathbf{A} = \mathbf{R}^{\mathrm{H}}\mathbf{R} \quad \Rightarrow \quad \|\mathbf{A}\|_2 = \sup_{\|\mathbf{x}\|_2=1} \mathbf{x}^{\mathrm{H}}\mathbf{R}^{\mathrm{H}}\mathbf{R}\mathbf{x} = \sup_{\|\mathbf{x}\|_2=1} (\mathbf{R}\mathbf{x})^{\mathrm{H}}(\mathbf{R}\mathbf{x}) = \|\mathbf{R}\|_2^2 .$$

➤ For all intermediate Cholesky factorization matrices holds: $\left\|(\mathbf{R}^{(k)})^{\mathrm{H}}\right\|_2 = \left\|\mathbf{R}^{(k)}\right\|_2 = \|\mathbf{A}\|_2^{1/2}$

This rules out a blowup of entries of the $\mathbf{R}^{(k)}$.

Computation of the Cholesky decomposition largely agrees with the computation of LU-factorization (without pivoting). Using the latter together with forward and backward substitution (→ Sect. 2.3.2) to solve a linear system of equations is algebraically and numerically equivalent to using Gaussian elimination without pivoting.

From these equivalences we conclude:

Solving LSE with s.p.d. system matrix via
Cholesky decomposition + forward & backward substitution
is numerically stable (→ Def. 1.5.85)

⇕

Gaussian elimination *without pivoting* is a *numerically stable* way to solve LSEs with s.p.d. system matrix.

Learning Outcomes

Principal take-home knowledge and skills from this chapter:

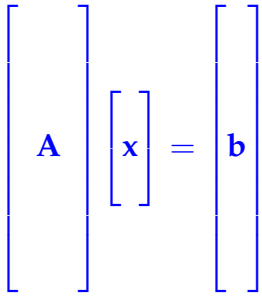
- A clear understanding of the algorithm of Gaussian elimination with and without pivoting (prerequisite knowledge from linear algebra)
- Insight into the relationship between Gaussian elimination and LU-decomposition and the algorithmic relevance of LU-decomposition
- Awareness of the asymptotic complexity of dense Gaussian elimination, LU-decomposition, and elimination for special matrices
- Familiarity with "sparse matrices": notion, data structures, initialization, benefits
- Insight into the reduced computational complexity of the direct solution of sparse linear systems of equations.

Chapter 3

DIrect Methods for Linear Least Squares Problems

In this chapter we study numerical methods for overdetermined linear systems of equations, that is, linear systems with a "tall" rectangular system matrix

$$\mathbf{x} \in \mathbb{R}^n\text{: } \text{“}\mathbf{A}\mathbf{x} = \mathbf{b}\text{”}\,, \qquad (3.0.1)$$
$$\mathbf{b} \in \mathbb{R}^m\,, \quad \mathbf{A} \in \mathbb{R}^{m,n}\,, \quad m \geq n\,.$$



In contrast to Chapter 1 we will mainly restrict ourselves to **real** linear systems in this chapter.

Note that the quotation marks in (3.0.1) indicate that this is not a well-defined problem in the sense of § 1.5.67; $\mathbf{A}\mathbf{x} = \mathbf{b}$ does no define a mapping $(\mathbf{A}, \mathbf{b}) \mapsto \mathbf{x}$, because

- such a vector $\mathbf{x} \in \mathbb{R}^n$ may not exist,
- and, even if it exists, it may not be unique.

Therefore, first we have to establish a crisp concept of that we mean by a "solution" of (3.0.1).

Contents

3.0.1 Overdetermined Linear Systems of Equations: Examples

You may think that overdetermined linear systems of equations are exotic, but this is not true. Rather they are very common in mathematical models.

Example 3.0.2 (Linear parameter estimation in 1D)

From first principles it is *known* that two physical quantities $x \in \mathbb{R}$ and $y \in \mathbb{R}$ (e.g., pressure and density of an ideal gas) are related by a linear relationship

$$y = \alpha x + \beta \quad \text{for some unknown coefficients} \quad \alpha, \beta \in \mathbb{R} . \tag{3.0.3}$$

We carry out $m \in \mathbb{N}$ measurements that yield pairs $(x_i, y_i) \in \mathbb{R}^2$, $i = 1, \ldots, m$, $m \geq 2$. If the measurements were perfect, we could expect that there exist $\alpha, \beta \in \mathbb{R}$ such that $y_i = \alpha x_i + \beta$ for all $i = 1, \ldots, m$. This is an overdetermined linear system of equations of the form (3.0.1):

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ \vdots \\ y_m \end{bmatrix} \quad \leftrightarrow \quad \mathbf{A}\mathbf{x} = \mathbf{b} , \quad \mathbf{A} \in \mathbb{R}^{m,2} , \mathbf{b} \in \mathbb{R}^m , \mathbf{x} \in \mathbb{R}^2 . \tag{3.0.4}$$

In practice inevitable ("random") *measurement errors* will thwart the solvability of (3.0.4) and for $m > 2$ the probability that a solution $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$ exists is zero, see Rem. 3.1.2.

Example 3.0.5 (Parameter estimation for a linear model)

§ 5.7.7 can be generalized to higher dimensions:

Given: *measured* data points $(\mathbf{x}_i, y_i)$, $\mathbf{x}_i \in \mathbb{R}^n$, $y_i \in \mathbb{R}$, $i = 1, \ldots, m$, $m \geq n+1$ (y_i, $\mathbf{x}_i$ affected by measurement errors).

Known: without measurement errors data would satisfy an affine linear relationship $y = \mathbf{a}^\top \mathbf{x} + \beta$, for some $\mathbf{a} \in \mathbb{R}^n$, $c \in \mathbb{R}$.

Plugging in the measured quantities gives $y_i = \mathbf{a}^\top \mathbf{x}_i + \beta$, $i = 1, \ldots, m$, a linear system of equations of the form

$$\begin{bmatrix} \mathbf{x}_1^\top & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \beta \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}, \tag{3.0.6}$$

overdetermined in case $m > n+1$.

Example 3.0.7 (Measuring the angles of a triangle [?, Sect. 5.1])

We measure the angles of a planar triangle and obtain $\widetilde{\alpha}, \widetilde{\beta}, \widetilde{\gamma}$ (in radians). In the case of perfect measurements the true angles α, β, γ would satisfy

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \widetilde{\alpha} \\ \widetilde{\beta} \\ \widetilde{\gamma} \\ \pi \end{bmatrix}. \tag{3.0.8}$$

Measurement errors will inevitably make the measured angles fail to add up to π so that (3.0.8) will not have a solution $[\alpha, \beta, \gamma]^\top$.

Then, why should we add this last equation? It turns out that solving (3.0.8) "in a suitable way" enhances cancellation of measurement errors and gives better estimates for the angles. We will not discuss this here and refer to statistics for an explanation. Ex. 2.7.23

Example 3.0.9 (Angles in a triangulation)

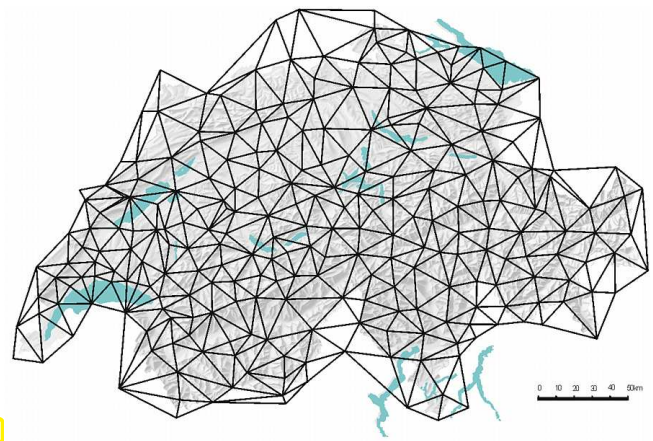


Fig. 94

In Ex. 2.7.23 we learned about the concept and data structures for planar triangulations → Def. 2.7.24. Such triangulations have been and continue to be of fundament importance for geodesy. In particular before distances could be measured accurately by means of lasers, triangulations were indispensable, because angles could already be determined with high precision. *C.F. Gauss* pioneered both the use of triangulations in geodesy and the use of the least squares method to deal with measurement errors → Wikipedia.

> Die Grundlagen seines Verfahrens hatte Gauss schon 1795 im Alter von 18 Jahren entwickelt. Basis war eine Idee von Pierre-Simon Laplace, die Beträge von Fehlern aufzusummieren, so dass sich die Fehler zu Null addieren. Gauss nahm stattdessen die Fehlerquadrate und konnte die künstliche Zusatzanforderung an die Fehler weglassen.

Gauss benutzte dann das Verfahren intensiv bei seiner Vermessung des Königreichs Hannover durch Triangulation. 1821 und 1823 erschien die zweiteilige Arbeit sowie 1826 eine Ergänzung zur *Theoria combinationis observationum erroribus minimis obnoxiae* (Theorie der den kleinsten Fehlern unterworfenen Kombination der Beobachtungen), in denen Gauss eine Begründung liefern konnte, weshalb sein Verfahren im Vergleich zu den anderen so erfolgreich war: Die Methode der kleinsten Quadrate ist in einer breiten Hinsicht optimal, also besser als andere Methoden.

We now extend Ex. 3.0.7 to planar triangulations, for which *measured* values for all internal angles are available. We obtain an overdetermined system of equations by combining the following linear relations:

1. each angle is supposed to be equal to its measured value,
2. the sum of interior angles is π for every triangle,
3. the sum of the angles at an interior node is 2π.

If the planar triangulation has N_0 interior vertices and M cells, then we end up with $4M + N_0$ equations for the $3M$ unknown angles.

Example 3.0.10 ((Relative) point locations from distances [?, Sect. 6.1])

Consider n points located on the real axis at unknown locations $x_i \in \mathbb{R}$, $i = 1, \ldots, n$. At least we know that $x_i < x_{i+1}$, $i = 1, \ldots, n-1$.

We *measure* the $m := \binom{n}{2} = \frac{1}{2}n(n-1)$ pairwise distances $d_{ij} := |x_i - x_j|$, $i, j \in \{1, \ldots, n\}$, $i \neq j$. They are connected to the point positions by the overdetermined linear system of equations

$$\begin{array}{l} x_i - x_j = d_{ij}\,, \\ 1 \le j < i \le n\,. \\ \updownarrow \\ \mathbf{A}\mathbf{x} = \mathbf{b} \end{array} \quad \leftrightarrow \quad \begin{bmatrix} -1 & 1 & 0 & \ldots & \ldots & 0 \\ -1 & 0 & 1 & 0 & & \\ \vdots & & \ddots & & & \\ \vdots & & & \ddots & & \\ -1 & \ldots & & & 0 & 1 \\ 0 & -1 & 1 & & & 0 \\ \vdots & & & & & \vdots \\ \vdots & & & & & \vdots \\ 0 & \ldots & & & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_{12} \\ d_{13} \\ \vdots \\ \vdots \\ d_{1n} \\ d_{23} \\ \vdots \\ \vdots \\ d_{n-1,n} \end{bmatrix} \tag{3.0.11}$$

Note that we can never expect a unique solution for $\mathbf{x} \in \mathbb{R}^n$, because adding a multiple of $[1, 1, \ldots, 1]^\top$ to any solution will again yield a solution, because $\mathbf{A}$ has a non-trivial kernel: $\mathcal{N}(\mathbf{A}) = [1, 1, \ldots, 1]^\top$. Non-uniqueness can be cured by setting $x_1 := 0$, thus removing one component of $\mathbf{x}$.

If the measurements were perfect, we could then find $x_2, \ldots, x_n$ from $d_{i-1,i}$, $i = 2, \ldots, n$ by solving a standard (square) linear system of equations. However, as in Ex. 3.0.7, using much more information through the overdetermined system (3.0.11) helps curb measurement errors.

3.1 Least Squares Solution Concepts

Throughout we consider the (possibly overdetermined) linear system of equations

$$\mathbf{x} \in \mathbb{R}^n: \quad \text{“}\mathbf{A}\mathbf{x} = \mathbf{b}\text{”}\,, \quad \mathbf{b} \in \mathbb{R}^m\,, \quad \mathbf{A} \in \mathbb{R}^{m,n}\,, \quad m \geq n\,. \tag{3.0.1}$$

Recall from linear algebra that $\mathbf{A}\mathbf{x} = \mathbf{b}$ has a solution, if and only if the right hand side vector $\mathbf{b}$ lies in the image (range space, → Def. 2.2.2) of the matrix $\mathbf{A}$:

$$\exists \mathbf{x} \in \mathbb{R}^n: \quad \mathbf{A}\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{b} \in \mathcal{R}(\mathbf{A})\,. \tag{3.1.1}$$

✎ Notation for important subspaces associated with a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ (→ Def. 2.2.2)

$$\begin{aligned} \text{image/range:} \quad & \mathcal{R}(\mathbf{A}) := \{\mathbf{A}\mathbf{x},\ \mathbf{x} \in \mathbb{K}^n\} \subset \mathbb{K}^m\,, \\ \text{kernel/nullspace:} \quad & \mathcal{N}(\mathbf{A}) := \{\mathbf{x} \in \mathbb{K}^n\colon \mathbf{A}\mathbf{x} = \mathbf{0}\}\,. \end{aligned}$$

Remark 3.1.2 (Consistent right hand side vectors are highly improbable)

If $\mathcal{R}(\mathbf{A}) \neq \mathbb{R}^m$, then “almost all” perturbations of $\mathbf{b}$ (e.g., due to measurement errors) will destroy $\mathbf{b} \in \mathcal{R}(\mathbf{A})$, because $\mathcal{R}(\mathbf{A})$ is a “set of measure zero” in $\mathbb{R}^m$.

3.1.1 Least Squares Solutions

Definition 3.1.3. Least squares solution

For given $\mathbf{A} \in \mathbb{K}^{m,n}$, $\mathbf{b} \in \mathbb{K}^m$ the vector $\mathbf{x} \in \mathbb{R}^n$ is a least squares solution of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, if

$$\begin{aligned} & \mathbf{x} \in \operatorname*{argmin}_{\mathbf{y} \in \mathbb{K}^n} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2\,, \\ & \Updownarrow \\ & \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 = \inf_{\mathbf{y} \in \mathbb{K}^n} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2\,. \end{aligned}$$

➡ A least squares solution is any vector $\mathbf{x}$ that minimizes the Euclidean norm of the residual $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$, see Def. 2.4.1.

We write $\mathrm{lsq}(\mathbf{A}, \mathbf{b})$ for the set of least squares solutions of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$:

$$\mathrm{lsq}(\mathbf{A}, \mathbf{b}) := \{\mathbf{x} \in \mathbb{R}^n\colon \mathbf{x} \text{ is a least squares solution of } \mathbf{A}\mathbf{x} = \mathbf{b}\} \subset \mathbb{R}^n\,. \tag{3.1.4}$$

Example 3.1.5 (linear regression → [?, Ex. 4.1])

We consider the problem of parameter estimation for a linear model from Ex. 3.0.5:

Given: *measured* data points $(\mathbf{x}_i, y_i)$, $\mathbf{x}_i \in \mathbb{R}^n$, $y_i \in \mathbb{R}$, $i = 1, \ldots, m$, $m \geq n+1$
(y_i, $\mathbf{x}_i$ affected by measurement errors).

Known: without measurement errors data would satisfy affine linear relationship

$$y = \mathbf{a}^\top \mathbf{x} + \beta \,, \tag{3.1.6}$$

from some parameters $\mathbf{a} \in \mathbb{R}^n$, $\beta \in \mathbb{R}$.

Solving the overdetermined linear system of equations in least squares sense we obtain a least squares estimate for the parameters $\mathbf{a}$ and β:

$$(\mathbf{a}, \beta) = \operatorname*{argmin}_{\mathbf{p}\in\mathbb{R}^n, \gamma\in\mathbb{R}} \sum_{i=1}^{m} |y_i - \mathbf{p}^\top \mathbf{x}_i - \gamma|^2 \tag{3.1.7}$$

In statistics, solving (3.1.7) is known as linear regression

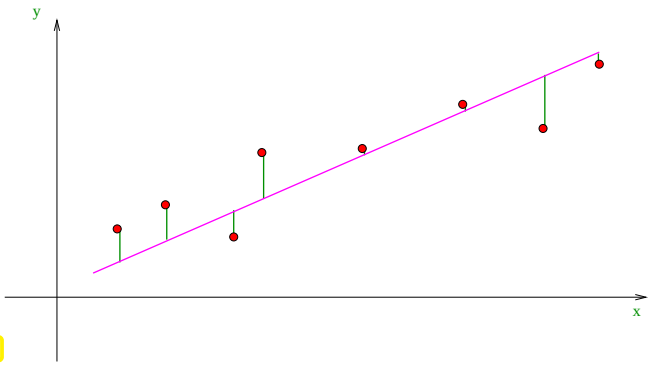


Fig. 105

linear regression for $n = 1, m = 8$: "fitting" a line to data points ▷

[**?**, Sect. 4.5]: In statistics we learn that the least squares estimate provides a maximum likelihood estimate, if the measurement errors are uniformly and independently normally distributed.

(3.1.8) The geometry of least squares problems

A geometric "proof" for the existence of least squares solutions ($\mathbb{K} = \mathbb{R}$)

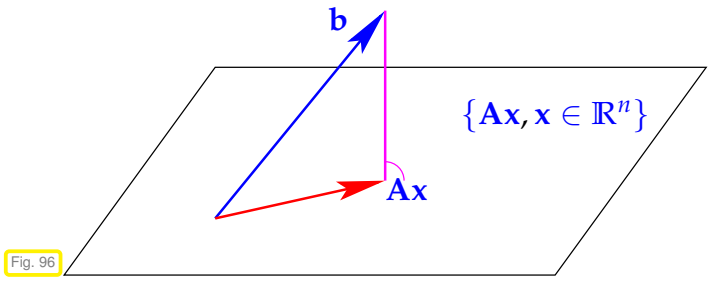


Fig. 96

◁ For a least squares solution $\mathbf{x} \in \mathbb{R}^n$ the vector $\mathbf{Ax} \in \mathbb{R}^m$ is the unique orthogonal projection of $\mathbf{b}$ onto

$$\mathcal{R}(\mathbf{A}) = \operatorname{Span}\{(\mathbf{A})_{:,1}, \ldots, (\mathbf{A})_{:,n}\} \,,$$

because the orthogonal projection provides the nearest (w.r.t. the Euclidean distance) point to $\mathbf{b}$ in the subspace (hyperplane) $\mathcal{R}(\mathbf{A})$.

From this geometric consideration we conclude that $\operatorname{lsq}(\mathbf{A}, \mathbf{b})$ is the space of solutions of $\mathbf{Ax} = \mathbf{b}^*$, where $\mathbf{b}^*$ is the orthogonal projection of $\mathbf{b}$ onto $\mathcal{R}(\mathbf{A})$. Since the set of solutions of a linear system of equations invariably is an affine space, this argument teaches that $\operatorname{lsq}(\mathbf{A}, \mathbf{b})$ is an affine subspace of $\mathbb{R}^n$!

Theorem 3.1.9. Existence of least squares solutions

For any $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$ a least squares solution of $\mathbf{Ax} = \mathbf{b}$ ($\rightarrow$ Def. 3.1.3) exists.

3.1.2 Normal Equations

Appealing to the geometric intuition gleaned from Fig. 96 we infer the orthogonality of $\mathbf{b} - A\mathbf{x}$, $\mathbf{x}$ a least squares solution of the overdetermined linear systems of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, to all columns of $\mathbf{A}$:

$$\mathbf{b} - \mathbf{A}\mathbf{x} \perp \mathcal{R}(\mathbf{A}) \quad \Leftrightarrow \quad \mathbf{b} - \mathbf{A}\mathbf{x} \perp (\mathbf{A})_{:,j}\,, \quad j = 1,\ldots,n \quad \Leftrightarrow \quad \mathbf{A}^\top(\mathbf{b} - \mathbf{A}\mathbf{x}) = 0\,.$$

Surprisingly, we have found a square linear system of equations satisfied by the least squares solution. The next theorem gives the formal statement is this discovery. It also completely characterizes $\mathrm{lsq}(\mathbf{A},\mathbf{b})$ and reveals a way to compute this set.

Theorem 3.1.10. Obtaining least squares solutions by solving normal equations

The vector $\mathbf{x} \in \mathbb{R}^n$ is a least squares solution (→ Def. 3.1.3) of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, if and only if *it solves the normal equations*

$$\boxed{\mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b}}\,. \tag{3.1.11}$$

Note that the normal equations (3.1.11) are an $n \times n$ square linear system of equations with a *symmetric positive semi-definite* coefficient matrix:

$$\begin{bmatrix} \mathbf{A}^\top \end{bmatrix} \begin{bmatrix} \mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^\top \end{bmatrix} \begin{bmatrix} \mathbf{b} \end{bmatrix},$$

$$\Leftrightarrow \quad \begin{bmatrix} \mathbf{A}^\top\mathbf{A} \end{bmatrix} \begin{bmatrix} \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^\top \end{bmatrix} \begin{bmatrix} \mathbf{b} \end{bmatrix}.$$

Proof. (of Thm. 3.1.10)

❶: We first show that a least squares solution satisfies the normal equations. Let $\mathbf{x} \in \mathbb{R}^n$ be a least squares solutions according to Def. 3.1.3. Pick an arbitrary $\mathbf{d} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ and define the function

$$\varphi_{\mathbf{d}} : \mathbb{R} \to \mathbb{R} \quad , \quad \varphi_{\mathbf{d}}(\tau) := \|\mathbf{A}(\mathbf{x} + \tau\mathbf{d}) - \mathbf{b}\|_2^2\,. \tag{3.1.12}$$

We find the equivalent expression

$$\varphi_{\mathbf{d}}(\tau) = \tau^2\mathbf{d}^\top\mathbf{A}^\top\mathbf{A}\mathbf{d} + 2\tau\mathbf{d}^\top\mathbf{A}^\top(\mathbf{A}\mathbf{x} - \mathbf{b}) + \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2\,,$$

which shows that $\tau \mapsto \varphi_{\mathbf{d}}(\tau)$ is a smooth (C^∞) function.

Moreover, since every $\mathbf{x} \in \mathrm{lsq}(\mathbf{A},\mathbf{b})$ is a minimizer of $\mathbf{y} \mapsto \|\mathbf{A}\mathbf{y}-\mathbf{b}\|_2^2$, we conclude that $\tau \mapsto \varphi_{\mathbf{d}}(\tau)$ has a global minimum in $\tau = 0$. Necessarily,

$$\left.\frac{d\varphi_{\mathbf{d}}}{d\tau}\right|_{\tau=0} = 2\mathbf{d}^\top\mathbf{A}^\top(\mathbf{A}\mathbf{x}-\mathbf{b}) = 0\,.$$

Since this holds for *any* vector $\mathbf{d} \neq 0$, we conclude (set $\mathbf{d}$ equal to all the Euclidean unit vectors in $\mathbb{R}^n$)

$$\mathbf{A}^\top(\mathbf{A}\mathbf{x}-\mathbf{b}) = \mathbf{0}\,,$$

which is equivalent to the normal equations (3.1.11).

❷: Let $\mathbf{x}$ be a solution of the normal equations. Then we find by tedious but straightforward computations

$$\begin{aligned}
&\|\mathbf{A}\mathbf{y}-\mathbf{b}\|_2^2 - \|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2^2 \\
&\quad = \mathbf{y}^\top\mathbf{A}^\top\mathbf{A}\mathbf{y} - 2\mathbf{y}^\top\mathbf{A}^\top\mathbf{b} + \mathbf{b}^\top\mathbf{b} - \mathbf{x}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x} + 2\mathbf{x}^\top\mathbf{A}^\top\mathbf{b} - \mathbf{b}^\top\mathbf{b} \\
&\overset{(3.1.11)}{=} \mathbf{y}^\top\mathbf{A}^\top\mathbf{A}\mathbf{y} - 2\mathbf{y}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x} + \mathbf{x}^\top\mathbf{A}^\top\mathbf{A}\mathbf{x} \\
&\quad = (\mathbf{y}-\mathbf{x})^\top\mathbf{A}^\top\mathbf{A}(\mathbf{y}-\mathbf{x}) = \|\mathbf{A}(\mathbf{x}-\mathbf{y})\|_2^2 \geq 0\,. \\
&\qquad\qquad \Longrightarrow \quad \|\mathbf{A}\mathbf{y}-\mathbf{b}\| \geq \|\mathbf{A}\mathbf{x}-\mathbf{b}\|\,.
\end{aligned}$$

Since this holds for *any* $\mathbf{y} \in \mathbb{R}^n$, $\mathbf{x}$ must be a global minimizer of $\mathbf{y} \mapsto \|\mathbf{A}\mathbf{y}-\mathbf{b}\|$! □

Example 3.1.13 (Normal equations for some examples from Section 3.0.1)

Given $\mathbf{A}$ and $\mathbf{b}$ it takes only elementary linear algebra operations to form the normal equations

$$\mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b}\,. \tag{3.1.11}$$

- For § 5.7.7, $\mathbf{A} \in \mathbb{R}^{m,2}$ given in (3.0.4) we obtain the normal equations linear system

$$\begin{bmatrix} x_1 & x_2 & \dots & \dots & x_m \\ 1 & 1 & \dots & \dots & 1 \end{bmatrix} \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ & \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \|\mathbf{x}\|_2^2 & \mathbf{1}^\top\mathbf{x} \\ \mathbf{1}^\top\mathbf{x} & m \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \mathbf{x}^\top\mathbf{y} \\ \mathbf{1}^\top\mathbf{y} \end{bmatrix},$$

  with $\mathbf{1} = [1,\dots,1]^\top$.

- In the case of Ex. 3.0.5 and the overdetermined $m \times (n+1)$ linear system (3.0.6), the normal equations read

$$\begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \dots & \dots & \mathbf{x}_m \\ 1 & 1 & \dots & \dots & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1^\top & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^\top & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \beta \end{bmatrix} = \begin{bmatrix} \mathbf{X}\mathbf{X}^\top & \mathbf{X}\mathbf{1} \\ \mathbf{1}^\top\mathbf{X}^\top & m \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ \beta \end{bmatrix} = \begin{bmatrix} \mathbf{X}\mathbf{y} \\ \mathbf{1}^\top\mathbf{y} \end{bmatrix},$$

  where $\mathbf{X} = [\mathbf{x}_1,\dots,\mathbf{x}_m] \in \mathbb{R}^{n,m}$.

Remark 3.1.14 (Normal equations from gradient)

We consider the function

$$J : \mathbb{R}^n \to \mathbb{R} \quad , \quad J(\mathbf{y}) := \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2^2 \, . \tag{3.1.15}$$

As above, using elementary identities for the Euclidean inner product on $\mathbb{R}^m$, J can be recast as

$$\begin{aligned} J(\mathbf{y}) &= \mathbf{y}^\top \mathbf{A}^\top \mathbf{A}\mathbf{y} - 2\mathbf{b}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{b} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} (\mathbf{A}^\top \mathbf{A})_{ij} y_i y_j - 2\sum_{i=1}^{m}\sum_{j=1}^{n} b_i (\mathbf{A})_{ij} y_j + \sum_{i=1}^{m} b_i^2 \, . \end{aligned}$$

Obviously, J is a multivariate polynomial in $y_1, \ldots, y_n$. As such, J is an infinitely differentiable function, $J \in C^\infty(\mathbb{R}^n, \mathbb{R})$, see [**?**, Bsp. 7.1.5].

According to [**?**, Satz 7.5.3] $\mathbf{x} \in \mathbb{R}^n$ is a minimizer of J only if

$$\mathbf{grad}\, J(\mathbf{x}) = 2\mathbf{A}^\top \mathbf{A}\mathbf{x} - 2\mathbf{A}^\top \mathbf{b} = \mathbf{0} \, . \tag{3.1.16}$$

This formula for the gradient of J can easily be confirmed by computing the partial derivatives $\frac{\partial J}{\partial y_i}$ from the above explicit formula. Observe that (3.1.16) is equivalent to the normal equations (3.1.11).

(3.1.17) The linear least squares problem (→ § 1.5.67)

Thm. 3.1.10 together with Thm. 3.1.9 already confirms that the normal equations will always have a solution and that $\mathrm{lsq}(\mathbf{A}, \mathbf{b})$ is a subspace of $\mathbb{R}^n$ parallel to $\mathcal{N}(\mathbf{A}^\top \mathbf{A})$. The next theorem gives even more detailed information.

Theorem 3.1.18. Kernel and range of $\mathbf{A}^\top \mathbf{A}$

For $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, holds

$$\mathcal{N}(\mathbf{A}^\top \mathbf{A}) = \mathcal{N}(\mathbf{A}) \, , \tag{3.1.19}$$

$$\mathcal{R}(\mathbf{A}^\top \mathbf{A}) = \mathcal{R}(\mathbf{A}^\top) \, . \tag{3.1.20}$$

For the proof we need an basic result from linear algebra:

Lemma 3.1.21. Kernel and range of (Hermitian) transposed matrices

For any matrix $\mathbf{A} \in \mathbb{K}^{m,n}$ holds

$$\mathcal{N}(\mathbf{A}) = \mathcal{R}(\mathbf{A}^{\mathrm{H}})^\perp \quad , \quad \mathcal{N}(\mathbf{A})^\perp = \mathcal{R}(\mathbf{A}^{\mathrm{H}}) \, .$$

✎ Notation: Orthogonal complement of a subspace $V \subset \mathbb{K}^k$:

$$V^\perp := \{\mathbf{x} \in \mathbb{K}^k : \mathbf{x}^{\mathrm{H}} \mathbf{y} = 0 \; \forall \mathbf{y} \in V\} \, .$$

Proof. (of Thm. 3.1.18)
❶: We first show (3.1.19)

$$\mathbf{z} \in \mathcal{N}(\mathbf{A}^\top \mathbf{A}) \Leftrightarrow \mathbf{A}^\top \mathbf{A}\mathbf{z} = \mathbf{0} \Rightarrow \mathbf{z}^\top \mathbf{A}^\top \mathbf{A}\mathbf{z} = \|\mathbf{A}\mathbf{z}\|_2^2 = 0 \Leftrightarrow \mathbf{A}\mathbf{z} = \mathbf{0} \, ,$$

$$\mathbf{Az} = \mathbf{0} \Rightarrow \mathbf{A}^\top\mathbf{Az} = \mathbf{0} \Leftrightarrow \mathbf{z} \in \mathcal{N}(\mathbf{A}^\top\mathbf{A}) \,.$$

❷: The relationship (3.1.20) follows from (3.1.19) and Lemma 3.1.21:

$$\mathcal{R}(\mathbf{A}^\top) \overset{\text{Lemma 3.1.21}}{=} \mathcal{N}(\mathbf{A})^\perp \overset{(3.1.19)}{=} \mathcal{N}(\mathbf{A}^\top\mathbf{A})^\perp \overset{\text{Lemma 3.1.21}}{=} \mathcal{R}(\mathbf{A}^\top\mathbf{A}) \,.$$

□

Corollary 3.1.22. Uniqueness of least squares solutions

If $m \geq n$ and $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$, then the linear system of equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, has a unique least squares solution (→ 3.1.3)

$$\mathbf{x} = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b} \,, \tag{3.1.23}$$

that can be obtained by solving the normal equations (3.1.11).

Note that $\mathbf{A}^\top\mathbf{A}$ is symmetric positive definite (→ Def. 1.1.8), if $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$.

Remark 3.1.24 (Full-rank condition (→ Def. 2.2.3))

For a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ with $m \geq n$ is equivalent

$$\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\} \iff \operatorname{rank}(\mathbf{A}) = n \,. \tag{3.1.25}$$

Hence the assumption $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$ of Cor. 3.1.22 is also called a full-rank condition, because the rank of $\mathbf{A}$ is maximal.

Example 3.1.26 (Meaning of full-rank condition for linear models)

We revisit the parameter estimation problem for a linear model.

- For § 5.7.7, $\mathbf{A} \in \mathbb{R}^{m,2}$ given in (3.0.4) it is easy to see

$$\operatorname{rank}\left(\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ & \\ \vdots & \vdots \\ x_m & 1 \end{bmatrix}\right) = 2 \quad \Leftrightarrow \quad \exists i,j \in \{1,\ldots,m\}\colon x_i \neq x_j \,,$$

  that is, the manifest condition, that the points do not lie on a vertical line.

- In the case of Ex. 3.0.5 and the overdetermined $m \times (n+1)$ linear system (3.0.6), we find

$$\operatorname{rank}\left(\begin{bmatrix} \mathbf{x}_1^\top & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^\top & 1 \end{bmatrix}\right) = n+1 \quad \Leftrightarrow$$

  There is a subset of $n+1$ points $\mathbf{x}_{i_1},\ldots\mathbf{x}_{i_{n+1}}$ such that $\{\mathbf{0}, \mathbf{x}_{i_1},\ldots\mathbf{x}_{i_{n+1}}\}$ spans a non-degenerate $n+1$-simplex.

Remark 3.1.27 (Rank defect in linear least squares problems)

In case the system matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, of an overdetermined linear system arising from a mathematical fails to have full rank, it hints at inadequate modelling:

In this case parameters are redundant, because different sets of parameters yield the same output quantities: the parameters are not "observable".

Remark 3.1.28 (Hesse matrix of least squares functional)

For the least squares functional

$$J : \mathbb{R}^n \to \mathbb{R} \quad , \quad J(\boldsymbol{y}) := \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2^2 \, . \tag{3.1.15}$$

and its explicit form as polynomial in the vector components y_j we find the Hessian ($\to$ Def. 8.4.11, [**?**, Satz 7.5.3]) of J:

$$\mathrm{H}\, J(\mathbf{y}) = 2\mathbf{A}^\top \mathbf{A} \, . \tag{3.1.29}$$

Thm. 3.1.18 implies that $\mathbf{A}^\top \mathbf{A}$ is positive definite ($\to$ Def. 1.1.8) if and only if $\mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$.

Therefore, by [**?**, Satz 7.5.3], under the full-rank condition J has a positive definite Hessian everywhere, and a minimum at every stationary point of its gradient, that is, at every solution of the normal equations.

Remark 3.1.30 (Convex least squares functional)

Another result from analysis tells us that real-valued C^1-functions on $\mathbb{R}^n$ whose Hessian has positive eigenvalues uniformly bounded away from zero are strictly convex. Hence, if $\mathbf{A}$ has full rank, the least squares functional J from (3.1.15) is a strictly convex function.

Visualization of a least squares functional $J : \mathbb{R}^2 \to \mathbb{R}$ for $n = 2$ ▷

Under the full-rank condition the graph of J is a paraboloid with $J(\mathbf{y}) \to \infty$ for $\|\mathbf{y}\| \to \infty$

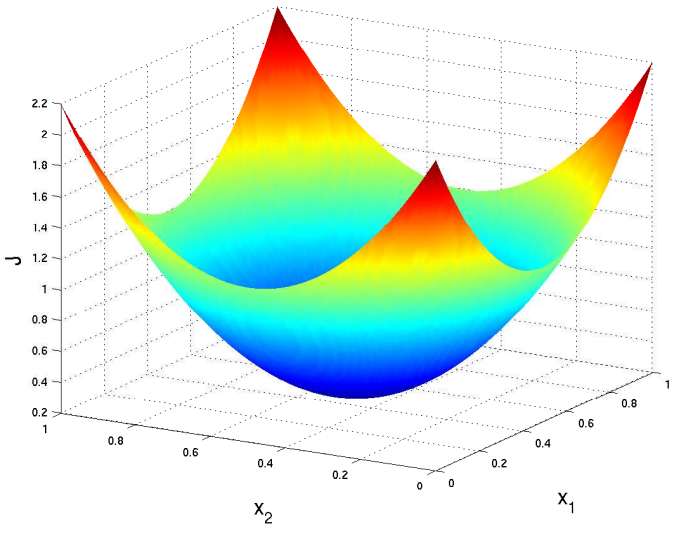


Fig. 97

Now we are in a position to state precisely what we mean by solving an overdetermined ($m \geq n$!) linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, provided that $\mathbf{A}$ has full (maximal) rank, *cf.* (3.1.25).

(Full rank linear) least squares problem: [**?**, Sect. 4.2]

$$\begin{aligned} &\text{given:} && \mathbf{A} \in \mathbb{R}^{m,n}, \quad m,n \in \mathbb{N}, m \geq n, \operatorname{rank}(\mathbf{A}) = n, \quad \mathbf{b} \in \mathbb{R}^m, \\ &\text{find:} && \mathbf{x} \in \mathbb{R}^n \text{ such that } \quad \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 = \inf\{\|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 \colon \mathbf{y} \in \mathbb{R}^n\} \\ &&& \qquad\qquad \Updownarrow \\ &&& \mathbf{x} = \operatorname*{argmin}_{\mathbf{y} \in \mathbb{R}^n} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 \end{aligned} \tag{3.1.31}$$

✎ A sloppy notation for the minimization problem (3.1.31) is $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \to \min$

3.1.3 Moore-Penrose Pseudoinverse

As we have seen in Ex. 3.0.10, there can be many least squares solutions of $\mathbf{A}\mathbf{x} = \mathbf{b}$, in case $\mathcal{N}(\mathbf{A}) \neq \{\mathbf{0}\}$. We can impose another condition to single out a unique element of $\operatorname{lsq}(\mathbf{A}, \mathbf{b})$:

Definition 3.1.32. Generalized solution of a lineasr system of equations

The generalized solution $\mathbf{x}^\dagger \in \mathbb{R}^n$ of a linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, is defined as

$$\mathbf{x}^\dagger := \operatorname{argmin}\{\|\mathbf{x}\|_2 \colon \mathbf{x} \in \operatorname{lsq}(\mathbf{A}, \mathbf{b})\} \,. \tag{3.1.33}$$

➡ The generalized solution is the least squares solution with minimal norm.

(3.1.34) Reduced normal equations

Elementary geometry teaches that the minimal norm element of an affine subspace L (a plane) in Euclidean space is the orthogonal projection of $\mathbf{0}$ onto L.

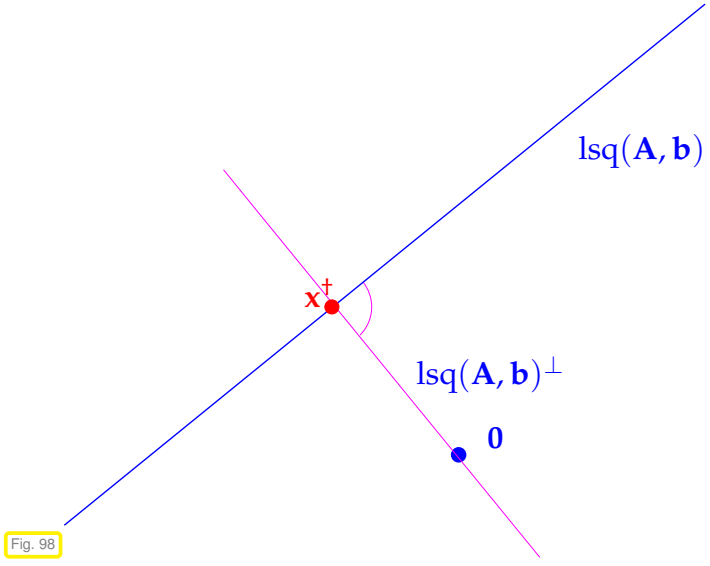


Fig. 98

◁ visualization:

The minimal norm element $\mathbf{x}^\dagger$ of the affine space $\operatorname{lsq}(\mathbf{A}, \mathbf{b}) \subset \mathbb{R}^n$ belongs to the subspace of $\mathbb{R}^n$ that is *orthogonal to* $\operatorname{lsq}(\mathbf{A}, \mathbf{b})$.

Since the space of least squares solutions of $\mathbf{A}\mathbf{x} = \mathbf{b}$ is an affine subspace parallel to $\mathcal{N}(\mathbf{A})$

$$\mathrm{lsq}(\mathbf{A},\mathbf{b}) = \mathbf{x}^0 + \mathcal{N}(\mathbf{A}) \;, \quad \mathbf{x}^0 \text{ solves normal equations,} \tag{3.1.35}$$

the generalized solution $\mathbf{x}^*$ of $\mathbf{A}\mathbf{x} = \mathbf{b}$ is contained in $\mathcal{N}(\mathbf{A})^\perp$. Therefore, given a basis $\{\mathbf{v}_1,\ldots,\mathbf{v}_k\} \subset \mathbb{R}^n$ of $\mathcal{N}(\mathbf{A})^\perp$, $k := \dim \mathcal{N}(\mathbf{A})$, we can find $\mathbf{y} \in \mathbb{R}^k$ such that $\mathbf{x}^* = \mathbf{V}\mathbf{y}$, $\mathbf{V} := [\mathbf{v}_1,\ldots,\mathbf{v}_k] \in \mathbb{R}^{n,k}$. Plugging this representation into the normal equations and multiplying with $\mathbf{V}^\top$ yields the reduced normal equations

$$\mathbf{V}^\top\mathbf{A}^\top\mathbf{A}\mathbf{V}\,\mathbf{y} = \mathbf{V}^\top\mathbf{A}^\top\mathbf{b} \tag{3.1.36}$$

$$\Updownarrow$$

$$\left[\mathbf{V}^\top\right]\left[\mathbf{A}^\top\right]\left[\mathbf{A}\right]\left[\mathbf{V}\right]\left[\mathbf{y}\right] = \left[\mathbf{V}^\top\right]\left[\mathbf{A}^\top\right]\left[\mathbf{b}\right] .$$

The very construction of $\mathbf{V}$ ensures $\mathcal{N}(\mathbf{A}\mathbf{V}) = \{\mathbf{0}\}$ so that, by Thm. 3.1.18 the $k \times k$ linear system of equations (3.1.36) has a unique solution. The next theorem summarizes our insights:

Theorem 3.1.37. Formula for generalized solution

Given $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, the generalized solution $\mathbf{x}^\dagger$ of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ is given by

$$\mathbf{x}^\dagger = \mathbf{V}(\mathbf{V}^\top\mathbf{A}^\top\mathbf{A}\mathbf{V})^{-1}(\mathbf{V}^\top\mathbf{A}^\top\mathbf{b}) \;,$$

where $\mathbf{V}$ is any *matrix whose columns form a basis of $\mathcal{N}(\mathbf{A})^\perp$.*

Terminology: The matrix $\mathbf{V}(\mathbf{V}^\top\mathbf{A}^\top\mathbf{A}\mathbf{V})^{-1}\mathbf{V}^\top$ is called the Moore-Penrose pseudoinverse of $\mathbf{A}$.

✎ notation: $\mathbf{A}^\dagger \in \mathbb{R}^{n,m} \hat{=}$ pseudoinverse of $\mathbf{A} \in \mathbb{R}^{m,n}$

Note that the Moore-Penrose pseudoinverse does not depend on the choice of $\mathbf{V}$.

Armed with the concept of generalized solution and the knowledge about its existence and uniqueness we can state the most generali linear least squares problem:

(General linear) least squares problem:

given: $\mathbf{A} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$, $\mathbf{b} \in \mathbb{R}^m$,
find: $\mathbf{x} \in \mathbb{R}^n$ such that

(i) $\|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2 = \inf\{\|\mathbf{A}\mathbf{y}-\mathbf{b}\|_2 : \mathbf{y} \in \mathbb{R}^n\}$,
(ii) $\|\mathbf{x}\|_2$ is minimal under the condition (i).

(3.1.38)

3.1.4 Sensitivity of Least Squares Problem

Consider the full-rank linear least squares problem introduced in (3.1.31):

$$\succ \quad \text{data} \quad (\mathbf{A},\mathbf{b}) \in \mathbb{R}^{m,n} \times \mathbb{R}^m \quad , \quad \text{result } \mathbf{x} = \operatorname{argmin}_{\mathbf{y}\in\mathbb{R}^n} \|\mathbf{A}\mathbf{y}-\mathbf{b}\|_2 \in \mathbb{R}^n$$

On data space and result space we use the Euclidean norm (2-norm $\|\cdot\|_2$) and the associated matrix norm, see § 1.5.69.

Recall Section 2.2.2, where we discussed the sensitivity of solutions of square linear systems, that is, the impact of perturbations in the problem data on the result. Now we study how (small) changes in $\mathbf{A}$ and $\mathbf{b}$ affect the unique ($\to$ Cor. 3.1.22) least solution $\mathbf{x}$ of $\mathbf{A}\mathbf{x} = \mathbf{b}$ in the case of $\mathbf{A}$ with full rank ($\Leftrightarrow \mathcal{N}(\mathbf{A}) = \{\mathbf{0}\}$)

Note: If the matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, has full rank, then there is a $c > 0$ such that $\mathbf{A} + \Delta\mathbf{A}$ still has full rank for all $\Delta\mathbf{A} \in \mathbb{R}^{m,n}$ with $\|\Delta\mathbf{A}\|_2 < c$. Hence, "sufficiently small" perturbations will not destroy the full-rank property of $\mathbf{A}$. This is a generalization of the Perturbation Lemma 2.2.11.

For square linear systems the condition number of the system matrix ($\to$ Def. 2.2.12) provided the key gauge of sensitivity. To express the sensitivity of linear least squares problems we also generalize this concept:

Definition 3.1.39. Generalized condition number of a matrix

Given $\mathbf{A} \in \mathbb{K}^{m,n}$, $m \geq n$, $\operatorname{rank}(\mathbf{A}) = n$, we define its generalized (Euclidean) condition number as

$$\operatorname{cond}_2(\mathbf{A}) := \sqrt{\frac{\lambda_{\max}(\mathbf{A}^H\mathbf{A})}{\lambda_{\min}(\mathbf{A}^H\mathbf{A})}} \, .$$

✎ notation: $\lambda_{\min}(\mathbf{A}) \mathrel{\hat{=}}$ smallest (in modulus) eigenvalue of matrix $\mathbf{A}$
$\lambda_{\max}(\mathbf{A}) \mathrel{\hat{=}}$ largest (in modulus) eigenvalue of matrix $\mathbf{A}$

For a square regular matrix this agrees with its condition number according to Def. 2.2.12, which follows from Cor. 1.5.82.

Theorem 3.1.40. Sensitivity of full-rank linear least squares problem

For $m \geq n$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\operatorname{rank}(\mathbf{A}) = n$, let $\mathbf{x} \in \mathbb{R}^n$ be the solution of the least squares problem $\|\mathbf{A}\mathbf{x}-\mathbf{b}\| \to \min$ and $\widehat{\mathbf{x}}$ the solution of the perturbed least squares problem $\|(\mathbf{A}+\Delta\mathbf{A})\widehat{\mathbf{x}}-\mathbf{b}\| \to \min$. Then

$$\frac{\|\mathbf{x}-\widehat{\mathbf{x}}\|_2}{\|\mathbf{x}\|_2} \overset{\cdot}{\leq} \left(2\operatorname{cond}_2(\mathbf{A}) + \operatorname{cond}_2^2(\mathbf{A})\frac{\|\mathbf{r}\|_2}{\|\mathbf{A}\|_2\|\mathbf{x}\|_2}\right)\frac{\|\Delta\mathbf{A}\|_2}{\|\mathbf{A}\|_2}$$

holds, where $\mathbf{r} = \mathbf{A}\mathbf{x}-\mathbf{b}$ is the residual.

This means: if $\|\mathbf{r}\|_2 \ll 1$ ➤ condition of the least squares problem $\approx \operatorname{cond}_2(\mathbf{A})$
if $\|\mathbf{r}\|_2$ "large" ➤ condition of the least squares problem $\approx \operatorname{cond}_2^2(\mathbf{A})$

For instance, in a linear parameter estimation problem ($\to$ Ex. 3.0.5) a small residual will be the consequence of small measurement errors.

3.2 Normal Equation Methods [?, Sect. 4.2], [?, Ch. 11]

Given $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\mathrm{rank}(\mathbf{A}) = n$, $\mathbf{b} \in \mathbb{R}^m$, we introduce a first practical numerical method to determine the unique least squares solution ($\rightarrow$ Def. 3.1.3) of the overdetermined linear system of equations $\mathbf{Ax} = \mathbf{b}$.

In fact, Cor. 3.1.22 suggests a simple algorithm for solving linear least squares problems of the form (3.1.31) satisfying the full (maximal) rank condition $\mathrm{rank}(\mathbf{A}) = n$: it boils down to solving the normal equations (3.1.11):

Algorithm: Normal equation method to solve full-rank least squares problem $\mathbf{Ax} = \mathbf{b}$

❶ Compute regular matrix $\mathbf{C} := \mathbf{A}^\top \mathbf{A} \in \mathbb{R}^{n,n}$.
❷ Compute right hand side vector $\mathbf{c} := \mathbf{A}^\top \mathbf{b}$.
❸ Solve s.p.d. ($\rightarrow$ Def. 1.1.8) linear system of equations: $\mathbf{Cx} = \mathbf{c}$ $\rightarrow$ § 2.8.13

This can be done in EIGEN in a single line of code:

C++11 code 3.2.1: Solving a linear least squares probel via normal equations

```
//!  Solving the overdetermined linear system of equations
//!  Ax = b by solving normal equations (3.1.11)
//!  The least squares solution is returned by value
VectorXd normeqsolve(const MatrixXd &A,const VectorXd &b) {
  if (b.size() != A.rows()) throw runtime_error("Dimension mismatch");
  // Cholesky solver
  VectorXd x = (A.transpose()*A).llt().solve(A.transpose()*b);
  return x;
}
```

By Thm. 2.8.11, for the s.p.d. matrix $\mathbf{A}^\top \mathbf{A}$ Gaussian elimination remains stable even without pivoting. This is taken into account by requesting the Cholesky decomposition of $\mathbf{A}^\top \mathbf{A}$ by calling the method `llt()`.

(3.2.2) Asymptotic complexity of normal equation method

The problem size parameters for the linear least squares problem (3.1.31) are the matrix dimensions $m, n \in \mathbb{N}$, where n small & fixed, $n \ll m$, is common.

In Section 1.4.2 and Thm. 2.5.2 we discussed the asymptotic complexity of the operations involved in step ❶-❸ of the normal equation method:

step ❶: cost $O(mn^2)$
step ❷: cost $O(nm)$
step ❸: cost $O(n^3)$

$\Rightarrow$ cost $O(n^2 m + n^3)$ for $m, n \to \infty$.

Note that for small fixed n, $n \ll m$, $m \to \infty$ the computational effort scales linearly with m.

Remark 3.2.3 (Conditioning of normal equations [?, pp. 128])

Normal equation method *vulnerable to instability*; immediate from Def. 3.1.39:

$$\mathrm{cond}_2(\mathbf{A}^{\mathrm{H}}\mathbf{A}) = \mathrm{cond}_2(\mathbf{A})^2 .$$

Recall from Thm. 2.2.10: $\mathrm{cond}_2(\mathbf{A}^{\mathrm{H}}\mathbf{A})$ governs amplification of (roundoff) errors in $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{A}^\top\mathbf{b}$ when solving normal equations (3.1.11).

➢ For fairly ill-conditioned $\mathbf{A}$ using the normal equations (3.1.11) to solve the linear least squares problem from Def. 3.1.3 numerically may run the risk of huge amplification of roundoff errors incurred during the computation of the right hand side $\mathbf{A}^{\mathrm{H}}\mathbf{b}$: potential instability ($\rightarrow$ Def. 1.5.85) of normal equation approach.

Example 3.2.4 (Roundoff effects in normal equations $\rightarrow$ [?, Ex. 4.12])

In this example we witness loss of information in the computation of $\mathbf{A}^{\mathrm{H}}\mathbf{A}$.

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \delta & 0 \\ 0 & \delta \end{bmatrix} \quad \Rightarrow \quad \mathbf{A}^\top\mathbf{A} = \begin{bmatrix} 1+\delta^2 & 1 \\ 1 & 1+\delta^2 \end{bmatrix}$$

Exp. 1.5.35: If $\delta \approx \sqrt{\mathrm{EPS}}$, then $1+\delta^2 \doteq 1$ in $\mathbb{M}$. Hence the computed $\mathbf{A}^\top\mathbf{A}$ will fail to be regular, though $\mathrm{rank}(\mathbf{A}) = 2$, $\mathrm{cond}_2(\mathbf{A}) \approx \sqrt{\mathrm{EPS}}$.

C++-code 3.2.5:

```
2  int main() {
3    MatrixXd A(3,2);
4    // Inquire about machine precision → Ex. 1.5.33
5    double eps = std::numeric_limits<double>::epsilon();
6    // « initialization of matrix → § 1.2.13
7    A << 1, 1, sqrt(eps), 0, 0, sqrt(eps);
8    // Output rank of A^TA
9    std::cout << "Rank of A: " << A.fullPivLu().rank() << std::endl
10             << "Rank of A^TA: "
11             << (A.transpose() * A).fullPivLu().rank() << std::endl;
12   return 0;
13 }
```

Output:

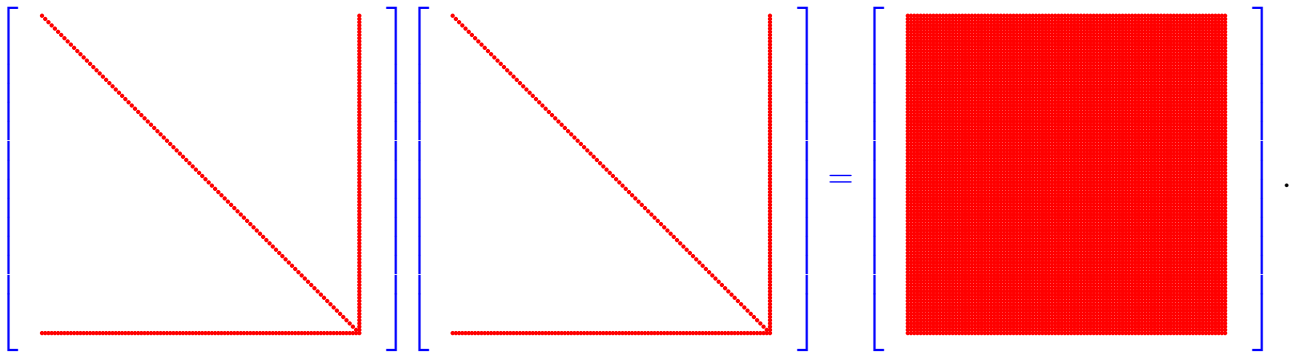
```
1 Rank of A: 2
2 Rank of A^T*A: 1
```

Remark 3.2.6 (Loss of sparsity when forming normal equations)

Another *reason not to compute* $\mathbf{A}^{\mathrm{H}}\mathbf{A}$, when both m, n large:

$$\mathbf{A} \text{ sparse} \quad \not\Rightarrow \quad \mathbf{A}^\top\mathbf{A} \text{ sparse}$$

Example from Rem. 1.3.5: "Arrow matrices"



Consequences for normal equation method, if both m, n large:

- Potential memory overflow, when computing $\mathbf{A}^\top\mathbf{A}$
- Squanders possibility to use efficient sparse direct elimination techniques, see Section 2.7.5

This situation is faced in Ex. 3.0.10, Ex. 3.0.9.

Remark 3.2.7 (Extended normal equations)

A way to avoid the computation of $\mathbf{A}^{\mathrm{H}}\mathbf{A}$:

Extend normal equations (3.1.11): introduce residual $\mathbf{r} := \mathbf{A}\mathbf{x} - \mathbf{b}$ as new unknown:

$$\mathbf{A}^{\mathrm{H}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathrm{H}}\mathbf{b} \quad \Leftrightarrow \quad \mathbf{B}\begin{bmatrix}\mathbf{r}\\ \mathbf{x}\end{bmatrix} := \begin{bmatrix}-\mathbf{I} & \mathbf{A}\\ \mathbf{A}^{\mathrm{H}} & 0\end{bmatrix}\begin{bmatrix}\mathbf{r}\\ \mathbf{x}\end{bmatrix} = \begin{bmatrix}\mathbf{b}\\ 0\end{bmatrix}. \tag{3.2.8}$$

The benefit of using (3.2.8) instead of the standard normal equations (3.1.11) is that sparsity is preserved. However, the conditioning of the system matrix in (3.2.8) is not better than that of $\mathbf{A}^\top\mathbf{A}$.

A more general substitution $\mathbf{r} := \alpha^{-1}(\mathbf{A}\mathbf{x} - \mathbf{b})$ with $\alpha > 0$ may improve the conditioning for suitably chosen parameter α

$$\mathbf{A}^{\mathrm{H}}\mathbf{A}\mathbf{x} = \mathbf{A}^{\mathrm{H}}\mathbf{b} \quad \Leftrightarrow \quad \mathbf{B}_\alpha\begin{bmatrix}\mathbf{r}\\ \mathbf{x}\end{bmatrix} := \begin{bmatrix}-\alpha\mathbf{I} & \mathbf{A}\\ \mathbf{A}^{\mathrm{H}} & 0\end{bmatrix}\begin{bmatrix}\mathbf{r}\\ \mathbf{x}\end{bmatrix} = \begin{bmatrix}\mathbf{b}\\ 0\end{bmatrix}. \tag{3.2.9}$$

For $m, n \gg 1$, $\mathbf{A}$ sparse, both (3.2.8) and (3.2.9) lead to large sparse linear systems of equations, amenable to sparse direct elimination techniques, see Section 2.7.5.

Example 3.2.10 (Conditioning of the extended normal equations)

In this example we explore empirically how the Euclidean condition number of the extended normal equations (3.2.9) is influenced by the coice of α
Consider (3.2.8), (3.2.9) for

$$\mathbf{A} = \begin{bmatrix} 1+\epsilon & 1 \\ 1-\epsilon & 1 \\ \epsilon & \epsilon \end{bmatrix} .$$

Plot of different condition numbers in dependence on ϵ
(Here $\alpha = \epsilon\|\mathbf{A}\|_2/\sqrt{2}$)

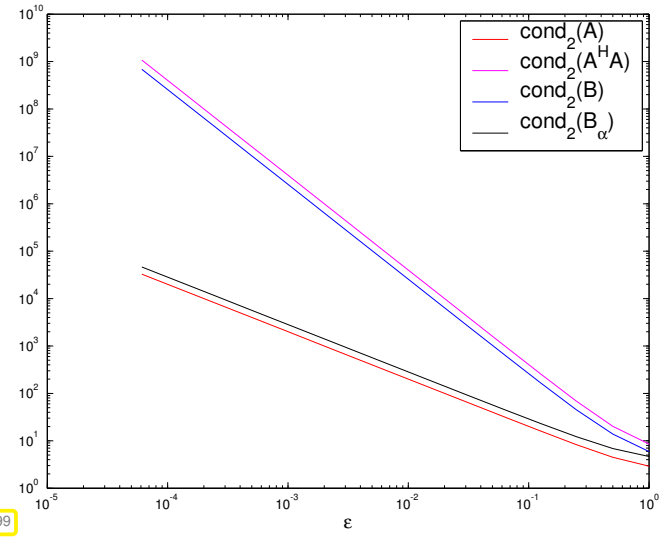


Fig. 99

3.3 Orthogonal Transformation Methods [?, Sect. 4.4.2]

3.3.1 Transformation Idea

We consider the full-rank linear least squares problem (3.1.31)

$$\text{given} \quad \mathbf{A} \in \mathbb{R}^{m,n} , \mathbf{b} \in \mathbb{R}^m \quad \text{find} \quad \mathbf{x} = \operatorname*{argmin}_{\mathbf{y}\in\mathbb{R}^n} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 . \tag{3.1.31}$$

Setting: $m \geq n$ and $\mathbf{A}$ has full (maximum) rank: $\operatorname{rank}(\mathbf{A}) = n$.

(3.3.1) Generalizing the policy underlying Gaussian elimination

Recall the rationale behind Gaussian elimination (→ Section 2.3, Ex. 2.3.1)

➥ By row transformations convert LSE $\mathbf{A}\mathbf{x} = \mathbf{b}$ to *equivalent* (in terms of set of solutions) LSE $\mathbf{U}\mathbf{x} = \widetilde{\mathbf{b}}$, which is easier to solve because it has triangular form.

How to adapt this policy to linear least squares problem (3.1.31) **?**

Two questions: ❶ What linear least squares problems are "easy to solve" ?

❷ How can we arrive at them by *equivalent transformations* of (3.1.31) ?

Here we call two overdetermined linear systems $\mathbf{A}\mathbf{x} = \mathbf{b}$ and $\widetilde{\mathbf{A}}\mathbf{x} = \widetilde{\mathbf{b}}$ equivalent in the sense of (3.1.31), if both have the same set of least squares solutions: $\operatorname{lsq}(\mathbf{A},\mathbf{b}) = \operatorname{lsq}(\widetilde{\mathbf{A}},\widetilde{\mathbf{b}})$, see (3.1.4).

(3.3.2) Triangular linear least squares problems

The answer to question ❶ is the same as for LSE:

Linear least squares problems (3.1.31) with upper *triangular* $\mathbf{A}$ are easy to solve!

$$\left\| \begin{bmatrix} \mathbf{R} \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} b_1 \\ \vdots \\ \\ \\ \vdots \\ b_m \end{bmatrix} \right\|_2 \to \min \overset{(*)}{\Longrightarrow} \mathbf{x} = \begin{bmatrix} \mathbf{R} \end{bmatrix}^{-1} \begin{bmatrix} b_1 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{bmatrix}$$

$\mathbf{x} \mathrel{\hat{=}}$ least squares solution

How can we draw the conclusion $(*)$? Obviously, the components $n+1, \ldots, m$ of the vector inside the norm are fixed and do not depend on $\mathbf{x}$. All we can do is to make the first components $1, \ldots, n$ vanish, by choosing a suitable $\mathbf{x}$, see [**?**, Thm. 4.13]. Obviously, $\mathbf{x} = \mathbf{R}^{-1}(\mathbf{b})_{1:n}$ accomplishes this.

Note: since $\mathbf{A}$ has full rank n, the upper triangular part $\mathbf{R} \in \mathbb{R}^{n,n}$ of $\mathbf{A}$ is regular!

Answer to question ❷:

Idea: If we have a (transformation) matrix $\mathbf{T} \in \mathbb{R}^{m,m}$ satisfying

$$\|\mathbf{T}\mathbf{y}\|_2 = \|\mathbf{y}\|_2 \quad \forall \mathbf{y} \in \mathbb{R}^m \,, \tag{3.3.3}$$

$$\text{then} \quad \operatorname*{argmin}_{\mathbf{y}\in\mathbb{R}^n} \|\mathbf{A}\mathbf{y} - \mathbf{b}\|_2 = \operatorname*{argmin}_{\mathbf{y}\in\mathbb{R}^n} \left\|\widetilde{\mathbf{A}}\mathbf{y} - \widetilde{\mathbf{b}}\right\|_2 ,$$

where $\widetilde{\mathbf{A}} = \mathbf{T}\mathbf{A}$ and $\widetilde{\mathbf{b}} = \mathbf{T}\mathbf{b}$.

The next section will characterize the class of eligible transformation matrices $\mathbf{T}$.

3.3.2 Orthogonal/Unitary Matrices

Definition 3.3.4. Unitary and orthogonal matrices → **[?, Sect. 2.8]**

- $\mathbf{Q} \in \mathbb{K}^{n,n}$, $n \in \mathbb{N}$, is unitary, if $\mathbf{Q}^{-1} = \mathbf{Q}^H$.
- $\mathbf{Q} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, is orthogonal, if $\mathbf{Q}^{-1} = \mathbf{Q}^T$.

Theorem 3.3.5. Preservation of Euclidean norm

A matrix is unitary/orthogonal, if and only if the associated linear mapping preserves the 2-norm:

$$\mathbf{Q} \in \mathbb{K}^{n,n} \quad \textit{unitary} \quad \Leftrightarrow \quad \|\mathbf{Q}\mathbf{x}\|_2 = \|\mathbf{x}\|_2 \quad \forall \mathbf{x} \in \mathbb{K}^n \,.$$

From Thm. 3.3.5 we immediately conclude that, if a matrix $\mathbf{Q} \in \mathbb{K}^{n,n}$ is unitary/orthogonal, then

- all rows/columns (regarded as vectors $\in \mathbb{K}^n$) have Euclidean norm $= 1$,
- all rows/columns are pairwise orthogonal (w.r.t. Euclidean inner product),
- $|\det \mathbf{Q}| = 1$, $\|\mathbf{Q}\|_2 = 1$, and all eigenvalues $\in \{z \in \mathbb{C}: |z| = 1\}$.
- $\|\mathbf{QA}\|_2 = \|\mathbf{A}\|_2$ for any matrix $\mathbf{A} \in \mathbb{K}^{n,m}$

3.3.3 QR-Decomposition [?, Sect. 13], [?, Sect. 7.3]

This section will answer the question whether and how it is possible to find orthogonal transformations that convert any given matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\operatorname{rank}(\mathbf{A}) = n$, to upper triangular form, as required for the application of the "equivalence transformation idea" to full-rank linear least squares problems.

3.3.3.1 Theory

(3.3.6) Gram-Schmidt orthogonalisation recalled → **§ 1.5.1**

Input: $\{\mathbf{a}^1, \ldots, \mathbf{a}^k\} \subset \mathbb{K}^n$
Output: $\{\mathbf{q}^1, \ldots, \mathbf{q}^k\}$ (assuming no premature termination!)

1: $\mathbf{q}^1 := \frac{\mathbf{a}^1}{\|\mathbf{a}^1\|_2}$ *% 1st output vector*
2: **for** $j = 2, \ldots, k$ **do**
{ *% Orthogonal projection*
3: $\mathbf{q}^j := \mathbf{a}^j$
4: **for** $\ell = 1, 2, \ldots, j-1$ **do** (GS)
5: { $\mathbf{q}^j \leftarrow \mathbf{q}^j - \langle \mathbf{a}^j, \mathbf{q}^\ell \rangle \mathbf{q}^\ell$ }
6: **if** ($\mathbf{q}^j = \mathbf{0}$) **then** STOP
7: **else** { $\mathbf{q}^j \leftarrow \frac{\mathbf{q}^j}{\|\mathbf{q}^j\|_2}$ }
8: }

Theorem 3.3.7. Span property of G.S. vectors

If $\{\mathbf{a}^1, \ldots, \mathbf{a}^k\}$ is linearly independent, then Algorithm (GS) *computes orthonormal vectors $\mathbf{q}^1, \ldots, \mathbf{q}^k$ satisfying*

$$\operatorname{Span}\{\mathbf{q}^1, \ldots, \mathbf{q}^\ell\} = \operatorname{Span}\{\mathbf{a}^1, \ldots, \mathbf{a}^\ell\}\,, \tag{1.5.2}$$

for all $\ell \in \{1, \ldots, k\}$.

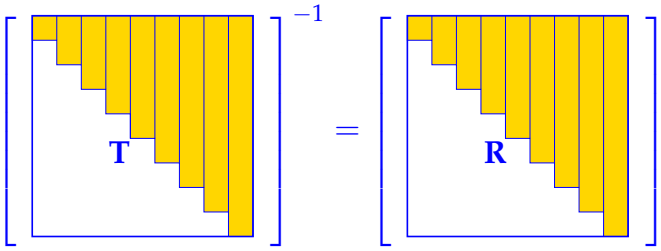
The span property (1.5.2) can be made more explicit in terms of the existence of linear combinations

$$\begin{aligned} \mathbf{q}_1 &= t_{11}\mathbf{a}_1 \\ \mathbf{q}_2 &= t_{12}\mathbf{a}_1 + t_{22}\mathbf{a}_2 \\ \mathbf{q}_3 &= t_{13}\mathbf{a}_1 + t_{23}\mathbf{a}_2 + t_{33}\mathbf{a}_3 \\ &\vdots \\ \mathbf{q}_k &= t_{1n}\mathbf{a}_1 + t_{2n}\mathbf{a}_2 + \cdots + t_{kk}\mathbf{a}_k\,. \end{aligned} \quad \blacktriangleright \quad \exists \mathbf{T} \in \mathbb{R}^{n,n} \text{ upper triangular: } \mathbf{Q} = \mathbf{AT}\,, \tag{3.3.8}$$

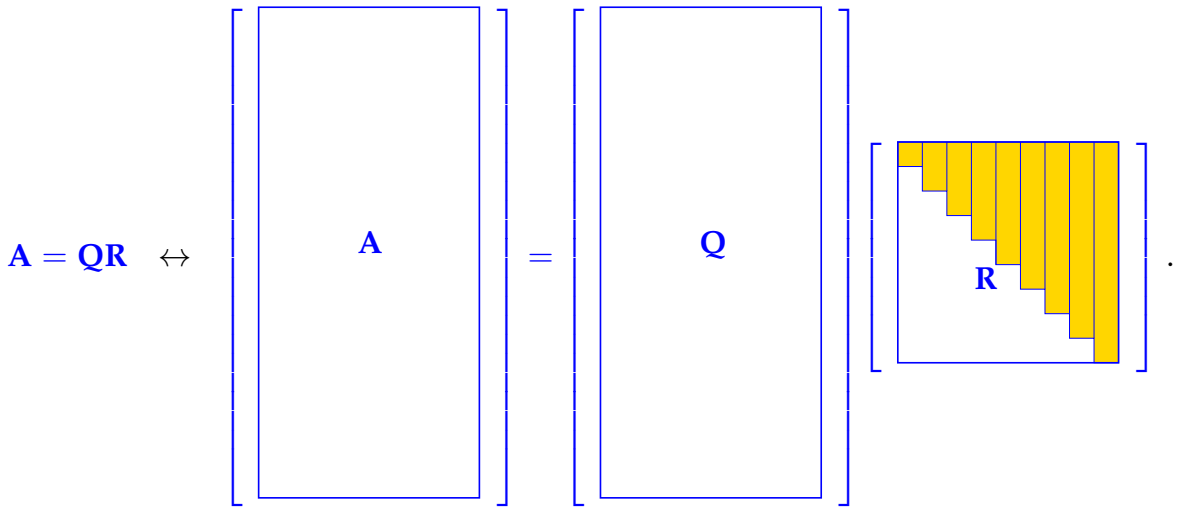
where $\mathbf{Q} = [\mathbf{q}_1, \ldots, \mathbf{q}_n] \in \mathbb{R}^{m,n}$ (with orthonormal columns), $\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_n] \in \mathbb{R}^{m,n}$. Note that thanks to the linear independence of $\{\mathbf{a}^1, \ldots, \mathbf{a}^k\}$ and $\{\mathbf{q}^1, \ldots, \mathbf{q}^k\}$, the matrix $\mathbf{T} = (t_{ij})_{i,j=1}^k \in \mathbb{R}^{k,k}$ is

regular ("non-existent" t_{ij} are set to zero, of course).

Recall from Lemma 1.3.9 that inverses of regular upper triangular matrices are upper triangular again.



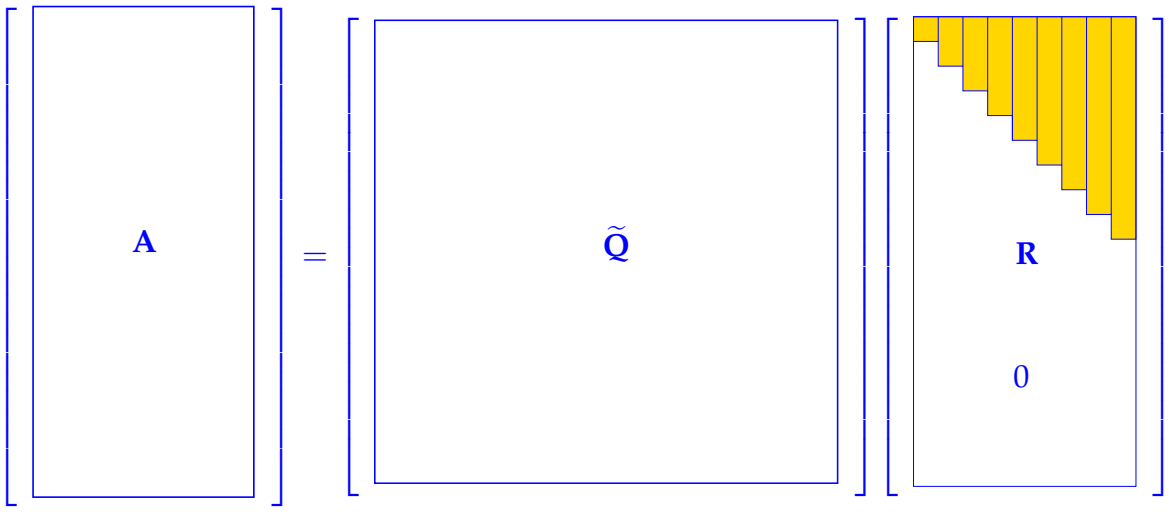
Thus, by (3.3.8), we have found an *upper triangular* $\mathbf{R} := \mathbf{T}^{-1} \in \mathbb{R}^{n,n}$ such that



Next "augmentation by zero": add $m-n$ zero rows at the bottom of $\mathbf{R}$ and complement columns of $\mathbf{Q}$ to an orthonormal basis of $\mathbb{R}^m$, which yields an orthogonal matrix $\widetilde{\mathbf{Q}} \in \mathbb{R}^{m,m}$:

$$\blacktriangleright \quad \mathbf{A} = \widetilde{\mathbf{Q}} \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix}$$

$$\updownarrow$$



$$\Leftrightarrow \quad \widetilde{\mathbf{Q}}^\top \mathbf{A} = \begin{bmatrix} \mathbf{R} \\ 0 \end{bmatrix} .$$

Thus the algorithm of Gram-Schmidt orthonormalization "proves" the following theorem.

Theorem 3.3.9. QR-decomposition → [?, Satz 5.2], [?, Sect. 7.3]

For any matrix $\mathbf{A} \in \mathbb{K}^{n,k}$ *with* $\operatorname{rank}(\mathbf{A}) = k$ *there exists*

(i) a unique *Matrix* $\mathbf{Q}_0 \in \mathbb{R}^{n,k}$ *that satisfies* $\mathbf{Q}_0^{\mathrm{H}}\mathbf{Q}_0 = \mathbf{I}_k$, *and a* unique *upper triangular Matrix* $\mathbf{R}_0 \in \mathbb{K}^{k,k}$ *with* $(\mathbf{R})_{i,i} > 0$, $i \in \{1,\ldots,k\}$, *such that*

$$\mathbf{A} = \mathbf{Q}_0 \cdot \mathbf{R}_0 \qquad \text{("economical" QR-decomposition)},$$

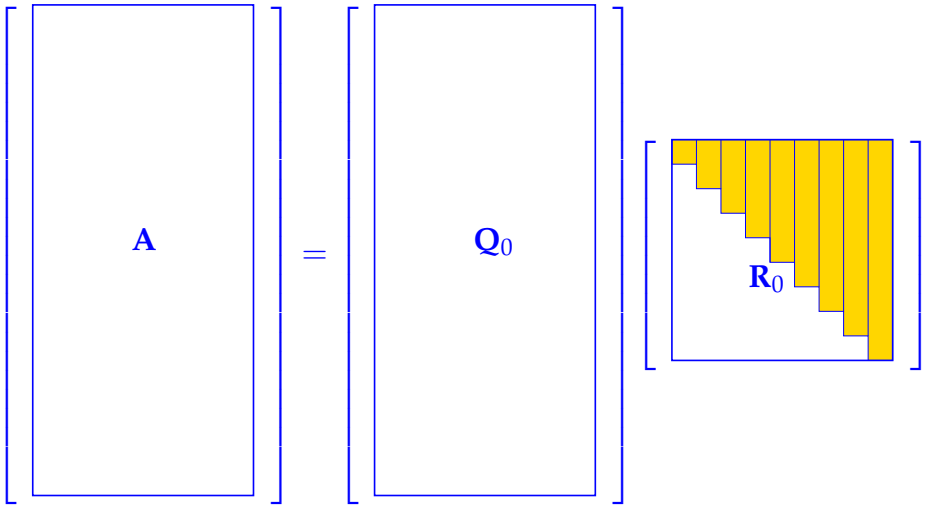
(ii) a unitary Matrix $\mathbf{Q} \in \mathbb{K}^{n,n}$ *and a* unique *upper triangular* $\mathbf{R} \in \mathbb{K}^{n,k}$ *with* $(\mathbf{R})_{i,i} > 0$, $i \in \{1,\ldots,n\}$, *such that*

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \qquad \text{(full QR-decomposition)}.$$

If $\mathbb{K} = \mathbb{R}$ *all matrices will be real and* $\mathbf{Q}$ *is then orthogonal.*

Visualisation: "economical" QR-decomposition: $\mathbf{Q}_0^{\mathrm{H}}\mathbf{Q}_0 = \mathbf{I}_k$ (orthonormal columns),

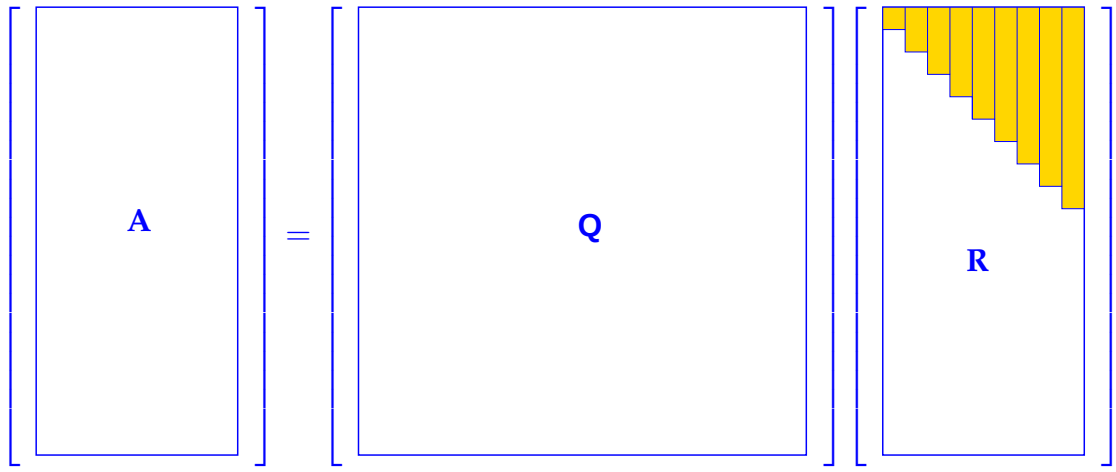
$\mathbf{A} = \mathbf{Q}_0\mathbf{R}_0$, $\mathbf{Q}_0 \in \mathbb{K}^{n,k}$, $\mathbf{R}_0 \in \mathbb{K}^{k,k}$ upper triangular,



(3.3.10)

Visualisation: full QR-decompisiton: $\mathbf{Q}^{\mathrm{H}}\mathbf{Q} = \mathbf{Q}\mathbf{Q}^{\mathrm{H}} = \mathbf{I}_n$ (orthogonal matrix),

$$\mathbf{A} = \mathbf{Q}\mathbf{R}, \quad \mathbf{Q} \in \mathbb{K}^{n,n}, \quad \mathbf{R} \in \mathbb{K}^{n,k},$$



. (3.3.11)

For square $\mathbf{A}$, that is, $n = k$, both QR-decompositions coincide.

Corollary 3.3.12. Uniqueness of QR-factorization

The "economical" QR-factorization (3.3.3.1) *of* $\mathbf{A} \in \mathbb{K}^{m,n}$, $m \geq n$, *with* $\operatorname{rank}(\mathbf{A}) = n$ *is unique, if we demand* $(\mathbf{R}_0)_{ii} > 0$.

Proof. We observe that $\mathbf{R}$ is regular, if $\mathbf{A}$ has full rank n. Since the regular upper triangular matrices form a group under multiplication:

$$\mathbf{Q}_1\mathbf{R}_1 = \mathbf{Q}_2\mathbf{R}_2 \quad \Rightarrow \quad \mathbf{Q}_1 = \mathbf{Q}_2\mathbf{R} \quad \text{with upper triangular } \mathbf{R} := \mathbf{R}_2\mathbf{R}_1^{-1} .$$

$$\blacktriangleright \quad \mathbf{I} = \mathbf{Q}_1^H\mathbf{Q}_1 = \mathbf{R}^H \underbrace{\mathbf{Q}_2^H\mathbf{Q}_2}_{=\mathbf{I}}\mathbf{R} = \mathbf{R}^H\mathbf{R} .$$

The assertion follows by uniqueness of Cholesky decomposition, Lemma 2.8.14. □

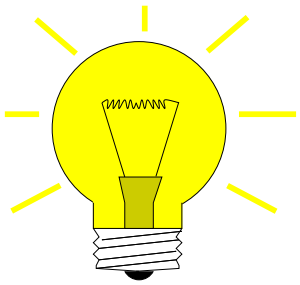
3.3.3.2 Computation of QR-Decomposition

In theory, Gram-Schmidt orthogonalization (GS) can be used to compute the QR-factorization of a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\operatorname{rank}(\mathbf{A}) = n$. However, as we saw in Exp. 1.5.5, Gram-Schmidt orthogonalization in the form of Code 1.5.3 is *not* a *stable* algorithm.

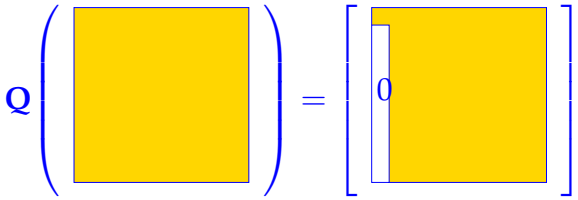
There is a stable way to compute QR-decompositions, based on the accumulation of orthogonal transformations.

Corollary 3.3.13. Composition of orthogonal transformations

The product of two orthogonal/unitary matrices of the same size is again orthogonal/unitary.



Idea: find simple unitary/orthogonal (row) transformations rendering certain matrix elements zero:



with $\mathbf{Q}^H = \mathbf{Q}^{-1}$.

Recall that this "annihilation of column entries" is the key operation in Gaussian forward elimination, where it is achieved by means of non-unitary row transformations, see Sect. 2.3.2. Now we want to find a counterpart of Gaussian elimination based on unitary row transformations on behalf of numerical stability.

Example 3.3.14 ("Annihilating" orthogonal transformations in 2D)

In 2D there are two possible orthogonal transformations make 2nd component of $\mathbf{a} \in \mathbb{R}^2$ vanish, which, in geometric terms, amounts to mapping the vector onto the x_1-axis.

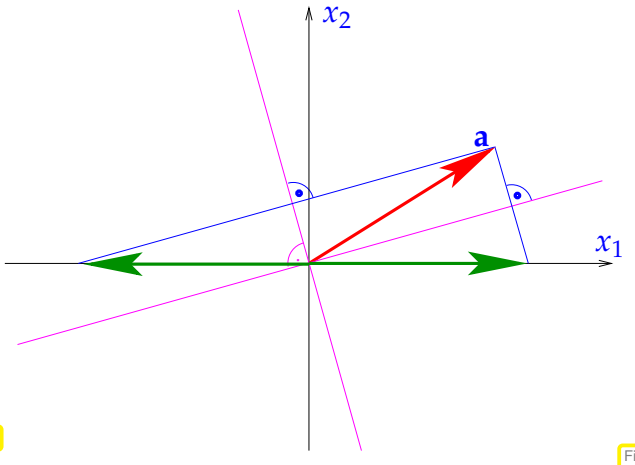


Fig. 100

reflections at angle bisector,

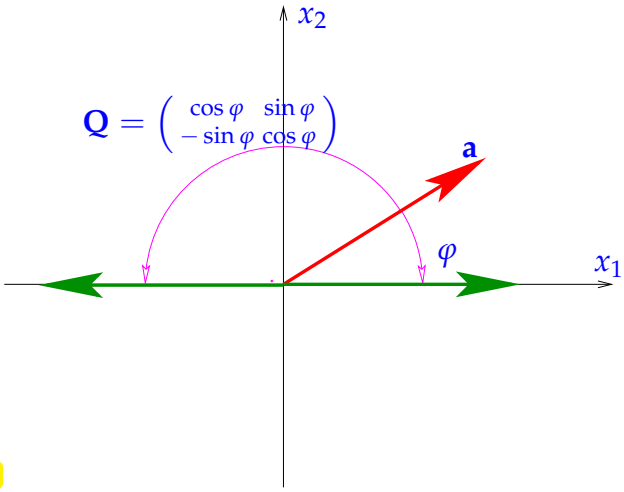


Fig. 101

rotations turning $\mathbf{a}$ onto x_1-axis.

Note that in each case we have *two different* length-preserving lineare mappings at our disposal. This flexibility will be important for curbing the impact of roundoff.

Both reflections and rotations are actually used in library routines and both are discussed in the sequel:

(3.3.15) Householder reflections $\rightarrow$ **[?, Sect. 5.1]**

The following so-called Householder matrices effect the reflection of a vector into a multiple of the first unit vector with the same length:

$$\mathbf{Q} = \mathbf{H}(\mathbf{v}) := \mathbf{I} - 2\frac{\mathbf{v}\mathbf{v}^H}{\mathbf{v}^H\mathbf{v}} \quad \text{with} \quad \mathbf{v} = \tfrac{1}{2}(\mathbf{a} \pm \|\mathbf{a}\|_2\mathbf{e}_1) \; . \tag{3.3.16}$$

Orthogonality of these matrices can be established by direct computation.

Fig. 102 sketches a "geometric derivation" of Householder reflections:

Given $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ with $\|\mathbf{a}\| = \|\mathbf{b}\|$, the difference vector $\mathbf{v} = \mathbf{b} - \mathbf{a}$ is orthogonal to the bisector.

$$\begin{aligned} \mathbf{b} = \mathbf{a} - (\mathbf{a} - \mathbf{b}) &= \mathbf{a} - \mathbf{v}\frac{\mathbf{v}^T\mathbf{v}}{\mathbf{v}^T\mathbf{v}} \\ &= \mathbf{a} - 2\mathbf{v}\frac{\mathbf{v}^T\mathbf{a}}{\mathbf{v}^T\mathbf{v}} = \mathbf{a} - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}\mathbf{a} = \mathbf{H}(\mathbf{v})\mathbf{a} \; , \end{aligned}$$

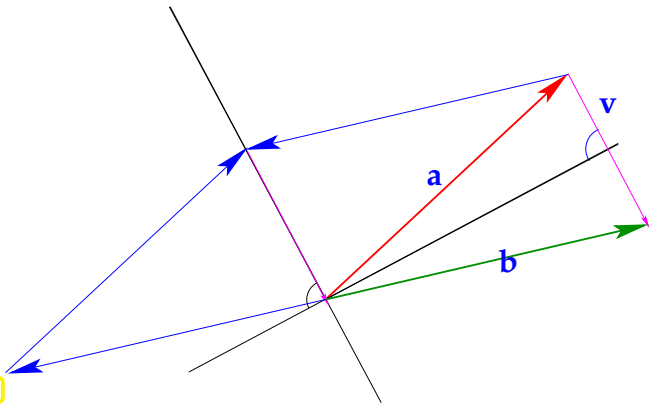
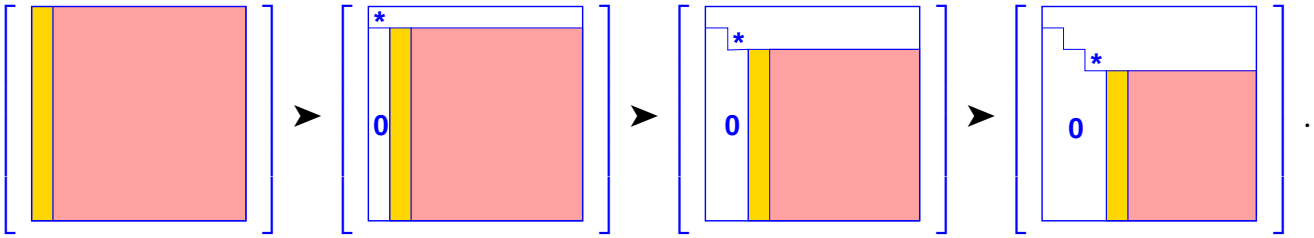


Fig. 102

because, due to orthogonality $(\mathbf{a}-\mathbf{b}) \perp (\mathbf{a}+\mathbf{b})$

$$(\mathbf{a}-\mathbf{b})^T(\mathbf{a}-\mathbf{b}) = (\mathbf{a}-\mathbf{b})^T(\mathbf{a}-\mathbf{b}+\mathbf{a}+\mathbf{b}) = 2(\mathbf{a}-\mathbf{b})^T\mathbf{a}\,.$$

Suitable successive Householder transformations determined by the leftmost column ("target column") of shrinking bottom right matrix blocks cen be used to achieve upper triangular form $\mathbf{R}$. Visualization of annihiliation of lower triangular matrix part for a square matrix:



= "target column $\mathbf{a}$" (determines unitary transformation),
= modified in course of transformations.

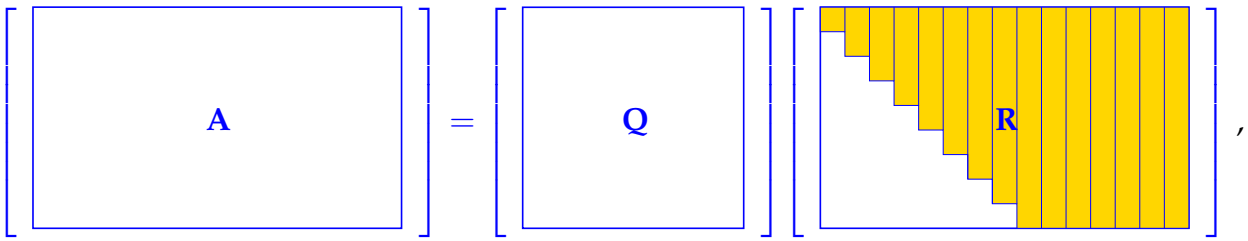
Writing $\mathbf{Q}_\ell$ for the Householder matrix used in the ℓ-th step we get

$$\mathbf{Q}_{n-1}\mathbf{Q}_{n-2}\cdot\cdots\cdot\mathbf{Q}_1\mathbf{A} = \mathbf{R}\,,$$

QR-factorization (QR-decomposition) of $\mathbf{A} \in \mathbb{C}^{n,n}$: $\mathbf{A} = \mathbf{QR}$, $\mathbf{Q} := \mathbf{Q}_1^H \cdot\cdots\cdot \mathbf{Q}_{n-1}^H$ orthogonal matrix, $\mathbf{R}$ upper triangular matrix.

Remark 3.3.17 (QR-decomposition of "fat" matrices)

We can also apply successive Householder transformation as outlined in § 3.3.15 to a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ with $m < n$. If the first m columns of $\mathbf{A}$ are linearly independent, we obtain another variant of the QR-decomposition:



$$\mathbf{A} = \mathbf{QR}\quad,\quad \mathbf{Q} \in \mathbb{R}^{m,m}\,,\quad \mathbf{R} \in \mathbb{R}^{m,n}\,,$$

where $\mathbf{Q}$ is orthogonal, $\mathbf{R}$ upper triangular, that is, $(\mathbf{R})_{i,j} = 0$ for $i > j$.

Remark 3.3.18 (Stable implementation of Householder reflections)

In (3.3.16) the computation of the vector $\mathbf{v}$ can be prone to cancellation ($\rightarrow$ Section 1.5.4), if the vector $\mathbf{a}$ encloses a very small angle with the first unit vector, because in this case $\mathbf{v}$ can be very small and beset with a huge relative error. This is a concern, because in the formula for the Householder matrix $\mathbf{v}$ is normalized to unit length (division by $\|\mathbf{v}\|_2^2$).

Fortunately, two choices for $\mathbf{v}$ are possible in (3.3.16) and at most one can be affected by cancellation. The right choice is

$$\mathbf{v} = \begin{cases} \frac{1}{2}(\mathbf{a} + \|\mathbf{a}\|_2 \mathbf{e}_1) & \text{, if } a_1 > 0 \,, \\ \frac{1}{2}(\mathbf{a} - \|\mathbf{a}\|_2 \mathbf{e}_1) & \text{, if } a_1 \leq 0 \,. \end{cases}$$

See [**?**, Sect. 19.1] and [**?**, Sect. 5.1.3] for a discussion.

(3.3.19) Givens rotations → **[?, Sect. 14]**

The 2D rotation displayed in Fig. 101 can be embedded in an identity matrix. Thus, the following orthogonal transformation, a Givens rotation, annihilates the k-th component of a vector $\mathbf{a} = [a_1, \ldots, a_n]^\top \in \mathbb{R}^n$. Here γ stands for $\cos(\varphi)$ and σ for $\sin(\varphi)$, φ the angle of rotation, see Fig. 101.

$$\mathbf{G}_{1k}(a_1, a_k)\mathbf{a} := \begin{bmatrix} \gamma & \cdots & \sigma & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ -\sigma & \cdots & \gamma & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ \vdots \\ a_n f \end{bmatrix} = \begin{bmatrix} a_1^{(1)} \\ \vdots \\ 0 \\ \vdots \\ a_n \end{bmatrix} \quad \text{, if } \begin{array}{rcl} \gamma & = & \frac{a_1}{\sqrt{|a_1|^2 + |a_k|^2}} \,, \\ \sigma & = & \frac{a_k}{\sqrt{|a_1|^2 + |a_k|^2}} \,. \end{array} \tag{3.3.20}$$

Orthogonality (→ Def. 6.2.2) of $\mathbf{G}_{1k}(a_1, a_k)$ is verified immediately. Again, we have two options for an annihilating rotation, see Ex. 3.3.14. It will always be possible to choose one that avoids cancellation [**?**, Sect. 5.1.8], see Code 3.3.21 for details.

C++11 code 3.3.21: Stable Givens rotation of a 2D vector

```
//!  plane (2D) Givens rotation avoiding cancellation
// Rotation matrix returned in G, rotated vector in x
void planerot(const Vector2d& a, Matrix2d& G, Vector2d& x) {
  if (a(1) != 0.0) {
    double t,s,c;
    if (std::abs(a(1)) > std::abs(a(0))) {
      t = -a(0)/a(1); s = 1.0/std::sqrt(1.0+t*t); c = s*t;
    }
    else {
      t = -a(1)/a(0); c = 1.0/std::sqrt(1+t*t); s = c*t;
    }
    // Form 2x2 Givens rotation matrix
    G << c,s,-s,c;
  }
  else G.setIdentity();
  x << a.norm(),0.0;
}
```

So far, we know how to annihilate a single component of a vector by means of a Givens rotation that targets that component and some other (the first in (3.3.20)). However, for the sake of QR-decomposition we aim to map *all* components to zero except for the first.

☞ This can be achieved by $n-1$ *successive* Givens rotations, see also Code 3.3.23

$$\begin{bmatrix} a_1 \\ \vdots \\ \vdots \\ \vdots \\ a_n \end{bmatrix} \xrightarrow{\mathbf{G}_{12}(a_1,a_2)} \begin{bmatrix} a_1^{(1)} \\ 0 \\ a_3 \\ \vdots \\ a_n \end{bmatrix} \xrightarrow{\mathbf{G}_{13}(a_1^{(1)},a_3)} \begin{bmatrix} a_1^{(2)} \\ 0 \\ 0 \\ a_4 \\ \vdots \\ a_n \end{bmatrix} \xrightarrow{\mathbf{G}_{14}(a_1^{(2)},a_4)} \cdots \xrightarrow{\mathbf{G}_{1n}(a_1^{(n-2)},a_n)} \begin{bmatrix} a_1^{(n-1)} \\ 0 \\ \vdots \\ \\ \vdots \\ 0 \end{bmatrix} \tag{3.3.22}$$

✎ Notation: $\mathbf{G}_{ij}(a_1,a_2) \hat{=}$ Givens rotation (3.3.20) aimed at rows i and j of the matrix.

C++11 code 3.3.23: Roating a vector onto the x_1-axis by successive Givens transformation

```
// Orthogonal transformation of a (column) vector into a multiple of
// the first unit vector by successive Givens transformations
void givenscoltrf(const VectorXd& aIn, MatrixXd& Q, VectorXd& aOut){
  unsigned int n = aIn.size();
  // Assemble rotations in a dense matrix.
  // For (more efficient) alternatives see Rem. Rem. 3.3.25
  Q.setIdentity();
  Matrix2d G; Vector2d tmp, xDummy;
  aOut = aIn;
  for(int j = 1; j < n; ++j) {
    tmp(0) = aOut(0); tmp(1) = aOut(j);
    planerot(tmp, G, xDummy); // see Code 3.3.21
    // select 1st and jth element of aOut and use the Map function
    // to prevent copying; equivalent to aOut([1,j]) in MATLAB
    Map<VectorXd, 0, InnerStride<> > aOutMap(aOut.data(), 2,
        InnerStride<>(j));
    aOutMap = G * aOutMap;
    // select 1st and jth column of Q (Q(:,[1,j]) in MATLAB)
    Map<MatrixXd, 0, OuterStride<> > QMap(Q.data(), n, 2,
        OuterStride<>(j*n));
    QMap = QMap * G.transpose();
  }
}
```

Armed with these compound Givens rotations we can proceed as in the case of Householder reflections to accomplish the orthogonal transformation of a full-rank matrix to upper triangular form, see

C++11 code 3.3.24: QR-decomposition by successive Givens rotations

```
//!  QR decomposition of square matrix A by successive Givens
    transformations
void qrgivens(const MatrixXd& A, MatrixXd& Q, MatrixXd& R){
  unsigned int n = A.rows();
  // Assemble rotations in a dense matrix.
  // For (more efficient) alternatives see Rem. Rem. 3.3.25
  Q.setIdentity();
  Matrix2d G; Vector2d tmp, xDummy;
  R = A; // In situ transformation
```

```
10    for(int i = 0; i < n-1; ++i) {
11      for(int j = n-1; j > i; --j) {
12        tmp(0) = R(j-1, i); tmp(1) = R(j,i);
13        planerot(tmp, G, xDummy); // see Code 3.3.21
14        R.block(j-1,0,2,n) = G * R.block(j-1,0,2,n);
15        Q.block(0, j-1, n, 2) = Q.block(0,j-1,n,2) * G.transpose();
16      }}
17  }
```

Remark 3.3.25 (Storing orthogonal transformations)

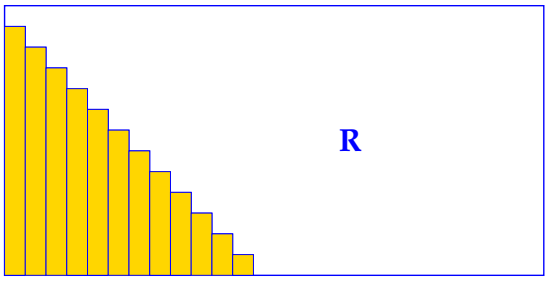
When doing successive orthogonal transformations as in the case of QR-decomposition by means of Householder reflections (→ § 3.3.15) or Givens rotations (→ § 3.3.19) it would be prohibitively expensive to assemble and even multiply the transformation matrices!

The matrices for the orthogonal transformation are never built in codes!
The transformations are stored in a compressed format.

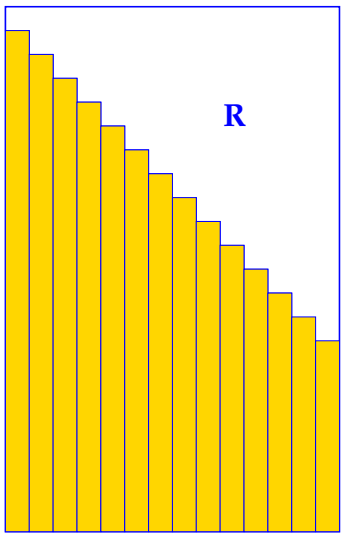
❶ In the case of Householder reflections $H(\mathbf{v}) \in \mathbb{R}^{m,m}$ (3.3.16)

➤ store only the last $n-1$ components of the normalized vector $\mathbf{v} \in \mathbb{R}^m$

For QR-decomposition of a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, by means of successive Householder reflections $\mathbf{H}(\mathbf{v}_1) \cdot \cdots \cdot \mathbf{H}(\mathbf{v}_k)$, $k := \min\{m,n\}$, we store the bottom parts of the vectors $\mathbf{v}_j \in \mathbb{R}^{m-j+1}$, $j = 1,\ldots,k$, whose lengths decrease, in place of the "annihilated" lower triangular part of $\mathbf{A}$:



↑ Case $m < n$



Case $m > n$ →

↔ space for Householder vectors

❷ In the case of Givens rotations, for a single rotation $G_{i,j}(a_1,a_2)$

➤ store row indices (i,j) and rotation angle [**?**, Sect. 5.1.11],

$$\text{for } \mathbf{G} = \begin{pmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{pmatrix} \Rightarrow \text{store } \rho := \begin{cases} 1 & \text{, if } \gamma = 0\,, \\ \frac{1}{2}\operatorname{sign}(\gamma)\sigma & \text{, if } |\sigma| < |\gamma|\,, \\ 2\operatorname{sign}(\sigma)/\gamma & \text{, if } |\sigma| \geq |\gamma|\,, \end{cases}$$

$$\text{which means} \quad \begin{cases} \rho = 1 & \Rightarrow \quad \gamma = 0\,, \quad \sigma = 1 \\ |\rho| < 1 & \Rightarrow \quad \sigma = 2\rho\,, \quad \gamma = \sqrt{1-\sigma^2} \\ |\rho| > 1 & \Rightarrow \quad \gamma = 2/\rho\,, \quad \sigma = \sqrt{1-\gamma^2}\,. \end{cases}$$

Then store $\mathbf{G}_{ij}(a,b)$ as triple (i,j,ρ). The parameter ρ forgets the sign of the matrix $\mathbf{G}_{ij}$, so the signs of the corresponding rows in the transformed matrix $\mathbf{R}$ have to be changed accordingly. The rationale behind the above convention is to curb the impact of roundoff errors.

Remark 3.3.26 (QR-decomposition of banded matrices)

The advantage of Givens rotations is its selectivity, which can be exploited for banded matrices, see Section 2.7.6, Def. 2.7.55.

Example: Orthogonal transformation of an $n \times n$ *tridiagonal* matrix to upper triangular form, that is, the annihilation of the sub-diagonal, by means of successive Givens rotations:

$$\begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{\mathbf{G}_{12}} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{\mathbf{G}_{23}\cdots\mathbf{G}_{n-1,n}} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}$$

$*$ $\hat{=}$ entry set to zero by Givens rotation, $*$ $\hat{=}$ new non-zero entry ("fill-in" $\rightarrow$ Def. 2.7.47).

This is a manifestation of a more general result, see Def. 2.7.55 for notatons:

Theorem 3.3.27. QR-decomposition "preserves bandwidth"

If $\mathbf{A} = \mathbf{QR}$ is the QR-decomposition of a regular matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, then $\mathrm{bw}(\mathbf{R}) \leq \mathrm{bw}(\mathbf{A})$.

A total of only n Givens rotations is required, involving an asymptotic total computational effort of $O(nm)$ for an $m \times n$-matrix.

3.3.3.3 QR-Decomposition: Stability

In numerical linear algebra orthogonal transformation methods usually give rise to reliable algorithms, thanks to the norm-preserving property of orthogonal transformations.

(3.3.28) Stability of unitary/orthogonal transformations

We consider the mapping (the "transformation" induced by $\mathbf{Q}$)

$$F : \mathbb{K}^n \to \mathbb{K}^n\,, \quad F(\mathbf{x}) := \mathbf{Q}\mathbf{x}\,, \quad \mathbf{Q} \in \mathbb{K}^{n,n} \text{ unitary/orthogonal}\,.$$

We are interested in the sensitivity of F, that is, the impact of relative errors in the data vector $\mathbf{x}$ on the output vector $\mathbf{y} := F(\mathbf{x})$.

We study the output for a perturbed input vector:

$$\left.\begin{array}{rcl} \mathbf{Q}\mathbf{x} = \mathbf{y} & \Rightarrow & \|\mathbf{x}\|_2 = \|\mathbf{y}\|_2 \\ \mathbf{Q}(\mathbf{x}+\Delta\mathbf{x}) = \mathbf{y}+\Delta\mathbf{y} \;\Rightarrow\; \mathbf{Q}\Delta\mathbf{x} = \Delta\mathbf{y} & \Rightarrow & \|\Delta\mathbf{y}\|_2 = \|\Delta\mathbf{x}\|_2 \end{array}\right\} \quad \blacktriangleright \quad \frac{\|\Delta\mathbf{y}\|_2}{\|\mathbf{y}\|_2} = \frac{\|\Delta\mathbf{x}\|_2}{\|\mathbf{x}\|_2} .$$

We conclude, that unitary/orthogonal transformations do *not* involve any *amplification of relative errors* in the data vectors.

Of course, this also applies to the "solution" of square linear systems with orthogonal coefficient matrix $\mathbf{Q} \in \mathbb{R}^{n,n}$, which, by Def. 6.2.2, boils down to multiplication of the right hand side vector with $\mathbf{Q}^{\mathrm{H}}$.

Remark 3.3.29 (Conditioning of conventional row transformations)

Gaussian elimination as presented in § 2.3.3 converts a matrix to upper triangular form by elementary row transformations. Those add a scalar multiple of a row of the matrix to another matrix and amount to left-multiplication with matrices

$$\mathbf{T} := \mathbf{I}_n + \mu \mathbf{e}_j \mathbf{e}_i^{\top} , \quad \mu \in \mathbb{K} , \quad i,j \in \{1,\ldots,n\},\ i \neq j . \tag{3.3.30}$$

However, these transformations can lead to a massive amplification of relative errors, which, by virtue of Ex. 2.2.7 can be linked to large condition numbers of $\mathbf{T}$.

This accounts for fact that the computation of LU-decompositions by means of Gaussian elimination might not be stable, see Ex. 2.4.5.

Experiment 3.3.31 (Conditioning of conventional row transformations, Rem. 3.3.29 cnt'd)

Study in 2D:

2×2 row transformation matrix, (*cf.* elimination matrices of Gaussian elimination.

$$\mathbf{T}(\mu) = \begin{pmatrix} 1 & 0 \\ \mu & 1 \end{pmatrix}$$

Euclidean condition numbers of $\mathbf{T}(\mu)$ ▷

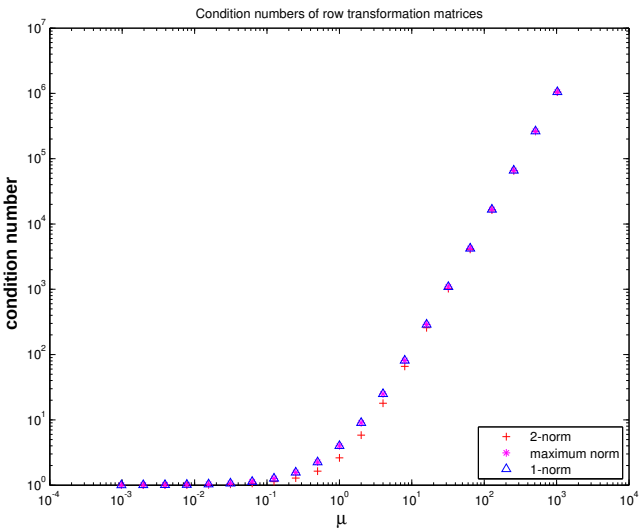


Fig. 103

The perfect conditioning of orthogonal transformation prevents the destructive build-up of roundoff errors.

Theorem 3.3.32. Stability of Householder QR [?, Thm. 19.4]

Let $\widetilde{\mathbf{R}} \in \mathbb{R}^{m,n}$ be the R-factor of the QR-decomposition of $\mathbf{A} \in \mathbb{R}^{m,n}$ computed by means of successive Householder reflections (→ § 3.3.15). Then there exists an orthogonal $\mathbf{Q} \in \mathbb{R}^{m,m}$ such that

$$\mathbf{A} + \Delta\mathbf{A} = \mathbf{Q}\widetilde{\mathbf{R}} \quad \text{with} \quad \|\Delta\mathbf{A}\|_2 \leq \frac{cmn\,\mathrm{EPS}}{1 - cmn\,\mathrm{EPS}}\|\mathbf{A}\|_2 , \tag{3.3.33}$$

where EPS is the machine precision and $c > 0$ a small constant independent of $\mathbf{A}$.

3.3.3.4 QR-Decomposition in EIGEN

EIGEN offers several classes dedicated to computing QR-type decompositions of matrices, for instance **HouseholderQR**. Internally the QR-decomposition is stored in compressed format as explained in Rem. 3.3.25. Its computation is triggered by the constructor.

C++-code 3.3.34: QR-decompositions in EIGEN

```
2  # include <Eigen/QR>
3
4  // Computation of full QR-decomposition (3.3.3.1),
5  // dense matrices built for both QR-factors (expensive!)
6  std::pair<MatrixXd,MatrixXd> qr_decomp_full(const MatrixXd& A) {
7    Eigen::HouseholderQR<MatrixXd> qr(A);
8    MatrixXd Q = qr.householderQ(); //
9    MatrixXd R = qr.matrixQR().template triangularView<Eigen::Upper>();
10   return std::pair<MatrixXd,MatrixXd>(Q,R);
11 }
12
13 // Computation of economical QR-decomposition (3.3.3.1),
14 // dense matrix built for Q-factor (possibly expensive!)
15 std::pair<MatrixXd,MatrixXd> qr_decomp_eco(const MatrixXd& A) {
16   using index_t = MatrixXd::Index;
17   const index_t m = A.rows(),n = A.cols();
18   Eigen::HouseholderQR<MatrixXd> qr(A);
19   MatrixXd Q = (qr.householderQ()*MatrixXd::Identity(m,n)); //
20   MatrixXd R = qr.matrixQR().block(0,0,n,n).template
       triangularView<Eigen::Upper>(); //
21   return std::pair<MatrixXd,MatrixXd>(Q,R);
22 }
```

Note that the method `householderQ` returns the Q-factor in compressed format → Rem. 3.3.25. Assignment to a matrix will convert it into a (dense) matrix format, see Line 8; only then the actual computation of the matrix entries performed. It can also be multiplied with another matrix of suitable size, which is used in Line 19 to extract the Q-factor $\mathbf{Q}_0 \in \mathbb{R}^{m,n}$ of the economical QR-decomposition (3.3.3.1).

The matrix returned by the method `matrixQR()` gives access to a matrix storing the QR-factors in compressed form. Its upper triangular part provides $\mathbf{R}$, see Line 20.

A close inspection of the algorithm for the computation of QR-decompositions of $\mathbf{A} \in \mathbb{R}^{m,n}$ by successive

Householder reflections (→ § 3.3.15) reveals, that n transformations costing $\sim mn$ operations each are required.

Asymptotic complexity of Householder QR-decomposition

The computational effort for **HouseholderQR()** of $\mathbf{A} \in \mathbb{R}^{m,n}$, $m > n$, is $O(mn^2)$ for $m, n \to \infty$.

Experiment 3.3.36 (Asymptotic complexity of Householder QR-factorization)

C++-code 3.3.37: timing QR-factorizations in EIGEN

```
2  int nruns = 3, minExp = 2, maxExp = 6;
3  MatrixXd tms(maxExp-minExp+1,4);
4  for(int i = 0; i <= maxExp-minExp; ++i){
5    Timer t1, t2, t3;    // timer class
6    int n = std::pow(2, minExp + i); int m = n*n;
7    // Initialization of matrix A
8    MatrixXd A(m,n); A.setZero();
9    A.setIdentity(); A.block(n,0,m-n,n).setOnes();
10   A += VectorXd::LinSpaced(m,1,m) * RowVectorXd::LinSpaced(n,1,n);
11   for(int j = 0; j < nruns; ++j) {
12     // plain QR-factorization in the constructor
13     t1.start(); HouseholderQR<MatrixXd> qr(A); t1.stop();
14     // full decomposition
15     t2.start(); std::pair<MatrixXd,MatrixXd> QR2 =
         qr_decomp_full(A); t2.stop();
16     // economic decomposition
17     t3.start(); std::pair<MatrixXd,MatrixXd> QR3 =
         qr_decomp_eco(A); t3.stop();
18   }
19   tms(i,0)=n;
20   tms(i,1)=t1.min(); tms(i,2)=t2.min(); tms(i,3)= t3.min();
21 }
```

Timings for

- plain QR-factorization in the constructor of **HouseholderQR**,
- invocation of function `qr_decomp_full()`, see Code 3.4.13,
- call to `qr_decomp_eco()` from Code 3.4.13.

Platform:

- ubuntu 14.04 LTS
- i7-3517U CPU @ 1.90GHz × 4
- L1 32 KB, L2 256 KB, L3 4096 KB, Mem 8 GB
- gcc 4.8.4, -O3

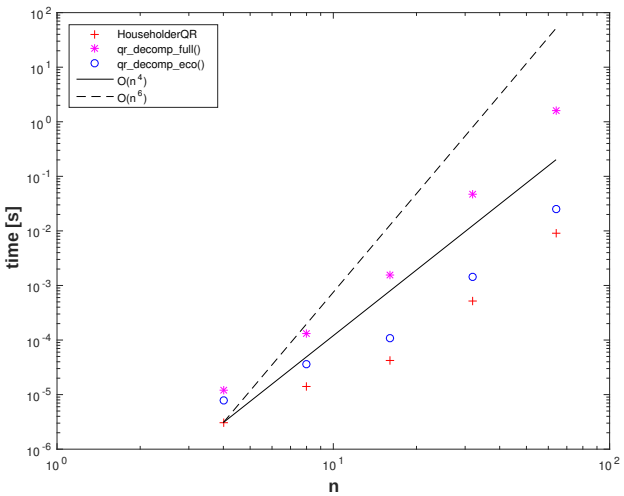


Fig.

3.3.4 QR-Based Solver for Linear Least Squares Problems

The QR-decomposition introduced in Section 3.3.3, Thm. 3.3.9, paves the way for the practical algorithmic realization of the "equivalent orthonormal transformation to upper triangular form"-idea from Section 3.3.1.

We consider the full-rank linear least squares problem Eq. (3.1.31): Given $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\mathrm{rank}(\mathbf{A}) = n$,

$$\text{seek} \quad \mathbf{x} \in \mathbb{R}^n \quad \text{such that} \quad \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \to \min \ .$$

We assume that we are given a QR-decomposition: $\mathbf{A} = \mathbf{Q}\mathbf{R}$, $\mathbf{Q} \in \mathbb{R}^{m,m}$ orthogonal, $\mathbf{R} \in \mathbb{R}^{m,n}$ (regular) upper triangular matrix.

We apply the orthogonal 2-norm preserving ($\to$ Thm. 3.3.5) transformation encoded in $\mathbf{Q}$ to $\mathbf{A}\mathbf{x} - \mathbf{b}$, the vector inside the 2-norm to be minimized:

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 = \left\|\mathbf{Q}(\mathbf{R}\mathbf{x} - \mathbf{Q}^{\mathrm{H}}\mathbf{b})\right\|_2 = \left\|\mathbf{R}\mathbf{x} - \widetilde{\mathbf{b}}\right\|_2 \quad , \quad \widetilde{\mathbf{b}} := \mathbf{Q}^{\mathrm{H}}\mathbf{b} \ .$$

Thus, we have obtained an *equivalent* triangular linear least squares problem:

$$\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \to \min \quad \Leftrightarrow \quad \left\| \begin{bmatrix} \mathbf{R}_0 \\ \mathbf{0} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} \widetilde{b}_1 \\ \vdots \\ \\ \\ \vdots \\ \widetilde{b}_m \end{bmatrix} \right\|_2 \to \min \ .$$

$$\mathbf{x} = \begin{bmatrix} \mathbf{R}_0 \end{bmatrix}^{-1} \begin{bmatrix} \widetilde{b}_1 \\ \vdots \\ \widetilde{b}_n \end{bmatrix} \quad , \quad \text{with residual} \quad \mathbf{r} = \mathbf{Q} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \widetilde{b}_{n+1} \\ \vdots \\ \widetilde{b}_m \end{bmatrix} \ .$$

Note: by Thm. 3.3.5 the norm of the residual is readily available: $\|\mathbf{r}\|_2 = \sqrt{\widetilde{b}_{n+1}^2 + \cdots + \widetilde{b}_m^2}$.

C++-code 3.3.38: QR-based solver for full rank linear least squares problem (3.1.31)

```
2 // Solution of linear least squares problem (3.1.31) by means of
    QR-decomposition
```

```
3  // Note:  A ∈ ℝ^{m,n} with m > n, rank(A) = n is assumed
4  // Least squares solution returned in x, residual norm as return value
5  double qrlsqsolve(const MatrixXd& A, const VectorXd& b,
6                    VectorXd& x) {
7    const unsigned m = A.rows(), n = A.cols();
8
9    MatrixXd Ab(m, n + 1); Ab << A, b; // Form extended matrix [A,b]
10
11   // QR-decomposition of extended matrix automatically transforms b
12   MatrixXd R = Ab.householderQr().matrixQR().template
13     triangularView<Eigen::Upper>(); //
14
15   MatrixXd R_nn = R.block(0, 0, n, n); // R-factor R_0
16   // Compute least squares solution x = (R)^{-1}_{1:n,1:n} (Q^T b)_{1:n}
17   x = R_nn.template triangularView<Eigen::Upper>().solve(R.block(0,
       n, n, 1));
18   return R(n, n); // residual norm = ||Ax̂ − b||_2 (why ?)
19 }
```

Discussion of (some) details of implementation in Code 3.3.38:

- The QR-decomposition is computed in a numerically stable way by means of Householder reflections (→ § 3.3.15) by EIGEN's built-in function **householderQR** available for matrix type. The computational cost of this function when called for an $m \times n$ matrix is, asymptotically for $m, n \to \infty$, $O(n^2 m)$.
- Line 9: We perform the QR-decomposition of the extended matrix $[\mathbf{A}, \mathbf{b}]$ with $\mathbf{b}$ as rightmost column. Thus, the orthogonal transformations are automatically applied to $\mathbf{b}$; the augmented matrix is converted into $[\mathbf{R}, \mathbf{Q}^\top \mathbf{b}]$, the data of the equivalent upper triangular linear least squares problem. Thus, actually, no information about $\mathbf{Q}$ needs to be stored, if one is interested in the least squares solution $\mathbf{x}$ only.

  The idea is borrowed from Gaussian elimination, see Code 2.3.4, Line 9.
- Line 13: `MatrixQR()` returns the compressed QR-factorization as a matrix, where the R-factor $\mathbf{R} \in \mathbb{R}^{m,n}$ is contained in the upper triangular part, whose top n rows give $\mathbf{R}_0$ from see (3.3.3.1).
- Line 18: the components $(\mathbf{b})_{n+2:m}$ of the vector $\mathbf{b}$ (treated as rightmost column of the augmented matrix) are annihilated when computing the QR-decomposition (by final Householder reflection): $\left(\mathbf{Q}^{\mathrm{H}}[\mathbf{A}, \mathbf{b}]\right)_{n+2:m,n} = 0$. Hence, $\left(\mathbf{Q}^{\mathrm{H}}[\mathbf{A}, \mathbf{b}]\right)_{n+1,n+1} = \left\| (\widetilde{\mathbf{b}})_{n+1:m} \right\|_2$, which gives the norm of the residual.

➤ A QR-based algorithm is implemented in the `solve()` method available for EIGEN's QR-decomposition, see Code 3.3.39.

C++11 code 3.3.39: EIGEN's built-in QR-based linear least squares solver

```
2  // Solving a full-rank least squares problem ||Ax − b||_2 → min in EIGEN
3  double lsqsolve_eigen(const MatrixXd& A, const VectorXd& b,
4                        VectorXd& x) {
5    x = A.householderQr().solve(b);
6    return ((A*x−b).norm());
```

```
7 }
```

Remark 3.3.40 (QR-based solution of linear systems of equations)

Applying the QR-based algorithm for full-rank linear least squares problems in the case $m = n$, that is, to a square linear system of equations $\mathbf{Ax} = \mathbf{b}$ with a regular coefficient matrix , will compute the solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. In a sense, the QR-decomposition offers an alternative to Gaussian elimination/LU-decomposition discussed in § 2.3.30.

The steps for solving a linear system of equations $\mathbf{Ax} = \mathbf{b}$ by means of QR-decomposition are as follows:

$\mathbf{Ax} = \mathbf{b}$:
① QR-decomposition $\mathbf{A} = \mathbf{QR}$, computational costs $\frac{2}{3}n^3 + O(n^2)$ (about twice as expensive as LU-decomposition without pivoting)
② orthogonal transformation $\mathbf{z} = \mathbf{Q}^H\mathbf{b}$, computational costs $4n^2 + O(n)$ (in the case of *compact storage* of reflections/rotations)
③ Backward substitution, solve $\mathbf{Rx} = \mathbf{z}$, computational costs $\frac{1}{2}n(n+1)$

Benefit: we can utterly dispense with any kind of pivoting:

✌ Computing the generalized QR-decomposition $\mathbf{A} = \mathbf{QR}$ by means of Householder reflections or Givens rotations is (numerically stable) for any $\mathbf{A} \in \mathbb{C}^{m,n}$.
✌ For *any* regular system matrix an LSE can be solved by means of
QR-decomposition + orthogonal transformation + backward substitution
in a stable manner.

Drawback: QR-decomposition can hardly ever avoid massive fill-in (→ Def. 2.7.47) also in situations, where LU-factorization greatly benefits from Thm. 2.7.58.

Remark 3.3.41 (QR-based solution of banded LSE)

From Rem. 3.3.26, Thm. 3.3.27, we know that that particular situtation, in which QR-decomposition can avoid fill-in (→ Def. 2.7.47) is the case of banded matrices, see Def. 2.7.55. For a banded $n \times n$ linear systems of equations with small fixed bandwidth $\mathrm{bw}(\mathbf{A}) \leq O(1)$ we incur an

➢ asymptotic computational effort: $O(n)$ for $n \to \infty$

The following code uses QR-decomposition computed by means of selective Givens rotations (→ § 3.3.19) to solve a tridiagonal linear system of equations $\mathbf{Ax} = \mathbf{b}$

$$\mathbf{A} = \begin{bmatrix} d_1 & c_1 & 0 & & \dots & & 0 \\ e_1 & d_2 & c_2 & & & & \vdots \\ 0 & e_2 & d_3 & c_3 & & & \\ \vdots & & \ddots & \ddots & \ddots & & c_{n-1} \\ 0 & & \dots & & 0 & e_{n-1} & d_n \end{bmatrix}$$

The matrix is passed in the form of three vectors $\mathbf{e}, \mathbf{c}, \mathbf{d}$ giving the entries in the non-zero bands.

C++11 code 3.3.42: Solving a tridiagonal system by means of QR-decomposition ➙ GITLAB

```
2 //!  @brief Solves the tridiagonal system Ax = b with QR-decomposition
```

```
//!  @param[in] d Vector of dim n; the diagonal elements
//!  @param[in] c Vector of dim n-1; the lower diagonal elements
//!  @param[in] e Vector of dim n-1; the upper diagonal elements
//!  @param[in] b Vector of dim n; the rhs.
//!  @param[out] x Vector of dim n
VectorXd tridiagqr(VectorXd c, VectorXd d, VectorXd e, VectorXd& b){
  int n = d.size();
  // resize the vectors c and d to correct length if needed
  c.conservativeResize(n); e.conservativeResize(n);
  double t = d.norm() + e.norm() + c.norm();
  Matrix2d R;  Vector2d z, tmp;
  for(int k = 0; k < n-1; ++k){
    tmp(0) = d(k); tmp(1) = e(k);
    // Use givensrotation to set the entries below the diagonal
    // to zero
    planerot(tmp, R, z); // see Code 3.3.21
    if( std::abs(z(0))/t < std::numeric_limits<double>::epsilon() )
      throw std::runtime_error("A nearly singular");
    // Update all other entries of the matrix and rhs.  which
    // were affected by the givensrotation
    d(k) = z(0);
    b.segment(k,2).applyOnTheLeft(R);    // rhs.
    // Block of the matrix affected by the givensrotation
    Matrix2d Z;
    Z << c(k),      0,  d(k+1), c(k+1);
    Z.applyOnTheLeft(R);
    // Write the transformed block back to the corresponding places
    c.segment(k,2) = Z.diagonal(); d(k+1) = Z(1,0); e(k) = Z(0,1);
  }
  // Note that the e is now above d and c
  // Backsubstitution acting on upper triangular matrix
  // with upper bandwidth 2 (stored in vectors).
  VectorXd x(n);
  // last row
  x(n-1) = b(n-1)/d(n-1);
  if(n >= 2) {
    // 2nd last row
    x(n-2) = (b(n-2)-c(n-2)*x(n-1))/d(n-2);
    // remaining rows
    for(int i = n-3; i >= 0; --i)
      x(i) = ( b(i) - c(i) * x(i+1) - e(i)*x(i+2) ) / d(i);
  }
  return x;
}
```

Example 3.3.43 (Stable solution of LSE by means of QR-decomposition)

Aiming to confirm the claim of superior stability of QR-based approaches (→ Rem. 3.3.40, § 3.3.28) we revisit Wilkinson's counterexample from Ex. 2.4.5 for which Gaussian elimination with partial pivoting does not yield an acceptable solution.

Wilkinson matrix $\mathbf{A} \in \mathbb{R}^{n,n}$

$$(\mathbf{A})_{i,j} := \begin{cases} -1 & \text{for } i > j,\ j < n\ , \\ 1 & \text{for } i = j\ , \\ 0 & \text{for } i < j,\ j < n\ , \\ 1 & \text{for } j = n\ . \end{cases}$$

QR-decomposition produces perfect solution ▷

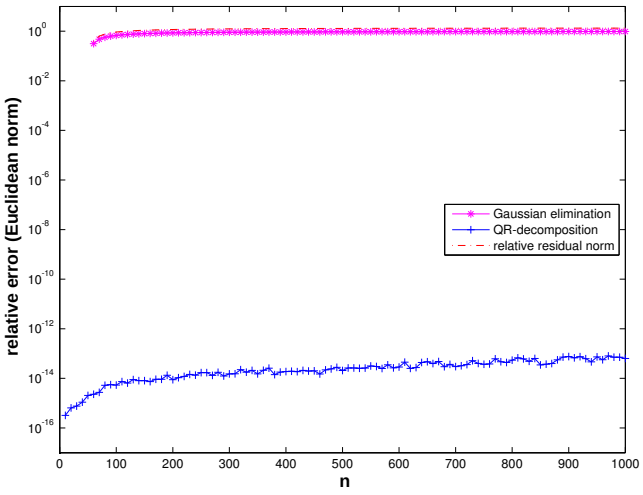


Fig. 105

Let us summarize the pros and cons of orthogonal transformation techniques for linear least squares problems:

Normal equations vs. orthogonal transformations method

Superior numerical stability (→ Def. 1.5.85) of orthogonal transformations methods:

- Use orthogonal transformations methods for least squares problems (3.1.38), whenever $\mathbf{A} \in \mathbb{R}^{m,n}$ *dense* and n small.

SVD/QR-factorization cannot exploit sparsity:

- Use normal equations in the expanded form (3.2.8)/(3.2.9), when $\mathbf{A} \in \mathbb{R}^{m,n}$ sparse (→ Notion 2.7.1) and m, n big.

3.3.5 Modification Techniques for QR-Decomposition

In § 2.6.13 we faced the task of solving a square linear system of equations $\widetilde{\mathbf{A}}\mathbf{x} = \mathbf{b}$ efficiently, whose coefficient matrix $\widetilde{\mathbf{A}}$ was a (rank-1) perturbation of $\mathbf{A}$, for which an LU-decomposition was available. Lemma 2.6.22 showed a way to reuse the information contained in the LU-decomposition.

A similar task can be posed, if the QR-decomposition of a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, has already been computed and then we have to solve a full-rank linear least squares problem $\left\|\widetilde{\mathbf{A}}\mathbf{x} - \mathbf{b}\right\|_2 \to \min$ with $\widetilde{\mathbf{A}} \in \mathbb{R}^{m,n}$ a "slight" perturbation of $\mathbf{A}$. If we aim to use orthogonalization techniques it would be desirable to compute the QR-decomposition of $\widetilde{\mathbf{A}}$ with recourse to the QR-decomposition of $\mathbf{A}$.

3.3.5.1 Rank-1 Modifications

For $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\operatorname{rank}(\mathbf{A}) = n$, we consider the rank-1 modification, *cf.* Eq. (2.6.17),

$$\mathbf{A} \quad \longrightarrow \quad \widetilde{\mathbf{A}} := \mathbf{A} + \mathbf{u}\mathbf{v}^\top\ , \quad \mathbf{u} \in \mathbb{R}^m\ , \quad \mathbf{v} \in \mathbb{R}^n\ . \tag{3.3.45}$$

We assume that still $\operatorname{rank}(\widetilde{\mathbf{A}}) = n$.

Given the (unique) full QR-decomposition $\mathbf{A} = \mathbf{QR}$, $\mathbf{Q} \in \mathbb{R}^{m,m}$ orthogonal, $\mathbf{R} \in \mathbb{R}^{m,n}$ upper triangular, according to Thm. 3.3.9, the goal is to find an efficient algorithm that yields the (unique) full QR-decomposition of $\widetilde{\mathbf{A}}$: $\widetilde{\mathbf{A}} = \widetilde{\mathbf{Q}}\widetilde{\mathbf{R}}$.

Step ❶: compute $\mathbf{w} = \mathbf{Q}^\top \mathbf{u}$, and observe $\mathbf{A} + \mathbf{uv}^\top = \mathbf{Q}(\mathbf{R} + \mathbf{wv}^\top)$.

➢ Computational effort $O(mn)$, if $\mathbf{Q}$ stored in suitable (compressed) format.

Step ❷: achieve $\mathbf{w} \to \|\mathbf{w}\|\mathbf{e}_1$ through applying $m-1$ Givens rotations, visualization for $m = n$:

$$\mathbf{w} = \begin{pmatrix} * \\ * \\ \vdots \\ * \\ * \\ * \\ * \end{pmatrix} \xrightarrow{\mathbf{G}_{n-1,n}} \begin{pmatrix} * \\ * \\ \vdots \\ * \\ * \\ * \\ 0 \end{pmatrix} \xrightarrow{\mathbf{G}_{n-2,n-1}} \begin{pmatrix} * \\ * \\ \vdots \\ * \\ * \\ 0 \\ 0 \end{pmatrix} \xrightarrow{\mathbf{G}_{n-3,n-2}} \cdots \xrightarrow{\mathbf{G}_{12}} \begin{pmatrix} * \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Note that this will affect $\mathbf{R}$ by creating a single non-zero subdiagonal:

$$\mathbf{R} = \begin{pmatrix} * & * & \cdots & * & * & * & * \\ 0 & * & \cdots & * & * & * & * \\ \vdots & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \\ 0 & \cdots & 0 & 0 & 0 & 0 & * \end{pmatrix} \xrightarrow{\mathbf{G}_{n-1,n}} \begin{pmatrix} * & * & \cdots & * & * & * & * \\ 0 & * & \cdots & * & * & * & * \\ \vdots & \ddots & & & & \vdots & \\ 0 & \cdots & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{n-2,n-1}}$$

$$\longrightarrow \begin{pmatrix} * & * & \cdots & * & * & * & * \\ 0 & * & \cdots & * & * & * & * \\ \vdots & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{n-3,n-2}} \cdots \xrightarrow{\mathbf{G}_{1,2}} \begin{pmatrix} * & * & \cdots & * & * & * & * \\ * & * & \cdots & * & * & * & * \\ & & \ddots & & & & \vdots \\ 0 & \cdots & * & * & * & * & * \\ 0 & \cdots & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \end{pmatrix} =: \mathbf{R}_1$$

We see $(\mathbf{R}_1)_{i,j} = 0$, if $i > j+1$, it a so-called upper Hessenberg matrix. If $\mathbf{Q}_1$ collects all orthogonal transformations used in Step ❷, then

$$\mathbf{A} + \mathbf{uv}^H = \mathbf{QQ}_1^H \big(\underbrace{\mathbf{R}_1 + \|\mathbf{w}\|_2 \mathbf{e}_1 \mathbf{v}^H}_{\text{upper Hessenberg matrix}} \big) \quad \text{with orthogonal } \mathbf{Q}_1 := \mathbf{G}_{12} \cdot \cdots \cdot \mathbf{G}_{n-1,n} .$$

➢ Computational effort $O(m + n^2)$

Step ❸: Convert $\mathbf{R}_1 + \|\mathbf{w}\|_2 \mathbf{e}_1 \mathbf{v}^H \mapsto$ upper triangular form by successive Givens rotations:

$$\mathbf{R}_1 + \|\mathbf{w}\|_2 \mathbf{e}_1 \mathbf{v}^H = \begin{pmatrix} * & * & \cdots & * & * & * & * \\ * & * & \cdots & * & * & * & * \\ & & \ddots & & & & \\ 0 & \cdots & * & * & * & * & * \\ 0 & \cdots & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{12}} \begin{pmatrix} * & * & \cdots & * & * & * & * \\ 0 & * & \cdots & * & * & * & * \\ & & \ddots & & & & \\ 0 & \cdots & * & * & * & * & * \\ 0 & \cdots & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{23}} \cdots$$

$$\xrightarrow{\mathbf{G}_{n-2,n-1}} \begin{pmatrix} * & * & \cdots & * & * & * & * \\ 0 & * & \cdots & * & * & * & * \\ & & \ddots & & & & \\ 0 & \cdots & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{n-1,n}} \begin{pmatrix} * & * & \cdots & * & * & * & * \\ 0 & * & \cdots & * & * & * & * \\ & & \ddots & & & & \\ 0 & \cdots & 0 & * & * & * & * \\ 0 & \cdots & 0 & 0 & * & * & * \\ 0 & \cdots & 0 & 0 & 0 & * & * \\ 0 & \cdots & 0 & 0 & 0 & 0 & * \end{pmatrix} =: \widetilde{\mathbf{R}} \,. \quad (3.3.46)$$

We need n Givens rotations acting on matrix rows of length n:

➢ Computational effort $O(n^2)$

➤ $\mathbf{A} + \mathbf{u}\mathbf{v}^H = \widetilde{\mathbf{Q}}\widetilde{\mathbf{R}}$ with $\widetilde{\mathbf{Q}} = \mathbf{Q}\mathbf{Q}_1^H\mathbf{G}_{n-1,n}^H \cdot \cdots \cdot \mathbf{G}_{12}^H$.

➢ Asymptotic total computational effort $O(mn)$ for $m, n \to \infty$

For large n this is much cheaper than the cost $O(n^2m)$ for computing the QR-decomposition of $\widetilde{\mathbf{A}}$ from scratch.

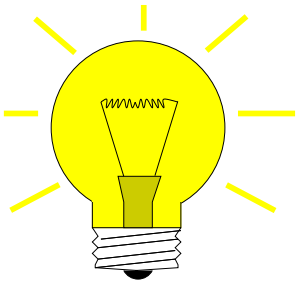
3.3.5.2 Adding a Column

We obtain an augmented matrix $\widetilde{\mathbf{A}} \in \mathbb{R}^{m,n+1}$ by inserting a column $\mathbf{v}$ into $\mathbf{A} \in \mathbb{R}^{m,n}$ at an arbitrary location.

$$\mathbf{A} \in \mathbb{R}^{m,n} \longrightarrow \widetilde{\mathbf{A}} = [\mathbf{a}_1, \ldots, \mathbf{a}_k, \mathbf{v}, \mathbf{a}_k, \ldots, \mathbf{a}_n] \,, \quad \mathbf{v} \in \mathbb{R}^m, \ \mathbf{a}_j := (\mathbf{A})_{:,j} \,. \quad (3.3.47)$$

Given: QR-decomposition $\mathbf{A} = \mathbf{Q}\mathbf{R}$, $\mathbf{Q} \in \mathbb{R}^{m,m}$ orthogonal, $\mathbf{R} \in \mathbb{R}^{m,n}$ upper triangular, of $\mathbf{A}$

Sought: QR-decomposition $\widetilde{\mathbf{A}} = \widetilde{\mathbf{Q}}\widetilde{\mathbf{R}}$, $\mathbf{Q} \in \mathbb{R}^{m,n}$ orthogonal, $\widetilde{\mathbf{R}} \in \mathbb{R}^{m,n+1}$ upper triangular, of $\widetilde{\mathbf{A}}$ from (3.3.47) computed efficiently.



Idea: Our task is easy in the case $k = n+1$ **!**

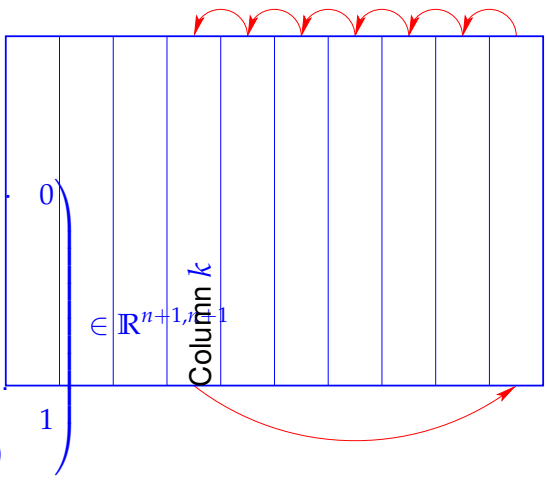
➤ First move new column to rightmost position

To move columns employ (partial) cyclic permutation of columns

$$k \mapsto n+1 \,, i \mapsto i-1 \,, i = k+1, \ldots, n+1 \,,$$

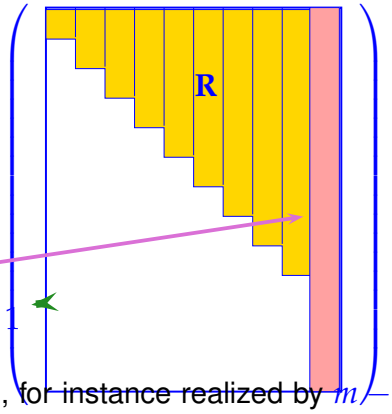
effected by multiplying with orthogonal permutation matrix

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & \cdots & & \cdots & 0 \\ 0 & \ddots & & & & \\ & & 1 & 0 & & \\ \vdots & & & 0 & 1 & \\ \vdots & & & & \ddots & \\ & & & & & 1 \\ 0 & \cdots & & 1 & 0 & \end{pmatrix}$$



Effects the following transformation of $\widetilde{\mathbf{A}}$:

$$\widetilde{\mathbf{A}} \longrightarrow \mathbf{A}_1 = \widetilde{\mathbf{A}}\mathbf{P} = [\mathbf{a}_1,\ldots,\mathbf{a}_n,\mathbf{v}] = \mathbf{Q}\begin{bmatrix}\mathbf{R} & \mathbf{Q}^\top\mathbf{v}\end{bmatrix} = \mathbf{Q}$$



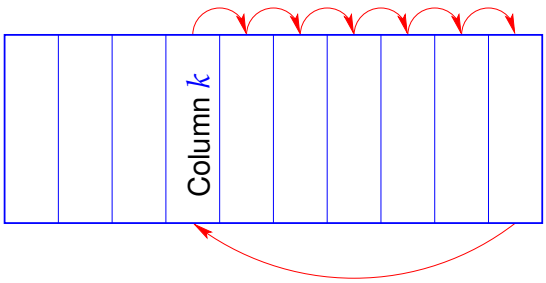
Column $\mathbf{Q}^\top\mathbf{v}$
Case $m > n+1$

① If $m > n+1$ there is an orthogonal transformation $\mathbf{Q}_1 \in \mathbb{R}^{m,m}$, for instance realized by $m-n-1$ Givens rotations or a single Householder reflection, such that

$$\mathbf{Q}_1\mathbf{Q}^\top\mathbf{v} = \begin{pmatrix} * \\ \vdots \\ * \\ * \\ 0 \\ \vdots \\ 0 \end{pmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \end{matrix}\right\} n+1 \\ \left.\begin{matrix} \\ \\ \end{matrix}\right\} m-n-1 \end{matrix} \quad \blacktriangleright \quad \mathbf{Q}_1\mathbf{Q}^\top\mathbf{A}_1 = \begin{pmatrix} * & \cdots & & \cdots & * \\ 0 & * & & & * \\ \vdots & & \ddots & & \vdots \\ & & & * & * \\ \vdots & & & & * \\ 0 & \cdots & & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & & \cdots & 0 \end{pmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} n+1 \\ \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} m-n-1 \end{matrix}$$

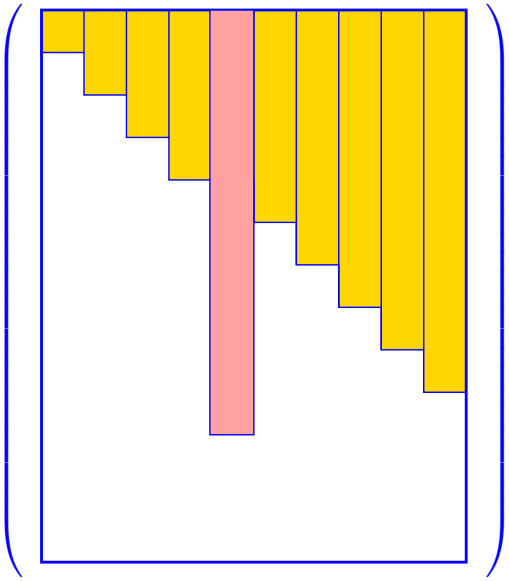
➢ Computational effort $O(m-n)$

To undo the initial moving of the new column we have to apply the inverse permutation:
∼ Right-multiply with $\mathbf{P}^\top$



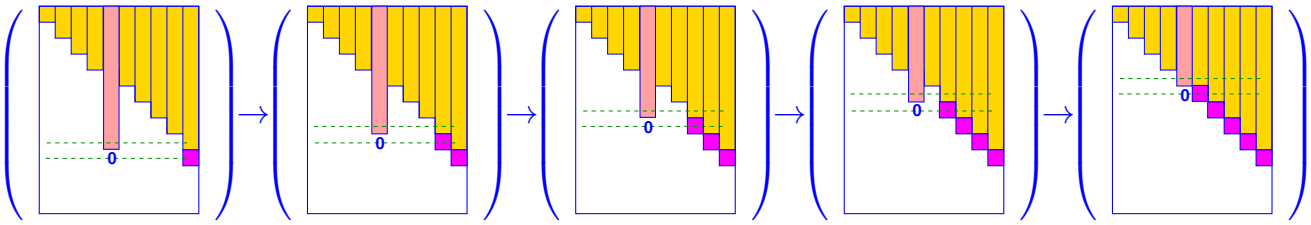
We can visualize the impact on the structure of the matrix:

$$\mathbf{Q}_1\mathbf{Q}^\top\widetilde{\mathbf{A}} = \mathbf{Q}_1\mathbf{Q}^\top\mathbf{A}_1\mathbf{P}^\top = \begin{pmatrix} * & \cdots & & * & & \cdots & * \\ 0 & * & & * & & & \vdots \\ \vdots & & \ddots & \vdots & & & \\ & & & \vdots & \ddots & & \vdots \\ & & & * & & * & * \\ \vdots & & & * & & & * \\ 0 & \cdots & & 0 & & \cdots & 0 \\ \vdots & & & \vdots & & & \vdots \\ 0 & \cdots & & 0 & & \cdots & 0 \end{pmatrix} =$$



② Apply $n+1-k$ successive Givens rotations to convert $\mathbf{Q}_1\mathbf{Q}^\top\widetilde{\mathbf{A}}$ into upper triangular form:

$$\mathbf{Q}_1\mathbf{Q}^\top\mathbf{A}_1 = \begin{pmatrix} * & \cdots & * & \cdots & * \\ 0 & * & * & & \vdots \\ \vdots & & \ddots & & \\ & & * & & * \\ & & * & & * \\ & & * & & 0 \\ 0 & \cdots & 0 & \cdots & 0 \end{pmatrix} \xrightarrow{\mathbf{G}_{n,n+1}} \cdots \xrightarrow{\mathbf{G}_{k,k+1}} \begin{pmatrix} * & \cdots & * & \cdots & * \\ 0 & * & * & & \vdots \\ \vdots & & \ddots & & \\ & & * & & \\ & & 0 & \ddots & \\ & & 0 & * & * \\ & & 0 & & * \\ 0 & \cdots & 0 & \cdots & 0 \end{pmatrix}$$



▬ ▬ ▬ ▪ ≙ target rows of Givens rotations, ■ ≙ new entries $\neq 0$

➢ Computational effort for this step $O\big((n-k)^2\big)$

➢ Total asymptotic computational cost $O(n^2+m)$

3.3.5.3 Adding a Row

We are given a matrix $\mathbf{A}\in\mathbb{R}^{m,n}$ of which a full QR-decomposition (→ Thm. 3.3.9) $\mathbf{A}=\mathbf{QR}$, $\mathbf{Q}\in\mathbb{R}^{m,m}$ orthogonal, $\mathbf{R}\in\mathbb{R}^{m,n}$ upper triangular, is already available, maybe only in encoded form (→ Rem. 3.3.25).

We add another row to the matrix $\mathbf{A}$ in arbitrary position $k\in\{1,\ldots,m\}$

$$\mathbf{A}\in\mathbb{R}^{m,n} \quad\mapsto\quad \widetilde{\mathbf{A}} = \begin{bmatrix} \mathbf{a}_{1,\cdot} \\ \vdots \\ \mathbf{a}_{k-1,\cdot} \\ \mathbf{v}^T \\ \mathbf{a}_{k,\cdot} \\ \vdots \\ \mathbf{a}_{m,\cdot} \end{bmatrix}, \quad \mathbf{v}\in\mathbb{R}^n\,. \tag{3.3.48}$$

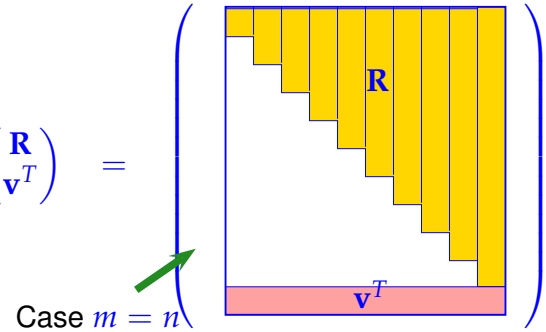
Task: Find an algorithm for the *efficient* computation of the QR-decomposition $\widetilde{\mathbf{A}}=\widetilde{\mathbf{Q}}\widetilde{\mathbf{R}}$ of $\widetilde{\mathbf{A}}$ from (3.3.48), $\widetilde{\mathbf{Q}}\in\mathbb{R}^{m+1,m+1}$ orthogonal, $\widetilde{\mathbf{R}}\in\mathbb{K}^{m+1,n+1}$ upper triangular.

Step ①: Move new row to the bottom.

Employ partial cyclic permutation of rows of $\widetilde{\mathbf{A}}$:

$$\text{row } m+1 \leftarrow \text{row } k\,, \quad \text{row } i \leftarrow \text{row } i+1\,, \quad i=k,\ldots,m\,.$$

$$\mathbf{P}\widetilde{\mathbf{A}} = \begin{pmatrix} \mathbf{A} \\ \mathbf{v}^T \end{pmatrix} \quad \blacktriangleright \quad \begin{pmatrix} \mathbf{Q}^H & 0 \\ 0 & 1 \end{pmatrix} \mathbf{P}\widetilde{\mathbf{A}} = \begin{pmatrix} \mathbf{R} \\ \mathbf{v}^T \end{pmatrix} \quad =$$



Case $m = n$

Step ②: Restore upper triangular form through Givens rotations (→ § 3.3.19)

Successively target bottom row and rows from the top to turn leftmost entries of bottom row into zeros.

$$\begin{pmatrix} * & \cdots & & & \cdots & * \\ 0 & * & & & & \vdots \\ \vdots & 0 & \ddots & & & \\ \vdots & \vdots & & * & & \vdots \\ 0 & 0 & & 0 & * & * \\ * & \cdots & \cdots & * & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{1,m}} \begin{pmatrix} * & \cdots & & & \cdots & * \\ 0 & * & & & & \vdots \\ \vdots & 0 & \ddots & & & \\ \vdots & \vdots & & * & & \vdots \\ 0 & 0 & & 0 & * & * \\ 0 & * & \cdots & * & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{2,m}} \cdots$$

$$\cdots \xrightarrow{\mathbf{G}_{m-2,m}} \begin{pmatrix} * & \cdots & & & \cdots & * \\ 0 & * & & & & \vdots \\ \vdots & 0 & \ddots & & & \\ \vdots & \vdots & & * & & \vdots \\ 0 & 0 & & 0 & * & * \\ 0 & \cdots & & 0 & * & * \end{pmatrix} \xrightarrow{\mathbf{G}_{m-1,m}} \begin{pmatrix} * & \cdots & & & \cdots & * \\ 0 & * & & & & \vdots \\ \vdots & 0 & \ddots & & & \\ \vdots & \vdots & & * & & \vdots \\ 0 & 0 & & 0 & * & * \\ 0 & \cdots & & & 0 & * \end{pmatrix} := \widetilde{\mathbf{R}} \quad (3.3.49)$$

Step ③: Incorporate rotations into $\mathbf{Q}$:

Setting $\mathbf{Q}_1 = \mathbf{G}_{m-1,m} \cdot \cdots \cdot \mathbf{G}_{1,m}$ the final QR-decomposition reads

$$\widetilde{\mathbf{A}} = \mathbf{P}^T \begin{pmatrix} \mathbf{Q} & 0 \\ 0 & 1 \end{pmatrix} \mathbf{Q}_1^H \widetilde{\mathbf{R}} = \widetilde{\mathbf{Q}}\widetilde{\mathbf{R}} \quad \text{with orthogonal } \widetilde{\mathbf{Q}} \in \mathbb{K}^{m+1,m+1} ,$$

because the product of orthogonal matrices is again orthogonal.

3.4 Singular Value Decomposition (SVD)

Beside the QR-decomposition of matrix $\mathbf{A} \in]\mathbb{R}^{m,n}$ there are other factorizations based on orthogonal transformations. The most important among them is the singular value decomposition (SVD), which can be used to tackle linear least squares problems and many other optimization problems beyond, see [**?**].

3.4.1 SVD: Definition and Theory

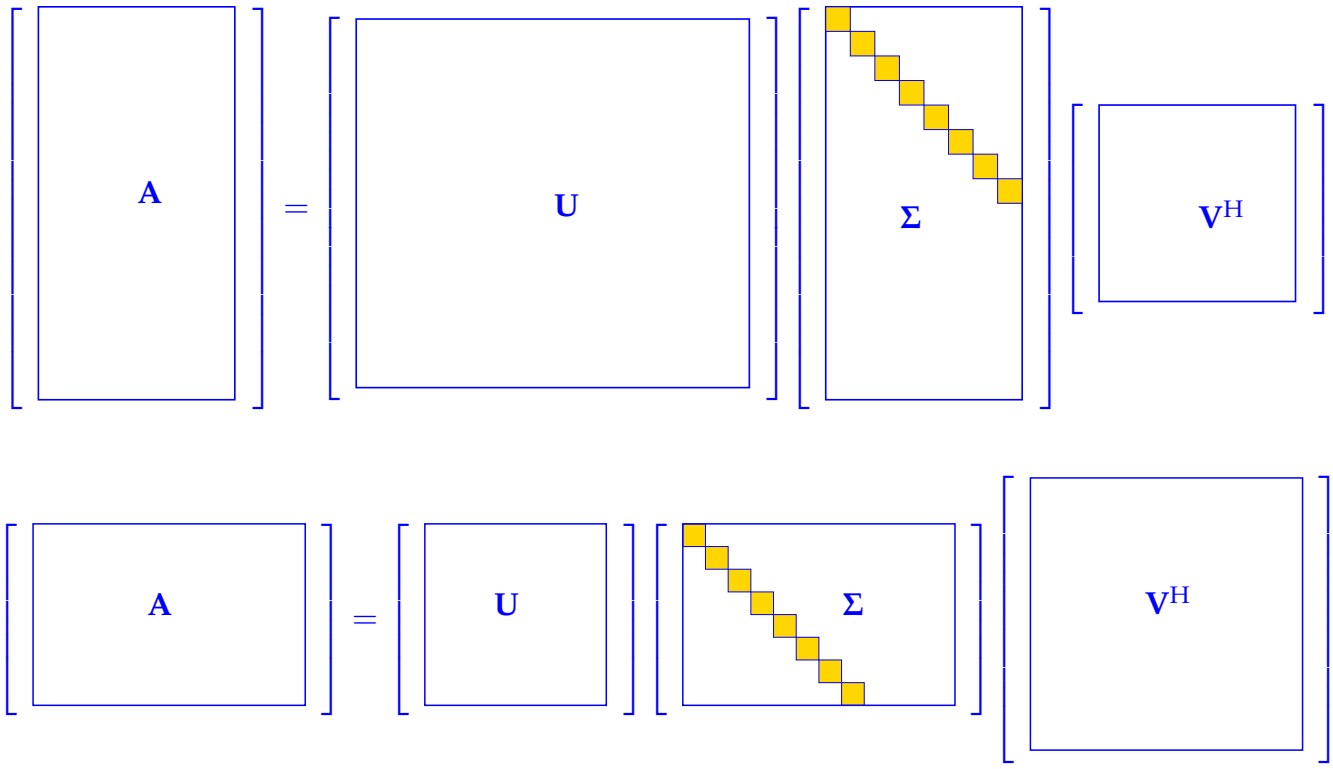
Theorem 3.4.1. singular value decomposition → **[?, Thm. 9.6], [?, Thm. 11.1]**

For any $\mathbf{A} \in \mathbb{K}^{m,n}$ *there are* unitary *matrices* $\mathbf{U} \in \mathbb{K}^{m,m}$, $\mathbf{V} \in \mathbb{K}^{n,n}$ *and a (generalized) diagonal* $^{(*)}$ *matrix* $\mathbf{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_p) \in \mathbb{R}^{m,n}$, $p := \min\{m,n\}$, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$ *such that*

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}} .$$

Terminology $(*)$: A matrix $\mathbf{\Sigma}$ is called a generalized diagonal matrix, if $(\mathbf{\Sigma})_{i,j} = 0$, if $i \neq j$, $1 \leq i \leq m$, $1 \leq j \leq n$. We still use the diag operator to create it from a vector.

Visualization of the structure of the singalar value decomposition of a matrix $\mathbf{A} = \mathbb{K}^{m,n}$.



Proof. (of Thm. 3.4.1, by induction)

To start the induction note that the assertion of the theorem is immediate for $n = 1$ or $m = 1$.

For the induction step $(n-1, m-1) \Rightarrow (m,n)$ first remember from analysis [**?**, Thm. 4.2.3]: Continuous functions attain extremal values on compact sets (here the unit ball $\{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x}\|_2 \leq 1\}$). In particular, consider the function $\mathbf{x} \mapsto \|\mathbf{A}\mathbf{x}\|$.

$$\blacktriangleright \quad \exists \mathbf{x} \in \mathbb{K}^n, \mathbf{y} \in \mathbb{K}^m , \quad \|\mathbf{x}\| = \|\mathbf{y}\|_2 = 1 : \quad \mathbf{A}\mathbf{x} = \sigma\mathbf{y} , \quad \sigma = \|\mathbf{A}\|_2 ,$$

where we used the definition of the matrix 2-norm, see Def. 1.5.76. By Gram-Schmidt orthogonalization or a similar procedure we can extend the single unit vectors $\mathbf{x}$ and $\mathbf{y}$ to orthonormal bases of $\mathbb{K}^n$ and $\mathbb{K}^m$, respectivelt: $\exists \widetilde{\mathbf{V}} \in \mathbb{K}^{n,n-1}$, $\widetilde{\mathbf{U}} \in \mathbb{K}^{m,m-1}$ such that

$$\mathbf{V} = [\mathbf{x}\ \widetilde{\mathbf{V}}] \in \mathbb{K}^{n,n} \quad , \quad \mathbf{U} = [\mathbf{y}\ \widetilde{\mathbf{U}}] \in \mathbb{K}^{m,m} \quad \text{are unitary.}$$

$$\blacktriangleright \quad \mathbf{U}^{\mathrm{H}}\mathbf{A}\mathbf{V} = \begin{bmatrix} \mathbf{y} & \widetilde{\mathbf{U}} \end{bmatrix}^{\mathrm{H}} \mathbf{A} \begin{bmatrix} \mathbf{x} & \widetilde{\mathbf{V}} \end{bmatrix} = \left[\begin{array}{c|c} \mathbf{y}^{\mathrm{H}}\mathbf{A}\mathbf{x} & \mathbf{y}^{\mathrm{H}}\mathbf{A}\widetilde{\mathbf{V}} \\ \hline \widetilde{\mathbf{U}}^{\mathrm{H}}\mathbf{A}\mathbf{x} & \widetilde{\mathbf{U}}^{\mathrm{H}}\mathbf{A}\widetilde{\mathbf{V}} \end{array}\right] = \left[\begin{array}{c|c} \sigma & \mathbf{w}^{\mathrm{H}} \\ \hline \mathbf{0} & \mathbf{B} \end{array}\right] =: \mathbf{A}_1 \,.$$

For the induction argument we have to show that $\mathbf{w} = \mathbf{0}$. Since

$$\left\| \mathbf{A}_1 \begin{bmatrix} \sigma \\ \mathbf{w} \end{bmatrix} \right\|_2^2 = \left\| \begin{bmatrix} \sigma^2 + \mathbf{w}^{\mathrm{H}}\mathbf{w} \\ \mathbf{B}\mathbf{w} \end{bmatrix} \right\|_2^2 = (\sigma^2 + \mathbf{w}^{\mathrm{H}}\mathbf{w})^2 + \|\mathbf{B}\mathbf{w}\|_2^2 \geq (\sigma^2 + \mathbf{w}^{\mathrm{H}}\mathbf{w})^2 \,,$$

we conclude

$$\|\mathbf{A}_1\|_2^2 = \sup_{0 \neq \mathbf{x} \in \mathbb{K}^n} \frac{\|\mathbf{A}_1\mathbf{x}\|_2^2}{\|\mathbf{x}\|_2^2} \geq \frac{\left\|\mathbf{A}_1 \begin{pmatrix} \sigma \\ \mathbf{w} \end{pmatrix}\right\|_2^2}{\left\| \begin{pmatrix} \sigma \\ \mathbf{w} \end{pmatrix} \right\|_2^2} \geq \frac{(\sigma^2 + \mathbf{w}^{\mathrm{H}}\mathbf{w})^2}{\sigma^2 + \mathbf{w}^{\mathrm{H}}\mathbf{w}} = \sigma^2 + \mathbf{w}^{\mathrm{H}}\mathbf{w} \,. \tag{3.4.2}$$

$$\sigma^2 = \|\mathbf{A}\|_2^2 = \left\|\mathbf{U}^{\mathrm{H}}\mathbf{A}\mathbf{V}\right\|_2^2 = \|\mathbf{A}_1\|_2^2 \overset{(3.4.2)}{\Longrightarrow} \|\mathbf{A}_1\|_2^2 = \|\mathbf{A}_1\|_2^2 + \|\mathbf{w}\|_2^2 \quad \Rightarrow \quad \mathbf{w} = \mathbf{0} \,.$$

$$\blacktriangleright \quad \mathbf{A}_1 = \left[\begin{array}{c|c} \sigma & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{B} \end{array}\right] .$$

Then apply the induction argument to $\mathbf{B}$.

□

Definition 3.4.3. Singular value decomposition (SVD)

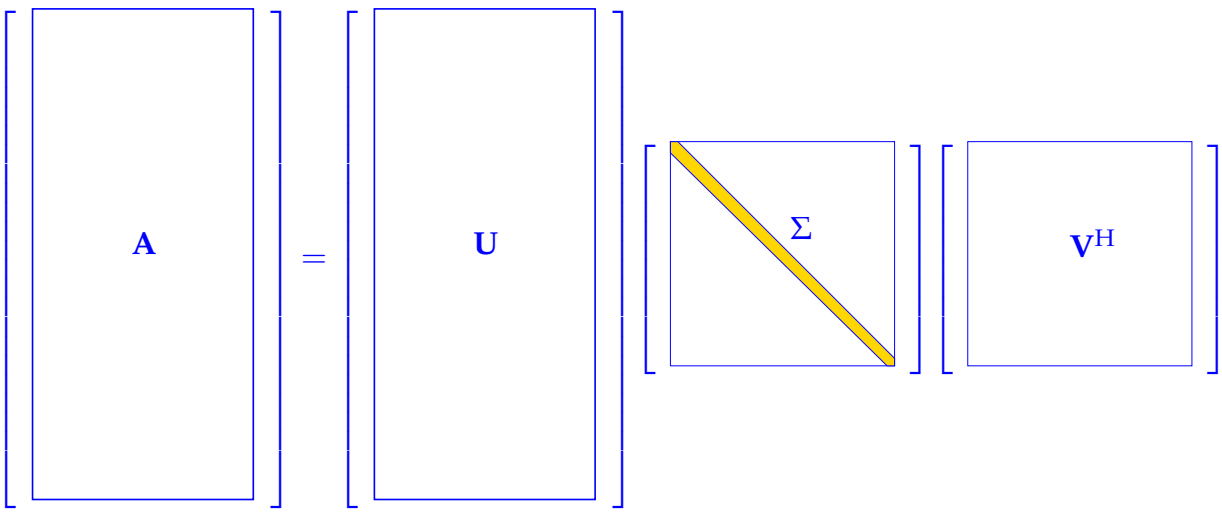
The decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}}$ of Thm. 3.4.1 is called singular value decomposition (SVD) of $\mathbf{A}$. The diagonal entries σ_i of $\mathbf{\Sigma}$ are the singular values of $\mathbf{A}$.

As in the case of the QR-decomposition, compare (3.3.3.1) and (3.3.3.1), we can also drop the bottom zero rows of $\mathbf{\Sigma}$ and the corresponding columns of $\mathbf{U}$ in the case of $m > n$. Thus we end up with an "economical" singular value decomposition of $\mathbf{A} \in \mathbb{K}^{m,n}$:

$$\begin{aligned} m \geq n\colon &: \quad \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}} \,, \quad \mathbf{U} \in \mathbb{K}^{m,n} \,, \quad \mathbf{\Sigma} \in \mathbb{K}^{n,n} \,, \quad \mathbf{V} \in \mathbb{K}^{n,n} \quad , \quad \mathbf{U}^{\mathrm{H}}\mathbf{U} = \mathbf{I}_n \,, \mathbf{V} \text{ unitary} \,, \\ m < n\colon &: \quad \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}} \,, \quad \mathbf{U} \in \mathbb{K}^{m,m} \,, \quad \mathbf{\Sigma} \in \mathbb{K}^{m,m} \,, \quad \mathbf{V} \in \mathbb{K}^{n,m} \quad , \quad \mathbf{U} \text{ unitary} \,, \mathbf{V}^{\mathrm{H}}\mathbf{V} = \mathbf{I}_m \,. \end{aligned} \tag{3.4.4}$$

with true diagonal matrices $\mathbf{\Sigma}$, whose diagonals contain the singular values of $\mathbf{A}$.

Visualization of economimcal SVD for $m > 0$:



Lemma 3.4.5.

The squares σ_i^2 of the non-zero singular values of $\mathbf{A}$ are the non-zero eigenvalues of $\mathbf{A}^H\mathbf{A}$, $\mathbf{A}\mathbf{A}^H$ with associated eigenvectors $(\mathbf{V})_{:,1},\ldots,(\mathbf{V})_{:,p}$, $(\mathbf{U})_{:,1},\ldots,(\mathbf{U})_{:,p}$, respectively.

Proof. $\mathbf{A}\mathbf{A}^H$ and $\mathbf{A}^H\mathbf{A}$ are similar ($\rightarrow$ Lemma 9.1.6) to diagonal matrices with non-zero diagonal entries σ_i^2 ($\sigma_i \neq 0$), e.g.,

$$\mathbf{A}\mathbf{A}^H = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H\mathbf{V}\boldsymbol{\Sigma}^H\mathbf{U}^H = \mathbf{U}\underbrace{\boldsymbol{\Sigma}\boldsymbol{\Sigma}^H}_{\text{diagonal matrix}}\mathbf{U}^H . \qquad \square$$

$\square$

Remark 3.4.6 (SVD and additive rank-1 decomposition $\rightarrow$ [?, Cor. 11.2], [?, Thm. 9.8])

Recall from linear algebra: rank-1 matrices are tensor products of vectors

$$\mathbf{A} \in \mathbb{K}^{m,n} \quad \text{and} \quad \operatorname{rank}(\mathbf{A}) = 1 \quad \Leftrightarrow \quad \exists \mathbf{u} \in \mathbb{K}^m, \mathbf{v} \in \mathbb{K}^n: \quad \mathbf{A} = \mathbf{u}\mathbf{v}^H , \tag{3.4.7}$$

because $\operatorname{rank}(\mathbf{A}) = 1$ means that $\mathbf{A}\mathbf{x} = \mu(\mathbf{x})\mathbf{u}$ for some $\mathbf{u} \in \mathbb{K}^m$ and linear form $\mathbf{x} \mapsto \mu(\mathbf{x})$. By the Riesz representation theorem the latter can be written as $\mu(\mathbf{x}) = \mathbf{v}^H\mathbf{x}$.

Singular value decomposition provides additive decomposition into rank-1 matrices:

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^H = \sum_{j=1}^{p} \sigma_j (\mathbf{U})_{:,j}(\mathbf{V})_{:,j}^H . \tag{3.4.8}$$

Remark 3.4.9 (Uniqueness of SVD)

The SVD from Def. 3.4.3 is not (necessarily) unique, but the singular values are.

Proof. Proof by contradiction: assume that $\mathbf{A}$ has two singular value decompositions

$$\mathbf{A} = \mathbf{U}_1\boldsymbol{\Sigma}_1\mathbf{V}_1^H = \mathbf{U}_2\boldsymbol{\Sigma}_2\mathbf{V}_2^H \quad \Rightarrow \quad \mathbf{U}_1 \underbrace{\boldsymbol{\Sigma}_1\boldsymbol{\Sigma}_1^H}_{=\operatorname{diag}(\sigma_1^2,\ldots,\sigma_m^2)} \mathbf{U}_1^H = \mathbf{A}\mathbf{A}^H = \mathbf{U}_2 \underbrace{\boldsymbol{\Sigma}_2\boldsymbol{\Sigma}_2^H}_{=\operatorname{diag}(\sigma_1^2,\ldots,\sigma_m^2)} \mathbf{U}_2^H .$$

Two similar diagonal matrices with non-increasing diagonal entries are equal !

$\square$

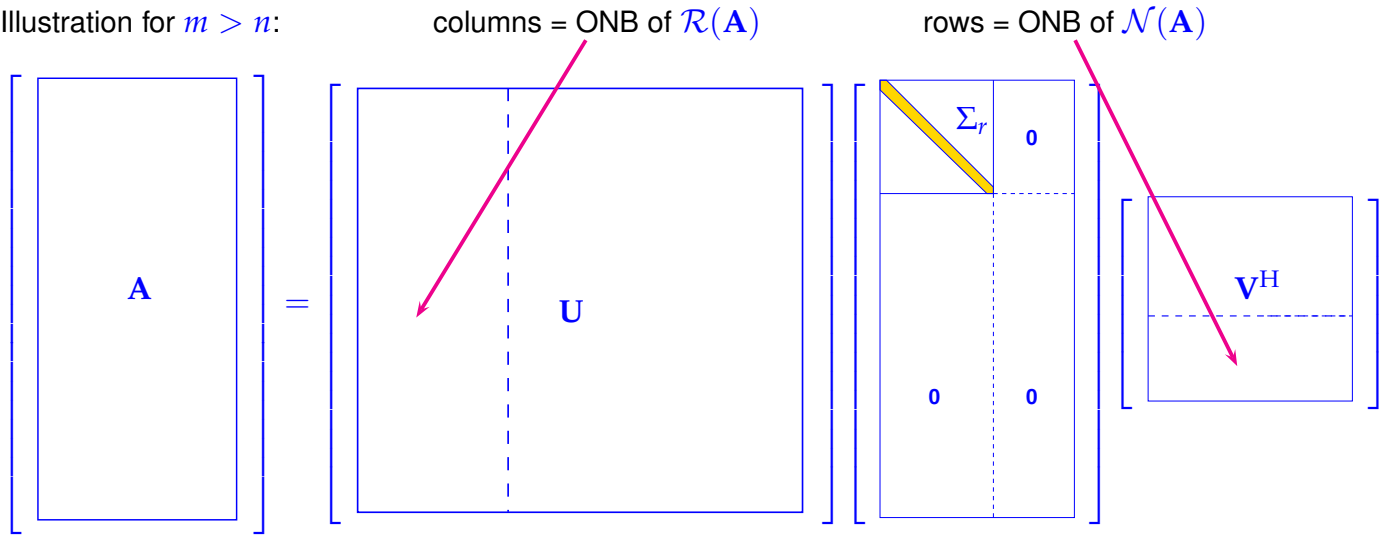
(3.4.10) SVD, nullspace, and image space

Lemma 3.4.11. SVD and rank of a matrix → **[?, Cor. 9.7]**

If, for some $1 \le r \le p := \min\{m,n\}$, *the singular values of* $\mathbf{A} \in \mathbb{K}^{m,n}$ *satisfy* $\sigma_1 \ge \cdots \ge \sigma_r > \sigma_{r+1} = \cdots \sigma_p = 0$, *then*

- $\operatorname{rank}(\mathbf{A}) = r$ *(no. of non-zero singular values)* ,
- $\mathcal{N}(\mathbf{A}) = \operatorname{Span}\{(\mathbf{V})_{:,r+1}, \ldots, (\mathbf{V})_{:,n}\}$,
- $\mathcal{R}(\mathbf{A}) = \operatorname{Span}\{(\mathbf{U})_{:,1}, \ldots, (\mathbf{U})_{:,r}\}$.

Illustration for $m > n$:



(3.4.12)

3.4.2 SVD in EIGEN

The EIGEN class **JacobiSVD** is constructed from a matrix data type, computes the SVD of its argument during construction and offers access methods `MatrixU()`, `singularValues()`, and `MatrixV()` to request the SVD-factors and singular values.

C++-code 3.4.13: Computing SVDs in EIGEN

```
2  # include <Eigen/SVD>
3
4  // Computation of (full) SVD A = UΣV^H → Thm. 3.4.1
5  // SVD factors are returned as dense matrices in natural order
6  std::tuple<MatrixXd,MatrixXd,MatrixXd> svd_full(const MatrixXd& A) {
7    Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeFullU |
       Eigen::ComputeFullV);
8    MatrixXd U = svd.matrixU(); // get unitary (square) matrix U
9    MatrixXd V = svd.matrixV(); // get unitary (square) matrix V
10   VectorXd sv = svd.singularValues(); // get singular values as vector
11   MatrixXd Sigma = MatrixXd::Zero(A.rows(), A.cols());
12   const unsigned p = sv.size(); // no. of singular values
13   Sigma.block(0,0,p,p) = sv.asDiagonal(); // set diagonal block of Σ
14   return std::tuple<MatrixXd,MatrixXd,MatrixXd>(U,Sigma,V);
15 }
```

```
16
17 // Computation of economical (thin) SVD A = UΣV^H, see (3.4.4)
18 // SVD factors are returned as dense matrices in natural order
19 std::tuple<MatrixXd,MatrixXd,MatrixXd> svd_eco(const MatrixXd& A) {
20   Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU |
       Eigen::ComputeThinV);
21   MatrixXd U = svd.matrixU(); // get unitary (square) matrix U
22   MatrixXd V = svd.matrixV(); // get unitary (square) matrix V
23   VectorXd sv = svd.singularValues(); // get singular values as vector
24   MatrixXd Sigma = sv.asDiagonal(); // build diagonal matrix Σ
25   return std::tuple<MatrixXd,MatrixXd,MatrixXd>(U,Sigma,V);
26 }
```

The second argument in the constructor of **JacobiSVD** determines, whether the methods `matrixU()` and `matrixV()` return the factor for the full SVD of Def. 3.4.3 or of the economical (thin) SVD (3.4.4): `Eigen::ComputeFull*` will select the full versions, whereas `Eigen::ComputeThin*` picks the economical versions → documentation.

Internally, the computation of the SVD is done by a sophisticated algorithm, for which key steps rely on orthogonal/unitary transformations. Also there we reap the benefit of the exceptional stability brought about by norm-preserving transformations → § 3.3.28.

EIGEN's algorithm for computing SVD is (numerically) stable → Def. 1.5.85

(3.4.14) Computational cost of computing the SVD

According to EIGEN's documentation the SVD of a general dense matrix involves the following asymptotic complexity:

$$\text{cost(economical SVD of } \mathbf{A} \in \mathbb{K}^{m,n}) = O(\min\{m,n\}^2 \max\{m,n\})$$

The computational effort is (asymptotically) *linear in the larger matrix dimension*.

Example 3.4.15 (SVD-based computation of the rank of a matrix)

Based on Lemma 3.4.11, the SVD is the main tool for the stable computation of the rank of a matrix (→ Def. 2.2.3)

However, theory as reflected in Lemma 3.4.11 entails identifying zero singular values, which must rely on a threshold condition in a numerical code, recall Rem. 1.5.36. Given the SVD $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H$, $\mathbf{\Sigma} = \operatorname{diag}(\sigma_1,\ldots,\sigma_{\min\{m,n\}})$, of a matrix $\mathbf{A} \in \mathbb{K}^{m,n}$, $\mathbf{A} \neq \mathbf{0}$ and a tolerance $\mathrm{tol} > 0$, we define the numerical rank

$$r := \sharp\left\{\sigma_i : |\sigma_i| \geq \mathrm{tol} \max_j\{|\sigma_j|\}\right\}. \tag{3.4.16}$$

The following code implements this rule.

C++-code 3.4.17: Computing rank of a matrix through SVD

```
// Computation of the numerical rank of a non-zero matrix by means of
// singular value decomposition, cf. (3.4.16).
MatrixXd::Index rank_by_svd(const MatrixXd &A, double tol = EPS) {
  if (A.norm() == 0) return MatrixXd::Index(0);
  Eigen::JacobiSVD<MatrixXd> svd(A);
  const VectorXd sv = svd.singularValues(); // Get sorted singular
     values as vector
  MatrixXd::Index n = sv.size();
  MatrixXd::Index r = 0;
  // Test relative size of singular values
  while ((r<n) && (sv(r) >= sv(0)*tol)) r++;
  return r;
}
```

EIGEN offers an equivalent built-in method `rank()` for objects representing singular value decompositions:

C++-code 3.4.18: Using `rank()` in EIGEN

```
// Computation of the numerical rank of a matrix by means of SVD
MatrixXd::Index rank_eigen(const MatrixXd &A, double tol = EPS) {
  return A.jacobiSvd().setThreshold(tol).rank();
}
```

The method `setThreshold()` passes tol from (3.4.16) to `rank()`.

Example 3.4.19 (Computation of nullspace and image space of matrices)

"Computing" a subspace of $\mathbb{R}^k$ amounts to making available a (stable) *basis* of that subspace, ideally an *orthonormal basis*.

Lemma 3.4.11 taught us how to glean orthonormal bases of $\mathcal{N}(\mathbf{A})$ and $\mathcal{R}(\mathbf{A})$ from the SVD of a matrix $\mathbf{A}$. This immediately gives a numerical method and its implementation is given in the next two codes.

C++-code 3.4.20: ONB of $\mathcal{N}(\mathbf{A})$ through SVD

```
// Computation of an ONB of the kernel of a matrix
MatrixXd nullspace(const MatrixXd &A, double tol = EPS) {
  using index_t = MatrixXd::Index;
  Eigen::JacobiSVD<MatrixXd> svd(A,Eigen::ComputeFullV);
  index_t r = svd.setThreshold(tol).rank();
  // Rightmost columns of V provide ONB of N(A)
  MatrixXd Z = svd.matrixV().rightCols(A.cols()-r);
  return Z;
}
```

C++-code 3.4.21: ONB of $\mathcal{R}(\mathbf{A})$ through SVD

```
// Computation of an ONB of the image space of a matrix
MatrixXd rangespace(const MatrixXd &A, double tol = EPS) {
  using index_t = MatrixXd::Index;
  Eigen::JacobiSVD<MatrixXd> svd(A,Eigen::ComputeFullV);
  index_t r = svd.setThreshold(tol).rank();
  // r left columns of U provide ONB of R(A)
  return svd.matrixV().leftCols(r);
}
```

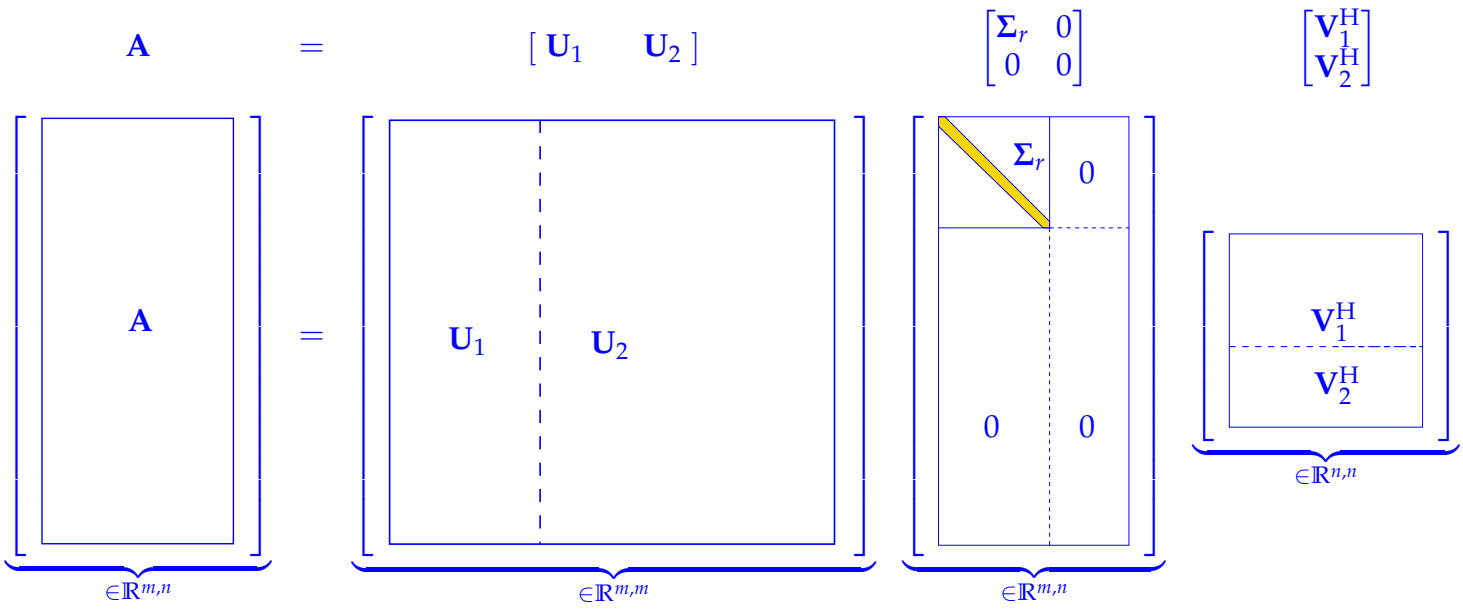
3.4.3 Generalized Solutions of LSE by SVD

In a similar fashion as explained for QR-decomposition in Section 3.3.4, the singular valued decompisition (SVD, → Def. 3.4.3) can be used to transform general linear least squares problems (3.1.38) into a simpler form. In the case of SVD based orthogonal transformation method based on SVD this simpler form involves merely a diagonal matrix.

Here we consider the most general setting: $\mathbf{A} \in \mathbb{K}^{m,n}$, $\operatorname{rank}(\mathbf{A}) = r \leq \min\{m,n\}$, *cf.* (3.1.38). In particular, we we drop the assumption of full rank of $\mathbf{A}$. This means that the minimum norm condition (ii) in the definition (3.1.38) of a linear least squares problem may be required for singling out a unique solution.

We recal the SVD of $\mathbf{A} \in \mathbb{R}^{m,n}$:

$$\mathbf{A} = \begin{bmatrix} \mathbf{U}_1 & \mathbf{U}_2 \end{bmatrix} \begin{bmatrix} \mathbf{\Sigma}_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^{\mathrm{H}} \\ \mathbf{V}_2^{\mathrm{H}} \end{bmatrix}$$



(3.4.22)

with $\mathbf{U}_1 \in \mathbb{R}^{m,r}$, $\mathbf{U}_2 \in \mathbb{R}^{m,m-r}$ with *orthonormal* columns,
$\mathbf{\Sigma}_r = \operatorname{diag}(\sigma_1, \ldots, \sigma_r) \in \mathbb{R}^{r,r}$ (singular values, Def. 3.4.3),
$\mathbf{V}_1 \in \mathbb{R}^{n,r}$, $\mathbf{V}_2 \in \mathbb{R}^{n,n-r}$ with *orthonormal* columns.

Then we use the invariance of the 2-norm of a vector with respect to multiplication with $\mathbf{U} = [\mathbf{U}_1, \mathbf{U}_2]$, see Thm. 3.3.5, together with the fact that $\mathbf{U}$ is unitary, see Def. 6.2.2:

$$[\mathbf{U}_1, \mathbf{U}_2] \cdot \begin{bmatrix} \begin{pmatrix} \mathbf{U}_1^{\mathrm{H}} \\ \mathbf{U}_2^{\mathrm{H}} \end{pmatrix} \end{bmatrix} = \mathbf{I} \,.$$

$$\blacktriangleright \quad \|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2 = \left\| [\mathbf{U}_1\ \mathbf{U}_2] \begin{bmatrix} \mathbf{\Sigma}_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^{\mathrm{H}} \\ \mathbf{V}_2^{\mathrm{H}} \end{bmatrix} \mathbf{x} - \mathbf{b} \right\|_2 = \left\| \begin{bmatrix} \mathbf{\Sigma}_r \mathbf{V}_1^{\mathrm{H}} \mathbf{x} \\ 0 \end{bmatrix} - \begin{bmatrix} \mathbf{U}_1^{\mathrm{H}} \mathbf{b} \\ \mathbf{U}_2^{\mathrm{H}} \mathbf{b} \end{bmatrix} \right\|_2 \tag{3.4.23}$$

We follow the same strategy as in the case of QR-based solvers for full-rank linear least squares problems. We choose $\mathbf{x}$ such that the first r components of $\begin{bmatrix} \mathbf{\Sigma}_r \mathbf{V}_1^{\mathrm{H}} \mathbf{x} \\ 0 \end{bmatrix} - \begin{bmatrix} \mathbf{U}_1^{\mathrm{H}} \mathbf{b} \\ \mathbf{U}_2^{\mathrm{H}} \mathbf{b} \end{bmatrix}$ vanish:

$$\succ \quad \text{(possibly } \textit{underdetermined}\text{) } r \times n \text{ linear system} \qquad \mathbf{\Sigma}_r \mathbf{V}_1^{\mathrm{H}} \mathbf{x} = \mathbf{U}_1^{\mathrm{H}} \mathbf{b} \ . \tag{3.4.24}$$

To fix a unique solution in the case $r < n$ we appeal to the minimal norm condition in (3.1.38): appealing to the considerations of § 3.1.34, the solution $\mathbf{x}$ of (3.4.24) is unique up to contributions from

$$\mathcal{N}(\mathbf{V}_1^{\mathrm{H}}) \overset{\text{Lemma 3.1.21}}{=} \mathcal{R}(\mathbf{V}_1)^{\perp} \overset{\text{orthonormality}}{=} \mathcal{R}(\mathbf{V}_2) \ . \tag{3.4.25}$$

Since $\mathbf{V}$ is unitary, the minimal norm solution is obtained by setting contributions from $\mathcal{R}(\mathbf{V}_2)$ to zero, which amounts to choosing $\mathbf{x} \in \mathcal{R}(\mathbf{V}_1)$. This converts (3.4.24) into

$$\mathbf{\Sigma}_r \underbrace{\mathbf{V}_1^{\mathrm{H}} \mathbf{V}_1}_{=\mathbf{I}} \mathbf{z} = \mathbf{U}_1^{\mathrm{H}} \mathbf{b} \quad \Rightarrow \quad \mathbf{z} = \mathbf{\Sigma}_r^{-1} \mathbf{U}_1^{\mathrm{H}} \mathbf{b} \ .$$

$$\blacktriangleright \quad \text{generalized solution} \rightarrow \text{Def. 3.1.32} \quad \boxed{\mathbf{x}^{+} = \mathbf{V}_1 \mathbf{\Sigma}_r^{-1} \mathbf{U}_1^{\mathrm{H}} \mathbf{b}} \quad , \quad \|\mathbf{r}\|_2 = \left\| \mathbf{U}_2^{\mathrm{H}} \mathbf{b} \right\|_2 \ . \tag{3.4.26}$$

In a practical implementation, as in Code 3.4.17, we have to resort to the numerical rank from (3.4.16):

$$r = \max\{i \colon \sigma_i / \sigma_1 > \texttt{tol}\} \ ,$$

where we have assumed that the singular values σ_j are sorted according to decreasing modulus.

C++-code 3.4.27: Computing generalized solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ via SVD

```
2  #include <Eigen/SVD>
3
4  VectorXd lsqsvd(const MatrixXd& A, const VectorXd& b) {
5    Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU |
       Eigen::ComputeThinV);
6    VectorXd sv = svd.singularValues();
7    unsigned r = svd.rank(); // No. of (numerically!) nonzero singular
       values
8    MatrixXd U = svd.matrixU(), V = svd.matrixV();
9
10   return V.leftCols(r)*( sv.head(r).cwiseInverse().asDiagonal() *
       (U.leftCols(r).adjoint()*b) );
11 }
```

The `solve()` method directly returns the generalized solution

C++-code 3.4.28: Computing generalized solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ via SVD

```
2  VectorXd lsqsvd_eigen(const MatrixXd& A, const VectorXd& b) {
3    Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinU |
```

```
    Eigen::ComputeThinV);
  return svd.solve(b);
}
```

Remark 3.4.29 (Pseudoinverse and SVD → [?, Ch. 12], [?, Sect. 4.7])

From Thm. 3.1.37 we could conclude a general formula for the Moore-Penrose pseudoinverse of any matrix $\mathbf{A} \in \mathbb{R}^{m,n}$. Now, the solution formula (3.4.26) directly yields a concrete incarnation of the pseudoinverse $\mathbf{A}^+$.

Theorem 3.4.30. Pseudoinverse and SVD

If $\mathbf{A} \in \mathbb{K}^{m,n}$ has the SVD decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}}$ partitioned as in (3.4.22)*, then its Moore-Penrose pseudoinverse (→ Thm. 3.1.37)is given by $\mathbf{A}^{\dagger} = \mathbf{V}_1\mathbf{\Sigma}_r^{-1}\mathbf{U}_1^{\mathrm{H}}$.*

3.4.4 SVD-Based Optimization and Approximation

For the general least squares problem (3.1.38) we have seen the use of SVD for its numerical solution in Section 3.4.3. The the SVD was a powerful tool for solving a minimization problem for a 2-norm. In many other contexts the SVD is also a key component in numerical optimization.

3.4.4.1 Norm-Constrained Extrema of Quadratic Forms

We consider the following problem of finding the extrema of quadratic forms on the Euclidean unit sphere $\{\mathbf{x} \in \mathbb{K}^n : \|\mathbf{x}\|_2 = 1\}$:

$$\text{given} \quad \mathbf{A} \in \mathbb{K}^{m,n}\,,\ m \geq n, \quad \text{find } \mathbf{x} \in \mathbb{K}^n,\ \|\mathbf{x}\|_2 = 1\,, \quad \|\mathbf{A}\mathbf{x}\|_2 \to \min\,. \tag{3.4.31}$$

Use that multiplication with orthogonal/unitary matrices preserves the 2-norm (→ Thm. 3.3.5) and resort to the singular value decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}}$ (→ Def. 3.4.3):

$$\begin{aligned}\min_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2^2 &= \min_{\|\mathbf{x}\|_2=1} \left\|\mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}}\mathbf{x}\right\|_2^2 = \min_{\|\mathbf{V}^{\mathrm{H}}\mathbf{x}\|_2=1} \left\|\mathbf{U}\mathbf{\Sigma}(\mathbf{V}^{\mathrm{H}}\mathbf{x})\right\|_2^2 \\ [\,\mathbf{y} = \mathbf{V}^{\mathrm{H}}\mathbf{x}\,] \quad &= \min_{\|\mathbf{y}\|_2=1} \|\mathbf{\Sigma}\mathbf{y}\|_2^2 = \min_{\|\mathbf{y}\|_2=1} (\sigma_1^2 y_1^2 + \cdots + \sigma_n^2 y_n^2) \geq \sigma_n^2\,.\end{aligned}$$

Since the singular values are assumed to be sorted as $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n$, the minimum with value σ_n^2 is attained for $\mathbf{V}^{\mathrm{H}}\mathbf{x} = \mathbf{y} = \mathbf{e}_n \quad \Rightarrow \quad$ minimizer $\mathbf{x} = \mathbf{V}\mathbf{e}_n = (\mathbf{V})_{:,n}$.

C++11 code 3.4.32: Solving (3.4.31) **with EIGEN ⇢ GITLAB**

```
// EIGEN based function for solving (3.4.31);
// minimizer returned nin x, mininum as return value
double minconst(VectorXd &x,const MatrixXd &A) {
  MatrixXd::Index m=A.rows(),n=A.cols();
  if (m < n) throw std::runtime_error("A must be tall matrix");
```

```
  // SVD factor U is not computed!
  Eigen::JacobiSVD<MatrixXd> svd(A, Eigen::ComputeThinV);
  x.resize(n); x.setZero(); x(n-1) = 1.0; // e_n
  x = svd.matrixV()*x;
  return (svd.singularValues())(n-1);
}
```

By similar arguments we can solve the corresponding norm constrained maximization problem

$$\text{given} \quad \mathbf{A} \in \mathbb{K}^{m,n}\,,\ m \geq n, \quad \text{find } \mathbf{x} \in \mathbb{K}^n,\ \|\mathbf{x}\|_2 = 1\,, \quad \|\mathbf{A}\mathbf{x}\|_2 \to \max\,,$$

and obtain the solution based on the SVD $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}}$ of $\mathbf{A}$:

$$\sigma_1 = \max_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2 \quad , \quad (\mathbf{V})_{:,1} = \operatorname*{argmax}_{\|\mathbf{x}\|_2=1} \|\mathbf{A}\mathbf{x}\|_2\,. \tag{3.4.33}$$

Recall: The Euclidean matrix norm (2-norm) of the matrix $\mathbf{A}$ ($\to$ Def. 1.5.76) is defined as the maximum in (3.4.33). Thus we have proved the following theorem:

Lemma 3.4.34. SVD and Euclidean matrix norm

If $\mathbf{A} \in \mathbb{K}^{m,n}$ has singular values $\sigma_1 \geq \sigma_2 \geq \cdots \geq \min\{m,n\}$, then its Euclidean matrix norm is given by $\|\mathbf{A}\|_2 = \sigma_1(\mathbf{A})$
If $m = n$ and $\mathbf{A}$ is regular/invertible, then its 2-norm condition number is $\operatorname{cond}_2(\mathbf{A}) = \sigma_1/\sigma_n$.

Example 3.4.35 (Fit of hyperplanes)

For an important application from computational geometry, this example studies the power and versatility of orthogonal transformations in the context of (generalized) least squares minimization problems.

From school recall the Hesse normal form of a hyperplane $\mathcal{H}$ (= affine subspace of dimension $d-1$) in $\mathbb{R}^d$:

$$\mathcal{H} = \{\mathbf{x} \in \mathbb{R}^d \colon c + \mathbf{n}^\top \mathbf{x} = 0\}\,, \quad \|\mathbf{n}\|_2 = 1\,. \tag{3.4.36}$$

where $\mathbf{n}$ is the unit normal to $\mathcal{H}$ and $|c|$ gives the distance of $\mathcal{H}$ from $\mathbf{0}$. The Hesse normal form is convenient for computing the distance of points from $\mathcal{H}$, because the

$$\text{Euclidean distance of } \mathbf{y} \in \mathbb{R}^d \text{ from the plane is} \quad \operatorname{dist}(\mathcal{H}, \mathbf{y}) = |c + \mathbf{n}^\top \mathbf{y}|\,, \tag{3.4.37}$$

Goal: given the points $\mathbf{y}_1, \ldots, \mathbf{y}_m$, $m > d$, find $\mathcal{H} \leftrightarrow \{c \in \mathbb{R}, \mathbf{n} \in \mathbb{R}^d, \|\mathbf{n}\|_2 = 1\}$, such that

$$\sum_{j=1}^m \operatorname{dist}(\mathcal{H}, \mathbf{y}_j)^2 = \sum_{j=1}^m |c + \mathbf{n}^\top \mathbf{y}_j|^2 \to \min\,. \tag{3.4.38}$$

Note that (3.4.38) is **not** a linear least squares problem due to the constraint $\|\mathbf{n}\|_2 = 1$. However, it turns out to be a minimization problem with *almost* the structure of (3.4.31):

$$(3.4.38) \quad \Leftrightarrow \quad \left\| \underbrace{\begin{bmatrix} 1 & y_{1,1} & \cdots & y_{1,d} \\ 1 & y_{2,1} & \cdots & y_{2,d} \\ \vdots & \vdots & & \vdots \\ 1 & y_{m,1} & \cdots & y_{m,d} \end{bmatrix}}_{=:\mathbf{A}} \begin{bmatrix} c \\ n_1 \\ \vdots \\ n_d \end{bmatrix} \right\|_2 \to \min \quad \text{under constraint} \quad \|\mathbf{n}\|_2 = 1\,.$$

Step ❶: To convert the minimization problem into the form (3.4.31) we start with a QR-decomposition (→ Section 3.3.3)

$$\mathbf{A} := \begin{bmatrix} 1 & y_{1,1} & \cdots & y_{1,d} \\ 1 & y_{2,1} & \cdots & y_{2,d} \\ \vdots & \vdots & & \vdots \\ 1 & y_{m,1} & \cdots & y_{m,d} \end{bmatrix} = \mathbf{QR} \quad , \quad \mathbf{R} := \begin{bmatrix} r_{11} & r_{12} & \cdots & \cdots & r_{1,d+1} \\ 0 & r_{22} & \cdots & \cdots & r_{2,d+1} \\ \vdots & & \ddots & & \vdots \\ 0 & & & & r_{d+1,d+1} \\ 0 & \cdots & & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & & \cdots & 0 \end{bmatrix} \in \mathbb{R}^{m,d+1} .$$

$$\|\mathbf{Ax}\|_2 \to \min \quad \Leftrightarrow \quad \|\mathbf{Rx}\|_2 = \left\| \begin{bmatrix} r_{11} & r_{12} & \cdots & \cdots & r_{1,d+1} \\ 0 & r_{22} & \cdots & \cdots & r_{2,d+1} \\ \vdots & & \ddots & & \vdots \\ 0 & & & & r_{d+1,d+1} \\ 0 & \cdots & & \cdots & 0 \\ \vdots & & & & \vdots \\ 0 & \cdots & & \cdots & 0 \end{bmatrix} \begin{bmatrix} c \\ n_1 \\ \vdots \\ \vdots \\ n_d \end{bmatrix} \right\|_2 \to \min . \tag{3.4.39}$$

Step ❷ Note that, if $\mathbf{n}$ is the solution of (3.4.38), then necessarily (why?)

$$c \cdot r_{11} + n_1 \cdot r_{12} + \cdots + r_{1,d+1} \cdot n_d = 0 .$$

This insight converts (3.4.39) to

$$\left\| \begin{bmatrix} r_{22} & r_{23} & \cdots & \cdots & r_{2,d+1} \\ 0 & r_{33} & \cdots & \cdots & r_{3,d+1} \\ \vdots & & \ddots & & \vdots \\ 0 & & & & r_{d+1,d+1} \end{bmatrix} \begin{bmatrix} n_1 \\ \vdots \\ \vdots \\ n_d \end{bmatrix} \right\|_2 \to \min \quad , \quad \|\mathbf{n}\|_2 = 1 . \tag{3.4.40}$$

(3.4.40) = problem of type (3.4.31), minimization on the Euclidean sphere.

➢ Solve (3.4.40) using SVD !

Note: Since $r_{11} = \|(\mathbf{A})_{:,1}\|_2 = \sqrt{m} \neq 0 \quad \Rightarrow \quad c = -r_{11}^{-1} \sum_{j=1}^{d} r_{1,j+1} n_j$.

This algorithm is implemented in the following code, making heavy use of EIGEN's block access operations and the built in QR-decomposition and SVD factorization.

C++-code 3.4.41: (Generalized) distance fitting of a hyperplane: solution of (3.4.42)

```
// Fitting of a hyperplane, returning Hesse normal form
void clsq(const MatrixXd& A, const unsigned dim,
          double& c, VectorXd& n) {
  unsigned p = A.cols(), m = A.rows();
  if (p < dim + 1) { cerr << "not enough unknowns\n"; return; }
  if (m < dim)  { cerr << "not enough equations\n"; return; }

  m = std::min(m, p);
```

```
10    // First step:  orthogonal transformation, see Code 3.3.38
11    MatrixXd R = A.householderQr().matrixQR().template
          triangularView<Eigen::Upper>();
12    // compute matrix V from SVD composition of R, solve (3.4.40)
13    MatrixXd V = R.block(p - dim, p - dim, m + dim - p, dim)
14                  .jacobiSvd(Eigen::ComputeFullV).matrixV();
15    n = V.col(dim - 1);
16
17    MatrixXd R_topleft = R.topLeftCorner(p - dim, p - dim);
18    c = -(R_topleft.template triangularView<Eigen::Upper>()
19          .solve(R.block(0, p - dim, p - dim, dim)) * n)(0);
20  }
```

Code 3.4.41 solves the general problem: For $\mathbf{A} \in \mathbb{K}^{m,n}$ find $\mathbf{n} \in \mathbb{R}^d$, $\mathbf{c} \in \mathbb{R}^{n-d}$ such that

$$\left\| \mathbf{A} \begin{bmatrix} \mathbf{c} \\ \mathbf{n} \end{bmatrix} \right\|_2 \to \min \qquad \text{with constraint} \|\mathbf{n}\|_2 = 1 \,. \tag{3.4.42}$$

3.4.4.2 Best Low-Rank Approximation

(3.4.43) Low-Rank matrix compression

Matrix compression addresses the problem of approximating a given "generic" matrix (of a certain class) by means of matrix, whose "information content", that is, the number of reals needed to store it, is significantly lower than the information content of the original matrix.

Sparse matrices (→ Notion 2.7.1) are a prominent class of matrices with "low information content". Unfortunately, they cannot approximate dense matrices very well. Another type of matrices that enjoy "low information content", also called data sparse, are low-rank matrices.

Lemma 3.4.44.

If $\mathbf{A} \in \mathbb{R}^{m,n}$ has rank $p \le \min\{m,n\}$ (→ Def. 2.2.3), then there exist $\mathbf{U} \in \mathbb{R}^{m,p}$ and $\mathbf{V} \in \mathbb{R}^{n,p}$, such that $\mathbf{A} = \mathbf{U}\mathbf{V}^\top$.

None of the columns of $\mathbf{U}$ and $\mathbf{V}$ can vanish. Hence, in addition, we may assume that the columns of $\mathbf{U}$ are normalized: $\|(\mathbf{U})_{:,j}\|_2 = 1$, $j = 1, \ldots, p$.

It takes only $p(m+n-p)$ real numbers to store $\mathbf{A} \in \mathbb{R}^{m,n}$ with $\operatorname{rank}(\mathbf{A}) = p$.

Thus approximating a given matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ with a rank-p matrix, $p \ll \min\{m,n\}$, can be regarded as an instance of matrix compression. The approximation error with respect to some matrix norm $\|\cdot\|$ will be minimal if we choose the best approximation

$$\mathbf{A}_p := \operatorname{argmin}\{\|\mathbf{A} - \mathbf{B}\| : \ \mathbf{B} \in \mathbb{R}^{m,n}, \ \operatorname{rank}(\mathbf{B}) = p\} \,. \tag{3.4.45}$$

Here we explore low-rank best approximation of general matrices with respect to the Euclidean matrix norm $\|\cdot\|_2$ induced by the 2-norm for vectors (→ Def. 1.5.76), and the Frobenius norm $\|\cdot\|_F$.

Definition 3.4.46. Frobenius norm

The Frobenius norm of $\mathbf{A} \in \mathbb{K}^{m,n}$ is defined as

$$\|\mathbf{A}\|_F^2 := \sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2 .$$

It should be obvious that $\|\mathbf{A}\|_F$ invariant under orthogonal/unitary transformations of $\mathbf{A}$. Thus the Frobenius norm of a matrix $\mathbf{A}$, $\operatorname{rank}(\mathbf{A}) = p$, can be expressed through its singular values σ_j:

Frobenius norm and SVD: $$\boxed{\|\mathbf{A}\|_F^2 = \sum_{j=1}^{p} \sigma_j^2} \tag{3.4.47}$$

✎ notation: $\mathcal{R}_k(m,n) := \{\mathbf{A} \in \mathbb{K}^{m,n}: \operatorname{rank}(\mathbf{A}) \le k\}$, $m,n,k \in \mathbb{N}$

The next profound result links best approximation in $\mathcal{R}_k(m,n)$ and the singular value decomposition (→ Def. 3.4.3).

Theorem 3.4.48. best low rank approximation → **[?, Thm. 11.6]**

Let $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^{\mathrm{H}}$ be the SVD of $\mathbf{A} \in \mathbb{K}^{m,n}$ (→ Thm. 3.4.1). For $1 \le k \le \operatorname{rank}(\mathbf{A})$ set $\mathbf{U}_k := [\mathbf{u}_{:,1},\ldots,\mathbf{u}_{:,k}] \in \mathbb{K}^{m,k}$, $\mathbf{V}_k := [\mathbf{v}_{:,1},\ldots,\mathbf{v}_{:,k}] \in \mathbb{K}^{n,k}$, $\mathbf{\Sigma}_k := \operatorname{diag}(\sigma_1,\ldots,\sigma_k) \in \mathbb{K}^{k,k}$. Then, for $\|\cdot\| = \|\cdot\|_F$ and $\|\cdot\| = \|\cdot\|_2$, holds true

$$\left\|\mathbf{A} - \mathbf{U}_k\Sigma_k\mathbf{V}_k^{\mathrm{H}}\right\| \le \|\mathbf{A} - \mathbf{F}\| \quad \forall \mathbf{F} \in \mathcal{R}_k(m,n) .$$

This theorem teaches that the rank-k-matrix that is *closest* to $\mathbf{A}$ (rank-k best approximation) in both the Euclidean matrix norm and the Frobenniusnorm (→ Def. 3.4.46) can be obtained by truncating the rank-1 sum expansion (3.4.8) obtained from the SVD of $\mathbf{A}$ after k terms.

Proof. (of Thm. 3.4.48) Write $\mathbf{A}_k = \mathbf{U}_k\Sigma_k\mathbf{V}_k^{\mathrm{H}}$. Obviously, with $r = \operatorname{rank}\mathbf{A}$,

$$\operatorname{rank}\mathbf{A}_k = k \quad \text{and} \quad \|\mathbf{A} - \mathbf{A}_k\| = \|\mathbf{\Sigma} - \mathbf{\Sigma}_k\| = \begin{cases} \sigma_{k+1} & \text{, for } \|\cdot\| = \|\cdot\|_2 , \\ \sqrt{\sigma_{k+1}^2 + \cdots + \sigma_r^2} & \text{, for } \|\cdot\| = \|\cdot\|_F . \end{cases}$$

❶ Pick $\mathbf{B} \in \mathbb{K}^{n,n}$, $\operatorname{rank}\mathbf{B} = k$.

► $\dim \mathcal{N}(\mathbf{B}) = n - k \;\Rightarrow\; \mathcal{N}(\mathbf{B}) \cap \operatorname{Span}\{\mathbf{v}_1,\ldots,\mathbf{v}_{k+1}\} \neq \{0\}$,

where $\mathbf{v}_i$, $i = 1,\ldots,n$ are the columns of $\mathbf{V}$. For $\mathbf{x} \in \mathcal{N}(\mathbf{B}) \cap \operatorname{Span}\{\mathbf{v}_1,\ldots,\mathbf{v}_{k+1}\}$, $\|\mathbf{x}\|_2 = 1$

$$\mathbf{x} = \sum_{j=1}^{k+1} (\mathbf{v}_j^{\mathrm{H}}\mathbf{x})\mathbf{v}_j ,$$

$$\|\mathbf{A}-\mathbf{B}\|_2^2 \geq \|(\mathbf{A}-\mathbf{B})\mathbf{x}\|_2^2 = \|\mathbf{A}\mathbf{x}\|_2^2 = \left\|\sum_{j=1}^{k+1}\sigma_j(\mathbf{v}_j^{\mathrm{H}}\mathbf{x})\mathbf{u}_j\right\|_2^2 = \sum_{j=1}^{k+1}\sigma_j^2(\mathbf{v}_j^{\mathrm{H}}\mathbf{x})^2 \geq \sigma_{j+1}^2 ,$$

because $\sum_{j=1}^{k+1}(\mathbf{v}_j^{\mathrm{H}}\mathbf{x})^2 = \|\mathbf{x}\|_2^2 = 1$.

❷ Find ONB $\{\mathbf{z}_1,\ldots,\mathbf{z}_{n-k}\}$ of $\mathcal{N}(\mathbf{B})$ and assemble it into a matrix $\mathbf{Z} = [\mathbf{z}_1 \ldots \mathbf{z}_{n-k}] \in \mathbb{K}^{n,n-k}$

$$\|\mathbf{A}-\mathbf{B}\|_F^2 \geq \|(\mathbf{A}-\mathbf{B})\mathbf{Z}\|_F^2 = \|\mathbf{A}\mathbf{Z}\|_F^2 = \sum_{i=1}^{n-k}\|\mathbf{A}\mathbf{z}_i\|_2^2 = \sum_{i=1}^{n-k}\sum_{j=1}^{r}\sigma_j^2(\mathbf{v}_j^{\mathrm{H}}\mathbf{z}_i)^2 \qquad \square$$

$\square$

Since the matrix norms $\|\cdot\|_2$ and $\|\cdot\|_F$ are invariant under multiplication with orthogonal (unitary) matrices, we immediately obtain expressions for the norms of the best approximation error:

$$\left\|\mathbf{A}-\mathbf{U}_k\Sigma_k\mathbf{V}_k^{\mathrm{H}}\right\|_2 = \sigma_{k+1} , \tag{3.4.49}$$

$$\left\|\mathbf{A}-\mathbf{U}_k\Sigma_k\mathbf{V}_k^{\mathrm{H}}\right\|_F^2 = \sum_{j=k+1}^{\min\{m,n\}}\sigma_j^2 . \tag{3.4.50}$$

This provides precise information about the best approximation error for rank-k matrices.

Example 3.4.51 (Image compression)

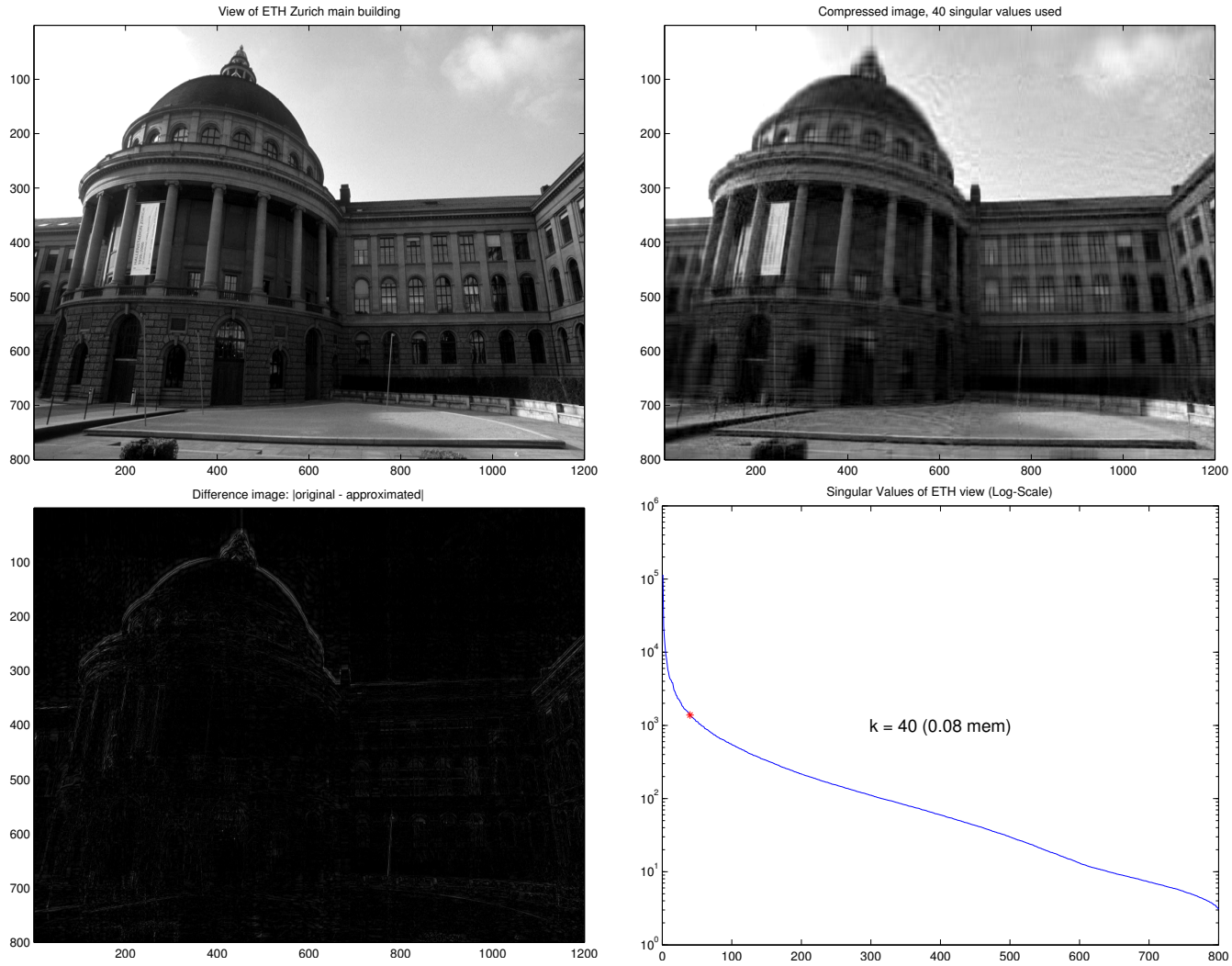
A rectangular greyscale image composed of $m \times n$ pixels (greyscale, BMP format) can be regarded as a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $a_{ij} \in \{0,\ldots,255\}$, *cf.* Ex. 9.3.24. Thus low-rank approximation of the image matrix is a way to compress the image.

 Thm. 3.4.48 ➢ best rank-k approximation of image: $\widetilde{\mathbf{A}} = \mathbf{U}_k\Sigma_k\mathbf{V}^{\top}$

Of course, the matrices $\mathbf{U}_l$, $\mathbf{V}_k$, and Σ_k are available from the economical (thin) SVD (3.4.4) of $\mathbf{A}$. This is the idea behind the following MATLAB code (This example is coded in MATLAB, because MATHGL lacks image rendering capabilities.).

MATLAB-code 3.4.52: Image compression

```
[img_data] = imread(<file>.jpg'); % read image from file
img_data = double(img_data);
image(img_data); % render image
[U,S,V] = svd(img_data(:,:,1));
s = diag(S);
k=20;  % rank of compressed image
img_data_comp = U(:,1:k)*diag(s(1:k))*V(:,1:k)';
image(img_data_comp); % render compressed image
col = [0:1/215:1]'*[1,1,1];
colormap(col);
```



The sample images were rendered by the following code.

MATLAB-code 3.4.53: SVD based image compression

```
P = double(imread('eth.pbm'));
[m,n] = size(P); [U,S,V] = svd(P); s = diag(S);
k = 40; S(k+1:end,k+1:end) = 0; PC = U*S*V';

figure('position',[0 0 1600 1200]); col = [0:1/215:1]'*[1,1,1];
subplot(2,2,1); image(P); title('original image'); colormap(col);
subplot(2,2,2); image(PC); title('compressed (40 S.V.)');
   colormap(col);
subplot(2,2,3); image(abs(P-PC)); title('difference');
   colormap(col);
subplot(2,2,4); cla; semilogy(s); hold on; plot(k,s(k),'ro');
```

Note that there are better and faster ways to compress images than SVD (JPEG, Wavelets, etc.)

3.4.4.3 Principal Component Data Analysis (PCA)

Example 3.4.54 (Trend analysis)

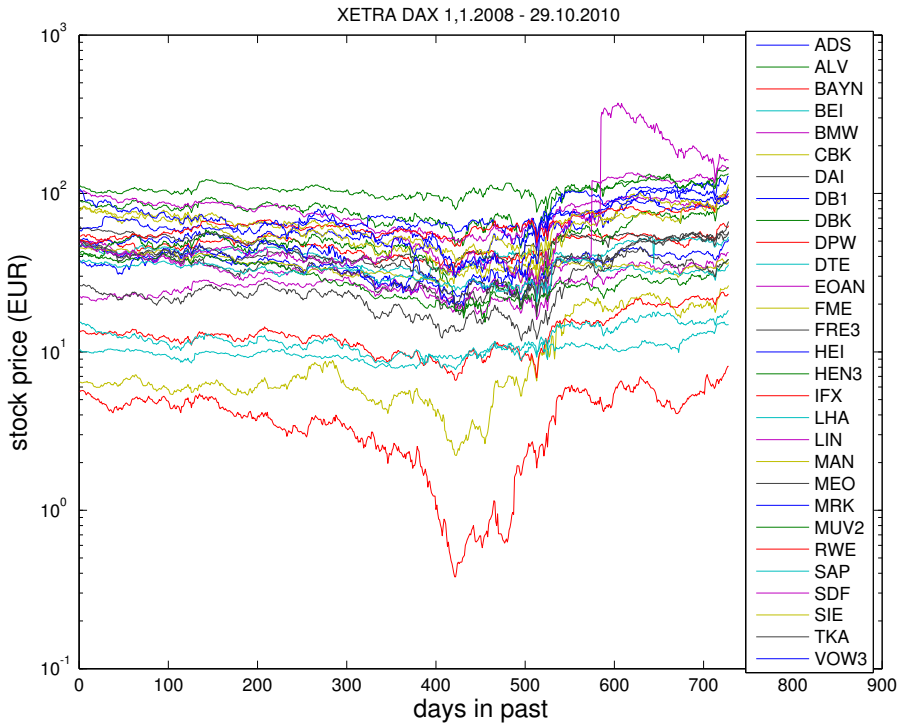


Fig. 106

◁ (end of day) stock prizes $\hat{=}$ n data vectors $\in \mathbb{R}^m$

Are there underlying governing trends **?**

Are there a few vectors $\mathbf{u}_1, \ldots, \mathbf{u}_p$, $p \ll n$, such that, *approximately* all other data vectors $\in \mathrm{Span}\{\mathbf{u}_1, \ldots, \mathbf{u}_p\}$.

Example 3.4.55 (Classification from measured data)

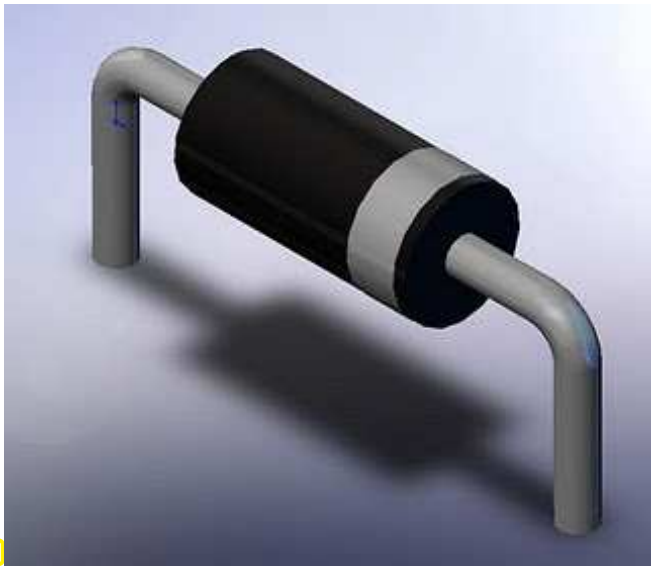


Fig. 107

Given: measured U-I characteristics of n diodes in a box
(data points $(U_j, I_j^{(k)})$, $j = 1, \ldots, m$, $k = 1, \ldots, n$)

Classification problem: find out

- how many different types of diodes in box,
- the U-I characteristic of each type.

Measurement errors **!**
Manufacturing tolerances **!**

Possible ("synthetic") measured data for two types of diodes; measurement errors and manufacturing tolerances taken into account by (Gaussian) random perturbations.

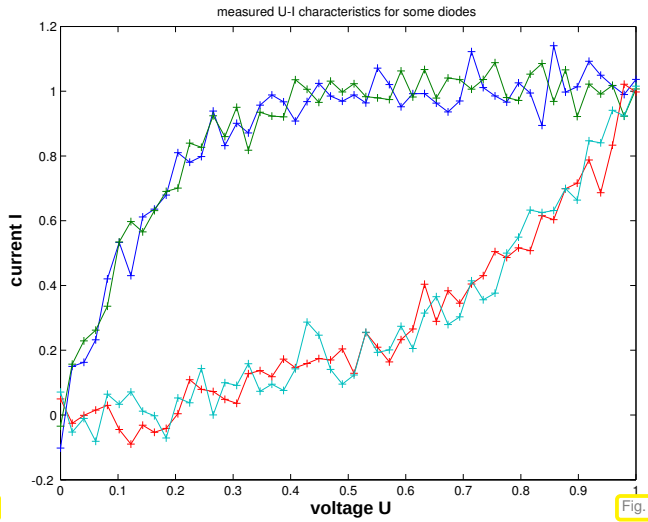


Fig. 108

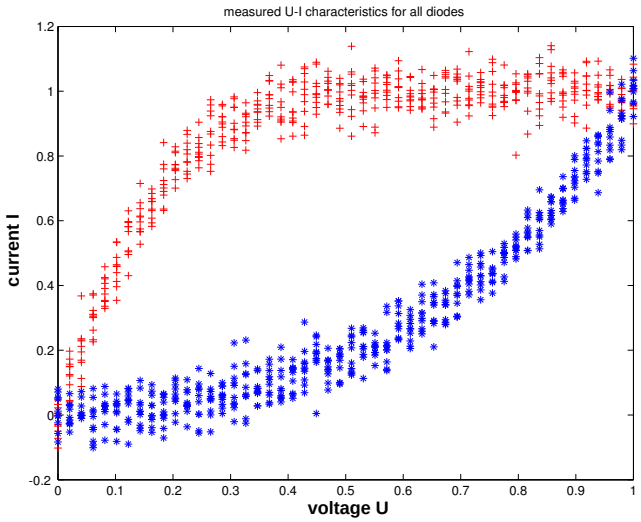


Fig. 109

Ex. 3.4.54 and Ex. 3.4.55 present typical tasks that can be tackled by principal component analysis.

Given: n data points $\mathbf{a}_j \in \mathbb{R}^m$, $j = 1, \ldots, n$, in m-dimensional (feature) space
(e.g., $\mathbf{a}_j$ may represent a finite *time series* or a *measured relationship* of physical quantities)

In Ex. 3.4.54: $n \triangleq$ number of stocks,
$m \triangleq$ number of days, for which stock prices are recorded

✦ Extreme case: all stocks follow exactly *one* trend

$$\leftrightarrow \quad \mathbf{a}_j \in \mathrm{Span}\{\mathbf{u}\} \quad \forall j = 1, \ldots, n \,,$$

for a trend vector $\mathbf{u} \in \mathbb{R}^m$, $\|\mathbf{u}\|_2 = 1$.

✦ Unlikely case: all stocks prices are governed by $p < n$ trends:

$$\leftrightarrow \quad \mathbf{a}_j \in \mathrm{Span}\{\mathbf{u}_1, \ldots, \mathbf{u}_p\} \quad \forall j = 1, \ldots, m \,, \tag{3.4.56}$$

with orthonormal trend vectors $\mathbf{u}_i \in \mathbb{R}^m$, $i = 1, \ldots, p$.

Why unlikely **?** Small *random fluctuations* will be present in each stock prize
Why orthonormal **?** Trends should be as "independent as possible" (minimally correlated)

Perspective of linear algebra:

$$(3.4.56) \quad \Leftrightarrow \quad \mathrm{rank}(\mathbf{A}) = p \text{ for } \mathbf{A} := [\mathbf{a}_1, \ldots, \mathbf{a}_n] \in \mathbb{R}^{m,n} \,, \quad \mathcal{R}(\mathbf{A}) = \mathrm{Span}\{\mathbf{u}_1, \ldots, \mathbf{u}_p\} \tag{3.4.57}$$

✦ Realistic: stock prizes *approximately* follow a few trends

$$\mathbf{a}_j \in \mathrm{Span}\{\mathbf{u}_1, \ldots, \mathbf{u}_p\} + \text{"small perturbations"} \quad \forall j = 1, \ldots, m \,,$$

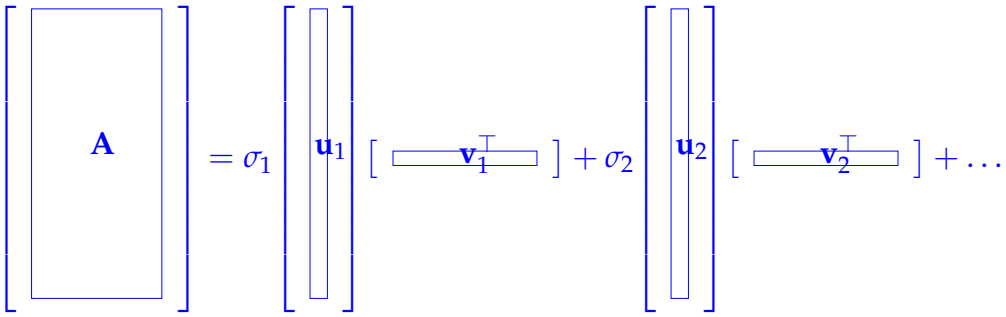
with orthonormal trend vectors $\mathbf{u}_i$, $i = 1, \ldots, p$.

Task (PCA): determine (minimal) p and orthonormal trend vectors $\mathbf{u}_i$, $i = 1, \ldots, p$

Now singular value decomposition (SVD) according to Def. 3.4.3 comes into play, because Lemma 3.4.11 tells us that it can supply an orthonormal basis of the image space of a matrix, *cf.* Code 3.4.21.

Issue: how to deal with (small, random) perturbations **?**

Recall Rem. 3.4.6, (3.4.8): If $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{H}^\top$ is the SVD of $\mathbf{A} \in \mathbb{R}^{m,n}$, then ($\mathbf{u}_j \hat{=}$ columns of $\mathbf{U}$, $\mathbf{v}_j \hat{=}$ columns of $\mathbf{V}$)



This already captures the case (3.4.56) and we see that the columns of $\mathbf{U}$ supply the trend vectors we are looking for!

❶ no perturbations:

$$\text{SVD:}\quad \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}} \quad \text{satisfies} \quad \sigma_1 \geq \sigma_2 \geq \ldots \sigma_p > \sigma_{p+1} = \cdots = \sigma_{\min\{m,n\}} = 0\,,$$
$$\text{orthonormal trend vectors} \quad (\mathbf{U})_{:,1}, \ldots, (\mathbf{U})_{:,p}\,.$$

❷ with perturbations:

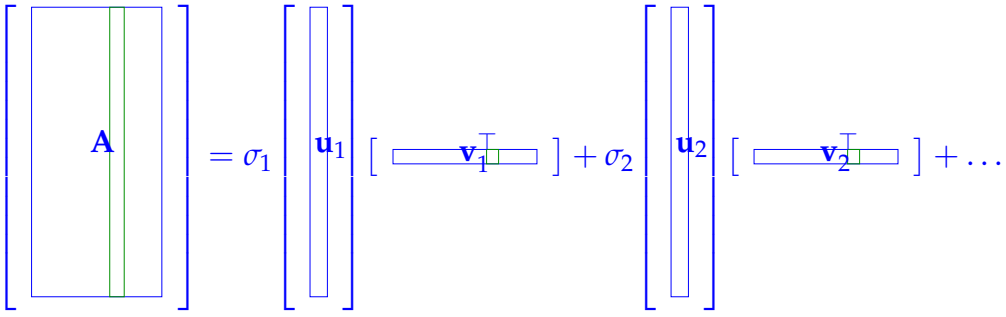
$$\text{SVD:}\quad \mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{\mathrm{H}} \quad \text{satisfies} \quad \sigma_1 \geq \sigma_2 \geq \ldots \sigma_p \gg \sigma_{p+1} \approx \cdots \approx \sigma_{\min\{m,n\}} \approx 0\,,$$
$$\text{orthonormal trend vectors} \quad (\mathbf{U})_{:,1}, \ldots, (\mathbf{U})_{:,p}\,.$$

If there is a pronounced gap in distribution of the singular values, which separates p large from $\min\{m,n\} - p$ relatively small singular values, this hints that $\mathcal{R}\mathbf{A}$ has essentially dimension p. It depends on the application what one accepts as a "pronounced gap".

Frequently used criterion:

$$p = \min\left\{ q\colon \sum_{j=1}^{q} \sigma_j^2 \geq (1-\tau) \sum_{j=1}^{\min\{m,n\}} \sigma_j^2 \right\} \quad \text{for} \quad \tau \ll 1\,.$$

What is the Information carried by $\mathbf{V}$ in PCA context **?**



j-th data set ($\leftrightarrow$ time series #j) in j-th column of $\mathbf{A}$

$$\text{(3.4.8)} \quad \Rightarrow \quad (\mathbf{A})_{:,j} = \sigma_1 \mathbf{u}_1 (\mathbf{v}_1)_j + \sigma_2 \mathbf{u}_2 (\mathbf{v}_2)_j + \ldots$$

The j-th row of $\mathbf{V}$ (up to the p-th component) gives the weights with which the p identified trends contribute to data set j.

Example 3.4.58 (Data points confined to a subspace)

MATLAB-code 3.4.59: PCA in three dimensions via SVD

```
% Use of SVD for PCA with perturbations

V = [1 , -1; 0 , 0.5; -1 , 0]; A = V*rand(2,20)+0.1*rand(3,20);
[U,S,V] = svd(A,0);

figure; sv = diag(S(1:3,1:3))

[X,Y] = meshgrid(-2:0.2:0,-1:0.2:1); n = size(X,1); m =
   size(X,2);
figure; plot3(A(1,:),A(2,:),A(3,:),'r*'); grid on; hold on;
M =
   U(:,1:2)*[reshape(X,1,prod(size(X)));reshape(Y,1,prod(size(Y)))];
mesh(reshape(M(1,:),n,m),reshape(M(2,:),n,m),reshape(M(3,:),n,m));
colormap(cool); view(35,10);

print -depsc2 '../PICTURES/svdpca.eps';
```

singular values:

$$\begin{matrix} 3.1378 \\ 1.8092 \\ \hline 0.1792 \end{matrix}$$

third singular value ➢ the data points essentially lie in a 2D subspace.

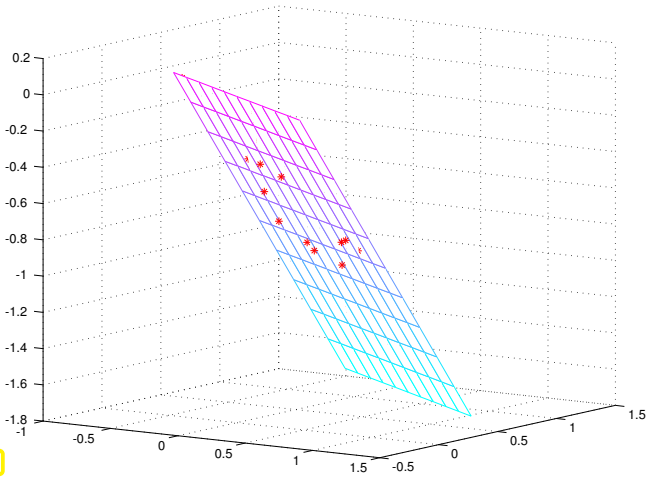


Fig. 110

Example 3.4.60 (Principal component analysis for data classification → Ex. 3.4.55 cnt'd)

Given: measured U-I characteristics of $n = 20$ unknown diodes, $I(U)$ available for $m = 50$ voltages.

Sought: Number of different types of diodes in batch and reconstructed U-I characteristic for each type.

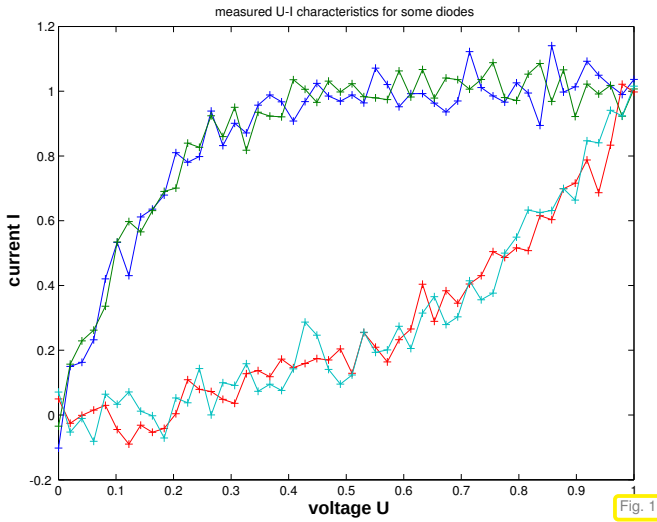


Fig. 111

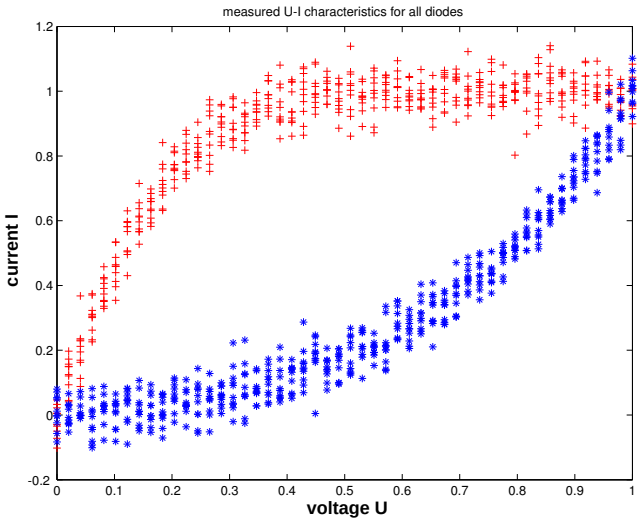


Fig. 112

MATLAB-code 3.4.61: Generation of synthetic perturbed U-I characteristics

```
% Generate synthetic measurement curves with random multiplicative and
   additive
% perturbations supposed to reflect measurement errors and
   manufacturing tolerances

% Voltage-current characteristics of both kinds of diodes
i1 = @(u) (2./(1+exp(-10*u)) - 1);
i2 = @(u) ((exp(3*u)-1)/(exp(3)-1));
% Simulated measurements
m = 50; % number of measurements for different input voltages
n = 10; % no.  of diodes of each kind
na = 0.05; % level of additive noise (normally distributed)
nm = 0.02; % level of multiplicative noise (normally distributed)

uvals = (0:1/(m-1):1);
D1 = (1+nm*randn(n,m)).*(i1(repmat(uvals,n,1)))+na*randn(n,m);
D2 = (1+nm*randn(n,m)).*(i2(repmat(uvals,n,1)))+na*randn(n,m);
A = ([D1;D2])'; A = A(1:size(A,1),randperm(1:size(A,2)));
```

Data matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \gg n$:

Columns of $\mathbf{A}$ $\rightarrow$ series of measurements for different diodes (times/locations etc.),
Rows of $\mathbf{A}$ $\rightarrow$ measured values corresponding to one diode (time/location etc.).

Goal of PCA: detect linear correlations between columns of $\mathbf{A}$

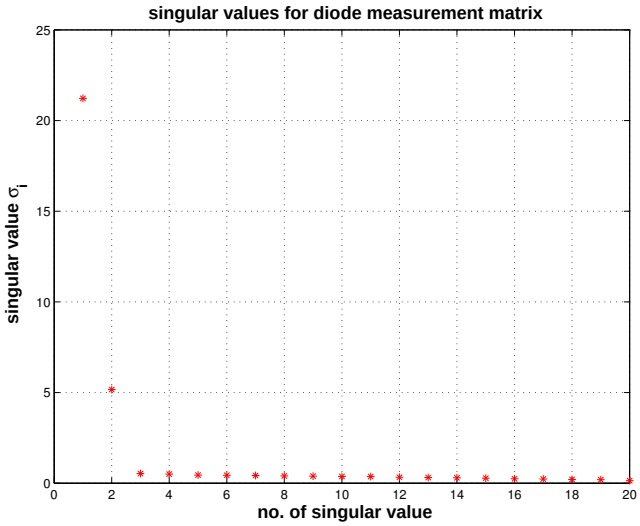


Fig. 113

← distribution of singular values of matrix

two dominant singular values !

measurements display linear correlation with **two** principal components

two types of diodes in batch

MATLAB-code 3.4.62: PCA for measured U-I characteristics

```
clear all; diodedata;

figure('name','U-I characteristics for a few diodes');
plot(uvals,(D(:,[1 7 13 17]))','-+');
xlabel('{\bf voltage U}','fontsize',14);
ylabel('{\bf current I}','fontsize',14);
title('measured U-I characteristics for some diodes');
print -depsc2 '../PICTURES/diodepcafewmeas.eps';

figure('name','measured U-I characteristics');
plot(uvals,D1','r+'); hold on;
plot(uvals,D2','b*');
xlabel('{\bf voltage U}','fontsize',14);
ylabel('{\bf current I}','fontsize',14);
title('measured U-I characteristics for all diodes');
print -depsc2 '../PICTURES/diodepcameas.eps';

% Perform SVD based PCA
[U,S,V] = svd(D);

figure('name','singular values');
sv = diag(S(1:2*n,1:2*n));
plot(1:2*n,sv,'r*'); grid on;
xlabel('{\bf index i of singular value}','fontsize',14);
ylabel('{\bf singular value \sigma_i}','fontsize',14);
title('{\bf singular values for diode measurement
   matrix}','fontsize',14);
print -depsc2 '../PICTURES/diodepcasv.eps';

figure('name','trend vectors');
plot(1:m,U(:,1:2),'+');
xlabel('{\bf voltage U}','fontsize',14);
ylabel('{\bf current I}','fontsize',14);
title('{\bf principal components (trend vectors) for diode
```

```
   measurements}','fontsize',14);
legend('dominant principal component','second principal
   component','location','best');
print -depsc2 '../PICTURES/diodepcau.eps';

figure('name','strength');
plot(V(:,1),V(:,2),'mo'); grid on;
xlabel('{\bf strength of singular component #1}','fontsize',14);
ylabel('{\bf strength of singular component #2}','fontsize',14);
title('{\bf strengths of contributions of singular
   components}','fontsize',14);
print -depsc2 '../PICTURES/diodepcav.eps';
```

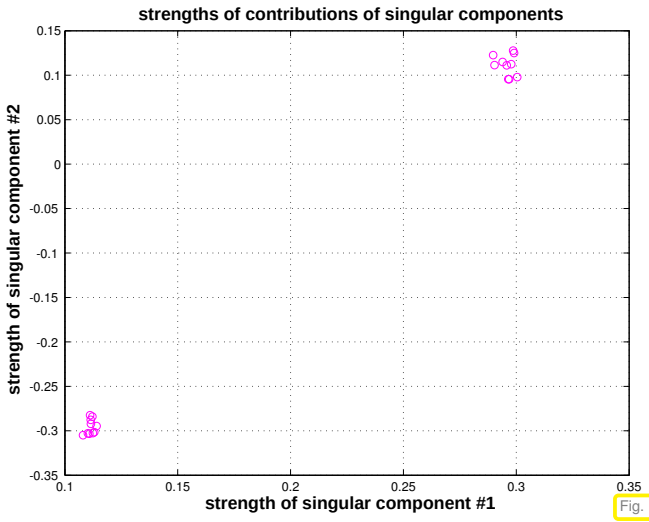


Fig. 114

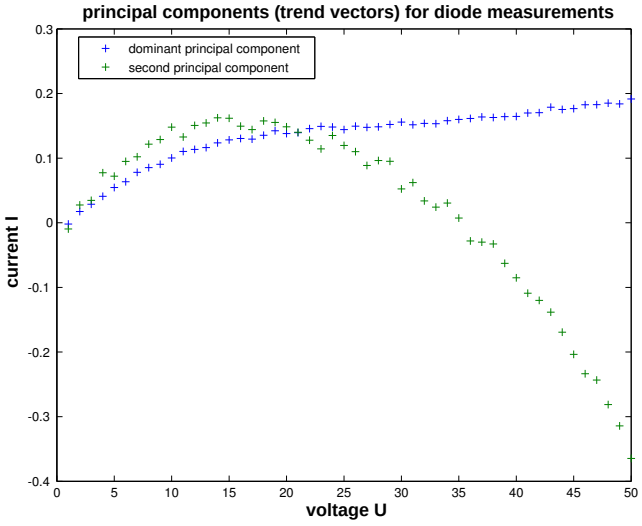


Fig. 115

Observations:

- First two rows of $\mathbf{V}$-matrix specify strength of contribution of the two leading principal components to each measurement

  ➢ Points $(\mathbf{V})_{:,1:2}$, which correspond to different diodes are neatly clustered in $\mathbb{R}^2$. To determine the type of diode i, we have to identify the cluster to which the point $((\mathbf{V})_{i,1}, \mathbf{V}_{i,2})$ belongs (→ cluster analysis, course "machine learning", next Rem. 3.4.65).

- The principal components themselves do not carry much useful information in this example.

Remark 3.4.63 (Principal axis of a point cloud)

Given $m > 2$ points $\mathbf{x}_j \in \mathbb{R}^k$, $j = 1, \ldots, m$, in k-dimensional space, we ask what is the "longest" and "shortest" diameter $\mathbf{d}_+$ and $\mathbf{d}_-$. This question can be stated rigorously in several different ways: here we ask for directions for which the point cloud will have maximal/minimal variance, when projected onto that direction:

$$\mathbf{d}_+ := \operatorname*{argmax}_{\|\mathbf{v}\|=1} Q(\mathbf{v}) \,, \quad \mathbf{d}_- := \operatorname*{argmin}_{\|\mathbf{v}\|=1} Q(\mathbf{v}) \,, \qquad Q(\mathbf{v}) := \sum_{j=1}^{m} |(\mathbf{x}_i - \mathbf{c})^\top \mathbf{v}|^2 \,, \quad \mathbf{c} = \frac{1}{m} \sum_{j=1}^{m} \mathbf{x}_j \,. \tag{3.4.64}$$

The directions $\mathbf{d}_+$, $\mathbf{d}_-$ are called the principal axes of the point cloud, a term borrowed from mechanics and connected with the axes of inertia of an assembly of point masses.

Principal axes of a point cloud in 2D

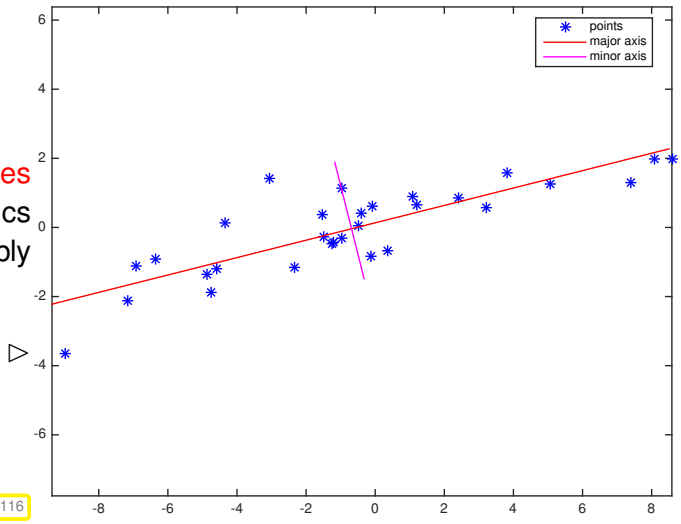


Fig. 116

$\mathbf{d}_+$, $\mathbf{d}_-$ can be computed by computing the extremizers of $\mathbf{x} \mapsto \|\mathbf{A}\mathbf{x}\|_2$ with

$$\mathbf{A} = \begin{bmatrix} (\mathbf{x}_1 - \mathbf{c})^\top \\ \vdots \\ (\mathbf{x}^m - \mathbf{c})^\top \end{bmatrix} \in \mathbb{R}^{m,k} ,$$

on $\{\mathbf{x} \in \mathbb{R}^k : \|\mathbf{x}\|_2 = 1\}$, using the SVD-based method presented in Section 3.4.4.1.

Remark 3.4.65 (Algorithm for cluster analysis)

Given: ✦ N data points $\mathbf{x}_i \in \mathbb{R}^k$, $i = 1, \ldots, N$,
✦ No. n of desired clusters.

Sought: Partitioning of index set $\{1, \ldots, N\} = I_1 \cup \cdots \cup I_n$, achieving minimal mean least squares error

$$\mathrm{mlse} := \sum_{l=1}^{n} \sum_{i \in I_l} \|\mathbf{x}_i - \mathbf{m}_l\|_2^2 , \quad \mathbf{m}_l = \frac{1}{\sharp I_l} \sum_{i \in I_l} \mathbf{x}_i . \tag{3.4.66}$$

The subsets $\{\mathbf{x}_i : i \in I_l\}$ are called the clusters. The points $\mathbf{m}_l$ are their centers of gravity.

The Algorithm involves two components:

❶ Improvement of clusters using the Lloyd-Max algorithm, see Code 3.4.71. It involves two steps in turns:

(a) Given centers of gravity $\mathbf{m}_l$ redistribute points according to

$$I_l := \{i \in \{1, \ldots, N\} : \|\mathbf{x}_i - \mathbf{m}_l\|_2 \leq \|\mathbf{x}_i - \mathbf{m}_k\|_2 \; \forall k \neq l\} . \tag{3.4.67}$$

(b) Recompute centers of gravity

$$\mathbf{m}_l = \frac{1}{\sharp I_l} \sum_{i \in I_l} \mathbf{x}_i . \tag{3.4.68}$$

❷ Splitting of cluster by separation along its principal axis, see Rem. 3.4.63 and Code 3.4.70:

$$\mathbf{a}_l := \operatorname*{argmax}_{\|\mathbf{v}\|_2 = 1} \{ \sum_{i \in I_l} |(\mathbf{x}_i - \mathbf{m}_l)^\top \mathbf{v}|^2 \} \tag{3.4.69}$$

C++-code 3.4.70: Principal axis point set separation

```
// Separation of a set of points whose coordinates are stored in the
// columns of X according to their location w.r.t. the principal axis
std::pair<VectorXi, VectorXi> princaxissep(const MatrixXd & X){
  int N = X.cols(); // no. of points
  VectorXd g = X.rowwise().sum() / N; // Center of gravity, cf. (3.4.68)
  MatrixXd Y = X - g.replicate(1,N); // Normalize point coordinates.
  // Compute principal axes, cf. (3.4.69) and (3.4.33). Note that the
  // SVD of a symmetric matrix is available through an orthonormal
  // basis of eigenvectors.
  SelfAdjointEigenSolver<MatrixXd> es(Y*Y.transpose());
  // Major principal axis
  Eigen::VectorXd a = es.eigenvectors().rightCols<1>();
  // Coordinates of points w.r.t. to major principal axis
  Eigen::VectorXd c = a.transpose()*Y;
  // Split point set according to locations of projections on principal
     axis
  // std::vector with indices to prevent resizing of matrices
  std::vector<int> i1, i2;
  for(int i = 0; i < c.size(); ++i){
        if(c(i) >= 0)
            i1.push_back(i);
        else
            i2.push_back(i);
  }
  // return the mapped std::vector as Eigen::VectorXd
  return std::pair<VectorXi, VectorXi>(VectorXi::Map(i1.data(),
     i1.size()), VectorXi::Map(i2.data(), i2.size()));
}
```

C++-code 3.4.71: Lloyd-Max algorithm for cluster indentification

```
template <class Derived>
std::tuple<double, VectorXi, VectorXd> distcomp(const MatrixXd & X,
   const MatrixBase<Derived> & C){
  // Compute squared distances
  // d.row(j) = squared distances from all points in X to cluster j
  MatrixXd d(C.cols(), X.cols());
  for(int j = 0; j < C.cols(); ++j){
      MatrixXd Dv = X - C.col(j).replicate(1, X.cols());
      d.row(j) = Dv.array().square().colwise().sum();
  }
  // Compute minimum distance point association and sum of minimal
     squared distances
  VectorXi idx(d.cols()); VectorXd mx(d.cols());
  for(int j = 0; j < d.cols(); ++j){
      // mx(j) tells the minimal squared distance of point j to the
         nearest cluster
      // idx(j) tells to which cluster point j belongs
      mx(j) = d.col(j).minCoeff(&idx(j));
```

```
17    }
18    double sumd = mx.sum();      // sum of all squared distances
19    // Computer sum of squared distances within each cluster
20    VectorXd cds(C.cols()); cds.setZero();
21    for(int j = 0; j < idx.size(); ++j)   // loop over all points
22        cds(idx(j)) += mx(j);
23    return std::make_tuple(sumd, idx, cds);
24  }
25
26  // Lloyd-Max iterative vector quantization algorithm for discrete point
27  // sets; the columns of X contain the points x_i, the columns of
28  // C initial approximations for the centers of the clusters.  The final
29  // centers are returned in C, the index vector idx specifies
30  // the association of points with centers.
31  template <class Derived>
32  void lloydmax(const MatrixXd & X, MatrixBase<Derived> & C, VectorXi &
        idx, VectorXd & cds, const double tol = 0.0001){
33    int k = X.rows(); // dimension of space
34    int N = X.cols(); // no.  of points
35    int n = C.cols(); // no.  of clusters
36    if(k != C.rows())
37      throw std::logic_error("dimension mismatch");
38    double sd_old = std::numeric_limits<double>::max();
39    double sd;
40    std::tie(sd, idx, cds) = distcomp(X,C);
41    // Terminate, if sum of squared minimal distances has not changed
        much
42    while( (sd_old-sd)/sd > tol ){
43      // Compute new centers of gravity according to (3.4.68)
44      MatrixXd Ctmp(C.rows(),C.cols());  Ctmp.setZero();
45      // number of points in cluster for normalization
46      VectorXi nj(n); nj.setZero();
47      for(int j = 0; j < N; ++j){ // loop over all points
48        Ctmp.col(idx(j)) += X.col(j);
49        ++nj(idx(j));      // count associated points for
            normalization
50      }
51      for(int i = 0; i < Ctmp.cols(); ++i){
52        if(nj(i) > 0)
53          C.col(i) = Ctmp.col(i)/nj(i); // normalization
54      }
55      sd_old = sd;
56      // Get new minimum association of the points to cluster points
57      // for next iteration
58      std::tie(sd, idx, cds) = distcomp(X,C);
59    }
60  }
61  // Note:  this function is needed to allow a call with an rvalue
62  // && stands for an rvalue reference and allows rvalue arguments
63  // such as C.leftCols(nc) to be passed by reference (C++ 11 feature)
64  template <class Derived>
```

```
65 void lloydmax(const MatrixXd & X, MatrixBase<Derived> && C, VectorXi
       & idx, VectorXd & cds, const double tol = 0.0001){
66   lloydmax(X, C, idx, cds);
67 }
```

C++-code 3.4.72: Clustering of point set

```
2  // n-quantization of point set in k-dimensional space based on
3  // minimizing the mean square error of Euclidean distances.  The
4  // columns of the matrix X contain the point coordinates, n specifies
5  // the desired number of clusters.
6  std::pair<MatrixXd, VectorXi> pointcluster(const MatrixXd & X, const
       int n){
7    int N = X.cols(); // no.  of points
8    int k = X.rows(); // dimension of space
9    // Start with two clusters obtained by principal axis separation
10   int nc = 1; // Current number of clusters
11   // Initial single cluster encompassing all points
12   VectorXi Ibig = VectorXi::LinSpaced(N,0,N-1);
13   int nbig = 0; // Index of largest cluster
14   MatrixXd C(X.rows(), n);     // matrix for cluster midpoints
15   C.col(0) = X.rowwise().sum()/N; // center of gravity
16   VectorXi idx(N); idx.setOnes();
17   // Split largest cluster into two using the principal axis separation
18   // algorithm
19   while(nc < n){
20     VectorXi i1, i2;
21     MatrixXd Xbig(k,Ibig.size());
22     for(int i = 0; i < Ibig.size(); ++i)  // slicing
23       Xbig.col(i) = X.col(Ibig(i));
24     // separete Xbig into two clusters, i1 and i2 are index vectors
25     std::tie(i1, i2) = princaxissep(Xbig);
26     // new cluster centers of gravity
27     VectorXd c1(k), c2(k); c1.setZero(); c2.setZero();
28     for(int i = 0; i < i1.size(); ++i)
29       c1 += X.col(Ibig(i1(i)));
30     for(int i = 0; i < i2.size(); ++i)
31       c2 += X.col(Ibig(i2(i)));
32     c1 /= i1.size();    c2 /= i2.size(); // normalization
33     C.col(nbig) = c1;
34     C.col(nbig+1) = c2;
35     ++nc; // Increase number of clusters
36     // Improve clusters by Lloyd-Max iteration
37     VectorXd cds;     // saves mean square error of clusters
38     // Note C.leftCols(nc) is passed as rvalue reference (C++ 11)
39     lloydmax(X,C.leftCols(nc),idx,cds);
40     // Identify cluster with biggest contribution to mean square
           error
41     cds.maxCoeff(&nbig);
42     int counter = 0;
```

```
43          // update Ibig with indices of points in cluster with biggest
               contribution
44          for(int i = 0; i < idx.size(); ++i){
45              if(idx(i) == nbig){
46                  Ibig(counter) = i;
47                  ++counter;
48              }
49          }
50          Ibig.conservativeResize(counter);
51      }
52      return std::make_pair(C, idx);
53  }
```

Example 3.4.73 (Principal component analysis for data analysis → Ex. 3.4.54 cnt'd)

$\mathbf{A} \in \mathbb{R}^{m,n}$, $m \gg n$:

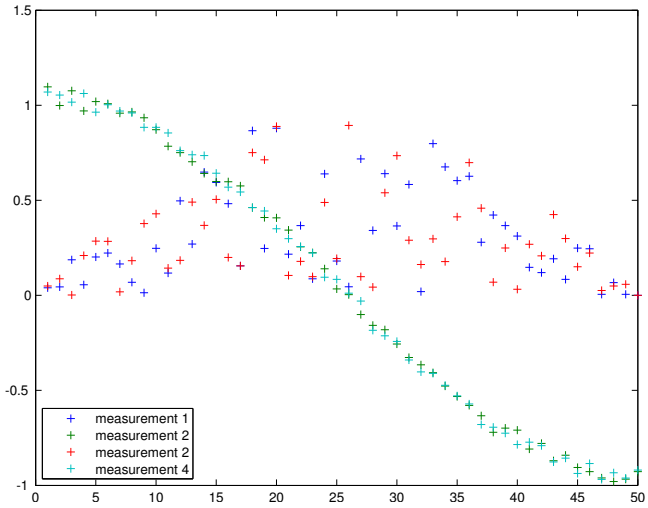
Columns $\mathbf{A}$ → series of measurements at different times/locations etc.
Rows of $\mathbf{A}$ → measured values corresponding to one time/location etc.

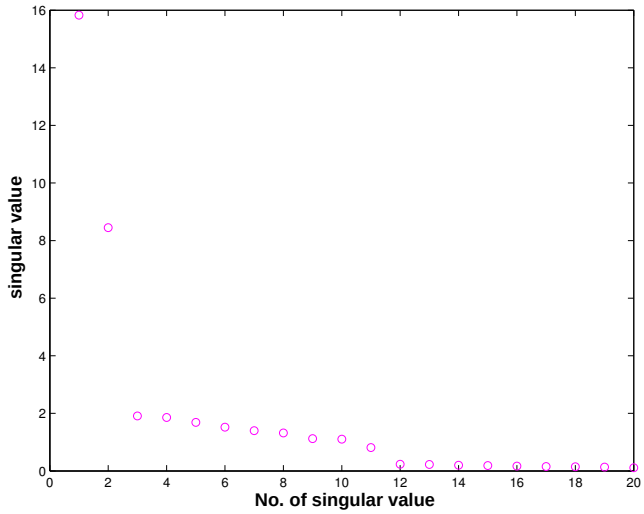
Goal: detect linear correlations

Concrete: two quantities measured over one year at 10 different sites

(Of course, measurements affected by errors/fluctuations)

```
n = 10;
m = 50;
x = sin(pi*(1:m)'/m);
y = cos(pi*(1:m)'/m);
A = [];
for i = 1:n
  A = [A, x.*rand(m,1),...
       y+0.1*rand(m,1)];
end
```





← distribution of singular values of matrix
two dominant singular values !

measurements display linear correlation with **two** principal components

principal components = $\mathbf{u}_{\cdot,1}$, $\mathbf{u}_{\cdot,2}$ (leftmost columns of $\mathbf{U}$-matrix of SVD)
their relative weights = $\mathbf{v}_{\cdot,1}$, $\mathbf{v}_{\cdot,2}$ (leftmost columns of $\mathbf{V}$-matrix of SVD)

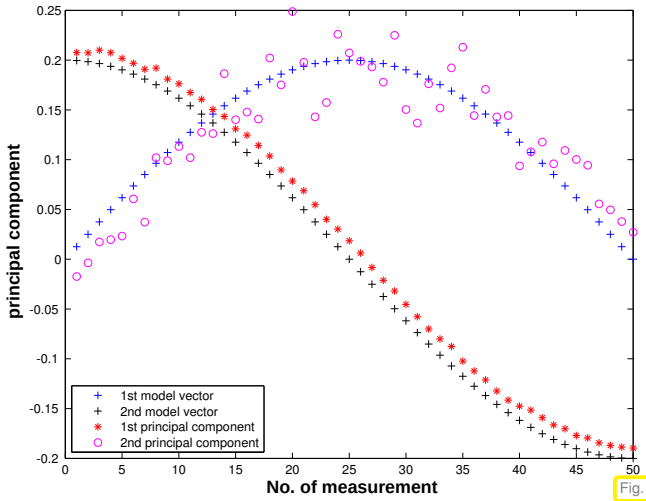


Fig. 117

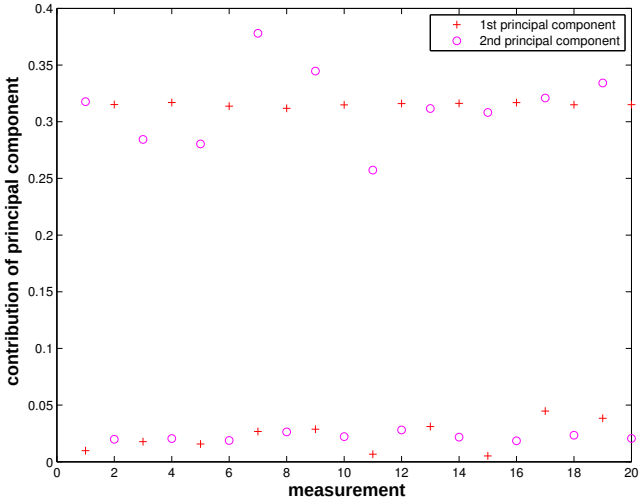


Fig. 118

Pronounced clustering of contributions around two points of dominant singular components matches the presence of two fundamental behaviors in the measured data ▷

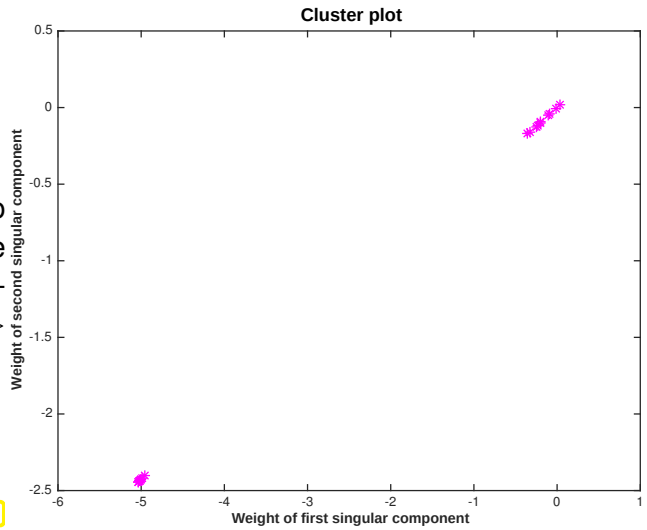


Fig. 119

Example 3.4.74 (PCA of stock prices → Ex. 3.4.54)

columns of $\mathbf{A}$ → time series of end of day stock prices of individual stocks
rows of $\mathbf{A}$ → closing prices of DAX stocks on a particular day

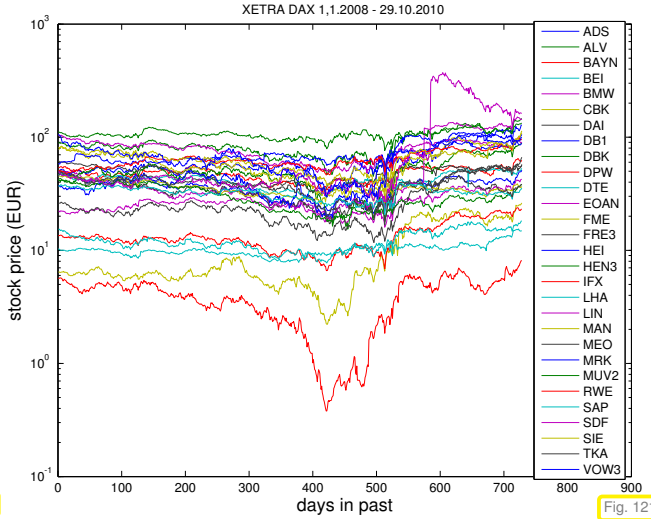


Fig. 120

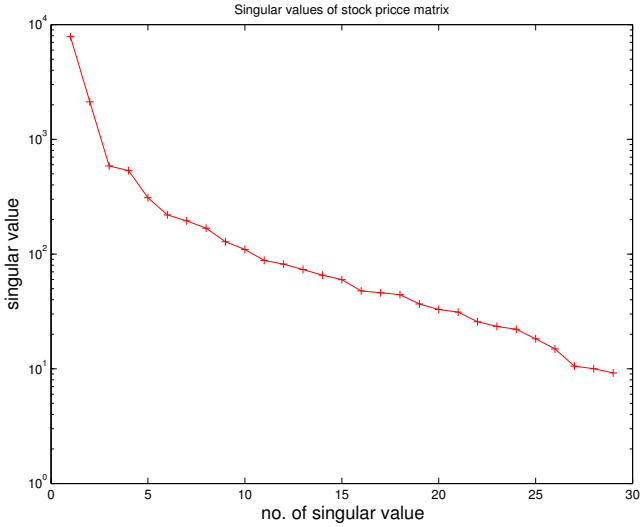


Fig. 121

We observe a pronounced decay of the singular values (≈ exponential decay, logarithmic scale in Fig. 121)

➢ a few trends (corresponding to a few of the largest singular values) govern the time series.

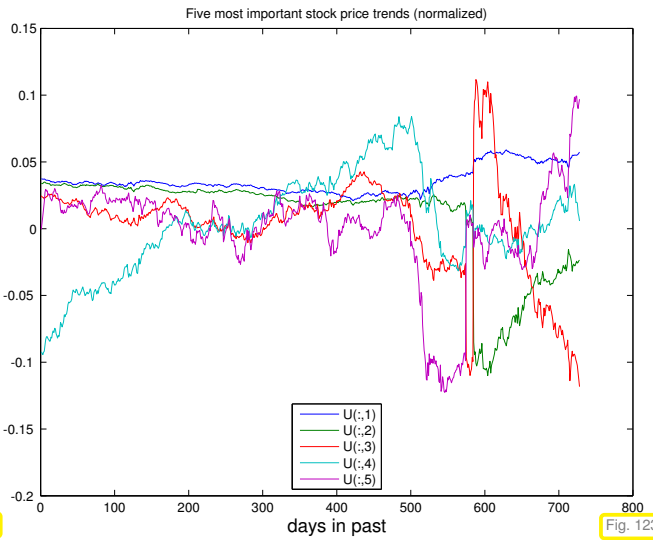


Fig. 122

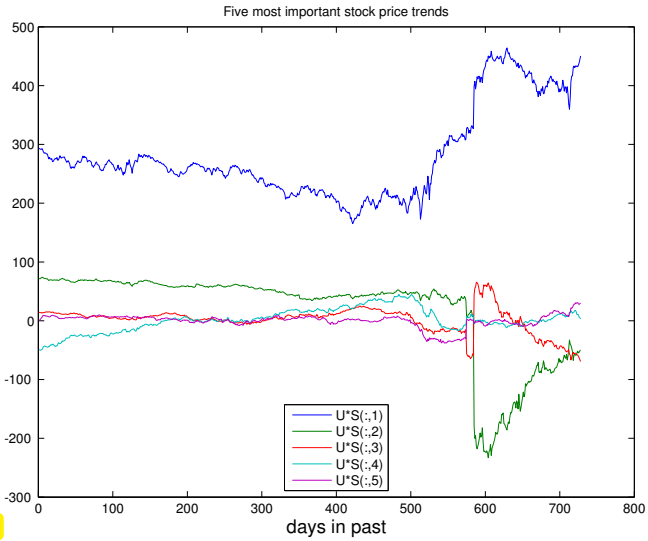


Fig. 123

Columns of $\mathbf{U}$ (→ Fig. 122) in SVD $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ provide trend vectors, *cf.* Ex. 3.4.54 & Ex. 3.4.73.

When weighted with the corresponding singular value, the importance of a trend contribution emerges, see Fig. 123

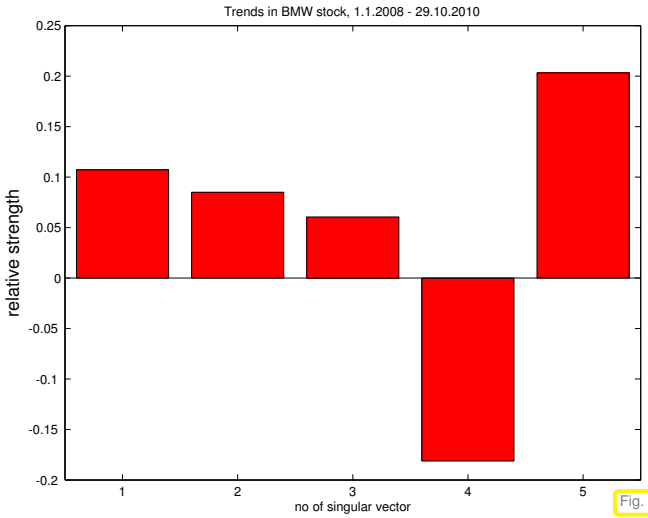


Fig. 124

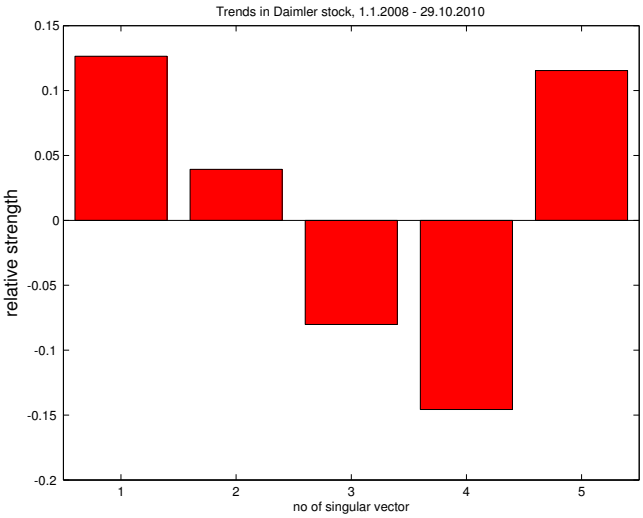


Fig. 125

Stocks of companies from the same sector of the economy should display similar contributions of major trend vectors, because their prices can be expected to be more closely correlated than stock prices in general.

Data obtained from Yahoo Finance:

```
#!/bin/csh
foreach i (ADS  ALV BAYN BEI BMW CBK DAI DBK DB1 LHA DPW DTE EOAN FRE3 \
           FME HEI HEN3 IFX SDF LIN MAN MRK MEO MUV2 RWE SAP SIE TKA VOW3)
wget -O "i".csv "http://ichart.finance.yahoo.com/table.csv?s=i.DE&a=00&b=1&
    c=2008&d=09&e=30&f=2010&g=d&ignore=.csv"
sed -i -e 's/-/,/g' "i".csv
end
```

MATLAB-code 3.4.75: PCA of stock prices in MATLAB

```
% Read end of day XETRA-DAX stock quotes into matrices
k = 1;
M{k} = dlmread('ADS.csv',',',1,0); leg{k} = 'ADS'; k = k + 1;
```

```
M{k} = dlmread('ALV.csv',',',1,0); leg{k} = 'ALV'; k = k + 1;
M{k} = dlmread('BAYN.csv',',',1,0); leg{k} = 'BAYN'; k = k + 1;
M{k} = dlmread('BEI.csv',',',1,0); leg{k} = 'BEI'; k = k + 1;
M{k} = dlmread('BMW.csv',',',1,0); leg{k} = 'BMW'; k = k + 1;
M{k} = dlmread('CBK.csv',',',1,0); leg{k} = 'CBK'; k = k + 1;
M{k} = dlmread('DAI.csv',',',1,0); leg{k} = 'DAI'; k = k + 1;
M{k} = dlmread('DB1.csv',',',1,0); leg{k} = 'DB1'; k = k + 1;
M{k} = dlmread('DBK.csv',',',1,0); leg{k} = 'DBK'; k = k + 1;
M{k} = dlmread('DPW.csv',',',1,0); leg{k} = 'DPW'; k = k + 1;
M{k} = dlmread('DTE.csv',',',1,0); leg{k} = 'DTE'; k = k + 1;
M{k} = dlmread('EOAN.csv',',',1,0); leg{k} = 'EOAN'; k = k + 1;
M{k} = dlmread('FME.csv',',',1,0); leg{k} = 'FME'; k = k + 1;
M{k} = dlmread('FRE3.csv',',',1,0); leg{k} = 'FRE3'; k = k + 1;
M{k} = dlmread('HEI.csv',',',1,0); leg{k} = 'HEI'; k = k + 1;
M{k} = dlmread('HEN3.csv',',',1,0); leg{k} = 'HEN3'; k = k + 1;
M{k} = dlmread('IFX.csv',',',1,0); leg{k} = 'IFX'; k = k + 1;
M{k} = dlmread('LHA.csv',',',1,0); leg{k} = 'LHA'; k = k + 1;
M{k} = dlmread('LIN.csv',',',1,0); leg{k} = 'LIN'; k = k + 1;
M{k} = dlmread('MAN.csv',',',1,0); leg{k} = 'MAN'; k = k + 1;
M{k} = dlmread('MEO.csv',',',1,0); leg{k} = 'MEO'; k = k + 1;
M{k} = dlmread('MRK.csv',',',1,0); leg{k} = 'MRK'; k = k + 1;
M{k} = dlmread('MUV2.csv',',',1,0); leg{k} = 'MUV2'; k = k + 1;
M{k} = dlmread('RWE.csv',',',1,0); leg{k} = 'RWE'; k = k + 1;
M{k} = dlmread('SAP.csv',',',1,0); leg{k} = 'SAP'; k = k + 1;
M{k} = dlmread('SDF.csv',',',1,0); leg{k} = 'SDF'; k = k + 1;
M{k} = dlmread('SIE.csv',',',1,0); leg{k} = 'SIE'; k = k + 1;
M{k} = dlmread('TKA.csv',',',1,0); leg{k} = 'TKA'; k = k + 1;
M{k} = dlmread('VOW3.csv',',',1,0); leg{k} = 'VOW3';
% Data format of matrices M:
% M(:,1): year, M(:,2): month, M(:,3): day, M(:,6): end of day
    stock quote
% Clean data: remove/interpolate days without trading
dv = [];
for j=1:k
  M{j}(:,1) = round(366*M{j}(:,1)+31*M{j}(:,2)+M{j}(:,3));
  dv = [dv; M{j}(:,1)];
end
dv = unique(dv); mxd = max(dv)+1;
dv = mxd - dv; mdays = max(dv);
A = zeros(mdays,k);
for j=1:k, A(mxd - M{j}(:,1),j) = M{j}(:,6); end
idx = find(sum(A') ~= 0); A = A(idx,:);

for j=size(A,1):-1:2
  zidx = find(A(j-1,:) == 0);
  A(j-1,zidx) = A(j,zidx);
end
for j=2:size(A,1)
  zidx = find(A(j,:) == 0);
  A(j,zidx) = A(j-1,zidx);
end

figure('name','DAX');
```

```
semilogy(A); set(gca,'xlim',[0 900]);
legend(leg,'location','east');
xlabel('days in past','fontsize',14);
ylabel('stock price (EUR)','fontsize',14);
title('XETRA DAX 1,1.2008 - 29.10.2010');
print -depsc2 '../../PICTURES/stockprizes.eps';

[U,S,V] = svd(A,0);
figure('name','singular values');
semilogy(diag(S),'r-+');
xlabel('no. of singular value','fontsize',14);
ylabel('singular value','fontsize',14);
title('Singular values of stock pricce matrix');
print -depsc2 '../../PICTURES/stocksingval.eps';

figure('name','trend vectors');
plot(U(:,1:5));
xlabel('days in past','fontsize',14);
title('Five most important stock price trends (normalized)');
legend('U(:,1)','U(:,2)','U(:,3)','U(:,4)','U(:,5)','location','south');
print -depsc2 '../../PICTURES/stocktrendsn.eps';

figure('name','trend vectors');
plot(U(:,1:5)*S(1:5,1:5));
xlabel('days in past','fontsize',14);
title('Five most important stock price trends');
legend('U*S(:,1)','U*S(:,2)','U*S(:,3)','U*S(:,4)','U*S(:,5)','location','south')
print -depsc2 '../../PICTURES/stocktrends.eps';

figure('name','trend components BMW');
trendcomp(V,5,5);
title('Trends in BMW stock, 1.1.2008 - 29.10.2010');
print -depsc2 '../../PICTURES/bmw.eps';

figure('name','trend components DAI');
trendcomp(V,7,5);
title('Trends in Daimler stock, 1.1.2008 - 29.10.2010');
print -depsc2 '../../PICTURES/dai.eps';

figure('name','trend components DBK');
trendcomp(V,9,5);
title('Trends in Deutsche Bank stock, 1.1.2008 - 29.10.2010');
print -depsc2 '../../PICTURES/dbk.eps';

figure('name','trend components CBK');
trendcomp(V,6,5);
title('Trends in Commerzbank stock, 1.1.2008 - 29.10.2010');
print -depsc2 '../../PICTURES/cbk.eps';
```

3.5 Total Least Squares

In the examples of Section 3.0.1 we generally considered overdetermined linear systems of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, for which only the right hand side vector $\mathbf{b}$ was affected by measurement errors. However, also the entries of the coefficient matrix $\mathbf{A}$ may have been obtained by measurement. This is the case, for instance, in the nodal analysis of electric circuits → Ex. 2.1.3. Then, it may be legitimate to seek a "better" matrix based on information contained in the whole linear system. This is the gist of the total least squares approach.

Given: overdetermined linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, $m > n$.

Known: LSE solvable $\Leftrightarrow$ $\mathbf{b} \in \mathrm{Im}(\mathbf{A})$, *if* $\mathbf{A}$, $\mathbf{b}$ *were not perturbed*,

but $\mathbf{A}$, $\mathbf{b}$ are perturbed (measurement errors).

Sought: *Solvable* overdetermined system of equations $\widehat{\mathbf{A}}\mathbf{x} = \widehat{\mathbf{b}}$, $\widehat{\mathbf{A}} \in \mathbb{R}^{m,n}$, $\widehat{\mathbf{b}} \in \mathbb{R}^m$, "nearest" to $\mathbf{A}\mathbf{x} = \mathbf{b}$.

☞ least squares problem "turned upside down": now we are allowed to tamper with system matrix and right hand side vector!

Total least squares problem:

Given: $\mathbf{A} \in \mathbb{R}^{m,n}$, $m > n$, $\mathrm{rank}(\mathbf{A}) = n$, $\mathbf{b} \in \mathbb{R}^m$,

find: $\widehat{\mathbf{A}} \in \mathbb{R}^{m,n}$, $\widehat{\mathbf{b}} \in \mathbb{R}^m$ with

$$\left\| [\mathbf{A} \quad \mathbf{b}] - \left[\widehat{\mathbf{A}} \quad \widehat{\mathbf{b}}\right] \right\|_F \to \min \quad , \quad \widehat{\mathbf{b}} \in \mathcal{R}(\widehat{\mathbf{A}}) \, . \tag{3.5.1}$$

$\widehat{\mathbf{b}} \in \mathcal{R}(\widehat{\mathbf{A}}) \;\Rightarrow\; \mathrm{rank}\left(\left[\widehat{\mathbf{A}} \quad \widehat{\mathbf{b}}\right]\right) = n$ ► (3.5.1) $\Rightarrow$ $\left[\widehat{\mathbf{A}} \quad \widehat{\mathbf{b}}\right] = \underset{\mathrm{rank}(\widehat{\mathbf{X}})=n}{\mathrm{argmin}} \left\| [\mathbf{A} \quad \mathbf{b}] - \widehat{\mathbf{X}} \right\|_F$.

☞ $\left[\widehat{\mathbf{A}} \quad \widehat{\mathbf{b}}\right]$ is the rank-n best approximation of $[\mathbf{A} \quad \mathbf{b}]$!

We face the problem to compute the best rank-n approximation of the given matrix $[\mathbf{A} \quad \mathbf{b}]$, a problem already treated in Section 3.4.4.2: Thm. 3.4.48 tells us how to use the SVD of $[\mathbf{A} \quad \mathbf{b}]$

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top \, , \quad \mathbf{U} \in \mathbb{R}^{m,n+1}, \; \mathbf{\Sigma} \in \mathbb{R}^{n+1,n+1}, \; \mathbf{V} \in \mathbb{R}^{n+1,n+1} \, , \tag{3.5.2}$$

to construct $\left[\widehat{\mathbf{A}} \quad \widehat{\mathbf{b}}\right]$:

$$[\mathbf{A} \quad \mathbf{b}] = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top = \sum_{j=1}^{n+1} \sigma_j (\mathbf{U})_{:,j} (\mathbf{V})_{:,j}^\top \overset{\text{Thm. 3.4.48}}{\Longrightarrow} \left[\widehat{\mathbf{A}} \quad \widehat{\mathbf{b}}\right] = \sum_{j=1}^{n} \sigma_j (\mathbf{U})_{:,j} (\mathbf{V})_{:,j}^\top \, . \tag{3.5.3}$$

$$\overset{\mathbf{V}\text{ orthogonal}}{\Longrightarrow} \left[\widehat{\mathbf{A}} \quad \widehat{\mathbf{b}}\right] (\mathbf{V})_{:,n+1} = \widehat{\mathbf{A}} (\mathbf{V})_{1:n,n+1} + \widehat{\mathbf{b}} (\mathbf{V})_{n+1,n+1} = 0 \, . \tag{3.5.4}$$

► (3.5.4) also provides the solution $\mathbf{x}$ of $\widehat{\mathbf{S}}\mathbf{x} = \widehat{\mathbf{b}}$,

$$\mathbf{x} := -\widehat{\mathbf{A}}^{-1}\widehat{\mathbf{b}} = -(\mathbf{V})_{1:n,n+1} / (\mathbf{V})_{n+1,n+1} \, . \tag{3.5.5}$$

C++-code 3.5.6: Total least squares via SVD

```
// computes only solution x of fitted consistent LSE
VectorXd lsqtotal(const MatrixXd& A, const VectorXd& b) {
  const unsigned m = A.rows(), n = A.cols();
  MatrixXd C(m, n + 1);  C << A, b; // C = [A,b]
  // We need only the SVD-factor V, see (3.5.3)
  MatrixXd V = C.jacobiSvd(Eigen::ComputeThinU |
     Eigen::ComputeThinV).matrixV();

  // Compute solution according to (3.5.5);
  double s = V(n, n);
  if (std::abs(s) < 1.0E-15) { cerr << "No solution!\n"; exit(1); }
  return (-V.col(n).head(n) / s);
}
```

3.6 Constrained Least Squares

In the examples of Section 3.0.1 we expected all components of the right hand side vectors to be possibly affected by measurement errors. However, it might happen that some data are very reliable and in this case we would like the corresponding equation to be satisfied exactly.

➤ linear least squares problem with linear constraint defined as follows:

Linear least squares problem with linear constraint:

Given: $\mathbf{A} \in \mathbb{R}^{m,n}$, $m \geq n$, $\mathrm{rank}(\mathbf{A}) = n$, $\mathbf{b} \in \mathbb{R}^m$,
$\mathbf{C} \in \mathbb{R}^{p,n}$, $p < n$, $\mathrm{rank}(\mathbf{C}) = p$, $\mathbf{d} \in \mathbb{R}^p$

Find: $\mathbf{x} \in \mathbb{R}^n$ such that $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2 \to \min$ and $\boxed{\mathbf{C}\mathbf{x} = \mathbf{d}}$. (3.6.1)

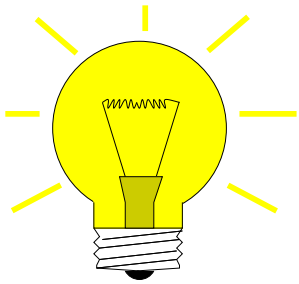
Linear constraint

Here the constraint matrix $\mathbf{C}$ collects all the coefficients of those p equations that are to be satisfied exactly, and the vector $\mathbf{d}$ the corresponding components of the right hand side vector. Conversely, the m equations of the (overdetermined) LSE $\mathbf{A}\mathbf{x} = \mathbf{b}$ cannot be satisfied and are treated in a least squares sense.

3.6.1 Solution via Lagrangian Multipliers

(3.6.2) A saddle point problem

Recall important technique from multidimensional calculus for tackling constrained minimization problems: Lagrange multipliers, see [**?**, Sect. 7.9].



Idea: coupling the constraint using the Lagrange multiplier $\mathbf{m} \in \mathbb{R}^p$

$$\mathbf{x} = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} \max_{\mathbf{m}\in\mathbb{R}^p} L(\mathbf{x},\mathbf{m}) \,, \tag{3.6.3}$$

$$L(\mathbf{x},\mathbf{m}) := \frac{1}{2}\|\mathbf{A}\mathbf{x}-\mathbf{b}\|^2 + \mathbf{m}^\top(\mathbf{C}\mathbf{x}-\mathbf{d}) \,. \tag{3.6.4}$$

L as defined in (3.6.4) is called a Lagrange function. The simple heuristics behind Lagrange multipliers is the observation:

$$\max_{\mathbf{m}\in\mathbb{R}^p} L(\mathbf{x},\mathbf{m}) = \infty, \quad \text{in case } \mathbf{C}\mathbf{x} \neq \mathbf{d}!$$

➥ A minimum in (3.6.3) can only attained, if the constraint is satisfied!

(3.6.3) is called a saddle point problem.

Solution of min-max problem like (3.6.3) is called a saddle point.

Saddle point of $F(x,m) = x^2 - 2xm$ ▷

Note that the function is "flat" in the saddle point •, that is, both the derivative with respect to $\mathbf{x}$ and with respect to $\mathbf{m}$ has to vanish.

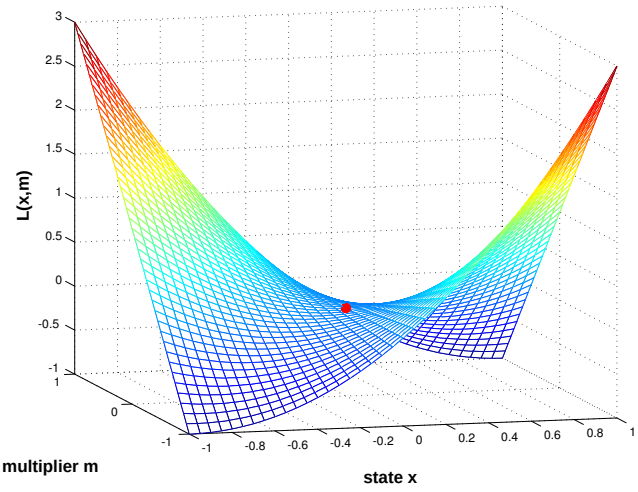


Fig. 126

(3.6.5) Augmented normal equations

In a saddle point the Lagrange function is "flat", that is, all its partial derivatives have to vanish there.

Necessary (and sufficient) conditions for the solution $\mathbf{x}$ of (3.6.3)
(For a similar technique employing multi-dimensional calculus see Rem. 3.1.14)

$$\frac{\partial L}{\partial \mathbf{x}}(\mathbf{x},\mathbf{m}) = \mathbf{A}^\top(\mathbf{A}\mathbf{x}-\mathbf{b}) + \mathbf{C}^\top\mathbf{m} \stackrel{!}{=} 0 \,, \tag{3.6.6a}$$

$$\frac{\partial L}{\partial \mathbf{m}}(\mathbf{x},\mathbf{m}) = \mathbf{C}\mathbf{x}-\mathbf{d} \stackrel{!}{=} 0 \,. \tag{3.6.6b}$$

$$\begin{bmatrix} \mathbf{A}^\top\mathbf{A} & \mathbf{C}^\top \\ \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{m} \end{bmatrix} = \begin{bmatrix} \mathbf{A}^\top\mathbf{b} \\ \mathbf{d} \end{bmatrix} \quad \text{Augmented normal equations (matrix saddle point problem)} \tag{3.6.7}$$

As we know, a direct elimination solution algorithm for (3.6.7) amounts to finding an LU-decomposition of the coefficient matrix. Here we opt for its symmetric variant, the Cholesky decomposition, see Section 2.8.

On the block-matrix level it can be found by considering the equation

$$\begin{bmatrix} \mathbf{A}^\top\mathbf{A} & \mathbf{C}^\top \\ \mathbf{C} & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{R}^\top & 0 \\ \mathbf{G} & -\mathbf{S}^\top \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{G}^\top \\ 0 & \mathbf{S} \end{bmatrix} \quad , \quad \begin{array}{l} \mathbf{R},\mathbf{S} \in \mathbb{R}^{n,n} \text{ upper triangular matrices,} \\ \mathbf{G} \in \mathbb{R}^{p,n} . \end{array}$$

Thus the blocks of the Cholesky factors of the coefficient matrix of the linear system (3.6.7) can be determined in four steps.

① Compute $\mathbf{R}$ from $\mathbf{R}^\top\mathbf{R} = \mathbf{A}^\top\mathbf{A}$ → Cholesky decomposition → Section 2.8,
② Compute $\mathbf{G}$ from $\mathbf{R}^\top\mathbf{G}^\top = \mathbf{C}^\top$ → n forward substitutions → Section 2.3.2,
③ Compute $\mathbf{S}$ from $\mathbf{S}^\top\mathbf{S} = \mathbf{G}\mathbf{G}^\top$ → Cholesky decomposition → Section 2.8.

(3.6.8) Extended augmented normal equations

The same caveats as those discussed for the regular normal equations in Rem. 3.2.3, Ex. 3.2.4, and Rem. 3.2.6, apply to the direct use of the augmented normal equations (3.6.7):

1. their condition number can be much bigger than that of the matrix $\mathbf{A}$,
2. forming $\mathbf{A}^\top\mathbf{A}$ may be vulnerable to roundoff,
3. the matrix $\mathbf{A}^\top\mathbf{A}$ may not be sparse, though $\mathbf{A}$ is.

As in Rem. 3.2.7 also in the case of the augmented normal equations (3.6.7) switching to an extended version by introducing the residual $\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}$ as a new unknown is a remedy, *cf.* (3.2.8). This leads to the following linear system of equations.

$$\begin{bmatrix} -\mathbf{I} & \mathbf{A} & 0 \\ \mathbf{A}^\top & 0 & \mathbf{C}^\top \\ 0 & \mathbf{C} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \\ \mathbf{m} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \\ \mathbf{d} \end{bmatrix} \quad \widehat{=} \quad \begin{array}{c} \text{Extended augmented} \\ \text{normal equations} \end{array} . \tag{3.6.9}$$

3.6.2 Solution via SVD

Idea: Identify the subspace in which the solution can vary without violating the constraint. Since $\mathbf{C}$ has *full rank*, this subspace agrees with the nullspace/kernel of $\mathbf{C}$.

From Lemma 3.4.11 and Ex. 3.4.19 we have learned that the SVD can be used to compute (an orthonormal basis of) the nullspace $\mathcal{N}(\mathbf{C})$. The suggests the following method for solving the constrained linear least squares problem (3.6.1).

① Compute an orthonormal basis of $\mathcal{N}(\mathbf{C})$ using SVD (→ Lemma 3.4.11, (3.4.22)):

$$\mathbf{C} = \mathbf{U}[\Sigma\ 0]\begin{bmatrix} \mathbf{V}_1^\top \\ \mathbf{V}_2^\top \end{bmatrix} \quad , \quad \mathbf{U} \in \mathbb{R}^{p,p}, \Sigma \in \mathbb{R}^{p,p}, \mathbf{V}_1 \in \mathbb{R}^{n,p}, \mathbf{V}_2 \in \mathbb{R}^{n,n-p}$$

$$\blacktriangleright \quad \mathcal{N}(\mathbf{C}) = \mathcal{R}(\mathbf{V}_2) .$$

and the particular solution of the constraint equation

$$\mathbf{x}_0 := \mathbf{V}_1\Sigma^{-1}\mathbf{U}^\top\mathbf{d} .$$

$\blacktriangleright$ Representation of the solution $\mathbf{x}$ of (3.6.1): $\mathbf{x} = \mathbf{x}_0 + \mathbf{V}_2\mathbf{y}, \quad \mathbf{y} \in \mathbb{R}^{n-p}$.

② Insert this representation in (3.6.1). This yields a standard linear least squares problem with coefficient matrix $\mathbf{AV}_2 \in \mathbb{R}^{m,n-p}$ and right hand side vector $\mathbf{b} - \mathbf{Ax}_0 \in \mathbb{R}^m$

$$\|\mathbf{A}(\mathbf{x}_0 + \mathbf{V}_2\mathbf{y}) - \mathbf{b}\|_2 \to \min \quad \Leftrightarrow \quad \|\mathbf{AV}_2\mathbf{y} - (\mathbf{b} - \mathbf{Ax}_0)\| \to \min \ .$$

Learning Outcomes

After having studied the contents of this chapter you should be able to

- give a rigorous definition of the least squares solution of an (overdetermined) linear system of equations,
- state the (extended) normal equations for any overdetermined linear system of equations,
- tell uniqueness and existence of solutions of the normal equations,
- define (economical) QR-decomposition and SVD of a matrix,
- explain the use of QR-decomposition and, in particular, Givens rotations, for solving (overdetermined) linear systems of equations (in least squares sense),
- use SVD to solve (constrained) optimization and low-rank best approximation problems
- formulate the augmented (extended) normal equations for a linearly constrained least squares problem.

Chapter 4

Filtering Algorithms

This chapter continues the theme of *numerical linear algebra*, also covered in Chapter 1, 2, 10. We will come across very special linear transformations ($\leftrightarrow$ matrices) and related algorithms. Surprisingly, these form the basis of a host of very important numerical methods for signal processing.

(4.0.1) Time-discrete signals and sampling

From the perspective of signal processing we can identify

$$\text{vector} \quad \mathbf{x} \in \mathbb{R}^n \quad \leftrightarrow \quad \text{finite discrete (= sampled) signal.}$$

Sampling converts a time-continuous signal, represented by some real-valued physical quantity (pressure, voltage, power, etc.) into a time-discrete signal:

$X = X(t) \mathrel{\hat{=}}$ *time-continuous* signal, $0 \le t \le T$,

"sampling": $x_j = X(j\Delta t)$, $j = 0, \ldots, n-1$,
$n \in \mathbb{N}$, $n\Delta t \le T$.

$\Delta t > 0 \mathrel{\hat{=}}$ time between samples.

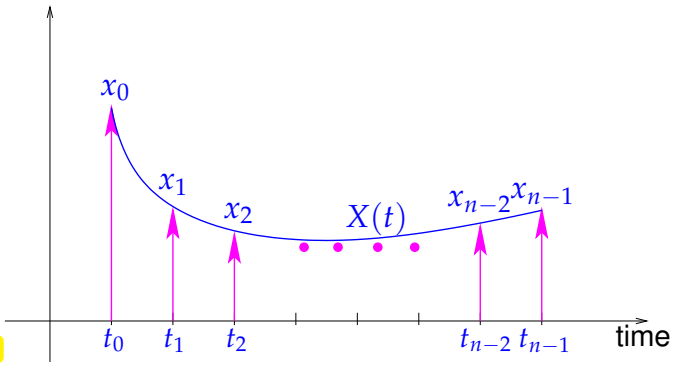


Fig. 127

As already indicated by the indexing the sampled values can be arranged in a vector $\mathbf{x} = [x_0, \ldots, x_{n-1}]^\top \in \mathbb{R}^n$.

Note that in this chapter, as is customary in signal processing, we adopt a C++-style indexing from 0: the components of a vector with length n carry indices $\in \{0, \ldots, n-1\}$.

As an idealization one sometimes considers a signal of infinite duration $X = X(t)$, $-\infty < t < \infty$. In this case sampling yields a bi-infinite time-discrete signal, represented by a sequence $(x_k)_{k\in\mathbb{Z}}$. If this sequence has a finite number of non-zero terms only, then we write $(0, \ldots, x_\ell, x_{\ell+1}, \ldots, x_{n-1}, x_n, 0, \ldots)$.

Contents

4.1 Discrete Convolutions

(4.1.1) Discrete finite linear time-invariant causal channels/filters

Now we study a finite linear time-invariant causal channel (filter), which is widely used model for digital communication channels, e.g. in wireless communication theory. Mathematically speaking, a (discrete) channel/filter is a mapping $F : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z})$ from the vector space $\ell^\infty(\mathbb{Z})$ of bounded input sequences $\{x_j\}_{j\in\mathbb{Z}}$ to bounded output sequences $\{y_j\}_{j\in\mathbb{Z}}$.

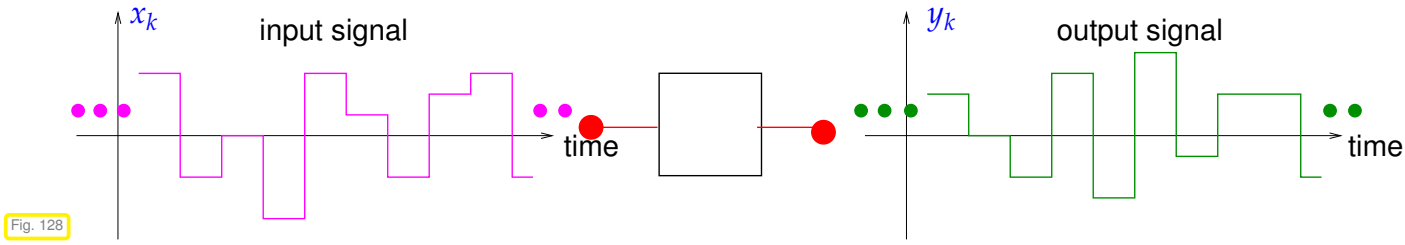


Fig. 128

$$\text{Channel/filter:} \quad F : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z}) \quad , \quad (y_j)_{j\in\mathbb{Z}} = F\big((x_j)_{j\in\mathbb{Z}}\big) \,. \tag{4.1.2}$$

For the description of filters we rely on special input signals, analogous to the description of a linear mapping $\mathbb{R}^n \mapsto \mathbb{R}^m$ through a matrix, that is, its action on unit vectors.

Definition 4.1.3. Impulse response

The impulse response of a channel/filter is the output for a single unit impulse at $t = 0$ as input, that is, the input signal is $x_j = \delta_{j,0} := \begin{cases} 1 & \text{, if } j = 0 \\ 0 & \text{else} \end{cases}$ (Kronecker symbol).

Visualization of the (finite!) impulse response $(\ldots, 0, h_0, \ldots, h_{n-1}, 0, \ldots)$ of a (causal, see Def. 4.1.11 below) channel/filter:

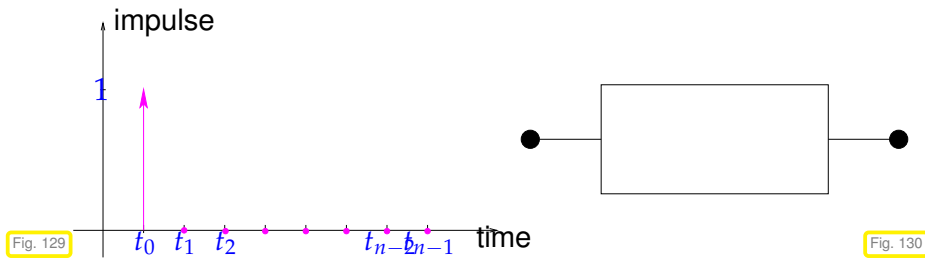


Fig. 129

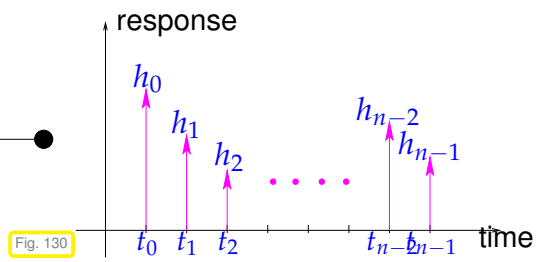


Fig. 130

In order to link digital filters to linear algebra, we have to assume certain properties that are indicated by the attributes "finite ", "linear", "time-invariant" and "causal":

Definition 4.1.4. Finite channel/filter

A filter $F : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z})$ is called finite, if every input signal of finite duration produces an output signal of finite duration,

$$\left\{ \exists M \in \mathbb{N}: \quad |j| > M \quad \Rightarrow \quad x_j = 0 \right\} \quad \Rightarrow \quad \exists N \in \mathbb{N}: \quad |k| > N \quad \Rightarrow \quad (F((x_j)_{j\in\mathbb{Z}}))_k = 0 \,, \tag{4.1.5}$$

➢ The impulse response of a finite filter can be described by a vector $\mathbf{h}$ of finite length n.

It should not matter when exactly signal is fed into the channel. To express this intuition more rigorously we introduce the time shift operator for signals: for $m \in \mathbb{Z}$

$$\mathsf{S}_m : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z}) \quad , \quad \mathsf{S}_m((x_j)_{j\in\mathbb{Z}}) = (x_{j-m})_{j\in\mathbb{Z}} \,. \tag{4.1.6}$$

Definition 4.1.7. Time-invariant channel/filter

A filter $F : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z})$ is called time-invariant, if shifting the input in time leads to the same output shifted in time by the same amount; it *commutes with the time shift operator* from (4.1.6):

$$\forall (x_j)_{j\in\mathbb{Z}} \in \ell^\infty(\mathbb{Z}),\ \forall m \in \mathbb{Z}: \quad F(\mathsf{S}_m((x_j)_{j\in\mathbb{Z}})) = \mathsf{S}_m(F((x_j)_{j\in\mathbb{Z}})) \,. \tag{4.1.8}$$

Definition 4.1.9. Linear channel/filter

A filter $F : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z})$ is called linear, if F is a linear mapping:

$$F(\alpha(x_j)_{j\in\mathbb{Z}} + \beta(y_j)_{j\in\mathbb{Z}}) = \alpha F((x_j)_{j\in\mathbb{Z}}) + \beta F((y_j)_{j\in\mathbb{Z}}) \quad \forall (x_j)_{j\in\mathbb{Z}}, (y_j)_{j\in\mathbb{Z}} \in \ell^\infty(\mathbb{Z}),\ \alpha,\beta \in \mathbb{R} \,. \tag{4.1.10}$$

Slightly rewritten, this means that for all scaling factors $\alpha, \beta \in \mathbb{R}$

$$\text{output}(\alpha \cdot \text{signal 1} + \beta \cdot \text{signal 2}) = \alpha \cdot \text{output}(\text{signal 1}) + \beta \cdot \text{output}(\text{signal 2}) \,.$$

Of course, a signal should not trigger an output before it arrives at the filter; output may depend only on past and present inputs, not on the future.

Definition 4.1.11. Causal channel/filter

A filter $F : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z})$ is called causal (or physical, or nonanticipative), if the output does not start before the input

$$\forall M \in \mathbb{N}: \quad (x_j)_{j\in\mathbb{Z}} \in \ell^\infty(\mathbb{Z}),\ x_j = 0 \quad \forall j \le M \quad \Rightarrow \quad F((x_j)_{j\in\mathbb{Z}})_k = 0 \quad \forall k \le M \,. \tag{4.1.12}$$

Acronym: LT-FIR $\hat{=}$ finite (→ Def. 4.1.4), linear (→ Def. 4.1.9), time-invariant (→ Def. 4.1.7), and causal (→ Def. 4.1.11) filter $F : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z})$

The input response of a finite and causal filter is a sequence of the form $(\ldots,0,h_0,h_1,\ldots,h_{n-1},0,\ldots)$, $n \in \mathbb{N}$. Such an impulse response is depicted in Fig. 130.

(4.1.13) Transmission through finite, time-invariant, linear, causal filters

Let $(\ldots,0,h_0,h_1,\ldots,h_{n-1},0,\ldots)$, $n \in \mathbb{N}$, be the impulse response (→ 4.1.3) of a finite (→ Def. 4.1.4), linear (→ Def. 4.1.9), time-invariant (→ Def. 4.1.7), and causal (→ Def. 4.1.11) filter (LT-FIR) $F : \ell^\infty(\mathbb{Z}) \to \ell^\infty(\mathbb{Z})$:

$$F(\left(\delta_{j,0}\right)_{j\in\mathbb{Z}}) = (\ldots,0,h_0,h_1,\ldots,h_{n-1},0,\ldots) .$$

Owing to time-invariance we already know the response to a shifted unit pulse

$$F(\left(\delta_{j,k}\right)_{j\in\mathbb{Z}}) = \left(h_{j-k}\right)_{j\in\mathbb{Z}} = \Big(\cdots \quad 0 \quad \underset{\substack{\uparrow\\ t=k\Delta t}}{h_0} \quad h_1 \quad \cdots \quad \underset{\substack{\uparrow\\ t=(k+n-1)\Delta t}}{h_{n-1}} \quad 0 \quad \cdots \Big) .$$

Every finite input signal $(\ldots,0,x_0,x_1,\ldots,x_{m-1},0,\ldots) \in \ell^\infty(\mathbb{Z})$ can be written as the superposition of scaled unit impulses, which, in turns, are time-shifted copies of a unit pulse at $t = 0$:

$$\left(x_j\right)_{j\in\mathbb{Z}} = \sum_{k=0}^{m-1} x_k \left(\delta_{j,k}\right)_{j\in\mathbb{Z}} = \sum_{k=0}^{m-1} x_k \mathsf{S}_k \Big(\left(\delta_{j,0}\right)_{j\in\mathbb{Z}} \Big) ,,$$

where S_k is the time-shift operator from (4.1.6). Applying the filter on both sides of this equation and using linarity leads to the general formula for the output signal $\left(y_j\right)_{j\in\mathbb{Z}}$

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \\ \vdots \\ \vdots \\ y_{m+n-3} \\ y_{m+n-2} \end{bmatrix} = x_0 \begin{bmatrix} h_0 \\ \vdots \\ h_{n-1} \\ 0 \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} + x_1 \begin{bmatrix} 0 \\ h_0 \\ \vdots \\ h_{n-1} \\ 0 \\ \vdots \\ \vdots \\ 0 \end{bmatrix} + x_2 \begin{bmatrix} 0 \\ 0 \\ h_0 \\ \vdots \\ h_{n-1} \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \cdots + x_{m-1} \begin{bmatrix} 0 \\ \vdots \\ \vdots \\ \vdots \\ 0 \\ h_0 \\ \vdots \\ h_{n-1} \end{bmatrix} .$$

Thius, in compact notation we can write the non-zero components of the output signal $\left(y_j\right)_{j\in\mathbb{Z}}$ as

channel is causal and finite!

$$y_k = \sum_{j=0}^{m+n-1} h_{k-j} x_j \quad , \quad k = 0,\ldots,m+n-2 \quad (h_j := 0 \text{ for } j < 0 \text{ and } j \geq n) . \tag{4.1.14}$$

Summary of the above considerations:

Superposition of impulse responses

The output $(\ldots, 0, y_0, y_1, y_2, \ldots)$ of a finite, time-invariant, linear, and causal channel for finite length input $\mathbf{x} = (\ldots, 0, x_0, \ldots, x_{n-1}, 0, \ldots) \in \ell^\infty(\mathbb{Z})$ is
a superposition of x_j-weighted $j\Delta t$ time-shifted impulse responses.

Example 4.1.16 (Visualization: superposition of impulse responses)

The following diagrams give a visual display of considerations of § 4.1.13, namely of the superposition of impulse responses for a particular finite, time-invariant, linear, and causal filter (LT-FIR), and an input signal of duration $3\Delta t$, $\Delta t \hat{=}$ time between samples.

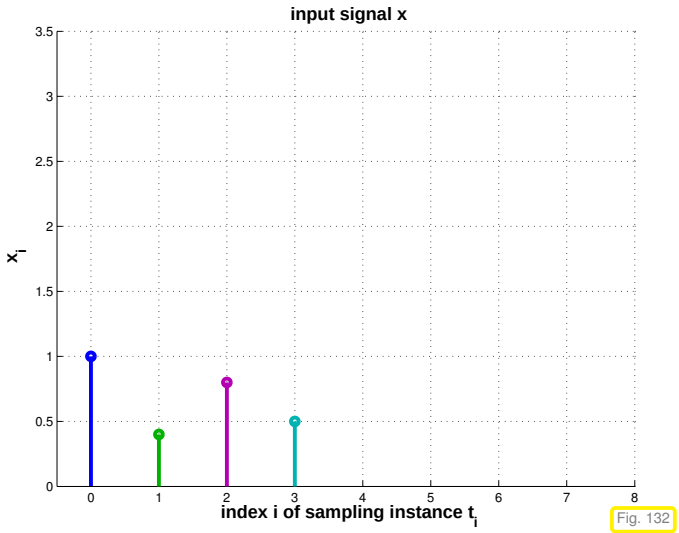


Fig. 131

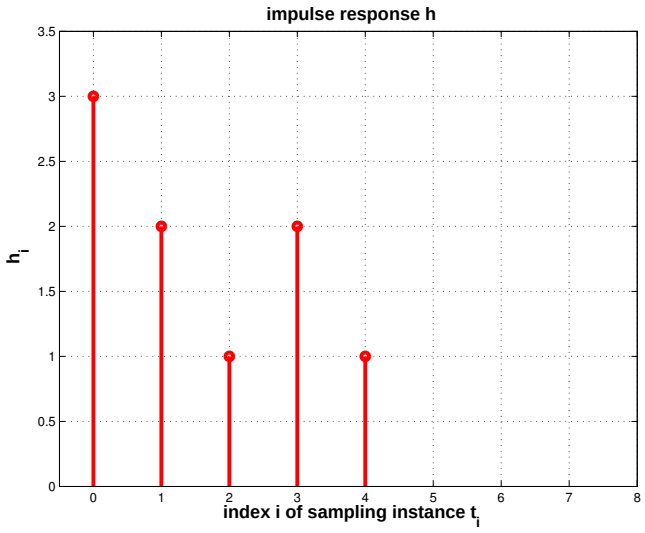


Fig. 132

Output = linear superposition of impulse responses:

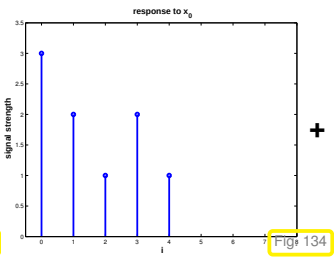


Fig. 133

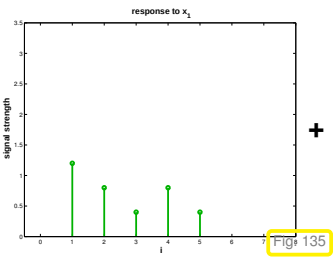


Fig. 134

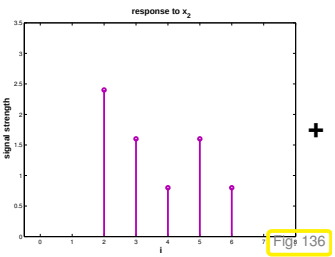


Fig. 135

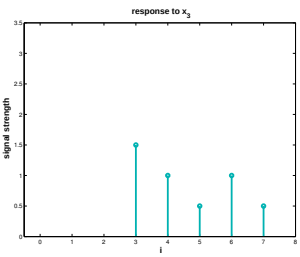


Fig. 136

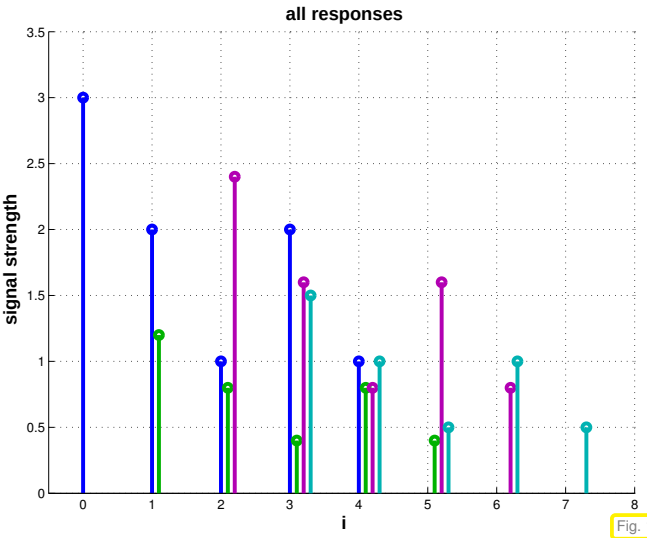


Fig. 137

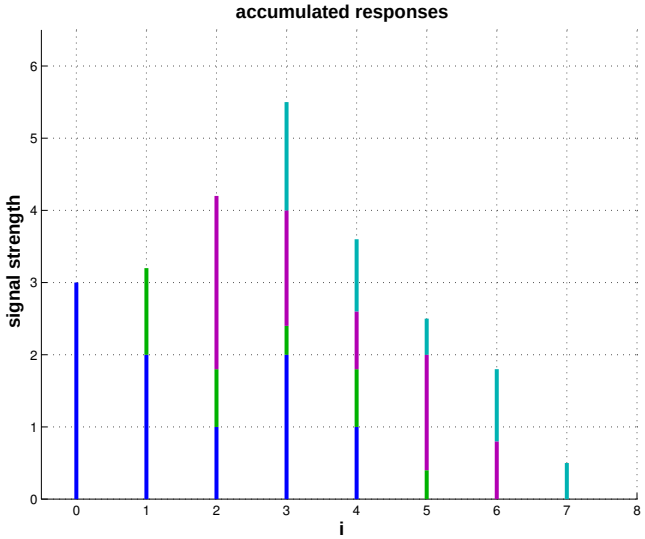


Fig. 138

Remark 4.1.17 (Reduction to finite signals)

We consider a finite (→ Def. 4.1.4), linear (→ Def. 4.1.9), time-invariant (→ Def. 4.1.7), and causal (→ Def. 4.1.11) filter (LT-FIR) with impulse response $(\ldots,0,h_0,h_1,\ldots,h_{n-1},0,\ldots)$, $n \in \mathbb{N}$. From (4.1.14) we learn that

$$\boxed{\text{duration(output signal)} \leq \text{duration(input signal)} + \text{duration(impulse response)}} .$$

Therefore, if we know that all input signals are of the form $(\ldots,x_0,x_1,\ldots,x_{m-1},0,\ldots)$, we can model them as vectors $\mathbf{x} = [x_0,\ldots,x_{m-1}]^\top \in \mathbb{R}^m$, *cf.* § 4.0.1, and the filter can be viewed as a linear mapping $F : \mathbb{R}^n \to \mathbb{R}^{m+n-1}$, which takes us to the realm of linear algebra.

Thus, for the filter we have a matrix representation of (4.1.14). Writing $\mathbf{y} = [y_0,\ldots,y_{2n-2}]^\top \in \mathbb{R}^{2n-1}$ for the vector of the output signal we find in the case $m = n$

$$\begin{bmatrix} y_0 \\ \vdots \\ \\ \\ \\ \\ \vdots \\ y_{2n-2} \end{bmatrix} = \begin{bmatrix} h_0 & 0 & \cdots & & 0 \\ h_1 & \ddots & \ddots & & \\ & \ddots & & \ddots & 0 \\ h_{n-1} & & \ddots & & h_0 \\ 0 & \ddots & & \ddots & h_1 \\ & \ddots & \ddots & & \\ 0 & & 0 & h_{n-1} & \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ \\ \\ \\ \\ \vdots \\ x_{n-1} \end{bmatrix} . \tag{4.1.18}$$

Example 4.1.19 (Multiplication of polynomials)

"Surprisingly" the bilinear operation (4.1.14) that takes two input vectors and produces an output vector with double the number of entries (-1) also governs the multiplication of polynomials:

$$p(z) = \sum_{k=0}^{n-1} a_k z^k \,, \quad q(z) = \sum_{k=0}^{n-1} b_k z^k \quad \blacktriangleright \quad (pq)(z) = \sum_{k=0}^{2n-2} \underbrace{\left(\sum_{j=0}^{k} a_j b_{k-j} \right)}_{=:c_k} z^k \tag{4.1.20}$$

Here the roles of h_k, x_k are played by the a_k and b_k.

➢ the coefficients of the product polynomial can be obtained through an operation similar to finite, time-invariant, linear, and causal (LT-FIR) filtering!

(4.1.21) Discrete convolution

Both in (4.1.14) and (4.1.20) we recognize the same pattern of a particular *bi-linear* combination of

- discrete signals in § 4.1.13,
- polynomial coefficient sequences in Ex. 4.1.19.

Definition 4.1.22. Discrete convolution

Given $\mathbf{x} = [x_0, \ldots, x_{n-1}]^\top \in \mathbb{K}^n$, $\mathbf{h} = [h_0, \ldots, h_{n-1}]^\top \in \mathbb{K}^n$ their discrete convolution is the vector $\mathbf{y} \in \mathbb{K}^{2n-1}$ with components

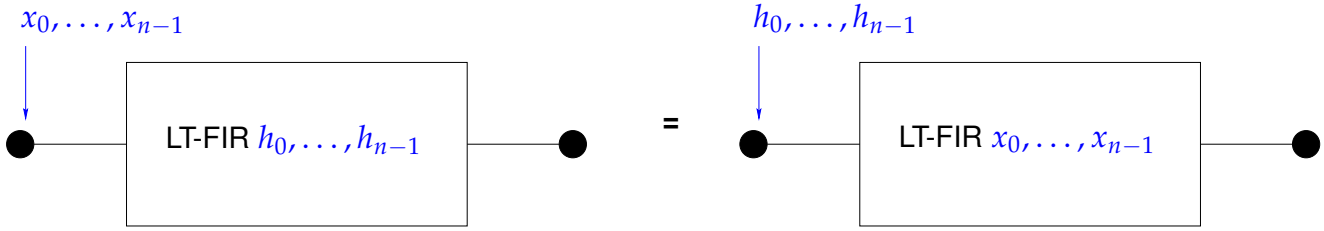
$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j \,, \quad k = 0, \ldots, 2n-2 \quad (h_j := 0 \text{ for } j < 0) \,. \tag{4.1.23}$$

✎ Notation for discrete convolution (4.1.23): $\mathbf{y} = \mathbf{h} * \mathbf{x}$.

Defining $x_j := 0$ for $j < 0$, we find that *discrete convolution is commutative*:

$$y_k = \sum_{j=0}^{n-1} h_{k-j} x_j = \sum_{l=0}^{n-1} h_l x_{k-l} \,, \quad k = 0, \ldots, 2n-2 \,, \qquad \text{(that is, } \mathbf{h} * \mathbf{x} = \mathbf{x} * \mathbf{h} \text{)} \,,$$

obtained by index transformation $l \leftarrow k - j$.



Remark 4.1.24 (Multiplication of polynomials)

The formula (4.1.23) for the discrete convolution also occurs in a context completely detached from signal processing.

Consider two polynomials in t of degree $n-1$, $n \in \mathbb{N}$,

$$p(t) = \sum_{j=0}^{n-1} a_j t^j \,, \quad q(t) = \sum_{j=0}^{n-1} b_j t^j \,, \qquad a_j, b_j \in \mathbb{K} \,.$$

Their product pq will be a polynomial of degree $2n-2$:

$$(pq)(t) = \sum_{j=0}^{2n-2} c_j t^j \,, \quad c_j = \sum_{\ell=\max\{0, j-(n-1)\}}^{\min\{j, n-1\}} a_\ell b_{j-\ell} \,, \quad j = 0, \ldots, 2n-2 \,. \tag{4.1.25}$$

Let us introduce dummy coefficients for $p(t)$ and $q(t)$, a_j, b_j, $j = 2n, \ldots, 2n-2$, all set to 0. This can be easily done in a computer code by resizing the coefficient vectors of p and q and filling the new entries

with zeros. The above formula for c_j can then be rewritten as

$$c_j = \sum_{\ell=0}^{j} a_\ell b_{j-\ell} \,, \quad j = 0, \ldots, 2n-2 \,. \tag{4.1.26}$$

Hence, the coefficients of the product polynomial can be obtained as the discrete convolution of the coefficient vectors of p and q:

$$[c_0\, c_1\, \ldots\, c_{2n-2}]^\top = \mathbf{a} * \mathbf{b} \quad ! \tag{4.1.27}$$

Moreover, this provides another proof for the commutativity of discrete convolution.

Remark 4.1.28 (Convolution of sequences)

The notion of a discrete convolution of Def. 4.1.22 naturally extends to sequences $\in \ell^\infty(\mathbb{N}_0)$, that is, bounded mappings $\mathbb{N}_0 \mapsto \mathbb{K}$: the (discrete) convolution of two sequences $(x_j)_{j\in\mathbb{N}_0}$, $(y_j)_{j\in\mathbb{N}_0}$ is the sequence $(z_j)_{j\in\mathbb{N}_0}$ defined by

$$z_k := \sum_{j=0}^{k} x_{k-j} y_j = \sum_{j=0}^{k} x_j y_{k-j} \,, \quad k \in \mathbb{N}_0 \,.$$

In this context recall the product formula for power series, Cauchy product, which can be viewed as a multiplication rule for "infinite polynoials" = power series.

Remark 4.1.29 (Linear filtering of periodic signals)

An n-periodic signal ($n \in \mathbb{N}$) is a sequence $(x_j)_{j\in\mathbb{Z}}$ satisfying $\boxed{x_{j+n} = x_j \quad \forall j \in \mathbb{Z}}$.

➢ An n-periodic signal $(x_j)_{j\in\mathbb{Z}}$ uniquely determined by the values $x_0, \ldots, x_{n-1}$ and can be associated with a vector $\mathbf{x} = [x_0, \ldots, x_{n-1}]^\top \in \mathbb{R}^n$.

Whenever the input signal of a finite, time-invariant filter is n-periodic, so will be the output signal. Thus, in the n-periodic setting, a causal, *linear*, and time-invariant filter (LT-FIR) will give rise to a *linear* mapping $\mathbb{R}^n \mapsto \mathbb{R}^n$ according to

$$y_k = \sum_{j=0}^{n-1} p_{k-j} x_j \quad \text{for some} \quad p_0, \ldots, p_{n-1} \in \mathbb{R} \,, \quad p_k := p_{k-n} \text{ for all } k \in \mathbb{Z} \,. \tag{4.1.30}$$

In matrix notation (4.1.30) reads

$$\begin{bmatrix} y_0 \\ \vdots \\ \\ \\ \vdots \\ y_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} p_0 & p_{n-1} & p_{n-2} & \cdots & & \cdots & p_1 \\ p_1 & p_0 & p_{n-1} & & & & \vdots \\ p_2 & p_1 & p_0 & \ddots & & & \\ \vdots & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ \vdots & & & & \ddots & \ddots & p_{n-1} \\ p_{n-1} & \cdots & & & & p_1 & p_0 \end{bmatrix}}_{=:\mathbf{P}} \begin{bmatrix} x_0 \\ \vdots \\ \\ \\ \vdots \\ x_{n-1} \end{bmatrix} . \tag{4.1.31}$$

where $(\mathbf{P})_{ij} = p_{i-j}$, $1 \le i,j \le n$, with $p_j := p_{j+n}$ for $1-n \le j < 0$.

Note that the coefficients $p_0,\dots,p_{n-1}$, which can be regarded as periodic impulse response, do **not** agree with the impulse response ($\to$ Def. 4.1.3) of the filter. However, they can easily be deduced from it, keeping in mind that the n-periodic sequence is the output signal for the input $\sum_{k\in\mathbb{Z}}(\delta_{nk,j})_{j\in\mathbb{Z}}$. If $(\dots,h_0,h_1,\dots,h_m,0,\dots)$ is the impulse response of LT-FIR filter, then

$$p_j = \sum_{\ell\in\mathbb{Z}} h_{j+\ell n}\,, \quad j\in\{0,\dots,n-1\}\,. \tag{4.1.32}$$

The following special variant of a discrete convolution operation is motivated by the preceding Rem. 4.1.29.

Definition 4.1.33. Discrete periodic convolution

The discrete periodic convolution of two n-periodic sequences $(p_k)_{k\in\mathbb{Z}}$, $(x_k)_{k\in\mathbb{Z}}$ yields the n-periodic sequence

$$(y_k) := (p_k) *_n (x_k) \quad , \quad y_k := \sum_{j=0}^{n-1} p_{k-j}x_j = \sum_{j=0}^{n-1} x_{k-j}p_j\,, \quad k\in\mathbb{Z}\,.$$

✎ notation for discrete periodic convolution: $(p_k) *_n (x_k)$

Since n-periodic sequences can be identified with vectors in $\mathbb{K}^n$ (see above), we can also introduce the discrete periodic convolution of vectors:

Def. 4.1.33 ➢ discrete periodic convolution of vectors: $\mathbf{y} = \mathbf{p} *_n \mathbf{x} \in \mathbb{K}^n$, $\mathbf{p},\mathbf{x}\in\mathbb{K}^n$.

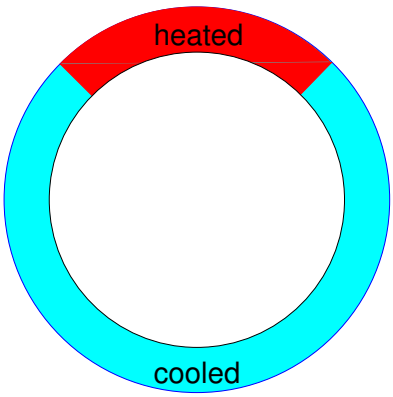
Example 4.1.34 (Radiative heat transfer)

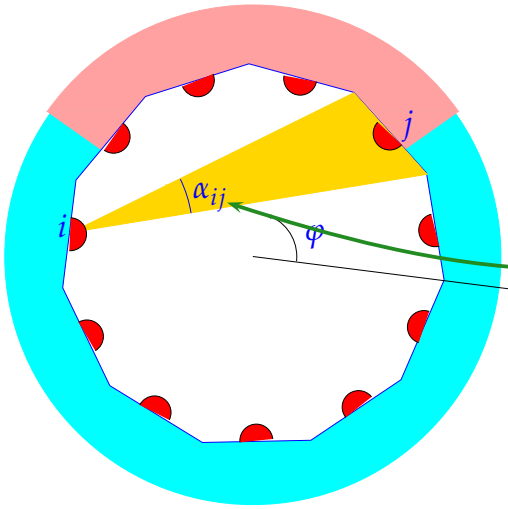
Beyond signal processing discrete periodic convolutions occur in many mathematical models:

An engineering problem:

- cylindrical pipe,
- heated on part Γ_H of its perimeter ($\to$ prescribed heat flux),
- cooled on remaining perimeter Γ_K ($\to$ constant heat flux).

Task: compute local heat fluxes.





Modeling (discretization):

- approximation by regular n-polygon, edges Γ_j,
- isotropic radiation of each edge Γ_j (power I_j),

radiative heat flow $\Gamma_j \to \Gamma_i$: $P_{ji} := \frac{\alpha_{ij}}{\pi} I_j$,

opening angle: $\alpha_{ij} = \pi\, \gamma_{|i-j|}$, $1 \le i,j \le n$,

power balance: $\underbrace{\sum_{i=1,i\neq j}^{n} P_{ji}}_{=I_j} - \sum_{i=1,i\neq j}^{n} P_{ij} = Q_j\,.$ (4.1.35)

$Q_j \mathrel{\hat{=}}$ heat flux through Γ_j, satisfies

$$Q_j := \int_{\frac{2\pi}{n}(j-1)}^{\frac{2\pi}{n}j} q(\varphi)\,\mathrm{d}\varphi\,, \quad q(\varphi) := \begin{cases} \text{local heating} & \text{, if } \varphi \in \Gamma_H\,, \\ -\frac{1}{|\Gamma_K|}\int_{\Gamma_H} q(\varphi)\,\mathrm{d}\varphi & \text{(const.), if } \varphi \in \Gamma_K\,. \end{cases}$$

$$\text{(4.1.35)} \quad \Rightarrow \quad \text{LSE:} \quad I_j - \sum_{i=1,i\neq j}^{n} \frac{\alpha_{ij}}{\pi} I_i = Q_j\,, \quad j = 1,\ldots,n\,.$$

e.g. $n = 8$:

$$\begin{bmatrix} 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 \\ -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 \\ -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 \\ -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_2 & -\gamma_4 \\ -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 & -\gamma_3 \\ -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 & -\gamma_2 \\ -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 & -\gamma_1 \\ -\gamma_1 & -\gamma_2 & -\gamma_3 & -\gamma_4 & -\gamma_3 & -\gamma_2 & -\gamma_1 & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \\ I_6 \\ I_7 \\ I_8 \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \\ Q_5 \\ Q_6 \\ Q_7 \\ Q_8 \end{bmatrix}. \quad \text{(4.1.36)}$$

This is a linear system of equations with symmetric, singular, and (by Lemma 9.1.5, $\sum \gamma_i \le 1$) positive semidefinite ($\to$ Def. 1.1.8) system matrix.

Note that the matrices from (4.1.31) and (4.1.36) have the same structure!

Also observe that the LSE from (4.1.36) can be written by means of the discrete periodic convolution ($\to$ Def. 4.1.33) of vectors $\mathbf{y} = (1, -\gamma_1, -\gamma_2, -\gamma_3, -\gamma_4, -\gamma_3, -\gamma_2, -\gamma_1)$, $\mathbf{x} = (I_1,\ldots,I_8)$

$$\text{(4.1.36)} \quad \leftrightarrow \quad \mathbf{y} *_8 \mathbf{x} = [Q_1,\ldots,Q_8]^\top\,.$$

(4.1.37) Circulant matrices

In Ex. 4.1.34 we have already seen a coefficient matrix of a special form, which is common enough to warrant giving it a particular name:

Definition 4.1.38. Circulant matrix → [?, Sect. 54]

A matrix $\mathbf{C} = [c_{ij}]_{i,j=1}^{n} \in \mathbb{K}^{n,n}$ is circulant

$:\Leftrightarrow \quad \exists\, (p_k)_{k\in\mathbb{Z}}$ n-periodic sequence: $c_{ij} = p_{j-i}$, $1 \le i,j \le n$.

✎ Notation: We write $\operatorname{circul}(\mathbf{p}) \in \mathbb{K}^{n,n}$ for the circulant matrix generated by the periodic sequence/vector $\mathbf{p} = [p_0,\ldots,p_{n-1}]^\top \in \mathbb{K}^n$

☞ Circulant matrix has constant (main, sub- and super-) diagonals (for which indices $j-i = \text{const.}$).

☞ columns/rows arise by *cyclic permutation* from first column/row.

Similar to the case of banded matrices (→ Section 2.7.6):

"information content" of circulant matrix $\mathbf{C} \in \mathbb{K}^{n,n}$ = n numbers $\in \mathbb{K}$.
(obviously, one vector $\mathbf{u} \in \mathbb{K}^n$ enough to define circulant matrix $\mathbf{C} \in \mathbb{K}^{n,n}$)

Structure of a generic circulant matrix ("constant diagonals"):

$$\operatorname{circul}(\mathbf{p}) = \begin{bmatrix} p_0 & p_1 & p_2 & \cdots & & \cdots & p_{n-1} \\ p_{n-1} & p_0 & & & & & p_{n-2} \\ p_{n-2} & & & & & & \vdots \\ \vdots & & & & & & \\ & & & & & & \vdots \\ \vdots & & & & & & p_1 \\ p_2 & & & & & & p_0 \\ p_1 & p_2 & \cdots & & \cdots & p_{n-1} & \end{bmatrix}$$

Write $\mathbf{Z}((u_k)) \in \mathbb{K}^{n,n}$ for the circulant matrix generated by the n-periodic sequence $(u_k)_{k\in\mathbb{Z}}$. Denote by $\mathbf{y} := (y_0,\ldots,y_{n-1})^\top$, $\mathbf{x} = (x_0,\ldots,x_{n-1})^\top$ the vectors associated to n-periodic sequences.

Then the commutativity of the discrete periodic convolution (→ Def. 4.1.33) involves

$$\operatorname{circul}(\mathbf{x})\mathbf{y} = \operatorname{circul}(\mathbf{y})\mathbf{x}\,. \tag{4.1.39}$$

Remark 4.1.40 (Reduction to periodic convolution)

Recall discrete convolution (→ Def. 4.1.22) of two vectors $\mathbf{a} = (a_0,\ldots,a_{n-1})^\top \in \mathbb{K}^n$, $\mathbf{b} = (b_0,\ldots,b_{n-1})^\top \in \mathbb{K}^n$.

$$z_k := (\mathbf{a} * \mathbf{b})_k = \sum_{j=0}^{n-1} a_j b_{k-j}\,, \quad k = 0,\ldots,2n-2\,.$$

Now expand $a_0,\ldots,a_{n-1}$ and $b_0,\ldots,b_{n-1}$ to $2n-1$-periodic sequences by zero padding.

$$x_k := \begin{cases} a_k & \text{, if } 0 \le k < n\,, \\ 0 & \text{, if } n \le k < 2n-1 \end{cases}\,, \quad y_k := \begin{cases} b_k & \text{, if } 0 \le k < n\,, \\ 0 & \text{, if } n \le k < 2n-1\,, \end{cases} \tag{4.1.41}$$

and periodic extension: $x_k = x_{2n-1+k}$, $y_k = y_{2n-1+k}$ for all $k \in \mathbb{Z}$. The zero components prevent interaction of different periods:

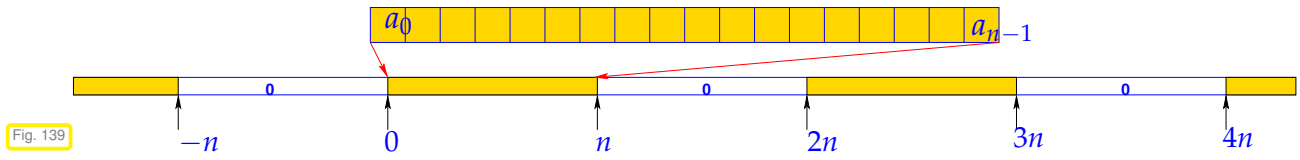


Fig. 139

$$\blacktriangleright \quad (\mathbf{a} * \mathbf{b})_k = (\mathbf{x} *_{2n-1} \mathbf{y})_k \,, \quad k = 0, \ldots, 2n-2 \,. \tag{4.1.42}$$

In the spirit of (4.1.18) we can switch to a matrix view of the reduction to periodic convolution:

$$\begin{bmatrix} z_0 \\ \vdots \\ \\ \\ \\ \\ \vdots \\ z_{2n-2} \end{bmatrix} = \underbrace{\begin{bmatrix} y_0 & 0 & \cdots & 0 & y_{n-1} & \cdots & y_1 & y_0 \\ y_1 & & & & 0 & & & \\ & & & & & & & \\ & & & 0 & & & & \\ y_{n-1} & & & & y_1 & y_0 & 0 & \cdots & 0 & y_{n-1} \\ 0 & & & & & y_1 & y_0 & & & 0 \\ & & & & & & & & & \\ 0 & & & & 0 & y_{n-1} & & & y_1 & y_0 \end{bmatrix}}_{\text{a } (2n-1)\times(2n-1) \text{ circulant matrix!}} \begin{bmatrix} x_0 \\ \vdots \\ \\ \\ x_{n-1} \\ 0 \\ \vdots \\ \\ \\ 0 \end{bmatrix} . \tag{4.1.43}$$

Discrete convolution can be realized by multiplication with a circulant matrix (→ 4.1.38)

4.2 Discrete Fourier Transform (DFT)

Algorithms dealing with circulant matrices make use of their very special spectral properties. Full understanding requires familiarity with the theory of eigenvalues and eigenvectors of matrices from linear algebra, see [**?**, Ch. 7], [**?**, Ch. 9].

Experiment 4.2.1 (Eigenvectors of circulant matrices)

Now we are about to discover a very deep truth . . .

Experimentally, we examine the eigenvalues and eigenvectors of two random 8×8 circulant matrices $\mathbf{C}_1$, $\mathbf{C}_2$ ($\rightarrow$ Def. 4.1.38), generated from random vectors with entries even distributed in $[0,1]$, `VectorXd::Random(n)`.

eigenvalues (real part) $\triangleright$

Little relationship between (complex!) eigenvalues can be observed, as can be expected from random matrices with entries $\in [0,1]$.

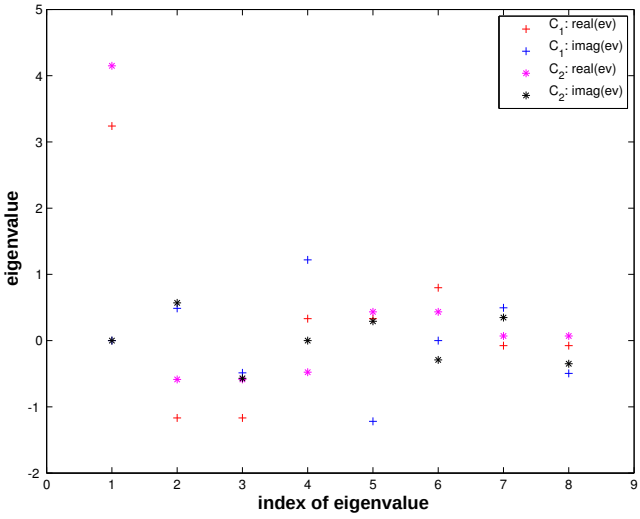
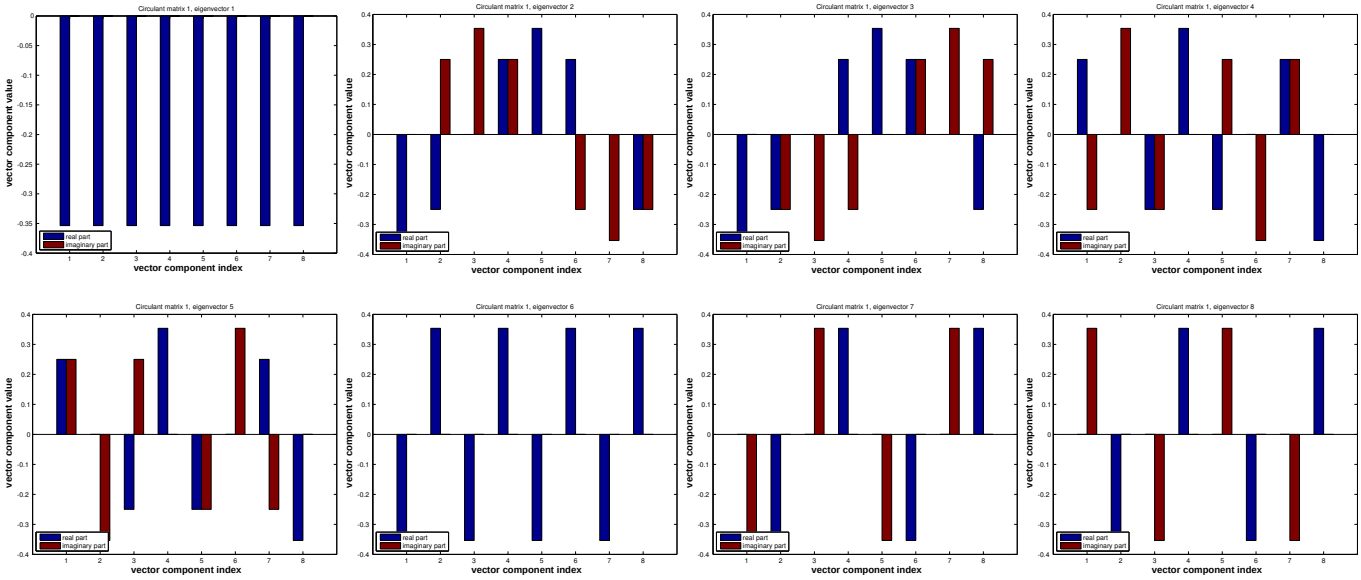


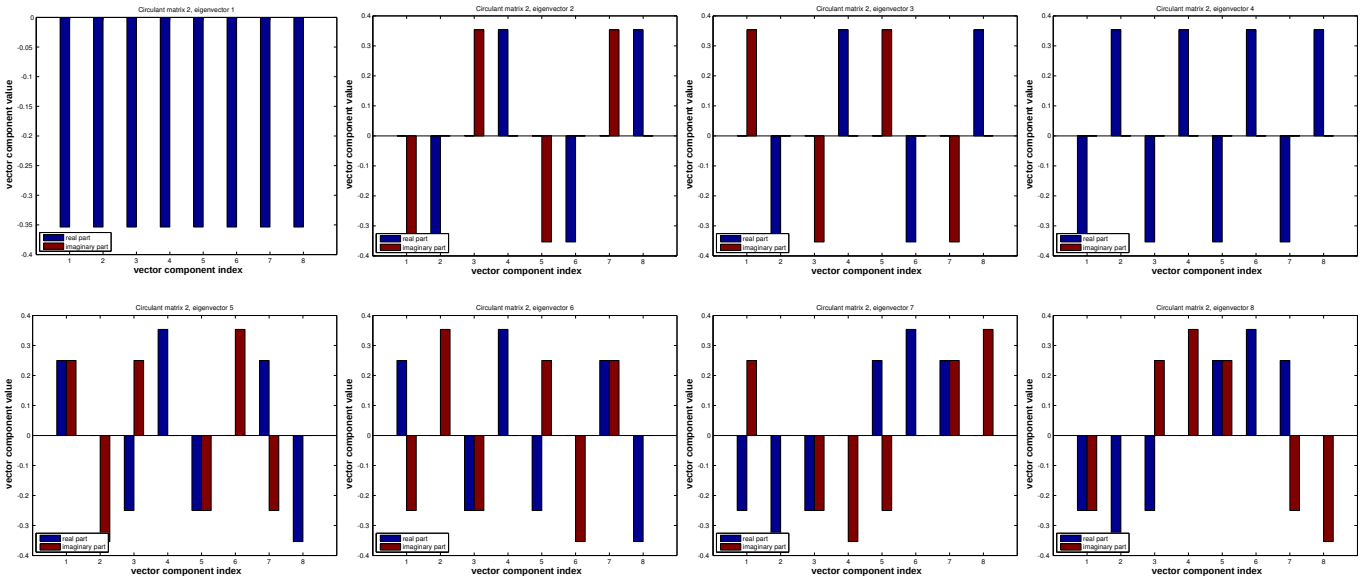
Fig. 140

Now: the **surprise** ...

Eigenvectors of matrix $\mathbf{C}_1$, visualized through the size of the real and imaginary parts of their components.

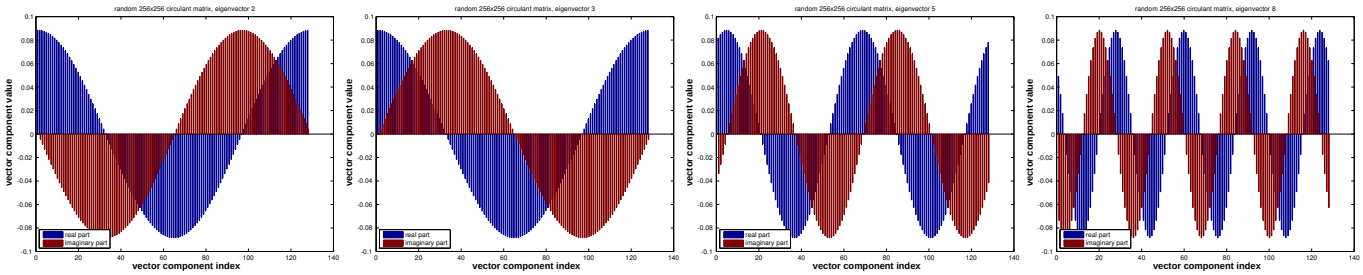


Eigenvectors of matrix $\mathbf{C}_2$



Observation: different random circulant matrices have the same eigenvectors!

Eigenvectors of C = gallery('circul',(1:128)); :



The eigenvectors remind us of sampled *trigonometric functions* $\cos(k/n)$, $\sin(k/n)$, $k = 0, \ldots, n-1$!

Remark 4.2.2 (Eigenvectors of commuting matrices)

An abstract result from linear algebra puts the surprising observation made in Exp. 4.2.1 in a wider context.

Theorem 4.2.3. Commuting matrices have the same eigenvectors

If $\mathbf{A}, \mathbf{B} \in \mathbb{K}^{n,n}$ commute, that is, $\mathbf{AB} = \mathbf{BA}$, and $\mathbf{A}$ has n distinct eigenvalues, then the eigenspaces of $\mathbf{A}$ and $\mathbf{B}$ coincide.

Proof. Let $\mathbf{v} \in \mathbb{K}^n \setminus \{\mathbf{0}\}$ be an eigenvector of $\mathbf{A}$ with eigenvalue λ. Then

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0 \quad \Rightarrow \quad \mathbf{B}(\mathbf{A} - \lambda\mathbf{I})\mathbf{v} = 0 \quad \overset{\mathbf{BA}=\mathbf{AB}}{\Rightarrow} \quad (\mathbf{A} - \lambda\mathbf{I})\mathbf{Bv} = 0\,.$$

Since in the case of n distinct eigenvalues $\dim \mathcal{N}(\mathbf{A} - \lambda\mathbf{I}) = 1$, we conclude that there is $\xi \in\in \mathbb{K}$: $\mathbf{Bv} = \xi\mathbf{v}$, $\mathbf{v}$ is an eigenvector of $\mathbf{B}$. Since the eigenvector of $\mathbf{A}$ span $\mathbb{K}^n$, there cannot be eigenvectors of $\mathbf{B}$ that are not eigenvectors of $\mathbf{A}$.

Moreover, there is a basis of $\mathbb{K}^n$ consisting of eigenvectors of $\mathbf{B}$; $\mathbf{B}$ can be diagonalized. □

Next, by straightforward calculation one verifies that every circulant matrix commutes with the unitary and circulant cyclic permutation matrix

$$\mathbf{S} = \begin{bmatrix} 0 & 0 & 0 & \cdots & \cdots & 0 & 1 \\ 1 & 0 & 0 & & & \vdots & 0 \\ 0 & 1 & 0 & \ddots & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & 0 & 0 \\ 0 & \cdots & & \cdots & 0 & 1 & 0 \end{bmatrix}. \tag{4.2.4}$$

As a unitary matrix $\mathbf{S}$ can be diagonalized. Observe that $\mathbf{S}^n - \mathbf{I} = \mathbf{O}$, that is the minimal polynomial of $\mathbf{S}$ is $\xi \mapsto \xi^n - 1$, which is irreducible, because it has n distinct roots (of unity). Therefore, by Thm. 4.2.3, $\mathbf{S}$ has n different eigenvalues and every eigenvector of $\mathbf{S}$ is also eigenvector of any circulant matrix.

Remark 4.2.5 (Why using $\mathbb{K} = \mathbb{C}$?)

In Exp. 4.2.1 we saw that we get *complex* eigenvalues/eigenvectors for general circulant matrices. More generally, in many cases real matrices can be diagonalized only in $\mathbb{C}$, which is the ultimate reason for the importance of complex numbers.

Complex numbers also allow an elegant handling on trigonometric functions: recall from analysis the unified treatment of trigonometric functions via the *complex exponential function*

$$\exp(it) = \cos(t) + i\sin(t)\,, \quad t \in \mathbb{R}\,.$$

$\mathbb{C}$! The field of complex numbers $\mathbb{C}$ is the *natural framework* for the analysis of linear, time-invariant filters, and the development of algorithms for circulant matrices.

(4.2.6) Eigenvectors of circulant matrices

Now we verify by direct computations that circulant matrices all have a particular set of eigenvectors. This will entail computing in $\mathbb{C}$, *cf.* Rem. 4.2.15.

✎ notation: nth root of unity $\omega_n := \exp(-2\pi i/n) = \cos(2\pi/n) - i\sin(2\pi/n)$, $n \in \mathbb{N}$

satisfies $\overline{\omega}_n = \omega_n^{-1}$, $\boxed{\omega_n^n = 1}$, $\omega_n^{n/2} = -1$, $\omega_n^k = \omega_n^{k+n}$ $\forall\, k \in \mathbb{Z}$, (4.2.7)

$$\sum_{k=0}^{n-1} \omega_n^{kj} = \begin{cases} n & \text{, if } j = 0 \mod n\,, \\ 0 & \text{, if } j \neq 0 \mod n\,. \end{cases} \tag{4.2.8}$$

(4.2.8) is a simple consequence of the geometric sum formula

$$\sum_{k=0}^{n-1} q^k = \frac{1-q^n}{1-q} \quad \forall\, q \in \mathbb{C}\setminus\{1\}\,, \quad n \in \mathbb{N}\,. \tag{4.2.9}$$

$$\Rightarrow \quad \sum_{k=0}^{n-1} \omega_n^{kj} = \frac{1-\omega_n^{nj}}{1-\omega_n^j} = \frac{1-\exp(-2\pi ij)}{1-\exp(-2\pi ij/n)} = 0\,,$$

because $\exp(-2\pi ij) = \omega_n^{nj} = (\omega_n^n)^j = 1$ for all $j \in \mathbb{Z}$.

In expressions like ω_n^{kl} the term "kl" will always designate an exponent and will never play the role of a superscript.

Now we want to confirm the conjecture gleaned from Exp. 4.2.1 that vectors with powers of roots of unity are eigenvectors for any circulant matrix. We do this by simple and straightforward computations:

Consider: circulant matrix $\mathbf{C} \in \mathbb{C}^{n,n}$ circulant matrix ($\rightarrow$ Def. 4.1.38), with $c_{ij} = u_{i-j}$, for a n-periodic sequence $(u_k)_{k\in\mathbb{Z}}$, $u_k \in \mathbb{C}$

We "guess" an eigenvector: $\mathbf{v}_k \in \mathbb{C}^n$ with $\mathbf{v}_k := \left[\omega_n^{jk}\right]_{j=0}^{n-1} \in \mathbb{C}^n$, $k \in \{0,\ldots,n-1\}$.

$(u_{j-l}\omega_n^{lk})_{l\in\mathbb{Z}}$ is n-periodic!

$$\begin{aligned} (\mathbf{C}\mathbf{v}_k)_j &= \sum_{l=0}^{n-1} u_{j-l}\omega_n^{lk} = \sum_{l=j-n+1}^{j} u_{j-l}\omega_n^{lk} \\ &= \sum_{l=0}^{n-1} u_l \omega_n^{(j-l)k} = \omega_n^{jk}\boxed{\sum_{l=0}^{n-1} u_l \omega_n^{-lk}} = \lambda_k \cdot \omega_n^{jk} = \lambda_k \cdot (\mathbf{v}_k)_j\,. \end{aligned} \tag{4.2.10}$$

change of summation index — independent of j !

$\mathbf{v}_k$ is eigenvector of $\mathbf{C}$ for eigenvalue $\lambda_k = \sum_{l=0}^{n-1} u_l \omega_n^{-lk}$.

The set $\{\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}\} \subset \mathbb{C}^n$ provides the so-called *orthogonal* trigonometric basis of $\mathbb{C}^n$ = *eigenvector basis* for circulant matrices

$$\{\mathbf{v}_0, \ldots, \mathbf{v}_{n-1}\} = \left\{ \begin{bmatrix} \omega_n^0 \\ \vdots \\ \\ \\ \vdots \\ \omega_n^0 \end{bmatrix} \begin{bmatrix} \omega_n^0 \\ \omega_n^1 \\ \vdots \\ \\ \vdots \\ \omega_n^{n-1} \end{bmatrix} \cdots \begin{bmatrix} \omega_n^0 \\ \omega_n^{n-2} \\ \omega_n^{2(n-2)} \\ \vdots \\ \vdots \\ \omega_n^{(n-1)(n-2)} \end{bmatrix} \begin{bmatrix} \omega_n^0 \\ \omega_n^{n-1} \\ \omega_n^{2(n-1)} \\ \vdots \\ \vdots \\ \omega_n^{(n-1)^2} \end{bmatrix} \right\}. \tag{4.2.11}$$

From (4.2.8) we can conclude orthogonality of the basis vectors by straightforward computations:

$$\mathbf{v}_k := (\omega_n^{jk})_{j=0}^{n-1} \in \mathbb{C}^n: \quad \mathbf{v}_k^{\mathrm{H}} \mathbf{v}_m = \sum_{j=0}^{n-1} \omega_n^{-jk} \omega_n^{jm} = \sum_{j=0}^{n-1} \omega_n^{(m-k)j} \overset{(4.2.8)}{=} 0 \quad \text{, if } k \neq m \text{ .} \tag{4.2.12}$$

The matrix effecting the change of basis trigonometrical basis $\to$ standard basis is called the Fourier-matrix

$$\mathbf{F}_n = \begin{bmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} = \left[\omega_n^{lj} \right]_{l,j=0}^{n-1} \in \mathbb{C}^{n,n} \text{ .} \tag{4.2.13}$$

Lemma 4.2.14. Properties of Fourier matrix

The scaled Fourier-matrix $\frac{1}{\sqrt{n}}\mathbf{F}_n$ is unitary ($\to$ Def. 6.2.2) : $\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^{\mathrm{H}} = \frac{1}{n}\overline{\mathbf{F}}_n$.

Proof. The lemma is immediate from (4.2.12) and (4.2.8),because

$$\left(\mathbf{F}_n \mathbf{F}_n^{\mathrm{H}}\right)_{l,j} = \sum_{k=0}^{n-1} \omega_n^{(l-1)k} \overline{\omega_n}^{(j-1)k} = \sum_{k=0}^{n-1} \omega_n^{(l-1)k} \omega_n^{-(j-1)k} = \sum_{k=0}^{n-1} \omega_n^{k(l-j)} \text{ , } \quad 1 \le l, j \le n \text{ .}$$

□

Remark 4.2.15 (Spectrum of Fourier matrix)

We draw a conclusion from the properties stated in Lemma 4.2.14:

$$\tfrac{1}{n^2}\mathbf{F}_n^4 = I \quad \Rightarrow \quad \sigma(\tfrac{1}{\sqrt{n}}\mathbf{F}_n) \subset \{1, -1, i, -i\} \text{ ,}$$

because, if $\lambda \in \mathbb{C}$ is an eigenvalue of $\mathbf{F}_n$, then there is an eigenvector $\mathbf{x} \in \mathbb{C}^n \setminus \{0\}$ such that $\mathbf{F}_n \mathbf{x} = \lambda \mathbf{x}$, see Def. 9.1.1.

Lemma 4.2.16. Diagonalization of circulant matrices (→ Def. 4.1.38)

For any circulant matrix $\mathbf{C} \in \mathbb{K}^{n,n}$, $c_{ij} = u_{i-j}$, $(u_k)_{k\in\mathbb{Z}}$ n-periodic sequence, holds true

$$\mathbf{C}\overline{\mathbf{F}}_n = \overline{\mathbf{F}}_n \operatorname{diag}(d_1,\ldots,d_n) \quad , \quad \mathbf{d} = \mathbf{F}_n[u_0,\ldots,u_{n-1}]^\top .$$

Proof. Straightforward computation, see (4.2.10). □

Conclusion (from $\overline{\mathbf{F}}_n = n\mathbf{F}_n^{-1}$):

$$\boxed{\mathbf{C} = \mathbf{F}_n^{-1} \operatorname{diag}(d_1,\ldots,d_n)\mathbf{F}_n} \; . \tag{4.2.17}$$

Lemma 4.2.16, (4.2.17) ➢ multiplication with Fourier-matrix will be crucial operation in algorithms for circulant matrices and discrete convolutions.

Therefore this operation has been given a special name:

Definition 4.2.18. Discrete Fourier transform (DFT)

The linear map $\mathcal{F}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathcal{F}_n(\mathbf{y}) := \mathbf{F}_n\mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called discrete Fourier transform (DFT), i.e. for $\mathbf{c} := \mathcal{F}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \quad , \quad k = 0,\ldots,n-1 . \tag{4.2.19}$$

Recall the convention also adopted for the discussion of the DFT: vector indexes range from 0 to $n-1$!

Terminology: $\mathbf{c} = \mathbf{F}_n\mathbf{y}$ is also called the (discrete) Fourier transform of $\mathbf{y}$

From $\mathbf{F}_n^{-1} = \frac{1}{n}\overline{\mathbf{F}}_n$ (→ Lemma 4.2.14) we find the inverse discrete Fourier transform:

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \quad \Leftrightarrow \quad y_j = \frac{1}{n}\sum_{k=0}^{n-1} c_k \omega_n^{-kj} \tag{4.2.20}$$

(4.2.21) Discrete Fourier transform in EIGEN and MATLAB

- EIGEN-functions for discrete Fourier transform (and its inverse):

  DFT: `c=fft.fwd(y)` ↔ inverse DFT: `y=fft.inv(c);`

  Before using `fft`, remember to:

  1. `# include <unsupported/Eigen/FFT>`
  2. Instantiate helper class `Eigen::FFT<double> fft;`
     (The template argument should always be **double**.)

C++11 code 4.2.22: Demo: discrete Fourier transform in EIGEN ↪ GITLAB

```
int main() {
  using Comp = complex<double>;
  const VectorXcd::Index n = 5;
  VectorXcd y(n),c(n),x(n);
  y << Comp(1,0),Comp(2,1),Comp(3,2),Comp(4,3),Comp(5,4);
  FFT<double> fft; // DFT transform object
  c = fft.fwd(y);  // DTF of y, see Def. 4.2.18
  x = fft.inv(c);  // inverse DFT of c, see (4.2.20)

  cout << "y = " << y.transpose() << endl
       << "c = " << c.transpose() << endl
       << "x = " << x.transpose() << endl;
  return 0;
}
```

- MATLAB-functions for discrete Fourier transform (and its inverse):

DFT: `c=fft(y)` ↔ inverse DFT: `y=ifft(c);`

4.2.1 Discrete Convolution via DFT

Coding the formula from Def. 4.1.33 one would code discrete periodic convolution as follows:

C++11 code 4.2.23: Discrete periodic convolution: straightforward implementation ↪ GITLAB

```
VectorXcd pconv(const VectorXcd& u, const VectorXcd& x) {
  using idx_t = VectorXcd::Index; // may be unsigned !
  const idx_t n = x.size();
  VectorXcd z = VectorXcd::Zero(n);
  // Need signed indices when differences are formed
  for (long k = 0; k < n; ++k) {
    for (long j = 0; j < n; ++j) {
      long ind = (k - j < 0 ? n + k - j : k - j);
      z(k) += u(ind)*x(j);
    }}
  return z;
}
```

In § 4.1.37 we have seen that periodic convolution (→ Def. 4.1.33) amounts to multiplication with a circulant matrix. In addition, (4.2.17) reduces multiplication with a circulant matrix to two multiplications with the Fourier matrix $\mathbf{F}_n$ (= DFT) and (componentwise) scaling operations.

Summary:

$$\text{discrete periodic convolution} \quad z_k = \sum_{j=0}^{n-1} u_{k-j} x_j \quad (\rightarrow \text{Def. 4.1.33}), \quad k = 0, \ldots, n-1$$

$$\updownarrow$$

$$\text{multiplication with circulant matrix } (\rightarrow \text{Def. 4.1.38}) \quad \mathbf{z} = \mathbf{C}\mathbf{x}, \quad \mathbf{C} := \left[u_{i-j}\right]_{i,j=1}^{n}.$$

Idea: (4.2.17) $\succ$ $\mathbf{z} = \mathbf{F}_n^{-1}\operatorname{diag}(\mathbf{F}_n\mathbf{u})\mathbf{F}_n\mathbf{x}$

This formula is usually referred to as *convolution theorem*:

Theorem 4.2.24. Convolution theorem

The discrete periodic convolution $*_n$ between n-dimensional vectors $\mathbf{u}$ and $\mathbf{x}$ is equal to the inverse DFT of the component-wise product between the DFTs of $\mathbf{u}$ and $\mathbf{u}$; i.e.:

$$(\mathbf{u}) *_n (\mathbf{x}) := \sum_{j=0}^{n-1} u_{k-j}x_j = \mathbf{F}_n^{-1}\left[(\mathbf{F}_n\mathbf{u})_j(\mathbf{F}_n\mathbf{x})_j\right]_{j=1}^{n}.$$

C++11 code 4.2.25: Discrete periodic convolution: DFT implementation ↪ GITLAB

```
VectorXcd pconvfft(const VectorXcd& u, const VectorXcd& x) {
  Eigen::FFT<double> fft;
  VectorXcd tmp = ( fft.fwd(u) ).cwiseProduct( fft.fwd(x) );
  return fft.inv(tmp);
}
```

In Rem. 4.1.40 we learned that the discrete convolution of n-vectors ($\rightarrow$ Def. 4.1.22) can be accomplished by the *periodic* discrete convolution of $2n-1$-vectors (obtained by zero padding, see Rem. 4.1.40):

C++11 code 4.2.26: Implementation of discrete convolution ($\rightarrow$ Def. 4.1.22) based on periodic discrete convolution ↪ GITLAB

```
VectorXcd myconv(const VectorXcd& h, const VectorXcd& x) {
  const long n = h.size();
  // Zero padding, cf. (4.1.41)
  VectorXcd hp(2*n - 1), xp(2*n - 1);
  hp << h, VectorXcd::Zero(n - 1);
  xp << x, VectorXcd::Zero(n - 1);
  // Periodic discrete convolution of length 2n-1, ??
  return pconvfft(hp, xp);
}
```

4.2.2 Frequency filtering via DFT

The trigonometric basis vectors, when interpreted as time-periodic signals, represent harmonic oscillations. This is illustrated when plotting some vectors of the trigonometric basis ($n = 16$):

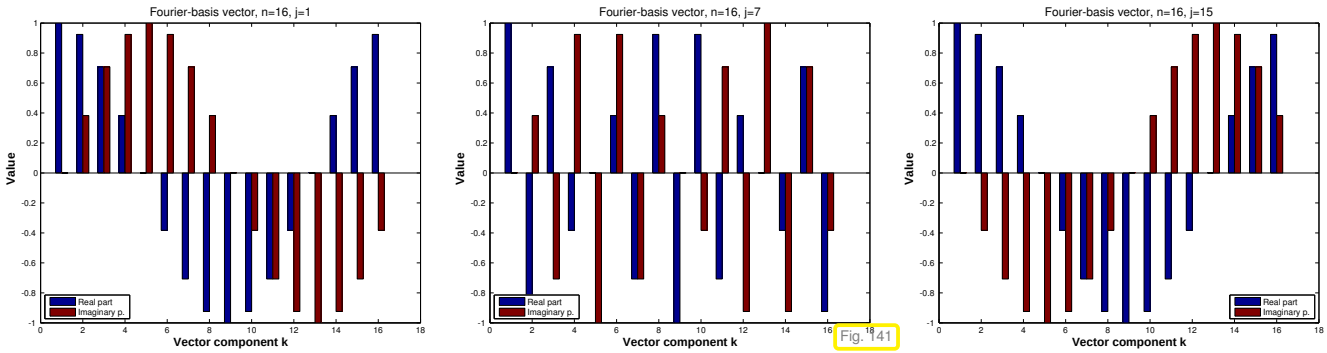


Fig. 141

"slow oscillation/low frequency" "fast oscillation/high frequency" "slow oscillation/low frequency"

Dominant coefficients of a signal after transformation to trigonometric basis indicate dominant frequency components.

Terminology: coefficients of a signal w.r.t. trigonometric basis = signal in frequency domain, original signal = time domain.

Recall: DFT (4.2.19) and inverse DFT (4.2.20)

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} \quad \Leftrightarrow \quad y_j = \frac{1}{n} \sum_{k=0}^{n-1} c_k \omega_n^{-kj} \tag{4.2.20}$$

Consider $y_k \in \mathbb{R}$ $\Rightarrow$ $c_k = \overline{c}_{n-k}$, because $\omega_n^{kj} = \overline{\omega}_n^{(n-k)j}$, and $n = 2m+1$

$$\begin{aligned} ny_j &= c_0 + \sum_{k=1}^{m} c_k \omega_n^{-kj} + \sum_{k=m+1}^{2m} c_k \omega_n^{-kj} = c_0 + \sum_{k=1}^{m} c_k \omega_n^{-kj} + c_{n-k} \omega_n^{(k-n)j} \\ &= c_0 + 2 \sum_{k=1}^{m} \mathrm{Re}(c_k) \cos(2\pi kj/n) + \mathrm{Im}(c_k) \sin(2\pi kj/n) \ , \end{aligned}$$

since $\omega_n^{\ell} = \cos(2\pi \ell/n) + i \sin(2\pi \ell/n)$.

➢ $|c_k|, |c_{n-k}|$ measures the strength with which an oscillation with frequency k is represented in the signal, $0 \le k \le \lfloor \frac{n}{2} \rfloor$.

Example 4.2.27 (Frequency identification with DFT)

Extraction of characteristic frequencies from a distorted discrete periodical signal, generated by the following C++ code:

C++11 code 4.2.28: Generation of noisy sinusoidal signal ↪ GITLAB

```
2 VectorXd signalgen() {
3   const int N = 64;
4   const ArrayXd t = ArrayXd::LinSpaced(N, 0, N);
5   const VectorXd x = ((2*M_PI/N*t).sin() +
      (14*M_PI/N*t).sin()).matrix();
6   return x + VectorXd::Random(N);
7 }
```

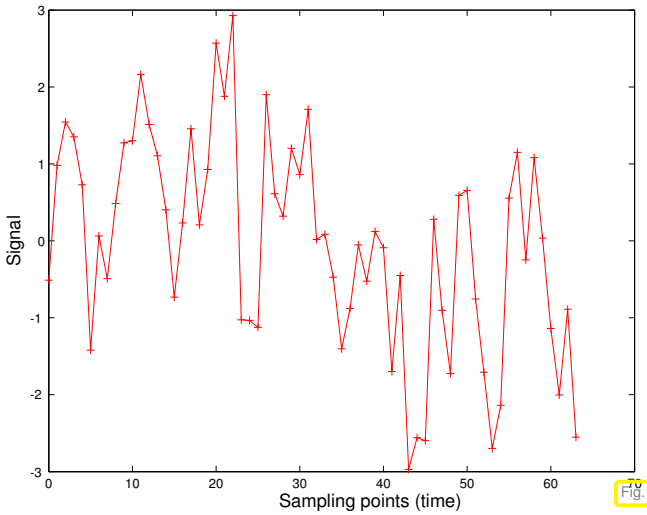


Fig. 142

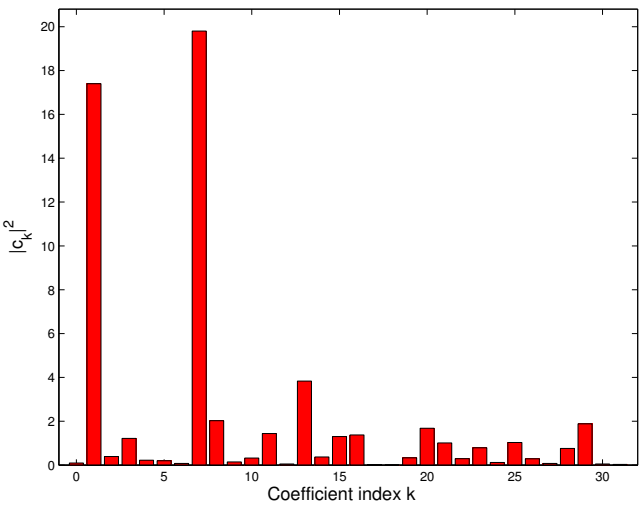


Fig. 143

Fig. 143 was generated by the following MATLAB code. A corresponding C++ code is also available ➙ GITLAB.

MATLAB code 4.2.29: Frequency extraction → Fig. 143

```
c = fft(y); p = (c.*conj(c))/64;

figure('Name','power spectrum');
bar(0:31,p(1:32),'r');
set(gca,'fontsize',14);
axis([-1 32 0 max(p)+1]);
xlabel('{\bf index k of Fourier coefficient}','Fontsize',14);
ylabel('{\bf |c_k|^2}','Fontsize',14);
```

We observe that frequencies present in unperturbed signal become evident in frequency domain, whereas it is hard to tell them from the time-domain signal.

Example 4.2.30 (Detecting periodicity in data)

DFT: a computer's eye for periodic patterns in data

The following C++ code processes actual web search data and performs a frequency analysis using DFT:

C++11 code 4.2.31: Extraction of periodic patterns by DFT ➙ GITLAB

```
  VectorXd x = read("trend.dat");
  const int n = x.size();

  mgl::Figure trend;
  trend.plot(x, "r");
  trend.title("Google: 'Vorlesungsverzeichnis'");
  trend.grid();
  trend.xlabel("week (1.1.2004-31.12.2010)");
  trend.ylabel("relative no. of searches");
  trend.save("searchdata");
```

```
12
13  Eigen::FFT<double> fft;
14  VectorXcd c = fft.fwd(x);
15  VectorXd p = c.cwiseProduct(c.conjugate()).real()
16               .segment(2, std::floor((n+1.)/2));
17
18  // plot power spectrum
19  mgl::Figure fig; fig.plot(p,"m"); fig.grid();
20  fig.title("Fourier spectrum");
21  fig.xlabel("index j of Fourier component");
22  fig.ylabel("|c_j|^2");
23
24  // mark 4 highest peaks with red star
25  VectorXd p_sorted = p;
26  // sort descending
27  std::sort(p_sorted.data(), p_sorted.data() + p.size(),
        std::greater<double>());
28  for (int i = 0; i < 4; ++i) {
29    // get position of p_sorted[i]
30    int idx = std::find(p.data(), p.data()+p.size(), p_sorted[i]) -
          p.data();
31    std::vector<double> ind = { double(idx + 1) }; // save in vectors
32    std::vector<double> val = { p_sorted[i] }; // to be able to plot
33    fig.plot(ind, val, " r*");
34  }
35  fig.save("fourierdata");
```

Google: 'Vorlesungsverzeichnis'

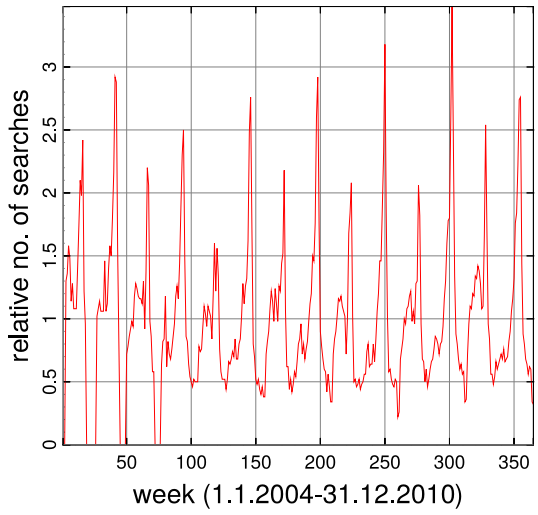


Fig. 144

Fourier spectrum

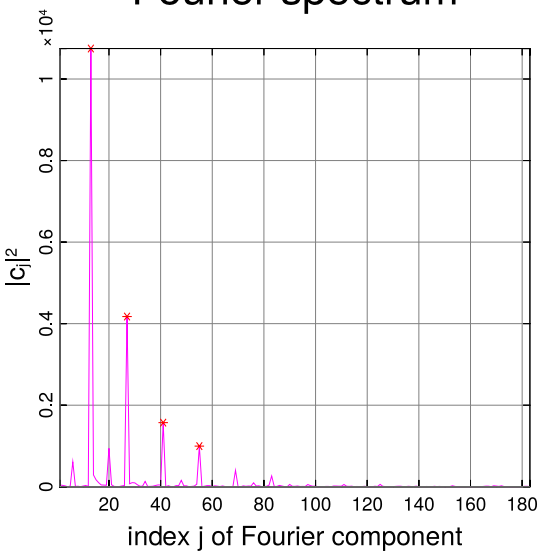
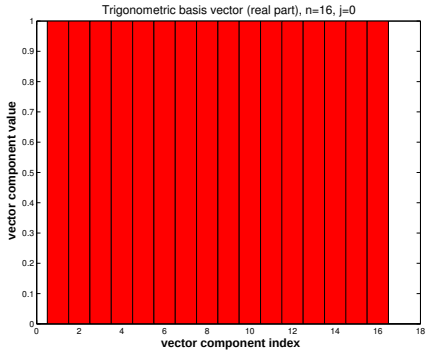


Fig. 145

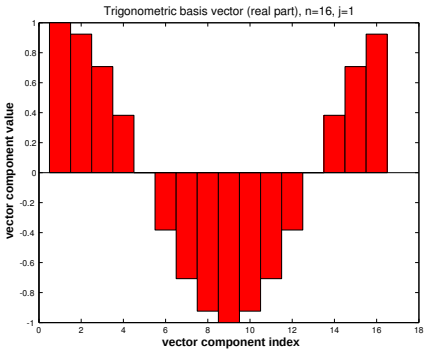
Pronounced peaks in the power spectrum point to periodic structure of the data. Location of peaks tells lengths of dominant periods.

Remark 4.2.32 ("Low" and "high" frequencies)

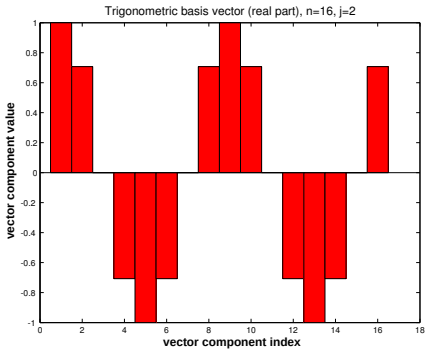
Plots of real parts of trigonometric basis vectors $(\mathbf{F}_n)_{:,j}$ (= columns of Fourier matrix), $n = 16$.



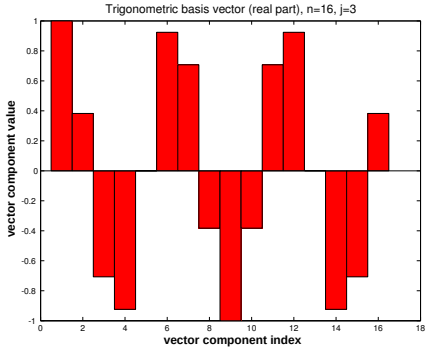
$\mathrm{Re}(\mathbf{F}_{16})_{:,0}$



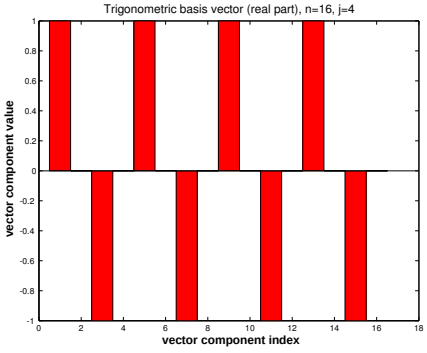
$\mathrm{Re}(\mathbf{F}_{16})_{:,1}$



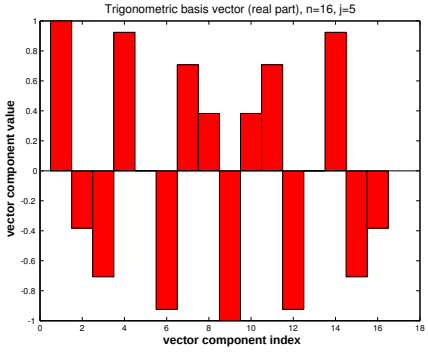
$\mathrm{Re}(\mathbf{F}_{16})_{:,2}$



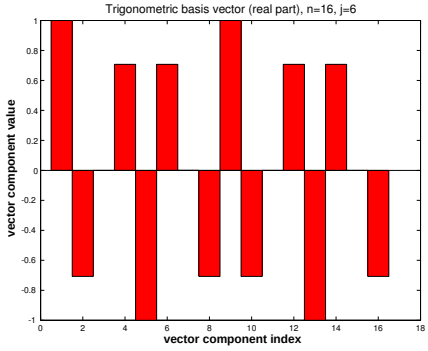
$\mathrm{Re}(\mathbf{F}_{16})_{:,3}$



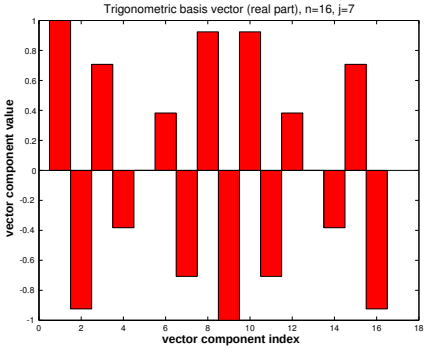
$\mathrm{Re}(\mathbf{F}_{16})_{:,4}$



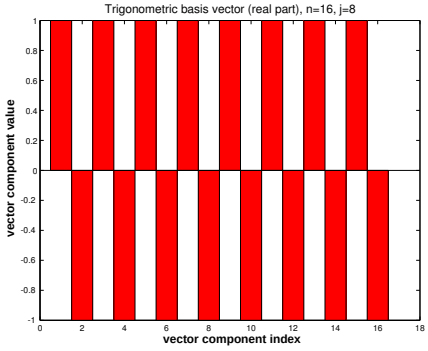
$\mathrm{Re}(\mathbf{F}_{16})_{:,5}$



$\mathrm{Re}(\mathbf{F}_{16})_{:,6}$



$\mathrm{Re}(\mathbf{F}_{16})_{:,7}$



$\mathrm{Re}(\mathbf{F}_{16})_{:,8}$

By elementary trigonometric identities:

$$\begin{aligned}\mathrm{Re}(\mathbf{F}_n)_{:,j} &= \left(\mathrm{Re}\,\omega_n^{(j-1)k}\right)_{k=0}^{n-1} \\ &= \left(\mathrm{Re}\exp(2\pi i(j-1)k/n)\right)_{k=0}^{n-1} \\ &= \left(\cos(2\pi(j-1)x)\right)_{x=0,\frac{1}{n},\ldots,1-\frac{1}{n}} .\end{aligned}$$

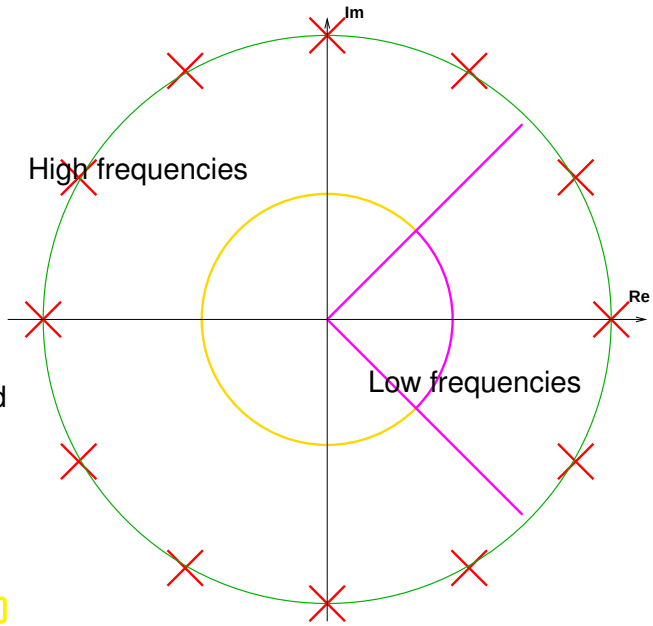


Fig. 146

Slow oscillations/low frequencies $\leftrightarrow$ $j \approx 1$ and $j \approx n$.

Fast oscillations/high frequencies $\leftrightarrow$ $j \approx n/2$.

Frequency filtering of real discrete periodic signals by suppressing certain "Fourier coefficients".

C++11-code 4.2.33: DFT-based frequency filtering ↪ GITLAB

```
void freqfilter(const VectorXd& y, int k,
                VectorXd& low, VectorXd& high) {
  const VectorXd::Index n = y.size();
  if (n%2 != 0)
    throw std::runtime_error("Even vector length required!");
  const VectorXd::Index m = y.size()/2;

  Eigen::FFT<double> fft; // DFT helper object
  VectorXcd c = fft.fwd(y); // Perform DFT of input vector

  VectorXcd clow = c;
  // Set high frequency coefficients to zero, Fig. 146
  for (int j = -k; j <= +k; ++j) clow(m+j) = 0;
  // (Complementary) vector of high frequency coefficients
  VectorXcd chigh = c - clow;

  // Recover filtered time-domain signals
  low = fft.inv(clow).real();
  high = fft.inv(chigh).real();
}
```

(It can be optimised exploiting $y_j \in \mathbb{R}$ and $c_{n/2-k} = \overline{c}_{n/2+k}$)

Map `y` $\mapsto$ `low` (in Code 4.2.33) $\widehat{=}$ low pass filter .

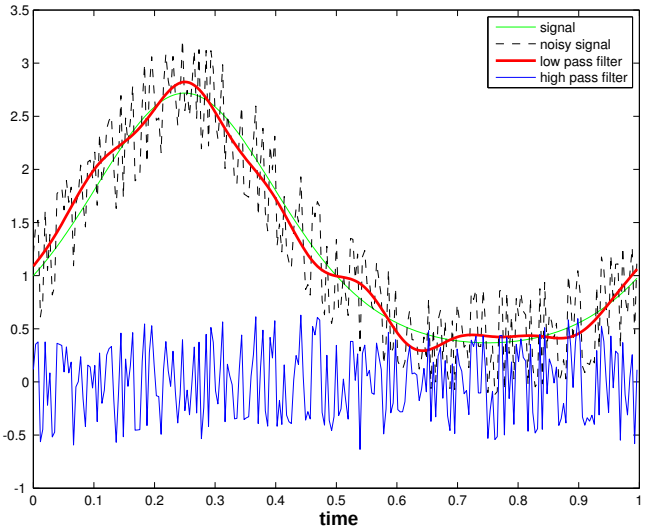
Map `y` $\mapsto$ `high` (in Code 4.2.33) $\widehat{=}$ high pass filter .

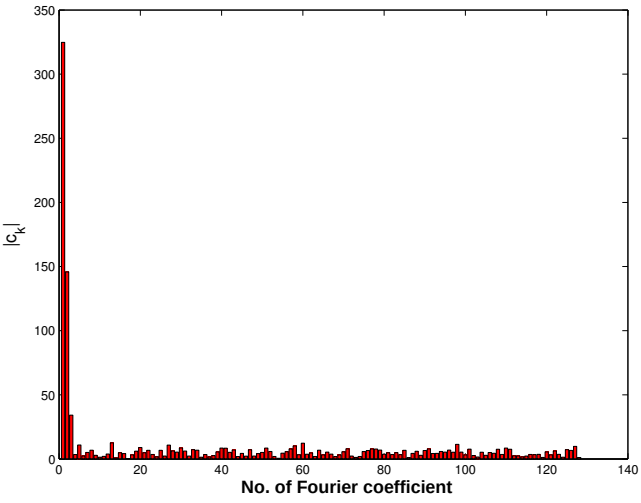
Example 4.2.34 (Frequency filtering by DFT)

Noisy signal:

```
n = 256; y = exp(sin(2*pi*((0:n-1)')/n)) + 0.5*sin(exp(1:n)');
```

Frequency filtering by Code 4.2.33 with $k = 120$.





Low pass filtering can be used for *denoising*, that is, the removal of high frequency perturbations of a signal.

Example 4.2.35 (Sound filtering by DFT)

Frequency filtering is ubiquitous in sound processing. Here we demonstrate it in MATLAB, which offers tools for audio processing.

MATLAB-code 4.2.36: DFT based sound compression

```
% Low pass sound filtering by DFT

% Read sound data
[y,Fs,nbits] = wavread('hello.wav');
sound(y,Fs);

n = length(y);
fprintf('Read wav File: %d samples, rate = %d/s, nbits = %d\n',
   n,Fs,nbits);
k = 1; s{k} = y; leg{k} = 'Sampled signal';

c = fft(y);

figure('name','sound signal');
plot((22000:44000)/Fs,s{1}(22000:44000),'r-');
title('samples sound signal','fontsize',14);
xlabel('{\bf time[s]}','fontsize',14);
ylabel('{\bf sound pressure}','fontsize',14);
grid on;

% print -depsc2 '../PICTURES/soundsignal.eps';

figure('name','sound frequencies');
plot(1:n,abs(c).^2,'m-');
title('power spectrum of sound signal','fontsize',14);
xlabel('{\bf index k of Fourier coefficient}','fontsize',14);
ylabel('{\bf |c_k|^2}','fontsize',14);
grid on;

% print -depsc2 '../PICTURES/soundpower.eps';

figure('name','sound frequencies');
plot(1:3000,abs(c(1:3000)).^2,'b-');
title('low frequency power spectrum','fontsize',14);
xlabel('{\bf index k of Fourier coefficient}','fontsize',14);
ylabel('{\bf |c_k|^2}','fontsize',14);
grid on;

% print -depsc2 '../PICTURES/soundlowpower.eps';
```

```

for m=[1000,3000,5000]

  % Low pass filtering
  cf = zeros(1,n);
  cf(1:m) = c(1:m); cf(n-m+1:end) = c(n-m+1:end);

  % Reconstruct filtered signal
  yf = ifft(cf);
  % No idea why this is necessary
  wavwrite(yf,Fs,nbits,sprintf('hellof%d.wav',m));
  cy = wavread(sprintf('hellof%d.wav',m));
  sound(cy,Fs,nbits);

  k = k+1;
  s{k} = real(yf);
  leg{k} = sprintf('cutt-off = %d',m');
end

% Plot original signal and filtered signals
figure('name','sound filtering');
plot((30000:32000)/Fs,s{1}(30000:32000),'r-',...
     (30000:32000)/Fs,s{2}(30000:32000),'b--',...
     (30000:32000)/Fs,s{3}(30000:32000),'m--',...
     (30000:32000)/Fs,s{2}(30000:32000),'k--');
xlabel('{\bf time[s]}','fontsize',14);
ylabel('{\bf sound pressure}','fontsize',14);
legend(leg,'location','southeast');

% print -depsc2 '../PICTURES/soundfiltered.eps';
```

MATLAB-code 4.2.37: DFT based low pass frequency filtering of sound

```
        [y,sf,nb] = wavread('hello.wav');
        c = fft(y); c(m+1:end-m) = 0;
        wavwrite(ifft(c),sf,nb,'filtered.wav');
```

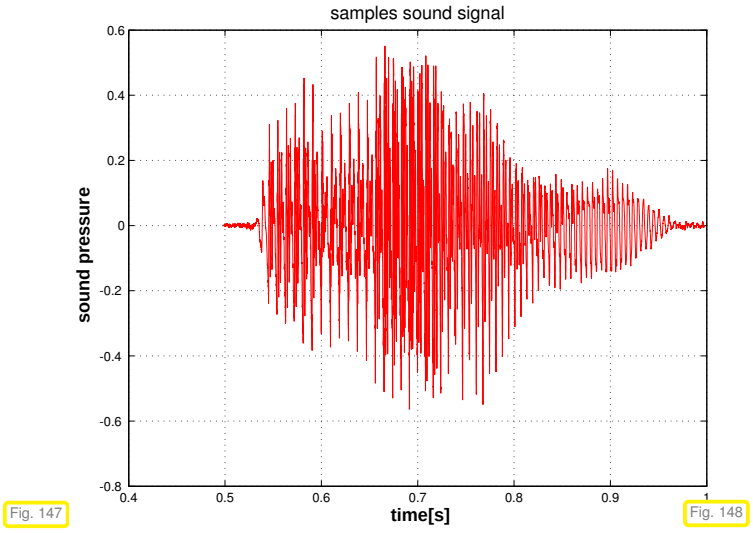


Fig. 147

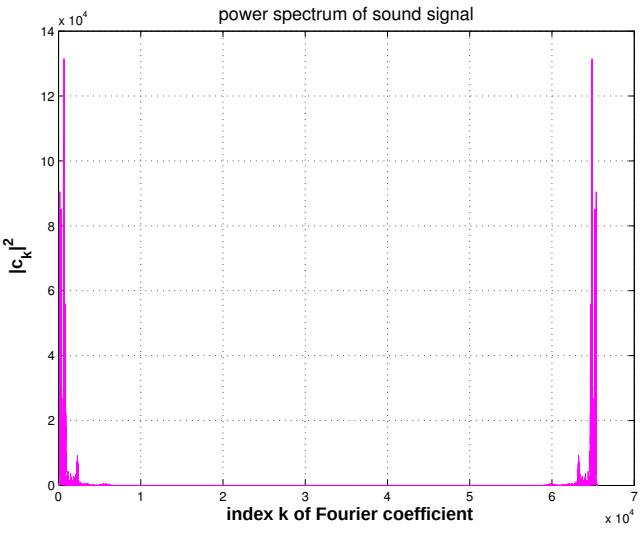


Fig. 148

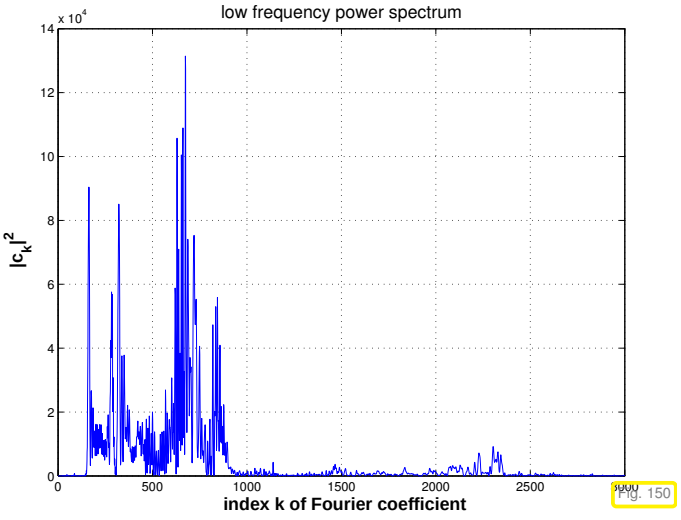


Fig. 149

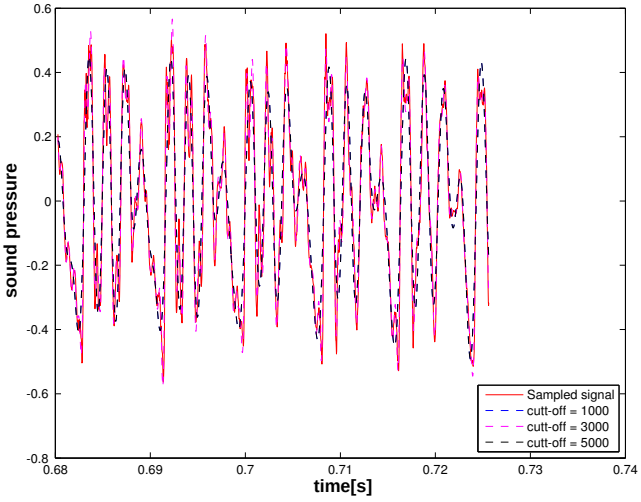


Fig. 150

The power spectrum of a signal $\mathbf{y} \in \mathbb{C}^n$ is the vector $\left(|c_j|^2\right)_{j=0}^{n-1}$, where $\mathbf{c} = \mathbf{F}_n\mathbf{y}$ is the discrete Fourier transform of $\mathbf{y}$.

4.2.3 Real DFT

Every time-discrete signal obtained from sampling a time-dependent physical quantity will yield a real vector. Of couse, a real vector contains only half the information compared to complex vector of the same length. We aim to exploit this for a more efficient implementation of DFT.

Task: Efficient implementation of DFT (Def. 4.2.18) $(c_0,\ldots,c_{n-1})$ for *real* coefficients $(y_0,\ldots,y_{n-1})^\top \in \mathbb{R}^n$, $n = 2m$, $m \in \mathbb{N}$.

If $y_j \in \mathbb{R}$ in DFT formula (4.2.19), we obtain redundant output

$$\omega_n^{(n-k)j} = \overline{\omega}_n^{kj}\,, \quad k = 0,\ldots,n-1\,,$$

$$\Rightarrow \quad \overline{c}_k = \sum_{j=0}^{n-1} y_j \overline{\omega}_n^{kj} = \sum_{j=0}^{n-1} y_j \omega_n^{(n-k)j} = c_{n-k}\,, \quad k = 1,\ldots,n-1\,.$$

Idea: map $\mathbf{y} \in \mathbb{R}^n$, to $\mathbb{C}^m$ and use DFT of length m.

$$h_k = \sum_{j=0}^{m-1} (y_{2j} + iy_{2j+1})\,\omega_m^{jk} = \boxed{\textstyle\sum_{j=0}^{m-1} y_{2j}\,\omega_m^{jk}} + i\cdot \boxed{\textstyle\sum_{j=0}^{m-1} y_{2j+1}\,\omega_m^{jk}}\,, \tag{4.2.38}$$

$$\overline{h}_{m-k} = \sum_{j=0}^{m-1} \overline{y_{2j} + iy_{2j+1}}\,\overline{\omega}_m^{j(m-k)} = \boxed{\sum_{j=0}^{m-1} y_{2j}\,\omega_m^{jk}} - i \cdot \boxed{\sum_{j=0}^{m-1} y_{2j+1}\,\omega_m^{jk}} . \tag{4.2.39}$$

$$\Rightarrow \quad \boxed{\sum_{j=0}^{m-1} y_{2j}\,\omega_m^{jk}} = \tfrac{1}{2}(h_k + \overline{h}_{m-k}) \quad , \quad \boxed{\sum_{j=0}^{m-1} y_{2j+1}\,\omega_m^{jk}} = -\tfrac{1}{2}i(h_k - \overline{h}_{m-k}) .$$

Use simple identities for roots of unity:

$$c_k = \sum_{j=0}^{n-1} y_j\,\omega_n^{jk} = \boxed{\sum_{j=0}^{m-1} y_{2j}\,\omega_m^{jk}} + \omega_n^k \cdot \boxed{\sum_{j=0}^{m-1} y_{2j+1}\,\omega_m^{jk}} . \tag{4.2.40}$$

$$\Rightarrow \quad \begin{cases} c_k = \frac{1}{2}(h_k + \overline{h}_{m-k}) - \frac{1}{2}i\omega_n^k(h_k - \overline{h}_{m-k}) , & k = 0,\ldots,m-1 , \\ c_m = \mathrm{Re}\{h_0\} - \mathrm{Im}\{h_0\} , & \\ c_k = \overline{c}_{n-k} , & k = m+1,\ldots,n-1 . \end{cases} \tag{4.2.41}$$

C++11 code 4.2.42: DFT of real vectors of length $n/2$ ↪ GITLAB

```
// Perform fft on a real vector y of even
// length and return (complex) coefficients in c
// Note: EIGEN's DFT method fwd() has this already implemented and
// we could also just call: c = fft.fwd(y);
void fftreal(const VectorXd& y, VectorXcd& c) {
  const unsigned n = y.size(), m = n/2;
  if (n % 2 != 0) { std::cout << "n must be even!\n"; return; }

  // Step I: compute h from (4.2.38), (4.2.39)
  std::complex<double> i(0,1); // Imaginary unit
  VectorXcd yc(m);
  for (unsigned j = 0; j < m; ++j)
    yc(j) = y(2*j) + i*y(2*j + 1);

  Eigen::FFT<double> fft;
  VectorXcd d = fft.fwd(yc), h(m + 1);
  h << d, d(0);

  c.resize(n);
  // Step II: implementation of (4.2.41)
  for (unsigned k = 0; k < m; ++k) {
    c(k) = (h(k) + std::conj(h(m-k)))/2. -
        i/2.*std::exp(-2.*k/n*M_PI*i)*(h(k) - std::conj(h(m-k)));
  }
  c(m) = std::real(h(0)) - std::imag(h(0));
  for (unsigned k = m+1; k < n; ++k) c(k) = std::conj(c(n-k));
}
```

4.2.4 Two-dimensional DFT

In this we study the frequency decomposition of matrices. Due to the natural analofy

one-dimensional data ("signal") $\longleftrightarrow$ vector $\mathbf{y} \in \mathbb{C}^n$,

two-dimensional data ("image") $\longleftrightarrow$ matrix. $\mathbf{Y} \in \mathbb{C}^{m,n}$,

these techniques are of fundamental importance for image processing.

(4.2.43) Matrix Fourier modes

A two-dimensional trigonometric basis of $\mathbb{C}^{m,n}$ is give by the tensor product matrices

$$\left\{ (\mathbf{F}_m)_{:,j} (\mathbf{F}_n)_{:,\ell}^\top, 1 \le j \le m, 1 \le \ell \le n \right\} \subset \mathbb{C}^{m,n} . \tag{4.2.44}$$

Let a matrix $\mathbf{C} \in \mathbb{C}^{m,n}$ be given as a linear combination of these basis matrices with coefficients $y_{j_1,j_2} \in \mathbb{C}$, $0 \le j_1 < m$, $0 \le j_2 < n$:

$$\mathbf{C} = \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} y_{j_1,j_2} (\mathbf{F}_m)_{:,j_1} (\mathbf{F}_n)_{:,j_2}^\top . \tag{4.2.45}$$

Then the entries of $\mathbf{C}$ can be computed by two nested discrete Fourier transforms:

$$(\mathbf{C})_{k_1,k_2} = \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} y_{j_1,j_2} \omega_m^{j_1 k_1} \omega_n^{j_2 k_2} = \sum_{j_1=0}^{m-1} \omega_m^{j_1 k_1} \left(\sum_{j_2=0}^{n-1} \omega_n^{j_2 k_2} y_{j_1,j_2} \right) , \quad 0 \le k_1 < m , 0 \le k_2 < n .$$

The coefficients can also be regarded as entries of a matrix $\mathbf{Y} \in \mathbb{C}^{m,n}$. Thus we can rewrite the above expressions: for all $0 \le k_1 < m$, $0 \le k_2 < n$

$$(\mathbf{C})_{k_1,k_2} = \sum_{j_1=0}^{m-1} \left(\mathbf{F}_n (\mathbf{Y})_{j_1,:} \right)_{k_2} \omega_m^{j_1 k_1} \quad \blacktriangleright \quad \boxed{\mathbf{C} = \mathbf{F}_m (\mathbf{F}_n \mathbf{Y}^\top)^\top = \mathbf{F}_m \mathbf{Y} \mathbf{F}_n} . \tag{4.2.46}$$

This formula defines the two-dimensional discrete Fourier transform of the matrix $\mathbf{Y} \in \mathbb{C}^{m,n}$. By Lemma 4.2.14 we immediately get the inversion formula:

$$\mathbf{C} = \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} y_{j_1,j_2} (\mathbf{F}_m)_{:,j_1} (\mathbf{F}_n)_{:,j_2}^\top \quad \Rightarrow \quad \boxed{\mathbf{Y} = \mathbf{F}_m^{-1} \mathbf{C} \mathbf{F}_n^{-1} = \frac{1}{mn} \overline{\mathbf{F}}_m \mathbf{C} \overline{\mathbf{F}}_n} . \tag{4.2.47}$$

C++11 code 4.2.48: Two-dimensional discrete Fourier transform ⇝ GITLAB

```
2 template <typename Scalar>
3 void fft2(Eigen::MatrixXcd &C, const Eigen::MatrixBase<Scalar> &Y) {
4   using idx_t = Eigen::MatrixXcd::Index;
5   const idx_t m = Y.rows(),n=Y.cols();
6   C.resize(m,n);
7   Eigen::MatrixXcd tmp(m,n);
8
9   Eigen::FFT<double> fft; // Helper class for DFT
```

```
10    // Transform rows of matrix Y
11    for (idx_t k=0;k<m;k++) {
12      Eigen::VectorXcd tv(Y.row(k));
13      tmp.row(k) = fft.fwd(tv).transpose();
14    }
15
16    // Transform columns of temporary matrix
17    for (idx_t k=0;k<n;k++) {
18      Eigen::VectorXcd tv(tmp.col(k));
19      C.col(k) = fft.fwd(tv);
20    }
21  }
```

C++11 code 4.2.49: Inverse two-dimensional discrete Fourier transform ↪ GITLAB

```
2  template <typename Scalar>
3  void ifft2(Eigen::MatrixXcd &C, const Eigen::MatrixBase<Scalar> &Y) {
4    using idx_t = Eigen::MatrixXcd::Index;
5    const idx_t m = Y.rows(),n=Y.cols();
6    fft2(C,Y.conjugate()); C = C.conjugate()/(m*n);
7  }
```

Remark 4.2.50 (Two-dimensional DFT in MATLAB)

The two-dimensional DFT is provided by the MATLAB function: fft2(Y) .

In MATLAB the two-dimensional DFT is defined through *nested one-dimensional DFTs* (4.2.19) as follows:

```
fft2(Y) = fft(fft(Y).').'
```

Here: .' simply transposes the matrix (no complex conjugation)

(4.2.51) Periodic convolution of matrices

We consider the following bilinear mapping $\mathsf{B} : \mathbb{C}^{m,n} \times \mathbb{C}^{m,n} \to \mathbb{C}^{m,n}$:

$$(\mathsf{B}(\mathbf{X},\mathbf{Y}))_{k,\ell} = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} (\mathbf{X})_{i,j}(\mathbf{Y})_{k-i \mod m,\ell-j \mod n}\,, \tag{4.2.52}$$

which defines the two-dimensional discrete periodic convolution, *cf.* Def. 4.1.33.

C++11 code 4.2.53: Straightforward implementation of 2D discrete periodic convolution ↪ GITLAB

```
2  // Straightforward implementation of 2D periodic convolution
3  template <typename Scalar1,typename Scalar2,class EigenMatrix>
4  void pmconv_basic(const Eigen::MatrixBase<Scalar1> &X,const
     Eigen::MatrixBase<Scalar2> &Y,
```

```
5              EigenMatrix &Z) {
6   using idx_t = typename EigenMatrix::Index;
7   using val_t = typename EigenMatrix::Scalar;
8   const idx_t n=X.cols(),m=X.rows();
9   if ((m!=Y.rows()) || (n!=Y.cols())) throw
       std::runtime_error("pmconv: size mismatch");
10  Z.resize(m,n);
11  // Implementation of (4.2.52)
12  auto idxwrap = [] (const idx_t L,int i)
13    { if (i>=L) i -= L; else if (i<0) i += L; return i; };
14  for (int i=0;i<m;i++) for (int j=0;j<n;j++) {
15      val_t s=0;
16      for (int k=0;k<m;k++) for (int l=0;l<n;l++)
17       s += X(k,l)*Y(idxwrap(m,i-k),idxwrap(n,j-l));
18      Z(i,j) = s;
19    }
20 }
```

The 2D discrete periodic convolution admits a diagonalization by switching to the trigonometric basis of $\mathbb{C}^{m,n}$, analogous to (4.2.17), see Section 4.2.1.

In (4.2.52) set $\mathbf{Y} = (\mathbf{F}_m)_{:,r}(\mathbf{F}_n)_{s,:} \in \mathbb{C}^{m,n} \quad \leftrightarrow \quad (\mathbf{Y})_{i,j} = \omega_m^{ri}\omega_n^{sj}$, $0 \le i < m$, $0 \le j < n$:

$$\begin{aligned}(\mathsf{B}(\mathbf{X},\mathbf{Y}))_{k,\ell} &= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}(\mathbf{Y})_{k-i \bmod m,\ell-j \bmod n}\\ &= \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}\omega_m^{r(k-i)}\omega_n^{s(\ell-j)}\\ &= \left(\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}\overline{\omega_m^{ri}\omega_n^{sj}}\right)\cdot\omega_m^{rk}\omega_n^{s\ell}\,.\end{aligned}$$

$$\blacktriangleright \quad \mathsf{B}(\mathbf{X},\underbrace{(\mathbf{F}_m)_{:,r}(\mathbf{F}_n)_{s,:}}_{\text{"eigenvector"}}) = \underbrace{\left(\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(\mathbf{X})_{i,j}\overline{\omega_m^{ri}\omega_n^{sj}}\right)}_{\text{see Eq. (4.2.45)}}(\mathbf{F}_m)_{:,r}(\mathbf{F}_n)_{s,:}\,. \tag{4.2.54}$$

Hence, the (complex conjugated) two-dimensional discrete Fourier transform of $\overline{\mathbf{X}}$ according to (4.2.45) provides the eigenvalues of the aniti-linear mapping $\mathbf{Y} \mapsto \mathsf{B}(\mathbf{X},\mathbf{Y})$, $\mathbf{X} \in \mathbb{C}^{m,n}$ fixed.

This suggests the following DFT-based algorithm for evaluating the periodic convolution of matrices:

❶ Compute $\hat{\mathbf{Y}}$ by inverse 2D DFT of $\overline{\mathbf{Y}}$, see Code 4.2.49

❷ Compute $\overline{\hat{\mathbf{X}}}$ by 2D DFT of $\overline{\mathbf{X}}$, see Code 4.2.48.

❸ Component-wise multiplication of $\hat{\mathbf{X}}$ and $\hat{\mathbf{Y}}$: $\hat{\mathbf{Z}} = \hat{\mathbf{X}}.*\hat{\mathbf{Y}}$.

❹ Compute $\mathbf{Z}$ through inverse 2D DTF of $\hat{\mathbf{Z}}$.

C++11 code 4.2.55: DFT-based 2D discrete periodic convolution ⇢ GITLAB

```
2 // DFT based implementation of 2D periodic convolution
3 template <typename Scalar1,typename Scalar2,class EigenMatrix>
```

```
4  void pmconv(const Eigen::MatrixBase<Scalar1> &X,const
       Eigen::MatrixBase<Scalar2> &Y,
5            EigenMatrix &Z) {
6    using Comp = std::complex<double>;
7    using idx_t = typename EigenMatrix::Index;
8    using val_t = typename EigenMatrix::Scalar;
9    const idx_t n=X.cols(),m=X.rows();
10   if ((m!=Y.rows()) || (n!=Y.cols())) throw
         std::runtime_error("pmconv: size mismatch");
11   Z.resize(m,n); Eigen::MatrixXcd Xh(m,n),Yh(m,n);
12   // Step ❶: 2D DFT of Y
13   fft2(Yh,(Y.template cast<Comp>()));
14   // Step ❷: 2D DFT of X
15   fft2(Xh,(X.template cast<Comp>()));
16   // Steps ❸, ❹: inverse DFT of component-wise product
17   ifft2(Z,Xh.cwiseProduct(Yh));
18 }
```

Example 4.2.56 (Deblurring by DFT)

2D discrete convolutions are important for image processing. Let a Gray-scale pixel image be stored in the matrix $\mathbf{P} \in \mathbb{R}^{m,n}$, actually $\mathbf{P} \in \{0,\ldots,255\}^{m,n}$, see also Ex. 9.3.24.

Write $(p_{l,k})_{l,k\in\mathbb{Z}}$ for the periodically extended image:

$$p_{l,j} = (\mathbf{P})_{l+1,j+1} \quad \text{for} \quad 1 \le l \le m, 1 \le j \le n\,, \quad p_{l,j} = p_{l+m,j+n} \quad \forall l,k \in \mathbb{Z}\,.$$

Blurring = pixel values get replaced by weighted averages of near-by pixel values
(effect of distortion in optical transmission systems)

$$c_{l,j} = \sum_{k=-L}^{L}\sum_{q=-L}^{L} s_{k,q}\, p_{l+k,j+q}\,, \quad \begin{matrix} 0 \le l < m\,, \\ 0 \le j < n\,, \end{matrix} \quad L \in \{1,\ldots,\min\{m,n\}\}\,. \tag{4.2.57}$$

$c_{l,j}$: blurred image; $s_{k,q}$: point spread function

Does this ring a bell? Hidden in (4.2.57) is a 2D discrete periodic convolution, see Eq. (4.2.52). In light of the algorithm implemented in Code 4.2.55 it is hardly surprising that DFT comes handy for reversing the effect of the blurring!

Note that usually: L small, $s_{k,m} \ge 0$, $\sum_{k=-L}^{L}\sum_{q=-L}^{L} s_{k,q} = 1$ (an averaging)

In the experiments we used: $L = 5$ and the PSF $s_{k,q} = \dfrac{1}{1+k^2+q^2}$.

C++11-code 4.2.58: Point spread function (PSF) ↪ GITLAB

```
void psf(const long L, MatrixXd& S) {
  VectorXd x = VectorXd::LinSpaced(2*L+1, -L, L);
  MatrixXd X,Y;
  meshgrid(x, x, X, Y);
  MatrixXd E = MatrixXd::Ones(2*L+1, 2*L+1);
  S = E.cwiseQuotient(E + X.cwiseProduct(X) + Y.cwiseProduct(Y));
  S /= S.sum();
}
```

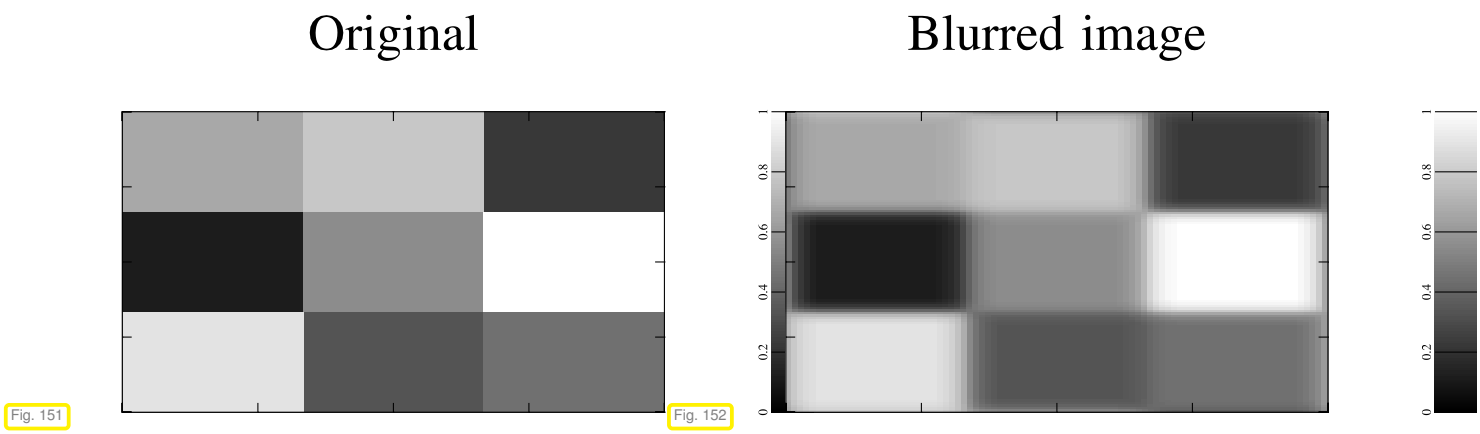


Fig. 151 Fig. 152

Note: (4.2.57) defines a linear operator $\mathcal{B} : \mathbb{R}^{m,n} \mapsto \mathbb{R}^{m,n}$ ("blurring operator")

C++11-code 4.2.59: Blurring operator ➙ GITLAB

```
MatrixXd blur(const MatrixXd& P, const MatrixXd& S) {
  const long m = P.rows(), n = P.cols(),
             M = S.rows(), N = S.cols(), L = (M-1)/2;
  if (M != N) {
    std::cout << "Error: S not quadratic!\n";
  }

  MatrixXd C(m,n);
  for (long l = 1; l <= m; ++l) {
    for (long j = 1; j <= n; ++j) {
      double s = 0;
      for (long k = 1; k <= (2*L+1); ++k) {
        for (long q = 1; q <= (2*L+1); ++q) {
          double kl = l + k - L - 1;
          if (kl < 1) kl += m;
          else if (kl > m) kl -= m;
          double jm = j + q - L - 1;
          if (jm < 1) jm += n;
          else if (jm > n) jm -= n;
          s += P(kl-1, jm-1)*S(k-1,q-1);
        }
      }
      C(l-1,j-1) = s;
    }
  }
  return C;
}
```

Now we revisit the considerations of § 4.2.43 and recall the derivation of (4.2.10) and Lemma 4.2.16.

$$\left(\mathcal{B}\left(\left(\omega_m^{\nu k}\omega_n^{\mu q}\right)_{k,q\in\mathbb{Z}}\right)\right)_{l,j} = \sum_{k=-L}^{L}\sum_{q=-L}^{L} s_{k,q}\omega_m^{\nu(l+k)}\omega_n^{\mu(j+q)} = \omega_m^{\nu l}\omega_n^{\mu j}\sum_{k=-L}^{L}\sum_{q=-L}^{L} s_{k,q}\omega_m^{\nu k}\omega_n^{\mu q}\,.$$

◮ $\mathbf{V}_{\nu,\mu} := \left(\omega_m^{\nu k}\omega_n^{\mu q}\right)_{k,q\in\mathbb{Z}}$, $0 \le \mu < m$, $0 \le \nu < n$ are the eigenvectors of $\mathcal{B}$:

$$\mathcal{B}\mathbf{V}_{\nu,\mu} = \lambda_{\nu,\mu}\mathbf{V}_{\nu,\mu} \quad , \quad \text{eigenvalue} \quad \lambda_{\nu,\mu} = \underbrace{\sum_{k=-L}^{L}\sum_{q=-L}^{L} s_{k,q}\omega_m^{\nu k}\omega_n^{\mu q}}_{\text{2-dimensional DFT of point spread function !}} \tag{4.2.60}$$

Thus the inversion of the blurring operator boils down to componentwise scaling in "Fourier domain", see See also Code 4.2.55 for the same idea.

C++11-code 4.2.61: DFT based deblurring ↪ GITLAB

```
MatrixXd deblur(const MatrixXd& C, const MatrixXd& S, const double
    tol=1e-3) {
  const long m = C.rows(), n = C.cols(),
             M = S.rows(), N = S.cols(), L = (M-1)/2;
  if (M != N) {
    std::cerr << "Error: S not quadratic!\n";
  }

  MatrixXd Spad = MatrixXd::Zero(m,n);
  // Zero padding
  Spad.block(0, 0, L+1, L+1) = S.block(L, L, L+1, L+1);
  Spad.block(m-L, n-L, L, L) = S.block(0, 0, L, L);
  Spad.block(0, n-L, L+1, L) = S.block(L, 0, L+1, L);
  Spad.block(m-L, 0, L, L+1) = S.block(0, L, L, L+1);
  // Inverse of blurring operator (fft2 expects a complex matrix)
  MatrixXcd SF = fft2(Spad.cast<complex>());
  // Test for invertibility
  if (SF.cwiseAbs().minCoeff() < tol*SF.cwiseAbs().maxCoeff()) {
    std::cerr << "Error: Deblurring impossible!\n";
  }
  // DFT based deblurring
  return fft2(ifft2(C.cast<complex>()).cwiseQuotient(SF)).real();
}
```

4.2.5 Semi-discrete Fourier Transform [?, Sect. 10.11]

Starting from Rem. 4.1.29 we mainly looked at time-discrete n-*periodic* signals, which can be mapped to vectors $\in \mathbb{R}^n$. This led to discrete periodic convolution ($\rightarrow$ Def. 4.1.33) and the discrete Fourier transform (DFT) ($\rightarrow$ Def. 4.2.18) as (bi-)linear mappings in $\mathbb{C}^n$.

In this section we are concerned with non-periodic signals of infinite duration as introduced in § 4.0.1.

Idea: Study the limit $n \to \infty$ for the n-periodic setting and DFT.

Let $(y_k)_{k\in\mathbb{Z}}$ be an n-periodic sequence (signal), $n = 2m+1$, $m \in \mathbb{N}$. Thanks to periodicity we can rewrite

$$\text{DFT} \rightarrow \text{Def. 4.2.18:} \qquad c_k = \sum_{j=-m}^{m} y_j \exp(-2\pi i \frac{kj}{n}) , \quad k = 0,\ldots,n-1 . \tag{4.2.62}$$

Now we associate a point $t_k \in [0,1[$ with each index k for the components of the transformed signal $(c_k)_{k=0}^{n-1}$:

$$k \in \{0,\ldots,n-1\} \quad \longleftrightarrow \quad t_k := \frac{k}{n} . \tag{4.2.63}$$

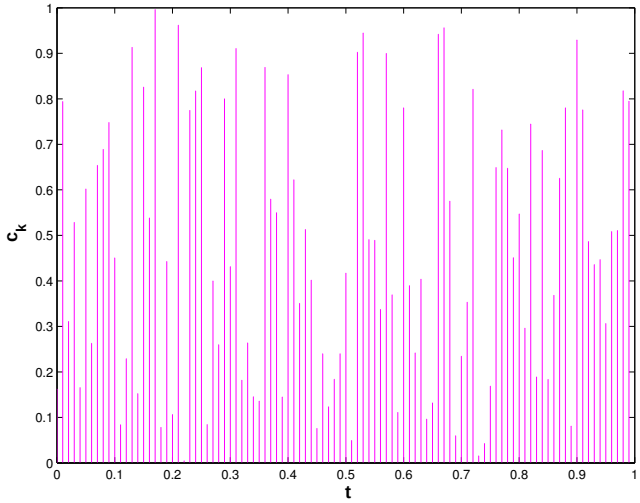


Fig. 153

◁ "Squeezing" a vector $\in \mathbb{R}^n$ into $[0,1[$.

Thus we can read the values c_k as sampled values of a functions defined on $[0,1]$

$$c_k \quad \leftrightarrow \quad c(t_k);,$$

$$t_k = \frac{k}{n}, \quad k = 0,\ldots,n-1\,.$$

This makes it possible to pass from a discrete finite signal to a continuous signal.

Thus, formally, we can rewrite (4.2.62) as

$$\text{DFT:} \qquad c(t_k) := c_k = \sum_{j=-m}^{m} y_j \exp(-2\pi\imath j t_k)\,, \quad k = 0,\ldots,n-1\,. \tag{4.2.64}$$

The notation indicates that we read c_k as the value of a *function* $c : [0,1[\mapsto \mathbb{C}$ for argument t_k.

Example 4.2.65 ("Squeezed" DFT of a periodically truncated signal)

Bi-infinite discrete signal, "concentrated around 0": $y_j = \dfrac{1}{1+j^2}, \quad j \in \mathbb{Z}$.

We examine the DFT of the $2m+1$-periodic signal obtained by periodic extension of $(y_k)_{k=-m}^{m}$.

Period of signal $y_i = 33$

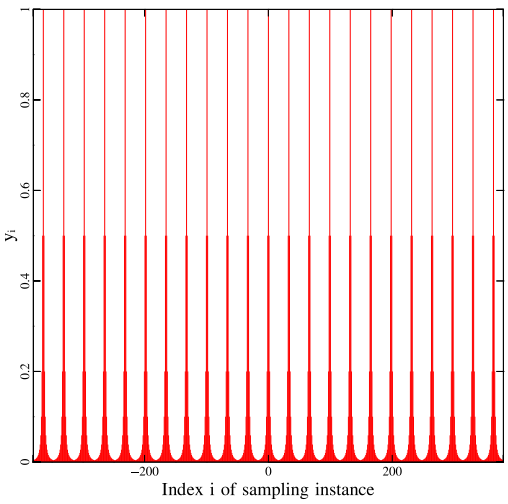


Fig. 154

DFT of period 33 signal

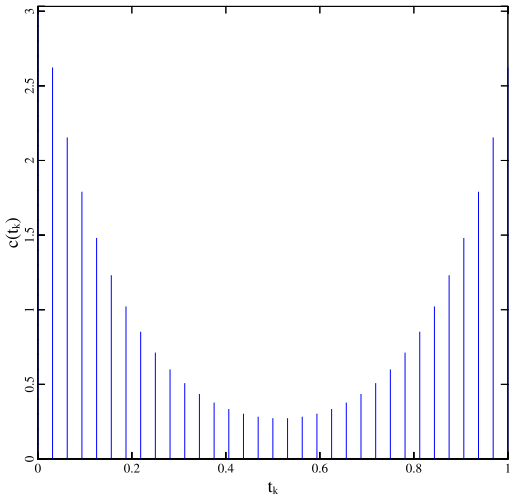


Fig. 155

Period of signal $y_i = 65$

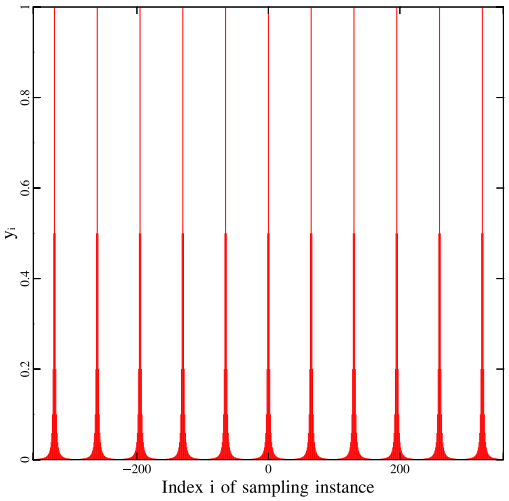


Fig. 156

DFT of period 65 signal

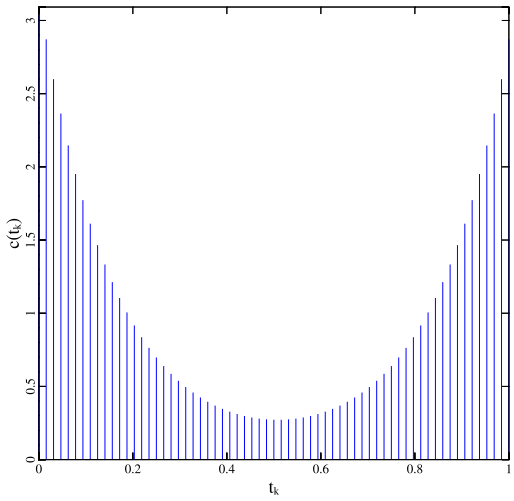


Fig. 157

Period of signal $y_i = 129$

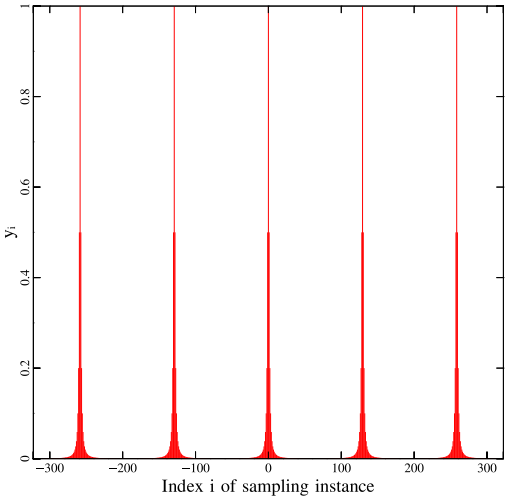


Fig. 158

DFT of period 129 signal

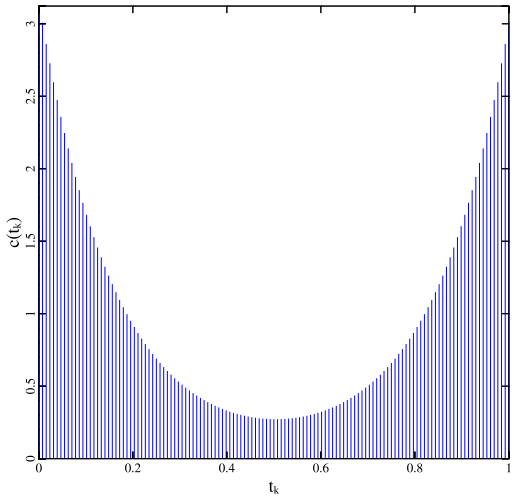


Fig. 159

Period of signal $y_i = 257$

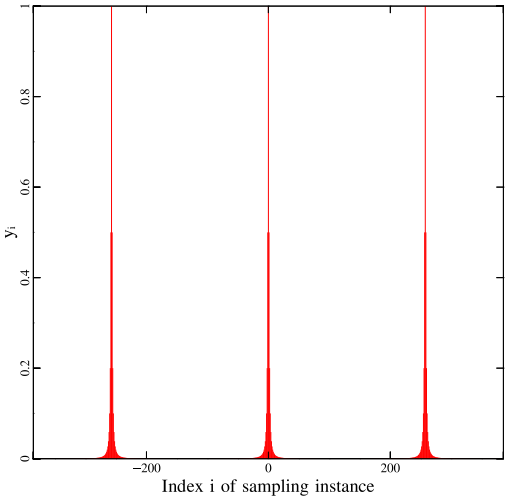


Fig. 160

DFT of period 257 signal

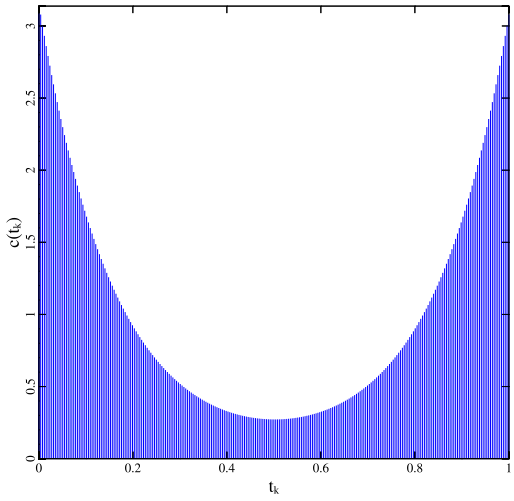


Fig. 161

Visual impression: $c(t_k)$ "converge" to a function $c : [0,1[\mapsto \mathbb{R}$.

C++11 code 4.2.66: Plotting a periodically truncated signal and its DFT ⇝ GITLAB

```
// Visualize limit m → ∞ for a 2m+!-periodic signal and
// its discrete Fourier transform "squeezed" into [0,1].
int main() {
  // range of plot for visualization of discrete signal
  const int N = 3*257;
  // function defining discrete signal
  auto yfn = [](VectorXd k) {
    return (1/(1 + k.array()*k.array())).matrix();
  };
  // loop over different periods 2^l + 1
  for (int mpow = 4; mpow <= 7; ++mpow) {
    int m = std::pow(2, mpow);
    VectorXd ybas = yfn(VectorXd::LinSpaced(2*m+1, -m, m));
    const int Ncp = std::floor(N/(2*m+1));
    const int n = ybas.size();

    VectorXd y(n*Ncp);
    for (int i = 0; i < Ncp; ++i)
      y.segment(n*i, n) = ybas;

    // Stem plots vertical lines from 0 to the y-value
    Stem s1;
    s1.title = "Period of signal y_i = " + std::to_string(2*m+1);
    s1.xlabel = "Index i of sampling instance";
    s1.ylabel = "y_i";
    s1.file = "persig" + std::to_string(mpow); // where to save
    s1.plot(VectorXd::LinSpaced(n*Ncp, -n*Ncp/2., n*Ncp/2.), y, "r");

    Eigen::FFT<double> fft;
    // DFT of wrapped signal (one period)
    VectorXd yconc(n); yconc << ybas.tail(m+1), ybas.head(m);
    VectorXcd c = fft.fwd(yconc);
    Stem s2;
    s2.title = "DFT of period " + std::to_string(2*m+1) + " signal";
    s2.xlabel = "t_k";
    s2.ylabel = "c(t_k)";
    s2.file = "persigdft" + std::to_string(mpow);
    s2.plot(VectorXd::LinSpaced(2*m+1, 0, 1), c.real(), "b");
  }
  return 0;
}
```

Now we pass to the limit $m \to \infty$ (and keep the function perspective $c_k = c(t_k)$)

Note: passing to the limit amounts to dropping the assumption of periodicity!

$$\Rightarrow \quad \boxed{c(t) = \sum_{k\in\mathbb{Z}} y_k \exp(-2\pi\imath kt)} \,. \tag{4.2.67}$$

Terminology: The series (= infinite sum) on the right hand side of (4.2.67) is called a Fourier series (link)

The function $c : [0,1[\mapsto \mathbb{C}$ defined by (4.2.67) is called the Fourier transform of the sequence $(y_k)_{k\in\mathbb{Z}}$ (, if the series converges).

Corollary 4.2.68. Periodicity of Fourier transforms

Fourier transforms $t \mapsto c(t)$ are 1-periodic functions $\mathbb{R} \to \mathbb{C}$.

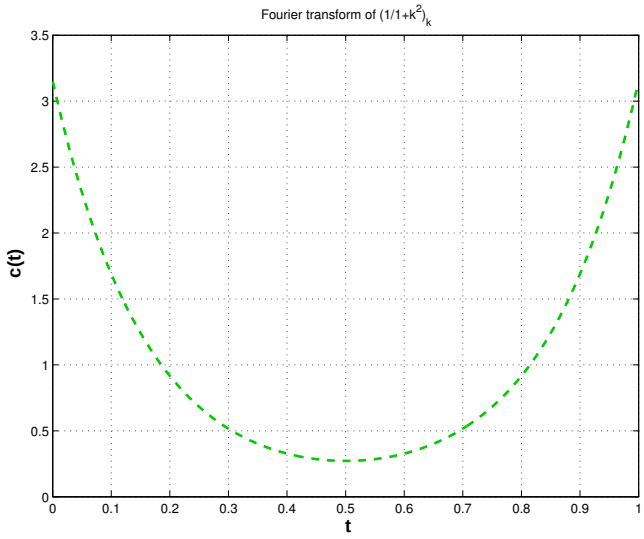


Fig. 162

Thus, the limit we "saw" in Ex. 4.2.65 is actually the Fourier transform of the sequence $(y_k)_{k\in\mathbb{Z}}$!

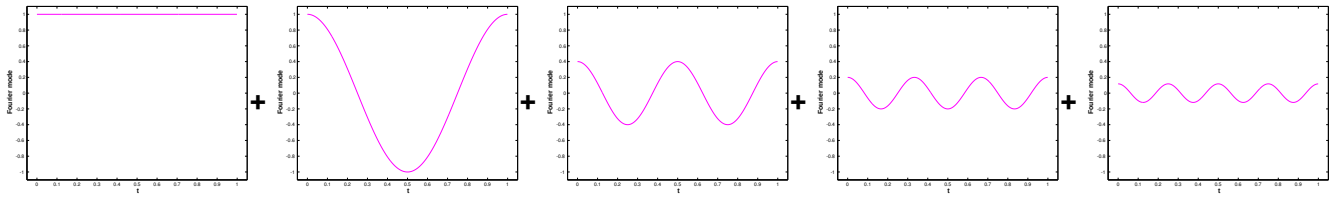
◁ Fourier transform of $y_k := \frac{1}{1+k^2}$

From (4.2.67) we conclude:

Fourier transform

=

weighted sum of Fourier modes $t \mapsto \exp(-2\pi\imath kt)$, $k \in \mathbb{Z}$



→ related animation on Wikipedia.

It is possible to derive a closed-form expression for the function displayed in Fig. 163:

$$c(t) = \sum_{k\in\mathbb{Z}} \frac{1}{1+k^2} \exp(-2\pi\imath kt) = \frac{\pi}{e^{\pi} - e^{-\pi}} \cdot \left(e^{\pi-2\pi t} + e^{2\pi t-\pi}\right) \in C^{\infty}([0,1]) \,.$$

Note that when considered as a 1-periodic function on $\mathbb{R}$, this $c(t)$ is merely continuous.

Remark 4.2.69 (Decay conditions for bi-infinite signals)

The considerations above were based on

- truncation of $(y_k)_{k\in\mathbb{Z}}$ to $(y_k)_{k=-m}^{m}$ and
- periodic continuation to an $2m+1$-periodic signal.

Obviously, only if the signal is *concentrated around* $k = 0$ this procedure will not lose essential information contained in the signal.

$$\text{Minimal requirement:} \quad \lim_{k\to\infty} |y_k| = 0 \,, \tag{4.2.70}$$

$$\text{Stronger requirement:} \quad \sum_{k\in\mathbb{Z}} |y_k| < \infty \,. \tag{4.2.71}$$

(4.2.71) ⇒ Fourier series (4.2.67) (link) converges uniformly [**?**, Def. 4.8.1]
⇒ $c : [0,1[\mapsto \mathbb{C}$ is continuous [**?**, Thm. 4.8.1].

Remark 4.2.72 (Numerical summation of Fourier series)

Assuming sufficiently fast decay of the signal $(y_k)_{k\in\mathbb{Z}}$ for $k \to \infty$ ($\to$ Rem. 4.2.69), we can *approximate* the Fourier series (4.2.67) by a Fourier sum

$$c(t) \approx c_M(t) := \sum_{k=-M}^{M} y_k \exp(-2\pi i k t) , \quad M \gg 1 . \tag{4.2.73}$$

Task: Approximate evaluation of $c(t)$ at N equidistant points $t_j := \frac{j}{N}$, $j = 0, \ldots, N$ (e.g., for plotting it).

$$c(t_j) = \lim_{M\to\infty} \sum_{k=-M}^{M} y_k \exp(-2\pi i k t_j) \approx \sum_{k=-M}^{M} y_k \exp(-2\pi i \frac{kj}{N}) , \tag{4.2.74}$$

for $j = 0, \ldots, N-1$.

Note: If $N = M$ ➢ (4.2.74) is a discrete Fourier transform ($\to$ Def. 4.2.18).

C++11 code 4.2.75: DFT-based evaluation of Fourier sum at equidistant points ↬ GITLAB

```
# include "feval.hpp" // evaluate scalar function with a vector
// DFT based approximate evaluation of Fourier series
// signal is a handle to a function providing the y_k
// M specifies truncation of series according to (4.2.73)
// N is the number of equidistant evaluation points for c in [0,1[.
template <class Function>
VectorXcd foursum(const Function& signal, int M, int N) {
  const int m = 2*M+1; // length of the signal
  // sample signal
  VectorXd y = feval(signal, VectorXd::LinSpaced(m, -M, M));
  // Ensure that there are more sampling points than terms in series
  int l; if (m > N) { l = ceil(double(m)/N); N *= l; } else l = 1;
  // Zero padding and wrapping of signal, see Code 4.2.33
  VectorXd y_ext = VectorXd::Zero(N);
  y_ext.head(M+1) = y.tail(M+1);
  y_ext.tail(M) = y.head(M);
  // Perform DFT and decimate output vector
  Eigen::FFT<double> fft;
  Eigen::VectorXcd k = fft.fwd(y_ext), c(N/l);
  for (int i = 0; i < N/l; ++i) c(i) = k(i*l);
  return c;
}
```

Example 4.2.76 (Convergence of Fourier sums)

Infinite signal, satisfying decay condition (4.2.71): $y_k = \frac{1}{1+k^2}$, see Ex. 4.2.65.

Monitored: approximation of Fourier transform $c(t)$ by Fourier sums $c_m(t)$, see (4.2.73).

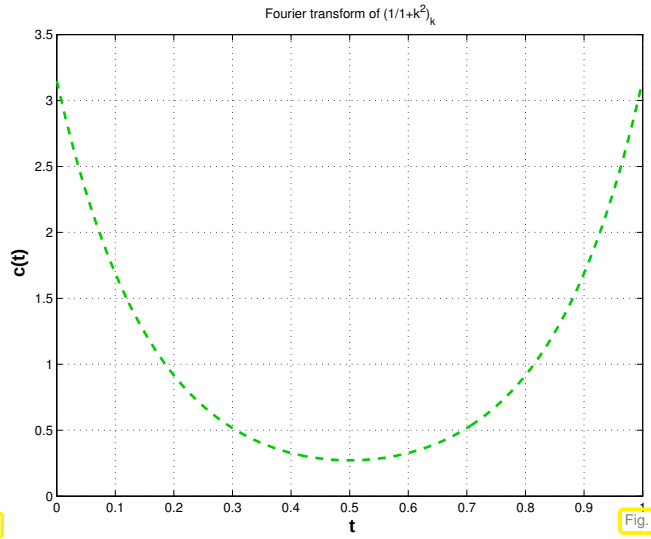


Fig. 163

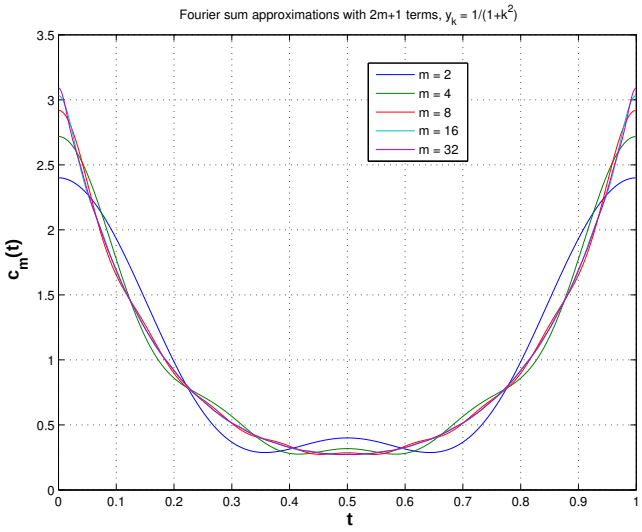


Fig. 164

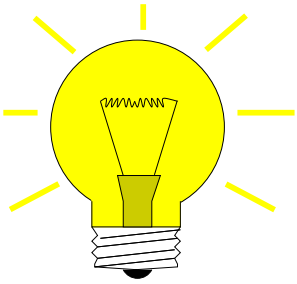
Observation: Convergence of Fourier sums in "eyeball norm"; quantitative statements about convergence can be deduced from Thm. 4.2.89.

Same limit considerations as above for the inverse DFT (4.2.20), $n = 2m+1$,

$$y_j = \frac{1}{n}\sum_{k=0}^{n-1} c_k \exp(2\pi i \frac{jk}{n}) \;, \quad j = -m, \ldots, m \;. \tag{4.2.77}$$

Adopt function perspective as before: $c_k \leftrightarrow c(t_k)$, $t_k = \frac{k}{n}$, *cf.* (4.2.63).

$$y_j = \frac{1}{n}\sum_{k=0}^{n-1} c(t_k) \exp(2\pi i j t_k) \;, \quad j = -m, \ldots, m \;. \tag{4.2.78}$$



Now: pass to the limit $m \to \infty$ in (4.2.78)

Idea: right hand side of (4.2.78) = Riemann sum, *cf.* [**?**, Sect. 6.2]

☛ in the limit $m \to \infty$ the sum becomes an integral!

$$\boxed{y_j = \int_0^1 c(t) \exp(2\pi i j t)\, \mathrm{d}t} \;. \tag{4.2.79}$$

This formula is the inversion of the summation of a Fourier series (4.2.67)!

The formula (4.2.79) allows to recover the signal $(y_k)_{k\in\mathbb{Z}}$ from its Fourier transform $c(t)$.

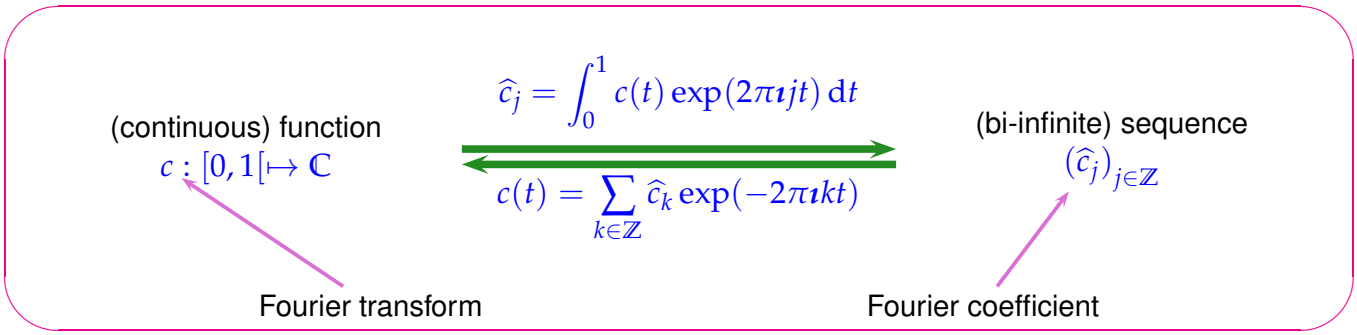
Terminology: y_j from (4.2.79) is called the j-th Fourier coefficient of the function c.

✎ Notation: $\widehat{c}_j := y_j$ with y_j defined by (4.2.79) $\hat{=}$ j-th Fourier coefficient of $c : [0,1[\to \mathbb{C}$

Note: Both, the mapping $(\widehat{c}_k)_{k\in\mathbb{Z}} \mapsto c$ from (4.2.67), and the mapping $c \mapsto (\widehat{c}_k)_{k\in\mathbb{Z}}$ from (4.2.79) are *linear*!

(Recall the concept of a linear mapping as explained in [**?**, Ch. 6].)

Summary of a fundamental correspondence:



Remark 4.2.80 (Filtering in Fourier domain)

What happens to the Fourier transform of a bi-infinite signal, if it passes through a channel?

Consider a (bi-)infinite signal $(x_k)_{k\in\mathbb{Z}}$ sent through a finite (→ Def. 4.1.4, linear (→ Def. 4.1.9) time-invariant (→ Def. 4.1.7) causal (→ Def. 4.1.11) channel with impulse response (→ 4.1.3) $(\ldots,0,h_0,\ldots,h_{n-1}0,\ldots)$ (→ § 4.1.1).

➢ this results in the output signal, see (4.1.14),

$$y_k = \sum_{j=0}^{n-1} h_j x_{k-j}\,, \quad k \in \mathbb{Z}\,. \tag{4.2.81}$$

Fourier transforms of signals: $(y_k)_{k\in\mathbb{Z}} \leftrightarrow c(t)$, $(x_j)_{j\in\mathbb{Z}} \leftrightarrow b(t)$

(Assume that $(x_k)_{k\in\mathbb{Z}}$ satisfies (4.2.71))

$$\begin{aligned} c(t) = \sum_{k\in\mathbb{Z}} y_k \exp(-2\pi\imath kt) &= \sum_{k\in\mathbb{Z}} \sum_{j=0}^{n-1} h_j x_{k-j} \exp(-2\pi\imath kt) \\ \text{[shift summation index } k\text{]} \quad &= \sum_{j=0}^{n-1} \sum_{k\in\mathbb{Z}} h_j x_k \exp(-2\pi\imath jt)\exp(-2\pi\imath kt) \\ &= \underbrace{\left(\sum_{j=0}^{n-1} h_j \exp(-2\pi\imath jt)\right)}_{\text{trigonometric polynomial of degree } n-1} b(t)\,. \end{aligned} \tag{4.2.82}$$

Definition 4.2.83. Trigonometric polynomial

A trigonometric polynomial is a function $\mathbb{R} \to \mathbb{C}$ that is a weighted sum of finitely many terms $t \to \exp(-2\pi\imath kt)$, $k \in \mathbb{Z}$.

We summarize the insight gleaned from (4.2.82):

Discrete convolution in Fourier domain

The discrete convolution of a signal with finite impulse response amounts to a multiplication of its Fourier transform with a trigonometric polynomial whose coefficients are given by the impulse response.

(4.2.85) Isometry property of Fourier transform

We find a conservation of power through Fourier transform:

Lemma 4.2.14 ➢ for Fourier matrix $\mathbf{F}_n$, see (4.2.13), $\frac{1}{\sqrt{n}}\mathbf{F}_n$ is *unitary* (→ Def. 6.2.2)

$$\overset{\text{Thm. 3.3.5}}{\blacktriangleright} \quad \left\| \frac{1}{\sqrt{n}}\mathbf{F}_n \mathbf{y} \right\|_2 = \|\mathbf{y}\|_2 \,. \tag{4.2.86}$$

Since DFT boils down to multiplication with $\mathbf{F}_n$ (→ Def. 4.2.18), we conclude from (4.2.86)

$$c_k \text{ from (4.2.62)} \quad \Rightarrow \quad \frac{1}{n}\sum_{k=0}^{n-1} |c_k|^2 = \sum_{j=-m}^{m} |y_j|^2 \,. \tag{4.2.87}$$

Now we adopt the function perspective again and associated $c_k \leftrightarrow c(t_k)$. Then we pass to the limit $m \to \infty$, appeal to Riemann summation (see above), and conclude

$$(4.2.87) \quad \overset{m\to\infty}{\Longrightarrow} \quad \int_0^1 |c(t)|^2 \,\mathrm{d}t = \sum_{j\in\mathbb{Z}} |y_j|^2 \,. \tag{4.2.88}$$

Theorem 4.2.89. Isometry property of Fourier transform

If $\sum_{k\in\mathbb{Z}} |\widehat{c}_j|^2 < \infty$, *then the Fourier series*

$$c(t) = \sum_{k\in\mathbb{Z}} \widehat{c}_k \exp(-2\pi\imath kt)$$

yields a function $c \in L^2(]0,1[)$ *that satisfies*

$$\|c\|^2_{L^2(]0,1[)} := \int_0^1 |c(t)|^2 \,\mathrm{d}t = \sum_{k\in\mathbb{Z}} |\widehat{c}_j|^2 \,.$$

Recalling the concept of the L^2-norm of a function, see (5.2.67), the theorem can be stated as follows:

Thm. 4.2.89 ↔ The L^2-norm of a Fourier transform agrees with the Euclidean norm of the corresponding signal.

Note: Euclidean norm of a sequence $\left\|(y_k)_{k\in\mathbb{Z}}\right\|_2^2 := \sum_{k\in\mathbb{Z}} |y_j|^2$.

4.3 Fast Fourier Transform (FFT)

You might have been wondering why the reduction to DFTs received sop much attention in Section 4.2.1. An explanation is given now.

Supplementary reading. [**?**, Sect. 8.7.3], [**?**, Sect. 53], [**?**, Sect. 10.9.2]

At first glance (at (4.2.19)): DFT in $\mathbb{C}^n$ seems to require asymptotic computational effort of $O(n^2)$ (matrix×vector multiplication with dense matrix).

C++-code 4.3.1: Naive DFT-implementation

```
// DFT (4.2.19) of vector y returned in c
void naivedft(const VectorXcd& y, VectorXcd& c) {
  using idx_t = VectorXcd::Index;
  const idx_t n = y.size();
  const std::complex<double> i(0,1);
  c.resize(n);
  // root of unity ω_n, w holds its powers
  std::complex<double> w = std::exp(-2*M_PI/n*i),s = w;
  c(0) = y.sum();
  for (long j = 1; j < n; ++j) {
    c(j) = y(n-1);
    for (long k = n-2; k >= 0; --k) c(j) = c(j)*s + y(k);
    s *= w;
  }
}
```

Example 4.3.2 (Efficiency of `fft`)

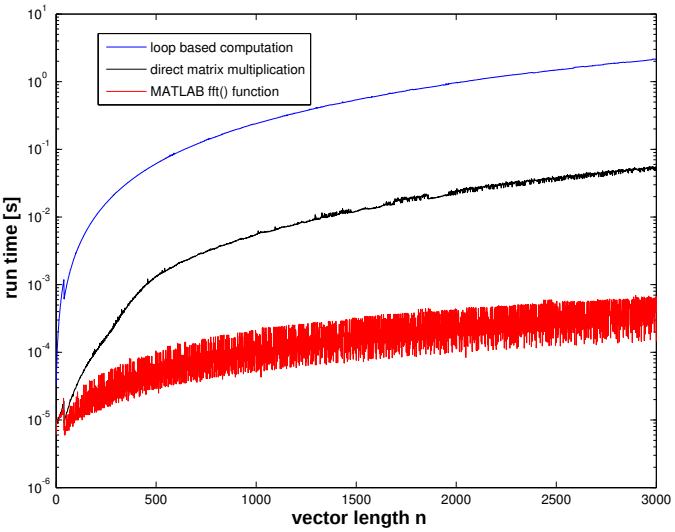
C++-code 4.3.3: timing of different implementations of DFT

```
#include <complex>
#include <Eigen/Dense>
#include <unsupported/Eigen/FFT>
#include <figure/figure.hpp>
#include "timer.h"
#include "meshgrid.hpp"
#include "eigen_fft_backend_name.hpp"

using namespace Eigen;

//!  Benchmarking discrete fourier transformations
// N = maximal length of the transformed vector
// nruns = no.  of runs on which to take the minimal timing
// If the size of the transformed array is a prime number the kissfft
// implementation of EIGEN performs extremly bad, use FFTW or MKL
// in these cases
void benchmark(const int N, const int nruns=3) {
  std::complex<double> i(0,1); // imaginary unit

//* Version which computes for every vector size (extremely slow)
// MatrixXd res(N-1, 4); // save timing results and vector size
// for (int n = 2; n <= N; ++n)
//* Version which computes for powers of two only
  MatrixXd res(int(std::log2(N)), 4); // save timing results and vector size
  for (int n = 2, j = 0; n <= N; n*=2, ++j) {
```

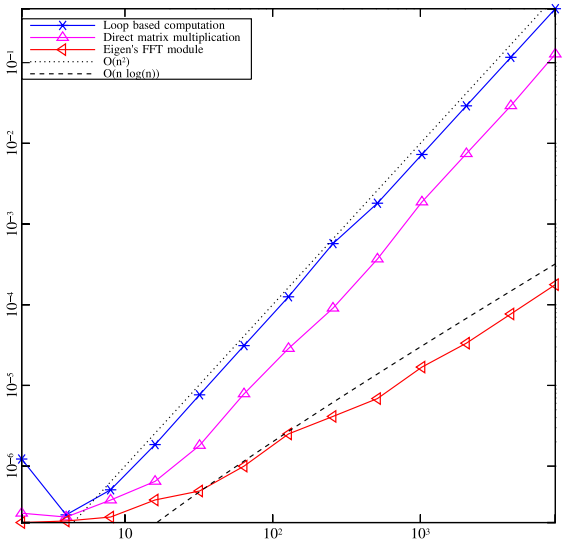
```
26      VectorXd y = VectorXd::Random(n);
27      VectorXcd c = VectorXcd::Zero(n);
28      // compute fourier transformed with a loop implementation
29      Timer t1; t1.start();
30      for (int k = 0; k < nruns; ++k) {
31        std::complex<double> omega = std::exp(-2*M_PI/n*i),
32                             s = omega;
33        c(0) = y.sum();
34        for (int j = 1; j < n; ++j) {
35          c(j) = y(n-1);
36          for (int l = n-1; l > 0; --l){ c(j) = c(j)*s + y(l); }
37          s *= omega;
38        }
39        t1.lap();
40      }
41      // compute fourier transformed through matrix multiplication
42      MatrixXd I, J; VectorXd lin = VectorXd::LinSpaced(n, 0, n-1);
43      meshgrid(lin, lin, I, J);
44      MatrixXcd F = (-2*M_PI/n*i*I.cwiseProduct(J)).array().exp().matrix();
45      Timer t2; t2.start();
46      for (int k = 0; k < nruns; ++k) {
47        c = F*y; t2.lap();
48      }
49      // use Eigen's FFT to compute fourier transformation
50      // Note:  slow for large primes!
51      Eigen::FFT<double> fft;
52      Timer t3; t3.start();
53      for (int k = 0; k < nruns; ++k) {
54        fft.fwd(c,y); t3.lap();
55      }
56      // save timings
57      res(j,0) = n; res(j,1) = t1.min();
58      res(j,2) = t2.min(); res(j,3) = t3.min();
59    }
60
61    mgl::Figure fig;
62    fig.title("FFT timing");
63    fig.setlog(true, true);
64    fig.plot(res.col(0), res.col(1), "b*").label("Loop based computation");
65    fig.plot(res.col(0), res.col(2), "m^").label("Direct matrix multiplication");
66    fig.plot(res.col(0), res.col(3), "r<").label("Eigen's FFT module");
67    fig.fplot("x^2*1e-8","k:").label("O(n^2)");
68    fig.fplot("x*lg(x)*1e-8","k;").label("O(n log(n))");
69    fig.legend(0,1);
70    fig.save("ffttime");
71    std::cout << res << std::endl;
72  }
73
74  int main() {
75    // print the backend used by eigen for the fft
76    std::cout << "FFT Backend: " << eigen_fft_backend_name() << std::endl;
77
78    // WARNING: this code may take very long to run
79    benchmark(1024*8);
80    return 0;
81  }
```

Timings in MATLAB ▷:

1. Straightforward implementation involving MATLAB loops
2. Multiplication with Fourier matrix (4.2.13)
3. MATLAB's built-in function `fft()`



FFT timing



Timings in EIGEN ▷: (only for $n = 2^L$!)

1. Straightforward implementation involving C++ loops, see Code 4.3.1
2. Multiplication with Fourier matrix (4.2.13)
3. EIGEN's built-in **FFT** class, method `fwd()`

Note: Eigen uses KISS FFT as default backend in its FFT module, which falls back loop-based slow DFT when used on data sizes, which are large primes. An superior alternative is FFTW, see § 4.3.11.

The secret of EIGEN's `fft()`:

the Fast Fourier Transform (FFT) algorithm [**?**]

(discovered by C.F. Gauss in 1805, rediscovered by Cooley & Tuckey in 1965, one of the "top ten algorithms of the century").

An elementary manipulation of (4.2.19) for $n = 2m$, $m \in \mathbb{N}$:

$$\begin{aligned} c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n}jk} \\ &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n}2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n}(2j+1)k} \\ &= \underbrace{\sum_{j=0}^{m-1} y_{2j} \underbrace{e^{-\frac{2\pi i}{m}jk}}_{=\omega_m^{jk}}}_{=:\widetilde{c}_k^{\mathtt{even}}} + e^{-\frac{2\pi i}{n}k} \cdot \underbrace{\sum_{j=0}^{m-1} y_{2k+1} \underbrace{e^{-\frac{2\pi i}{m}jk}}_{=\omega_m^{jk}}}_{=:\widetilde{c}_k^{\mathtt{odd}}} . \end{aligned} \tag{4.3.4}$$

Note m-periodicity: $\widetilde{c}_k^{\text{even}} = \widetilde{c}_{k+m}^{\text{even}}$, $\widetilde{c}_k^{\text{odd}} = \widetilde{c}_{k+m}^{\text{odd}}$.

Note: $\widetilde{c}_k^{\text{even}}$, $\widetilde{c}_k^{\text{odd}}$ from DFTs of length m!

$$\text{with } \mathbf{y}_{\text{even}} := (y_0, y_2, \ldots, y_{n-2})^\top \in \mathbb{C}^m: \quad (\widetilde{c}_k^{\text{even}})_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\text{even}} ,$$

$$\text{with } \mathbf{y}_{\text{odd}} := (y_1, y_3, \ldots, y_{n-1})^\top \in \mathbb{C}^m: \quad (\widetilde{c}_k^{\text{odd}})_{k=0}^{m-1} = \mathbf{F}_m \mathbf{y}_{\text{odd}} .$$

(4.3.4): DFT of length $2m$ = $2\times$ DFT of length m + 2m additions & multiplications

Idea: divide & conquer recursion

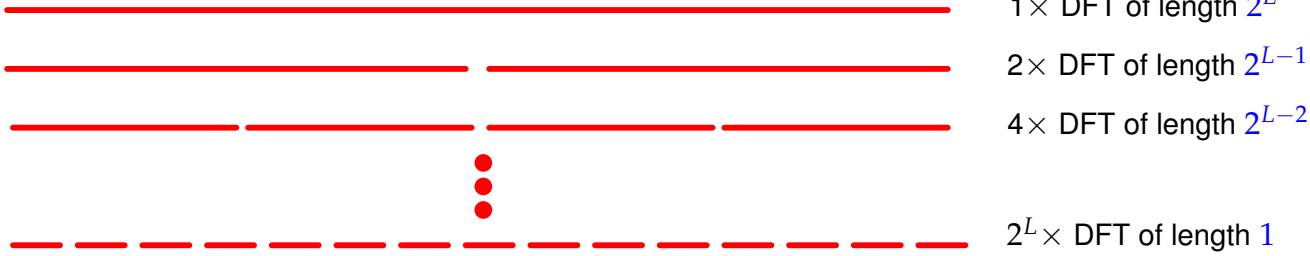
FFT-algorithm

Recursive demonstration code for DFT of length $n = 2^L$:

C++11 code 4.3.5: Recursive FFT ↪ GITLAB

```
// Recursive DFT for vectors of length n = 2^L
VectorXcd fftrec(const VectorXcd& y) {
  using idx_t = VectorXcd::Index;
  using comp = std::complex<double>;
  // Nothing to do for DFT of length 1
  const idx_t n = y.size();
  if (n == 1) return y;
  if (n % 2 != 0) throw std::runtime_error("size(y) must be even!");
  const Eigen::Map<const
      Eigen::Matrix<comp,Eigen::Dynamic,Eigen::Dynamic,Eigen::RowMajor>>
    Y(y.data(),n/2,2); // for selecting even and off compoents
  const VectorXcd c1 = fftrec(Y.col(0)),c2 = fftrec(Y.col(1));
  const comp omega = std::exp(-2*M_PI/n*comp(0,1)); // root of unity
     ω_n
  comp s(1.0,0.0);
  VectorXcd c(n);
  // Scaling of DFT of odd components plus periodic copying
  for (long k = 0; k < n; ++k) {
    c(k) = c1(k%(n/2)) + c2(k%(n/2))*s;
    s *= omega;
  }
  return c;
}
```

Computational cost of `fftrec`:



Code 4.3.5: each level of the recursion requires $O(2^L)$ elementary operations.

Asymptotic complexity

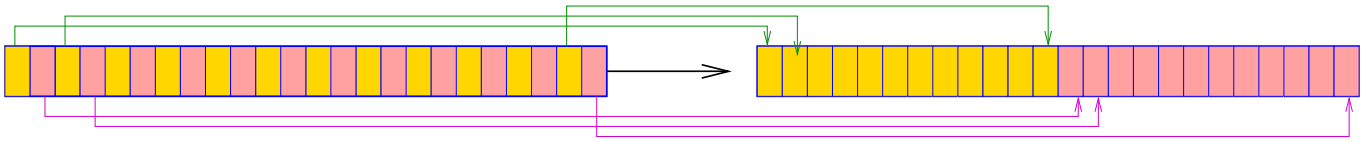
Asymptotic complexity of FFT algorithm, $n = 2^L$: $O(L2^L) = O(n \log_2 n)$

(`fft.fwd()`/`fft.inv()`-function calls: computational cost is $\approx 5n \log_2 n$).

Remark 4.3.7 (FFT algorithm by matrix factorization)

For $n = 2m$, $m \in \mathbb{N}$,

consider even-odd sorting $P_m^{\mathrm{OE}}(1, \ldots, n) = (1, 3, \ldots, n-1, 2, 4, \ldots, n)$.



As $\omega_n^{2j} = \omega_m^j$:

$$\text{permutation of rows} \quad P_m^{\mathrm{OE}} \mathbf{F}_n = \left[\begin{array}{c|c} \mathbf{F}_m & \mathbf{F}_m \\ \hline \mathbf{F}_m \begin{bmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \end{bmatrix} & \mathbf{F}_m \begin{bmatrix} \omega_n^{n/2} & & & \\ & \omega_n^{n/2+1} & & \\ & & \ddots & \\ & & & \omega_n^{n-1} \end{bmatrix} \end{array}\right] =$$

$$\left[\begin{array}{c|c} \mathbf{F}_m & \\ \hline & \mathbf{F}_m \end{array}\right] \left[\begin{array}{c|c} \mathbf{I} & \mathbf{I} \\ \hline \begin{bmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \end{bmatrix} & \begin{bmatrix} -\omega_n^0 & & & \\ & -\omega_n^1 & & \\ & & \ddots & \\ & & & -\omega_n^{n/2-1} \end{bmatrix} \end{array}\right]$$

This reveals how to apply a divide-and-conquer idea when evaluating $\mathbf{F}_n \mathbf{x}$.

Example: partitioning of Fourier matrix for $n = 10$

$$P_5^{\mathrm{OE}}\mathbf{F}_{10} = \left[\begin{array}{ccccc|ccccc} \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 & \omega^0 & \omega^2 & \omega^4 & \omega^6 & \omega^8 \\ \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 & \omega^0 & \omega^4 & \omega^8 & \omega^2 & \omega^6 \\ \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 & \omega^0 & \omega^6 & \omega^2 & \omega^8 & \omega^4 \\ \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 & \omega^0 & \omega^8 & \omega^6 & \omega^4 & \omega^2 \\ \hline \omega^0 & \omega^1 & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 & \omega^8 & \omega^9 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 & \omega^2 & \omega^5 & \omega^8 & \omega^1 & \omega^4 & \omega^7 \\ \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 & \omega^0 & \omega^5 \\ \omega^0 & \omega^7 & \omega^4 & \omega^1 & \omega^8 & \omega^5 & \omega^2 & \omega^9 & \omega^6 & \omega^3 \\ \omega^0 & \omega^9 & \omega^8 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega^1 \end{array}\right], \quad \omega := \omega_{10}\,.$$

What if $n \neq 2^L$? Quoted from MATLAB manual:

To compute an n-point DFT when n is composite (that is, when $n = pq$), the FFTW library decomposes the problem using the Cooley-Tukey algorithm, which first computes p transforms of size q, and then computes q transforms of size p. The decomposition is applied recursively to both the p- and q-point DFTs until the problem can be solved using one of several machine-generated fixed-size "codelets." The codelets in turn use several algorithms in combination, including a variation of Cooley-Tukey, a prime factor algorithm, and a split-radix algorithm. The particular factorization of n is chosen heuristically.

The execution time for fft depends on the length of the transform. It is fastest for powers of two. It is almost as fast for lengths that have only small prime factors. It is typically several times slower for lengths that are prime or which have large prime factors → Ex. 4.3.12.

Remark 4.3.8 (FFT based on general factorization)

Fast Fourier transform algorithm for DFT of length $n = pq$, $p, q \in \mathbb{N}$ (Cooley-Tuckey-Algorithm)

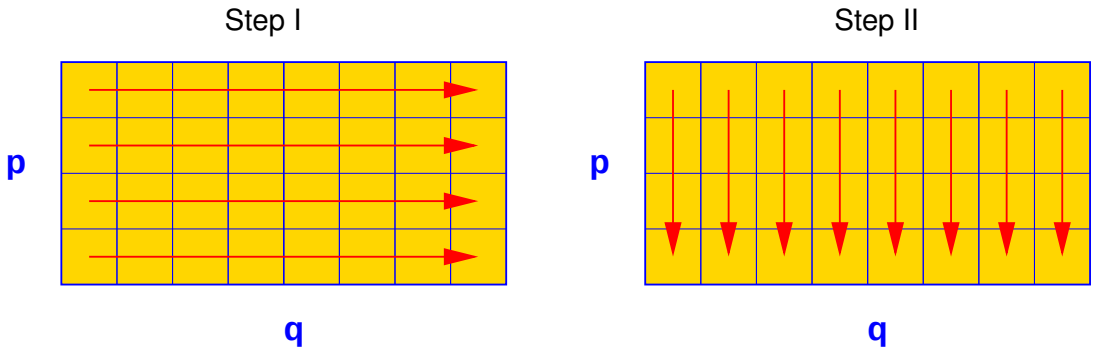
$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{jk} \overset{[j=:lp+m]}{=} \sum_{m=0}^{p-1}\sum_{l=0}^{q-1} y_{lp+m} e^{-\frac{2\pi i}{pq}(lp+m)k} = \sum_{m=0}^{p-1} \omega_n^{mk} \sum_{l=0}^{q-1} y_{lp+m}\, \omega_q^{l(k \mod q)}. \tag{4.3.9}$$

Step I: perform p DFTs of length q $\quad z_{m,k} := \sum_{l=0}^{q-1} y_{lp+m}\, \omega_q^{lk}, \quad 0 \le m < p,\ 0 \le k < q.$

Step II: for $k =: rq + s, \quad 0 \le r < p,\ 0 \le s < q$

$$c_{rq+s} = \sum_{m=0}^{p-1} e^{-\frac{2\pi i}{pq}(rq+s)m} z_{m,s} = \sum_{m=0}^{p-1} (\omega_n^{ms} z_{m,s}) \omega_p^{mr}$$

and hence q DFTs of length p give all c_k.



Remark 4.3.10 (FFT for prime n)

When $n \neq 2^L$, even the Cooley-Tuckey algorithm of Rem. 4.3.8 will eventually lead to a DFT for a vector with prime length.

Quoted from the MATLAB manual:

When n is a prime number, the FFTW library first decomposes an n-point problem into three $(n-1)$-point problems using Rader's algorithm [**?**]. It then uses the Cooley-Tukey decomposition described above to compute the $(n-1)$-point DFTs.

Details of Rader's algorithm: starting point is a theorem from number theory:

$$\forall p \in \mathbb{N} \text{ prime} \quad \exists g \in \{1,\ldots,p-1\}: \quad \{g^k \mod p : k = 1,\ldots,p-1\} = \{1,\ldots,p-1\}\,,$$

permutation $P_{p,g} : \{1,\ldots,p-1\} \mapsto \{1,\ldots,p-1\}$, $P_{p,g}(k) = g^k \mod p$,
reversing permutation $P_k : \{1,\ldots,k\} \mapsto \{1,\ldots,k\}$, $P_k(i) = k - i + 1$.

With these two permutations we can achieve something amazing:

For the Fourier matrix $\mathbf{F}_p = (f_{ij})_{i,j=1}^{p}$ the permuted block $P_{p-1}P_{p,g}(f_{ij})_{i,j=2}^{p-1}P_{p,g}^{\top}$ is circulant.

Example for $p = 13$: $g = 2$, permutation $(2\ 4\ 8\ 3\ 6\ 12\ 11\ 9\ 5\ 10\ 7\ 1)$

$$\mathbf{F}_{13} \longrightarrow \begin{array}{c|cccccccccccc}
\omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \hline
\omega^0 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 \\
\omega^0 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 \\
\omega^0 & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} \\
\omega^0 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 \\
\omega^0 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 \\
\omega^0 & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} \\
\omega^0 & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} \\
\omega^0 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 & \omega^6 \\
\omega^0 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 & \omega^3 \\
\omega^0 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 & \omega^8 \\
\omega^0 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2 & \omega^4 \\
\omega^0 & \omega^4 & \omega^8 & \omega^3 & \omega^6 & \omega^{12} & \omega^{11} & \omega^9 & \omega^5 & \omega^{10} & \omega^7 & \omega^1 & \omega^2
\end{array}$$

Then apply fast algorithms for multiplication with circulant matrices (= discrete periodic convolution, see § 4.1.37) to right lower $(n-1)\times(n-1)$ block of permuted Fourier matrix. These fast algorithms rely on DTFs of length $n-1$, see Code 4.2.25.

Asymptotic complexity of `fft.fwd()/fft.inv()` for $\mathbf{y}\in\mathbb{C}^n$ = $O(n\log n)$.

← Section 4.2.1

Asymptotic complexity of discrete periodic convolution,see Code 4.2.25:
Cost(`pconvfft(u,x)`, $\mathbf{u},\mathbf{x}\in\mathbb{C}^n$) = $O(n\log n)$.

Asymptotic complexity of discrete convolution, see Code 4.2.26:
Cost(`myconv(h,x)`, $\mathbf{h},\mathbf{x}\in\mathbb{C}^n$) = $O(n\log n)$.

(4.3.11) FFTW - A highly-performing self-tuning library for FFT

From FFTW homepage: **FFTW** is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data.
FFTW will perform well on most architectures without modification. Hence the name, "FFTW," which stands for the somewhat whimsical title of "Fastest Fourier Transform in the West."

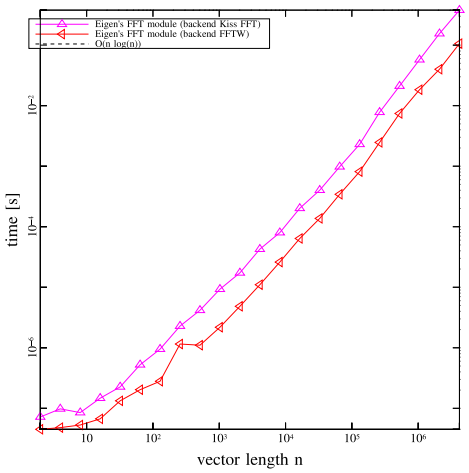
Supplementary reading. [**?**] offers a comprehensive presentation of the design and implementation of the FFTW library (version 3.x). This paper also conveys the many tricks it takes to achieve satisfactory performance for DFTs of arbitrary length.

FFTW can be installed from source following the instructions from the installation page after downloading the source code of FFTW 3.3.5 from the download page. Precompiled binaries for various linux distributions are available in their main package repositories:

- Ubuntu/Debian: `apt-get install fftw3 fftw3-dev`
- Fedora: `dnf install fftw fftw-devel`

EIGEN's FFT module can use different backend implementations, one of which is the **FFTW** library. The backend may be enabled by defining the preprocessor directive **Eigen_FFTW_DEFAULT** (prior to inclusion of `unsupported/Eigen/FFT`) and linking with the FFTW library (`-lfftw3`). This setup precedure may be handled automatically by a build system like CMake (see `set_eigen_fft_backend` macro on ↪ GITLAB).

Example 4.3.12 (Efficiency of `fft` for different backend implementations)



Platform:

- Linux (Ubuntu 16.04 64bit)
- Intel(R) Core(TM) i7-4600U CPU @ 2.10GHz
- L2 256KB, L3 4 MB, 8 GB DDR3 @ 1.60GHz
- Clang 3.8.0, -O3

For reasonably high input sizes the FFTW backend gives, compared to EIGEN's default backend (Kiss FFT), a speedup of 2-4x.

4.4 Trigonometric transformations

Supplementary reading. [**?**, Sect. 55], see also [**?**] for an excellent presentation of various variants of the cosine transform.

Keeping in mind $\exp(2\pi ix) = \cos(2\pi x) + \imath \sin(2\pi x)$ we may also consider the real/imaginary parts of the Fourier basis vectors $(\mathbf{F}_n)_{:,j}$ as bases of $\mathbb{R}^n$ and define the corresponding basis transformation. They can all be realized by means of `fft` with an asymptotic computational effort of $O(n \log n)$. These transformations avoid the use of complex numbers.

Details are given in the sequel.

4.4.1 Sine transform

Another trigonometric basis transform in $\mathbb{R}^{n-1}$, $n \in \mathbb{N}$:

Standard basis of $\mathbb{R}^{n-1}$ "Sine basis"

$$\left\{ \begin{bmatrix}1\\0\\\vdots\\0\end{bmatrix} \begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix} \cdots \begin{bmatrix}0\\\vdots\\0\\1\\0\end{bmatrix} \begin{bmatrix}0\\\vdots\\0\\1\end{bmatrix} \right\} \longleftarrow \left\{ \begin{bmatrix}\sin(\frac{\pi}{n})\\\sin(\frac{2\pi}{n})\\\vdots\\\sin(\frac{(n-1)\pi}{n})\end{bmatrix} \begin{bmatrix}\sin(\frac{2\pi}{n})\\\sin(\frac{4\pi}{n})\\\vdots\\\sin(\frac{2(n-1)\pi}{n})\end{bmatrix} \cdots \begin{bmatrix}\sin(\frac{(n-1)\pi}{n})\\\sin(\frac{2(n-1)\pi}{n})\\\vdots\\\sin(\frac{(n-1)^2\pi}{n})\end{bmatrix} \right\}$$

Basis transform matrix (sine basis $\to$ standard basis): $\mathbf{S}_n := (\sin(jk\pi/n))_{j,k=1}^{n-1} \in \mathbb{R}^{n-1,n-1}$.

Lemma 4.4.1. Properties of the sine matrix

$\sqrt{2/n}\,\mathbf{S}_n \in \mathbb{R}^{n,n}$ *is real, symmetric and orthogonal (→ Def. 6.2.2)*

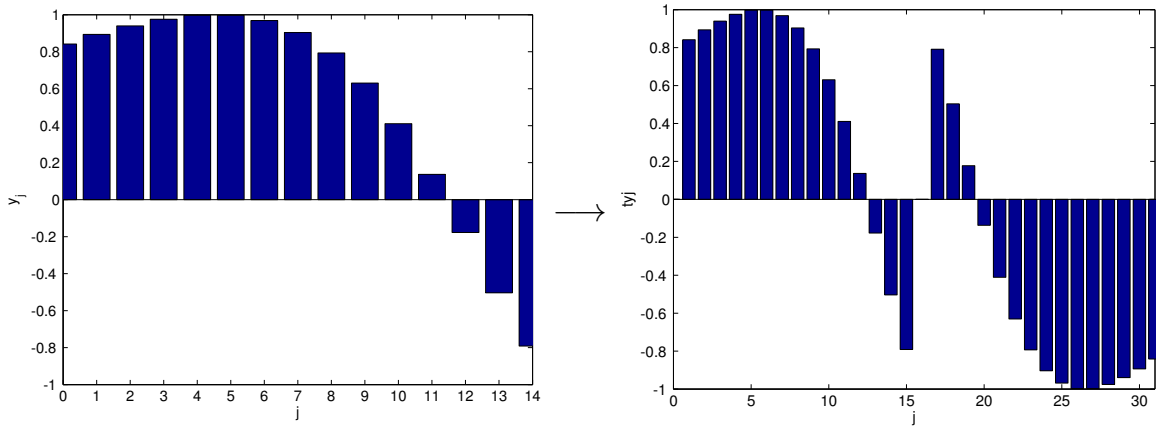
Sine transform of $\mathbf{y} = [y_1, \ldots, y_{n-1}]^\top \in \mathbb{R}^{n-1}$:

$$s_k = \sum_{j=1}^{n-1} y_j \sin(\pi jk/n) \quad , \quad k = 1, \ldots, n-1 \, . \tag{4.4.2}$$

By elementary consideration we can devise a DFT-based algorithm for the sine transform ($\hat{=}\ \mathbf{S}_n \times$vector):

tool: "wrap around": $\widetilde{\mathbf{y}} \in \mathbb{R}^{2n}$: $\widetilde{y}_j = \begin{cases} y_j & \text{, if } j = 1, \ldots, n-1 \, , \\ 0 & \text{, if } j = 0, n \, , \\ -y_{2n-j} & \text{, if } j = n+1, \ldots, 2n-1 \, . \end{cases}$ ($\widetilde{\mathbf{y}}$ "odd")

This "wrap around" transformation can be visualized as follows:



Next we use $\sin(x) = \frac{1}{2\imath}(\exp(\imath x) - \exp(-\imath u x)$ to identify the DFT of a wrapped around vector as a sine transform:

$$(\mathbf{F}_{2n}\widetilde{\mathbf{y}})_k \overset{(4.2.19)}{=} \sum_{j=1}^{2n-1} \widetilde{y}_j e^{-\frac{2\pi\imath}{2n}kj} = \sum_{j=1}^{n-1} y_j e^{-\frac{\pi\imath}{n}kj} - \sum_{j=n+1}^{2n-1} y_{2n-j} e^{-\frac{\pi\imath}{n}kj}$$

$$= \sum_{j=1}^{n-1} y_j (e^{-\frac{\pi\imath}{n}kj} - e^{\frac{\pi\imath}{n}kj}) = -2i\,(\mathbf{S}_n \mathbf{y})_k \quad , k = 1, \ldots, n-1 \, .$$

C++11 code 4.4.3: Wrap-around implementation of sine transform ⇝ GITLAB

```
// Simple sine transform of y ∈ R^n-1 into c ∈ R^n-1 by (4.4.2)
void sinetrfwrap(const VectorXd &y, VectorXd& c)
{
  VectorXd::Index n = y.size()+1;
  // Create wrapped vector ỹ
  VectorXd yt(2*n); yt << 0,y,0,-y.reverse();

```

```
  Eigen::VectorXcd ct;
  Eigen::FFT<double> fft; // DFT helper class
  fft.SetFlag(Eigen::FFT<double>::Flag::Unscaled);
  fft.fwd(ct,yt);

  const std::complex<double> v(0,2); // factor 2ı
  c = (-ct.middleRows(1,n-1)/v).real();
}
```

Remark 4.4.4 (Sine transform via DFT of half length)

The simple Code 4.4.3 relies on a DFT for vectors of length $2n$, which may be a waste of computational resources in some applications. A DFT of length n is sufficient as demonstrated by the following manipulations.

Step ①: transform of the coefficients

$$\widetilde{y}_j = \sin(j\pi/n)(y_j + y_{n-j}) + \tfrac{1}{2}(y_j - y_{n-j}) \,, \quad j = 1,\ldots,n-1 \quad , \quad \widetilde{y}_0 = 0 \,.$$

Step ②: real DFT (→ Section 4.2.3) of $(\widetilde{y}_0,\ldots,\widetilde{y}_{n-1}) \in \mathbb{R}^n$: $\quad c_k := \sum_{j=0}^{n-1} \widetilde{y}_j \, e^{-\frac{2\pi i}{n}jk}$

$$\begin{aligned}
\text{Hence} \quad \mathrm{Re}\{c_k\} &= \sum_{j=0}^{n-1} \widetilde{y}_j \cos(-\tfrac{2\pi i}{n}jk) = \sum_{j=1}^{n-1} (y_j + y_{n-j}) \sin(\tfrac{\pi j}{n}) \cos(\tfrac{2\pi i}{n}jk) \\
&= \sum_{j=0}^{n-1} 2y_j \sin(\tfrac{\pi j}{n}) \cos(\tfrac{2\pi i}{n}jk) = \sum_{j=0}^{n-1} y_j \left(\sin(\frac{2k+1}{n}\pi j) - \sin(\frac{2k-1}{n}\pi j) \right) \\
&= s_{2k+1} - s_{2k-1} \,. \\
\mathrm{Im}\{c_k\} &= \sum_{j=0}^{n-1} \widetilde{y}_j \sin(-\tfrac{2\pi i}{n}jk) = -\sum_{j=1}^{n-1} \tfrac{1}{2}(y_j - y_{n-j}) \sin(\tfrac{2\pi i}{n}jk) = -\sum_{j=1}^{n-1} y_j \sin(\tfrac{2\pi i}{n}jk) \\
&= -s_{2k} \,.
\end{aligned}$$

Step ③: extraction of s_k

s_{2k+1}, $k = 0,\ldots,\frac{n}{2}-1$ ➤ from recursion $s_{2k+1} - s_{2k-1} = \mathrm{Re}\{c_k\}$, $s_1 = \sum_{j=1}^{n-1} y_j \sin(\pi j/n)$,

s_{2k}, $k = 1,\ldots,\frac{n}{2}-2$ ➤ $s_{2k} = -\mathrm{Im}\{c_k\}$.

Implementation (via a `fft` of length $n/2$):

C++11-code 4.4.5: Sine transform ⇢ GITLAB

```
void sinetransform(const Eigen::VectorXd &y, Eigen::VectorXd& s)
{
      int n = y.rows() + 1;
      std::complex<double> i(0,1);

```

```
7       // Prepare sine terms
8       Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(n-1, 1, n-1);
9       Eigen::VectorXd sinevals = x.unaryExpr([&](double z){ return
            imag(std::pow(std::exp(i*M_PI/(double)n), z)); });
10
11      // Transform coefficients
12      Eigen::VectorXd yt(n);
13      yt(0) = 0;
14      yt.tail(n-1) = sinevals.array() * (y + y.reverse()).array() +
            0.5*(y-y.reverse()).array();
15
16      // FFT
17      Eigen::VectorXcd c;
18      Eigen::FFT<double> fft;
19      fft.fwd(c,yt);
20
21      s.resize(n);
22      s(0) = sinevals.dot(y);
23
24      for (int k=2; k<=n-1; ++k)
25      {
26          int j = k-1; // Shift index to consider indices starting from
                0
27          if (k%2==0)
28              s(j) = -c(k/2).imag();
29          else
30              s(j) = s(j-2) + c((k-1)/2).real();
31      }
32  }
```

Example 4.4.6 (Diagonalization of local translation invariant linear grid operators)

We consider a so-called 5-points-stencil-operator on $\mathbb{R}^{n,n}$, $n \in \mathbb{N}$, defined as follows

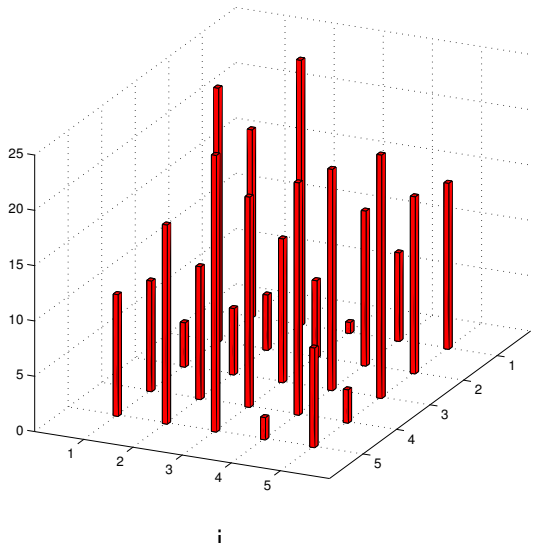
$$\mathsf{T} : \begin{cases} \mathbb{R}^{n,n} & \to & \mathbb{R}^{n,n} \\ \mathbf{X} & \mapsto & \mathsf{T}(\mathbf{X}) \end{cases}, \quad (\mathsf{T}(\mathbf{X}))_{ij} := cx_{ij} + c_y x_{i,j+1} + c_y x_{i,j-1} + c_x x_{i+1,j} + c_x x_{i-1,j} \tag{4.4.7}$$

with coefficients $c, c_y, c_x \in \mathbb{R}$, convention: $x_{ij} := 0$ for $(i,j) \notin \{1,\ldots,n\}^2$.

A matrix can be regarded as a function that assigns values (= matrix entries) to the points of a 2D lattice/-grid:

$$\text{Matrix } \mathbf{X} \in \mathbb{R}^{n,n}$$
$$\updownarrow$$
$$\text{grid function} \in \{1,\ldots,n\}^2 \mapsto \mathbb{R}$$

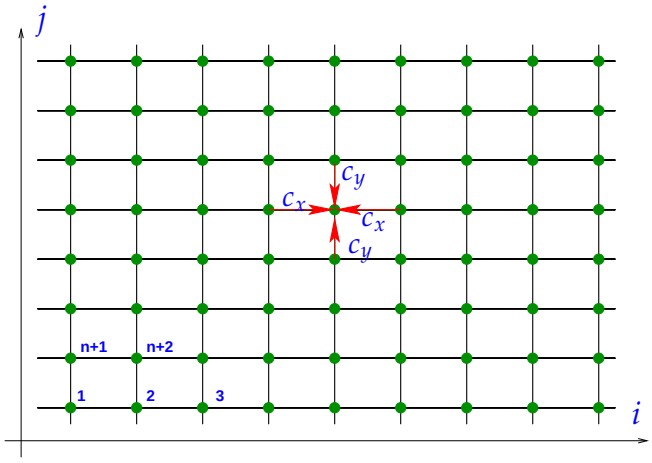
Visualization of a grid function ▷



Identification $\mathbb{R}^{n,n} \cong \mathbb{R}^{n^2}$, $x_{ij} \sim \widetilde{x}_{(j-1)n+i}$ (row-wise numbering) gives a matrix representation $\mathbf{T} \in \mathbb{R}^{n^2,n^2}$ of T:

$$\mathbf{T} = \begin{bmatrix} \mathbf{C} & c_y\mathbf{I} & 0 & \cdots & \cdots & 0 \\ c_y\mathbf{I} & \mathbf{C} & c_y\mathbf{I} & & & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & c_y\mathbf{I} & \mathbf{C} & c_y\mathbf{I} \\ 0 & \cdots & \cdots & 0 & c_y\mathbf{I} & \mathbf{C} \end{bmatrix} \in \mathbb{R}^{n^2,n^2},$$

$$\mathbf{C} = \begin{bmatrix} c & c_x & 0 & \cdots & \cdots & 0 \\ c_x & c & c_x & & & \vdots \\ 0 & \ddots & \ddots & \ddots & & \\ \vdots & & & c_x & c & c_x \\ 0 & \cdots & \cdots & 0 & c_x & c \end{bmatrix} \in \mathbb{R}^{n,n}.$$

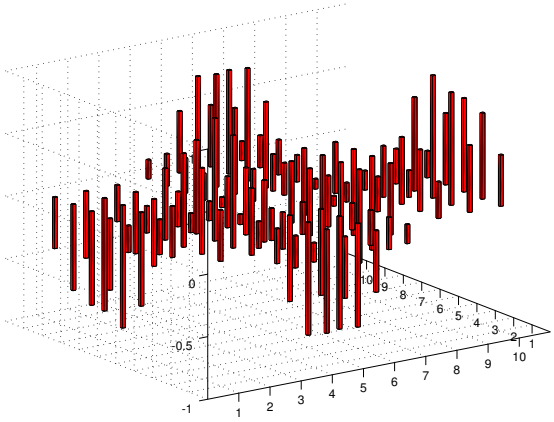


We already know the sine basis of $\mathbb{R}^{n,n}$:

$$\mathbf{B}^{kl} = \left(\sin\left(\tfrac{\pi}{n+1}ki\right)\sin\left(\tfrac{\pi}{n+1}lj\right)\right)_{i,j=1}^{n}. \qquad (4.4.8)$$

These matrices will also provide a basis of the vector space of grid functions $\{1,\ldots,n\}^2 \mapsto \mathbb{R}$.

$n = 10$: grid function $\mathbf{B}^{2,3}$ ➢



The key observation is that elements of the sine basis are eigenvectors of T:

$$\begin{aligned}(T(\mathbf{B}^{kl}))_{ij} &= c\sin\left(\tfrac{\pi}{n+1}ki\right)\sin\left(\tfrac{\pi}{n+1}lj\right) + c_y\sin\left(\tfrac{\pi}{n+1}ki\right)\left(\sin\left(\tfrac{\pi}{n+1}l(j-1)\right)+\sin\left(\tfrac{\pi}{n+1}l(j+1)\right)\right)+ \\ &\qquad c_x\sin\left(\tfrac{\pi}{n+1}lj\right)\left(\sin\left(\tfrac{\pi}{n+1}k(i-1)\right)+\sin\left(\tfrac{\pi}{n+1}k(i+1)\right)\right) \\ &= \sin\left(\tfrac{\pi}{n+1}ki\right)\sin\left(\tfrac{\pi}{n+1}lj\right)\left(c+2c_y\cos\left(\tfrac{\pi}{n+1}l\right)+2c_x\cos\left(\tfrac{\pi}{n+1}k\right)\right)\end{aligned}$$

Hence $\mathbf{B}^{kl}$ is eigenvector of $\mathbf{T}$ (or $\mathbf{T}$ after row-wise numbering) and the corresponding eigenvalue is given by $c + 2c_y \cos(\frac{\pi}{n+1}l) + 2c_x \cos(\frac{\pi}{n+1}k)$. Recall very similar considerations for discrete (periodic) convolutions in 1D (→ § 4.2.6) and 2D (→ § 4.2.51)

The basis transform can be implemented efficiently based on the 1D sine transform:

$$\mathbf{X} = \sum_{k=1}^{n}\sum_{l=1}^{n} y_{kl}\mathbf{B}^{kl} \quad \Rightarrow \quad x_{ij} = \sum_{k=1}^{n} \sin(\tfrac{\pi}{n+1}ki) \sum_{l=1}^{n} y_{kl} \sin(\tfrac{\pi}{n+1}lj) \; .$$

Hence nested sine transforms (→ Section 4.2.4) for rows/columns of $\mathbf{Y} = (y_{kl})_{k,l=1}^{n}$.

Here: implementation of sine transform (4.4.2) with "wrapping"-technique.

C++11-code 4.4.9: 2D sine transform ⇝ GITLAB

```
2  void sinetransform2d(const Eigen::MatrixXd& Y, Eigen::MatrixXd& S)
3  {
4      int m = Y.rows();
5      int n = Y.cols();
6
7      Eigen::VectorXcd c;
8      Eigen::FFT<double> fft;
9      std::complex<double> i(0,1);
10
11     Eigen::MatrixXcd C(2*m+2,n);
12     C.row(0) = Eigen::VectorXcd::Zero(n);
13     C.middleRows(1, m) = Y.cast<std::complex<double>>();
14     C.row(m+1) = Eigen::VectorXcd::Zero(n);
15     C.middleRows(m+2, m) =
           -Y.colwise().reverse().cast<std::complex<double>>();
16
17     // FFT on each column of C - Eigen::fft only operates on vectors
18     for (int i=0; i<n; ++i)
19     {
20         fft.fwd(c,C.col(i));
21         C.col(i) = c;
22     }
23
24     C.middleRows(1,m) = i*C.middleRows(1,m)/2.;
25
26     Eigen::MatrixXcd C2(2*n+2,m);
27     C2.row(0) = Eigen::VectorXcd::Zero(m);
28     C2.middleRows(1,n) = C.middleRows(1,m).transpose();
29     C2.row(n+1) = Eigen::VectorXcd::Zero(m);
30     C2.middleRows(n+2, n) =
           -C.middleRows(1,m).transpose().colwise().reverse();
31
32     // FFT on each column of C2 - Eigen::fft only operates on vectors
33     for (int i=0; i<m; ++i)
34     {
35         fft.fwd(c,C2.col(i));
36         C2.col(i) = c;
37     }
```

```
38
39      S = (i*C2.middleRows(1,n).transpose() / 2.).real();
40  }
```

C++11-code 4.4.10: FFT-based solution of local translation invariant linear operators
➷ GITLAB

```
2  void fftbasedsolutionlocal(const Eigen::MatrixXd& B,
3   double c, double cx, double cy, Eigen::MatrixXd& X)
4  {
5       size_t m = B.rows();
6       size_t n = B.cols();
7
8       // Eigen's meshgrid
9       Eigen::MatrixXd I =
            Eigen::RowVectorXd::LinSpaced(n,1,n).replicate(m,1);
10      Eigen::MatrixXd J =
            Eigen::VectorXd::LinSpaced(m,1,m).replicate(1,n);
11
12      // FFT
13      Eigen::MatrixXd X_;
14      sinetransform2d(B, X_);
15
16      // Translation
17      Eigen::MatrixXd T;
18      T = c + 2*cx*(M_PI/(n+1)*I).array().cos() +
19                2*cy*(M_PI/(m+1)*J).array().cos();
20      X_ = X_.cwiseQuotient(T);
21
22      sinetransform2d(X_, X);
23      X = 4*X/((m+1)*(n+1));
24  }
```

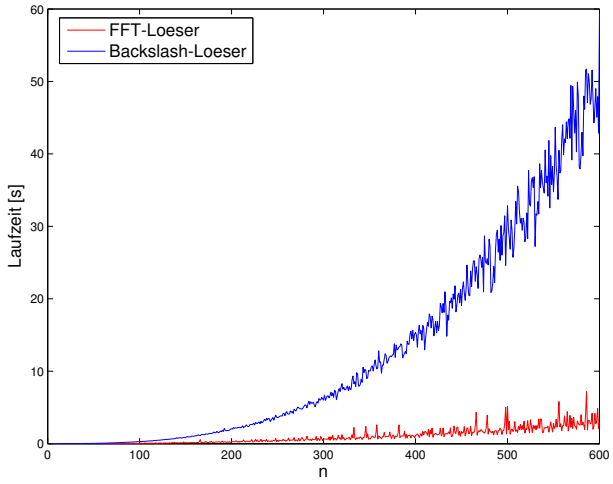
Thus the diagonalization of $\mathbf{T}$ via 2D sine transformyields an efficient algorithm for solving linear system of equations $\mathsf{T}(\mathbf{X}) = \mathbf{B}$: computational cost $O(n^2 \log n)$.

Experiment 4.4.11 (Efficiency of FFT-based solver)

In the experiment we test the gain in runtime obtained by using DFT-based algorithms for solving linear systems of equations with coefficient matrix $\mathbf{T}$ induced by the operator T from (4.4.7)

`tic-toc`-timing (MATLAB V7, Linux, Intel Pentium 4 Mobile CPU 1.80GHz)

```
A = gallery('poisson',n);
B = magic(n);
b = reshape(B,n*n,1);
tic;
C = fftsolve(B,4,-1,-1);
t1 = toc;
tic; x = A\b; t2 = toc;
```



4.4.2 Cosine transform

Another trigonometric basis transform in $\mathbb{R}^n$, $n \in \mathbb{N}$:

standard basis of $\mathbb{R}^n$ "cosine basis"

$$\left\{ \begin{bmatrix}1\\0\\\vdots\\0\end{bmatrix} \begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix} \cdots \begin{bmatrix}0\\\vdots\\0\\1\\0\end{bmatrix} \begin{bmatrix}0\\\vdots\\\\0\\1\end{bmatrix} \right\} \leftarrow \left\{ \begin{bmatrix}2^{-1/2}\\\cos(\frac{\pi}{2n})\\\cos(\frac{2\pi}{2n})\\\vdots\\\\\vdots\\\cos(\frac{(n-1)\pi}{2n})\end{bmatrix} \begin{bmatrix}2^{-1/2}\\\cos(\frac{3\pi}{2n})\\\cos(\frac{6\pi}{2n})\\\vdots\\\\\vdots\\\cos(\frac{3(n-1)\pi}{2n})\end{bmatrix} \cdots \begin{bmatrix}2^{-1/2}\\\cos(\frac{(2n-1)\pi}{2n})\\\cos(\frac{2(2n-1)\pi}{2n})\\\vdots\\\\\vdots\\\cos(\frac{(n-1)(2n-1)\pi}{2n})\end{bmatrix} \right\} .$$

Basis transform matrix (cosine basis $\to$ standard basis):

$$\mathbf{C}_n = \left[c_{ij}\right]_{i,j=0}^{n-1} \in \mathbb{R}^{n,n} \quad \text{with} \quad c_{ij} = \begin{cases} 2^{-1/2} & \text{, if } i = 0 \text{ ,} \\ \cos(i\frac{2j+1}{2n}\pi) & \text{, if } i > 0 \text{ .} \end{cases}$$

Lemma 4.4.12. Properties of cosine matrix

The matrix $\sqrt{2/n}\,\mathbf{C}_n \in \mathbb{R}^{n,n}$ is real and orthogonal ($\to$ Def. 6.2.2) .

Note: $\mathbf{C}_n$ is not symmetric.

cosine transform of $\mathbf{y} = [y_0, \ldots, y_{n-1}]^\top$:

$$c_k = \sum_{j=0}^{n-1} y_j \cos(k\tfrac{2j+1}{2n}\pi) \quad , \quad k = 1, \ldots, n-1 \text{ ,} \tag{4.4.13}$$

$$c_0 = \frac{1}{\sqrt{2}} \sum_{j=0}^{n-1} y_j \text{ .}$$

Implementation of $\mathbf{Cy}$ using the "wrapping"-technique as in Code 4.4.3:

C++11-code 4.4.14: Cosine transform ↪ GITLAB

```
void cosinetransform(const Eigen::VectorXd& y, Eigen::VectorXd& c)
{
    int n = y.size();

    Eigen::VectorXd y_(2*n);
    y_.head(n) = y;
    y_.tail(n) = y.reverse();

    // FFT
    Eigen::VectorXcd z;
    Eigen::FFT<double> fft;
    fft.fwd(z,y_);

    std::complex<double> i(0,1);
    c.resize(n);
    c(0) = z(0).real()/(2*sqrt(2));
    for(size_t j=1; j<n; ++j) {
        c(j) = (0.5 * pow(exp(-i*M_PI/(2*(double)n)), j) *
            z(j)).real();
    }
}
```

Implementation of $\mathbf{C}_n^{-1}\mathbf{y}$ ("Wrapping"-technique):

C++11-code 4.4.15: Inverse cosine transform ↪ GITLAB

```
void icosinetransform(const Eigen::VectorXd& c, Eigen::VectorXd& y)
{
    size_t n = c.size();

    std::complex<double> i(0,1);
    Eigen::VectorXcd c_1(n);
    c_1(0) = sqrt(2)*c(0);
    for(size_t j=1; j<n; ++j) {
        c_1(j) = pow(exp(-i*M_PI/(2*(double)n)), j) * c(j);
    }

    Eigen::VectorXcd c_2(2*n);
    c_2.head(n) = c_1;
    c_2(n) = 0;
    c_2.tail(n-1) = c_1.tail(n-1).reverse().conjugate();

    // FFT
    Eigen::VectorXd z;
    Eigen::FFT<double> fft;
    fft.inv(z,c_2);

    // To obtain the same result of Matlab,
```

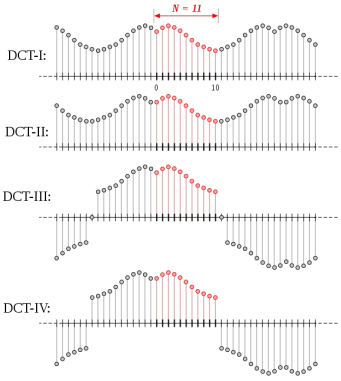
```
24        // shift the inverse FFT result by 1.
25        Eigen::VectorXd y_(2*n);
26        y_.head(2*n-1) = z.tail(2*n-1);
27        y_(2*n-1) = z(0);
28
29        y = 2*y_.head(n);
30    }
```

Remark 4.4.16 (Cosine transforms for compression)

The cosine transforms discussed above are named DCT-II and DCT-III.
Various cosine transforms arise by imposing various boundary conditions:

- DCT-II: even around $-1/2$ and $N-1/2$
- DCT-III: even around 0 and odd around N

DCT-II is used in JPEG-compression while a slightly modified DCT-IV makes the main component of MP3, AAC and WMA formats.



4.5 Toeplitz Matrix Techniques

Example 4.5.1 (Parameter identification for linear time-invariant filters)

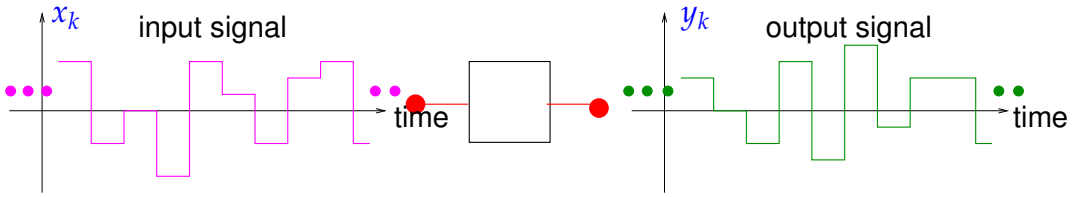
- $(x_k)_{k\in\mathbb{Z}}$ m-periodic discrete signal = *known* input
- $(y_k)_{k\in\mathbb{Z}}$ m-periodic *measured*$^{(*)}$ output signal of a linear time-invariant filter, see § 4.1.1.
  $(*)$ ➔ inevitably affected by measurement errors!
- Sought: Estimate for the impulse response ($\rightarrow$ Def. 4.1.3) of the filter

This task reminds us of the parameter estimation problem from Ex. 3.0.5, which we tackled with least squares techniques. We employ similar ideas for the current problem

- Known: impulse response of filter has maximal duration $n\Delta t$, $n \in \mathbb{N}$, $n \le m$

cf. (4.1.14)

$$\exists \mathbf{h} = (h_0,\ldots,h_{n-1})^\top \in \mathbb{R}^n, \quad n \le m: \quad y_k = \sum_{j=0}^{n-1} h_j x_{k-j}\,. \tag{4.5.2}$$



If the y_k were exact, we could retrieve $h_0,\ldots,h_{n-1}$ by examining only $y_0,\ldots,y_{n-1}$ and inverting the discrete periodic convolution ($\rightarrow$ Def. 4.1.33) using (4.2.17).

However, in case the y_k are affected by measurements errors it is advisable to use all available y_k for a least squares estimate of the impulse response.

We can now formulate the least squares parameter identification problem: seek $\mathbf{h} = (h_0, \ldots, h_{n-1})^\top \in \mathbb{R}^n$ with

$$\|\mathbf{A}\mathbf{h} - \mathbf{y}\|_2 = \left\| \begin{bmatrix} x_0 & x_{-1} & \cdots & & \cdots & x_{1-n} \\ x_1 & x_0 & x_{-1} & & & \vdots \\ \vdots & x_1 & x_0 & \ddots & & \\ & & \ddots & \ddots & & \vdots \\ \vdots & & & \ddots & \ddots & x_{-1} \\ x_{n-1} & & & & x_1 & x_0 \\ x_n & x_{n-1} & & & & x_1 \\ \vdots & & & & & \vdots \\ x_{m-1} & \cdots & & & \cdots & x_{m-n} \end{bmatrix} \begin{bmatrix} h_0 \\ \vdots \\ \\ \\ \vdots \\ h_{n-1} \end{bmatrix} - \begin{bmatrix} y_0 \\ \vdots \\ \\ \\ \vdots \\ y_{m-1} \end{bmatrix} \right\|_2 \to \min \ .$$

➢ Linear least squares problem, → Chapter 3 with a coefficient matrix $\mathbf{A}$ that enjoys the property that $(\mathbf{A})_{ij} = x_{i-j}$ (constant entries of diagonals).

The coefficient matrix for the normal equations (→ Section 3.1.2, Thm. 3.1.10) corresponding to the above linear least squares problem is

$$\mathbf{M} := \mathbf{A}^{\mathrm{H}}\mathbf{A} \quad , \quad (\mathbf{M})_{ij} = \sum_{k=1}^{m} x_{k-i}x_{k-j} = z_{i-j} \quad \text{due to periodicity of } (x_k)_{k\in\mathbb{Z}} \ .$$

➢ Again, $\mathbf{M} \in \mathbb{R}^{n,n}$ is a *matrix with constant diagonals* & s.p.d.
("constant diagonals" $\Leftrightarrow$ $(\mathbf{M})_{i,j}$ depends only on $i-j$)

Example 4.5.3 (Linear regression for stationary Markov chains)

We consider a sequence of scalar random variables: $(\mathsf{Y}_k)_{k\in\mathbb{Z}}$, a so-called Markov chain. These can be thought of as values for a random quantity sampled at equidistant points in time.

Assume: stationary (time-independent) correlation, that is, with $(\mathcal{A}, \Omega, dP)$ denoting the underlying probability space,

$$\mathcal{E}(\mathsf{Y}_{i-j}\mathsf{Y}_{i-k}) = \int_\Omega \mathsf{Y}_{i-j}(\omega)\mathsf{Y}_{i-k}(\omega)\,\mathrm{d}\mathbb{P}(\omega) = u_{k-j} \quad \forall i,j,k \in \mathbb{Z} \ , \quad u_i = u_{-i} \ .$$

Here $\mathcal{E}$ stands for the expectation of a random variable.

Model: We assume a finite linear dependency of the form

$$\exists \mathbf{x} = (x_1, \ldots, x_n)^\top \in \mathbb{R}^n\colon \quad \mathsf{Y}_k = \sum_{j=1}^{n} x_j \mathsf{Y}_{k-j} \quad \forall k \in \mathbb{Z} \ .$$

with *unknown* parameters x_j, $j = 1, \ldots, n$: for fixed $i \in \mathbb{Z}$

$$\text{Estimator} \qquad \mathbf{x} = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} \mathcal{E}\left|\mathsf{Y}_i - \sum_{j=1}^{n} x_j \mathsf{Y}_{i-j}\right|^2 \tag{4.5.4}$$

Next, we use the linearity of the expectation:

$$\blacktriangleright \quad E|Y_i|^2 - 2\sum_{j=1}^{n} x_j u_k + \sum_{k,j=1}^{n} x_k x_j u_{k-j} \to \min \ .$$

$$\blacktriangleright \quad \mathbf{x}^\top \mathbf{A}\mathbf{x} - 2\mathbf{b}^\top \mathbf{x} \to \min \quad \text{with} \quad \mathbf{b} = (u_k)_{k=1}^n \ , \quad \mathbf{A} = (u_{i-j})_{i,j=1}^n \ . \tag{4.5.5}$$

By definition $\mathbf{A}$ is a so-called covariance matrix and, as such, has to be symmetric and positive definite ($\to$ Def. 1.1.8). Also note that

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} - 2\mathbf{b}^\top \mathbf{x} = (\mathbf{x} - \mathbf{x}^*)^\top \mathbf{A}(\mathbf{x} - \mathbf{x}^*) - \mathbf{x}^* \mathbf{A}\mathbf{x}^* \ , \tag{4.5.6}$$

with $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$. Therefore $\mathbf{x}^*$ is the unique minimizer of $\mathbf{x}^\top \mathbf{A}\mathbf{x} - 2\mathbf{b}^\top \mathbf{x}$. The problem is reduced to solving the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ (Yule-Walker-equation, see below).

Note that the covariance matrix $\mathbf{A}$ has constant diagonals, too.

(4.5.7) Matrices with constant diagonals

Matrices with constant diagonals occur frequently in mathematical models, see Ex. 4.5.1, **??**. They generalize circulant matrices ($\to$ Def. 4.1.38).

Note: "Information content" of a matrix $\mathbf{M} \in \mathbb{K}^{m,n}$ with constant diagonals, that is, $(\mathbf{M})_{i,j} = m_{i-j}$, is $m + n - 1$ numbers $\in \mathbb{K}$.

Definition 4.5.8. Toeplitz matrix

$\mathbf{T} = (t_{ij})_{i,j=1}^n \in \mathbb{K}^{m,n}$ is a Toeplitz matrix, if there is a vector $\mathbf{u} = [u_{-m+1}, \ldots, u_{n-1}] \in \mathbb{K}^{m+n-1}$ such that $t_{ij} = u_{j-i}$, $1 \le i \le m$, $1 \le j \le n$.

$$\mathbf{T} = \begin{bmatrix} u_0 & u_1 & \cdots & & \cdots & u_{n-1} \\ u_{-1} & u_0 & u_1 & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & u_1 \\ u_{1-m} & \cdots & & \cdots & u_{-1} & u_0 \end{bmatrix}$$

4.5.1 Toeplitz Matrix Arithmetic

Given: $\mathbf{T} = (u_{j-i}) \in \mathbb{K}^{m,n}$, a Toeplitz matrix with generating vector $\mathbf{u} = [u_{-m+1}, \ldots, u_{n-1}]^\top \in \mathbb{K}^{m+n-1}$, see Def. 4.5.8.

Task: Efficient evaluation of matrix×vector product $\mathbf{T}\mathbf{x}$, $\mathbf{x} \in \mathbb{K}^n$

To motivate the approach we realize that we have already encountered Toeplitz matrices in the convolution of finite signals discussed in Rem. 4.1.17, see (4.1.18). The trick introduced in Rem. 4.1.40 was to extend

the matrix to a circulant matrix by zero padding, compare (4.1.43).

Idea: Extend $\mathbf{T} \in \mathbb{K}^{m,n}$ to a circulant matrix ($\rightarrow$ Def. 4.1.38) $\mathbf{C} \in \mathbb{K}^{m+n,m+n}$ generated by the $m+n$-periodic sequence $(c_j)_{j\in\mathbb{Z}}$ given by

$$c_j = \begin{cases} u_j & \text{for } j = -m+1,\ldots,n-1\,, \\ 0 & \text{for } j = n\,, \end{cases} \quad + \text{ periodic extension.}$$

The upper left $m \times n$ block of $\mathbf{C}$ contains $\mathbf{T}$:

$$(\mathbf{C})_{ij} = c_{i-j}\,, \quad 1 \le i,j \le m+n \quad \Rightarrow \quad (\mathbf{C})_{1:m,1:n} = \mathbf{T}\,. \tag{4.5.9}$$

The following formula demonstrates the structure of $\mathbf{C}$ in the case $m = n$.

$$\mathbf{C} = \left[\begin{array}{ccccccc|ccccccc}
u_0 & u_1 & \cdots & & \cdots & u_{n-1} & & 0 & u_{1-n} & \cdots & & \cdots & u_{-1} & \\
u_{-1} & u_0 & u_1 & & & \vdots & & u_{n-1} & 0 & \ddots & & & \vdots & \\
\vdots & \ddots & \ddots & \ddots & & \vdots & & \vdots & \ddots & \ddots & & & & \\
\vdots & & \ddots & \ddots & \ddots & \vdots & & & & & & \ddots & & \\
\vdots & & & \ddots & \ddots & u_1 & & \vdots & & & \ddots & \ddots & u_{1-n} & \\
u_{1-n} & \cdots & & \cdots & u_{-1} & u_0 & & u_1 & & & & u_{n-1} & 0 & \\
\hline
0 & u_{1-n} & \cdots & & \cdots & u_{-1} & & u_0 & u_1 & \cdots & & \cdots & u_{n-1} & \\
u_{n-1} & 0 & \ddots & & & \vdots & & u_{-1} & u_0 & u_1 & & & \vdots & \\
\vdots & \ddots & \ddots & & & & & \vdots & \ddots & \ddots & \ddots & & \vdots & \\
 & & & & \ddots & & & \vdots & & \ddots & \ddots & \ddots & \vdots & \\
\vdots & & & \ddots & \ddots & u_{1-n} & & \vdots & & & \ddots & \ddots & u_1 & \\
u_1 & & & & u_{n-1} & 0 & & u_{1-n} & \cdots & & \cdots & u_{-1} & u_0 &
\end{array}\right]$$

Recall from 4.3 that the multiplication with a circulant $(m+n)\times(m+n)$-matrix (= discrete periodic convolution $\rightarrow$ Def. 4.1.33) can be carried out by means of DFT/FFT with an asymptotic computational effort of $O(m+n)\log(m+n)$ for $m,n \to \infty$, see Code 4.2.25.

From (4.5.9) it is clear how to implement matrix$\times$vector for the Toeplitz matrix $\mathbf{T}$

$$\text{zero padding} \longrightarrow \quad \mathbf{C}\begin{bmatrix}\mathbf{x}\\ 0\end{bmatrix} = \begin{bmatrix}\mathbf{Tx}\\ *\end{bmatrix}$$

Computational effort for computing $\mathbf{Tx}$: $O((n+m)\log(m+n))$ (FFT based, Section 4.3)
Almost optimal in light of the data complexity $O(m+n)$ of a Toeplitz matrix.

4.5.2 The Levinson Algorithm

Given: Symmetric positive definite (s.p.d.) ($\rightarrow$ Def. 1.1.8) Toeplitz matrix $\mathbf{T} = (u_{j-i})_{i,j=1}^{n} \in \mathbb{K}^{n,n}$ with generating vector $\mathbf{u} = [u_{-n+1},\ldots,u_{n-1}] \in \mathbb{C}^{2n-1}$

Note that the symmetry of a Toeplitz matrix is induced by the property $u_{-k} = u_k$ of its generating vector.

Task: Find an efficient solution algorithm for the LSE $\mathbf{Tx} = \mathbf{b}$, $\mathbf{b} \in \mathbb{C}^n$, the Yule-Walker problem from 4.5.3.

Without loss of generality we assume thet $\mathbf{T}$ has unit diagonal, $u_0 = 1$.

Recursive (inductive) solution strategy:

Define:

- $\mathbf{T}_k := \left[u_{j-i}\right]_{i,j=1}^{k} \in \mathbb{K}^{k,k}$ (left upper block of $\mathbf{T}$) ➢ $\mathbf{T}_k$ is s.p.d. Toeplitz matrix ,
- $\mathbf{x}^k \in \mathbb{K}^k$: $\mathbf{T}_k\mathbf{x}^k = [b_1,\ldots,b_k]^\top \Leftrightarrow \mathbf{x}^k = \mathbf{T}_k^{-1}\mathbf{b}^k$,
- $\mathbf{u}^k := (u_1,\ldots,u_k)^\top$

Thus we can block partition the LSE

$$\mathbf{T}_{k+1}\mathbf{x}^{k+1} = \left[\begin{array}{ccc|c} & \mathbf{T}_k & & \begin{matrix}u_k\\ \vdots\\ u_1\end{matrix} \\ \hline u_k & \cdots & u_1 & 1 \end{array}\right] \left[\begin{array}{c} \widetilde{\mathbf{x}}^{k+1} \\ \hline x_{k+1}^{k+1} \end{array}\right] = \left[\begin{array}{c} b_1 \\ \vdots \\ b_k \\ \hline b_{k+1} \end{array}\right] = \begin{bmatrix} \widetilde{\mathbf{b}}^{k+1} \\ b_{k+1} \end{bmatrix} \tag{4.5.10}$$

Now recall block Gaussian elimination/block-LU decomposition from Rem. 2.3.14, Rem. 2.3.34. They teach us how to eliminate $\widetilde{\mathbf{x}}^{k+1}$ and obtain an expression for x_{k+1}^{k+1}.

To state the formulas concisely, we introduce the reversing permutation:

$$P_k : \{1,\ldots,k\} \mapsto \{1,\ldots,k\} \quad , \quad P_k(i) := k-i+1 \,. \tag{4.5.11}$$

Then we carry out block elimination:

$$\begin{aligned} \widetilde{\mathbf{x}}_{k+1} &= \mathbf{T}_k^{-1}(\widetilde{\mathbf{b}}^{k+1} - x_{k+1}^{k+1}P_k\mathbf{u}^k) = \mathbf{x}^k - x_{k+1}^{k+1}\mathbf{T}_k^{-1}P_k\mathbf{u}^k \,, \\ x_{k+1}^{k+1} &= b_{k+1} - P_k\mathbf{u}^k \cdot \widetilde{\mathbf{x}}^{k+1} = b_{k+1} - P_k \cdot \mathbf{x}^k + x_{k+1}^{k+1}P_k \cdot \mathbf{T}_k^{-1}P_k\mathbf{u}^k \,. \end{aligned} \tag{4.5.12}$$

The recursive idea is clear after introducing the *auxiliary vectors*: $\mathbf{y}^k := \mathbf{T}_k^{-1}P_k\mathbf{u}^k$

$$\mathbf{x}^{k+1} = \begin{bmatrix} \widetilde{\mathbf{x}}^{k+1} \\ x_{k+1}^{k+1} \end{bmatrix} \quad \text{with} \quad \begin{aligned} x_{k+1}^{k+1} &= (b_{k+1} - P_k\mathbf{u}^k)/\sigma_k \\ \widetilde{\mathbf{x}}^{k+1} &= \mathbf{x}^k - x_{k+1}^{k+1}\mathbf{y}^k \end{aligned} \quad , \quad \sigma_k := 1 - P_k\mathbf{u}^k \cdot \mathbf{y}^k \,. \tag{4.5.13}$$

Below: Levinson algorithm for the solution of the Yule-Walker problem $\mathbf{T}\mathbf{x} = \mathbf{b}$ with an s.p.d. Toeplitz matrix described by its generating vector $\mathbf{u}$ (recursive, u_{n+1} not used!)

Linear recursion: Computational cost $\sim (n-k)$ on level k, $k = 0,\ldots,n-1$

➢ Asymptotic complexity $O(n^2)$

C++11 code 4.5.14: Levinson algorithm ↬ GITLAB

```
2  void levinson(const Eigen::VectorXd& u, const Eigen::VectorXd& b,
3  Eigen::VectorXd& x, Eigen::VectorXd& y)
4  {
5      size_t k = u.size() - 1;
6      if(k == 0) {
7          x.resize(1); y.resize(1);
8          x(0) = b(0); y(0) = u(0);
9          return;
10     }
11
12     Eigen::VectorXd xk, yk;
13     levinson(u.head(k), b.head(k), xk, yk);
```

```
14
15      double sigma = 1 - u.head(k).dot(yk);
16
17      double t = (b(k) - u.head(k).reverse().dot(xk))/sigma;
18      x = xk - t*yk.head(k).reverse();
19      x.conservativeResize(x.size()+1);
20      x(x.size()-1) = t;
21
22      double s = (u(k) - u.head(k).reverse().dot(yk))/sigma;
23      y = yk - s*yk.head(k).reverse();
24      y.conservativeResize(y.size()+1);
25      y(y.size()-1)= s;
26  }
```

Remark 4.5.15 (Fast Toeplitz solvers)

FFT-based algorithms for solving $\mathbf{Tx} = \mathbf{b}$ with asymptotic complexity $O(n \log^3 n)$ [**?**] **!**

Supplementary reading. [**?**, Sect. 8.5]: Very detailed and elementary presentation, but the discrete Fourier transform through trigonometric interpolation, which is not covered in this chapter. Hardly addresses discrete convolution.

[**?**, Ch. IX] presents the topic from a mathematical point of few stressing approximation and trigonometric interpolation. Good reference for algorithms for circulant and Toeplitz matrices.

[**?**, Ch. 10] also discusses the discrete Fourier transform with emphasis on interpolation and (least squares) approximation. The presentation of signal processing differs from that of the course.

There is a vast number of books and survey papers dedicated to discrete Fourier transforms, see, for instance, [**?**, **?**]. Issues and technical details way beyond the scope of the course are discussed in these monographs.

Chapter 5

Data Interpolation and Data Fitting in 1D

Contents

5.1 Abstract interpolation

The task of (one-dimensional, scalar) **data interpolation** (point interpolation) can be described as follows:

Given: data points (t_i, y_i), $i = 0, \ldots, n$, $n \in \mathbb{N}$, $t_i \in I \subset \mathbb{R}$, $y_i \in \mathbb{R}$

Objective: *reconstruction of a (continuous) function* $f : I \mapsto \mathbb{R}$ satisfying the $n+1$ interpolation conditions

$$f(t_i) = y_i, \quad i = 0, \ldots, n.$$

The function f we find is called the interpolant of the given data set $\{t_i, y_i)\}_{i=0}^{n}$.

Minimal requirement on data: t_i pairwise distinct: $t_i \neq t_i$, if $i \neq j$, $i, j \in \{0, \ldots, n\}$.

Parlance: The numbers $t_i \in \mathbb{R}$ are called nodes, the $y_i \in \mathbb{R}$ (data) values.

For ease of presentation we will usually assume that the nodes are ordered: $t_0 < t_1 < \cdots < t_n$ and $[t_0, t_n] \subset I$. However, algorithms often must not take for granted sorted nodes.

Remark 5.1.1 (Interpolation of vector-valued data)

A natural generalization is data interpolation with vector-valued data values, seeking a function $\mathbf{f} : I \to \mathbb{R}^d$, $d \in \mathbb{N}$, such that, for given data points $(t_i, \mathbf{y}_i)$, $t_i \in I$ mutually different, $\mathbf{y}_i \in \mathbb{R}^d$, it satisfies the interpolation conditions $\mathbf{f}(t_i) = \mathbf{y}_i$, $i = 0, \ldots, n$.

In this case all methods available for scalar data can be applied component-wise.

An important application is curve reconstruction, that is the interpolation of points $\mathbf{y}_0, \ldots, Vy_n \in \mathbb{R}^2$ in the plane.

A particular aspect of this problem is that the nodes t_i also have to be found, usually from the location of the $\mathbf{y}_i$ in a preprocessing step.

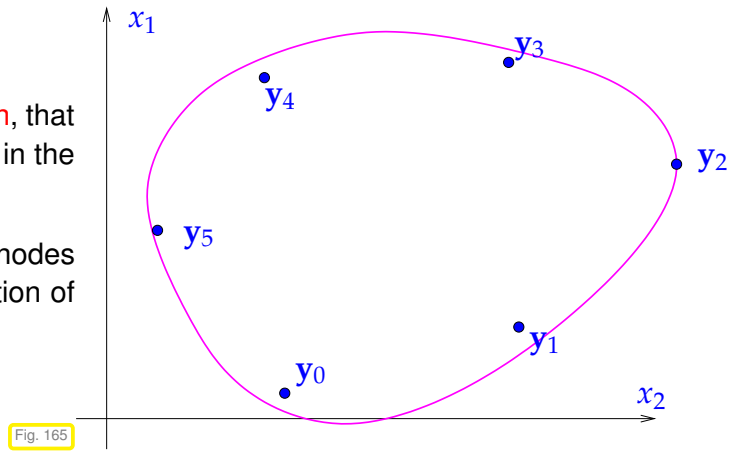


Fig. 165

Remark 5.1.2 (Multi-dimensional data interpolation)

In many applications (computer graphics, computer vision, numerical method for partial differential equations, remote sensing, geodesy, etc.) one has to reconstruct functions of several variables.

This leads to task of multi-dimensional data interpolation:

Given: data points $(\mathbf{x}_i, \mathbf{y}_i)$, $i = 0, \ldots, n$, $n \in \mathbb{N}$, $\mathbf{x}_i \in D \subset \mathbb{R}^m$, $m > 1$, $\mathbf{y}_i \in \mathbb{R}^d$

Objective: *reconstruction of a (continuous) function* $\mathbf{f} : D \mapsto \mathbb{R}^d$ satisfying the $n+1$ interpolation conditions $\mathbf{f}(\mathbf{x}_i) = \mathbf{y}_i$, $i = 0, \ldots, n$.

Significant additional challenges arise in a genuine multidimensional setting. A treatment is beyond the scope of this course. However, the one-dimensional techniques presented in this chapter are relevant even for multi-dimensional data interpolation, if the points $\mathbf{x}_i \in \mathbb{R}^m$ are points of a finite lattice also called tensor product grid.

For instance, for $m = 2$ this is the case, if

$$\{\mathbf{x}_i\}_i = \left\{ [t_k, s_l]^\top \in \mathbb{R}^2 : k \in \{0, \ldots, K\},\ l \in \{0, \ldots, L\} \right\}, \tag{5.1.3}$$

where $t_k \in \mathbb{R}$, $k = 0, \ldots, K$, and s_l, $l = 0, \ldots, L$, $K, L \in \mathbb{N}$, are pairwise distinct nodes.

(5.1.4) Interpolation schemes

When we talk about "interpolation schemes" in 1D, we mean a mapping

$$\mathrm{I} : \begin{cases} \mathbb{R}^{n+1} \times \mathbb{R}^{n+1} & \to & \{f : I \to \mathbb{R}\} \\ \left([t_i]_{i=0}^n, [y_i]_{i=0}^n\right) & \mapsto & \text{interpolant} \end{cases} .$$

Once the function space to which the interpolant belongs is specified, then an interpolation scheme defines an "interpolation problem" in the sense of § 1.5.67. Sometimes, only the data values y_i are consider input data, whereas the dependence of the interpolant on the nodes t_i is suppressed, see Section 5.2.4.

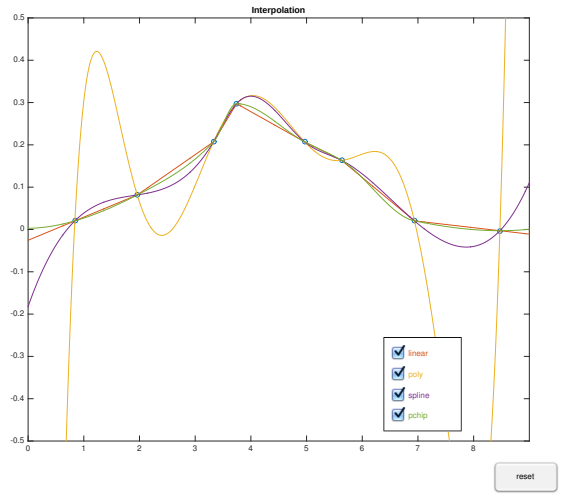


Fig. 166

◁ There are infinitely many ways to fix an interpolant for given data points.

Interpolants can have vastly different properties.

This chapter we will discuss a few widely used method to build interpolants and their different properties will become apparent.

We may (have to!) impose additional requirements on the interpolant:

- minimal smoothness of $\mathbf{f}$, e.g. $\mathbf{f} \in C^1$, etc.
- special shape of $\mathbf{f}$ (positivity, monotonicity, convexity → Section 5.3 below)

Example 5.1.5 (Constitutive relations from measurements)

This example addresses an important application of data interpolation in 1D.

In this context: t, y $\hat{=}$ two state variables of a physical system, where t determines y: a functional dependence $y = y(t)$ is assumed.

Examples: t and y could be

t	y
voltage U	current I
pressure p	density ρ
magnetic field H	magnetic flux B
...	...

Known: several *accurate* $(*)$ measurements

$$(t_i, y_i)\,, \quad i = 1, \ldots, m$$

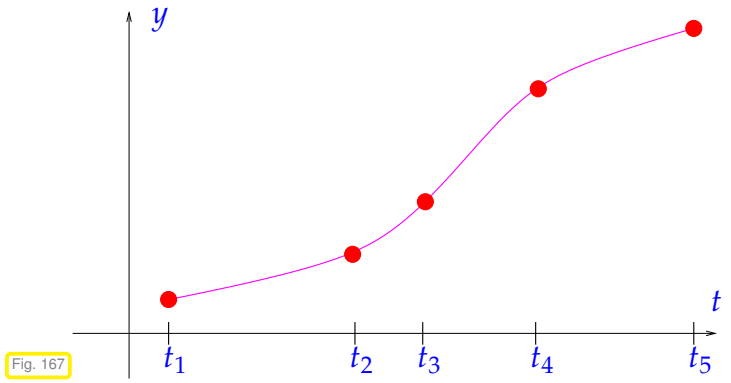


Fig. 167

Imagine that t, y correspond to the voltage U and current I measured for a 2-port non-linear circuit element (like a diode). This element will be part of a circuit, which we want to simulate based on nodal analysis as in Ex. 8.0.1. In order to solve the resulting non-linear system of equations $F(\mathbf{u}) = 0$ for the nodal potentials (collected in the vector $\mathbf{u}$) by means of Newton's method ($\rightarrow$ Section 8.4) we need the voltage-current relationship for the circuit element as a continuously differentiable function $I = f(U)$.

$(*)$ Meaning of attribute "accurate": justification for interpolation. If measured values y_i were affected by considerable errors, one would not impose the interpolation conditions (**??**), but opt for data fitting ($\rightarrow$ Section 5.7).

We can distinguish two aspects of the interpolation problem:

❶ Find interpolant $f : I \subset \mathbb{R} \to \mathbb{R}$ and store/represent it (internally).

❷ Evaluate f at a few or many evaluation points $x \in I$

Remark 5.1.6 (Mathematical functions in a numerical code)

What does is meant to "represent" or "make available" a function $f : I \subset \mathbb{R} \mapsto \mathbb{R}$ in a computer code?

! A general "mathematical" function $f : I \subset \mathbb{R} \mapsto \mathbb{R}^d$, I an interval, contains an "infinite amount of information".

Rather, in the context of numerical methods, "function" should be read as "subroutine", a piece of code that can, for any $x \in I$, compute $f(x)$ in finite time. Even this has to be qualified, because we can only pass machine numbers $x \in I \cap \mathbb{M}$ ($\rightarrow$ § 1.5.12) and, of course, in most cases, $f(x)$ will be an approximation. In a C++ code a simple real valued function can be incarnated through a function object of a type as given in Code 5.1.7, see also Section 0.2.3.

C++-code 5.1.7: C++ data type representing a real-valued function

```
1 class Function {
2   private:
3     // various internal data describing f
```

```
4   public:
5     // Constructor: expects information for specifying the function
6     Function(/* ... */);
7     // Evaluation operator
8     double operator() (double t) const;
9   };
```

(5.1.8) Internal representation of classes of mathematical functions

➡ Idea: parametrization, a finite number of parameters $c_0, \ldots, c_m$, $m \in \mathbb{N}$, characterizes f.

Special case: Representation with *finite linear combination* of basis functions $b_j : I \subset \mathbb{R} \mapsto \mathbb{R}$, $j = 0, \ldots, m$:

$$f = \sum_{j=0}^{m} c_j b_j \quad , \quad c_j \in \mathbb{R}^d \, . \tag{5.1.9}$$

↣ $f \in$ finite dimensional function space $V_m := \operatorname{Span}\{b_0, \ldots, b_m\}$, $\dim V_m = m + 1$.

Of course, the basis functions b_j should be "simple" in the sense that $b_j(x)$ can be computed efficiently for every $x \in I$ and every $j = 0, \ldots, m$.

Note that the basis functions *may depend on the nodes* t_i, but they *must not depend on the values* y_i.

➡ The internal representation of f (in the data member section of the class **Function** from Code 5.1.7) will then boil down to storing the coefficients/parameters c_j, $j = 0, \ldots, m$.

Note: The focus in this chapter will be on the special case that the data interpolants belong to a finite-dimensional space of functions spanned by "simple" basis functions.

Example 5.1.10 (Piecewise linear interpolation, see also Section 5.3.2)

Recall: A linear function in 1D is a function of the form $x \mapsto a + bx$, $a, b \in \mathbb{R}$ (polynomial of degree 1).

Piecewise linear interpolation

= connect data points (t_i, y_i), $i = 0, \ldots, n$, $t_{i-1} < t_i$, by line segments

➢ interpolating polygon

Piecewise linear interpolant of data ▷

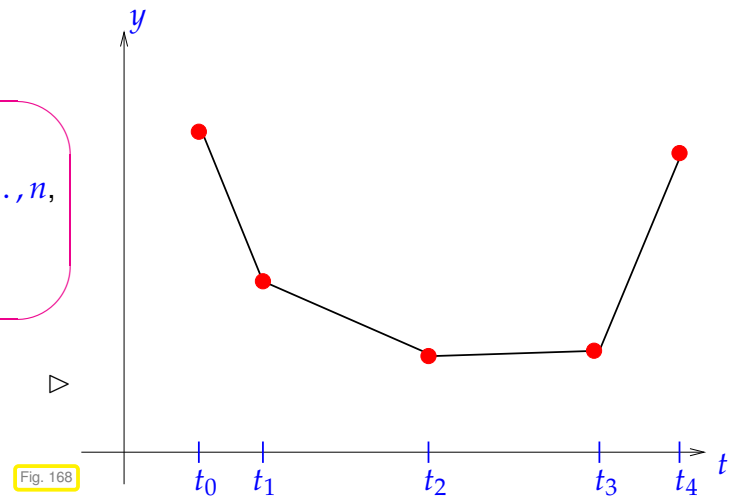


Fig. 168

What could be a convenient set of basis functions $\{b_j\}_{j=0}^n$ for representing the piecewise linear interpolant through $n+1$ data points?

"Tent function" ("hat function") basis:

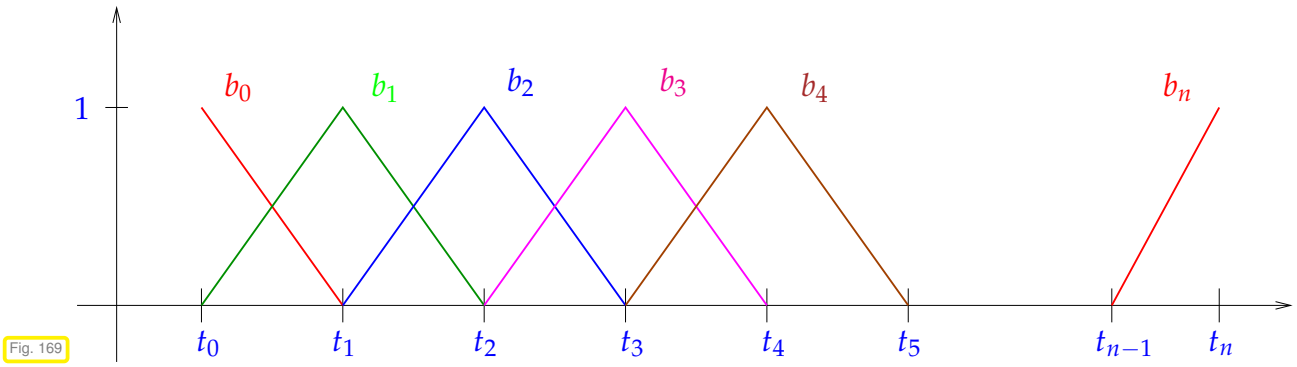


Fig. 169

Note: in Fig. 169 the basis functions have to be extended by zero outside the t-range where they are drawn.

Explicit formulas for these basis functions can be given and bear out that they are really "simple":

$$b_0(t) = \begin{cases} 1 - \frac{t-t_0}{t_1-t_0} & \text{for } t_0 \le t < t_1 \,, \\ 0 & \text{for } t \ge t_1 \,. \end{cases}$$

$$b_j(t) = \begin{cases} 1 - \frac{t_j - t}{t_j - t_{j-1}} & \text{for } t_{j-1} \le t < t_j \,, \\ 1 - \frac{t-t_j}{t_{j+1}-t_j} & \text{for } t_j \le t < t_{j+1} \,, \\ 0 & \text{elsewhere in } [t_0, t_n] \,. \end{cases} \quad , \quad j = 1, \ldots, n-1 \,, \tag{5.1.11}$$

$$b_n(t) = \begin{cases} 1 - \frac{t_n - t}{t_n - t_{n-1}} & \text{for } t_{n-1} \le t < t_n \,, \\ 0 & \text{for } t < t_{n-1} \,. \end{cases}$$

Moreover, these basis functions are *uniquely* determined by the conditions

- b_j is continuous on $[t_0, t_n]$,
- b_j is linear on each interval $[t_{i-1}, t_i]$, $i = 1, \ldots, n$,
- $b_j(t_i) = \delta_{ij} := \begin{cases} 1 & \text{, if } i = j \,, \\ 0 & \text{else.} \end{cases}$ ➢ a so-called cardinal basis for the node set $\{t_i\}_{i=0}^n$.

This last condition implies a simple basis representation of a (the ?) piecewise linear interpolant of the data points (t_i, y_i), $i = 0, \ldots, n$:

$$f(t) = \sum_{j=0}^{n} y_j b_j(t) \,, \quad t_0 \le t \le t_n \,, \tag{5.1.12}$$

where the b_j are given by (5.1.11).

(5.1.13) Interpolation as a linear mapping

We consider the setting for interpolation that the interpolant belongs to a finite-dimension space V_m of functions spanned by basis functions $b_0, \ldots, b_m$, see Rem. 5.1.6. Then the interpolation conditions imply

that the basis expansion coefficients satisfy a linear system of equations:

$$\text{(??) \& (5.1.9)} \quad \Rightarrow \quad f(t_i) = \sum_{j=0}^{m} c_j b_j(t_i) = y_i\,, \quad i = 0,\ldots,n\,, \tag{5.1.14}$$

$$\Updownarrow$$

$$\mathbf{A}\mathbf{c} := \begin{bmatrix} b_0(t_0) & \ldots & b_m(t_0) \\ \vdots & & \vdots \\ b_0(t_n) & \ldots & b_m(t_n) \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_m \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix} =: \mathbf{y}\,. \tag{5.1.15}$$

This is an $(m+1) \times (n+1)$ linear system of equations !

The interpolation problem in V_m and the linear system (5.1.15) are really equivalent in the sense that (unique) solvability of one implies (unique) solvability of the other.

Necessary condition for unique solvability of interpolation problem (5.1.14) : $m = n$

If $m = n$ and $\mathbf{A}$ from (5.1.15) regular ($\to$ Def. 2.2.1),then for *any* values y_j, $j = 0,\ldots,n$ we can find coefficients c_j, $j = 0,\ldots,n$, and, from them build the interpolant according to (5.1.9):

$$f = \sum_{j=0}^{n} (\mathbf{A}^{-1}\mathbf{y})_j b_j\,. \tag{5.1.16}$$

For *fixed* nodes t_i the interpolation problem (5.1.14) defines linear mapping

$$\mathsf{I} : \begin{cases} \mathbb{R}^{n+1} & \mapsto & V_n \\ \mathbf{y} & \mapsto & f \end{cases}$$

data space function space

Beware, "linear" in the statement above has nothing to do with a linear function or piecewise linear interpolation discussed in Ex. 5.1.10!

Definition 5.1.17. Linear interpolation operator

An interpolation operator $\mathsf{I} : \mathbb{R}^{n+1} \mapsto C^0([t_0,t_m])$ for the given nodes $t_0 < t_1 < \cdots < t_n$ is called linear, if

$$\mathsf{I}(\alpha\mathbf{y} + \beta\mathbf{z}) = \alpha\mathsf{I}(\mathbf{y}) + \beta\mathsf{I}(\mathbf{z}) \quad \forall \mathbf{y},\mathbf{z} \in \mathbb{R}^{n+1},\ \alpha,\beta \in \mathbb{R}\,. \tag{5.1.18}$$

✎ Notation: $C^0([t_0,t_m]) \hat{=}$ vector space of continuous functions on $[t_0,t_m]$

Remark 5.1.19 (A data type designed for of interpolation problem)

If a constitutive relationship for a circuit element is needed in a C++ simulation code ($\to$ Ex. 5.1.5), the following data type could be used to represent it:

C++-code 5.1.20: C++ class representing an interpolant in 1D

```
class Interpolant {
  private:
    // Various internal data describing f
    // Can be the coefficients of a basis representation (5.1.9)
  public:
    // Constructor: computation of coefficients c_j of representation (5.1.9)
    Interpolant(const vector<double>& t, const vector<double>& y);
    // Evaluation operator for interpolant f
    double operator() (double t) const;;
```

Practical object oriented implementation of interpolation operator:

- Constructor: "setup phase", e.g. building and solving linear system of equations (5.1.15)
- Evaluation operator, e.g., implemented as evaluation of linear combination (5.1.9)

Crucial issue: computational effort for evaluation of interpolant at single point: $O(1)$ or $O(n)$ (or in between)?

5.2 Global Polynomial Interpolation

(Global) polynomial interpolation, that is, interpolation into spaces of functions spanned by polynomials up to a certain degree, is the simplest interpolation scheme and of great importance as building block for more complex algorithms.

5.2.1 Polynomials

Notation: Vector space of the polynomials of degree $\leq k$, $k \in \mathbb{N}$:

$$\mathcal{P}_k := \{t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_1 t + \alpha_0 \,,\, \alpha_j \in \mathbb{R}\} \,. \tag{5.2.1}$$

leading coefficient

Terminology: the functions $t \mapsto t^k$, $k \in \mathbb{N}_0$, are called monomials

$t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_0$ = monomial representation of a polynomial.

Monomial representation is a linear combination of basis functions $t \mapsto t^k$, see Rem. 5.1.6.

Obvious: $\mathcal{P}_k$ is a vector space, see [**?**, Sect. 4.2, Bsp. 4]. What is its dimension?

Theorem 5.2.2. Dimension of space of polynomials

$$\dim \mathcal{P}_k = k+1 \quad \textit{and} \quad \mathcal{P}_k \subset C^\infty(\mathbb{R}).$$

Proof. Dimension formula by linear independence of monomials. □

As a consequence of Thm. 5.2.2 the monomial representation of a polynomial is unique.

(5.2.3) The charms of polynomials

Why are polynomials important in computational mathematics?

- ➙ Easy to compute (only elementary operations required), integrate and differentiate
- ➙ Vector space & algebra
- ➙ Analysis: Taylor polynomials & power series

Remark 5.2.4 (Monomial representation)

Polynomials (of degree k) in monomial representation are stored as a vector of their coefficients a_j, $j = 0,\ldots,k$. A convention for the ordering has to be fixed. For instance, MATLAB functions expect a monomial representation through a vector of their monomial coefficients in descending order:

$$\text{MATLAB: } \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_0 \quad \rightarrow \quad \text{vector } [\alpha_k, \alpha_{k-1}, \ldots, \alpha_0]^\top \text{ (ordered!).}$$

Remark 5.2.5 (Horner scheme → [?, Bem. 8.11])

Efficient evaluation of a polynomial in monomial representation through Horner scheme as indicated by the following representation:

$$p(t) = t(\cdots t(t(\alpha_n t + \alpha_{n-1}) + \alpha_{n-2}) + \cdots + \alpha_1) + \alpha_0 \,. \tag{5.2.6}$$

The following code gives an implementation based on vector data types of EIGEN. The function is vectorized in the sense that many evaluation points are processed in parallel.

C++-code 5.2.7: Horner scheme (vectorized version)

```
2  // Efficient evaluation of a polynomial in monomial representation
3  // using the Horner scheme (5.2.6)
4  // IN: p = vector of monomial coefficients, length = degree + 1
5  // (leading coefficient in p(0), MATLAB convention Rem. 5.2.4)
6  // t = vector of evaluation points t_i
7  // OUT: y = polynomial evaluated at t_i
8  void horner(const VectorXd& p, const VectorXd& t, VectorXd& y) {
9    const VectorXd::Index n = t.size();
10   y.resize(n); y = p(0)*VectorXd::Ones(n);
11   for (unsigned i = 1; i < p.size(); ++i)
12     y = t.cwiseProduct(y) + p(i)*VectorXd::Ones(n);
13 }
```

Optimal asymptotic complexity: $O(n)$

Realized in MATLAB "built-in"-function `polyval(p,x)`. (The argument `x` can be a matrix or a vector. In this case the function evaluates the polynomial described by `p` for each entry/component.)

5.2.2 Polynomial Interpolation: Theory

Supplementary reading. This topic is also presented in [**?**, Sect. 8.2.1], [**?**, Sect. 8.1], [**?**, Ch. 10].

Now we consider the interpolation problem introduced in Section 5.1 for the special case that the sought interpolant belongs to the polynomial space $\mathcal{P}_k$ (with suitable degree k).

Lagrange polynomial interpolation problem

Given the simple nodes $t_0, \ldots, t_n$, $n \in \mathbb{N}$, $-\infty < t_0 < t_1 < \cdots < t_n < \infty$ and the values $y_0, \ldots, y_n \in \mathbb{R}$ compute $p \in \mathcal{P}_n$ such that

$$p(t_j) = y_j \quad \text{for} \quad j = 0, \ldots, n\,. \tag{5.2.9}$$

Is this a well-defined problem? Obviously, it fits the framework developed in Rem. 5.1.6 and § 5.1.13, because $\mathcal{P}_n$ is a finite-dimensional space of functions, for which we already know a basis, the monomials. Thus, in principle, we could examine the matrix $\mathbf{A}$ from (5.1.15) to decide, whether the polynomial interpolant exists and is unique. However, there is a shorter way.

(5.2.10) Lagrange polynomials

For nodes $t_0 < t_1 < \cdots < t_n$ ($\rightarrow$ Lagrange interpolation) consider the

$$\text{Lagrange polynomials} \qquad L_i(t) := \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{t - t_j}{t_i - t_j}\,, \quad i = 0, \ldots, n\,. \tag{5.2.11}$$

➙ Evidently, the Lagrange polynomials satisfy $L_i \in \mathcal{P}_n$ and $\boxed{L_i(t_j) = \delta_{ij}}$

Recall the Kronecker symbol $\delta_{ij} = \begin{cases} 1 & \text{if } i = j\,, \\ 0 & \text{else.} \end{cases}$

From this relationship we infer that the Lagrange polynomials are linearly independent. Since there are $n + 1 = \dim \mathcal{P}_n$ different Lagrange polynomials, we conclude that they form a basis of $\mathcal{P}_n$, which is a cardinal basis for the node set $\{t_i\}_{i=0}^n$.

Example 5.2.12 (Lagrange polynomials for uniformly spaced nodes)

Consider the equidistant nodes in $[-1,1]$:

$$\mathcal{T} := \{t_j = -1 + \tfrac{2}{n}j\}\,, \quad j = 0,\ldots,n\,.$$

The plot shows the Lagrange polynomials for this set of nodes that do not vanish in the nodes t_0, t_2, and t_5, respectively.

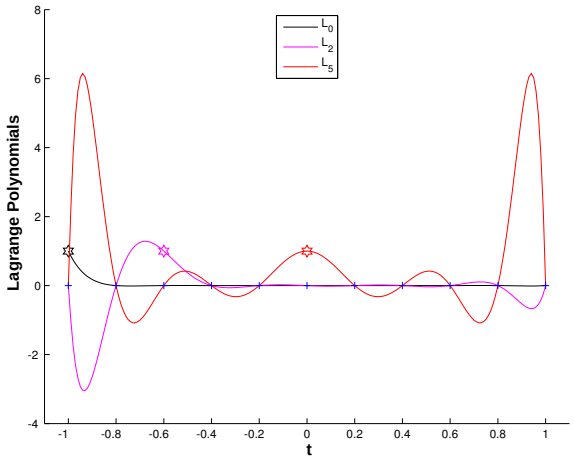


Fig. 170

The Lagrange polynomial interpolant p for data points $(t_i, y_i)_{i=0}^n$ allows a straightforward representation with respect to the basis of Lagrange polynomials for the node set $\{t_i\}_{i=0}^n$:

$$p(t) = \sum_{i=0}^{n} y_i L_i(t) \quad \Leftrightarrow \quad p \in \mathcal{P}_n \quad \text{and} \quad p(t_i) = y_i \,. \tag{5.2.13}$$

Theorem 5.2.14. Existence & uniqueness of Lagrange interpolation polynomial → [?, Thm. 8.1], [?, Satz 8.3]

The general Lagrange polynomial interpolation problem admits a unique solution $p \in \mathcal{P}_n$.

Proof. Consider the *linear* evaluation operator

$$\mathrm{eval}_{\mathcal{T}} : \begin{cases} \mathcal{P}_n & \mapsto \mathbb{R}^{n+1}\,, \\ p & \mapsto (p(t_i))_{i=0}^n\,, \end{cases}$$

which maps between finite-dimensional vector spaces of the same dimension, see Thm. 5.2.2.

Representation (5.2.13) $\Rightarrow$ existence of interpolating polynomial
$\Rightarrow$ $\mathrm{eval}_{\mathcal{T}}$ is surjective ("onto")

Known from linear algebra: for a linear mapping $T : V \mapsto W$ between finite-dimensional vector spaces with $\dim V = \dim W$ holds the equivalence

$$T \text{ surjective} \quad \Leftrightarrow \quad T \text{ bijective} \quad \Leftrightarrow \quad T \text{ injective.}$$

Applying this equivalence to $\mathrm{eval}_{\mathcal{T}}$ yields the assertion of the theorem □

Corollary 5.2.15. Lagrange interpolation as linear mapping → § 5.1.13

The polynomial interpolation in the nodes $\mathcal{T} := \{t_j\}_{j=0}^n$ defines a linear operator

$$\mathsf{I}_{\mathcal{T}} : \begin{cases} \mathbb{R}^{n+1} & \to \mathcal{P}_n\,, \\ (y_0,\ldots,y_n)^T & \mapsto \textit{interpolating polynomial } p\,. \end{cases} \tag{5.2.16}$$

Remark 5.2.17 (Vandermonde matrix)

Lagrangian polynomial interpolation leads to linear systems of equations also for the representation coefficients of the polynomial interpolant in monomial basis, see § 5.1.13:

$$p(t_j) = y_j \iff \sum_{i=0}^{n} a_i t_j^i = y_j\,, \quad j = 0,\ldots,n$$
$$\iff \text{solution of } (n+1)\times(n+1) \text{ linear system } \mathbf{Va} = \mathbf{y} \text{ with matrix}$$

$$\mathbf{V} = \begin{bmatrix} 1 & t_0 & t_0^2 & \cdots & t_0^n \\ 1 & t_1 & t_1^2 & \cdots & t_1^n \\ 1 & t_2 & t_2^2 & \cdots & t_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & t_n^2 & \cdots & t_n^n \end{bmatrix} \in \mathbb{R}^{n+1,n+1}\,. \tag{5.2.18}$$

A matrix in the form of $\mathbf{V}$ is called Vandermonde matrix.

The following code initializes a Vandermonde matrix in EIGEN:

C++11 code 5.2.19: Initialization of Vandermonde matrix

```
// Initialization of a Vandermonde matrix (5.2.18)
// from interpolation points t_i.
MatrixXd vander(const VectorXd &t) {
  const VectorXd::Index n = t.size();
  MatrixXd V(n,n); V.col(0) = VectorXd::Ones(n); V.col(1) = t;
  // Store componentwise integer powers of point coordinate vector
  // into the columns of the Vandermonde matrix
  for (int j=2;j<n;j++) V.col(j) = (t.array().pow(j)).matrix();
  return V;
}
```

Remark 5.2.20 (Matrix representation of interpolation operator)

In the case of Lagrange interpolation:

- if Lagrange polynomials are chosen as basis for $\mathcal{P}_n$, then $\mathrm{I}_{\mathcal{T}}$ is represented by the identity matrix;
- if monomials are chosen as basis for $\mathcal{P}_n$, then $\mathrm{I}_{\mathcal{T}}$ is represented by the inverse of the Vandermonde matrix $\mathbf{V}$, see Eq. (5.2.18).

Remark 5.2.21 (Generalized polynomial interpolation → [?, Sect. 8.2.7], [?, Sect. 8.4])

The following generalization of Lagrange interpolation is possible: We still seek a polynomial interpolant, but beside function values also prescribe derivatives up to a certain order for interpolating polynomial at given nodes.

Convention: indicate occurrence of derivatives as interpolation conditions by *multiple nodes*.

Generalized polynomial interpolation problem

Given the (possibly multiple) nodes $t_0, \ldots, t_n$, $n \in \mathbb{N}$, $-\infty < t_0 \leq t_1 \leq \cdots \leq t_n < \infty$ and the values $y_0, \ldots, y_n \in \mathbb{R}$ compute $p \in \mathcal{P}_n$ such that

$$\frac{\mathrm{d}^k}{\mathrm{d}t^k} p(t_j) = y_j \quad \text{for} \quad k = 0, \ldots, l_j \quad \text{and} \quad j = 0, \ldots, n \,, \tag{5.2.22}$$

where $l_j := \max\{i - i' : t_j = t_i = t_{i'},\, i, i' = 0, \ldots, n\}$ is the multiplicity of the nodes t_j.

The most important case of generalized Lagrange interpolatioon is when all the multiplicities are equal to 2. It is called Hermite interpolation (or osculatory interpolation) and the generalized interpolation conditions read for nodes $t_0 = t_1 < t_2 = t_3 < \cdots < t_{n-1} = t_n$ (note the double nodes!) [**?**, Ex. 8.6]:

$$p(t_{2j}) = y_{2j} \quad , \quad p'(t_{2j}) = y_{2j+1} \,, \quad j = 0, \ldots, n/2 \,.$$

Theorem 5.2.23. Existence & uniqueness of generalized Lagrange interpolation polynomials

The generalized polynomial interpolation problem Eq. (5.2.22) *admits a unique solution* $p \in \mathcal{P}_n$.

Definition 5.2.24. Generalized Lagrange polynomials

The generalized Lagrange polynomials for the nodes $\mathcal{T} = \{t_j\}_{j=0}^n \subset \mathbb{R}$ (multiple nodes allowed) are defined as $L_i := \mathrm{I}_{\mathcal{T}}(\mathbf{e}_{i+1})$, $i = 0, \ldots, n$, where $\mathbf{e}_i = (0, \ldots, 0, 1, 0, \ldots, 0)^T \in \mathbb{R}^{n+1}$ are the unit vectors.

Note: The linear interpolation operator $\mathrm{I}_{\mathcal{T}}$ in this definition refers to generalized Lagrangian interpolation. Its existence is guaranteed by Thm. 5.2.23.

Example 5.2.25 (Generalized Lagrange polynomials for Hermite Interpolation)

Consider the node set

$$\mathcal{T} = \{t_0 = 0, t_1 = 0, t_2 = 1, t_3 = 1\} \,.$$

The plot shows the four unique generalized Lagrange polynomials of degree $n = 3$ for these nodes. They satisfy

$$\begin{aligned} p_0(0) = 1, p_0(1) = p_0'(0) = p_0'(1) = 0 \,, \\ p_1(1) = 1, p_1(0) = p_1'(0) = p_1'(1) = 0 \,, \\ p_2'(0) = 1, p_2(1) = p_2(0) = p_2'(1) = 0 \,, \\ p_3'(1) = 1, p_3(1) = p_3(0) = p_3'(0) = 0 \,. \end{aligned}$$

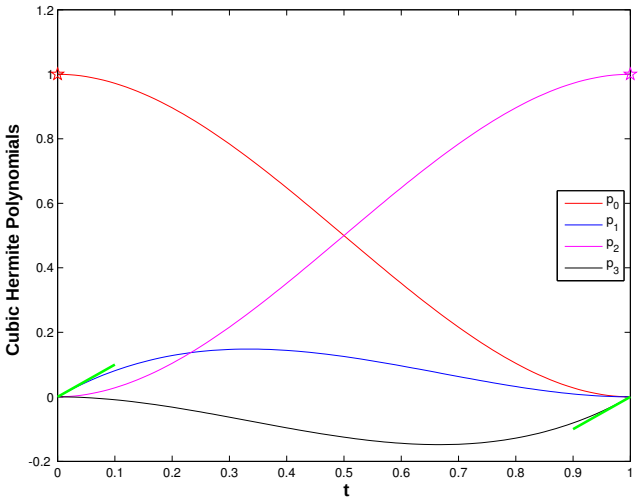


Fig. 171

More details are given in Section 5.4. For explicit formulas for the polynomials see (5.4.5).

5.2.3 Polynomial Interpolation: Algorithms

Now we consider the algorithmic realization of Lagrange interpolation as introduced in Section 5.2.2. The setting is a follows:

Given: nodes $\mathcal{T} := \{-\infty < t_0 < t_1 < \ldots < t_n < \infty\}$,
values $\boldsymbol{y} := \{y_0, y_1, \ldots, y_n\}$,

We also write $p := \mathrm{I}_{\mathcal{T}}(\boldsymbol{y})$ for the unique Lagrange interpolation polynomial given by Thm. 5.2.14.

When used in a numerical code, different demands can be made for a routine that implements Lagrange interpolation. They determine, which algorithm is most suitable.

5.2.3.1 Multiple evaluations

Task: For ❶ a *fixed* set $\{t_0, \ldots, t_n\}$ of nodes,
and ❷ *many* different given data values y_i^k, $i = 0, \ldots, n$
and ❸ *many* arguments x_k, $k = 1, \ldots, N$, $N \gg 1$,
efficiently compute all $p(x_k)$ for $p \in \mathcal{P}_n$ interpolating in (t_i, y_i^k), $i = 0, \ldots, n$.

The definition of a possible interpolator data type could be as follows:

C++-code 5.2.26: Polynomial Interpolation

```
class PolyInterp {
  private:
    // various internal data describing p
    Eigen::VectorXd t;
  public:
    // Constructors taking node vector (t_0,...,t_n) as argument
  PolyInterp(const Eigen::VectorXd &_t);
  template <typename SeqContainer>
           PolyInterp(const SeqContainer &v);
  // Evaluation operator for data (y_0,...,y_n); computes
  // p(x_k) for x_k's passed in x
  Eigen::VectorXd eval(const Eigen::VectorXd &y,
                       const Eigen::VectorXd &x) const;
 };
```

The member function `eval(y,x)` expects n data values in `y` and (any number of) evaluation points in `x` ($\leftrightarrow [x_1, \ldots, x_N]^\top$) and returns the vector $[p(x_1), \ldots, p()x_N)]^\top$, where p is the Lagrange polynomial interpolant.

An implementation directly based on the evaluation of Lagrange polynomials (5.2.11) and (5.2.13) would incur an asymptotic computational effort of $O(n^2 N)$ for every single invocation of `eval` and large n, N.

(5.2.27) Barycentric interpolation formula

By means of *pre-calculations* the asymptotic effort for `eval` can be reduced substantially:

Simple manipulations starting from (5.2.13) give an altenative representation of p:

$$p(t) = \sum_{i=0}^{n} L_i(t)\, y_i = \sum_{i=0}^{n} \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{t-t_j}{t_i-t_j}\, y_i = \sum_{i=0}^{n} \lambda_i \prod_{\substack{j=0 \\ j\neq i}}^{n} (t-t_j)\, y_i = \prod_{j=0}^{n}(t-t_j) \cdot \sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}\, y_i \,.$$

where $\lambda_i = \dfrac{1}{(t_i-t_0)\cdots(t_i-t_{i-1})(t_i-t_{i+1})\cdots(t_i-t_n)}$, $i = 0,\ldots,n$.

From the above formula, with $p(t) \equiv 1$, $y_i = 1$:

$$1 = \prod_{j=0}^{n}(t-t_j) \sum_{i=0}^{n} \frac{\lambda_i}{t-t_i} \quad \Rightarrow \quad \prod_{j=0}^{n}(t-t_j) = \frac{1}{\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}}$$

$$\blacktriangleright \quad \text{Barycentric interpolation formula} \quad p(t) = \frac{\displaystyle\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}\, y_i}{\displaystyle\sum_{i=0}^{n} \frac{\lambda_i}{t-t_i}} \,. \tag{5.2.28}$$

with $\lambda_i = \dfrac{1}{(t_i-t_0)\cdots(t_i-t_{i-1})(t_i-t_{i+1})\cdots(t_i-t_n)}$, $i = 0,\ldots,n$, independent of t and y_i
$\rightarrow$ precompute **!**

The use of (5.2.28) involves

- computation of weights λ_i, $i = 0,\ldots,n$: cost $O(n^2)$ (only once!),
- cost $O(n)$ for every subsequent evaluation of p.

$\Rightarrow$ total asymptotic complexity $O(Nn) + O(n^2)$

The following C++ class demonstrated the use of the barycentric interpolation formula for efficient multiple point evaluation of a Lagrange interpolation polynomial:

C++-code 5.2.29: Class for multiple data/multiple point evaluations

```
2  template <typename NODESCALAR = double>
3  class BarycPolyInterp {
4  private:
5    using nodeVec_t = Eigen::Matrix<NODESCALAR, Eigen::Dynamic, 1>;
6    using idx_t = typename nodeVec_t::Index;
7    // Number n of interpolation points, deg polynomial +1
8    const idx_t n;
9    // Locations of n interpolation points
10   nodeVec_t t;
11   // Precomputed values λ_i, i = 0,...,n-1
12   nodeVec_t lambda;
13 public:
14   // Constructors taking node vector [t_0,...,t_n]^T as argument
15   BarycPolyInterp(const nodeVec_t &_t);
```

```
16   // The interpolation points may also be passed in an STL container
17   template <typename SeqContainer>
18     BarycPolyInterp(const SeqContainer &v);
19   // Computation of p(x_k) for data values (y_0,...,y_n) and
20   // evaluation points x_k
21   template <typename RESVEC,typename DATAVEC>
22   RESVEC eval(const DATAVEC &y,const nodeVec_t &x) const;
23 private:
24   void init_lambda(void);
25  };
```

C++-code 5.2.30: Interpolation class: constructors

```
2  template <typename NODESCALAR>
3  BarycPolyInterp<NODESCALAR>::BarycPolyInterp(const nodeVec_t
     &_t):n(_t.size()),t(_t),lambda(n) {
4    init_lambda(); }
5
6  template <typename NODESCALAR>
7  template <typename SeqContainer>
8  BarycPolyInterp<NODESCALAR>::BarycPolyInterp(const SeqContainer
     &v):n(v.size()),t(n),lambda(n) {
9    idx_t ti = 0; for(auto tp: v) t(ti++) = tp;
10   init_lambda();
11 }
```

C++-code 5.2.31: Interpolation class: precomputations

```
2  template <typename NODESCALAR>
3  void BarycPolyInterp<NODESCALAR>::init_lambda(void) {
4    // Precompute the weights λ_i with effort O(n^2)
5    for (unsigned k = 0; k < n; ++k) {
6      // little workaround: in EIGEN cannot subtract a vector
7      // from a scalar; multiply scalar by vector of ones
8      lambda(k) = 1./((t(k)*nodeVec_t::Ones(k)-t.head(k)).prod()*
9              (t(k)*nodeVec_t::Ones(n-k-1)-t.tail(n-k-1)).prod());
10   }
11 }
```

C++-code 5.2.32: Interpolation class: multiple point evaluations

```
2  template <typename NODESCALAR>
3  template <typename RESVEC,typename DATAVEC>
4  RESVEC BarycPolyInterp<NODESCALAR>::eval
5    (const DATAVEC &y,const nodeVec_t &x) const {
6    const idx_t N = x.size(); // No. of evaluation points
7    RESVEC p(N); // Ouput vector
8    // Compute quotient of weighted sums of λ_i/(t-t_i), effort O(n)
```

```
9   for (unsigned i = 0; i < N; ++i) {
10    nodeVec_t z = (x(i)*nodeVec_t::Ones(n) - t);
11
12    // check if we want to evaluate at a node <-> avoid division by
        zero
13    NODESCALAR* ptr = std::find(z.data(), z.data() + n,
        NODESCALAR(0)); //
14    if (ptr != z.data() + n) { // if ptr = z.data + n = z.end no zero
        was found
15      p(i) = y(ptr - z.data()); // ptr - z.data gives the position of
          the zero
16    }
17    else {
18      const nodeVec_t mu = lambda.cwiseQuotient(z);
19      p(i) = (mu.cwiseProduct(y)).sum()/mu.sum();
20    }
21  } // end for
22  return p;
23 }
```

As an exception, the test for equality with zero in Line 13 is possible. If `x(i)` is almost equal to `t`, then the corresponding entry of the vector `mu` will be huge and the value of the barycentric sum will almost agree with `p(i)`, the same value returned, if `x(i)` is exactly zero. Hence, it does not matter, if the test returns `false` because of some small perturbations of the values.

Runtime measurements of direct evaluation of a polynomial in monomial representation vs. barycentric formula are reported in Exp. 5.2.38.

5.2.3.2 Single evaluation

Supplementary reading. This topic is also discussed in [**?**, Sect. 8.2.2].

Task: Given a set of interpolation points (t_j, y_j), $j = 0, \ldots, n$, with pairwise different interpolation nodes t_j, perform *a single* point evaluation of the Lagrange polynomial interpolant p at $x \in \mathbb{R}$.

We discuss the efficient implementation of the following function for $n \gg 1$. It is meant for a single evaluation of a Lagrange interpolant.

```
double eval(const Eigen::VectorXd &t,const Eigen::VectorXd &y,
            double x);
```

(5.2.33) Aitken-Neville scheme

The starting point is a recursion formula for partial Lagrange interpolants: For $0 \le k \le \ell \le n$ define

$$p_{k,\ell} := \text{unique interpolating polynomial of degree } \ell - k \text{ through } (t_k, y_k), \ldots, (t_\ell, y_\ell),$$

From the uniqueness of polynomial interpolants ($\rightarrow$ Thm. 5.2.14) we find

$$\begin{aligned} p_{k,k}(x) &\equiv y_k \quad \text{(“constant polynomial”)}\,, \quad k = 0, \ldots, n\,, \\ p_{k,\ell}(x) &= \frac{(x - t_k)p_{k+1,\ell}(x) - (x - t_\ell)p_{k,\ell-1}(x)}{t_\ell - t_k} \\ &= p_{k+1,\ell}(x) + \frac{x - t_\ell}{t_\ell - t_k}\,(p_{k+1,\ell}(x) - p_{k,\ell-1}(x))\,, \quad 0 \le k \le \ell \le n\,, \end{aligned} \tag{5.2.34}$$

because the left and right hand sides represent polynomials of degree $\ell - k$ through the points (t_j, y_j), $j = k, \ldots, \ell$.

Thus the values of the partial Lagrange interpolants can be computed sequentially and their dependencies can be expressed by the following so-called Aitken-Neville scheme:

$n =$	0	1	2	3
t_0	$y_0 =: p_{0,0}(x)$	$\nearrow p_{0,1}(x)$	$\nearrow p_{0,2}(x)$	$\nearrow p_{0,3}(x)$
t_1	$y_1 =: p_{1,1}(x)$	$\nearrow p_{1,2}(x)$	$\nearrow p_{1,3}(x)$	
t_2	$y_2 =: p_{2,2}(x)$	$\nearrow p_{2,3}(x)$		
t_3	$y_3 =: p_{3,3}(x)$			

(ANS)

Here, the arrows indicate contributions to the convex linear combinations of (5.2.34). The computation can advance from left to right, which is done in following C++ code.

C++-code 5.2.35: Aitken-Neville algorithm

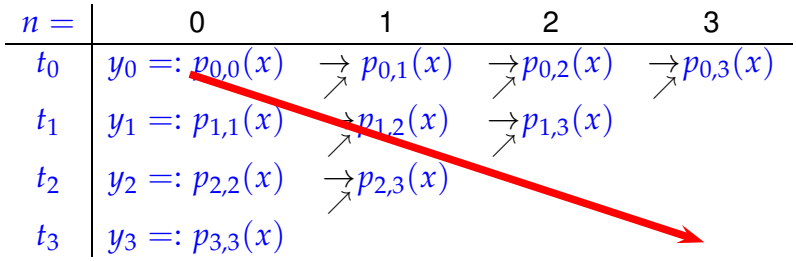
```
// Aitken-Neville algorithm for evaluation of interpolating polynomial
// IN: t, y:  (vectors of) interpolation data points
// x:  (single) evaluation point
// OUT: value of interpolant in x
double ANipoleval(const VectorXd& t, VectorXd y, const double x) {
  for (int i = 0; i < y.size(); ++i) {
    for (int k = i - 1; k >= 0; --k) {
      // Recursion (5.2.34)
      y(k) = y(k + 1) + (y(k + 1) - y(k))*(x - t(i))/(t(i) - t(k));
    }
  }
  return y(0);
}
```

The vector `y` contains the columns of the above triangular tableaux in turns from left to right.

Asymptotic complexity of `ANipoeval` in terms of number of data points: $O(n^2)$ (two nested loops).

(5.2.36) Polynomial interpolation with data updates

The Aitken-Neville algorithm has another interesting feature, when we run through the Aitken-Neville scheme from the top left corner:



Thus, the values of partial polynomial interpolants at x can be computed before all data points are even processed. This results in an "update-friendly" algorithm that can efficiently supply the point values $p_{0,k}(x)$, $k = 0, \ldots, n$, while being supplied with the data points (t_i, y_i). It can be used for the efficient implementation of the following interpolator class:

C++-code 5.2.37: Single point evaluation with data updates

```
1  class PolyEval {
2      private:
3       // evaluation point and various internal data describing the
           polynomials
4      public:
5       // Constructor taking the evaluation point as argument
6       PolyEval(double x);
7       // Add another data point and update internal information
8       void addPoint(t,y);
9       // Value of current interpolating polynomial at x
10      double eval(void) const;
11 };
```

Experiment 5.2.38 (Timing polynomial evaluations)

Timing for single-point evaluation

Comparison of the computational time needed for polynomial interpolation of

$$\{t_i = i\}_{i=1,\ldots,n}, \qquad \{y_i = \sqrt{i}\}_{i=1,\ldots,n},$$

$n = 3,\ldots,200$, and evaluation in a single point $x \in [0,n]$.

Minimum computational time over 100 runs ➙

The measurements were carried out with code Code 5.2.39.

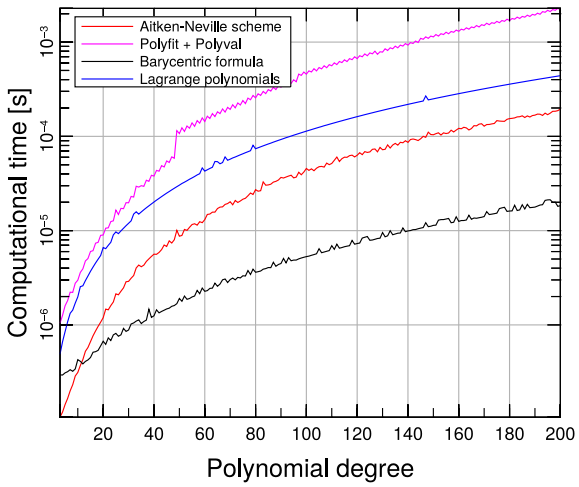


Fig. 172

C++-code 5.2.39: Timing polynomial evaluations

```
#include <cmath>
#include <vector>
#include <Eigen/Dense>
#include <figure/figure.hpp>
// ----- include timer library
#include "timer.h"
// ----- includes for Interpolation functions
#include "ANipoleval.hpp"
#include "ipolyeval.hpp"
#include "intpolyval.hpp"
#include "intpolyval_lag.hpp"

/**
 * Benchmarking 4 different interpolation attempts:
 * - AitkenNeville      - Barycentric formula
 * - Polyfit + Polyval  - Lagrange polynomials
 **/
int main() {
  // function to interpolate
  auto f = [](const Eigen::VectorXd& x){ return x.cwiseSqrt(); };

  const unsigned min_deg = 3, max_deg = 200;

  Eigen::VectorXd buffer;
  std::vector<double> t1, t2, t3, t4, N;

  // Number of repeats for each eval
  const int repeats = 100;

  // n = increasing polynomial degree
  for (unsigned n = min_deg; n <= max_deg; n++) {

    const Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(n, 1, n),
                          y = f(t);

    // ANipoleval takes a double as argument
    const double x = n*drand48(); // drand48 returns random double ∈ [0,1]
    // all other functions take a vector as argument
    const Eigen::VectorXd xv = n*Eigen::VectorXd::Random(1);

    std::cout << "Degree = " << n << "\n";
    Timer aitken, ipol, intpol, intpol_lag;

    // do the same many times and choose the best result
    // Aitken-Neville ----------------------
    aitken.start();
    for (unsigned i = 0; i < repeats; ++i){
```

```
48        ANipoleval(t, y, x); aitken.lap(); }
49      t1.push_back(aitken.min());
50
51      // Polyfit + Polyval ---------------------
52      ipol.start();
53      for (unsigned i = 0; i < repeats; ++i) {
54        ipoleval(t, y, xv, buffer); ipol.lap(); }
55      t2.push_back(ipol.min());
56
57      // Barycentric formula -------------------
58      intpol.start();
59      for (unsigned i = 0; i < repeats; ++i) {
60        intpolyval(t, y, xv, buffer); intpol.lap(); }
61      t3.push_back(intpol.min());
62
63      // Lagrange polynomials -------------------
64      intpol_lag.start();
65      for (unsigned i = 0; i < repeats; ++i) {
66        intpolyval_lag(t, y, xv, buffer); intpol_lag.lap(); }
67      t4.push_back(intpol_lag.min());
68
69      N.push_back(n);
70      }
71
72    mgl::Figure fig;
73    fig.setlog(false, true);
74    fig.title("Timing for single-point evaluation");
75    fig.plot(N, t1, "r").label("Aitken-Neville scheme");
76    fig.plot(N, t2, "m").label("Polyfit + Polyval");
77    fig.plot(N, t3, "k").label("Barycentric formula");
78    fig.plot(N, t4, "b").label("Lagrange polynomials");
79    fig.xlabel("Polynomial degree");
80    fig.ylabel("Computational time [s]");
81    fig.legend(0,1);
82    fig.save("pevaltime.eps");
83
84    return 0;
85  }
```

This uses functions given in Code 5.2.32, Code 5.2.35 and the function `polyfit` (with a clearly greater computational effort !)

`polyfit` is the equivalent to MATLAB's built-in `polyfit`. The implementation can be found on GitLab.

C++-code 5.2.40: Polynomial evaluation using `polyfit`

```
1  # include <Eigen/Dense>
2  # include "polyfit.hpp"
3  # include "polyval.hpp"
4
5  using Eigen::VectorXd;
6  /* Evaluation of the interpolation polynomials with polyfit+polyval
7   * IN:  t = nodes
8   *      y = values in t
9   *      x = evaluation points
10  *      v will be used to save the values of interpolant in x      */
11 void ipoleval(const VectorXd& t, const VectorXd& y, const VectorXd& x, VectorXd& v) {
12   // get coefficients using polyfit
13   VectorXd p = polyfit(t, y, y.size() - 1);
14   // evaluate using polyval (<-> horner scheme)
15   polyval(p, x, v);
16 }
```

C++-code 5.2.41: Lagrange polynomial interpolation and evaluation

```
2  // Evaluation of the interpolation polynomials with Lagrange polynomials
3  // IN: t = vector of interpolation nodes
4  // y = values in t
5  // x = evaluation points
```

```
// OUT p will be used to save the values of the interpolant evaluated in x
void intpolyval_lag(const VectorXd& t, const VectorXd& y, const VectorXd& x, VectorXd& p) {
  p = VectorXd::Zero(x.size());
  for (unsigned k = 0; k < t.size(); ++k) {
    // Compute values of k-th Lagrange polynomial in evaluation points
    VectorXd L; lagrangepoly(x, k, t, L);
    p += y(k)*L;
  }
}
```

5.2.3.3 Extrapolation to zero

Extrapolation is the same as interpolation but the evaluation point t is outside the interval $[\inf_{j=0,\ldots,n} t_j, \sup_{j=0,\ldots,n} t_j]$. In the sequel we assume $t = 0$, $t_i > 0$.

Of course, Lagrangian interpolation can also be used for extrapolation. In this section we give a very important application of this "Lagrangian extrapolation".

Task: compute the limit $\lim_{h\to 0} \psi(h)$ with prescribed accuracy, though the evaluation of the function $\psi = \psi(h)$ (maybe given in procedural form only) for very small arguments $|h| \ll 1$ is difficult, usually because of numerically instability (→ Section 1.5.5).

The extrapolation technique introduced below works well, if

- ψ is an *even function* of its argument: $\psi(t) = \psi(-t)$,
- $\psi = \psi(h)$ behaves "nicely" around $h = 0$.

Theory: existence of an asymptotic expansion in h^2

$$f(h) = f(0) + A_1 h^2 + A_2 h^4 + \cdots + A_n h^{2n} + R(h) \quad , \quad A_k \in \mathbb{R} \, ,$$

with remainder estimate $|R(h)| = O(h^{2n+2})$ for $h \to 0$.

Idea: computing inaccessible limit by extrapolation to zero

① Pick $h_0, \ldots, h_n$ for which ψ can be evaluated "safely".
② evaluation of $\psi(h_i)$ for different h_i, $i = 0, \ldots, n$, $|t_i| > 0$.
③ $\psi(0) \approx p(0)$ with interpolating polynomial $p \in \mathcal{P}_n$, $p(h_i) = \psi(h_i)$.

(5.2.43) Numerical differentiation through extrapolation

In Ex. 1.5.45 we have already seen a situation, where we wanted to compute the limit of a function $\psi(h)$ for $h \to 0$, but could not do it with sufficient accuracy. In this case $\psi(h)$ was a one-sided difference quotient with span h, meant to approximate $f'(x)$ for a differentiable function f. The cause of numerical difficulties was cancellation → § 1.5.43.

Now we will see how to dodge cancellation in difference quotients and how to use extrapolation to zero to computes derivatives with high accuracy:

Given: smooth function $f : I \subset \mathbb{R} \mapsto \mathbb{R}$ in procedural form: `function y = f(x)`

Sought: (approximation of) $f'(x)$, $x \in I$.

Natural idea: approximation of derivative by (symmetric) difference quotient

$$\frac{df}{dx}(x) \approx \frac{f(x+h) - f(x-h)}{2h} \,. \tag{5.2.44}$$

straightforward implementation fails due to cancellation in the numerator, see also Ex. 1.5.45.

C++-code 5.2.45: Numeric differentiation through difference quotients

```
// numerical differentiation using difference quotient
// f'(x) = lim_{h→0} (f(x+h)-f(x))/h
// IN: f (function object) = function to derive
// df = exact derivative (to compute error),
// name = string of function name (for plot filename)
// OUT: plot of error will be saved as "<name>numdiff.eps"
template <class Function, class Derivative>
void diff(const double x, Function& f, Derivative& df, const
    std::string name) {
  std::vector<long double> error, h;
  // build vector of widths of difference quotients
  for (int i = -1; i >= -61; i -= 5) h.push_back(std::pow(2,i));
  for (unsigned j = 0; j < h.size(); ++j) {
    // compute approximate solution using difference quotient
    double df_approx = (f(x + h[j]) - f(x))/h[j];
    // compute relative error
    double rel_err = std::abs((df_approx - df(x))/df(x));
    error.push_back(rel_err);
  }
```

$f(x) = \arctan(x)$

h	Relative error
2^{-1}	0.20786640808609
2^{-6}	0.00773341103991
2^{-11}	0.00024299312415
2^{-16}	0.00000759482296
2^{-21}	0.00000023712637
2^{-26}	0.00000001020730
2^{-31}	0.00000005960464
2^{-36}	0.00000679016113

$f(x) = \sqrt{x}$

h	Relative error
2^{-1}	0.09340033543136
2^{-6}	0.00352613693103
2^{-11}	0.00011094838842
2^{-16}	0.00000346787667
2^{-21}	0.00000010812198
2^{-26}	0.00000001923506
2^{-31}	0.00000001202188
2^{-36}	0.00000198842224

$f(x) = \exp(x)$

h	Relative error
2^{-1}	0.29744254140026
2^{-6}	0.00785334954789
2^{-11}	0.00024418036620
2^{-16}	0.00000762943394
2^{-21}	0.00000023835113
2^{-26}	0.00000000429331
2^{-31}	0.00000012467100
2^{-36}	0.00000495453865

Recall the considerations elaborated in Ex. 1.5.45. Owing to the impact of roundoff errors amplified by cancellation, $h \to 0$ does not achieve arbitrarily high accuracy. Rather, we observe fewer correct digits for very small h!

Extrapolation offers a numerically stable (→ Def. 1.5.85) alternative, because for a $2(n+1)$-times continuously differentiable function $f : I \subset \mathbb{R} \mapsto \mathbb{R}$, $x \in I$ we find that the symmetric difference quotient behaves like a polynomial in h^2 in the vicinity of $h = 0$. Consider Taylor sum of f in x with Lagrange

remainder term:

$$\psi(h) := \frac{f(x+h)-f(x-h)}{2h} \sim f'(x) + \sum_{k=1}^{n} \frac{1}{(2k)!} \frac{\mathrm{d}^{2k} f}{\mathrm{d}x^{2k}}(x)\, h^{2k} + \frac{1}{(2n+2)!} f^{(2n+2)}(\xi(x)) \,.$$

Since $\lim_{h\to 0} \psi(h) = f'(x)$ ➙ approximate $f'(x)$ by interpolation of ψ in points h_i.

C++-code 5.2.46: Numerical differentiation by extrapolation to zero

```
2  // Extrapolation based numerical differentation
3  // with a posteriori error control
4  // f:  handle of a function defined in a neighbourhood of x ∈ ℝ
5  // x:  point at which approximate derivative is desired
6  // h0:  initial distance from x
7  // rtol:  relative target tolerance, atol:  absolute tolerance
8  template <class Function>
9  double diffex(Function& f, const double x, const double h0, const
       double rtol, const double atol) {
10   const unsigned nit = 10; // Maximum depth of extrapolation
11   VectorXd h(nit); h(0) = h0; // Widths of difference quotients
12   VectorXd y(nit); // Approximations returned by difference quotients
13   y(0) = (f(x + h0) - f(x - h0))/(2*h0); // Widest difference quotients
14
15   // using Aitken-Neville scheme with x = 0, see Code 5.2.35
16   for (unsigned i = 1; i < nit; ++i) {
17     // create data points for extrapolation
18     h(i) = h(i-1)/2; // Next width half a big
19     y(i) = ( f(x + h(i)) - f(x - h(i)) )/h(i - 1);
20     // Aitken-Neville update
21     for (int k = i - 1; k >= 0; --k)
22       y(k) = y(k+1) - (y(k+1)-y(k))*h(i)/(h(i)-h(k));
23     // termination of extrapolation when desired tolerance is reached
24     const double errest = std::abs(y(1)-y(0)); // error indicator
25     if ( errest < rtol*std::abs(y(0)) || errest < atol ) //
26       break;
27   }
28   return y(1);
29 }
```

While the extrapolation table (→ § 5.2.36) is computed, more and more accurate approximations of $f'(x)$ become available. Thus, the difference between the two last approximations can be used to gauge the error of the current approximation, it provides an error indicator, which can be used to decide when the level of extrapolation is sufficient, see Line 25.

```
auto f = [](double x) {return std::atan(x);}
diffex(f,1.1,0.5)
```

Degree	Relative error
0	0.04262829970946
1	0.02044767428982
2	0.00051308519253
3	0.00004087236665
4	0.00000048930018
5	0.00000000746031
6	0.00000000001224

```
auto g = [](double x) {return std::sqrt(x);}
diffex(g,1.1,0.5)
```

Degree	Relative error
0	0.02849215135713
1	0.01527790811946
2	0.00061205284652
3	0.00004936258481
4	0.00000067201034
5	0.00000001253250
6	0.00000000004816
7	0.00000000000021

```
auto h = [](double x) {return std::exp(x);}
diffex(h,1.1,0.5)
```

Degree	Relative error
0	0.04219061098749
1	0.02129207652215
2	0.00011487434095
3	0.00000825582406
4	0.00000000589624
5	0.00000000009546
6	0.00000000000002

Advantage: guaranteed accuracy → efficiency

5.2.3.4 Newton basis and divided differences

Supplementary reading. We also refer to [**?**, Sect. 8.2.4], [**?**, Sect. 8.2].

In § 5.2.33 we have seen a method to evaluate partial polynomial interpolants for a single or a few evaluation points efficiently. Now we want to do this for *many* evaluation points that may not be known when we receive information about the first interpolation points.

C++code 5.2.47: Polynomial evaluation

```
class PolyEval {
  private:
    // evaluation point and various internal data describing the
       polynomials
  public:
    // Idle Constructor
    PolyEval();
    // Add another data point and update internal information
    void addPoint(t,y);
    // Evaluation of current interpolating polynomial at x
    Eigen::VectorXd operator () (const Eigen::VectorXd &x) const;
};
```

The challenge: Both `addPoint()` and the evaluation operator may be called many times and the implementation has to remain efficient under these circumstances.

Why not use the techniques from § 5.2.27? Drawback of the Lagrange basis or barycentric formula: adding another data point affects *all* basis polynomials/all precomputed values!

(5.2.48) Newton basis for $\mathcal{P}_n$

Our tool now: "update friendly" representation: Newton basis for $\mathcal{P}_n$

$$N_0(t) := 1\,, \quad N_1(t) := (t-t_0)\,, \quad \ldots \quad , \quad N_n(t) := \prod_{i=0}^{n-1}(t-t_i)\,. \tag{5.2.49}$$

Note: $N_n \in \mathcal{P}_n$ with *leading coefficient* 1 ➢ linear indepdence ➢ basis property.

The abstract considerations of § 5.1.13 still apply and we get a linear system of equations for the coefficients a_j of the polynomial interpolant in Newton basis:

$$a_j \in \mathbb{R}: \quad a_0 N_0(t_j) + a_1 N_1(t_j) + \cdots + a_n N_n(t_j) = y_j, \quad j = 0,\ldots,n\,.$$

$\Leftrightarrow$ triangular linear system

$$\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & (t_1-t_0) & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 1 & (t_n-t_0) & \cdots & \prod_{i=0}^{n-1}(t_n-t_i) \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Solution of the system with forward substitution:

$$\begin{aligned} a_0 &= y_0\,, \\ a_1 &= \frac{y_1 - a_0}{t_1 - t_0} = \frac{y_1 - y_0}{t_1 - t_0}\,, \\ a_2 &= \frac{y_2 - a_0 - (t_2-t_0)a_1}{(t_2-t_0)(t_2-t_1)} = \frac{y_2 - y_0 - (t_2-t_0)\frac{y_1-y_0}{t_1-t_0}}{(t_2-t_0)(t_2-t_1)} = \frac{\frac{y_2-y_0}{t_2-t_0} - \frac{y_1-y_0}{t_1-t_0}}{t_2-t_1}\,, \\ &\vdots \end{aligned}$$

Observation: same quantities computed again and again !

(5.2.50) Divided differences

In order to reveal the pattern, we turn to a new interpretation of the coefficients a_j of the interpolating polynomials in Newton basis.

> Newton basis polynomial $N_j(t)$: degree j and leading coefficient 1
> $\Rightarrow$ a_j is the leading coefficient of the interpolating polynomial $p_{0,j}$.

(the notation $p_{\ell,m}$ for partial polynomial interpolants through the data points $(t_\ell, y_\ell), \ldots, (t_m, y_m)$ was introduced in Section 5.2.3.2, see (5.2.34))

➢ Recursion (5.2.34) implies a recursion for the leading coefficients $a_{\ell,m}$ of the interpolating polynomials $p_{\ell,m}$, $0 \le \ell \le m \le n$:

$$a_{\ell,m} = \frac{a_{\ell+1,m} - a_{\ell,m-1}}{t_m - t_\ell}\,. \tag{5.2.51}$$

Hence, instead of using elimination for a triangular linear system, we find a simpler and more efficient algorithm using the so-called divided differences:

$$\begin{aligned} y[t_i] &= y_i \\ y[t_i,\ldots,t_{i+k}] &= \frac{y[t_{i+1},\ldots,t_{i+k}] - y[t_i,\ldots,t_{i+k-1}]}{t_{i+k}-t_i} \quad \text{(recursion)} \end{aligned} \tag{5.2.52}$$

(5.2.53) Efficient computation of divided differences

Recursive calculation by divided differences scheme, *cf.* Aitken-Neville scheme, Code 5.2.35:

$$\begin{array}{c|ccccc} t_0 & y[t_0] & & & & \\ & & > & y[t_0,t_1] & & \\ t_1 & y[t_1] & & & > & y[t_0,t_1,t_2] \\ & & > & y[t_1,t_2] & & & > & y[t_0,t_1,t_2,t_3], \\ t_2 & y[t_2] & & & > & y[t_1,t_2,t_3] \\ & & > & y[t_2,t_3] & & \\ t_3 & y[t_3] & & & & \end{array} \tag{5.2.54}$$

The elements can be computed from left to right, every "$>$" indicates the evaluation of the recursion formula (5.2.52).

However, we can again resort to the idea of § 5.2.36 and traverse (5.2.54) along the diagonals from top to bottom: If a new datum (t_{n+1}, y_{n+1}) is added, it is enough to compute the $n+2$ new terms

$$y[t_{n+1}],\ y[t_n,t_{n+1}],\ \ldots, y[t_0,\ldots,t_{n+1}]\ .$$

The following MATLAB code computes divided differences for data points (t_i, y_i), $i = 0,\ldots,n$, in this fashion. It is implemented by recursion to elucidate the successive use of data points. The divided differences $y[t_0], y[t_0,t_1], \ldots, y[t_0,\ldots,t_n]$, are accumulated in the vector $\mathtt{y}$.

C++-code 5.2.55: Divided differences, recursive implementation, in situ computation

```
// IN: t = node set (mutually different)
// y = nodal values
// OUT: y = coefficients of polynomial in Newton basis
void divdiff(const VectorXd& t, VectorXd& y) {
  const unsigned n = y.size() - 1;
  // Follow scheme (5.2.54), recursion (5.2.51)
  for (unsigned l = 0; l < n; ++l)
    for (unsigned j = l; j < n; ++j)
      y(j+1) = (y(j+1)-y(l))/(t(j+1)-t(l));
}
```

By derivation: computed finite differences are the coefficients of interpolating polynomials in Newton basis:

$$p(t) = a_0 + a_1(t-t_0) + a_2(t-t_0)(t-t_1) + \cdots + a_n \prod_{j=0}^{n-1}(t-t_j) \tag{5.2.56}$$

$$a_0 = y[t_0],\ a_1 = y[t_0,t_1],\ a_2 = y[t_0,t_1,t_2],\ \ldots.$$

Thus, Code 5.2.55 computes the coefficients a_j, $j = 0,\ldots,n$, of the polynomial interpolant with respect to the Newton basis. It uses only the first $j+1$ data points to find a_j.

(5.2.57) Efficient evaluation of polynomial in Newton form

"Backward evaluation" of $p(t)$ in the spirit of Horner's scheme (→ Rem. 5.2.5, [**?**, Alg. 8.20]):

$$p \leftarrow a_n,\quad p \leftarrow (t-t_{n-1})p + a_{n-1},\quad p \leftarrow (t-t_{n-2})p + a_{n-2},\quad \ldots.$$

C++-code 5.2.58: Divided differences evaluation by modified Horner scheme

```
2  // Evaluation of polynomial in Newton basis (divided differences)
3  // IN: t = nodes (mutually different)
4  // y = values in t
5  // x = evaluation points (as Eigen::Vector)
6  // OUT: p = values in x */
7  void evaldivdiff(const VectorXd& t, const VectorXd& y, const
       VectorXd& x, VectorXd& p) {
8    const unsigned n = y.size() - 1;
9
10   // get Newton coefficients of polynomial (non in-situ
       implementation!)
11   VectorXd coeffs; divdiff(t, y, coeffs);
12
13   // evaluate
14   VectorXd ones = VectorXd::Ones(x.size());
15   p = coeffs(n)*ones;
16   for (int j = n - 1; j >= 0; --j) {
17     p = (x - t(j)*ones).cwiseProduct(p) + coeffs(j)*ones;
18   }
19 }
```

Computational effort:

- $O(n^2)$ for computation of divided differences ("setup phase"),
- $O(n)$ for every single evaluation of $p(t)$.

(both operations can be interleaved, see Code 5.2.47)

Example 5.2.59 (Class `PolyEval`)

Implementation of a C++ class supporting the efficient update and evaluation of an interpolating polynomial making use of

- presentation in Newton basis (5.2.49),
- computation of representation coefficients through divided difference scheme (5.2.54), see Code 5.2.55,
- evaluation by means of Horner scheme, see Code 5.2.58.

C++-code 5.2.60: Definition of a class for "update friendly" polynomial interpolant

```
class PolyEval {
private:
  std::vector<double> t;  // Interpolation nodes
  std::vector<double> y;  // Coefficients in Newton representation
public:
  PolyEval(); // Idle constructor
  void addPoint(double t, double y); // Add another data point
  // evaluate value of current interpolating polynomial at x,
  double operator() (double x) const;
private:
  // Update internal representation, called by addPoint()
  void divdiff();
};
```

C++-code 5.2.61: Implementation of class PolyEval

```
PolyEval::PolyEval() {}

void PolyEval::addPoint(double td, double yd) {
  t.push_back(td); y.push_back(yd); divdiff();
}

void PolyEval::divdiff() {
  int n = t.size();
  for(int j=0; j<n-1; j++) y[n-1] = ((y[n-1]-y[j])/(t[n-1]-t[j]));
}

double PolyEval::operator() (double x) const {
  double s = y.back();
  for(int i = y.size()-2; i>= 0; --i) s = s*(x-t[i])+y[i];
  return s;
}
```

Remark 5.2.62 (Divided differences and derivatives)

If $y_0, \ldots, y_n$ are the values of a smooth function f in the points $t_0, \ldots, t_n$, that is, $y_j := f(t_j)$, then

$$y[t_i, \ldots, t_{i+k}] = \frac{f^{(k)}(\xi)}{k!}$$

for a certain $\xi \in [t_i, t_{i+k}]$, see [**?**, Thm. 8.21].

5.2.4 Polynomial Interpolation: Sensitivity

Supplementary reading. For related discussions see [**?**, Sect. 8.1.3].

This section addresses a *major shortcoming of polynomial interpolation* in case the interpolation knots t_i are imposed, which is usually the case when given data points have to be interpolated, *cf.* Ex. 5.1.5.

Example 5.2.63 (Oscillating polynomial interpolant (Runge's counterexample) → [?, Sect. 8.3], [?, Ex. 8.1])

This example offers a glimpse of the problems haunting polynomial interpolation.

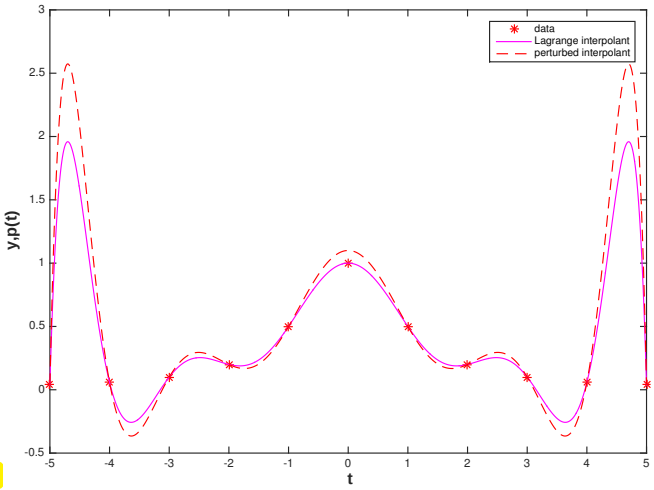
We examine the polynomial Lagrange interpolant (→ Section 5.2.2, (5.2.9)) for uniformly spaced nodes and the following data:

$$\mathcal{T} := \left\{ -5 + \tfrac{10}{n} j \right\}_{j=0}^{n},$$

$$y_j = \frac{1}{1+t_j^2}, \quad j = 0, \ldots n.$$

Plotted is the interpolant for $n = 10$ and the interpolant for which the data value at $t = 0$ has been perturbed by 0.1
(See also Ex. 6.1.41 below.)

➞ Fig. 173



☞ possible strong oscillations of interpolating polynomials of *high degree* on *uniformly spaced* nodes!

☞ Slight perturbations of data values can engender strong variations of a *high-degree* Lagrange interpolant "far away".

(5.2.64) Problem map and sensitivity

In Section 2.2.2 we introduced the concept of sensitivity to describe how perturbations of the data affect the output for a problem map as defined in § 1.5.67. Concretely, in Section 2.2.2 we discussed the sensitivity for linear systems of equations. Motivated by Ex. 5.2.63, we now examine the sensitivity of Lagrange interpolation with respect to perturbations in the data values.

The problem: data ↔ values y_i, $i = 0, \ldots, n$ ➢ data space $\mathbb{R}^{n+1}$.
problem map ↔ polynomial interpolation mapping $I_{\mathcal{T}}$, see Cor. 5.2.15, result space $\mathcal{P}_n$.

Thus, the (pointwise) sensitivity of polynomial interpolation will tell us to what extent perturbations in the y-data will affect the values of the interpolating function somewhere else. In the case of high sensitivity small perturbations in the data can cause big variations in some function values, which is clearly undesirable.

(5.2.65) Norms on spaces of functions

Necessary for studying sensitivity of polynomial interpolation in quantitative terms are norms (→ Def. 1.5.70) on the vector space of continuous functions $C(I)$, $I \subset \mathbb{R}$. The following norms are the most relevant:

$$\text{supremum norm} \quad \|f\|_{L^\infty(I)} := \sup\{|f(t)|: t \in I\} , \tag{5.2.66}$$

$$L^2\text{-norm} \quad \|f\|^2_{L^2(I)} := \int_I |f(t)|^2 \,\mathrm{d}t , \tag{5.2.67}$$

$$L^1\text{-norm} \quad \|f\|_{L^1(I)} := \int_I |f(t)| \,\mathrm{d}t . \tag{5.2.68}$$

Note the relationship with the vector norms introduced in § 1.5.69.

(5.2.69) Sensitivity of linear problem maps

In § 5.1.13 we have learned that (polynomial) interpolation gives rise to a linear problem map, see Def. 5.1.17. For this class of problem maps the investigation of sensitivity has to study operator norms, a generalization of matrix norms (→ Def. 1.5.76).

Let $\mathsf{L} : X \to Y$ be a *linear* problem map between two normed spaces, the data space X (with norm $\|\cdot\|_X$) and the result space Y (with norm $\|\cdot\|_Y$). Thanks to linearity, perturbations of the result $\mathbf{y} := \mathsf{L}(\mathbf{x})$ for the input $\mathbf{x} \in X$ can be expressed as follows:

$$\mathsf{L}(\mathbf{x} + \delta\mathbf{x}) = \mathsf{L}(\mathbf{x}) + \mathsf{L}(\delta\mathbf{x}) = \mathbf{y} + \mathsf{L}(\delta\mathbf{x}) .$$

Hence, the sensitivity (in terms of propagation of absolute errors) can be measured by the operator norm

$$\|\mathsf{L}\|_{X\to Y} := \sup_{\delta\mathbf{x}\in X\setminus\{\mathbf{0}\}} \frac{\|\mathsf{L}(\delta\mathbf{x})\|_Y}{\delta\|\mathbf{x}\|_X} . \tag{5.2.70}$$

This can be read as the "matrix norm of L", *cf.* Def. 1.5.76.

It seems challenging to compute the operator norm (5.2.70) for $\mathsf{L} = \mathsf{I}_\mathcal{T}$ ($\mathsf{I}_\mathcal{T}$ the Lagrange interpolation operator for node set $\mathcal{T} \subset I$), $X = \mathbb{R}^{n+1}$ (equipped with a vector norm), and $Y = C(I)$ (endowed with a norm from § 5.2.65). The next lemma will provide surprisingly simple concrete formulas.

Lemma 5.2.71. Absolute conditioning of polynomial interpolation

Given a mesh $\mathcal{T} \subset \mathbb{R}$ with generalized Lagrange polynomials L_i, $i = 0,\ldots,n$, and fixed $I \subset \mathbb{R}$, the norm of the interpolation operator satisfies

$$\|\mathsf{I}_\mathcal{T}\|_{\infty\to\infty} := \sup_{\mathbf{y}\in\mathbb{R}^{n+1}\setminus\{0\}} \frac{\|\mathsf{I}_\mathcal{T}(\mathbf{y})\|_{L^\infty(I)}}{\|\mathbf{y}\|_\infty} = \left\|\sum_{i=0}^n |L_i|\right\|_{L^\infty(I)} , \tag{5.2.72}$$

$$\|\mathsf{I}_\mathcal{T}\|_{2\to 2} := \sup_{\mathbf{y}\in\mathbb{R}^{n+1}\setminus\{0\}} \frac{\|\mathsf{I}_\mathcal{T}(\mathbf{y})\|_{L^2(I)}}{\|\mathbf{y}\|_2} \le \left(\sum_{i=0}^n \|L_i\|^2_{L^2(I)}\right)^{\frac{1}{2}} . \tag{5.2.73}$$

Proof. (for the L^∞-Norm) By $\triangle$-inequality

$$\|\mathrm{I}_{\mathcal{T}}(\mathbf{y})\|_{L^\infty(I)} = \left\|\sum_{j=0}^n y_j L_j\right\|_{L^\infty(I)} \leq \sup_{t\in I} \sum_{j=0}^n |y_j||L_j(t)| \leq \|\mathbf{y}\|_\infty \left\|\sum_{i=0}^n |L_i|\right\|_{L^\infty(I)},$$

equality in (5.2.72) for $\mathbf{y} := (\mathrm{sgn}(L_j(t^*)))_{j=0}^n$, $t^* := \mathrm{argmax}_{t\in I} \sum_{i=0}^n |L_i(t)|$. □

Proof. (for the L^2-Norm) By $\triangle$-inequality and Cauchy-Schwarz inequality

$$\|\mathrm{I}_{\mathcal{T}}(\mathbf{y})\|_{L^2(I)} \leq \sum_{j=0}^n |y_j| \|L_j\|_{L^2(I)} \leq \left(\sum_{j=0}^n |y_j|^2\right)^{\frac{1}{2}} \left(\sum_{j=0}^n \|L_j\|_{L^2(I)}^2\right)^{\frac{1}{2}}.$$

□

Terminology: Lebesgue constant of $\mathcal{T}$: $\lambda_{\mathcal{T}} := \left\|\sum_{i=0}^n |L_i|\right\|_{L^\infty(I)} = \|\mathrm{I}_{\mathcal{T}}\|_{\infty\to\infty}$

Remark 5.2.74 (Lebesgue constant for equidistant nodes)

We consider Lagrange interpolation for the special setting

$$I = [-1,1], \quad \mathcal{T} = \{-1 + \tfrac{2k}{n}\}_{k=0}^n \quad \text{(uniformly spaced nodes)}.$$

Asymptotic estimate (with (5.2.11) and Stirling formula): for $n = 2m$

$$|L_m(1-\tfrac{1}{n})| = \frac{\frac{1}{n}\cdot\frac{1}{n}\cdot\frac{3}{n}\cdot\ldots\cdot\frac{n-3}{n}\cdot\frac{n+1}{n}\cdot\ldots\cdot\frac{2n-1}{n}}{\left(\frac{2}{n}\cdot\frac{4}{n}\cdot\ldots\cdot\frac{n-2}{n}\cdot 1\right)^2} = \frac{(2n)!}{(n-1)2^{2n}((n/2)!)^2 n!} \sim \frac{2^{n+3/2}}{\pi\,(n-1)\,n}$$

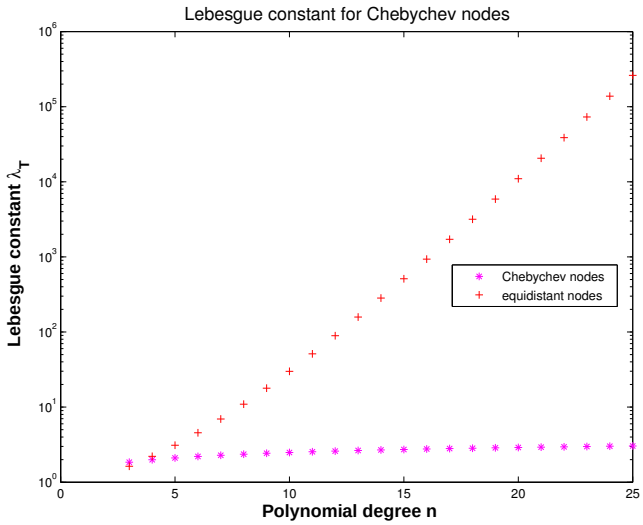


Fig. 174

Sophisticated theory [**?**] gives a lower bound for the Lebesgue constant for uniformly spaced nodes:

$$\lambda_{\mathcal{T}} \geq Ce^{n/2}$$

with $C > 0$ independent of n.

We can also perform a numerical evaluation of the expression

$$\lambda_{\mathcal{T}} = \left\|\sum_{i=0}^n |L_i|\right\|_{L^\infty(I)},$$

for the Lebesgue constant of polynomial interpolation, see Lemma 5.2.71.

C++-code 5.2.75: C++ code for *approximate* computation of Lebesgue constants

```
// Computation of Lebesgue constant of polynomial interpolation
// with knots t_i passed in the vector t based on (5.2.72).
// N specifies the number of sampling points for the approximate
// computation of the maximum norm of the Lagrange polynimial
// on the interval [-1,1].
double lebesgue(const VectorXd& t, const unsigned& N) {
  const unsigned n = t.size();
  // compute denominators of normalized Lagrange polynomials relative
     to the nodes t
  VectorXd den(n);
  for (unsigned i = 0; i < n; ++i) {
    VectorXd tmp(n - 1);
    // Note:  comma initializer can't be used with vectors of length 0
    if (i == 0) tmp = t.tail(n - 1);
    else if (i == n - 1) tmp = t.head(n - 1);
    else tmp << t.head(i), t.tail(n - (i + 1));
    den(i) = (t(i) - tmp.array()).prod();
  }

  double l = 0; // return value
  for (unsigned j = 0; j < N; ++j) {
    const double x = -1 + j*(2./N); // sampling point for ||·||_{L^∞([-1,1])}
    double s = 0;
    for (unsigned k = 0; k < n; ++k) {
      // v provides value of normalized Lagrange polynomials
      VectorXd tmp(n - 1);
      if (k == 0) tmp = t.tail(n - 1);
      else if (k == n - 1) tmp = t.head(n - 1);
      else tmp << t.head(k), t.tail(n - (k + 1));
      double v = (x - tmp.array()).prod()/den(k);
      s += std::abs(v); // sum over modulus of the polynomials
    }
    l = std::max(l, s); // maximum of sampled values
  }
  return l;
}
```

Note: In Code 5.2.75 the norm $\|L_i\|_{L^\infty(I)}$ can be computed only approximate by taking the maximum modulus of function values in many sampling points.

(5.2.76) Importance of knowing the sensitivity of polynomial interpolation

In Ex. 5.1.5 we learned that interpolation is an important technique for obtaining a mathematical (and algorithmic) description of a constitutive relationship from measured data.

If the interpolation operator is poorly conditioned, tiny measurement errors will lead to big (local) deviations of the interpolant from its "true" form.

Since measurement errors are inevitable, poorly conditioned interpolation procedures are useless for determining constitutive relationships from measurements.

5.3 Shape preserving interpolation

When reconstructing a quantitative dependence of quantities from measurements, first principles from physics often stipulate qualitative constraints, which translate into *shape properties* of the function f, e.g., when modelling the material law for a gas:

$$t_i \text{ pressure values, } y_i \text{ densities} \quad \succ \quad f \text{ positive \& monotonic.}$$

Notation: given data: $(t_i, y_i) \in \mathbb{R}^2$, $i = 0, \ldots, n$, $n \in \mathbb{N}$, $t_0 < t_1 < \cdots < t_n$.

Example 5.3.1 (Magnetization curves)

For many materials physics stipulates properties of the functional dependence of magnetic flux B from magnetic field strength H:

- $H \mapsto B(H)$ smooth (at least C^1),
- $H \mapsto B(H)$ monotonic (increasing),
- $H \mapsto B(H)$ concave

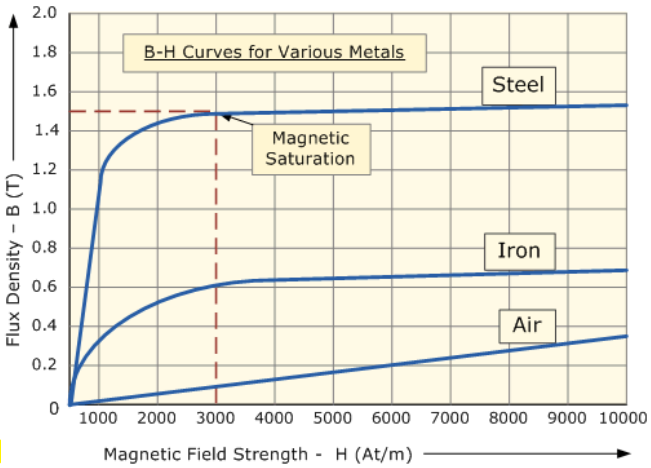


Fig. 175

5.3.1 Shape properties of functions and data

(5.3.2) The "shape" of data

The section is about "shape preservation". In the previous example we have already seen a few properties that constitute the "shape" of a function: sign, monotonicity and curvature. Now we have to identify analogous properties of data sets in the form of sequences of interpolation points (t_j, y_j), $j = 0, \ldots, n$, t_j pairwise distinct.

Definition 5.3.3. monotonic data

The data (t_i, y_i) are called monotonic when $y_i \geq y_{i-1}$ or $y_i \leq y_{i-1}$, $i = 1, \ldots, n$.

Definition 5.3.4. Convex/concave data

The data $\{(t_i, y_i)\}_{i=0}^n$ are called convex (concave) if

$$\Delta_j \overset{(\geq)}{\leq} \Delta_{j+1}\,, \quad j = 1,\ldots,n-1 \quad , \quad \Delta_j := \frac{y_j - y_{j-1}}{t_j - t_{j-1}}\,, \quad j = 1,\ldots,n\,.$$

Mathematical characterization of convex data:

$$y_i \leq \frac{(t_{i+1} - t_i)y_{i-1} + (t_i - t_{i-1})y_{i+1}}{t_{i+1} - t_{i-1}} \qquad \forall\, i = 1,\ldots,n-1,$$

i.e., each data point lies below the line segment connecting the other data, *cf.* definition of convexity of a function [**?**, Def. 5.5.2].

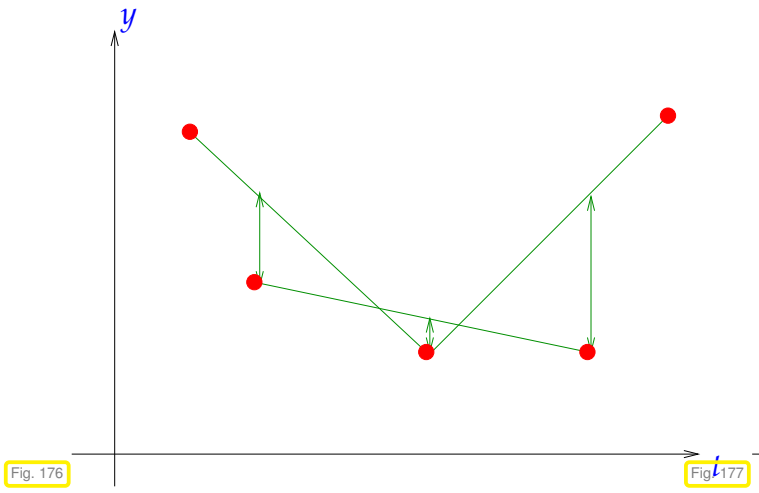


Fig. 176

Convex data

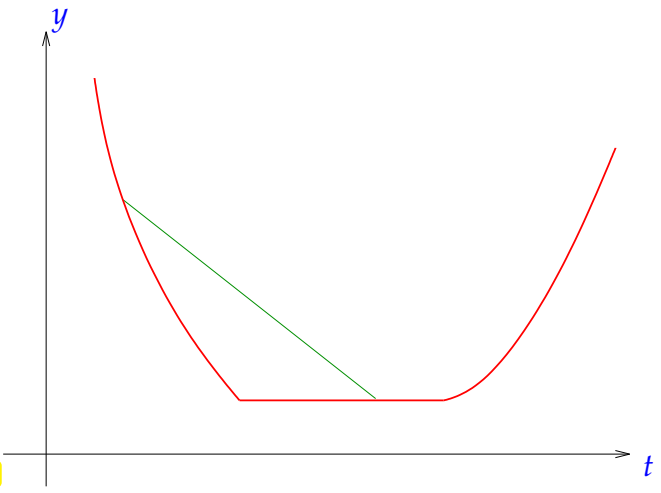


Fig. 177

Convex function

Definition 5.3.5. Convex/concave function → **[?, Def. 5.5.2]**

$$f : I \subset \mathbb{R} \mapsto \mathbb{R} \quad \begin{matrix}\text{convex}\\ \text{concave}\end{matrix} \quad :\Leftrightarrow \quad \begin{matrix} f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y) \\ f(\lambda x + (1-\lambda)y) \geq \lambda f(x) + (1-\lambda)f(y)\end{matrix} \quad \begin{matrix}\forall\, 0 \leq \lambda \leq 1\,,\\ \forall\, x,y \in I\,.\end{matrix}$$

(5.3.6) (Local) shape preservation

Now consider interpolation problem: data (t_i, y_i), $i = 0,\ldots,n$ $\longrightarrow$ interpolant f

Goal: shape preserving interpolation:

positive data	$\longrightarrow$	positive interpolant f,
monotonic data	$\longrightarrow$	monotonic interpolant f,
convex data	$\longrightarrow$	convex interpolant f.

More ambitious goal: local shape preserving interpolation: for each subinterval $I = (t_i, t_{i+j})$

positive data in I	$\longrightarrow$	locally positive interpolant $f\|_I$,
monotonic data in I	$\longrightarrow$	locally monotonic interpolant $f\|_I$,
convex data in I	$\longrightarrow$	locally convex interpolant $f\|_I$.

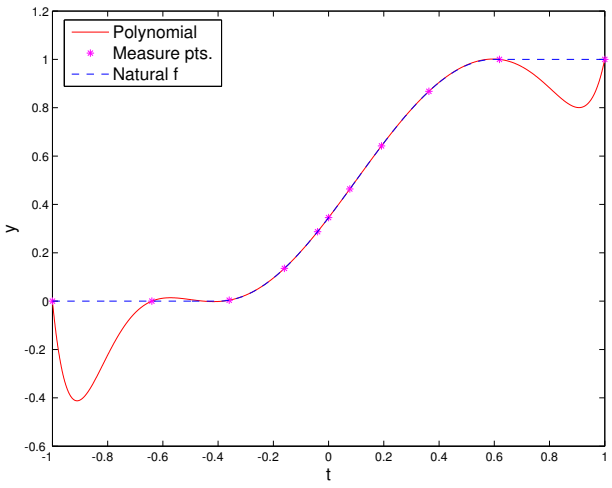
Experiment 5.3.7 (Bad behavior of global polynomial interpolants)

We perform Lagrange interpolation for the following positive and monotonic data:

t_i	-1.0000	-0.6400	-0.3600	-0.1600	-0.0400	0.0000	0.0770	0.1918	0.3631	0.6187	1.0000
y_i	0.0000	0.0000	0.0039	0.1355	0.2871	0.3455	0.4639	0.6422	0.8678	1.0000	1.0000

created by taking points on the graph of

$$f(t) = \begin{cases} 0 & \text{if } t < -\frac{2}{5}\,, \\ \frac{1}{2}(1+\cos(\pi(t-\frac{3}{5}))) & \text{if } -\frac{2}{5} < t < \frac{3}{5}\,, \\ 1 & \text{otherwise.} \end{cases}$$



← Interpolating polynomial, degree = 10

Oscillations at the endpoints of the interval (see Fig. 173)

- No locality
- No positivity
- No monotonicity
- No local conservation of the curvature

5.3.2 Piecewise linear interpolation

There is a very simple method of achieving perfect shape preservation by means of a linear (→ § 5.1.13) interpolation operator into the space of continuous functions:

Data: $(t_i, y_i) \in \mathbb{R}^2$, $i = 0, \ldots, n$, $n \in \mathbb{N}$, $t_0 < t_1 < \cdots < t_n$.

Then the piecewise linear interpolant $s : [t_0, t_n] \to \mathbb{R}$ is defined as, *cf.* Ex. 5.1.10:

$$s(t) = \frac{(t_{i+1} - t)y_i + (t - t_i)y_{i+1}}{t_{i+1} - t_i} \quad \text{for} \quad t \in [t_i, t_{i+1}]\,. \tag{5.3.8}$$

The piecewise linear interpolant is also called a polygonal curve. It is continuous and consists of n line segments.

Piecewise linear interpolant of data from Fig. 176 ▷

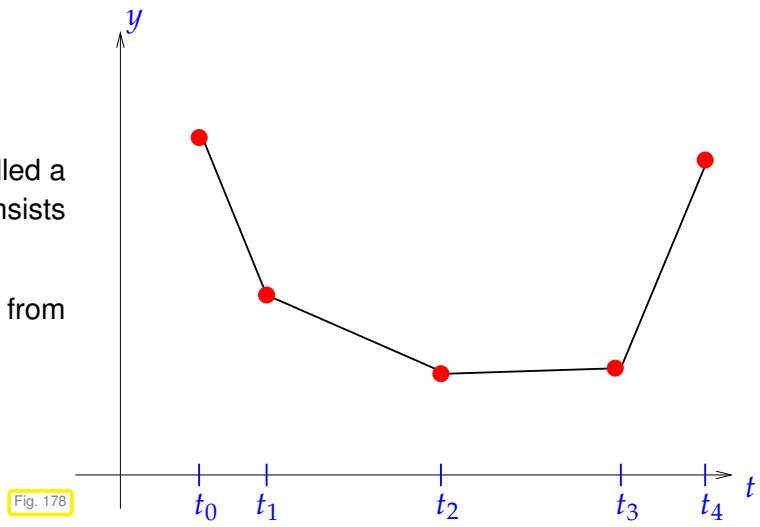


Fig. 178

Piecewise linear interpolation means simply "connect the data points in $\mathbb{R}^2$ using straight lines".

Obvious: linear interpolation is linear (as mapping $\mathbf{y} \mapsto s$, see Def. 5.1.17) and local in the following sense:

$$y_j = \delta_{ij}\,, \quad i,j = 0,\ldots,n \quad \Rightarrow \quad \operatorname{supp}(s) \subset [t_{i-1}, t_{i+1}]\,. \tag{5.3.9}$$

As obvious are the properties asserted in the following theorem. The local preservation of curvature is a straightforward consequence of Def. 5.3.4.

Theorem 5.3.10. Local shape preservation by piecewise linear interpolation

Let $s \in C([t_0,t_n])$ be the piecewise linear interpolant of $(t_i,y_i) \in \mathbb{R}^2$, $i = 0,\ldots,n$, for every subinterval $I = [t_j,t_k] \subset [t_0,t_n]$:

if $(t_i,y_i)|_I$ are positive/negative	$\Rightarrow$	*$s|_I$ is positive/negative,*
if $(t_i,y_i)|_I$ are monotonic (increasing/decreasing)	$\Rightarrow$	*$s|_I$ is monotonic (increasing/decreasing),*
if $(t_i,y_i)|_I$ are convex/concave	$\Rightarrow$	*$s|_I$ is convex/concave.*

Local shape preservation = perfect shape preservation!

Bad news: none of this properties carries over to local polynomial interpolation of higher polynomial degree $d > 1$.

Example 5.3.11 (Piecewise quadratic interpolation)

We consider the following generalization of piecewise linear interpolation of data points $(t_j,y_j) \in \mathbb{R} \times \mathbb{R}$, $j = 0,\ldots,n$.

From Thm. 5.2.14 we know that a parabola (polynomial of degree 2) is uniquely determined by 3 data points. Thus, the idea is to form groups of three adjacent data points and interpolate each of these triplets by a 2nd-degree polynomial (parabola).

Assume: $n = 2m$ even

◤ piecewise quadratic interpolant $q : [\min\{t_i\}, \max\{t_i\}] \mapsto \mathbb{R}$ is defined by

$$q_j := q_{|[t_{2j-2},t_{2j}]} \in \mathcal{P}_2 \quad , \quad q_j(t_i) = y_i\,, \quad i = 2j-2, 2j-1, 2j\,, \quad j = 1,\ldots,m\,. \tag{5.3.12}$$

Nodes as in Exp. 5.3.7

Piecewise linear (blue) and quadratic (red) interpolants ▷

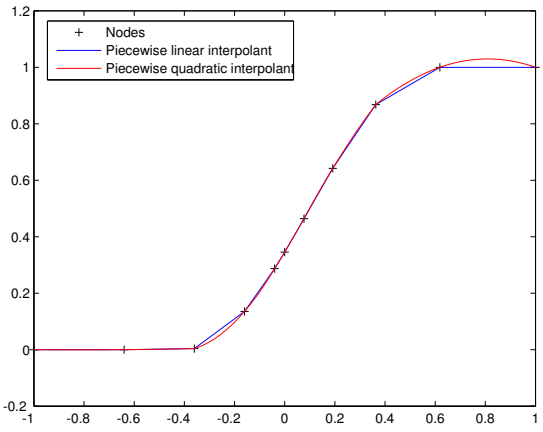


Fig. 179

No shape preservation for piecewise quadratic interpolant

The "only" drawback of piecewise linear interpolation:

interpolant is only C^0 but not C^1 (no continuous derivative).

However: Interpolant usually serves as input for other numerical methods like a Newton-method for solving non-linear systems of equations, see Section 8.4, which requires derivatives.

5.4 Cubic Hermite Interpolation

Aim: construct local shape-preserving (→ Section 5.3) (linear ?) interpolation operator that fixes shortcoming of piecewise linear interpolation by ensuring C^1-smoothness of the interpolant.

✎ notation: $C^1([a,b]) \triangleq$ space of *continuously differentiable* functions $[a,b] \mapsto \mathbb{R}$.

5.4.1 Definition and algorithms

Given: mesh points $(t_i, y_i) \in \mathbb{R}^2$, $i = 0,\ldots,n$, $t_0 < t_1 < \cdots < t_n$.

Goal: build function $f \in C^1([t_0,t_n])$ satisfying the interpolation conditions $f(t_i) = y_i$, $i = 0,\ldots,n$.

Definition 5.4.1. Cubic Hermite polynomial interpolant

Given data points $(t_j, y_j) \in \mathbb{R} \times \mathbb{R}$, $j = 0,\ldots,n$, with pairwise distinct ordered nodes t_j, and slopes $c_j \in \mathbb{R}$, the piecewise cubic Hermite interpolant $s : [t_0,t_n] \to \mathbb{R}$ is defined by the requirements

$$s_{|[t_{i-1},t_i]} \in \mathcal{P}_3\,, \quad i = 1,\ldots,n \quad , \quad \boxed{s(t_i) = y_i} \quad , \quad \boxed{s'(t_i) = c_i} \quad , \quad i = 0,\ldots,n\,.$$

Corollary 5.4.2. Smoothness of cubic Hermite polynomial interpolant

Piecewise cubic Hermite interpolants are continuously differentiable on their interval of definition.

Proof. The assertion of the corollary follows from the agreement of function values and first derivative values on nodes shared by two intervals, on each of which the piecewise cubic Hermite interpolant is a polynomial of degree 3.

□

(5.4.3) Local representation of piecewise cubic Hermite interpolant

Locally, we can write a piecewise cubic Hermite interpolant as a linear combination of generalized cardinal basis functions with coefficients supplied by the data values y_j and the slopes c_j:

$$s(t) = y_{i-1}H_1(t) + y_iH_2(t) + c_{i-1}H_3(t) + c_iH_4(t) \quad , \quad t \in [t_{i-1}, t_i] \, , \tag{5.4.4}$$

where the basic functions H_k, $k = 1,2,3,4$, are as follows:

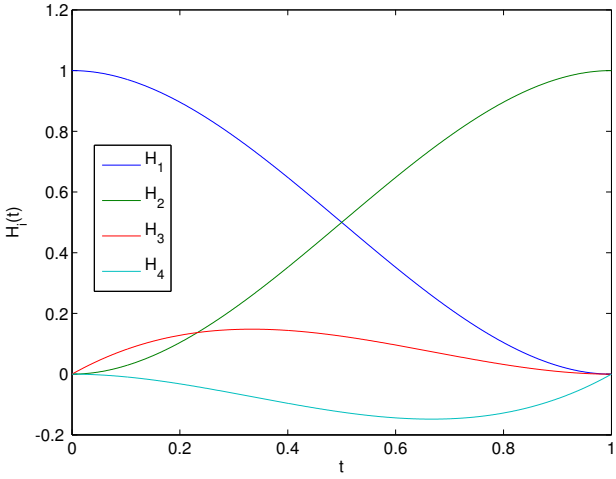


Fig. 180

$$\begin{aligned} H_1(t) &:= \phi(\tfrac{t_i - t}{h_i}) \, , \quad H_2(t) := \phi(\tfrac{t - t_{i-1}}{h_i}) \, , \\ H_3(t) &:= -h_i\psi(\tfrac{t_i - t}{h_i}) \, , \quad H_4(t) := h_i\psi(\tfrac{t - t_{i-1}}{h_i}) \, , \\ h_i &:= t_i - t_{i-1} \quad , \\ \phi(\tau) &:= 3\tau^2 - 2\tau^3 \quad , \\ \psi(\tau) &:= \tau^3 - \tau^2 \, . \end{aligned} \tag{5.4.5}$$

◁ Local basis polynomial on $[0,1]$

By tedious, but straightforward computations using the chain rule we find the following values for H_k and H_k' at the endpoints of the interval $[t_{i-1}, t_i]$.

	$H(t_{i-1})$	$H(t_i)$	$H'(t_{i-1})$	$H'(t_i)$
H_1	1	0	0	0
H_2	0	1	0	0
H_3	0	0	1	0
H_4	0	0	0	1

This amounts to a proof for (5.4.4) (why?).

The formula (5.4.4) is handy for the local evaluation of piecewise cubic Hermit interpolants. The function `hermloceval` in Code 5.4.6 performs the efficient evaluation (in multiple points) of a piecewise cubic polynomial s on t_1, t_2 uniquely defined by the constraints $s(t_1) = y_1, s(t_2) = y_2, s'(t_1) = c_1, s'(t_2) = c_2$:

C++11 code 5.4.6: Local evaluation of cubic Hermite polynomial

```
2 // Multiple point evaluation of Hermite polynomial
3 // y1, y2:  data values
4 // c1, c2:  slopes
5 VectorXd hermloceval(VectorXd t, double t1, double t2,
6                       double y1, double y2,
```

```
7                  double c1, double c2) {
8   const double h = t2-t1,a1 = y2-y1, a2 = a1-h*c1, a3 = h*c2-a1-a2;
9   t = ((t.array()-t1)/h).matrix();
10  return (y1+(a1+(a2+a3*t.array())*(t.array()-1))*t.array()).matrix();
11 }
```

(5.4.7) Linear Hermite interpolation

However, the data for an interpolation problem (→ Section 5.1) are merely the interpolation points (t_j, y_j), $j = 0, \ldots, n$, but not the slopes of the interpolant at the nodes. Thus, in order to define an interpolation operator into the space of piecewise cubic Hermite functions, we have supply a mapping $\mathbb{R}^{n+1} \times \mathbb{R}^{n+1} \to \mathbb{R}^{n+1}$ computing the slopes c_j from the data points.

Since this mapping should be local it is natural to rely on (weighted) averages of the local slopes Δ_j (→ Def. 5.3.4) of the data, for instance

$$c_i = \begin{cases} \Delta_1 & \text{, for } i = 0 \text{ ,} \\ \Delta_n & \text{, for } i = n \text{ ,} \\ \frac{t_{i+1}-t_i}{t_{i+1}-t_{i-1}}\Delta_i + \frac{t_i - t_{i-1}}{t_{i+1}-t_{i-1}}\Delta_{i+1} & \text{, if } 1 \le i < n \text{ .} \end{cases} \quad , \quad \Delta_j := \frac{y_j - y_{j-1}}{t_j - t_{j-1}} \text{ ,} j = 1, \ldots, n \text{ .} \tag{5.4.8}$$

Leads to a linear (→ Def. 5.1.17), local Hermite interpolation operator

"Local" means, that, if the values y_j are non-zero for only a few adjacent data points with indices $j = k, \ldots, k+m$, $m \in \mathbb{N}$ small, then the Hermite interpolant s is supported on $[t_{k-\ell}, t_{k+m+\ell}]$ for small $\ell \in \mathbb{N}$ independent of k and m.

Example 5.4.9 (Piecewise cubic Hermite interpolation)

Hermite interpolation

Data points:

- 11 equispaced nodes

$$t_j = -1 + 0.2\,j, \quad j = 0, \ldots, 10.$$

in the interval $I = [-1, 1]$,

- $y_i = f(t_i)$ with

$$f(x) := \sin(5x)\, e^x \,.$$

Here we used weighted averages of slopes as in Eq. (5.4.8).

For details see Code 5.4.10.

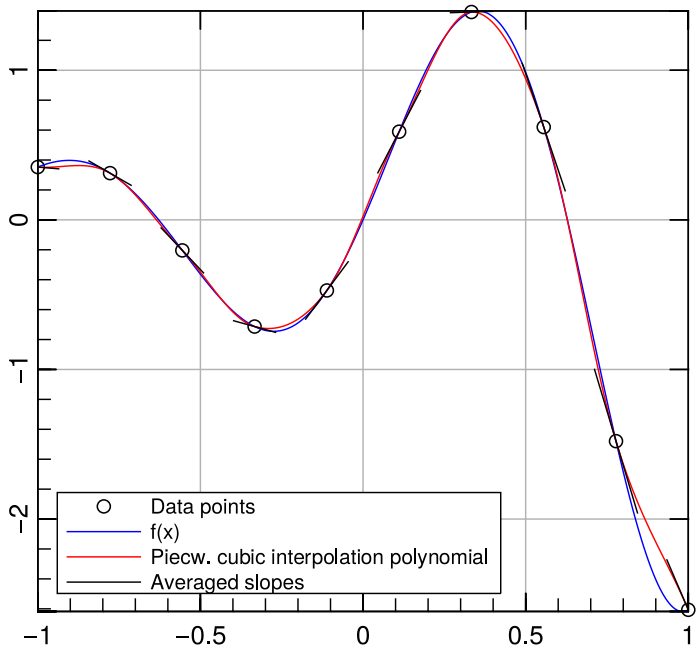


Fig. 181

No local/global preservation of monotonicity!

C++-code 5.4.10: Piecewise cubic Hermite interpolation

```
# include <Eigen/Dense>
# include <figure/figure.hpp>
# include "hermloceval.hpp"

using Eigen::VectorXd;

// Compute and plot the cubic Hermite interpolant of the function f in
    the nodes t
// using weighted averages according to (5.4.8) as local slopes
template <class Function>
void hermintp1(Function& f, const VectorXd& t) {
  const unsigned n = t.size();
  // length of intervals between nodes:  h_i := t_{i+1} - t_i
  const VectorXd h = t.tail(n - 1) - t.head(n - 1);

  // evaluate f at the nodes and for plotting
  VectorXd y(n),
           tplot = VectorXd::LinSpaced(500, t(0), t(n - 1)),
           yplot(500);
  for (unsigned i = 0; i < n; ++i) y(i) = f(t(i));
  for (unsigned j = 0; j < 500; ++j) yplot(j) = f(tplot(j));

  // slopes of piecewise linear interpolant
  const VectorXd delta = (y.tail(n - 1) - y.head(n - 1)).cwiseQuotient(h);
  VectorXd c(n);
  c(0) = delta(0); c(n - 1) = delta(n - 2);
  // slopes from weighted averages, see (5.4.8)
  for (unsigned i = 1; i < n - 1; ++i) {
    c(i) = ( h(i)*delta(i - 1) + h(i - 1)*delta(i) )/( t(i + 1) - t(i - 1) );
  }

  mgl::Figure fig;
  fig.title("Hermite interpolation");
  fig.legend(0, 0);
  fig.plot(t, y, " ko").label("Data points");
  fig.plot(tplot, yplot).label("f(x)");

```

```
37   // compute and plot the Hermite interpolant with slopes c
38   for (unsigned j = 0; j < n - 1; ++j) {
39     VectorXd vx = VectorXd::LinSpaced(100, t(j), t(j + 1)),
40              px;
41     hermloceval(vx, t(j), t(j+1), y(j), y(j+1), c(j), c(j+1), px);
42     fig.plot(vx, px, "r");
43   }
44   // manually adding label for pw hermite interpolants
45   fig.addlabel("Piecw. cubic interpolation polynomial", "r");
46
47   // plot segments indicating the slopes c_i
48   for (unsigned k = 1; k < n - 1; ++k) {
49     VectorXd t_sl(2), y_sl(2);
50     t_sl << t(k) - 0.3*h(k - 1), t(k) + 0.3*h(k);
51     y_sl << y(k) - 0.3*h(k - 1)*c(k), y(k) + 0.3*h(k)*c(k);
52     fig.plot(t_sl, y_sl, "k");
53   }
54   // slope segments at beginning and end
55   VectorXd t_sl_0(2), y_sl_0(2),
56            t_sl_n(2), y_sl_n(2);
57
58   t_sl_0 << t(0), t(0) + 0.3*h(0);
59   y_sl_0 << y(0), y(0) + 0.3*h(0)*c(0);
60   t_sl_n << t(n - 1) - 0.3*h(n - 2), t(n - 1);
61   y_sl_n << y(n - 1) - 0.3*h(n - 2)*c(n - 1), y(n - 1);
62   fig.plot(t_sl_0, y_sl_0, "k");
63   fig.plot(t_sl_n, y_sl_n, "k");
64
65   fig.addlabel("Averaged slopes", "k");
66
67   // save figure
68   fig.save("hermintp1");
69 }
```

Invocation:

```
auto f = [](double x)  return sin(5*x)*exp(x);
Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(10, -1, 1);
hermintp(f, t);
```

5.4.2 Local monotonicity preserving Hermite interpolation

From Ex. 5.4.9 we learn that, if the slopes are chosen according to Eq. (5.4.8), then he resulting Hermite interpolation does not preserve monotonicity.

Consider the situation sketched on the right ▷

The red circles (●) represent data points, the blue line (—) the piecewise linear interpolant → Section 5.3.2.

In the nodes marked with ↦ the first derivative of a monotonicity preserving C^1-smooth interpolant must vanish! Otherwise an "overshoot" occurs, see also Fig. 179.

Of course, this will be violated, when a (weighted) arithmetic average is used for the computation of slopes for cubic Hermite interpolation.

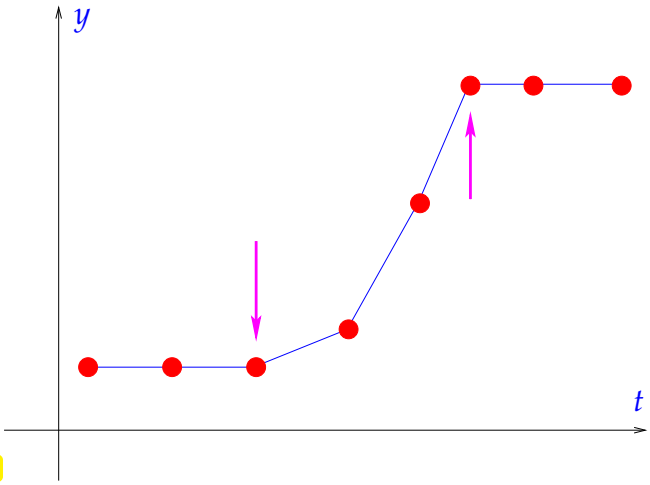


Fig. 182

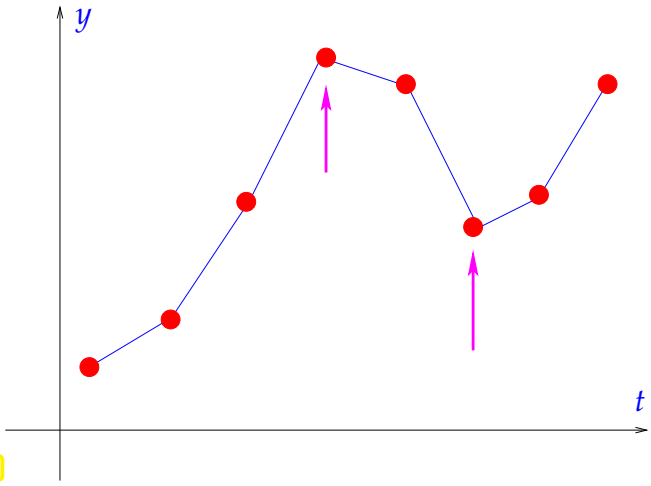


Fig. 183

◁ Consider the situation sketched on the left.

The red circles (●) represent data points, the blue line (—) the piecewise linear interpolant → Section 5.3.2.

A *local* monotonicity preserving C^1-smooth interpolant (→ § 5.3.6) s must be flat (= vanishing first derivative) in data points (t_j, y_j), for which

$$y_{j-1} < y_j \quad \text{and} \quad y_{j+1} < y_j \,, \\ y_{j-1} > y_j \quad \text{and} \quad y_{j+1} > y_j \,,$$

in "local extrema" of the data set.

Otherwise, overshoots or undershoots would destroy local monotonicity on one side of the extremum.

(5.4.11) Limiting of local slopes

From the discussion of Fig. 182 and Fig. 183 it is clear that local monotonicity preservation entails that the local slopes c_i of a cubic Hermite interpolant (→ Def. 5.4.1) have to fulfill

$$c_i = \begin{cases} 0 & \text{, if } \operatorname{sgn}(\Delta_i) \neq \operatorname{sgn}(\Delta_{i+1}) \,, \\ \text{some "average" of } \Delta_i, \Delta_{i+1} & \text{otherwise} \end{cases} \,, \quad i = 1, \ldots, n-1 \,. \tag{5.4.12}$$

✎ notation: sign function $\operatorname{sgn}(\xi) = \begin{cases} 1 & \text{, if } \xi > 0 \,, \\ 0 & \text{, if } \xi = 0 \,, \\ -1 & \text{, if } \xi < 0 \,. \end{cases}$

A slope selection rule that enforces (5.4.12) is called a limiter.

Of course, testing for equality with zero does not make sense for data that may be affected by measurement or roundoff errors. Thus, the "average" in (5.4.12) must be close to zero already when either $\Delta_i \approx 0$

or $\Delta_{i+1} \approx 0$. This is satisfied by the weighted harmonic mean

$$c_i = \frac{1}{\frac{w_a}{\Delta_i} + \frac{w_b}{\Delta_{i+1}}}\,, \tag{5.4.13}$$

with weights $w_a > 0$, $w_b > 0$, $(w_a + w_b = 1)$.

The harmonic mean = "smoothed $\min(\cdot,\cdot)$-function".

Obviously $\Delta_i \to 0$ or $\Delta_{i+1} \to 0$ in (5.4.13), then $c_i \to 0$.

Contour plot of the harmonic mean of a and b ➙ ($w_a = w_b = 1/2$).

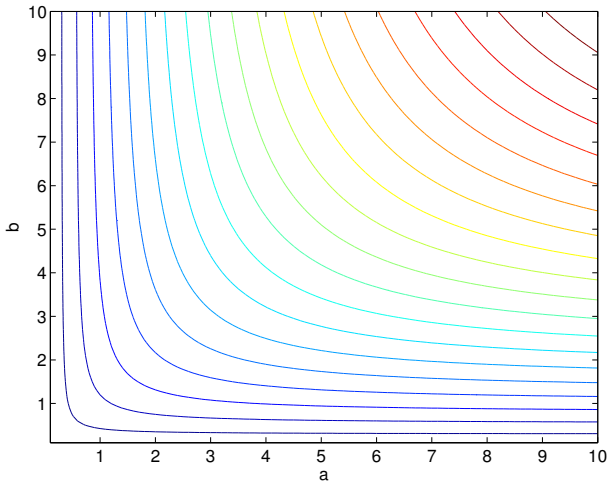


Fig. 184

A good choice of the weights is:

$$w_a = \frac{2h_{i+1} + h_i}{3(h_{i+1} + h_i)} \quad , \quad w_b = \frac{h_{i+1} + 2h_i}{3(h_{i+1} + h_i)}\,,$$

This yields the following local slopes, unless (5.4.12) enforces $c_i = 0$:

$$\overset{\operatorname{sgn}(\Delta_1)=\operatorname{sgn}(\Delta_2)}{\to} \quad c_i = \begin{cases} \Delta_1 & \text{, if } i = 0\,, \\ \dfrac{3(h_{i+1}+h_i)}{\frac{2h_{i+1}+h_i}{\Delta_i} + \frac{2h_i+h_{i+1}}{\Delta_{i+1}}} & \text{, for } i \in \{1,\ldots,n-1\}\,, \\ \Delta_n & \text{, if } i = n\,, \end{cases} \quad h_i := t_i - t_{i-1}\,. \tag{5.4.14}$$

Piecewise cubic Hermite interpolation with local slopes chosen according to (5.4.12) and (5.4.14) is available through the MATLAB function `v = pchip(t,y,x);`, where `t` passes the interpolation nodes, `y` the corresponding data values, and `x` is a vector of evaluation points, see `doc phchip` for details.

Example 5.4.15 (Monotonicity preserving piecewise cubic polynomial interpolation)

Data from Exp. 5.3.7

Plot created with MATLAB-function call

```
v = pchip(t,y,x);
```

- `t`: Data nodes t_j
- `y`: Data values y_j
- `x`: Evaluation points x_i
- `v`: Vector $s(x_i)$

We observe perfect local monotonicity preservation, no under- or overshoots at extrema. ▷

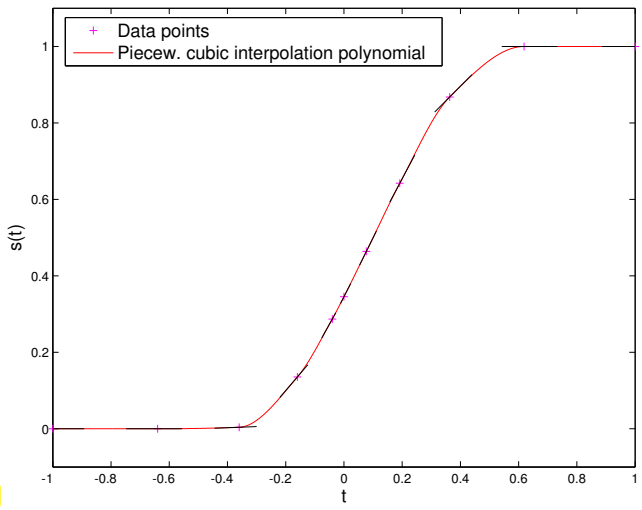


Fig. 185

Remark 5.4.16 (Non-linear cubic Hermite interpolation)

Note that the mapping $\mathbf{y} := [y_0, \ldots, y_n] \to c_i$ defined by (5.4.12) and (5.4.14) is not linear.

➢ The "`pchip` interpolaton operator" does **not** provide a **linear** mapping from data space $\mathbb{R}^{n+1}$ into $C^1([t_0, t_n])$ (in the sense of Def. 5.1.17).

In fact, the non-linearity of the piecewise cubic Hermite interpolation operator is necessary for (only global) monotonicity preservation:

Theorem 5.4.17. Property of linear, monotonicity preserving interpolation into C^1

If, for fixed node set $\{t_j\}_{j=0}^n$, $n \geq 2$, an interpolation scheme $\mathsf{I} : \mathbb{R}^{n+1} \to C^1(I)$ is linear as a mapping from data values to continuous functions on the interval covered by the nodes ($\to$ Def. 5.1.17), and monotonicity preserving, then $\mathsf{I}(\mathbf{y})'(t_j) = 0$ for all $\mathbf{y} \in \mathbb{R}^{n+1}$ and $j = 1, \ldots, n-1$.

Of course, an interpolant that is flat in all data points, as stipulated by Thm. 5.4.17 for a lineaer, monotonicity preserving, C^1-smooth interpolation scheme, does not make much sense.

At least, the piecewise cubic Hermite interpolation operator is local (in the sense discussed in § 5.4.7).

Theorem 5.4.18. Monotonicity preservation of limited cubic Hermite interpolation

The cubic Hermite interpolation polynomial with slopes as in Eq. (5.4.14) *provides a* local monotonicity-preserving C^1*-interpolant.*

Proof. See F. Fritsch und R. Carlson, *Monotone piecewise cubic interpolation*, SIAM J. Numer. Anal., 17 (1980), S. 238–246. □

The next code demonstrates the calculation of the slopes c_i in MATLAB's `pchip` (details in [**?**]):

C++11 code 5.4.19: Monotonicity preserving slopes in pchip

```
1  # include <Eigen/Dense>
2
3  using Eigen::VectorXd;
4
5  // using forward declaration of the function pchipend, implementation
       below
6  double pchipend(const double, const double, const double, const
       double);
7
8  void pchipslopes(const VectorXd& t, const VectorXd& y, VectorXd& c) {
9    // Calculation of local slopes c_i for shape preserving cubic Hermite
       interpolation, see (5.4.12), (5.4.14)
10   // t, y are vectors passing the data points
11   const unsigned n = t.size();
12   const VectorXd h = t.tail(n - 1) - t.head(n - 1),
```

```
13                  delta = (y.tail(n - 1) - y.head(n -
                        1)).cwiseQuotient(h); // linear
                        slopes
14    c = VectorXd::Zero(n);
15
16    // compute reconstruction slope according to (5.4.14)
17    for (unsigned i = 0; i < n - 2; ++i) {
18      if (delta(i)*delta(i + 1) > 0) {
19        const double w1 = 2*h(i + 1) + h(i),
20                     w2 = h(i + 1) + 2*h(i);
21        c(i + 1) = (w1 + w2)/(w1/delta(i) + w2/delta(i + 1));
22      }
23    }
24    // Special slopes at endpoints, beyond (5.4.14)
25    c(0) = pchipend(h(0), h(1), delta(0), delta(1));
26    c(n - 1) = pchipend(h(n - 2), h(n - 3), delta(n - 2), delta(n - 3));
27  }
28
29  double pchipend(const double h1, const double h2, const double del1,
        const double del2) {
30    // Non-centered, shape-preserving, three-point formula
31    double d = ((2*h1 + h2)*del1 - h1*del2)/(h1 + h2);
32    if (d*del1 < 0) {
33      d = 0;
34    }
35    else if (del1*del2 < 0 && std::abs(d) > std::abs(3*del1)) {
36      d = 3*del1;
37    }
38    return d;
39  }
```

5.5 Splines

Supplementary reading. Splines are also presented in [**?**, Ch. 9].

Piecewise cubic Hermite Interpolation presented in Section 5.4 entailed determining reconstruction slopes c_i. Now we learn about a way how to do piecewise polynomial interpolation, which results in C^k-interpolants, $k > 0$, and dispenses with auxiliary slopes. The idea is to obtain the missing conditions implicitly from extra continuity conditions.

Definition 5.5.1. Spline space → [?, Def. 8.1]

Given an interval $I := [a,b] \subset \mathbb{R}$ and a knot set/mesh $\mathcal{M} := \{a = t_0 < t_1 < \ldots < t_{n-1} < t_n = b\}$, the vector space $\mathcal{S}_{d,\mathcal{M}}$ of the spline functions of degree d (or order $d+1$) is defined by

$$\mathcal{S}_{d,\mathcal{M}} := \{s \in C^{d-1}(I) : s_j := s_{|[t_{j-1},t_j]} \in \mathcal{P}_d \ \forall j = 1,\ldots,n\} .$$

$d-1$-times continuously differentiable — *locally* polynomial of degree d

Obviously, spline spaces are mapped onto each other by differentiation & integration:

$$s \in \mathcal{S}_{d,\mathcal{M}} \quad \Rightarrow \quad s' \in \mathcal{S}_{d-1,\mathcal{M}} \quad \wedge \quad \int_a^t s(\tau)\,\mathrm{d}\tau \in \mathcal{S}_{d+1,\mathcal{M}} .$$

Spline spaces of the lowest degrees:

- $d = 0$: $\mathcal{M}$-piecewise constant *discontinuous* functions
- $d = 1$: $\mathcal{M}$-piecewise linear *continuous* functions
- $d = 2$: *continuously differentiable* $\mathcal{M}$-piecewise quadratic functions

The dimension of spline space can be found by a counting argument (heuristic): We count the number of "degrees of freedom" (d.o.f.s) possessed by a $\mathcal{M}$-piecewise polynomial of degree d, and subtract the number of linear constraints implicit contained in Def. 5.5.1:

$$\dim \mathcal{S}_{d,\mathcal{M}} = n \cdot \dim \mathcal{P}_d - \#\{C^{d-1} \text{ continuity constraints}\} = n \cdot (d+1) - (n-1) \cdot d = n + d .$$

Theorem 5.5.2. Dimension of spline space

The space $\mathcal{S}_{d,\mathcal{M}}$ from Def. 5.5.1 has dimension

$$\dim \mathcal{S}_{d,\mathcal{M}} = n + d .$$

We already know the special case of interpolation in $\mathcal{S}_{1,\mathcal{M}}$, when the interpolation nodes are the knots of $\mathcal{M}$, because this boils down to simple piecewise linear interpolation, see Section 5.3.2.

5.5.1 Cubic spline interpolation

Supplementary reading. More details in [**?**, XIII, 46], [**?**, Sect. 8.6.1].

Cognitive psychology teaches us that the human eye perceives a C^2-functions as "smooth", while it can still spot the abrupt change of curvature at the possible discontinuities of the second derivatives of a cube Hermite interpolant (→ Def. 5.4.1).

For this reason the simplest spline functions featuring C^2-smoothness are of great importance in computer aided design (CAD). They are the cubic splines, $\mathcal{M}$-piecewise polynomials of degree 3 contained in $\mathcal{S}_{3,\mathcal{M}}$ (→ Def. 5.5.1).

(5.5.3) Definition of cubic spline interpolant

In this section we study cubic spline interpolation (related to cubic Hermite interpolation, Section 5.4)

Task: Given a mesh $\mathcal{M} := \{t_0 < t_1 < \cdots < t_n\}$, $n \in \mathbb{N}$, "find" a cubic spline $s \in \mathcal{S}_{3,\mathcal{M}}$ that complies with the interpolation conditions

$$s(t_j) = y_j \quad , \quad j = 0, \ldots, n \, . \tag{5.5.4}$$

$\hat{=}$ interpolation at knots !

From dimensional considerations it is clear that the interpolation conditions will fail to fix the interpolating cubic spline uniquely:

$$\dim \mathcal{S}_{3,\mathcal{M}} - \#\{\text{ interpolation conditions}\} = (n+3) - (n+1) = 2 \text{ free d.o.f.}$$

"two conditions are missing" ➢ interpolation problem is not yet well defined!

(5.5.5) Computing cubic spline interpolants

We opt for a linear interpolation scheme (→ Def. 5.1.17) into the spline space $\mathcal{S}_{3,\mathcal{M}}$. As explained in § 5.1.13, this will lead to an equivalent linear system of equations for expansion coefficients with respect to a suitable basis.

We reuse the local representation of a cubic spline through cubic Hermite cardinal basis polynomials from (5.4.5):

(5.4.4) ➘

$$\begin{aligned} s_{|[t_{j-1},t_j]}(t) = \; & s(t_{j-1}) \cdot (1 - 3\tau^2 + 2\tau^3) + \\ & s(t_j) \cdot (3\tau^2 - 2\tau^3) + \\ & h_j s'(t_{j-1}) \cdot (\tau - 2\tau^2 + \tau^3) + \\ & h_j s'(t_j) \cdot (-\tau^2 + \tau^3) \, , \end{aligned} \tag{5.5.6}$$

with $h_j := t_j - t_{j-1}$, $\tau := (t - t_{j-1})/h_j$.

➢ Task of cubic spline interpolation boils down to finding slopes $s'(t_j)$ in the knots of the mesh $\mathcal{M}$.

Once these slopes are known, the efficient local evaluation of a cubic spline function can be done as for a cubic Hermite interpolant, see Section 5.4.1, Code 5.4.6.

Note: if $s(t_j)$, $s'(t_j)$, $j = 0, \ldots, n$, are fixed, then the representation Eq. (5.5.6) already guarantees $s \in C^1([t_0, t_n])$, *cf.* the discussion for cubic Hermite interpolation, Section 5.4.

➢ only continuity of s''

- ✦ has to be enforced by choice of $s'(t_j)$
- ⇕
- ✦ will yield extra conditions to fix the $s'(t_j)$

However, do the

✦ interpolation conditions Eq. (5.5.4) $s(t_j) = y_j, \quad j = 0,\ldots,n$, and the
✦ smoothness constraint $s \in C^2([t_0,t_n])$

uniquely determine the unknown slopes $c_j := s'(t_j)$ **?**

From $s \in C^2([t_0,t_n])$ we obtain $n-1$ continuity constraints for $s''(t)$ at the internal nodes

$$s''_{|[t_{j-1},t_j]}(t_j) = s''_{|[t_j,t_{j+1}]}(t_j)\,, \quad j = 1,\ldots,n-1\,. \tag{5.5.7}$$

Based on Eq. (5.5.6), we express Eq. (5.5.7) in concrete terms, using

$$\begin{aligned} s''_{|[t_{j-1},t_j]}(t) = {} & s(t_{j-1})h_j^{-2}6(-1+2\tau) + s(t_j)h_j^{-2}6(1-2\tau) \\ & + h_j^{-1}s'(t_{j-1})(-4+6\tau) + h_j^{-1}s'(t_j)(-2+6\tau)\,, \quad \tau := (t-t_{j-1})/h_j\,, \end{aligned} \tag{5.5.8}$$

which can be obtained by the chain rule and from $\frac{d\tau}{dt} = h_j^{-1}$.

$$\overset{\text{Eq. (5.5.8)}}{\Rightarrow} \quad \begin{aligned} s''_{|[t_{j-1},t_j]}(t_{j-1}) &= -6\cdot s(t_{j-1})h_j^{-2} + 6\cdot s(t_j)h_j^{-2} - 4\cdot h_j^{-1}s'(t_{j-1}) - 2\cdot h_j^{-1}s'(t_j)\,, \\ s''_{|[t_{j-1},t_j]}(t_j) &= 6\cdot s(t_{j-1})h_j^{-2} + -6\cdot s(t_j)h_j^{-2} + 2\cdot h_j^{-1}s'(t_{j-1}) + 4\cdot h_j^{-1}s'(t_j)\,. \end{aligned}$$

Eq. (5.5.7) ➙ $n-1$ linear equations for n slopes $c_j := s'(t_j)$; with $s(t_i) = y_i$,

$$\frac{1}{h_j}c_{j-1} + \left(\frac{2}{h_j} + \frac{2}{h_{j+1}}\right)c_j + \frac{1}{h_{j+1}}c_{j+1} = 3\left(\frac{y_j - y_{j-1}}{h_j^2} + \frac{y_{j+1}-y_j}{h_{j+1}^2}\right)\,, \tag{5.5.9}$$

for $j = 1,\ldots,n-1$.

Eq. (5.5.9) ⇔ *underdetermined* $(n-1)\times(n+1)$ linear system of equations.

$n-1$: no. of interpolation conditions
$n+1$: dimension of cubic spline space on knot set $\{t_0 < t_1 < \cdots < t_n\}$

$$\begin{bmatrix} b_0 & a_1 & b_1 & 0 & \cdots & & \cdots & 0 \\ 0 & b_1 & a_2 & b_2 & & & & \\ & 0 & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & a_{n-2} & b_{n-2} & 0 \\ 0 & \cdots & & \cdots & 0 & b_{n-2} & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} c_0 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 3\left(\frac{y_1-y_0}{h_1^2} + \frac{y_2-y_1}{h_2^2}\right) \\ \vdots \\ 3\left(\frac{y_{n-1}-y_{n-2}}{h_{n-1}^2} + \frac{y_n-y_{n-1}}{h_n^2}\right) \end{bmatrix}. \tag{5.5.10}$$

with

$$b_i := \frac{1}{h_{i+1}}\,, \quad a_i := \frac{2}{h_i} + \frac{2}{h_{i+1}}\,, \quad \begin{array}{l} i = 0,1,\ldots,n-1\,, \\ [\quad b_i, a_i > 0 \quad , \quad a_i = 2(b_i + b_{i-1}) \quad];. \end{array}$$

➙ two additional constraints are required, as already noted in § 5.5.3.

(5.5.11) Types of cubic spline interpolants

To saturate the remaining two degrees of freedom the following three approaches are popular:

① Complete cubic spline interpolation: $s'(t_0) = c_0$, $s'(t_n) = c_n$ prescribed.

Then the first and last column can be removed from the system matrix of (5.5.10). Their products with c_0 and c_n, respectively, have to be subtracted from the right hand side of (5.5.10).

② Natural cubic spline interpolation: $s''(t_0) = s''(t_n) = 0$

$$\frac{2}{h_1}c_0 + \frac{1}{h_1}c_1 = 3\frac{y_1 - y_0}{h_1^2} \quad , \quad \frac{1}{h_n}c_{n-1} + \frac{2}{h_n}c_n = 3\frac{y_n - y_{n-1}}{h_n^2} \, .$$

Combining these two extra equations with (5.5.10), we arrive at a linear system of equations with tridiagonal s.p.d. ($\rightarrow$ Def. 1.1.8, Lemma 2.8.12) system matrix and unknowns $c_0, \ldots, c_n$. Due to Thm. 2.7.58 it can be solved with an asymptotic computational effort of $O(n)$.

③ Periodic cubic spline interpolation: $s'(t_0) = s'(t_n)$ ($\succ c_0 = c_n$), $s''(t_0) = s''(t_n)$

This removes one unknown and adds another equations so that we end up with an $n \times n$-linear system with s.p.d. ($\rightarrow$ Def. 1.1.8) system matrix

$$\mathbf{A} := \begin{bmatrix} a_1 & b_1 & 0 & \cdots & 0 & b_0 \\ b_1 & a_2 & b_2 & & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & & & \ddots & a_{n-1} & b_{n-1} \\ b_0 & 0 & \cdots & 0 & b_{n-1} & a_0 \end{bmatrix} \quad , \quad \begin{aligned} b_i &:= \frac{1}{h_{i+1}}, \quad i = 0, 1, \ldots, n-1 \, , \\ a_i &:= \frac{2}{h_i} + \frac{2}{h_{i+1}}, \quad i = 0, 1, \ldots, n-1 \, . \end{aligned}$$

This linear system can be solved with rank-1-modifications techniques (see § 2.6.13, Lemma 2.6.22) + tridiagonal elimination: asymptotic computational effort $O(n)$.

Remark 5.5.12 (Splines in MATLAB)

MATLAB provides many tools for computing and dealing with splines.

MATLAB-function: `v = spline(t,y,x)`: natural / complete spline interpolation

There is even a "Curve Fitting Toolbox" that provides extended functionality.

Remark 5.5.13 (Piecewise cubic interpolation schemes)

- Piecewise cubic local Lagrange interpolation
  - $\succ$ Extra degrees of freedom fixed by putting four nodes in one interval
  - $\rightarrow$ yields merely C^0-interpolant; perfectly local.
- Cubic Hermite interpolation
  - $\succ$ Extra degrees of freedom fixed by reconstruction slopes
  - $\rightarrow$ yields C^1-interpolant; still local.

✦ Cubic spline interpolation

➢ Extra degrees of freedom fixed by C^2-smoothness, complete/natural/periodic constraint.

➥ yields C^2-interpolant; non-local.

5.5.2 Structural properties of cubic spline interpolants

(5.5.14) Extremal properties of natural cubic spline interpolants → [?, Sect. 8.6.1, Property 8.2]

For a function $f : [a,b] \mapsto \mathbb{R}$, $f \in C^2([a,b])$, the term

$$E_{\mathrm{bd}}(f) := \tfrac{1}{2}\int_a^b |f''(t)|^2 \,\mathrm{d}t\,,$$

models the elastic bending energy of a rod, whose shape is described by the graph of f (Soundness check: zero bending energy for straight rod). We will show that cubic spline interpolants have minimal bending energy among all C^2-smooth interpolating functions.

Given: mesh $\mathcal{M} := \{a = t_0 < t_1 < \cdots < t_n = b\}$ of $[a,b]$ with knots t_j

Set $s \in \mathcal{S}_{3,\mathcal{M}}$ **:=** natural cubic spline interpolant of data points $(t_i, y_i) \in \mathbb{R}^2$, $i = 0,\ldots,n$.

Theorem 5.5.15. Optimality of natural cubic spline interpolant

The natural cubic spline interpolant s minimizes the elastic curvature energy among all interpolating functions in $C^2([a,b])$, that is

$$E_{bd}(s) \le E_{bd}(f) \quad \forall f \in C^2([a,b]),\ f(t_i) = y_i,\ i = 0,\ldots,n\,.$$

Idea of proof: variational calculus

We show that any small perturbation of s such that the perturbed spline still satisfies the interpolation conditions leads to an increase in elastic energy.

Pick *perturbation direction* $k \in C^2([t_0,t_n])$ satisfying $k(t_i) = 0$, $i = 0,\ldots,n$:

$$\begin{aligned} E_{\mathrm{bd}}(s+k) &= \tfrac{1}{2}\int_a^b |s'' + \lambda k''|^2 \,\mathrm{d}t \\ &= E_{\mathrm{bd}}(s) + \underbrace{\int_a^b s''(t)k''(t)\,\mathrm{d}t}_{:=I} + \underbrace{\tfrac{1}{2}\int_a^b |k''|^2 \,\mathrm{d}t}_{\ge 0}\,. \end{aligned} \tag{5.5.16}$$

Scrutiny of I: split in interval contributions, integrate by parts twice, and use $s^{(4)} \equiv 0$:

$$\begin{aligned} I &= \sum_{j=1}^{n} \int_{t_{j-1}}^{t_j} s''(t)k''(t)\,\mathrm{d}t \\ &= -\sum_{j=1}^{n} \left(s'''(t_j^-)\underbrace{k(t_j)}_{=0} - s'''(t_{j-1}^+)\underbrace{k(t_{j-1})}_{=0} \right) + \underbrace{s''(t_n)}_{=0}k'(t_n) - \underbrace{s''(t_0)}_{=0}k'(t_0) = 0\,. \end{aligned}$$

In light of (5.5.16): non perturbation compatible with interpolation conditions can make the bending energy of s decrease!

Remark 5.5.17 (Origin of the term "Spline")

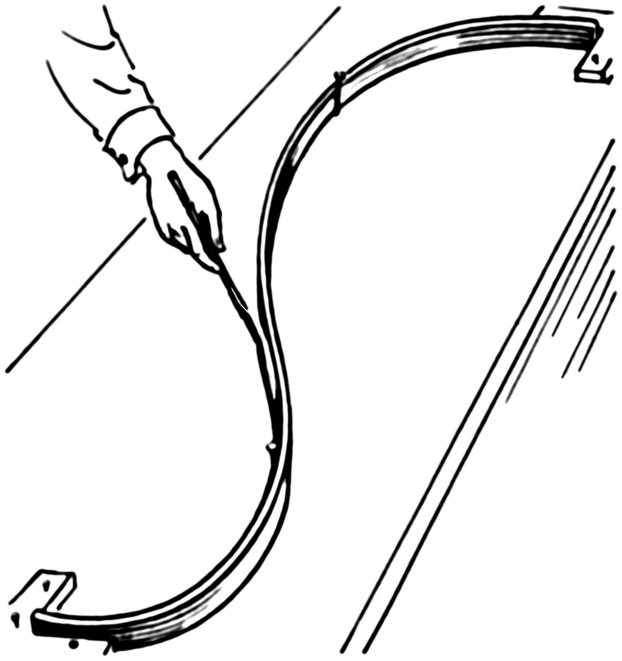
§ 5.5.14: (Natural) cubic spline interpolant provides C^2-curve of minimal elastic bending energy that travels through prescribed points.

$$\Updownarrow$$

Nature: A thin elastic rod fixed a certain points attains a shape that minimizes its potential bending energy (virtual work principle of statics).

Cubic spline interpolation approximates shape of elastic rods. Such rods were in fact used in the manufacturing of ship hulls as "analog computers" for "interpolating points" that were specified by the designer of the ship.

Cubic spline interpolation before MATLAB ▷



Remark 5.5.18 (Shape preservation of cubic spline interpolation)

Data $s(t_j) = y_j$ from Exp. 5.3.7 and

$$c_0 := \frac{y_1 - y_0}{t_1 - t_0} ,$$
$$c_n := \frac{y_n - y_{n-1}}{t_n - t_{n-1}} .$$

The cubic spline interpolant is nor monotonicity nor curvature preserving

This is not surprising in light of Thm. 5.4.17, because cubic spline interpolation is a linear interpolation scheme.

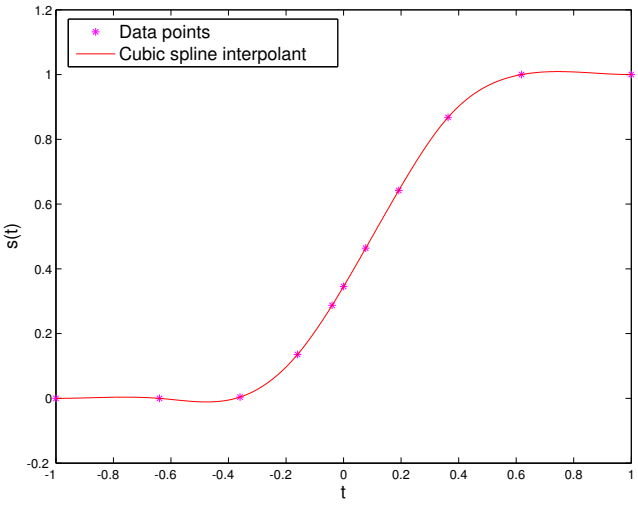


Fig. 186

(5.5.19) Weak locality of an interpolation scheme

In terms of locality of interpolation schemes, in the sense of § 5.4.7, we habe seen:

- Piecewise linear interpolation ($\rightarrow$ Section 5.3.2) is strictly local: Changing a single data value y_j affects the interpolant only on the interval $]t_{j-1}, t_{j+1}[$.
- Monotonicity preserving piecewise cubic Hermite interpolation ($\rightarrow$ Section 5.4.2) is still local, because changing y_j will lead to a change in the interpolant only in $]t_{j-2}, t_{j+2}[$ (the remote intervals are affected through the averaging of local slopes).
- Polynomial Lagrange interpolation is highly non-local, see Ex. 5.2.63.

We can weaken the notion of locality of an interpolation scheme on an ordered node set $\{t_i\}_{i=0}^n$:

➢ (weak) locality measures the impact of a perturbation of a data value y_j at points $t \in [t_0, t_n]$ as a function of $|t - t_i|$.

➢ an interpolation scheme is weakly local, if the impact of the perturbation of y_i displays a rapid (e.g. exponential) decay as $|t - t_i|$ increases.

For a *linear* interpolation scheme ($\rightarrow$ § 5.1.13) locality can be deduced from the decay of the cardinal interpolants/cardinal basis functions ($\rightarrow$ Lagrange polynomials of § 5.2.10), that is, the functions $b_j := \mathsf{I}(\mathbf{e}_j)$, where $\mathbf{e}_j$ is the j-th unit vector, and I the interpolation operator. Then weak locality can be quantified as

$$\exists \lambda > 0 | : \quad b_j(t)| \leq \exp(-\lambda |t_{t_j}|) , \quad t \in [t_0, t_n] . \tag{5.5.20}$$

Remember:

- Lagrange polynomials satisfying (5.2.11) provide cardinal interpolants for polynomial interpolation $\rightarrow$ § 5.2.10. As is clear from Fig. 170, they do not display any decay away from their "base node". Rather, they grow strongly. Hence, there is no locality in global polynomial interpolation.
- Tent functions ($\rightarrow$ Fig. 169) are the cardinal basis functions for piecewise linear interpolation, see Ex. 5.1.10. Hence, this scheme is perfectly local, see (5.3.9).

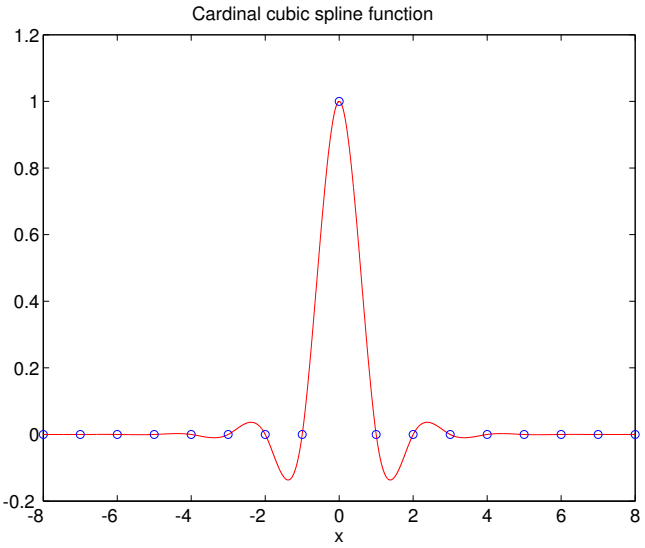
Experiment 5.5.21 (Weak locality of the natural cubic spline interpolation)

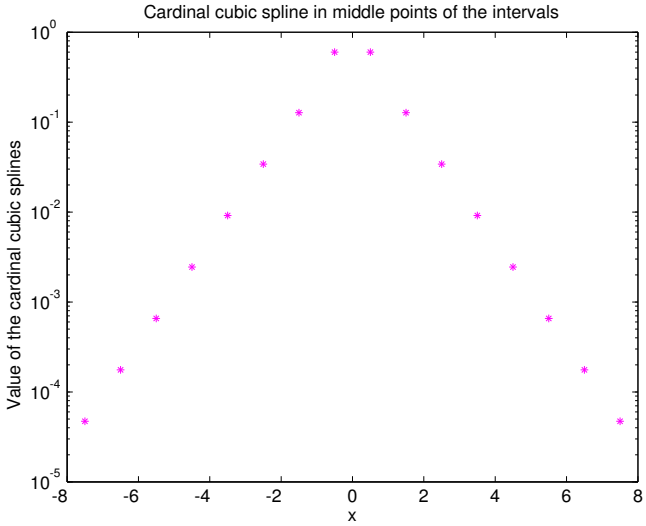
Given a grid $\mathcal{M} := \{t_0 < t_1 < \cdots < t_n\}$ the ith natural cardinal spline is defined as

$$L_i \in \mathcal{S}_{3,\mathcal{M}}, \quad L_i(t_j) = \delta_{ij}\,, \quad L_i''(t_0) = L_i''(t_n) = 0\,. \tag{5.5.22}$$

Natural spline interpolant: $s(t) = \sum_{j=0}^{n} y_j L_j(t)$.

Decay of L_i $\leftrightarrow$ weak locality of natural cubic spline interpolation.





Exponential decay of the cardinal splines ➞ cubic spline interpolation is weakly local.

5.5.3 Shape Preserving Spline Interpolation

According to Rem. 5.5.18 is cubic spline interpolation neither monotonicity preserving nor curvature preserving. Necessarily so, because it is a *linear* interpolation scheme, see Thm. 5.4.17.

This section presents a non-linear *quadratic spline* ($\rightarrow$ Def. 5.5.1, C^1-functions) based interpolation scheme that manages to preserve both monotonicity and curvature of data even in a local sense, *cf.* Section 5.3.

Given: data points $(t_i, y_i) \in \mathbb{R}^2$, $i = 0, \ldots, n$, assume ordering $t_0 < t_1 < \cdots < t_n$.

Sought:

- extended knot set $\mathcal{M} \subset [t_0, t_n]$ ($\rightarrow$ Def. 5.5.1),
- an interpolating quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$, $s(t_i) = y_i$, $i = 0, \ldots, n$ that preserves the "shape" of the data in the sense of § 5.3.2.

Notice that here $\mathcal{M} \neq \{t_j\}_{j=0}^{n}$: s interpolates the data in the points t_i but is piecewise polynomial with respect to $\mathcal{M}$! The interpolation nodes will usually not belong to $\mathcal{M}$.

We proceed in four steps:

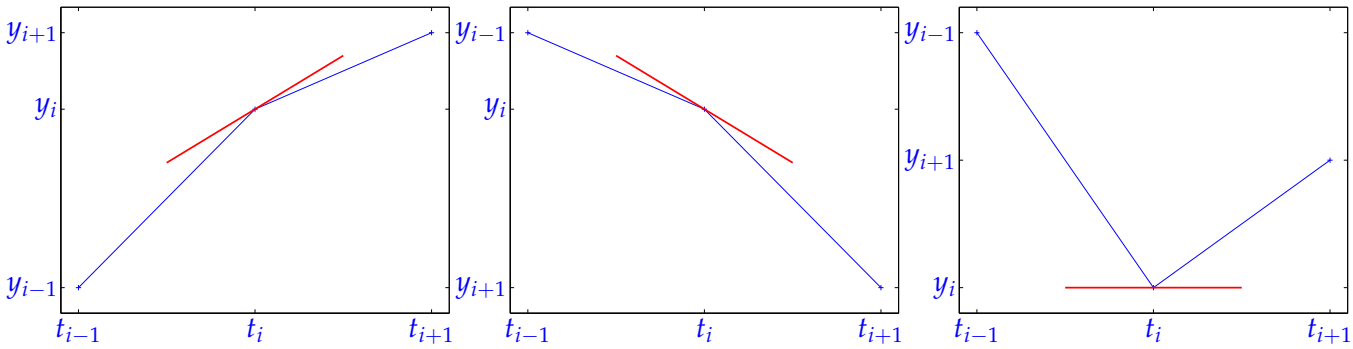
① Shape preserving choice of slopes c_i, $i = 0, \ldots, n$ [**?**, **?**], analogous to Section 5.4.2

Recall Eq. (5.4.12) and Eq. (5.4.14): we fix the slopes c_i in the nodes using the harmonic mean of data slopes Δ_j, the final interpolant will be tangent to these segments in the points (t_i, y_i). If (t_i, y_i) is a local maximum or minimum of the data, c_j is set to zero ($\rightarrow$ § 5.4.11)

$$\text{Limiter} \quad c_i := \begin{cases} \frac{2}{\Delta_i^{-1}+\Delta_{i+1}^{-1}} & \text{, if } \operatorname{sign}(\Delta_i) = \operatorname{sign}(\Delta_{i+1}) \text{ ,} \\ 0 & \text{otherwise,} \end{cases} \qquad i = 1, \ldots, n-1 \text{ .}$$

$$c_0 := 2\Delta_1 - c_1 \text{ ,} \qquad c_n := 2\Delta_n - c_{n-1} \text{ ,}$$

where $\Delta_j = \frac{y_j - y_{j-1}}{t_j - t_{j-1}}$.



Figures: slopes according to limited harmonic mean formula

② Choice of "extra knots" $p_i \in]t_{i-1}, t_i]$, $i = 1, \ldots, n$:

Rule:

Let T_i be the unique straight line through (t_i, y_i) with slope c_i; — in figure ▷

If the intersection of T_{i-1} and T_i is non-empty and has a t-coordinate $\in]t_{i-1}, t_i]$,

☞ then $p_i :=$ t-coordinate of $T_{i-1} \cap T_i$,

☞ otherwise $p_i = \frac{1}{2}(t_{i-1} + t_i)$.

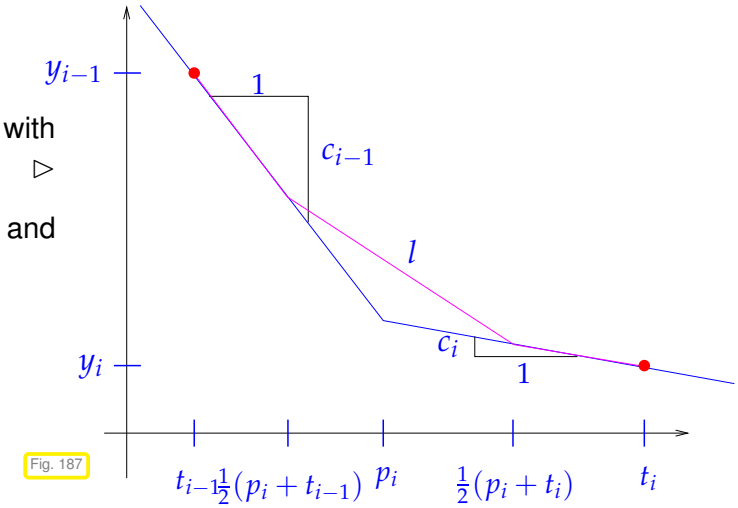


Fig. 187

These points will be used to build the knot set for the final quadratic spline:

$$\mathcal{M} = \{t_0 < p_1 \leq t_1 < p_2 \leq \cdots < p_n \leq t_n\} \text{ .}$$

C++-code 5.5.23: Quadratic spline: selection of p_i

```
# include <Eigen/Dense>
using Eigen::VectorXd;

// choice of extra knots p_i ∈ ]t_{i-1}, t_i[
// for given nodeset t and coefficients c
void extra_knots(const VectorXd& t, const VectorXd& c, VectorXd& p) {
  const unsigned n = t.size();
  p = (t(n-1)-1)*VectorXd::Ones(n-1);
  for (unsigned j = 0; j < n-1; ++j) {
    if (c(j) != c(j+1)) {
      p(j) = (y(j+1)-y(j)+t(j)*c(j)-t(j+1)*c(j+1))/(c(j+1)-
          c(j));
    }
    if (p(j) < t(j) || p(j) > t(j+1)) {
      p(j) = 0.5*(t(j)+t(j+1));
    }
  }
}
```

③ Set l = linear spline (polygon) on the knot set $\mathcal{M}'$ (middle points of $\mathcal{M}$)

$$\mathcal{M}' = \{t_0 < \tfrac{1}{2}(t_0+p_1) < \tfrac{1}{2}(p_1+t_1) < \tfrac{1}{2}(t_1+p_2) < \cdots < \tfrac{1}{2}(t_{n-1}+p_n) < \tfrac{1}{2}(p_n+t_n) < t_n\}\,,$$

with $l(t_i) = y_i \quad , \quad l'(t_i) = c_i$.

- In each interval $(\frac{1}{2}(p_j+t_j), \frac{1}{2}(t_j+p_{j+1}))$ the spline corresponds to the segment of slope c_j passing through the data point (t_j, y_j).
- In each interval $(\frac{1}{2}(t_j+p_{j+1}), \frac{1}{2}(p_{j+1}+t_{j+1}))$ the spline corresponds to the segment connecting the previous ones, see Fig. 187.

l "inherits" local monotonicity and curvature from the data.

Example 5.5.24 (Auxiliary construction for shape preserving quadratic spline interpolation)

Data points: `t=(0:12); y = cos(t);`

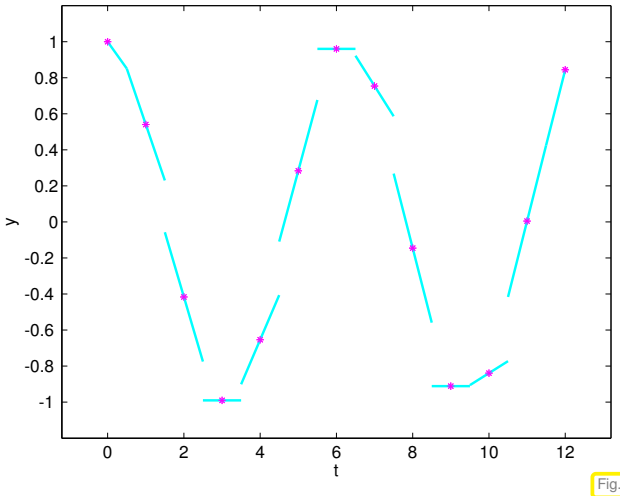


Fig. 188

Local slopes c_i, $i = 0, \ldots, n$

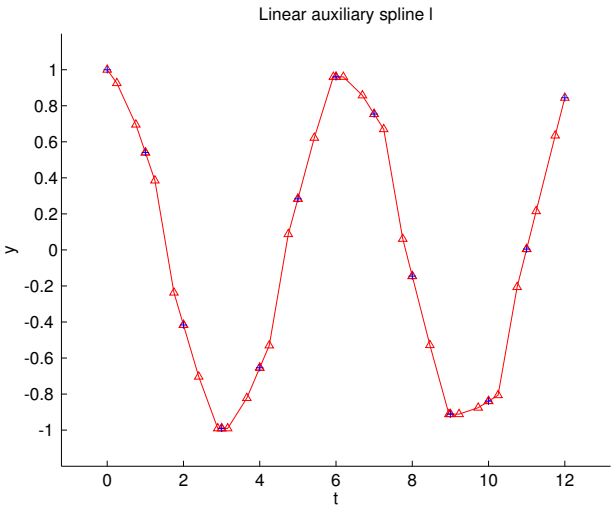


Fig. 189

Linear auxiliary spline l

④ Local quadratic approximation / interpolation of l:

Tedious, but elementary calculus, confirms the following fact:

Lemma 5.5.25.

If g is a linear spline through the three points

$$(a, y_a)\ ,\ (\frac{1}{2}(a+b), w)\ ,\ (b, y_b) \quad \text{with} a < b\ ,\quad y_a, y_b, w \in \mathbb{R}\ ,$$

then the parabola

$$p(t) := (y_a(b-t)^2 + 2w(t-a)(b-t) + y_b(t-a)^2)/(b-a)^2\ ,\quad a \le t \le b\ ,$$

satisfies

1. $p(a) = y_a \quad , \quad p(b) = y_b \quad , \quad p'(a) = g'(a) \quad , \quad p'(b) = g'(b)$,
2. g *monotonic increasing / decreasing* $\Rightarrow$ p *monotonic increasing / decreasing,*
3. g *convex / concave* $\Rightarrow$ p *convex / concave.*

The proof boils down to discussing many cases as indiated in the following plots:

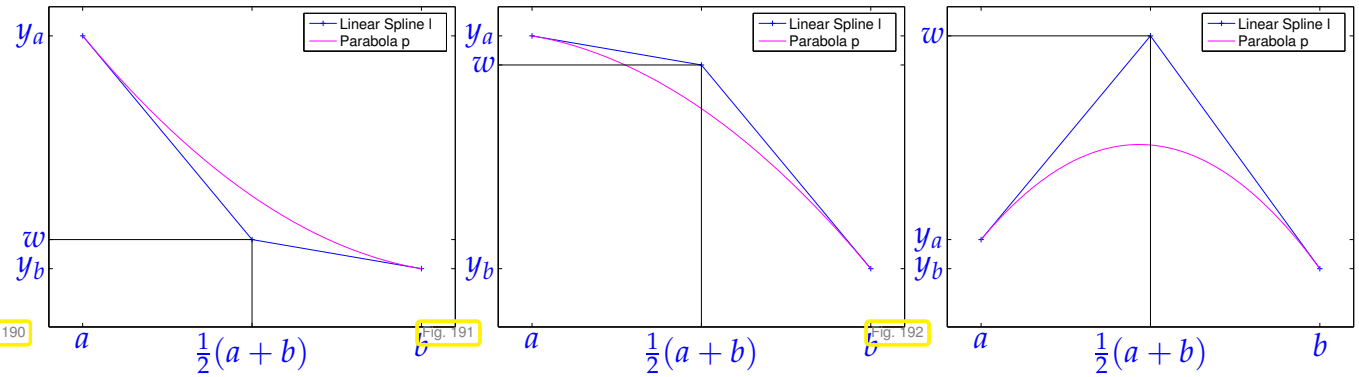


Fig. 190 Fig. 191 Fig. 192

Lemma 5.5.25 implies that the final quadratic spline that passes through the points (t_j, y_j) with slopes c_j can be built locally as the parabola p using the linear spline l that plays the role of g in the lemma.

Continuation of Ex. 5.5.24:

Interpolating quadratic spline →

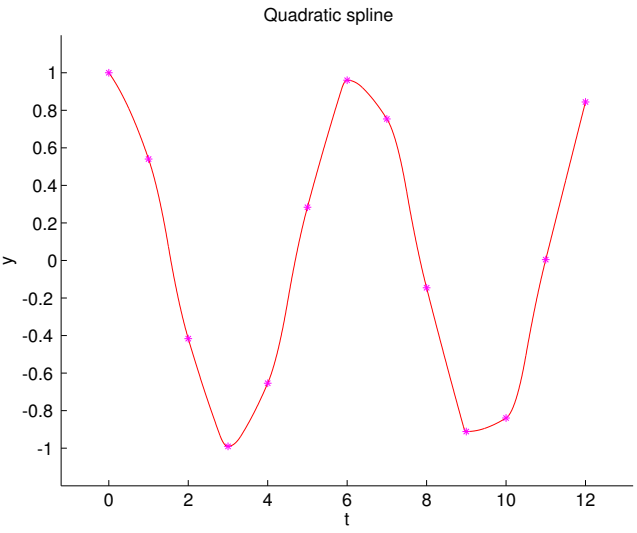


Fig. 193

Example 5.5.26 (Cardinal shape preserving quadratic spline)

We examine the shape preserving quadratic spline that interpolates data values $y_j = 0$, $j \neq i$, y_i, $i \in \{0, \ldots, n\}$, on an equidistant node set.

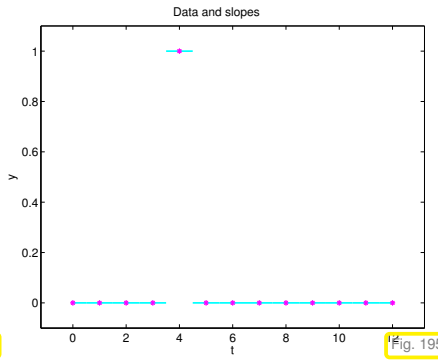


Fig. 194

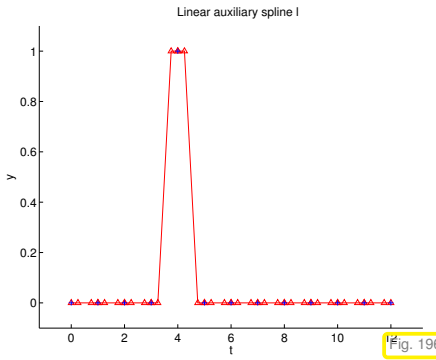


Fig. 195

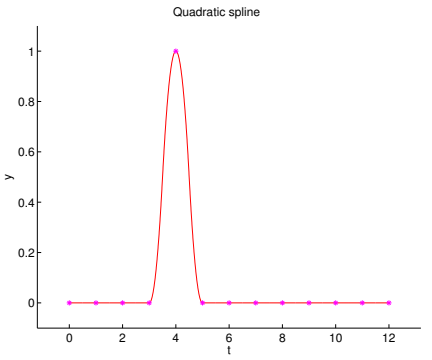


Fig. 196

Shape preserving quadratic spline interpolation = local **but** not linear

Example 5.5.27 (Shape preserving quadratic spline interpolation)

Data from Exp. 5.3.7:

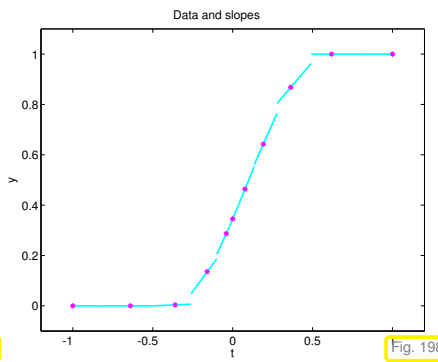


Fig. 197

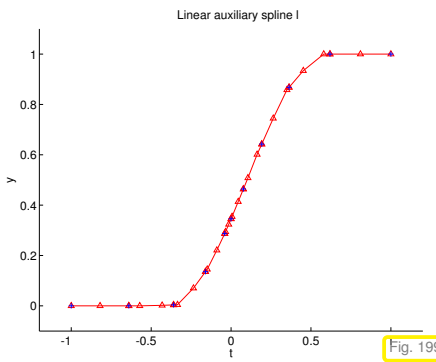


Fig. 198

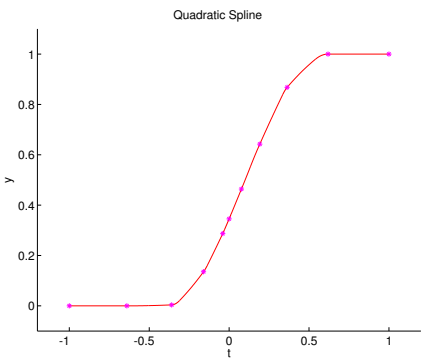


Fig. 199

Data from [**?**]:

t_i	0	1	2	3	4	5	6	7	8	9	10	11	12
y_i	0	0.3	0.5	0.2	0.6	1.2	1.3	1	1	1	1	0	-1

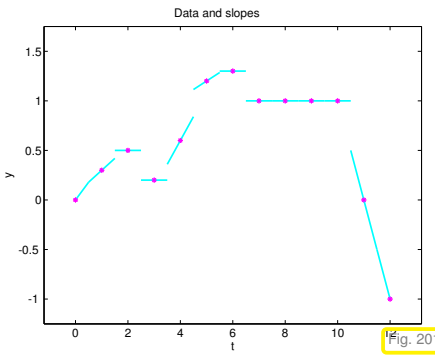


Fig. 200

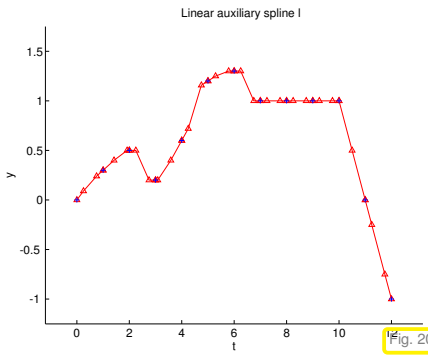


Fig. 201

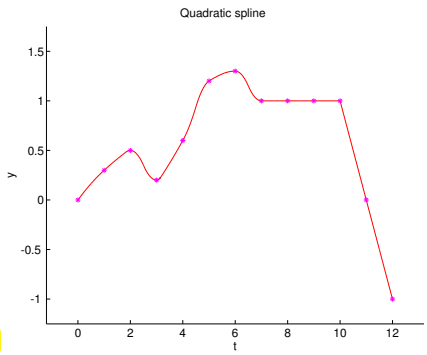


Fig. 202

In all cases we observe excellent local shape preservation.

C++-code 5.5.28: Step by step shape preserving spline interpolation

```
#include <iostream>
#include <cmath>
#include <string>
#include <vector>
#include <Eigen/Dense>
#include <figure/figure.hpp>

using Eigen::VectorXd;
using Eigen::ArrayXd;

void shapepresintp(const VectorXd& t, const VectorXd& y) {
  const long n = t.size() - 1;

  // ======= Step 1:  choice of slopes ========
  // shape-faithful slopes (c) in the nodes using harmonic mean of data slopes
  // the final interpolant will be tangents to these segments
  std::cout << "STEP 1 - Shape-faithful slopes ...\n";

  const VectorXd h = t.tail(n) - t.head(n); // n = t.size() - 1
  const VectorXd delta = (y.tail(n) - y.head(n)).cwiseQuotient(h);

  std::cout << delta.transpose() << "\n";

  VectorXd c = VectorXd::Zero(n + 1);
  for (long j = 0; j < n - 1; ++j) {
    if (delta(j)*delta(j + 1) > 0) { // no change of sign
      c(j + 1) = 2./(1./delta(j) + 1./delta(j + 1));
    }
    else { // change of sign
      // as c is already zero we have nothing to do here
    }
  }
  // take care of first and last slope
  c(0) = 2*delta(0) - c(1);
  c(n) = 2*delta(n - 1) - c(n - 1);

  // plot segments indicating the slopes c(i)
  mgl::Figure segs;
  std::vector<double> x(2), s(2);
  for (long j = 1; j < n; ++j) {
    x[0] = t(j) - 0.3*h(j - 1); x[1] = t(j) + 0.3*h(j);
    s[0] = y(j) - 0.3*h(j - 1)*c(j); s[1] = y(j) + 0.3*h(j)*c(j);
    segs.plot(x, s, "C");
  }
  // plot first segment
  x[0] = t(0); x[1] = t(0) + 0.3*h(0);
  s[0] = y(0); s[1] = y(0) + 0.3*h(0)*c(0);
  segs.plot(x, s, "C");

  // plot last segment
  x[0] = t(n) - 0.3*h(n - 1); x[1] = t(n);
  s[0] = y(n) - 0.3*h(n - 1)*c(n); s[1] = y(n);
  segs.plot(x, s, "C");

  // plot data points
```

```
56  segs.plot(t, y, " #mo");
57
58  // save plot
59  segs.title("Data and slopes");
60  segs.grid(false);
61  segs.save("segments");
62
63  // ==== Step 2:  choice of middle points ====
64  // fix point p_j in [t_j,t_{j+1}] depending on the slopes c_j,c_{j+1}
65
66  std::cout << "Step 2 - Middle points ...\n";
67
68  VectorXd p = (t(0) - 1)*VectorXd::Ones(n);
69  for (long j = 0; j < n; ++j) {
70    if (c(j) != c(j + 1)) { // avoid division by zero
71      p(j) = (y(j + 1) - y(j) + t(j)*c(j) - t(j + 1)*c(j + 1)) / (c(j) - c(j + 1));
72    }
73    // check and repair if p_j is outside its interval
74    if (p(j) < t(j) || p(j) > t(j + 1)) {
75      p(j) = 0.5*(t(j) + t(j + 1));
76    }
77  }
78
79  // ==== Step 3:  auxiliary line spline =====
80  // build the linear spline with nodes in:
81  // - t_j
82  // - the middle points between t_j and p_j
83  // - the middle points between p_j and t_{j+1}
84  // - t_{j+1}
85  // and with slopes c_j in t_j, for every j
86
87  std::cout << "Step 3 - Auxiliary line spline ...\n";
88
89  x = std::vector<double>(4);
90  s = std::vector<double>(4);
91
92  mgl::Figure lin;
93
94  for (long j = 0; j < n; ++j) {
95    x[0] = t(j); s[0] = y(j);
96    x[1] = (p(j) + t(j))/2.; s[1] = y(j) + c(j)*(p(j) - t(j))/2.;
97    x[2] = (p(j) + t(j+1))/2.; s[2] = y(j+1) + c(j+1)*(p(j) - t(j+1))/2.;
98    x[3] = t(j+1); s[3] = y(j+1);
99
100   // plot linear spline
101   lin.plot(x, s, "r");
102
103   // plot additional evaluation points (p_j+t_j)/2, (p_j+t_{j+1})/2
104   std::vector<double> pt = {x[1], x[2]},
105                       py = {s[1], s[2]};
106   lin.plot(pt, py, " r^");
107 }
108 // plot data points
109 lin.plot(t, y, " #r^").label("Data");
110 lin.addlabel("Additional points", " r^");
111 lin.addlabel("Linear splines", "r");
112 // lin.legend(0, 0.3);
113 lin.title("Linear auxiliary spline I");
114 lin.grid(false);
115 lin.save("linearsplines");
116
117 // ======= Step 4:  quadratic spline ========
118 // final quadratic shape preserving spline
119 // quadratic polynomial in the intervals [t_j,p_j] and [p_j,t_{j+1}]
120 // tangent in t_j and p_j to the linear spline of step 3
121
122 std::cout << "Step 4 - Quadratic spline ...\n";
123
124 // for every interval 2 quadratic interpolations
125 // a, b, ya, yb = extremes and values in subinterval
126 // w = value in middle point that gives the red slope
127 mgl::Figure quad;
```

```
128   for (long j = 0; j < n; ++j) {
129     // handling the interval [t_j, p_j]
130     double a = t(j), b = p(j), ya = y(j),
131            w = y(j) + 0.5*c(j)*(p(j) - t(j)),
132            yb = ( (t(j+1) - p(j))*(y(j) + 0.5*c(j)*(p(j) - t(j))) +
133                   (p(j) - t(j))*(y(j+1) + 0.5*c(j+1)*(p(j) - t(j+1))) ) / (t(j+1) - t(j));
134
135     // te = points in which we evaluate the interpolant,
136     // pe = values in te
137     ArrayXd te = ArrayXd::LinSpaced(100, a, b),
138             pe = (ya*(b - te).pow(2) + 2*w*(te - a)*(b - te) + yb*(te - a).pow(2))/std::pow(b - a, 2);
139
140     quad.plot(te.matrix(), pe.matrix(), "r");
141
142     // now the same for interval [p_j, t_{j+1}]
143     a = b; b = t(j+1); ya = yb; yb = y(j+1);
144     w = y(j+1) + 0.5*c(j+1)*(p(j) - t(j + 1));
145     te = ArrayXd::LinSpaced(100, a, b);
146     pe = (ya*(b - te).pow(2) + 2*w*(te - a)*(b - te) + yb*(te - a).pow(2))/std::pow(b - a, 2);
147
148     quad.plot(te.matrix(), pe.matrix(), "r");
149   }
150
151   // plot data points
152   quad.plot(t, y, " #mo");
153   quad.title("Quadratic spline");
154   quad.grid(false);
155   quad.save("quadraticspline");
156
157 }
```

5.6 Trigonometric Interpolation

Supplementary reading. This topic is also presented in [**?**, Sect. 8.5].

We consider time series data (t_i, y_i), $i = 0, \ldots, n$, $t_i, y_i \in \mathbb{R}$, obtained by sampling a time-dependent scalar physical quantity $t \mapsto \varphi(t)$. We **know** that φ is a T-periodic function with period $T > 0$, that is $\varphi(t) = \varphi(t + T)$ for all $t \in \mathbb{R}$. In the spirit of shape preservation ($\rightarrow$ Section 5.3) an interpolant f of the time series should also be T-periodic: $f(t + T) = f(t)$ for all $t \in \mathbb{R}$.

Assumption 5.6.1. Sampling in a period

We assume the period $T > 0$ to be known and $t_i \in [0, T[$ for all interpolation nodes t_i, $i = 0, \ldots, n$.

In the sequel, for the case of simplicity, we consider only $T = 1$.

Task: Given data points (t_i, y_i), $y_i \in \mathbb{K}$, $t_i \in [0, T[$, find a 1-periodic function $f : \mathbb{R} \to \mathbb{K}$ (the interpolant), $f(t + T) = f(t)$ $\forall t \in \mathbb{R}$, that satisfies the interpolation conditions

$$f(t_i) = y_i \,, \quad i = 0, \ldots, n \,. \tag{5.6.2}$$

5.6.1 Trigonometric Polynomials

The most fundamental periodic functions are derived from the trigonometric functions sin and cos and dilations of them (A dilation of a function $t \mapsto \psi(t)$ is a function of the form $t \mapsto \psi(ct)$ with some $c > 0$).

Definition 5.6.3. Space of trigonometric polynomials

The vector space of 1-periodic trigonometric polynomials of degree $2n$, $n \in \mathbb{N}$, is given by

$$\mathcal{P}_{2n}^T := \mathrm{Span}\{t \mapsto \cos(2\pi jt), t \mapsto \sin(2\pi jt)\}_{j=0}^{n} \subset C^\infty(\mathbb{R}) .$$

The terminology is natural after recalling expressions for trigonometric functions via complex exponentials ("Euler's formula")

$$e^{it} = \cos t + \imath \sin t \quad \Rightarrow \quad \begin{cases} \cos t = \frac{1}{2}(e^{\imath t} + e^{-\imath t}) \\ \sin t = \frac{1}{2\imath}(e^{\imath t} - e^{-\imath t}) . \end{cases} \tag{5.6.4}$$

Thus we can rewrite $q \in \mathcal{P}_{2n}^T$ given in the form

$$q(t) = \alpha_0 + \sum_{j=1} \alpha_j \cos(2\pi jt) + \beta_j \sin(2\pi jt) , \quad \alpha_j, \beta_j \in \mathbb{R} , \tag{5.6.5}$$

by means of complex exponentials making use of Euler's formula (5.6.4):

$$\begin{aligned} q(t) &= \alpha_0 + \sum_{j=1}^{n} \alpha_j \cos(2\pi jt) + \beta_j \sin(2\pi jt) \\ &= \alpha_0 + \frac{1}{2}\Big\{ \sum_{j=1}^{n} (\alpha_j - \imath\beta_j) e^{2\pi\imath jt} + (\alpha_j + \imath\beta_j) e^{-2\pi\imath jt} \Big\} \\ &= \alpha_0 + \tfrac{1}{2} \sum_{j=-n}^{-1} (\alpha_{-j} + \imath\beta_{-j}) e^{2\pi\imath jt} + \tfrac{1}{2} \sum_{j=1}^{n} (\alpha_j - \imath\beta_j) e^{2\pi\imath jt} \\ &= e^{-2\pi\imath nt} \sum_{j=0}^{2n} \gamma_j e^{2\pi\imath jt} , \quad \text{with} \quad \gamma_j = \begin{cases} \frac{1}{2}(\alpha_{n-j} + \imath\beta_{n-j}) & \text{for } j = 0, \ldots, n-1 , \\ \alpha_0 & \text{for } j = n , \\ \frac{1}{2}(\alpha_{j-n} - \imath\beta_{j-n}) & \text{for } j = n+1, \ldots, 2n . \end{cases} \end{aligned} \tag{5.6.6}$$

Note: $\gamma_j \in \mathbb{C}$ ➢ work in $\mathbb{C}$ in the context of trigonometric interpolation! Admit $y_k \in \mathbb{C}$.

From the above manipulations we conclude

$$q \in \mathcal{P}_{2n+1}^T \quad \Rightarrow \quad q(t) = e^{-2\pi\imath nt} \cdot p(e^{2\pi it}) \quad \text{with} \quad \underbrace{p(z)}_{\text{a polynomial !}} = \sum_{j=0}^{2n} \gamma_j z^j \in \mathcal{P}_{2n} , \tag{5.6.7}$$

and γ_j from (5.6.6).

(After scaling) a trigonometric polynomial of degree $2n$ is a regular polynomial $\in \mathcal{P}_{2n}$ (in $\mathbb{C}$) restricted to the unit circle $\mathbb{S}^1 \subset \mathbb{C}$.

✎ notation: $\mathbb{S}^1 := \{z \in \mathbb{C}: |z| = 1\}$ is the unit circle in the complex plane.

The relationship (5.6.7) justifies calling $\mathcal{P}_{2n}^T$ a space of trigonometric *polynomials* and immediately reveals the dimension of $\mathcal{P}_{2n}^T$:

Corollary 5.6.8. Dimension of $\mathcal{P}_{2n}^T$

The vector space $\mathcal{P}_{2n}^T$ has dimension $\dim \mathcal{P}_{2n}^T = 2n+1$.

5.6.2 Reduction to Lagrange Interpolation

We observed that trigonometric polynomials are standard (complex) polynomials in disguise. Next we can relate trigonometric interpolation to well-known standard Lagrangian interpolation discussed in Section 5.2.2. In fact, we slightly extend the method, because now we admit complex interpolation nodes. All results obtained earlier carry over to this setting.

The key tool is a smooth bijective mapping between $I := [0,1[$ and $\mathbb{S}^1$ defined as

$$\Phi_{\mathbb{S}^1} : I \to \mathbb{S}^1 \quad , \quad t \mapsto z := \exp(2\pi \imath t) \; . \tag{5.6.9}$$

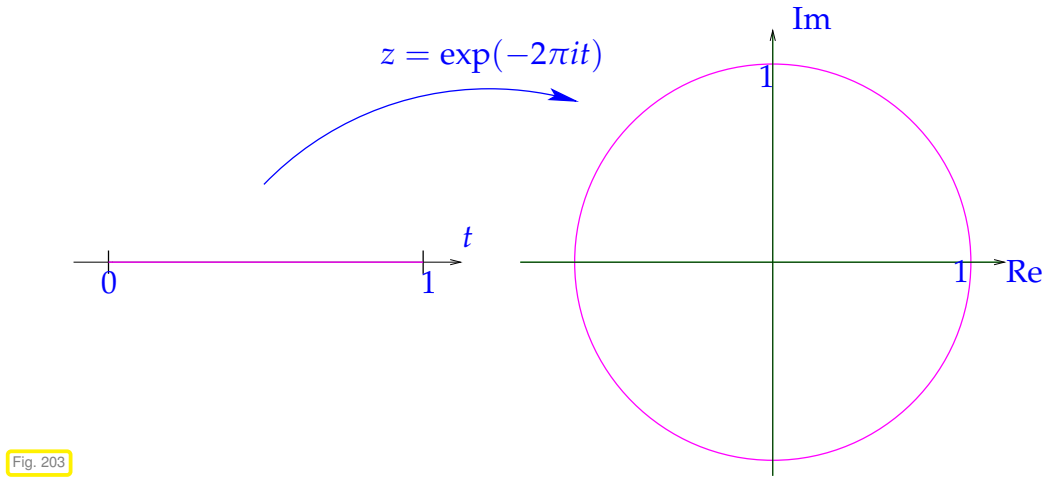


Fig. 203

Trigonometric interpolation through data points (t_k, y_k)	$\Longleftrightarrow$ Polynomial interpolation through data points $(e^{2\pi i t_k}, y_k)$
Trigonometric interpolation of f on I	$\Longleftrightarrow$ Polynomial interpolation of pullback $(\Phi_{\mathbb{S}^1}^{-1})^* f$ on $\mathbb{S}^1$

Here we deal with a non-affine pullback, but the definition is the same as the one given in (6.1.20) for an affine pullback:

$$(\Phi_{\mathbb{S}^1}^{-1})^* : C^0([0,1[) \to C^0(\mathbb{S}^1) \quad , \quad \big((\Phi_{\mathbb{S}^1}^{-1})^* f\big)(z) := f(\Phi_{\mathbb{S}^1}^{-1}(z)) \; , \quad z \in \mathbb{S}^1 \; . \tag{5.6.10}$$

Trigonometric interpolation $\longleftrightarrow$ polynomial interpolation on $\mathbb{S}^1$

All theoretical results and algorithms from polynomial interpolation carry over to trigonometric interpolation

- Existence and uniqueness of trigonometric interpolation polynomial, see Thm. 5.2.14,
- Concept of Lagrange polynomials, see (5.2.11),
- the algorithms and representations discussed in Section 5.2.3.

The next code demonstrates the use of standard routines for polynomial interpolation provided by **BarycPolyInterp** (→ Code 5.2.32) for trigonometric interpolation.

C++-code 5.6.11: Evaluation of trigonometric interpolation polynomial in many points

```
// Evaluation of trigonometric interpolant at numerous points
// IN : t = vector of nodes t_0,...,t_n ∈ [0,1[
// y = vector of data y_0,...,y_n
// x = vector of evaluation points x_1,...,x_N
// OUT : vector of values of the interpolant at points in x
VectorXd trigpolyval(const VectorXd& t, const VectorXd& y, const
    VectorXd& x) {
  using idx_t = VectorXd::Index;
  using comp = std::complex<double>;
  const idx_t N = y.size(); // Number of data points
  if (N % 2 == 0) throw std::runtime_error("Number of points must be
     odd!");
  const idx_t n = (N - 1)/2;
  const std::complex<double> M_I(0,1); // imaginary unit
  // interpolation nodes and evalutation points on unit circle
  VectorXcd tc = ( 2*M_PI*M_I*t ).array().exp().matrix(),
            xc = ( 2*M_PI*M_I*x ).array().exp().matrix();
  // Rescaled values, according to q(t) = e^{-2πint} · p(e^{2πit}), see (5.6.6)
  VectorXcd z = ((2*n*M_PI*M_I*t).array().exp() * y.array()).matrix();
  // Evaluation of interpolating polynomial on unit circle using the
  // barycentric interpolation formula in ℂ, see Code 5.2.29
  BarycPolyInterp<comp> Interpol(tc);
  VectorXcd p = Interpol.eval<VectorXcd>(z,xc);
  // Undo the scaling, see (5.6.7)
  VectorXcd qc = ((-2*n*M_PI*M_I*x).array().exp() *
     p.array()).matrix();
  return (qc.real()); // imaginary part is zero, cut it off
}
```

The computational effort is $O(n^2 + Nn)$, $N \hat{=}$ no. of evaluation points, as for barycentric polynomial interpolation studied in § 5.2.27.

The next code finds the coefficients $\alpha_j, \beta_j \in \mathbb{R}$ of a trigonometric interpolation polynomial in the real-valued representation (5.6.5) for real-valued data $y_j \in \mathbb{R}$ by simply solving the linear system of equations arising from the interpolation conditions (5.6.2).

C++-code 5.6.12: Computation of coefficients of trigonometric interpolation polynomial, general nodes

```
//Computes expansion coefficients of trigonometric polyonomials (5.6.5)
// IN : t = vector of nodes t_0,...,t_n ∈ [0,1[
// y = vector of data y_0,...,y_n
```

```
// OUT: pair of vectors storing the basis expansion coefficients α_j, β_j, see Def. 5.6.3
std::pair<VectorXd,VectorXd>
trigpolycoeff(const VectorXd& t, const VectorXd& y) {
  const unsigned N = y.size(), n = (N-1)/2;
  if (N % 2 == 0) throw "Number of points must be odd!\n";

  // build system matrix M
  MatrixXd M(N, N);
  M.col(0) = VectorXd::Ones(N);
  for (unsigned c = 1; c <= n; ++c) {
    M.col(c) = ( 2*M_PI*c*t ).array().cos().matrix();
    M.col(n + c) = ( 2*M_PI*c*t ).array().sin().matrix();
  }
  // solve LSE and extract coefficients α_j and β_j
  VectorXd c = M.lu().solve(y);
  return std::pair<VectorXd,VectorXd>(c.head(n + 1),c.tail(n));
}
```

The asymptotic computational effort of this implementation is dominated by the cost for Gaussian elimination applied to a fully populated (dense) matrix, see Thm. 2.5.2: $O(n^3)$ for $n \to \infty$.

5.6.3 Equidistant Trigonometric Interpolation

Often timeseries data for a time-periodic quantity are measured with a constant rhythm over the entire (known) period of duration $T > 0$, that is, $t_j = j\Delta t$, $\Delta t = \frac{T}{n+1}$, $j = 0,\ldots,n$. In this case, the formulas for computing the coefficients of the interpolating trigonometric polynomial ($\to$ Def. 5.6.3) become special versions of the discrete Fourier transform (DFT, see Def. 4.2.18) studied in Section 4.2. Efficient implementation can thus harness the speed of FFT introduced in Section 4.3.

Now: 1-periodic setting, uniformly distributed interpolation nodes $t_k = \dfrac{k}{2n+1}$, $k = 0,\ldots,2n$

$\blacktriangleright$ $(2n+1) \times (2n+1)$ linear system of equations:

$$\sum_{j=0}^{2n} \gamma_j \exp\big(2\pi\imath \frac{jk}{2n+1}\big) = (\mathbf{b})_k := \exp\big(2\pi\imath \frac{nk}{2n+1}\big) y_k\,, \quad k = 0,\ldots,2n\,. \tag{5.6.13}$$

$$\Updownarrow$$

$$\overline{\mathbf{F}}_{2n+1}\mathbf{c} = \mathbf{b}\,, \quad \mathbf{c} = [\gamma_0,\ldots,\gamma_{2n}]^\top \quad \overset{\text{Lemma 4.2.14}}{\Rightarrow} \quad \mathbf{c} = \frac{1}{2n+1}\mathbf{F}_{2n+1}\mathbf{b}\,. \tag{5.6.14}$$

$(2n+1) \times (2n+1)$ (conjugate) Fourier matrix, see (4.2.13)

$\blacktriangleright$ Fast solution by means of FFT: $O(n \log n)$ asymptotic complexity, see Section 4.3

C++-code 5.6.15: Efficient computation of coefficient of trigonometric interpolation polynomial (*equidistant nodes*)

```
// Efficient FFT-based computation of coefficients in expansion (5.6.5)
// for a trigonometric interpolation polynomial in equidistant points
```

```
// (j/(2n+1), y_j), j = 0,...,2n.
// IN : y has to be a row vector of odd length, return values are column vectors
// OUT: vectors [α_j]_j, [α_j]_j of expansion coefficients
// with respect to trigonometric basis from Def. 5.6.3
std::pair<VectorXd,VectorXd> trigipequid(const VectorXd& y) {
  using index_t = VectorXcd::Index;
  const index_t N = y.size(), n = (N - 1)/2;
  if (N % 2 != 1) throw "Number of points must be odd!";
  // prepare data for fft
  std::complex<double> M_I(0,1); // imaginary unit
  // right hand side vector b from (5.6.14)
  VectorXcd b(N);
  for (index_t k = 0; k < N; ++k)
    b(k) = y(k) * std::exp(2*M_PI*M_I*(double(n)/N*k));
  Eigen::FFT<double> fft; // DFT helper class
  VectorXcd c = fft.fwd(b); // means that "c = fft(b)"

  // By (5.6.6) we can recover
  // α_j = 1/2(γ_{n-j} + γ_{n+j}) and β_j = 1/2i(γ_{n-j} - γ_{n+j}), j = 1,...,n, α_0 = γ_n.
  VectorXcd alpha(n + 1),beta(n); alpha(0) = c(n)/((double)N);
  for (index_t l = 1; l <= n; ++l) {
    alpha(l) = (c(n-l)+c(n+l))/((double)N);
    beta(l-1) = -M_I*(c(n-l)-c(n+l))/((double)N);
  }
  return std::pair<VectorXd,VectorXd>(alpha.real(),beta.real());
}
```

Example 5.6.16 (Runtime comparison for computation of coefficient of trigonometric interpolation polynomials)

Runtime measurements for MATLAB equivalents of codes Code 5.6.12 and Code 5.6.15

`tic-toc`-timings ▷

MATLAB 7.10.0.499 (R2010a)

CPU: Intel Core i7, 2.66 GHz, 2 cores, L2 256 KB, L3 4 MB

OS: Mac OS X, Darwin Kernel Version 10.5.0

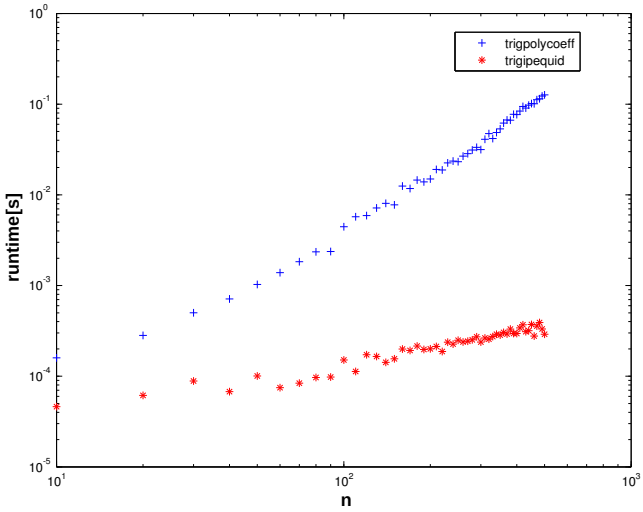


Fig. 204

MATLAB-code 5.6.17: Runtime comparison

```
function trigipequidtiming
% Runtime comparison between efficient (→ Code ??) and direct computation
```

```
% (→ Code ?? of coefficients of trigonoetric interpolation polynomial
    in
% equidistant points.
Nruns = 3; times = [];
for n = 10:10:500
  disp(n)
  N = 2*n+1; t = 0:1/N:(1-1/N); y = exp(cos(2*pi*t));
  t1 = realmax; t2 = realmax;
  for k=1:Nruns
    tic; [a,b] = trigpolycoeff(t,y); t1 = min(t1,toc);
    tic;  [a,b] = trigipequid(y); t2 = min(t2,toc);
  end
  times = [times; n , t1 , t2];
end

figure; loglog(times(:,1),times(:,2),'b+',...
               times(:,1),times(:,3),'r*');
xlabel('{\bf n}','fontsize',14);
ylabel('{\bf runtime[s]}','fontsize',14);
legend('trigpolycoeff','trigipequid','location','best');

print -depsc2 '../PICTURES/trigipequidtiming.eps';
```

Same observation as in Ex. 4.3.12: massive gain in efficiency through relying on FFT.

Remark 5.6.18 (Efficient evaluation of trigonometric interpolation polynomials)

Task: Find an efficient way to evaluate the trigonometric polynomial

$$q(t) = \alpha_0 + \sum_{j=1} \alpha_j \cos(2\pi jt) + \beta_j \sin(2\pi jt) \;, \quad \alpha_j, \beta_j \in \mathbb{R} \;, \tag{5.6.5}$$

at *equidistant* points $\frac{k}{N}$, $N > 2n$. $k = 0, \ldots, N-1$.

As in (5.6.6) we can rewrite

$$q(t) = e^{-2\pi\imath nt} \sum_{j=0}^{2n} \gamma_j e^{2\pi\imath jt} \;, \quad \text{with} \quad \gamma_j = \begin{cases} \frac{1}{2}(\alpha_{n-j} + \imath\beta_{n-j}) & \text{for } j = 0, \ldots, n-1 \;, \\ \alpha_0 & \text{for } j = n \;, \\ \frac{1}{2}(\alpha_{j-n} - \imath\beta_{j-n}) & \text{for } j = n+1, \ldots, 2n \;. \end{cases}$$

(5.6.6) ⇒ $q(k/N) = e^{-2\pi\imath nk/N} \sum_{j=0}^{2n} \gamma_j \exp(2\pi\imath \frac{kj}{N}) \;, \quad k = 0, \ldots, N-1 \;.$

⇒ $q(k/N) = e^{-2\pi\imath kn/N} v_j \quad \text{with} \quad \mathbf{v} = \overline{\mathbf{F}}_N \widetilde{\mathbf{c}}$ (5.6.19)

Fourier matrix, see (4.2.13).

where $\widetilde{\mathbf{c}} \in \mathbb{C}^N$ is obtained by zero padding of $\mathbf{c} := (\gamma_0, \ldots, \gamma_{2n})^T$:

$$(\widetilde{\mathbf{c}})_k = \begin{cases} \gamma_j & \text{, for } k = 0, \ldots, 2n \;, \\ 0 & \text{, for } k = 2n+1, \ldots, N-1 \;. \end{cases}$$

The FFT-based implementation has been fine in the following code.

C++-code 5.6.20: Fast evaluation of trigonometric polynomial at *equidistant* points

```
2  void trigipequidcomp(const VectorXcd& a, const VectorXcd& b, const
       unsigned N, VectorXcd& y) {
3    const unsigned n = a.size() - 1;
4    if (N < (2*n - 1)) {
5      std::cerr << "N is too small! Must be larger than 2*n";
6      return;
7    }
8    const std::complex<double> iu(0,1); // imaginary unit
9    // build vector γ
10   VectorXcd gamma(2*n + 1);
11   gamma(n) = a(0);
12   for (unsigned k = 0; k < n; ++k) {
13     gamma(k) = 0.5*( a(n - k) + iu*b(n - k - 1) );
14     gamma(n + k + 1) = 0.5*( a(k + 1) - iu*b(k) );
15   }
16   // zero padding to obtain c̃
17   VectorXcd ch(N); ch << gamma, VectorXcd::Zero(N - (2*n + 1));
18
19   // realize multiplication with conjugate fourier matrix
20   Eigen::FFT<double> fft;
21   VectorXcd chCon = ch.conjugate();
22   VectorXcd v = fft.fwd(chCon).conjugate();
23
24   // final scaling, implemented without efficiency considerations
25   y = VectorXcd(N);
26   for (unsigned k = 0; k < N; ++k)
27     y(k) = v(k) * std::exp( -2.*k*n*M_PI/N*iu );
28 }
```

The next code merges the steps of computing the coefficient of the trigonometric interpolation polynomial in equidistant points and its evaluation in another set of equidistant points.

C++11 code 5.6.21: *Equidistant* points: fast on the fly evaluation of trigonometric interpolation polynomial

```
2  // Evaluation of trigonometric interpolation polynomial through
       (j/(2n+1), y_j), j = 0,...,2n
3  // in equidistant points k/N, k = 0, N-1
4  // IN : y = vector of values to be interpolated
5  // q (COMPLEX!) will be used to save the return values
6  void trigpolyvalequid(const VectorXd y, const int M, VectorXd& q) {
7    const int N = y.size();
8    if (N % 2 == 0) {
9      std::cerr << "Number of points must be odd!\n";
10     return;
```

```
11    }
12    const int n = (N - 1)/2;
13    // computing coefficient γ_j, see (5.6.14)
14    VectorXcd a, b;
15    trigipequid(y, a, b);
16
17    std::complex<double> i(0,1);
18    VectorXcd gamma(2*n + 1);
19    gamma(n) = a(0);
20    for (int k = 0; k < n; ++k) {
21      gamma(k) = 0.5*( a(n - k) + i*b(n - k - 1) );
22      gamma(n + k + 1) = 0.5*( a(k + 1) - i*b(k) );
23    }
24
25    // zero padding
26    VectorXcd ch(M); ch << gamma, VectorXcd::Zero(M - (2*n + 1));
27
28    // build conjugate fourier matrix
29    Eigen::FFT<double> fft;
30    const VectorXcd chCon = ch.conjugate();
31    const VectorXcd v = fft.fwd(chCon).conjugate();
32
33    // multiplicate with conjugate fourier matrix
34    VectorXcd q_complex = VectorXcd(M);
35    for (int k = 0; k < M; ++k) {
36      q_complex(k) = v(k) * std::exp( -2.*k*n*M_PI/M*i );
37    }
38    // complex part is zero up to machine precision, cut off!
39    q = q_complex.real();
40  }
```

5.7 Least Squares Data Fitting

As remarked in Ex. 5.1.5 the basic assumption underlying the reconstructing of the functional dependence of two quantities by means of interpolation as that of accurate data. In case of data uncertainty or measurement errors the exact satisfaction of interpolation conditions ceases to make sense and we are better

off reconstructing a fitting function that is merely "close to the data" in a sense to be made precise next.

The task of (multidimensional, vector-valued) **least squares data fitting** can be described as follows:

Given: data points $(\mathbf{t}_i, \mathbf{y}_i)$, $i = 1, \ldots, m$, $m \in \mathbb{N}$, $\mathbf{t}_i \in D \subset \mathbb{R}^k$, $\mathbf{y}_i \in \mathbb{R}^d$

Objective: Find a (continuous) function $\mathbf{f} : D \mapsto \mathbb{R}^d$ in some set $S \subset C^0(D)$ of admissible functions satisfying

$$\mathbf{f} \in \operatorname*{argmin}_{\mathbf{g} \in S} \sum_{i=1}^{m} \|\mathbf{g}(\mathbf{t}_i) - \mathbf{y}_i\|_2^2 \,. \tag{5.7.1}$$

The function $\mathbf{f}$ is called the (best) least squares fit for the data in S.

Focus in this section: $k = 1$, $d = 1$ (one-dimensional scalar setting)

$\leftrightarrow$ fitting of *scalar* data depending on *one* parameter ($t \in \mathbb{R}$),

$\leftrightarrow$ Data points (t_i, y_i), $t_i \in I \subset \mathbb{R}$, $y_i \in \mathbb{R}$ $\succ$ $S \subset C^0(I)$

$\leftrightarrow$ (best) least squares fit $f : I \subset \mathbb{R} \to \mathbb{R}$ is a function of one real variable.

$$m = 1, d = 1: \quad (5.7.1) \quad \Leftrightarrow \quad f \in \operatorname*{argmin}_{g \in S} \sum_{i=1}^{m} |g(t_i) - y_i|^2 \,. \tag{5.7.2}$$

(5.7.3) Linear spaces of admissible functions

Consider a special variant of the general least squares data fitting problem: The set S of admissible continuous functions is now chosen as a finite-dimensional vector space $V_n \subset C^0(D)$, $\dim V_n = n \in \mathbb{N}$, *cf.* the discussion in § 5.1.13 for interpolation.

Choose basis of V_n: $V_n = \operatorname{Span}\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, $\mathbf{b}_j : D \to \mathbb{R}^d$ continuous

The best least squares fit $\mathbf{f} \in V_n$ can be represented by a *finite linear combination* of the basis functions $\mathbf{b}_j$:

$$\mathbf{f}(t) = \sum_{j=1}^{n} x_j \mathbf{b}_j(t) \quad , \quad x_j \in \mathbb{R} \,. \tag{5.7.4}$$

Often, in the case $d > 1$, V_n is chosen as a product space

$$V_n = \underbrace{W \times \cdots \times W}_{d \text{ factors}} \,, \tag{5.7.5}$$

of a space $W \subset C^0(D)$ of $\mathbb{R}$-valued functions $D \mapsto \mathbb{R}$. In this case

$$\dim V_n = d \cdot \dim W \,.$$

If $\ell := \dim W$ and $q_1, \ldots, q_\ell$ is a basis of W, then

$$\left\{\mathbf{e}_i q_j\right\}_{\substack{i=1,\ldots,d \\ j=1,\ldots,\ell}} = \{\mathbf{e}_1 q_1, \mathbf{e}_2 q_1, \ldots, \mathbf{e}_d q_1, \mathbf{e}_1 q_2, \ldots, \mathbf{e}_d q_2, \mathbf{e}_1 q_3, \ldots, \mathbf{e}_1 q_\ell, \ldots \mathbf{e}_d q_\ell\} \tag{5.7.6}$$

is a basis of V_n ($\mathbf{e}_i \hat{=} i$-th unit vector).

(5.7.7) Linear data fitting → **[?, Sect. 4.1]**

We adopt the setting of § 5.7.3 of an n-dimensional space V_n of admissible functions with basis $\{\mathbf{b}_q, \ldots, \mathbf{b}_n\}$. Then the least squares data fitting problem can be recast as follows.

General Linear least squares fitting problem:

Given:
- data points $(\mathbf{t}_i, \mathbf{y}_i) \in \mathbb{R}^k \times \mathbb{R}^d, \quad i = 1, \ldots, m$
- basis functions $b_j : D \subset \mathbb{R}^k \mapsto \mathbb{R}$, $j = 1, \ldots, n$, $n < m$

Sought: coefficients $x_j \in \mathbb{R}$, $j = 1, \ldots, n$, such that

$$(x_1, \ldots, x_n) = \operatorname*{argmin}_{z_j \in \mathbb{R}^d} \sum_{i=1}^{m} \left\| \sum_{j=1}^{n} z_j \mathbf{b}_j(\mathbf{t}_i) - \mathbf{y}_i \right\|_2^2 . \tag{5.7.8}$$

Special cases:

- For $k = 1$, $d = 1$, data points $(t_i, y_i) \in \mathbb{R} \times \mathbb{R}$ (scalar, one-dimensional setting), and $V_n = \operatorname{Span}\{b_1, \ldots, b_n\}$, we seek coefficients $x_j \in \mathbb{R}$, $j = 1, \ldots, n$, as the components of a vector $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$ satisfying

$$\mathbf{x} = \operatorname*{argmin}_{\mathbf{z} \in \mathbb{R}^n} \sum_{i=1}^{m} \left| \sum_{j=1}^{n} (\mathbf{z})_j b_j(t_i) - y_i \right|^2 . \tag{5.7.9}$$

- If V_n is a product space according to (5.7.5) with basis (5.7.6), then (5.7.8) amounts to finding vectors $\mathbf{x}_j \in \mathbb{R}^d$, $j = 1, \ldots, \ell$ with

$$(\mathbf{x}_1, \ldots, \mathbf{x}_\ell) = \operatorname*{argmin}_{\mathbf{z}_j \in \mathbb{R}^d} \sum_{i=1}^{m} \left\| \sum_{j=1}^{n} \mathbf{z}_j q_j(\mathbf{t}_i) - \mathbf{y}_i \right\|_2^2 . \tag{5.7.10}$$

Example 5.7.11 (Linear parameter estimation = linear data fitting → **Ex. 3.0.5, Ex. 3.1.5)**

The linear parameter estimation/linear regression problem presented in Ex. 3.0.5 can be recast as a linear data fitting problem with

- $k = n$, $d = 1$, data points $(\mathbf{x}_i, y_i) \in \mathbb{R}^k \times \mathbb{R}$,
- an $k+1$-dimensional space $V_n = \{\mathbf{x} \mapsto \mathbf{a}^\top \mathbf{x} + \beta,\ \mathbf{a} \in \mathbb{R}^k, \beta \in \mathbb{R}\}$ of affine linear admissible functions,
- the choice of basis $\{\mathbf{x} \mapsto (\mathbf{x})_1, \ldots, \mathbf{x} \mapsto (\mathbf{x})_k, \mathbf{x} \mapsto 1\}$.

(5.7.12) Linear data fitting as a lineare least squares problem

Linear (least squares) data fitting leads to an overdetermined linear system of equations for which we seek a least squares solution (→ Def. 3.1.3) as in Section 3.1.1. To see this rewrite

$$\sum_{i=1}^{m}\left\|\sum_{j=1}^{n} z_j \mathbf{b}_j(\mathbf{t}_i) - \mathbf{y}_i\right\|_2^2 = \sum_{i=1}^{m}\left(\sum_{j=1}^{n}\sum_{r=1}^{d}\left(\mathbf{b}_j(\mathbf{t}_i)\right)_r z_j - (\mathbf{y}_i)_r\right)^2 .$$

Theorem 5.7.13. Least squares solution of data fitting problem

The solution $\mathbf{x} = [x_1,\ldots,x_n]^\top \in \mathbb{R}^n$ *of the linear least squares fitting problem* (5.7.8) *is the least squares solution of the overdetermined linear system of equations*

$$\begin{bmatrix}\mathbf{A}_1\\ \vdots \\ \mathbf{A}_d\end{bmatrix}\mathbf{x} = \begin{bmatrix}\mathbf{b}_1\\ \vdots \\ \mathbf{b}_d\end{bmatrix}, \tag{5.7.14}$$

with $\mathbf{A}_r := \begin{bmatrix}(\mathbf{b}_1(\mathbf{t}_1))_r & \ldots & (\mathbf{b}_n(\mathbf{t}_1))_r\\ \vdots & & \vdots \\ (\mathbf{b}_q(\mathbf{t}_m))_r & \ldots & (\mathbf{b}_n(\mathbf{t}_m))_r\end{bmatrix} \in \mathbb{R}^{m,n}$, $\mathbf{b}_r := \begin{bmatrix}(\mathbf{y}_1)_r\\ \vdots \\ (\mathbf{y}_m)_r\end{bmatrix} \in \mathbb{R}^m$, $r = 1,\ldots,d$.

In the one-dimensional, scalar case ($= 1$, $d = 1$) of (5.7.9) the related overdetermined linear system of equations is

$$\begin{bmatrix}b_1(t_1) & \ldots & b_n(t_1)\\ \vdots & & \vdots \\ b_1(t_m) & \ldots & b_n(t_m)\end{bmatrix}\mathbf{x} = \begin{bmatrix}y_1\\ \vdots \\ y_m\end{bmatrix}. \tag{5.7.15}$$

Obviously, for $m = n$ we recover the 1D data interpolation problem from § 5.1.13.

Having reduced the linear least squares data fitting problem to finding the least squares solution of an overdetermined linear system of equations, we can now apply theoretical results about least squares solutions, for instance, Cor. 3.1.22. The key issue is, whether the coefficient matrix of (5.7.15) has full rank n. Of course, this will depend on the location of the t_i.

Lemma 5.7.16. Unique solvability of linear least squares fitting problem

The scalar one-dimensional linear least squares fitting problem (5.7.9) *with* $\dim V_n = n$, V_n *the vector space of admissible functions, has a unique solution,* if and only if *there are* $t_{i_1},\ldots,t_{i_n}$ *such that*

$$\begin{bmatrix}b_1(t_{i_1}) & \ldots & b_n(t_{i_1})\\ \vdots & & \vdots \\ b_1(t_{i_n}) & \ldots & b_n(t_{i_n})\end{bmatrix} \in \mathbb{R}^{n,n} \quad \textit{is invertible,} \tag{5.7.17}$$

which is independent of the choice of basis of V_n.

Equivalent to (5.7.17) is the requirement, that there is an n-subset of $\{t_1, \ldots, t_n\}$ such that the corresponding interpolation problem for V_n has a unique solution for any data values y_i.

Example 5.7.18 (Polynomial fitting)

Special variant of scalar ($d = 1$), one-dimensional $k = 1$ linear data fitting (→ § 5.7.7): we choose the space of admissible functions as polynomials of degree $n - 1$,

$$V_n = \mathcal{P}_{n-1} \quad , \quad \textit{e.g.} \text{ with basis } \quad b_j(t) = t^{j-1} \quad \text{(monomial basis, Section 5.2.1)} .$$

The corresponding overdetermined linear system of equations (5.7.15) now reads:

$$\begin{bmatrix} 1 & t_1 & t_1^2 & \ldots & t_1^{n-1} \\ 1 & t_2 & t_2^2 & \ldots & t_2^{n-1} \\ \vdots & \vdots & v\ldots & & \vdots \\ 1 & t_m & t_m^2 & \ldots & t_m^{n-1} \end{bmatrix} \mathbf{x} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} , \tag{5.7.19}$$

which, for $M \geq n$, has full rank, because it contains invertible Vandermonde matrices (5.2.18), Rem. 5.2.17.

The next code demonstrates the computation of the fitting polynomial with respect to the monomial basis of $\mathcal{P}_{n-1}$:

C++11-code 5.7.20: Polynomial fitting ↪ GITLAB

```
// Solver for polynomial linear least squares data fitting problem
// data points passed in t and y, 'order' = degree + 1
VectorXd polyfit(const VectorXd& t, const VectorXd& y, const
   unsigned& order) {
  // Initialize the coefficient matrix of (5.7.19)
  Eigen::MatrixXd A = Eigen::MatrixXd::Ones(t.size(),order + 1);
  for (unsigned j = 1; j < order + 1; ++j)
    A.col(j) = A.col(j - 1).cwiseProduct(t);
  // Use EIGEN's built-in least squares solver, see Code 3.3.39
  Eigen::VectorXd coeffs = A.householderQr().solve(y);
  // leading coefficients have low indices.
  return coeffs.reverse();
}
```

The function `polyfit` returns a vector $[x_1, x_2, \ldots, x_n]^\top$ describing the fitting polynomial according to the convention

$$p(t) = x_1 t^{n-1} + x_2 t^{n-2} + \cdots + x_{n-1} t + x_n . \tag{5.7.21}$$

Example 5.7.22 (Polybomial interpolation vs. polynomial fitting)

C++-code 5.7.23: Fitting and interpolating polynomial

```
////////////////////////////////////////////////////////////////////////
/// Demonstration code for lecture "Numerical Methods for CSE" @ ETH Zurich
/// (C) 2016 SAM, D-MATH
/// Author(s): Xiaolin Guo, Julien Gacon
/// Repository: https://gitlab.math.ethz.ch/NumCSE/NumCSE/
/// Do not remove this header.
////////////////////////////////////////////////////////////////////////

# include <Eigen/Dense>
# include <figure/figure.hpp>
# include <polyfit.hpp> // NCSE's polyfit (equivalent to Matlab's)
# include <polyval.hpp> // NCSE's polyval (equivalent to Matlab's)

// Comparison of polynomial interpolation and polynomial fitting
// ("Quick and dirty", see 5.2.3)
int main() {
  // use C++ lambda functions to define runge function f(x) = 1/(1+x^2)
  auto f = [](const Eigen::VectorXd& x){
    return ( 1./(1 + x.array()*x.array()) ).matrix();
  };

  const unsigned d = 10; // Polynomial degree
  Eigen::VectorXd tip(d + 1); // d+1 nodes for interpolation
  for (unsigned i = 0; i <= d; ++i)
    tip(i) = -5 + i*10./d;

  Eigen::VectorXd tft(3*d + 1); // 3d+1 nodes for polynomial fitting
  for (unsigned i = 0; i <= 3*d; ++i)
    tft(i) = -5 + i*10./(3*d);

  Eigen::VectorXd ftip = f(Eigen::VectorXd::Ones(2));
  Eigen::VectorXd pip = polyfit(tip, f(tip), d), // Interpolating polynomial (deg = d)
                  pft = polyfit(tft, f(tft), d); // Fitting polynomial (deg = d)

  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(1000,-5,5);
  mgl::Figure fig;
  fig.plot(x, f(x), "g|").label("Function f");
  fig.plot(x, polyval(pip, x), "b").label("Interpolating polynomial");
  fig.plot(x, polyval(pft, x), "r").label("Fitting polynomial");
  fig.plot(tip, f(tip), " b*");
  fig.save("interpfit");

  return 0;
}
```

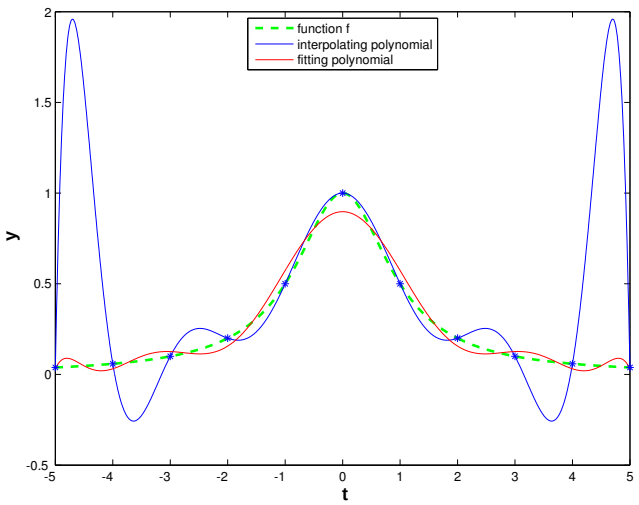


Fig. 205

Data from function $f(t) = \dfrac{1}{1+t^2}$,

- polynomial degree $d = 10$,
- interpolation through data points $(t_j, f(t_j))$, $j = 0, \ldots, d$, $t_j = -5 + j$, see Ex. 5.2.63,
- fitting to data points $(t_j, f(t_j))$, $j = 0, \ldots, 3d$, $t_j = -5 + j/3$.

Fitting helps curb oscillations that haunt polynomial interpolation!

Learning outcomes

After you have studied this chapter you should

- understand the use of basis functions for representing functions on a computer,
- know the concept of a interpolation operator and what its linearity means,
- know the connection between linear interpolation operators and linear systems of equations,
- be familiar with efficient algorithms for polynomial interpolation in different settings,
- know the meaning and significance of "sensitivity" in the context of interpolation,
- Be familiar with the notions of "shape preservation" for an interpolation scheme and its different aspects (monotonicity, curvature),
- know the details of cubic Hermite interpolation and how to ensure that it is monotonicity preserving.
- know what splines are and how cubic spline interpolation with different endpoint constraints works.

Chapter 6

Approximation of Functions in 1D

(6.0.1) General approximation problem

Approximation of functions: Generic view

Given: function $\mathbf{f} : D \subset \mathbb{R}^n \mapsto \mathbb{R}^d$ (often in procedural form **double** `f(double)`, Rem. 5.1.6)

Goal: Find a "SIMPLE" function $\widetilde{\mathbf{f}} : D \mapsto \mathbb{R}^d$ such that the approximation error $\mathbf{f} - \widetilde{\mathbf{f}}$ is "SMALL"

$(*)$: What is "SIMPLE" ?

The function $\widetilde{\mathbf{f}}$ can be encoded by small amount of information and is easy to evaluate.

For instance, this is the case for polynomial or piecewise polynomial $\widetilde{\mathbf{f}}$.

$(\clubsuit)$: What does "SMALL approximation error" mean ?

$\left\|\mathbf{f} - \widetilde{\mathbf{f}}\right\|$ is small for some norm $\|\cdot\|$ on the space $C^0(\overline{D})$ of (piecewise) continous functions.

The most commonly used norms are

- the supremum norm $\|\mathbf{g}\|_\infty := \|\mathbf{g}\|_{L^\infty(D)} := \max_{x\in D} |\mathbf{g}(x)|$, see (5.2.66).

  If the approximation error is small with respect to the supremum norm, $\widetilde{\mathbf{f}}$ is also called a good uniform approximant of $\mathbf{f}$.
- the L^2-norm $\|\mathbf{g}\|_2^2 := \|\mathbf{g}\|_{L^2(D)}^2 = \int_D |\mathbf{g}(x)|^2 dx$, see (5.2.67).

Below we consider only the case $n = d = 1$: approximation of scalar valued functions defined on an interval. The techniques can be applied componentwise in order to cope with the case of vector valued function ($d > 1$).

Example 6.0.3 (Model reduction by interpolation)

The non-linear circuit ($\triangleright|$ stands for a diode, a circuit element, whose resistance strongly depends on the voltage) sketched beside $\triangleright$ has two ports.

For the sake of circuit simulation it should be replaced by a non-linear lumped circuit element characterized by a single voltage-current constitutive relationship $I = I(U)$. For any value of the voltage U the current I can be computed by solving a (large) non-linear system of equations, see Ex. 8.0.1.

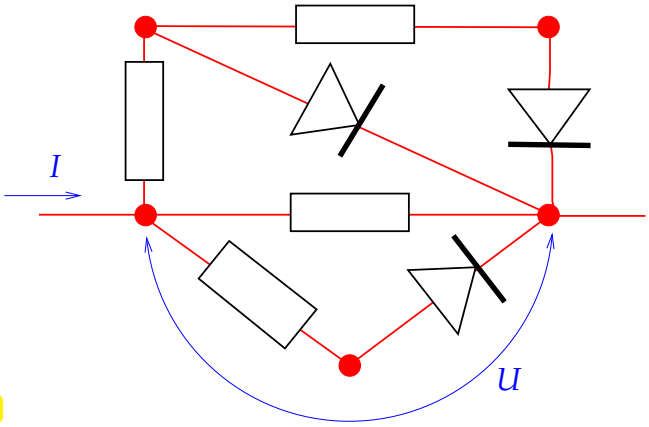


Fig. 206

A faster alternative is the advance approximation of the function $U \mapsto I(U)$ based on a few computed values $I(U_i)$, $i = 0,\ldots,n$, followed by the fast evaluation of the approximant $U \mapsto \widetilde{I}(U)$ during actual circuit simulations. This is an example of model reduction by approximation of functions: a complex subsystem in a mathematical model is replaced by a surrogate function.

In this example we also encounter a typical situation: we have nothing at our disposal but, possibly expensive, point evaluations of the function $U \mapsto I(U)$ ($U \mapsto I(U)$ in "procedural form", see Rem. 5.1.6). The number of evaluations of $I(U)$ will largely determine the cost of building $\widetilde{I}$.

This application displays a fundamental difference compared to the reconstruction of constitutive relationships from a priori measurements → Ex. 5.1.5: Now we are free to choose the number and location of the data points, because we can simply evaluate the function $U \mapsto I(U)$ for any U and as often as needed.

C++11 code 6.0.4: Class describing a 2-port circuit element for circuit simulation

```
class CircuitElement {
  private:
  // internal data describing U ↦ Ĩ(U).
  public:
    // Constructor taking some parameters and building Ĩ
    CircuitElement(const Parameters &P);
    // Point evaluation operators for Ĩ and d/dU Ĩ
    double I(double U) const;
    double dIdU(double U) const;
};
```

(6.0.5) Approximation schemes

We define an abstract concept for the sake of clarity: When in this chapter we talk about an "approximation scheme" (in 1D) we refer to a mapping $A : X \mapsto V$, where X and V are spaces of functions $I \mapsto \mathbb{K}$, $I \subset \mathbb{R}$ an interval.

Examples are

- $X = C^k(I)$, the spaces of functions $I \mapsto \mathbb{K}$ that are k times continuously differentiable, $k \in \mathbb{N}$.
- $V = \mathcal{P}_m(I)$, the space of polynomials of degree $\leq k$, see Section 5.2.1
- $V = \mathcal{S}_{d,\mathcal{M}}$, the space of splines of degree d on the knot set $\mathcal{M} \subset I$, see Def. 5.5.1.

- $V = \mathcal{P}_{2n}^T$, the space of trigonometric polynomials of degree $2n$, see Def. 5.6.3.

(6.0.6) Approximation by interpolation

In Chapter 5 we discussed ways to construct functions whos graph runs through given data points, see 5.1. We can hope that the interpolant will approximate the function, if the data points are also located on the graph of that function. Thus every interpolation scheme, see § 5.1.4, spawns a corresponding approximation scheme.

Interpolation scheme + sampling → approximation scheme

$$f : I \subset \mathbb{R} \to \mathbb{K} \xrightarrow{\text{sampling}} (t_i, y_i := f(t_i))_{i=0}^m \xrightarrow{\text{interpolation}} \widetilde{f} := \mathrm{I}_{\mathcal{T}}\mathbf{y} \quad (\widetilde{f}(t_i) = y_i)\,.$$

free choice of nodes $t_i \in I$

In this chapter we will mainly study approximation by interpolation relying on the interpolation schemes (→ § 5.1.4) introduced in Section 5.2, Section 5.4, and Section 5.5.

There is *additional freedom* compared to data interpolation: we can choose the interpolation nodes in a smart way in order to obtain an accurate interpolant $\widetilde{f}$.

Remark 6.0.8 (Interpolation and approximation: enabling technologies)

Approximation and interpolation (→ Chapter 5) are key components of many numerical methods, like for integration, differentiation and computation of the solutions of differential equations, as well as for computer graphics and generation of smooth curves and surfaces.

This chapter is a "foundations" part of the course

Contents

6.1 Approximation by Global Polynomials

The space $\mathcal{P}_k$ of polynomials of degree $\leq k$ has been introduced in Section 5.2.1. For reasons listed in § 5.2.3 polynomials are the most important theoretical and practical tool for the approximation of functions. The next example presents an important case of approximation by polynomials.

Example 6.1.1 (Taylor approximation → [?, Sect. 5.5])

The local approximation of sufficiently smooth functions by polynomials is a key idea in calculus, which manifests itself in the importance of approximation by Taylor polynomials: For $f \in C^k(I)$, $k \in \mathbb{N}$, $I \subset \mathbb{R}$ an interval, we approximate

$$f(t) \approx \underbrace{\sum_{j=0}^{k} \frac{f^{(j)}(t_0)}{j!}\,(t-t_0)^j}_{=:T_k(t)} \quad , \quad \text{for some } t_0 \in I \, .$$

✎ Notation: $f^{(k)} \triangleq k$-th derivative of function $f : I \subset \mathbb{R} \to \mathbb{K}$

The Taylor polynomial T_k of degree k approximates f in a neighbourhood $J \subset I$ of t_0 (J can be small!). This can be quantified by the remainder formulas [**?**, Bem. 5.5.1]

$$f(t) - T_k(t) = \int_{t_0}^{t} f^{(k+1)}(\tau)\frac{(t-\tau)^k}{k!}\,\mathrm{d}\tau \tag{6.1.2a}$$

$$= f^{(k+1)}(\xi)\frac{(t-t_0)^{k+1}}{(k+1)!} \quad , \quad \xi = \xi(t,t_0) \in]\min(t,t_0), \max(t,t_0)[\, , \tag{6.1.2b}$$

which shows that for $f \in C^{k+1}(I)$ the Taylor polynomial T_k is pointwise close to $f \in C^{k+1}(I)$, if the interval I is small and $f^{(k+1)}$ is bounded pointwise.

Approximation by Taylor polynomials is easy and direct but inefficient: a polynomial of lower degree often gives the same accuracy. Moreover, when f is available only in procedural form as **double** `f(double)`, (approximations of) higher order derivatives are difficult to obtain.

(6.1.3) Nested approximation spaces of polynomials

Obviously, for every interval $I \subset \mathbb{R}$, the spaces of polynomials are nested in the following sense:

$$\mathcal{P}_0 \subset \mathcal{P}_1 \subset \cdots \subset \mathcal{P}_m \subset \mathcal{P}_{m+1} \subset \cdots \subset C^\infty(I) \,, \tag{6.1.4}$$

with finite, but increasing dimensions $\dim \mathcal{P}_m = m+1$ according to Thm. 5.2.2.

With this family of nested spaces of polynomials at our disposal, it is natural to study *associated families of approximation schemes*, one for each degree, mapping into $\mathcal{P}_m$, $m \in \mathbb{N}_0$.

6.1.1 Polynomial approximation: Theory

(6.1.5) Scope of polynomial approximation

Sloppily speaking, according to (6.1.2b) the Taylor polynomials from Ex. 6.1.1 provide *uniform* (→ § 6.0.1) approximation of a *smooth* function f in (small) intervals, provided that its derivatives do not blow up "too fast" (We do not want to make this precise here).

The question is, whether polynomials still offer uniform approximation on arbitrary bounded closed intervals and for functions that are merely continuous, but not any smoother. The answer is YES and this profound result is known as the Weierstrass Approximation Theorem. Here we give an extended version with a concrete formula due to Bernstein, see [**?**, Section 6.2].

Theorem 6.1.6. Uniform approximation by polynomials

For $f \in C^0([0,1])$, define the n-th Bernstein approximant as

$$p_n(t) = \sum_{j=0}^{n} f(j/n) \binom{n}{j} t^j (1-t)^{n-j} \,, \quad p_n \in \mathcal{P}_n \,. \tag{6.1.7}$$

It satisfies $\|f - p_n\|_\infty \to 0$ for $n \to \infty$. If $f \in C^m([0,1])$, then even $\left\| f^{(k)} - p_n^{(k)} \right\|_\infty \to 0$ for $n \to \infty$ and all $0 \le k \le m$.

✎ Notation: $g^{(k)} \triangleq k$-th derivative of a function $g : I \subset \mathbb{R} \to \mathbb{K}$.

In (6.1.7) the function f is approximated by a linear combination of Bernstein polynomials of degree n

$$B_j^n(t) = \binom{n}{j} t^j (1-t)^{n-j} \,, \quad p_n \in \mathcal{P}_n \,. \tag{6.1.8}$$

Plots of Bernstein polynomials of degree $n = 7$ ▷

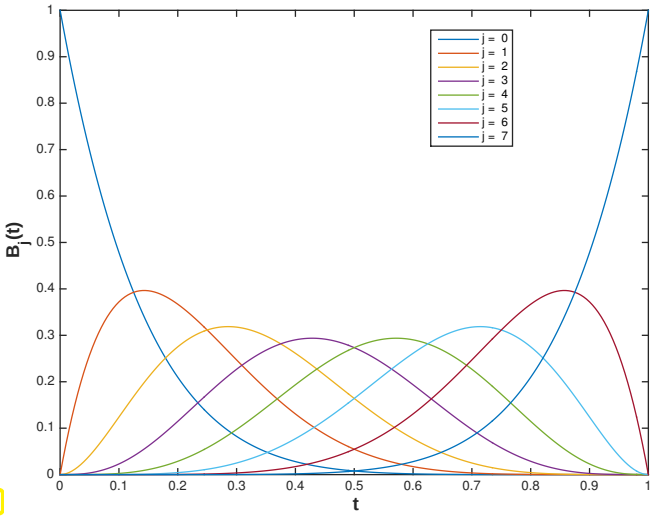


Fig. 207

Bernstein polynomials satisfy:

$$\sum_{j=0}^{n} B_j^n(t) \equiv 1 \,, \tag{6.1.9}$$

$$0 \le B_j^n(t) \le 1 \quad \forall 0 \le t \le 1 \,. \tag{6.1.10}$$

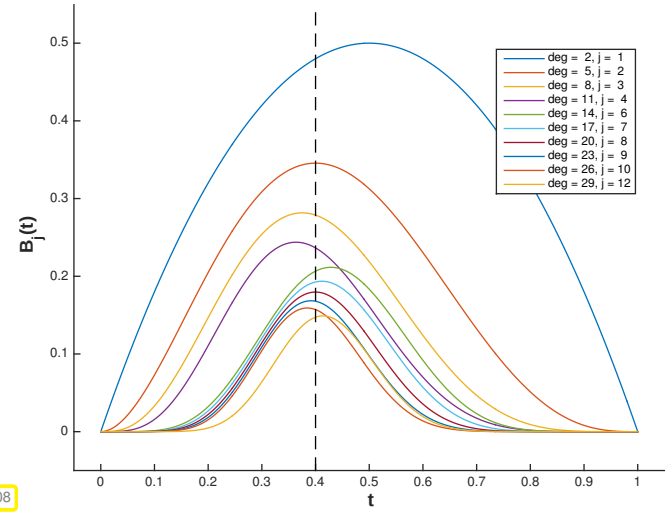


Fig. 208

◁ Since $\frac{d}{dt}B_j^n(t) = B_j^n(t)(\frac{j}{t} - \frac{n-j}{1-t})$, B_j^n has its unique local maximum in $[0,1]$ at the site $t_{\max} := \frac{j}{n}$. As $n \to \infty$ the Bernstein polynomials become more and more concentrated around the maximum.

Proof. (of Thm. 6.1.6, first part) Fix $t \in [0,1]$. Using the notations from (6.1.7) and the identity (6.1.9) we find

$$f(t) - p_n(t) = \sum_{k=0}^{n} (f(t) - f(j/n)) B_j^n(t) \,. \tag{6.1.11}$$

As we see from Fig. 208, for large n the bulk of sum will be contributed by Bernstein polynomial with index $j/n \approx x$, because for every $\delta > 0$

$$\sum_{|j/n-t|>\delta} B_j^n(t) \le \frac{1}{\delta^2} \sum_{|j/n-t|>\delta} (j/n - t)^2 B_j^n(t) \le \frac{1}{\delta^2} \sum_{j=0}^{n} (j/n - t)^2 B_j^n(t) \overset{(*)}{=} \frac{nt(1-t)}{\delta^2 n^2} \le \frac{1}{4n\delta^2} \,.$$

$\sum_{|j/n-t|>\delta}$ means summation over $j \in \mathbb{N}_0$ with summation indices confined to the set $\{j : |j/n - t| > \delta\}$.
The identity $(*)$ can be established by direct but tedious computations.

Combining this estimate with (6.1.10) and (6.1.11) we arrive at

$$|f(t) - p_n(t)| \le \sum_{|j/n-t|>\delta} \frac{1}{4n\delta^2} |f(t) - f(j/n)| + \sum_{|j/n-t|\le\delta} |f(t) - f(j/n)| \,.$$

Since, f is uniformly continuous on $[0,1]$, given $\epsilon > 0$ we can choose $\delta > 0$ independently of t such that $|f(s) - f(t)| < \epsilon$, if $|s - t| < \delta$. The, if we choose $n > (\epsilon\delta^2)^{-1}$, we can bound

$$|f(t) - p_n(t)| \le (\|f\|_\infty + 1)\epsilon \quad \forall t \in [0,1] \,.$$

□

Experiment 6.1.12 (Bernstein approximants)

We compute and plot p_n, $n = 1, \ldots, 25$, for two functions

$$f_1(t) := \begin{cases} 0 & \text{, if } |2t-1| > \frac{1}{2}\,, \\ \frac{1}{2}(1+\cos(2\pi(2t-1))) & \text{else,} \end{cases} \quad , \quad f_2(t) := \frac{1}{1+e^{-12(x-1/2)}}\,.$$

The following plots display the sequences of the polynomials p_n for $n = 2, \ldots, 25$.

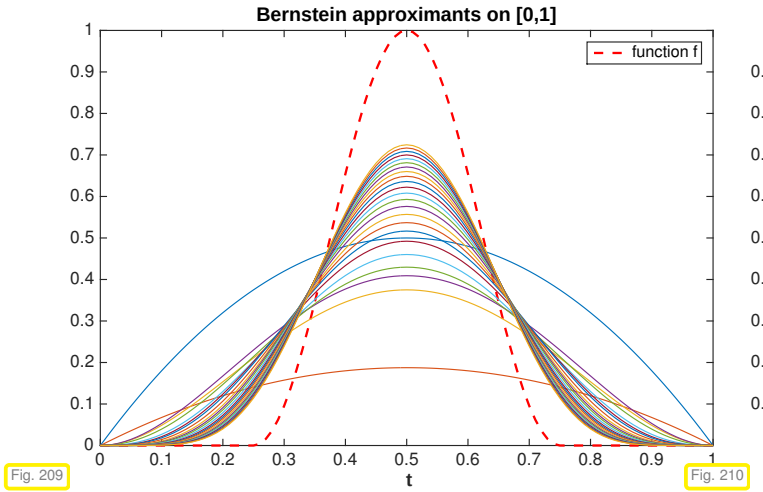


Fig. 209

$f = f_1$

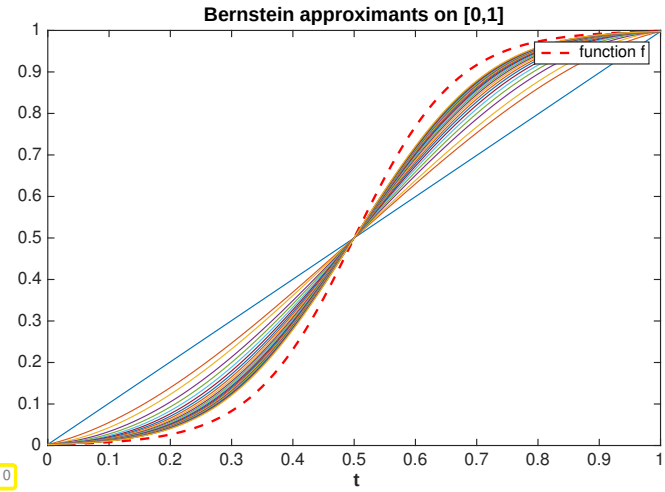


Fig. 210

$f = f_2$

We see that the Bernstein approximants "slowly" edge closer and closer to f. Apparently it takes a very large degree to get really close to f.

(6.1.13) Best approximation

Now we introduce a concept needed to gauge how close an approximation scheme gets to the best possible performance.

Definition 6.1.14. (Size of) best approximaton error

Let $\|\cdot\|$ be a (semi-)norm on a space X of functions $I \mapsto \mathbb{K}$, $I \subset \mathbb{R}$ an interval. The (size of the) best approximation error of $f \in X$ in the space $\mathcal{P}_k$ of polynomials of degree $\leq k$ with respect to $\|\cdot\|$ is

$$\operatorname{dist}_{\|\cdot\|}(f, \mathcal{P}_k) := \inf_{p \in \mathcal{P}_k} \|f - p\|\,.$$

The notation $\operatorname{dist}_{\|\cdot\|}$ is motivated by the notation of "distance" as distance to the nearest point in a set.

For the L^2-norm $\|\cdot\|_2$ and the supremum norm $\|\cdot\|_\infty$ the best approximation error is well defined for

$C = C^0(I)$.

The polynomial realizing best approximation w.r.t. $\|\cdot\|$ may neither be unique nor computable with reasonable effort. Often one is content with rather sharp upper bounds like those asserted in the next theorem, due to Jackson [**?**, Thm. 13.3.7].

Theorem 6.1.15. L^∞ polynomial best approximation estimate

If $f \in C^r([-1,1])$ (r times continuously differentiable), $r \in \mathbb{N}$, then, for any polynomial degree $n \geq r$,

$$\inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([-1,1])} \leq (1+\pi^2/2)^r \frac{(n-r)!}{n!} \left\|f^{(r)}\right\|_{L^\infty([-1,1])} .$$

As above, $f^{(r)}$ stands for the r-th derivative of f. Using Stirling's formula

$$\sqrt{2\pi}\, n^{n+1/2}e^{-n} \leq n! \leq e\, n^{n+1/2}e^{-n} \quad \forall n \in \mathbb{N} , \tag{6.1.16}$$

we can get a looser bound of the form

$$\inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([-1,1])} \leq C(r) n^{-r} \left\|f^{(r)}\right\|_{L^\infty([-1,1])} , \tag{6.1.17}$$

with $C(r)$ dependent on r, but *independent of* f and, in particular, the polynomial *degree* n. Using the Landau symbol from Def. 1.4.5 we can rewrite the statement of (6.1.17) in asymptotic form

$$\text{(6.1.17)} \quad \Rightarrow \quad \inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([-1,1])} = O(n^{-r}) \quad \text{for} \quad n \to \infty .$$

Remark 6.1.18 (Transformation of polynomial approximation schemes)

What if a polynomial approximation scheme is defined only on a special interval, say $[-1,1]$. Then by the following trick it can be transferred to any interval $[a,b] \subset \mathbb{R}$.

Assume that an interval $[a,b] \subset \mathbb{R}$, $a < b$, and a polynomial approximation scheme $\widehat{\mathrm{A}} : C^0([-1,1]) \to \mathcal{P}_n$ are given. Based on the affine linear mapping

$$\Phi : [-1,1] \to [a,b] \quad , \quad \Phi(\hat{t}) := a + \tfrac{1}{2}(\hat{t}+1)(b-a) , \quad -1 \leq \hat{t} \leq 1 , \tag{6.1.19}$$

we can introduce the affine pullback of functions:

$$\boxed{\Phi^* : C^0([a,b]) \to C^0([-1,1]) \quad , \quad \Phi^*(f)(\hat{t}) := f(\Phi(\hat{t})) , \quad -1 \leq \hat{t} \leq 1} . \tag{6.1.20}$$

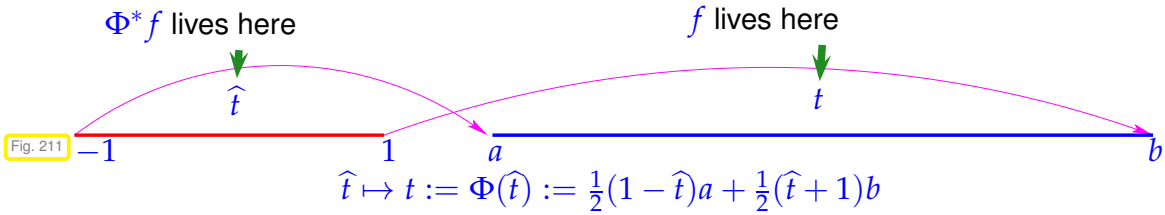


Fig. 211

We add the important observations that affine pullbacks are linear and bijective, they are isomorphisms of the involved vector spaces of functions (what is the inverse?).

Lemma 6.1.21. Affine pullbacks preserve polynomials

If $\Phi^* : C^0([a,b]) \to C^0([-1,1])$ is an affine pullback according to (6.1.19) *and* (6.1.20)*, then $\Phi^* : \mathcal{P}_n \to \mathcal{P}_n$ is a bijective linear mapping for any $n \in \mathbb{N}_0$.*

Proof. This is a consequence of the fact that translations and dilations take polynomials to polynomials of the *same* degree: for monomials we find

$$\Phi^*\{t \to t^n\} = \{\hat{t} \to (a + \tfrac{1}{2}(\hat{t}+1)(b-a))^n\} \in \mathcal{P}_n \,.$$

□

The lemma tells us that the spaces of polynomials of some maximal degree are *invariant under affine pullback*. Thus, we can *define* a *polynomial* approximation scheme A on $C^0([a,b])$ by

$$\mathsf{A} : C^0([a,b]) \to \mathcal{P}_n \quad , \quad \boxed{\mathsf{A} := (\Phi^*)^{-1} \circ \widehat{\mathsf{A}} \circ \Phi^*} \,, \tag{6.1.22}$$

whenever $\widehat{\mathsf{A}}$ is a polynomial approximation scheme on $[-1,1]$.

Remark 6.1.23 (Transforming approximation error estimates)

Thm. 6.1.15 targets only the special interval $[-1,1]$. What does it imply for polynomial best approximation on a general interval $[a,b]$? To answer this question we apply techniques from Rem. 6.1.18, in particular the pullback (6.1.20).

We first have to study the change of norms of functions under the action of affine pullbacks:

Lemma 6.1.24. Transformation of norms under affine pullbacks

For every $f \in C^0([a,b])$ we have

$$\|f\|_{L^\infty([a,b])} = \|\Phi^* f\|_{L^\infty([-1,1])} \quad , \quad \|f\|_{L^2([a,b])} = \sqrt{|b-a|}\, \|\Phi^* f\|_{L^2([-1,1])} \,. \tag{6.1.25}$$

Proof. The first estimate should be evident, and the second is a consequence of the transformation formula for integrals [**?**, Satz 6.1.5]

$$\int_{-1}^{1} \Phi^* t(\widehat{t})\, \mathrm{d}\widehat{t} = \frac{b-a}{2} \int_a^b f(t)\, \mathrm{d}t \,, \tag{6.1.26}$$

and the definition of the L^2-norm from (5.2.67).

□

Thus, for norms of the approximation errors of polynomial approximation schemes defined by affine trans-

formation (6.1.22) we get

$$\begin{aligned} \|f - \mathsf{A}f\|_{L^\infty([a,b])} &= \left\|\Phi^* f - \widehat{\mathsf{A}}(\Phi^* f)\right\|_{L^\infty([-1,1])} , \\ \|f - \mathsf{A}f\|_{L^2([a,b])} &= \sqrt{|b-a|}\left\|\Phi^* f - \widehat{\mathsf{A}}(\Phi^* f)\right\|_{L^2([-1,1])} , \end{aligned} \quad , \quad \forall f \in C^0([a,b]) . \tag{6.1.27}$$

Equipped with approximation error estimates for A, we can infer corresponding estimates for $\widehat{\mathsf{A}}$.

The bounds for approximation errors often involve norms of derivatives as in Thm. 6.1.15. Hence, it is important to understand the interplay of pullback and differentiation: By the 1D chain rule

$$\frac{d}{d\hat{t}}(\Phi^* f)(\hat{t}) = \frac{df}{dt}(\Phi(\hat{t}))\frac{d\Phi}{d\hat{t}} = \frac{df}{dt}(\Phi(\hat{t}))\cdot\tfrac{1}{2}(b-a) ,$$

which implies a simple scaling rule for derivatives of arbitrary order $r \in \mathbb{N}_0$:

$$(\Phi^* f)^{(r)} = \left(\frac{b-a}{2}\right)^r \Phi^*(f^{(r)}) . \tag{6.1.28}$$

Lemma 6.1.24

$$\left\|(\Phi^* f)^{(r)}\right\|_{L^\infty([-1,1])} = \left(\frac{b-a}{2}\right)^r \left\|f^{(r)}\right\|_{L^\infty([a,b])} , \quad f \in C^r([a,b]), r \in \mathbb{N}_0 . \tag{6.1.29}$$

Remark 6.1.30 (Polynomial best approximation on general intervals)

The estimate (6.1.28) together with Thm. 6.1.15 paves the way for bounding the polynomial best approximation error on arbitrary intervals $[a,b]$, $a,b \in \mathbb{R}$. Based on the affine mapping $\Phi : [-1,1] \to [a,b]$ from (6.1.19) and writing Φ^* for the pullback according to (6.1.20) we can chain estimates. If $f \in C^r([a,b])$ and $n \geq r$, then

$$\begin{aligned} \inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([a,b])} &\overset{(*)}{=} \inf_{p\in\mathcal{P}_n} \|\Phi^* f - p\|_{L^\infty([-1,1])} \\ &\overset{\text{Thm. 6.1.15}}{\leq} (1+\pi^2/2)^r \frac{(n-r)!}{n!}\left\|(\Phi^* f)^{(r)}\right\|_{L^\infty([-1,1])} \\ &\overset{(6.1.28)}{=} (1+\pi^2/2)^r \frac{(n-r)!}{n!}\left(\frac{b-a}{2}\right)^r \left\|f^{(r)}\right\|_{L^\infty([a,b])} . \end{aligned}$$

In step $(*)$ we used the result of Lemma 6.1.21 that $\Phi^* p \in \mathcal{P}_n$ for all $p \in \mathcal{P}_n$. Invoking the arguments that gave us (6.1.17), we end up with the simpler bound

$$\inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([a,b])} \leq C(r)\left(\frac{b-a}{n}\right)^r \left\|f^{(r)}\right\|_{L^\infty([a,b])} . \tag{6.1.31}$$

Observe that the length of the interval enters the bound in r-th power.

6.1.2 Error estimates for polynomial interpolation

In Section 5.2.2, Cor. 5.2.15, we introduced the Lagrangian polynomial interpolation operator $\mathsf{I}_\mathcal{T} : \mathbb{K}^{n+1} \to \mathcal{P}_n$ belonging to a node set $\mathcal{T} = \{t_j\}_{j=0}^n$. In the spirit of § 6.0.6 it induces an approximation scheme on $C^0(I)$, $I \subset \mathbb{R}$ an interval, if $\mathcal{T} \subset I$.

Definition 6.1.32. Lagrangian (interpolation polynomial) approximation scheme

Given an interval $I \subset \mathbb{R}$, $n \in \mathbb{N}$, a node set $\mathcal{T} = \{t_0, \ldots, t_n\} \subset I$, the Lagrangian (interpolation polynomial) approximation scheme $\mathsf{L}_{\mathcal{T}} : C^0(I) \to \mathcal{P}_n$ is defined by

$$\mathsf{L}_{\mathcal{T}}(f) := \mathsf{I}_{\mathcal{T}}(\mathbf{y}) \in \mathcal{P}_n \quad \text{with} \quad \mathbf{y} := (f(t_0), \ldots, f(t_n))^T \in \mathbb{K}^{n+1}.$$

Our goal in this section will be

to estimate the norm of the interpolation error $\|f - \mathsf{I}_{\mathcal{T}} f\|$ (for relevant norm on $C(I)$).

(6.1.33) Families of Lagrangian interpolation polynomial approximation schemes

Already Thm. 6.1.15 considered the size of the best approximation error in $\mathcal{P}_n$ as a function of the polynomial degree n. In the same vein, we may study a *family* of Lagrange interpolation schemes $\{\mathsf{L}_{\mathcal{T}_n}\}_{n\in\mathbb{N}_0}$ on $I \subset \mathbb{R}$ induced by a *family of node sets* $\{\mathcal{T}_n\}_{n\in\mathbb{N}_0}$, $\mathcal{T}_n \subset I$, according to Def. 6.1.32.

An example for such a family of node sets on $I := [a, b]$ are the equidistant or equispaced nodes

$$\mathcal{T}_n := \{t_j^{(n)} := a + (b-a)\frac{j}{n} : j = 0, \ldots, n\} \subset I\,. \tag{6.1.34}$$

For families of Lagrange interpolation schemes $\{\mathsf{L}_{\mathcal{T}_n}\}_{n\in\mathbb{N}_0}$ we can shift the focus onto estimating the asymptotic behavior of the norm of the interpolation error for $n \to \infty$.

Experiment 6.1.35 (Asymptotic behavior of Lagrange interpolation error)

We perform polynomial interpolation of $f(t) = \sin t$ on equispaced nodes in $I = [0, \pi]$: $\mathcal{T}_n = \{j\pi/n\}_{j=0}^n$. Write p for the polynomial interpolant: $p := \mathsf{L}_{\mathcal{T}_n} f \in \mathcal{P}_n$.

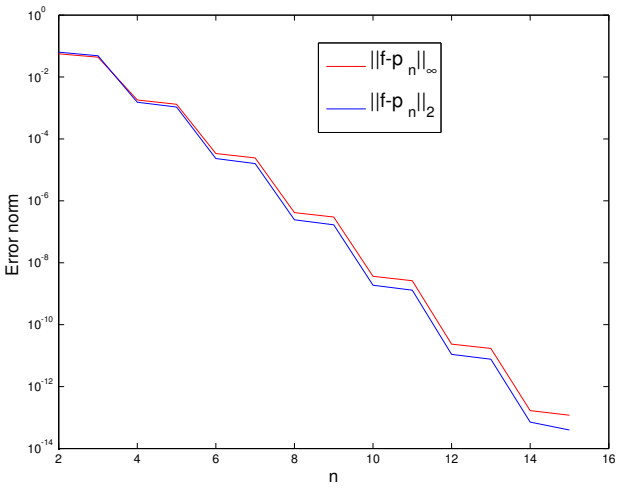


Fig. 212

In the numerical experiment the norms of the interpolation errors can be computed only approximately as follows.

- L^∞-norm: approximated by sampling on a grid of meshsize $\pi/1000$.
- L^2-norm: numerical quadrature (→ Chapter 7) with trapezoidal rule (7.4.4) on a grid of meshsize $\pi/1000$.

◁ approximate norms $\|f - \mathsf{L}_{\mathcal{T}_n} f\|_*$, $* = 2, \infty$.

(6.1.36) Classification of asymptotic behavior of norms of the interpolation error

In the previous experiment we observed a clearly visible regular behavior of $\|f - \mathrm{L}_{\mathcal{T}_n} f\|$ as we increased the polynomial degree n. The prediction of the decay law for $\|f - \mathrm{L}_{\mathcal{T}_n} f\|$ for $n \to \infty$ is one goal in the study of interpolation errors.

Often this goal can be achieved, even if a rigorous quantitative bound for a norm of the interpolation error remains elusive. In other words, in many cases

> no bound for $\|f - \mathrm{L}_{\mathcal{T}_n} f\|$ can be given, but its decay for increasing n can be described precisely.

Now we introduce some important terminology for the qualitative description of the behavior of $\|f - \mathrm{L}_{\mathcal{T}_n} f\|$ as a function of the polynomial degree n. We assume that

$$\exists\, C \neq C(n) > 0: \quad \|f - \mathrm{L}_{\mathcal{T}_n} f\| \leq C\, T(n) \quad \text{for } n \to \infty\,. \tag{6.1.37}$$

Definition 6.1.38. Types of asymptotic convergence of approximation schemes

Writing $T(n)$ for the bound of the norm of the interpolation error according to (6.1.37) we distinguish the following *types of asymptotic behavior*:

$$\begin{array}{lll} \exists\, p > 0: & T(n) \leq n^{-p} & : \text{ algebraic convergence, with rate } p > 0\,, \\ \exists\, 0 < q < 1: & T(n) \leq q^n & : \text{ exponential convergence}\,, \end{array} \quad \forall n \in \mathbb{N}\,.$$

The bounds are assumed to be sharp in the sense, that no bounds with larger rate p (for algebraic convergence) or smaller q (for exponential convergence) can be found.

Convergence behavior of norms of the interpolation error is often expressed by means of the Landau-O-notation, *cf.* Def. 1.4.5:

$$\begin{array}{ll} \text{Algebraic convergence:} & \|f - \mathrm{I}_{\mathcal{T}} f\| = O(n^{-p}) \\ \text{Exponential convergence:} & \|f - \mathrm{I}_{\mathcal{T}} f\| = O(q^n) \end{array} \quad \text{for } n \to \infty \text{ ("asymptotic!")}$$

Remark 6.1.39 (Different meanings of "convergence")

Beware: same concept $\leftrightarrow$ different meanings:

- convergence of a sequence (e.g. of iterates $\boldsymbol{x}^{(k)}$ $\to$ Section 8.1)
- convergence of an approximation (dependent on an approximation parameter, e.g. n)

Remark 6.1.40 (Determining the type of convergence in numerical experiments $\to$ § 1.4.9)

Given pairs (n_i, ϵ_i), $i = 1, 2, 3, \ldots$, $n_i \mathrel{\hat{=}}$ polynomial degrees, $\epsilon_i \mathrel{\hat{=}}$ (measured) norms of interpolation errors, how can be tease out the likely type of convergence according to Def. 6.1.38? A similar task was already encountered in § 1.4.9, where we had to extract information about asymptotic complexity from runtime measurements.

❶ Conjectured: algebraic convergence: $\epsilon_i \approx C n^{-p}$

$$\log(\epsilon_i) \approx \log(C) - p \log n_i \qquad \text{(affine linear in log-log scale).}$$

Apply linear regression from Ex. 3.1.5 for data points $(\log n_i, \log \epsilon_i)$ ➢ least squares estimate for rate p.

❶ Conjectured: exponential convergence: $\epsilon_i \approx C\exp(-\beta n_i)$

$$\log \epsilon_i \approx \log(C) - \beta n_i \quad \text{(affine linear in lin-log scale). .}$$

Apply linear regression (→ Ex. 3.1.5)to points $(n_i, \log \epsilon_i)$ ➢ estimate for $q := \exp(-\beta)$.

☞ Fig. 212: we suspect exponential convergence in Exp. 6.1.35.

Example 6.1.41 (Runge's example → Ex. 5.2.63)

We examine the polynomial interpolant of $f(t) = \frac{1}{1+t^2}$ for equispaced nodes:

$$\mathcal{T}_n := \left\{ t_j := -5 + \tfrac{10}{n} j \right\}_{j=0}^{n}, \quad j = 0, \ldots, n \quad ➢ \quad y_j = \frac{1}{1+t_j^2}.$$

We rely on an approximate computation of the supremum norm of the interpolation error by means of sampling as in Exp. 6.1.35.

C++11 code 6.1.42: Computing the interpolation error for Runge's example

```
// Note:  "quick & dirty" implementation !
// Lambda function representing x ↦ (1+x^2)^-1
auto f = [](double x) { return 1./(1 + x*x); };
// sampling points for approximate maximum norm
const VectorXd x = VectorXd::LinSpaced(1000, -5, 5);
// Sample function
const VectorXd fx = feval(f, x); // evaluate f at x

std::vector<double> err; // Accumulate error norms here
for (int d = 1; d <= 20; ++d) {
  // Interpolation nodes
  const VectorXd t = Eigen::VectorXd::LinSpaced(d + 1, -5, 5);
  // Interpolation data values
  const VectorXd ft = feval(f, t);
  // Compute interpolating polynomial
  const VectorXd p = polyfit(t, ft, d);
  // Evaluate polynomial interpolant
  const VectorXd y = polyval(p, x);
  // Approximate supremum norm of interpolation error
  err.push_back( (y - fx).cwiseAbs().maxCoeff() );
}
```

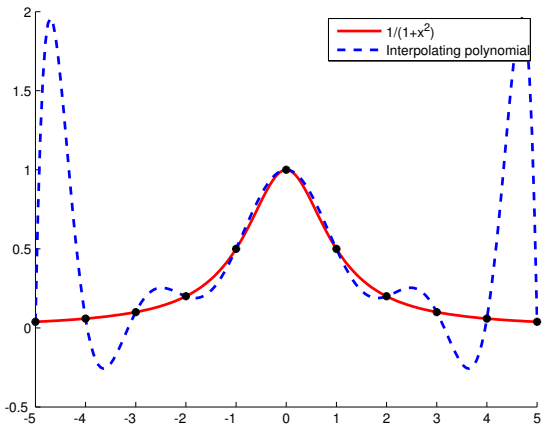


Fig. 213

Interpolating polynomial, $n = 10$

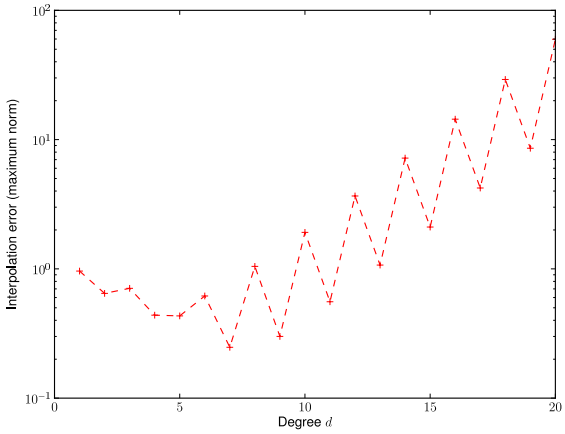


Fig. 214

Approximate $\left\| f - \mathrm{L}_{\mathcal{T}_n} f \right\|_\infty$ on $[-5,5]$

Note: approximation of $\|f - \mathsf{L}_{\mathcal{T}_n} f\|_\infty$ by *sampling* in 1000 equidistant points.

Observation: Strong oscillations of $\mathsf{I}_{\mathcal{T}} f$ near the endpoints of the interval, which seem to cause

$$\|f - \mathsf{L}_{\mathcal{T}} f\|_{L^\infty(]-5,5[)} \xrightarrow{n\to\infty} \infty \,.$$

Though polynomials possess great power to approximate functions, see Thm. 6.1.15 and Thm. 6.1.6, here polynomial interpolants fail completely. Approximation theorists even discovered the following "negative result":

Theorem 6.1.43. Divergent polynomial interpolants

Given a sequence of meshes of increasing size $\{\mathcal{T}_n\}_{n=1}^\infty$, $\mathcal{T}_j = \{t_0^{(n)}, \ldots, t_n^{(n)}\} \subset [a,b]$, $a \le t_0^{(n)} < t_2^{(j)} < \cdots < t_n^{(n)} \le b$, there exists a continuous function f such that the sequence of interpolating polynomials $(\mathsf{L}_{\mathcal{T}_n} f)_{n=1}^\infty$ does not converge to f uniformly as $n \to \infty$.

Now we aim to establish bounds for the supremum norm of the interpolation error of Lagrangian interpolation similar to the result of Thm. 6.1.15.

Theorem 6.1.44. Representation of interpolation error [?, Thm. 8.22], [?, Thm. 37.4]

We consider $f \in C^{n+1}(I)$ and the Lagrangian interpolation approximation scheme ($\to$ Def. 6.1.32) for a node set $\mathcal{T} := \{t_0, \ldots, t_n\} \subset I$. Then,

for every $t \in I$ there exists a $\tau_t \in]\min\{t, t_0, \ldots, t_n\}, \max\{t, t_0, \ldots, t_n\}[$ such that

$$f(t) - \mathsf{L}_{\mathcal{T}}(f)(t) = \frac{f^{(n+1)}(\tau_t)}{(n+1)!} \cdot \prod_{j=0}^{n} (t - t_j) \,. \tag{6.1.45}$$

Proof. Write $w_{\mathcal{T}}(t) := \prod_{j=0}^{n} (t - t_j) \in \mathcal{P}_{n+1}$ and *fix* $t \in I \setminus \mathcal{T}$.

$$t \neq t_j \;\Rightarrow\; w_{\mathcal{T}}(t) \neq 0 \;\Rightarrow\; \exists c = c(t) \in \mathbb{R}\colon\; f(t) - \mathsf{L}_{\mathcal{T}}(f)(t) = c(t) w_{\mathcal{T}}(t) \tag{6.1.46}$$

Consider the *auxiliary function* $\varphi(x) := f(x) - \mathsf{L}_{\mathcal{T}}(f)(x) - c w_{\mathcal{T}}(x)$ that has $n+2$ *distinct* zeros $t_0, \ldots, t_n, t$. By iterated application of the mean value theorem [**?**, Thm .5.2.1]

$$f \in C^1([a,b]),\; f(a) = f(b) = 0 \;\Rightarrow\; \exists \xi \in]a,b[\colon\; f'(\xi) = 0 \,,$$

to higher and higher derivatives, we conclude that

$$\varphi^{(m)} \text{ has } n + 2 - m \text{ distinct zeros in } I.$$

$$\overset{m:=n+1}{\Rightarrow} \quad \exists \tau_t \in I\colon\; \varphi^{(n+1)}(\tau_t) = f^{(n+1)}(\tau_t) - c(n+1)! = 0 \,.$$

This fixes the value of $c = \frac{f^{(n+1)}(\tau_t)}{(n+1)!}$ and by (6.1.46) this amounts to the assertion of the theorem. □

Remark 6.1.47 (Explicit representation of error of polynomial interpolation)

The previous theorem can be refined:

Lemma 6.1.48. Error representation for polynomial Lagrange interpolation

For $f \in C^{n+1}(I)$ let $\mathsf{I}_{\mathcal{T}} \in \mathcal{P}_n$ stand for the unique Lagrange interpolant ($\rightarrow$ Thm. 5.2.14) of f in the node set $\mathcal{T} := \{t_0, \ldots, t_n\} \subset I$. Then for all $t \in I$ the interpolation error is

$$f(t) - \mathsf{I}_{\mathcal{T}}(f)(t) = \int_0^1 \int_0^{\tau_1} \cdots \int_0^{\tau_{n-1}} \int_0^{\tau_n} f^{(n+1)}(t_0 + \tau_1(t_1 - t_0) + \cdots + \tau_n(t_n - t_{n-1}) + \tau(t - t_n))\, \mathrm{d}\tau \mathrm{d}\tau_n \cdots \mathrm{d}\tau_1 \cdot \prod_{j=0}^{n}(t - t_j) \,.$$

Proof. By induction on n, use (5.2.34) and the fundamental theorem of calculus [**?**, Sect. 3.1]: □

Remark 6.1.49 (Error representation for generalized Lagrangian interpolation)

A result analogous top Lemma 6.1.48 holds also for general polynomial interpolation with multiple nodes as defined in (5.2.22).

Lemma 6.1.48 provides an *exact* formula (6.1.45) for the interpolation error. From it we can derive *estimates* for the supremum norm of the interpolation error on the interval I as follows:

❶ first bound the right hand side via $f^{(n+1)}(\tau_t) \leq \left\| f^{(n+1)} \right\|_{L^\infty(I)}$,

❷ then increase the right hand side further by switching to the maximum (in modulus) w.r.t. t (the resulting bound does no longer depend on t!),

❸ and, finally, take the maximum w.r.t. t on the left of $\leq$.

This yields the following interpolation error estimate for degree-n Lagrange interpolation on the node set $\{t_0, \ldots, t_n\}$:

$$\text{Thm. 6.1.44} \quad \Rightarrow \quad \boxed{\|f - \mathsf{I}_{\mathcal{T}} f\|_{L^\infty(I)} \leq \frac{\|f^{(n+1)}\|_{L^\infty(I)}}{(n+1)!} \max_{t \in I} |(t - t_0) \cdot \cdots \cdot (t - t_n)|} \,. \tag{6.1.50}$$

Remark 6.1.51 (Significance of smoothness of interpoland)

The estimate (6.1.50) hinges on bounds for (higher) derivatives of the interpoland f, which, essentially, should belong to $C^{n+1}(I)$. The same can be said about the estimate of Thm. 6.1.15.

This reflects a general truth about estimates of norms of the interpolation error:

Quantitative interpolation error estimates rely on smoothness!

Example 6.1.52 (Error of polynomial interpolation Exp. 6.1.35 cnt'd)

Now we are in a position to give a theoretical explanation for exponential convergence observed for polynomial interpolation of $f(t) = \sin(t)$ on equidistant nodes: by Lemma 6.1.48 and (6.1.50)

$$\left\| f^{(k)} \right\|_{L^\infty(I)} \leq 1 \,, \quad \forall k \in \mathbb{N}_0 \quad \Rightarrow \quad \begin{aligned} \|f - p\|_{L^\infty(I)} &\leq \frac{1}{(1+n)!} \max_{t \in I} \left| (t-0)(t-\tfrac{\pi}{n})(t-\tfrac{2\pi}{n}) \cdot \cdots \cdot (t-\pi) \right| \\ &\leq \frac{1}{n+1} \left(\frac{\pi}{n} \right)^{n+1} . \end{aligned}$$

➙ Uniform asymptotic (even more than) exponential convergence of the interpolation polynomials (independently of the set of nodes $\mathcal{T}$. In fact, $\|f - p\|_{L^\infty(I)}$ decays even faster than exponential!)

Example 6.1.53 (Runge's example Ex. 6.1.41 cnt'd)

How can the blow-up of the interpolation error observed in Ex. 6.1.41 be reconciled with Lemma 6.1.48 ?

Here $f(t) = \frac{1}{1+t^2}$ allows only to conclude $|f^{(n)}(t)| = 2^n n! \cdot O(|t|^{-2-n})$ for $n \to \infty$.

➙ Possible blow-up of error bound from Thm. 6.1.44 $\to \infty$ for $n \to \infty$.

Remark 6.1.54 (L^2-error estimates for polynomial interpolation)

Thm. 6.1.44 gives error estimates for the L^∞-norm. What about other norms?

From Lemma 6.1.48 using the Cauchy-Schwarz inequality

$$\left| \int_a^b f(t) g(t) \, \mathrm{d}t \right|^2 \leq \int_a^b |f(t)|^2 \, \mathrm{d}t \int_a^b |g(t)|^2 \, \mathrm{d}t \,, \quad \forall f, g \in C^0([a,b]) \,, \tag{6.1.55}$$

repeatedly, we can estimate

$$\begin{aligned} \|f - \mathsf{L}_{\mathcal{T}}(f)\|^2_{L^2(I)} &= \int_I \left| \int_0^1 \int_0^{\tau_1} \cdots \int_0^{\tau_{n-1}} \int_0^{\tau_n} f^{(n+1)}(\ldots) \, \mathrm{d}\tau \mathrm{d}\tau_n \cdots \mathrm{d}\tau_1 \cdot \underbrace{\prod_{j=0}^n (t - t_j)}_{|t-t_j| \leq |I|} \right|^2 \mathrm{d}t \\ &\leq \int_I |I|^{2n+2} \underbrace{\mathrm{vol}_{(n+1)}(S_{n+1})}_{=1/(n+1)!} \int_{S_{n+1}} |f^{(n+1)}(\ldots)|^2 \, \mathrm{d}\boldsymbol{\tau} \, \mathrm{d}t \\ &= \int_I \frac{|I|^{2n+2}}{(n+1)!} \int_I \underbrace{\mathrm{vol}_{(n)}(C_{t,\tau})}_{\leq 2^{(n-1)/2}/n!} |f^{(n+1)}(\tau)|^2 \, \mathrm{d}\tau \mathrm{d}t \,, \end{aligned}$$

where

$$S_{n+1} := \{ \mathbf{x} \in \mathbb{R}^{n+1} \colon 0 \leq x_n \leq x_{n-1} \leq \cdots \leq x_1 \leq 1 \} \quad \text{(unit simplex)} \,,$$

$$C_{t,\tau} := \{\mathbf{x} \in S_{n+1}\colon t_0 + x_1(t_1 - t_0) + \cdots + x_n(t_n - t_{n-1}) + x_{n+1}(t - t_n) = \tau\} \,.$$

This gives the bound for the L^2-norm of the error:

$$\Rightarrow \qquad \|f - \mathsf{L}_{\mathcal{T}}(f)\|_{L^2(I)} \le \frac{2^{(n-1)/4}|I|^{n+1}}{\sqrt{(n+1)!n!}} \left(\int_I |f^{(n+1)}(\tau)|^2 \,\mathrm{d}\tau\right)^{1/2} . \tag{6.1.56}$$

Notice: $f \mapsto \left\|f^{(n)}\right\|_{L^2(I)}$ defines a seminorm on $C^{n+1}(I)$
(Sobolev-seminorm, measure of the smoothness of a function).

Estimates like (6.1.56) play a key role in the analysis of numerical methods for solving partial differential equations ($\rightarrow$ course "Numerical methods for partial differential equations").

Remark 6.1.57 (Interpolation error estimates and the Lebesgue constant)

The sensitivity of a polynomial interpolation scheme $\mathsf{I}_{\mathcal{T}} : \mathbb{K}^{n+1} \to C^0(I)$, $\mathcal{T} \subset I$ a node set, as introduced in Section 5.2.4 and expressed by the Lebesgue constant ($\rightarrow$ Lemma 5.2.71)

$$\lambda_{\mathcal{T}} := \|\mathsf{I}_{\mathcal{T}}\|_{\infty\to\infty} := \sup_{\mathbf{y}\in\mathbb{R}^{n+1}\setminus\{0\}} \frac{\|\mathsf{I}_{\mathcal{T}}(\mathbf{y})\|_{L^\infty(I)}}{\|\mathbf{y}\|_\infty} \,,$$

establishes an important connection between the norms of the interpolation error and of the best approximation error.

We first observe that the polynomial approximation scheme $\mathsf{L}_{\mathcal{T}}$ induced by $\mathsf{I}_{\mathcal{T}}$ *preserves polynomials* of degree $\le n$:

$$\mathsf{L}_{\mathcal{T}} p = \mathsf{I}_{\mathcal{T}}\big([p(t)]_{t\in\mathcal{T}}\big) = p \quad \forall p \in \mathcal{P}_n \,. \tag{6.1.58}$$

Thus, by the triangle inequality, for a generic norm on $C^0(I)$ and $\|\mathsf{L}_{\mathcal{T}}\|$ designating the associated operator norm of the linear mapping $\mathsf{L}_{\mathcal{T}}$, *cf.* (5.2.70),

$$\|f - \mathsf{L}_{\mathcal{T}} f\| \overset{(6.1.58)}{=} \|(f-p) - \mathsf{L}_{\mathcal{T}}(f-p)\| \le (1 + \|\mathsf{L}_{\mathcal{T}}\|)\|f - p\| \quad \forall p \in \mathcal{P}_n \,,$$

$$\blacktriangleright \qquad \|f - \mathsf{L}_{\mathcal{T}} f\| \le (1 + \|\mathsf{L}_{\mathcal{T}}\|) \underbrace{\inf_{p\in\mathcal{P}_n} \|f - p\|}_{\text{best approximation error}} . \tag{6.1.59}$$

Note that for $\|\cdot\| = \|\cdot\|_{L^\infty(I)}$, since $\left\|[f(t)]_{t\in\mathcal{T}}\right\|_\infty \le \|f\|_{L^\infty(I)}$, we can estimate the operator norm, *cf.* (5.2.70),

$$\|\mathsf{L}_{\mathcal{T}}\|_{L^\infty(I)\to L^\infty(I)} \le \|\mathsf{I}_{\mathcal{T}}\|_{\mathbb{R}^{n+1}\to L^\infty(I)} = \lambda_{\mathcal{T}} \,, \tag{6.1.60}$$

$$\blacktriangleright \qquad \boxed{\|f - \mathsf{L}_{\mathcal{T}} f\|_{L^\infty(I)} \le (1 + \lambda_{\mathcal{T}}) \inf_{p\in\mathcal{P}_n} \|f - p\|_{L^\infty(I)} \quad \forall f \in C^0(I)} \,. \tag{6.1.61}$$

Hence, if a bound for $\lambda_{\mathcal{T}}$ is available, the best approximation error estimate of Thm. 6.1.15 immediately yields interpolation error estimates.

(6.1.62) Interpolation error estimates for analytic interpolands

Exponential convergence can often be observed for families of Lagrangian approximation schemes, when they are applied to an analytic interpoland.

Definition 6.1.63. Analyticity of a complex valued function

let $D \subset \mathbb{C}$ be an *open* set in the complex plane. A function $f : D \to \mathbb{C}$ is called analytic/holomorphic in D, if $f \in C^\infty(D)$ and it possesses a convergent Taylor series at every point $z \in D$.

The mathematical area of complex analysis ($\to$ course in the BSc program CSE) studies analytic functions. Analyticity gives access to powerful tools provided by complex analysis. One of theses tools is the residue theorem.

Theorem 6.1.64. Residue theorem [?, Ch. 13]

Let $D \subset \mathbb{C}$ be an open set, $G \subset D$ a closed set contained in D, $\gamma := \partial G$ its (piecewise smooth and oriented) boundary, and Π a finite *set contained in the interior of G.*

Then for each function f that is analytic in $D \setminus \Pi$ holds

$$\frac{1}{2\pi\imath}\int_\gamma f(z)\,dz = \sum_{p\in\Pi} \operatorname{res}_p f \ ,$$

where $\operatorname{res}_p f$ is the residual of f in $p \in \mathbb{C}$.

- Note that the integral $\int_\gamma$ in Thm. 6.1.64 is a path integral in the complex plane ("contour integral"): If the path of integration γ is described by a parameterization $\tau \in J \mapsto \gamma(\tau) \in \mathbb{C}$, $J \subset \mathbb{R}$, then

$$\int_\gamma f(z)\,\mathrm{d}z := \int_J f(\gamma(\tau))\cdot\dot\gamma(\tau)\,\mathrm{d}\tau \ , \tag{6.1.65}$$

  where $\dot\gamma$ designates the derivative of γ with respect to the parameter, and $\cdot$ indicates multiplication in $\mathbb{C}$. For contour integrals we have the estimate

$$\left|\int_\gamma f(z)\,\mathrm{d}z\right| \le |\gamma| \max_{z\in\gamma} |f(z)| \ . \tag{6.1.66}$$

- Π often stands for the set of poles of f, that is, points where "f attains the value ∞".

The residue theorem is very useful, because there are simple formulas for $\operatorname{res}_p f$:

Lemma 6.1.67. Residual formula for quotients

let g and h be complex valued functions that are both analytic in a neighborhood of $p \in \mathbb{C}$, and satisfy $h(p) = 0$, $h'(p) \neq 0$. Then

$$\operatorname{res}_p \frac{g}{h} = \frac{g(p)}{h'(p)} .$$

Now we consider a polynomial Lagrangian approximation scheme on the interval $I := [a,b] \subset \mathbb{R}$, based on the node set $\mathcal{T} := \{t_0, \ldots, t_n\} \subset I$.

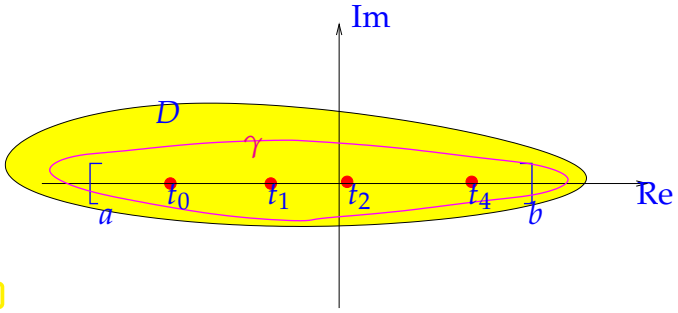


Fig. 215

Assumption 6.1.68. Analyticity of interpoland

We assume that the interpoland $f : I \to \mathbb{C}$ *can be extended* to a function $f : D \subset \mathbb{C} \to \mathbb{C}$, which is analytic ($\to$ Def. 6.1.63) on the open set $D \subset \mathbb{C}$ with $[a,b] \subset D$.

Key is the following representation of the Lagrange polynomials (5.2.11) for node set $\mathcal{T} = \{t_0, \ldots, t_n\}$:

$$L_j(t) = \prod_{k=0, k\neq j}^{n} \frac{t - t_k}{t_j - t_k} = \frac{w(t)}{(t - t_j) \prod_{k=0, k\neq j}^{n} (t_j - t_k)} = \frac{w(t)}{(t - t_j) w'(t_j)} , \tag{6.1.69}$$

$$\text{where} \quad w(t) = (t - t_0) \cdot \cdots \cdot (t - t_n) \in \mathcal{P}_{n+1} .$$

Consider the following parameter dependent function g_t , whose set of poles in D is $\Pi = \{t, t_0, \ldots, t_n\}$

$$g_t(z) := \frac{f(z)}{(z - t) w(z)} , \quad z \in \mathbb{C} \setminus \Pi , \quad t \in [a,b] \setminus \{t_0, \ldots, t_n\} .$$

➢ g_t is analytic on $D \setminus \{t, t_0, \ldots, t_n\}$ (t must be regarded as parameter!)

Apply residue theorem Thm. 6.1.64 to g_t and a closed path of integration $\gamma \subset D$ winding once around $[a,b]$, such that its interior is simply connected, see the magenta curve in Fig. 215:

$$\frac{1}{2\pi\imath} \int_\gamma g_t(z)\,\mathrm{d}z = \operatorname{res}_t g_t + \sum_{j=0}^{n} \operatorname{res}_{t_j} g_t \overset{\text{Lemma 6.1.67}}{=} \frac{f(t)}{w(t)} + \sum_{j=0}^{n} \frac{f(t_j)}{(t_j - t) w'(t_j)}$$

Possible, because all zeros of w are single zeros!

$$f(t) = \underbrace{-\sum_{j=1}^{n} f(t_j) \underbrace{\frac{w(t)}{(t_j - t) w'(t_j)}}_{-\text{Lagrange polynomial !}}}_{\text{polynomial interpolant !}} + \underbrace{\frac{w(t)}{2\pi\imath} \int_\gamma g_t(z)\,\mathrm{d}z}_{\text{interpolation error !}} . \tag{6.1.70}$$

This is another representation formula for the interpolation error, an alternative to that of Thm. 6.1.44 and Lemma 6.1.48. We conclude that for all $t \in [a,b]$

$$|f(t) - \mathsf{L}_{\mathcal{T}} f(t)| \leq \left| \frac{w(t)}{2\pi\imath} \int_\gamma \frac{f(z)}{(z-t)w(z)} \,\mathrm{d}z \right| \leq \frac{|\gamma|}{2\pi} \frac{\max\limits_{a\leq\tau\leq b} |w(\tau)|}{\min\limits_{z\in\gamma} |w(z)|} \cdot \frac{\max\limits_{z\in\gamma} |f(z)|}{\operatorname{dist}([a,b],\gamma)} \,. \tag{6.1.71}$$

In concrete setting, in order to exploit the estimate (6.1.71) to study the n-dependence of the supremum norm of the interpolation error, we need to know

- an upper bound for $|w(t)|$ for $a \leq t \leq b$,
- an a lower bound for $|w(z)|$, $z \in \gamma$, for a suitable path of integration $\gamma \subset D$,
- a lower bound for the distance of the path γ and the interval $[a,b]$ in the complex plane.

Remark 6.1.72 (Determining the domain of analyticity)

The subset of $\mathbb{C}$, where a function f given by a formula, is analytic can often be determined without computing derivatives using the following consequence of the chain rule:

Theorem 6.1.73. Composition of analytic functions

If $f : D \subset \mathbb{C} \to \mathbb{C}$ and $g : U \subset \mathbb{C} \to \mathbb{C}$ are analytic in the open sets D and U, respectively, then their composition $f \circ g$ is analytic in

$$\{z \in U : g(z) \in D\} \,.$$

This can be combined with the following facts:

- Polynomials, $\exp(z)$, $\sin(z)$, $\cos(z)$, $\sinh(z)$, $\cosh(z)$ are analytic on $\mathbb{C}$ (entire functions).
- Rational functions (quotients of polynomials) are analytic everywhere except in the zeros of their denominator.
- The square root $z \to \sqrt{z}$ is analytic in $\mathbb{C}\backslash]-\infty, 0]$.

For example, according to these rules the function $f(t) = (1+t^2)^{-1}$ can be extended to an analytic function on $\mathbb{C} \setminus \{-\imath, \imath\}$. If $\mathbf{A} \in \mathbb{C}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$, $n \in \mathbb{N}$, then $z \mapsto (\mathbf{A} + z\mathbf{I})^{-1}\mathbf{b} \in \mathbb{C}^n$ is (componentwise) analytic in $\mathbb{C} \setminus \sigma(\mathbf{A})$, where $\sigma(\mathbf{A})$ denotes the set of eigenvalues of $\mathbf{A}$ (the spectrum).

6.1.3 Chebychev Interpolation

As pointed out in § 6.0.6, when we build approximation schemes from interpolation schemes, we have the extra freedom to choose the sampling points (= interpolation nodes). Now, based on the insight into the structure of the interpolation error gained from Thm. 6.1.44, we seek to choose "optimal" sampling points. They will give rise to the so-called Chebychev polynomial approximation schemes, also known as Chebychev interpolation.

6.1.3.1 Motivation and definition

Setting:
- ✦ Without loss of generality (→ Rem. 6.1.18): $I = [-1,1]$,
- ✦ interpoland $f : I \to \mathbb{R}$ at least continuous, $f \in C^0(I)$,
- ✦ set of interpolation nodes $\mathcal{T} := \{-1 \le t_0 < t_1 < \cdots < t_{n-1} < t_n \le 1\}$, $n \in \mathbb{N}$.

Recall Thm. 6.1.44: $$\|f - \mathsf{L}_{\mathcal{T}} f\|_{L^\infty(I)} \le \frac{1}{(n+1)!} \left\| f^{(n+1)} \right\|_{L^\infty(I)} \|w\|_{L^\infty(I)} ,$$

with nodal polynomial $w(t) := (t - t_0) \cdot \cdots \cdot (t - t_n)$.

Optimal choice of interpolation nodes independent of interpoland

Idea: choose nodes $t_0, \ldots, t_n$ such that $\|w\|_{L^\infty(I)}$ is minimal!

This is equivalent to finding a polynomial $q \in \mathcal{P}_{n+1}$
- ✦ with leading coefficient $= 1$,
- ✦ such that it minimizes the norm $\|q\|_{L^\infty(I)}$.

➤ Then choose nodes $t_0, \ldots, t_n$ as zeros of q (caution: t_j must belong to I).

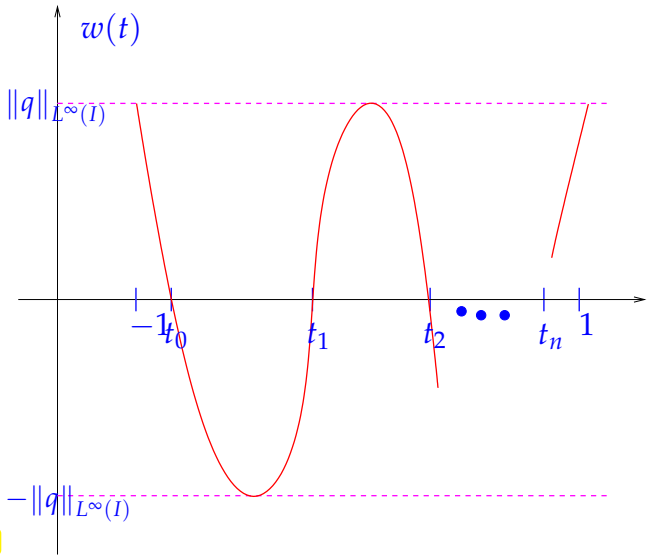


Fig. 216

Requirements on q (by heuristic reasoning)

Remark 6.1.75 (A priori and a posteriori choice of optimal interpolation nodes)

We stress that we aim for an "optimal" *a priori* choice of interpolation nodes, a choice that is made **before** any information about the interpoland becomes available.

Of couse, an *a posteriori* choice based on information gleaned from evaluations of the interpoland f may yield much better interpolants (in the sense of smaller norm of the interpolation error). Many modern algorithms employ this a posteriori adaptive approximation policy, but this chapter will not cover them.

However, see Section 7.5 for the discussion of an a posteriori adaptive approach for the numerical approximation of definite integrals.

Optimal polynomials q will exist, but they seem to be elusive. First, we develop some insights in how they must "look like".
- If t^* is an extremal point of q ➙ $|q(t^*)| = \|q\|_{L^\infty(I)}$,
- q has $n+1$ zeros in I $(*)$,
- $|q(-1)| = |q(1)| = \|q\|_{L^\infty(I)}$.

➢ q has $n+2$ extrema in $[-1,1]$

$(*)$ is motivated by an indirect argument:

If $q(t) = (t-t_0)\cdot\cdot\cdot\cdot\cdot(t-t_{n+1})$ with $t_0 < -1$, then

$$|p(t)| := |(t+1)\cdot\cdot\cdot\cdot\cdot(t-t_{n+1})| < |q(t)| \quad \forall t \in I \quad \text{(why ?)},$$

which contradicts the minimality property of q. Same argument for $t_0 > 1$. The reasonings leading to the above heuristic demands will be elaborated in the proof of Thm. 6.1.82.

Are there polynomials satisfying these requirements? If so, do they allow a simple characterization?

Definition 6.1.76. Chebychev polynomials $\rightarrow$ **[?, Ch. 32]**

The n^{th} Chebychev polynomial is $T_n(t) := \cos(n \arccos t), \quad -1 \le t \le 1, n \in \mathbb{N}.$

The next result confirms that the T_n are polynomials, indeed.

Theorem 6.1.77. 3-term recursion for Chebychev polynomials $\rightarrow$ **[?, (32.2)]**

The function T_n defined in Def. 6.1.76 satisfy the 3-term recursion

$$T_{n+1}(t) = 2t\,T_n(t) - T_{n-1}(t) \quad , \quad T_0 \equiv 1\,, \quad T_1(t) = t\,, \quad n \in \mathbb{N}\,. \tag{6.1.78}$$

Proof. Just use the trigonometric identity $\cos(n+1)x = 2\cos nx \cos x - \cos(n-1)x$ with $\cos x = t$. □

The theorem implies:
- $T_n \in \mathcal{P}_n$,
- their leading coefficients are equal to 2^{n-1},
- the T_n are linearly independent,
- $\{T_j\}_{j=0}^n$ is a basis of $\mathcal{P}_n = \mathrm{Span}\{T_0,\ldots,T_n\}$, $n \in \mathbb{N}_0$.

See Code 6.1.79 for algorithmic use of the 3-term recursion (6.1.78).

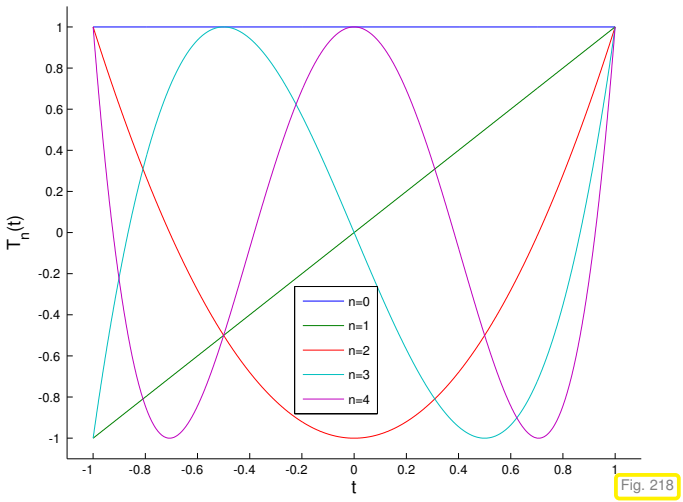


Fig. 217

Chebychev polynomials $T_0,\ldots,T_4$

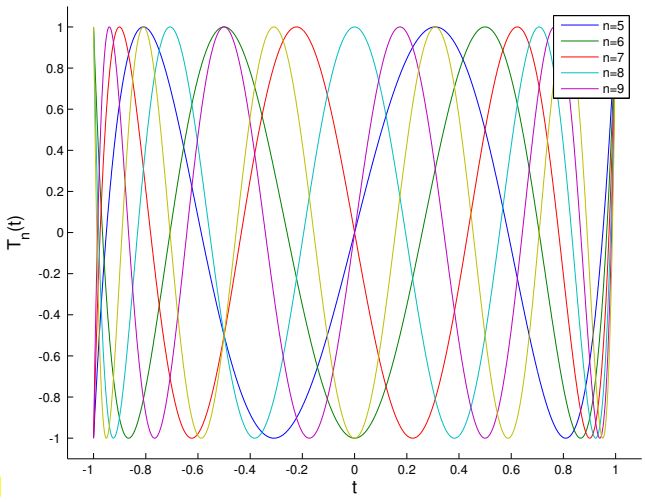


Fig. 218

Chebychev polynomials $T_5,\ldots,T_9$

C++11 code 6.1.79: Efficient evaluation of Chebychev polynomials up to a certain degree

```
2 // Computes the values of the Chebychev polynomials T_0,...,T_d, d >= 2
3 // at points passed in x using the 3-term recursion (6.1.78).
4 // The values T_k(x_j), are returned in row k+1 of V.
5 void chebpolmult(const int d, const RowVectorXd& x, MatrixXd& V) {
```

```
  const unsigned n = x.size();
  V = MatrixXd::Ones(d + 1, n); // T_0 ≡ 1
  V.block(1, 0, 1, n) = x; // T_1(x) = x
  for (int k = 1; k < d; ++k) {
    const RowVectorXd p = V.block(k, 0, 1, n); // p = T_k
    const RowVectorXd q = V.block(k - 1, 0, 1, n); // q = T_{k-1}
    V.block(k + 1, 0, 1, n) = 2*x.cwiseProduct(p) - q; // 3-term recursion
  }
}
```

C++11 code 6.1.80: Plotting Chebychev polynomials, see Fig. 217, 218

```
// plots Chebychev polynomials up to degree nmax on [-1,1]
void chebpolplot(const unsigned nmax) {
  Eigen::RowVectorXd x = Eigen::RowVectorXd::LinSpaced(500, -1, 1); // evaluation points
  Eigen::MatrixXd V; chebpolmult(nmax, x, V); // get values of cheb. polynomials

  mgl::Figure fig;
  // iterate over rows of V, which contain the values of the cheb. polynomials
  for (unsigned r = 0; r < nmax + 1; ++r)
    fig.plot(x, V.row(r)).label("n = " + std::to_string(r));

  fig.title("First " + std::to_string(nmax) + " cheb. polynomials");
  fig.xlabel("x"); fig.ylabel("T_n(x)");
  fig.ranges(-1.1, 1.1, -1.1, 1.1);
  fig.legend(); fig.save("chebpols.eps");
}
```

From Def. 6.1.76 we conclude that T_n attains the values ± 1 in its extrema with alternating signs, thus matching our heuristic demands:

$$|T_n(\bar{t}_k)| = 1 \Leftrightarrow \exists\, k = 0,\dots,n\colon \quad \bar{t}_k = \cos\frac{k\pi}{n}\,, \qquad \|T_n\|_{L^\infty([-1,1])} = 1\,. \tag{6.1.81}$$

What is still open is the validity of the heuristics guiding the choice of the optimal nodes. The next fundamental theorem will demonstrate that, after scaling, the T_n really supply polynomials on $[-1,1]$ with fixed leading coefficient and minimal supremum norm.

Theorem 6.1.82. Minimax property of the Chebychev polynomials [?, Section 7.1.4.], [?, Thm. 32.2]

The polynomials T_n from Def. 6.1.76 minimize the supremum norm in the following sense:

$$\|T_n\|_{L^\infty([-1,1])} = \inf\{\|p\|_{L^\infty([-1,1])}\colon p \in \mathcal{P}_n, p(t) = 2^{n-1}t^n + \cdots\}\,, \quad \forall n \in \mathbb{N}\,.$$

Proof. (indirect) Assume

$$\exists q \in \mathcal{P}_n\,, \text{ leading coefficient } = 2^{n-1}\colon \quad \|q\|_{L^\infty([-1,1])} < \|T_n\|_{L^\infty([-1,1])}\,. \tag{6.1.83}$$

$(T_n - q)(x) > 0$ in local maxima of T_n
$(T_n - q)(x) < 0$ in local minima of T_n

From knowledge of local extrema of T_n, see (6.1.81):

$T_n - q$ changes sign at least $n+1$ times

$\Rightarrow$ $T_n - q$ has at least n zeros

$T_n - q \equiv 0$, because $T_n - q \in \mathcal{P}_{n-1}$ (same leading coefficient!)

This cannot be reconciled with the properties (6.1.83) of q and, thus, leads to a contradiction. □

The zeros of T_n are $$t_k = \cos\left(\frac{2k+1}{2n}\pi\right), \quad k = 0,\ldots,n-1 \,. \tag{6.1.84}$$

Too see this, notice

$$T_n(t) = 0 \overset{\text{zeros of } \cos}{\Leftrightarrow} n\arccos t \in (2\mathbb{Z}+1)\frac{\pi}{2}$$
$$\overset{\arccos \in [0,\pi]}{\Leftrightarrow} t \in \left\{\cos\left(\frac{2k+1}{n}\frac{\pi}{2}\right),\ k = 0,\ldots,n-1\right\}.$$

Thus, we have identified the t_k from (6.1.84) as optimal interpolation nodes for a Lagrangian approximation scheme. The t_k are known as Chebychev nodes. Their distribution in $[-1,1]$ is plotted in Fig. 219, a geometric construction is indicated in Fig. 220.

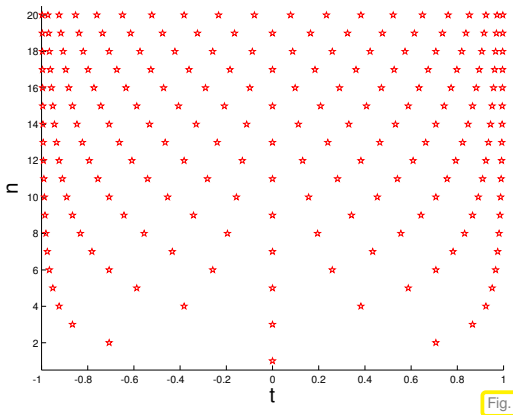


Fig. 219

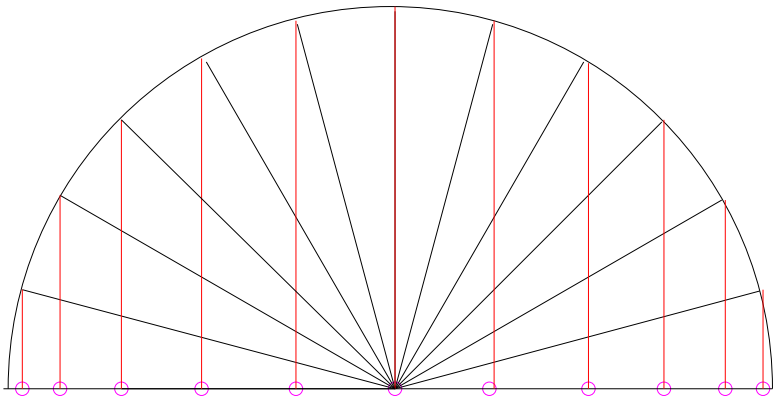


Fig. 220

When we use Chebychev nodes for polynomial interpolation we call the resulting Lagrangian approximation scheme Chebychev interpolation. On the interval $[-1,1]$ is is characterized by:

- "optimal" interpolation nodes $\mathcal{T} = \left\{\cos\left(\frac{2k+1}{2(n+1)}\pi\right), k = 0,\ldots,n\right\}$,
- $w(t) = (t-t_0)\cdots(t-t_n) = 2^{-n}T_{n+1}(t)$, $\|w\|_{L^\infty(I)} = 2^{-n}$, with leading coefficient 1.

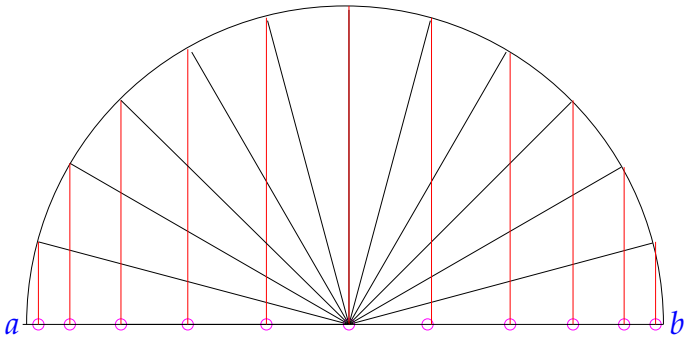
Then, by Thm. 6.1.44, we immediately get an interpolation error estimate for Chebychev interpolation of $f \in C^{n+1}([-1,1])$:

$$\|f - I_{\mathcal{T}}(f)\|_{L^\infty([-1,1])} \le \frac{2^{-n}}{(n+1)!}\left\|f^{(n+1)}\right\|_{L^\infty([-1,1])} \,. \tag{6.1.85}$$

Remark 6.1.86 (Chebychev polynomials on arbitrary interval)

Following the recipe of Rem. 6.1.18 Chebychev interpolation on an arbitrary interval $[a,b]$ can immediately be defined. The same polynomial Lagrangian approximation scheme is obtained by transforming the Chebychev nodes (6.1.84) from $[-1,1]$ to $[a,b]$ using the unique affine transformation (6.1.19):

$$\widehat{t} \in [-1,1] \mapsto t := a + \tfrac{1}{2}(\widehat{t}+1)(b-a) \in [a,b]\ ].$$



The Chebychev nodes in the interval $I = [a,b]$ are

$$t_k := a + \tfrac{1}{2}(b-a)\left(\cos\left(\frac{2k+1}{2(n+1)}\,\pi\right)+1\right), \quad k = 0,\ldots,n\,. \tag{6.1.87}$$

$$p \in \mathcal{P}_n \quad \wedge \quad p(t_j) = f(t_j) \quad \Leftrightarrow \quad \widehat{p} \in \mathcal{P}_n \quad \wedge \quad \widehat{p}(\widehat{t}_j) = \widehat{f}(\widehat{t}_j)\,.$$

With transformation formula for the integrals & $\dfrac{\mathrm{d}^n \widehat{f}}{\mathrm{d}\widehat{t}^n}(\widehat{t}) = (\frac{1}{2}|I|)^n \dfrac{\mathrm{d}^n f}{\mathrm{d}t^n}(t)$:

$$\begin{aligned}\|f - \mathrm{I}_{\mathcal{T}}(f)\|_{L^\infty(I)} &= \left\|\widehat{f} - \mathrm{I}_{\widehat{\mathcal{T}}}(\widehat{f})\right\|_{L^\infty([-1,1])} \le \frac{2^{-n}}{(n+1)!}\left\|\frac{\mathrm{d}^{n+1}\widehat{f}}{\mathrm{d}\widehat{t}^{n+1}}\right\|_{L^\infty([-1,1])} \\ &\le \frac{2^{-2n-1}}{(n+1)!}|I|^{n+1}\left\|f^{(n+1)}\right\|_{L^\infty(I)}\,. \end{aligned} \tag{6.1.88}$$

6.1.3.2 Chebychev interpolation error estimates

Example 6.1.89 (Polynomial interpolation: Chebychev nodes versus equidistant nodes)

We consider Runge's function $f(t) = \frac{1}{1+t^2}$, see Ex. 6.1.41, and compare polynomial interpolation based on uniformly spaced nodes and Chebychev nodes in terms of behavior of interpolants.

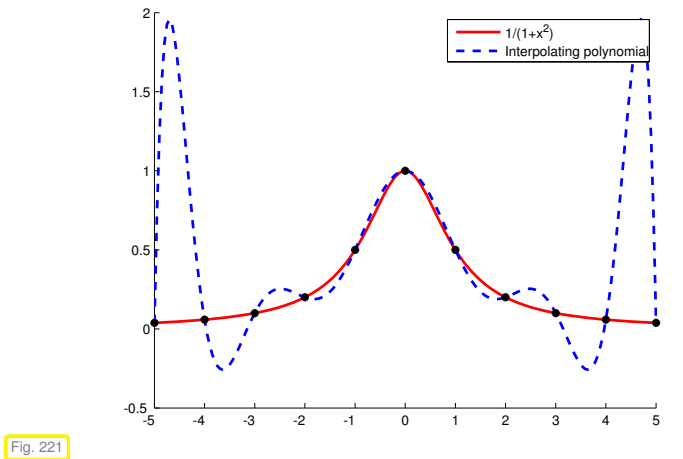


Fig. 221

Equidistant nodes

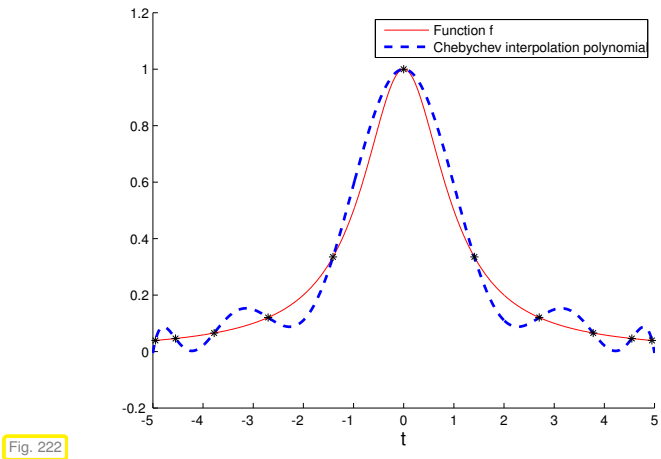


Fig. 222

Chebychev nodes

We observe that the Chebychev nodes cluster at the endpoints of the interval, which successfully suppresses the huge oscillations haunting equidistant interpolation there.

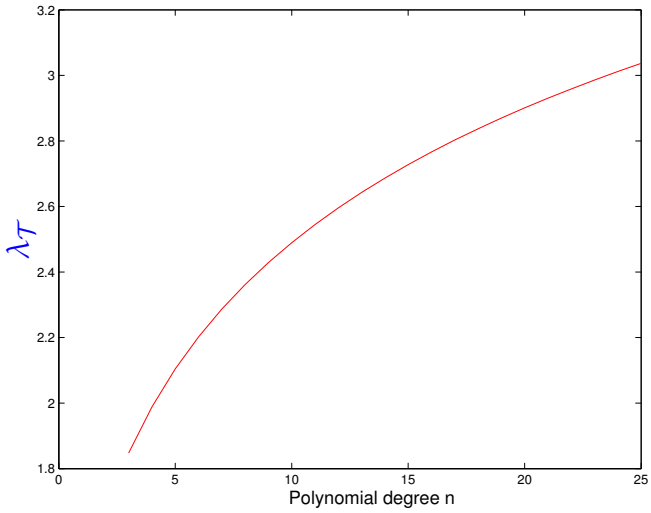
Remark 6.1.90 (Lebesgue Constant for Chebychev nodes → Section 5.2.4)

We saw in Rem. 5.2.74 that the Lebesgue constant $\lambda_{\mathcal{T}}$ that measures the sensitivity of a polynomial interpolation scheme, blows up exponentially with increasing number of *equispaced* interpolation nodes. In stark contrast $\lambda_{\mathcal{T}}$ grows only logarithmically in the number of Chebychev nodes.

More precisely, sophisticated theory [**?**, **?**, **?**] supplies the bound

$$\lambda_{\mathcal{T}} \leq \frac{2}{\pi}\log(1+n)+1 \quad . \tag{6.1.91}$$

Measured Lebesgue constant for Chebychev nodes based on approximate evaluation of (5.2.72) by sampling. ▷



Combining (6.1.91) with the general estimate from Rem. 6.1.57

$$\|f - \mathsf{L}_{\mathcal{T}} f\|_{L^\infty(I)} \leq (1+\lambda_{\mathcal{T}}) \inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty(I)} \quad \forall f \in C^0(I)\,, \tag{6.1.61}$$

and the bound for the best approximation error for polynomaial from Thm. 6.1.15

$$\inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([-1,1])} \leq (1+\pi^2/2)^r \frac{(n-r)!}{n!} \left\| f^{(r)} \right\|_{L^\infty([-1,1])}\,,$$

we end up with a bound for the supremum norm of the interpolation error in the case of Chebychev interpolation on $[-1,1]$

$$\|f - \mathsf{L}_{\mathcal{T}} f\|_{L^\infty([-1,1])} \leq (2/\pi\log(1+n)+2)(1+\pi^2/2)^r \frac{(n-r)!}{n!} \left\| f^{(r)} \right\|_{L^\infty([-1,1])}\,. \tag{6.1.92}$$

Experiment 6.1.93 (Chebychev interpolation errors)

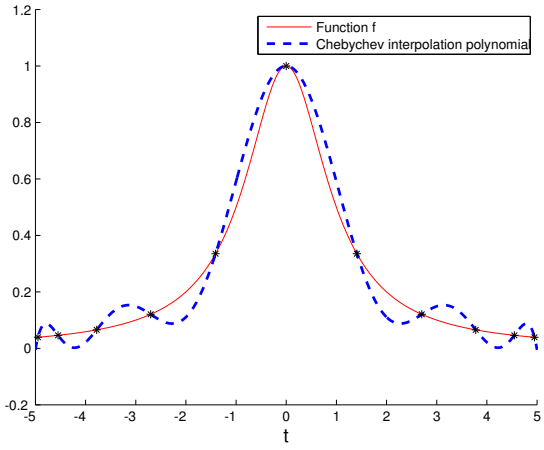
Now we empirically investigate the behavior of norms of the interpolation error for Chebychev interpolation and functions with different (smoothness) properties as we increase the number of interpolation nodes.

In the experiments, for $I=[a,b]$ we set $x_l := a + \frac{b-a}{N} l$, $l = 0,\ldots,N$, $N = 1000$, and we approximate the norms of the interpolation error as follows ($p \mathrel{\hat{=}}$ interpolating polynomial):

$$\|f-p\|_\infty \approx \max_{0\leq l\leq N} |f(x_l) - p(x_l)| \tag{6.1.94}$$

$$\|f-p\|_2^2 \approx \frac{b-a}{2N} \sum_{0\leq l<N} \left(|f(x_l)-p(x_l)|^2 + |f(x_{l+1})-p(x_{l+1})|^2 \right) \tag{6.1.95}$$

① $f(t) = (1+t^2)^{-1}$, $I = [-5,5]$ (see Ex. 6.1.41): analytic in a neighborhood of I.
Interpolation with $n = 10$ Chebychev nodes (plot on the left).



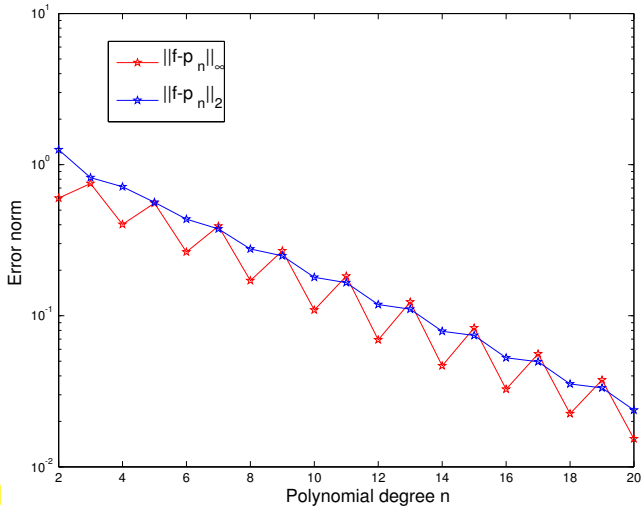


Fig. 223

Notice: exponential convergence ($\to$ Def. 6.1.38) of the Chebychev interpolation:

$$p_n \to f, \quad \|f - \mathsf{I}_n f\|_{L^\infty([-5,5])} \approx 0.8^n$$

.

② $f(t) = \max\{1-|t|, 0\}$, $I = [-2,2]$, $n = 10$ nodes (plot on the left).

Now $f \in C^0(I)$ but $f \notin C^1(I)$.

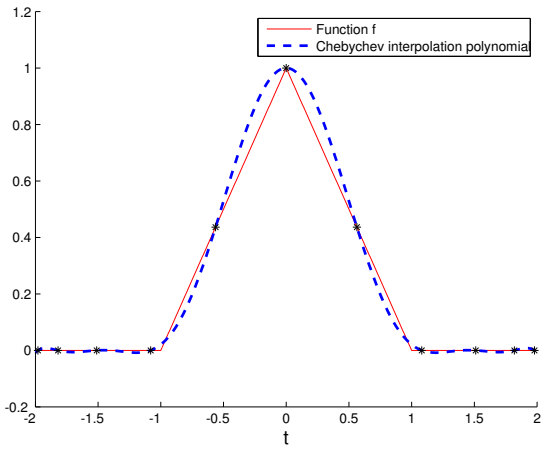


Fig. 224

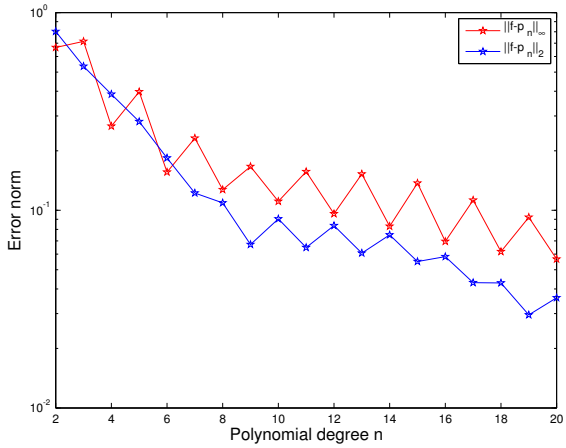


Fig. 225

From the doubly logarithmic plot we conclude ➞

- no exponential convergence
- algebraic convergence (**?**)

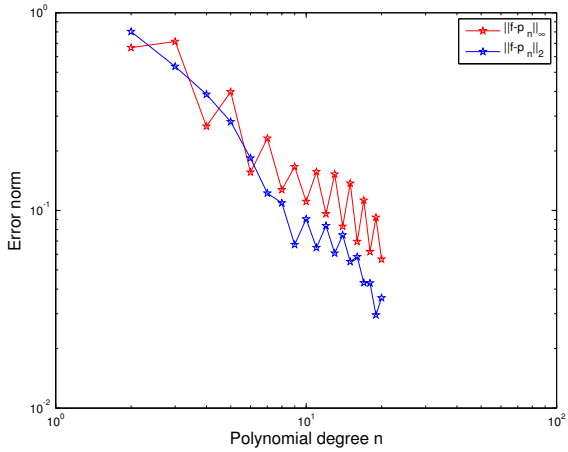


Fig. 226

③ $f(t) = \begin{cases} \frac{1}{2}(1+\cos \pi t) & |t| < 1 \\ 0 & 1 \le |t| \le 2 \end{cases}$ $I = [-2,2]$, $n = 10$ (plot on the left).

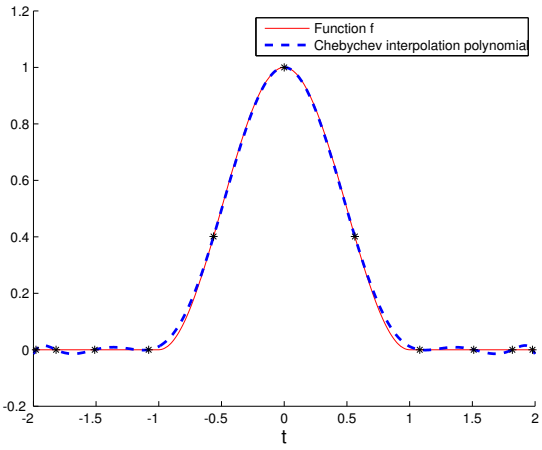


Fig. 227

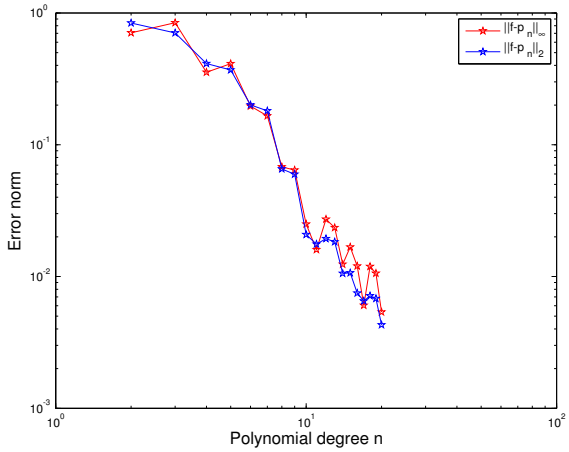


Fig. 228

Notice: only (vaguely) algebraic convergence.

Summary of observations, *cf.* Rem. 6.1.40:

- Essential role of smoothness of f: slow convergence of approximation error of the Cheychev interpolant if f enjoys little smoothness, *cf.* also (6.1.50),
- for analytic $f \in C^\infty$ ($\rightarrow$ Def. 6.1.63) the approximation error of the Cheychev interpolant seems to decay to zero exponentially in the polynomial degree n.

Remark 6.1.96 (Chebychev interpolation of analytic functions)

Assuming that the interpoland f possesses an analytic extension to a complex neighborhood D of $[-1,1]$, we now apply the theory of § 6.1.62 to bound the supremum norm of the Chebychev interpolation error of f on $[-1,1]$.

To convert

$$|f(t) - \mathsf{L}_{\mathcal{T}} f(t)| \le \left| \frac{w(t)}{2\pi\imath} \int_\gamma \frac{f(z)}{(z-t)w(z)} \,\mathrm{d}z \right| \le \frac{|\gamma|}{2\pi} \frac{\max\limits_{a\le\tau\le b} |w(\tau)|}{\min\limits_{z\in\gamma} |w(z)|} \cdot \frac{\max\limits_{z\in\gamma} |f(z)|}{\operatorname{dist}([a,b],\gamma)} \,, \tag{6.1.71}$$

as obtained in § 6.1.62, into a more concrete estimate, we have to study the behavior of

$$w_n(t) = (t-t_0)(t-t_1)\cdot\cdots\cdot(t-t_n) \,, \quad t_k = \cos\left(\frac{2k+1}{2n+2}\,\pi\right), k = 0,\ldots,n \,,$$

where the t_k are the Chebychev nodes according to (6.1.87). They are the zeros of the Chebychev polynomial ($\rightarrow$ Def. 6.1.76) of degree $n+1$. Since w has leading coefficient 1, we conclude $w = 2^{-n} T_{n+1}$, and

$$\max_{-1\le t\le 1} |w(t)| \le 2^{-n} \,. \tag{6.1.97}$$

Next, we fix a suitable path $\gamma \subset D$ for integration:For a constant $\rho > 1$ we set

$$\begin{aligned}\gamma &:= \{z = \cos(\theta - \imath \log\rho),\ 0 \le \theta \le 2\pi\}\\ &= \left\{z = \frac{1}{2}(\exp(\imath(\theta - \imath\log\rho)) + \exp(-\imath(\theta - \imath\log\rho))),\ 0 \le \theta \le 2\pi\right\}\\ &= \left\{z = \frac{1}{2}\left(\rho e^{\imath\theta} + \rho^{-1}e^{-iu\theta}\right),\ 0 \le \theta \le 2\pi\right\}\\ &= \left\{z = \tfrac{1}{2}(\rho+\rho^{-1})\cos\theta + \imath\tfrac{1}{2}(\rho-\rho^{-1})\sin\theta,\ 0 \le \theta \le 2\pi\right\}\end{aligned}$$

Thus, we see that γ is an ellipsis with foci ± 1, large axis $\frac{1}{2}(\rho+\rho^{-1}) > 1$ and small axis $\frac{1}{2}(\rho-\rho^{-1}) > 0$.

Elliptical integration contours for different values of ρ
▷

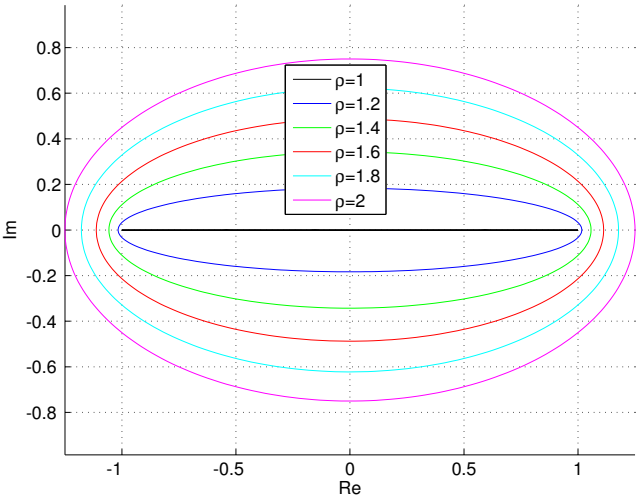


Fig. 229

Appealing to geometric evidence, we find $\operatorname{dist}(\gamma,[-1,1]) = \frac{1}{2}(\rho+\rho^{-1}) - 1$, which gives another term in (6.1.71).

The rationale for choosing this particular integration contour is that the cos in its defintion nicely cancels the arccos in the formula for the Chebychev polynomials. This lets us compute ($s := n+1$)

$$\begin{aligned}|T_s(\cos(\theta - \imath\log\rho))|^2 &= |\cos(s(\theta - \imath\log\rho))|^2\\ &= \cos(s(\theta - \imath\log\rho)) \cdot \overline{\cos(s(\theta - \imath\log\rho))}\\ &= \frac{1}{4}(\rho^s e^{\imath s\theta} + \rho^{-s}e^{-\imath s\theta})(\rho^s e^{-\imath s\theta} + \rho^{-s}e^{\imath s\theta})\\ &= \frac{1}{4}(\rho^{2s} + \rho^{-2s} + e^{2\imath s\theta} + e^{-2\imath s\theta})\\ &= \frac{1}{4}(\rho^s - \underbrace{\rho^{-s}}_{<1})^2 + \underbrace{\frac{1}{4}(e^{\imath s\theta} + e^{-\imath s\theta})^2}_{=\cos^2(s\theta)\ge 0} \ge \frac{1}{4}(\rho^s - 1)^2\,,\end{aligned}$$

for all $0 \le \theta \le 2\pi$, which provides a lower bound for $|w_n|$ on γ. Plugging all these estimates into (6.1.71) we arrive at

$$\|f - \mathsf{L}_\mathcal{T} f\|_{L^\infty([-1,1])} \le \frac{2|\gamma|}{\pi}\frac{1}{(\rho^{n+1}-1)(\rho+\rho^{-1}-2)} \cdot \max_{z\in\gamma}|f(z)|\,. \tag{6.1.98}$$

Note that instead on the nodal polynomial w we have inserted T_{n+1} into (6.1.71), which is a simple multiple. The factor will cancel.

The supremum norm of the interpolation converges exponentially ($\rho > 1$!):

$$\|f - \mathsf{L}_\mathcal{T} f\|_{L^\infty([-1,1])} = O((2\rho)^{-n}) \quad \text{for} \quad n \to \infty\,.$$

Experiment 6.1.99 (Chebychev interpolation of analytic function → Exp. 6.1.93 cnt'd)

Modification: the same function $f(t) = (1+t^2)^{-1}$

on a smaller interval $I = [-1,1]$.

(Faster) exponential convergence than on the interval $I =]-5,5[$:

$$\|f - \mathrm{I}_n f\|_{L^2([-1,1])} \approx 0.42^n.$$

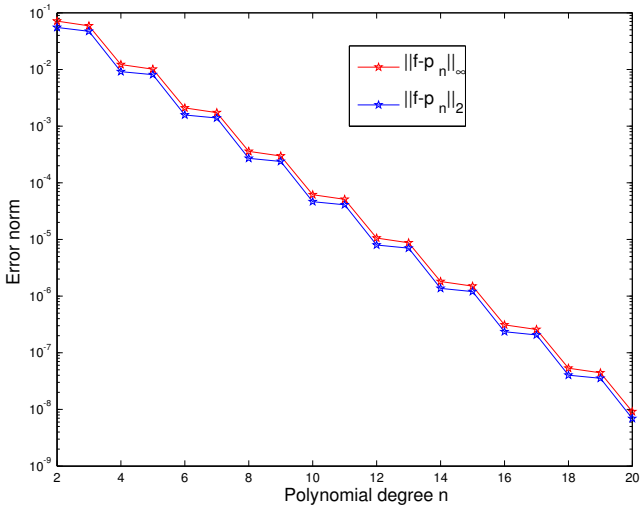


Fig. 230

Explanation, *cf.* Rem. 6.1.96: for $I = [-1,1]$ the poles $\pm i$ of f are farther away relative to the size of the interval than for $I = [-5,5]$.

6.1.3.3 Chebychev interpolation: computational aspects

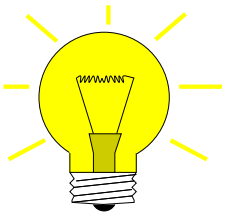
Task: Given: given degree $n \in \mathbb{N}$, continuous function $f : [-1,1] \mapsto \mathbb{R}$

Sought: efficient representation/evaluation of Chebychev interpolant $p \in \mathcal{P}_n$ (= polynomial Lagrange interpolant of degree $\leq n$ in Chebychev nodes (6.1.87) on $[-1,1]$).

More concretely, this boils down to a implementation of the following class:

C++11 code 6.1.100: Definition of class for Chebychev interpolation

```
class ChebInterp {
  private:
  // various internal data describing Chebychev interpolating
     polynomial p
  public:
    // Constructor taking function f and degree n as arguments
    template <typename Function>
      PolyInterp(const Function &f,unsigned int n);
    // Evaluation operator:  y_j = p(x_j), j = 1,...,m  (m "large")
    void eval(const vector<double> &x,vector <double> &y) const;
};
```



Trick: internally represent p as a linear combination of Chebychev polynomials, a Chebychev expansion:

$$p = \sum_{j=0}^{n} \alpha_j T_j \,, \quad \alpha_j \in \mathbb{R} \,. \tag{6.1.101}$$

The representation (6.1.101) is always possible, because $\{T_0, \ldots, T_n\}$ is a *basis* of $\mathcal{P}_n$, owing to $\deg T_n = n$. The representation is amenable to efficient evaluation and computation by means of special algorithms.

(6.1.102) Fast evaluation of Chebychev expansion → [?, Alg. 32.1]

[Fast evaluation of Chebychev expansion]

Task: Given $n \in \mathbb{N}$, $x \in \mathbb{R}$, and the Chebychev expansion coefficients $\alpha_j \in \mathbb{R}$, $j = 0, \ldots, n$, compute $p(x)$ with

$$p(x) = \sum_{j=0}^{n} \alpha_j T_j(x) \,, \quad \alpha_j \in \mathbb{R} \,. \tag{6.1.101}$$

Idea: Use the 3-term recurrence (6.1.78)

$$T_j(x) = 2xT_{j-1}(x) - T_{j-2}(x) \,, \quad j = 2, 3, \ldots \,, \tag{6.1.78}$$

to design a recursive evaluation scheme.

By means of (6.1.78) rewrite (6.1.101) as

$$\begin{aligned} p(x) &= \sum_{j=0}^{n-1} \alpha_j T_j(x) + \alpha_n T_n(x) \\ &\overset{(6.1.78)}{=} \sum_{j=0}^{n-1} \alpha_j T_j(x) + \alpha_n (2xT_{n-1}(x) - T_{n-2}(x)) \\ &= \sum_{j=0}^{n-3} \alpha_j T_j(x) + (\alpha_{n-2} - \alpha_n) T_{n-2}(x) + (\alpha_{n-1} + 2x\alpha_n) T_{n-1}(x) \,. \end{aligned}$$

We recover the point value $p(x)$ as the point value of another polynomial of degree $n-1$ with know Chebychev expansion:

$$p(x) = \sum_{j=0}^{n-1} \widetilde{\alpha}_j T_j(x) \quad \text{with} \quad \widetilde{\alpha}_j = \begin{cases} \alpha_j + 2x\alpha_{j+1} & \text{, if } j = n-1 \,, \\ \alpha_j - \alpha_{j+2} & \text{, if } j = n-2 \,, \\ \alpha_j & \text{else.} \end{cases} \tag{6.1.103}$$

recursive algorithm, see Code 6.1.104.

C++11 code 6.1.104: Recursive evaluation of Chebychev expansion (6.1.101)

```
2 // Recursive evaluation of a polynomial p = sum_{j=1}^{n+1} a_j T_{j-1} at point x
3 // based on (6.1.103)
4 // IN : Vector of coefficients a
5 // evaluation point x
```

```
// OUT: Value at point x
double recclenshaw(const VectorXd& a, const double x) {
  const VectorXd::Index n = a.size() - 1;
  if      (n == 0) return a(0);             // Constant polynomial
  else if (n == 1) return (x*a(1) + a(0)); // Value α_1 * x + α_0
  else {
    VectorXd new_a(n);
    new_a << a.head(n - 2), a(n - 2) - a(n), a(n - 1)+ 2*x*a(n);
    return recclenshaw(new_a,x); // recursion
  }
}
```

Non-recursive version: Clenshaw algorithm

C++11 code 6.1.105: Clenshaw algorithm for evalation of Chebychev expansion (6.1.101)

```
// Clenshaw algorithm for evaluating p = ∑_{j=1}^{n+1} a_j T_{j-1}
// at points passed in vector x
// IN : a = [α_j], coefficients for p = ∑_{j=1}^{n+1} α_j T_{j-1}
// x = (many) evaluation points
// OUT: values p(x_j) for all j
VectorXd clenshaw(const VectorXd& a, const VectorXd& x) {
  const int n = a.size() - 1;  // degree of polynomial
  MatrixXd d(n + 1, x.size()); // temporary storage for intermediate
     values
  for (int c = 0; c < x.size(); ++c) d.col(c) = a;
  for (int j = n - 1; j > 0; --j) {
    d.row(j) += 2*x.transpose().cwiseProduct(d.row(j+1)); // see
       (6.1.103)
    d.row(j-1) -= d.row(j + 1);
  }
  return d.row(0) + x.transpose().cwiseProduct(d.row(1));
}
```

Computational effort : $O(nm)$ for evaluation at m points

(6.1.106) Computation of Chebychev expansions of interpolants

Chebychev interpolation is a linear interpolation scheme, see § 5.1.13. Thus, the expansion α_j in (6.1.101) can be computed by solving a linear system of equations of the form (5.1.15). However, for Chebychev interpolation this linear system can be cast into a very special form, which paves the way for its fast direct solution:

Task: Efficiently compute the Chebychev expansion coefficients α_j in (6.1.101) from the interpolation conditions

$$p(t_k) = f(t_k)\,, \quad k = 0,\ldots,n\,, \quad \text{for} \quad t_k := \cos\left(\frac{2k+1}{2(n+1)}\pi\right)\,. \tag{6.1.107}$$

Chebychev nodes

Trick: transformation of p into a 1-periodic function, which turns out to be a Fourier sum (= finite Fourier series):

$$\begin{aligned} q(s) &:= p(\cos 2\pi s) = \sum_{j=0}^{n} \alpha_j T_j(\cos 2\pi s) \overset{\text{Def. 6.1.76}}{=} \sum_{j=0}^{n} \alpha_j \cos(2\pi j s) \\ &= \sum_{j=0}^{n} \tfrac{1}{2}\alpha_j\big(\exp(2\pi\imath j s) + \exp(-2\pi\imath j s)\big) \quad [\text{ by } \cos z = \tfrac{1}{2}(e^z + e^{-z})\,] \\ &= \sum_{j=-n}^{n+1} \beta_j \exp(-2\pi\imath j s)\,, \quad \text{with} \quad \beta_j := \begin{cases} 0 & \text{, for } j = n+1\,, \\ \frac{1}{2}\alpha_j & \text{, for } j = 1,\ldots,n\,, \\ \alpha_0 & \text{, for } j = 0\,, \\ \frac{1}{2}\alpha_{n-j} & \text{, for } j = -n,\ldots,-1\,. \end{cases} \end{aligned} \tag{6.1.108}$$

Transformed interpolation conditions (6.1.107) for q:

$$t = cos(2\pi s) \overset{(6.1.107)}{\Longrightarrow} q\left(\frac{2k+1}{4(n+1)}\right) = y_k := f(t_k)\,, \quad k = 0,\ldots,n\,. \tag{6.1.109}$$

This is an interpolation problem for *equidistant points on the unit circle* as we have seen them in Section 5.6.3.

Also observe the symmetry

$$\begin{aligned} &q(s) = q(1-s) \\ &\Downarrow\leftarrow \text{(6.1.109)} \\ &q(1 - \frac{2k+1}{4(n+1)}) = y_k\,, \quad k = 0,\ldots,n\,. \end{aligned}$$

It ensures that the coefficients β_j actually satisfy the constraints implied by their relationship with α_j.

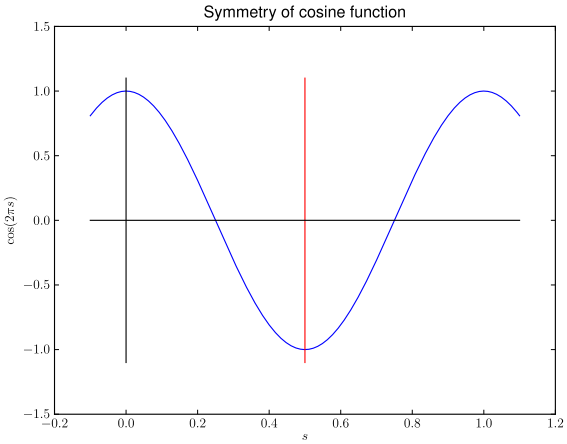


Fig. 231

Extend interpolation conditions (6.1.109) by symmetry, see Fig. 232

$$q\left(\frac{k}{2(n+1)} + \frac{1}{4(n+1)}\right) = z_k := \begin{cases} y_k & \text{, for } k = 0,\ldots,n\,, \\ y_{2n+1-k} & \text{, for } k = n+1,\ldots,2n+1\,. \end{cases} \tag{6.1.110}$$

In a sense, we can mirror the interpolation conditions at $x = \frac{1}{2}$:

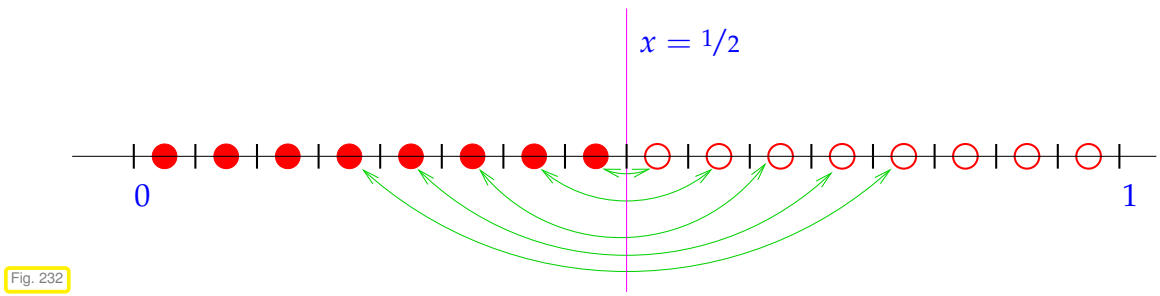


Fig. 232

Chebychev expansion for Chebychev interpolants
⇕
Trigonometric interpolation at equidistant points → Section 5.6.3

Trigonometric interpolation at equidistant points can be done very efficiently by means of FFT-based algorithms, see Code 5.6.15. We can also apply these for the computation of Chebychev expansion coefficients.

Details: (6.1.110) ⟺ square linear system of equations for unknown β_j.

$$q\left(\frac{k}{2(n+1)}+\frac{1}{4(n+1)}\right) = \sum_{j=-n}^{n+1}\left(\beta_j \exp\left(-\frac{2\pi\imath j}{4(n+1)}\right)\right)\exp\left(-\frac{2\pi\imath}{2n+1}kj\right) = z_k\,.$$

$$\Updownarrow$$

$$\sum_{j=0}^{2n+1}\left(\beta_j \exp\left(-\frac{2\pi\imath(j-n)}{4(n+1)}\right)\right)\underbrace{= \exp\left(-\frac{2\pi\imath}{2n+1}kj\right)}_{=\omega_{2(n+1)}^{kj}!} = \exp\left(-\pi\imath\frac{nk}{n+1}\right)z_k\,,\quad k=0,\ldots,2n+1\,.$$

$$\Updownarrow$$

$$\mathbf{F}_{2(n+1)}\mathbf{c}=\mathbf{b} \quad \text{with} \quad \begin{aligned}\mathbf{c} &= \left[\beta_j \exp\left(-\tfrac{2\pi\imath j}{4(n+1)}\right)\right]_{j=0}^{2n+1}\,,\\ \mathbf{b} &= \left[\exp\left(-\pi\imath\tfrac{nk}{n+1}\right)z_k\right]_{k=0}^{2n+1}\,.\end{aligned} \tag{6.1.111}$$

$(2n+2)\times(2n+2)$ Fourier matrix, see (4.2.13)

➤ solve (6.1.111) with inverse discrete Fourier transform, see 4.2:

asymptotic complexity $O(n\log n)$ (→ Section 4.3)

Note: by symmetry of $\mathbf{z}$ ➥ $\beta_{2n+1}=0$, *cf.* (6.1.108)!

MATLAB-code 6.1.112: Efficient computation of Chebychev expansion coefficient of Chebychev interpolant

```
// efficiently compute coefficients α_j in the Chebychev expansion
// p = Σ_{j=0}^{n} α_j T_j of p ∈ P_n based on values y_k,
// k = 0,...,n, in Chebychev nodes t_k, k = 0,...,n
// IN: values y_k passed in y
// OUT: coefficients α_j
VectorXd chebexp(const VectorXd& y) {
  const int n = y.size() - 1;              // degree of polynomial
  const std::complex<double> M_I(0, 1); // imaginary unit
  // create vector z, see (6.1.110)
  VectorXcd b(2*(n + 1));
  const std::complex<double> om = -M_I*(M_PI*n)/((double)(n+1));
  for (int j = 0; j <= n; ++j) {
    b(j) = std::exp(om*double(j))*y(j); // this cast to double is
```

```
        necessary!!
15      b(2*n+1-j) = std::exp(om*double(2*n+1-j))*y(j);
16    }
17
18    // Solve linear system (6.1.111) with effort O(n log n)
19    Eigen::FFT<double> fft;  // EIGEN's helper class for DFT
20    VectorXcd c = fft.inv(b); // -> c = ifft(z), inverse fourier
        transform
21    // recover β_j, see (6.1.111)
22    VectorXd beta(c.size());
23    const std::complex<double> sc = M_PI_2/(n + 1)*M_I;
24    for (unsigned j = 0; j < c.size(); ++j)
25      beta(j) = ( std::exp(sc*double(-n+j))*c[j] ).real();
26    // recover α_j, see (6.1.108)
27    VectorXd alpha = 2*beta.segment(n,n); alpha(0) = beta(n);
28    return alpha;
29  }
```

Remark 6.1.113 (Chebychev representation of built-in functions)

Computers use approximation by sums of Chebychev polynomials in the computation of functions like $\log, \exp, \sin, \cos, \ldots$. The evaluation by means of Clenshaw algorithm according to Code 6.1.105 is more efficient and stable than the approximation by Taylor polynomials.

6.2 Mean Square Best Approximation

There is a particular family of norms for which the best approximant of a function f in a finite dimensional function space V_N, that is, the element of V_N that is closest to f with respect to that particular norm can actually be computed. It turns out that this computation boils down to solving a kind of least squares problem, similar to the least squares problems in $\mathbb{K}^n$ discussed in Chapter 3.

6.2.1 Abstract theory

Concerning mean square best approximation it is useful to learn an abstract framework first into which the concrete examples can be fit later.

6.2.1.1 Mean square norms

Mean square norms generalize the Euclidean norm on $\mathbb{K}^n$, see [**?**, Sect. 4.4]. In a sense, they endow a vector space with a geometry and give a meaning to concepts like "orthogonality".

Definition 6.2.1. (Semi-)inner product [?, Sect. 4.4]

Let V be a vector space over the field $\mathbb{K}$. A mapping $b : V \times V \to \mathbb{K}$ is called an inner product on V, if it satisfies

(i) b is linear in the first argument: $b(\alpha v + \beta w, u) = \alpha b(v,u) + \beta b(w,u)$ for all $\alpha, \beta \in \mathbb{K}$, $u, v, w \in V$,

(ii) b is (anti-)symmetric: $b(v,w) = \overline{b(w,v)}$ ($\overline{} \hat{=}$ complex conjugation),

(iii) b is positive definite: $v \neq 0 \Leftrightarrow b(v,v) > 0$.

b is a semi-inner product, if it still complies with (i) and (ii), but is only positive semi-definite: $b(v,v) \geq 0$ for all $v \in V$.

✎ notation: usually we write $(\cdot,\cdot)_V$ for an inner product on the vector space V.

Definition 6.2.2. Orthogonality

Let V be a vector space equipped with a (semi-)inner product $(\cdot,\cdot)_V$. Any two elements v and w of V are called orthogonal, if $(v,w)_V = 0$. We write $v \perp w$.

✎ notation: If $W \subset V$ is a (closed) subsspace: $v \perp W :\Leftrightarrow (v,w)_V = 0 \; \forall \, w \in W$.

Theorem 6.2.3. Mean square (semi-)norm/Inner product (semi-)norm

If $(\cdot,\cdot)_V$ is a (semi-)inner product ($\rightarrow$ Def. 6.2.1) on the vector space V, then

$$\|v\|_V := \sqrt{(v,v)_V}$$

defines a (semi-)norm ($\rightarrow$ Def. 1.5.70) on V, the mean square (semi-)norm/ inner product (semi-)norm induced by $(\cdot,\cdot)_V$.

(6.2.4) Examples for mean square norms

✦ The Euclidean norm on $\mathbb{K}^n$ induced by the dot product (Euclidean inner product).

$$(\mathbf{x},\mathbf{y})_{\mathbb{K}^n} := \sum_{j=1}^{n} (\mathbf{x})_j \overline{(\mathbf{y})_j} \quad \text{["Mathematical indexing" !]} \quad \mathbf{x},\mathbf{y} \in \mathbb{K}^n \, .$$

✦ The L^2-norm (5.2.67) on $C^0([a,b])$ induced by the $L^2([a,b])$ inner product

$$(f,g)_{L^2([a,b])} := \int_a^b f(\tau)\overline{g}(\tau)\,\mathrm{d}\tau \, , \quad f,g, \in C^0([a,b]) \, . \tag{6.2.5}$$

6.2.1.2 Normal equations

Mean square best approximation = best approximation in a mean square norm

From § 3.1.8 we know that in Euclidean space $\mathbb{K}^n$ the best approximation of vector $\mathbf{x} \in \mathbb{K}^n$ in a subspace $V \subset \mathbb{K}^n$ is unique and given by the orthogonal projection of $\mathbf{x}$ onto V. Now we generalize this to vector spaces equipped with inner products.

$X \triangleq$ a vector space over $\mathbb{K} = \mathbb{R}$, equipped with an mean square semi-norm $\|\cdot\|_X$ induced by a semi-inner product $(\cdot,\cdot)_X$, see Thm. 6.2.3.
It can be an infinite dimensional function space, e.g., $X = C^0([a,b])$.

$V \triangleq$ a finite-dimensional subspace of X, with basis $\mathfrak{B}_V := \{b_1,\ldots,b_N\} \subset V$, $N := \dim V$.

Assumption 6.2.6.

The semi-inner product $(\cdot,\cdot)_X$ is a genuine inner product ($\rightarrow$ Def. 6.2.1) on V, that is, it is positive definite: $(v,v)_X > 0 \; \forall v \in V \setminus \{0\}$.

Now we give a *formula* for the element q of V, which is nearest to a given element f of X with respect to the norm $\|\cdot\|_X$. This is a genuine generalization of Thm. 3.1.10.

Theorem 6.2.7. Mean square norm best approximation through normal equations

Given any $f \in X$ there is a unique *$q \in V$ such that*

$$\|f-q\|_X = \inf_{p\in V} \|f-p\|_X \,.$$

Its coefficients γ_j, $j = 1,\ldots,N$, with respect to the basis $\mathfrak{B}_V := \{b_1,\ldots,b_N\}$ of V ($q = \sum_{j=1}^N \gamma_j b_j$) are the unique solution of the normal equations

$$\mathbf{M}\left[\gamma_j\right]_{j=1}^N = \left[\left(f,b_j\right)_X\right]_{j=1}^N, \quad \mathbf{M} := \begin{bmatrix} (b_1,b_1)_X & \ldots & (b_1,b_N)_X \\ \vdots & & \vdots \\ (b_N,b_1)_X & \ldots & (b_N,b_N)_X \end{bmatrix} \in \mathbb{K}^{N,N} \,. \tag{6.2.8}$$

Proof. (inspired by Rem. 3.1.14) We first show that $\mathbf{M}$ is s.p.d. ($\rightarrow$ Def. 1.1.8). Symmetry is clear from the definition and the symmetry of $(\cdot,\cdot)_X$. That $\mathbf{M}$ is even positive definite follows from

$$\mathbf{x}^{\mathrm{H}}\mathbf{M}\mathbf{x} = \sum_{k=1}^N \sum_{j=1}^N \xi_k \overline{\xi}_j \left(b_k,b_j\right)_X = \left\|\sum_{j=1}^N \xi_j b_j\right\|_X^2 > 0 \,, \tag{6.2.9}$$

if $\mathbf{x} := \left[\xi_j\right]_{j=1}^N \neq \mathbf{0} \;\Leftrightarrow\; \sum_{j=1}^N \xi_j b_j \neq 0$, since $\|\cdot\|_X$ is a *norm* on V by Ass. 6.2.6.

Now, writing $\mathbf{c} := \left[\gamma_j\right]_{j=1}^N \in \mathbb{K}^N$, $\mathbf{b} := \left[\left(f,b_j\right)_X\right]_{j=1}^N \in \mathbb{K}^N$, and using the basis representation

$$q = \sum_{j=1}^N \gamma_j b_j \,,$$

we find

$$\Phi(\mathbf{c}) := \|f-q\|_X^2 = \|f\|_X^2 - 2\mathbf{b}^\top\mathbf{c} + \mathbf{c}^\top\mathbf{M}\mathbf{c} \,.$$

Applying the differentiation rules from Ex. 8.4.12 to $\Phi : \mathbb{K}^N \to \mathbb{R}$, $\mathbf{c} \mapsto \Phi(\mathbf{c})$, we obtain

$$\mathbf{grad}\,\Phi(\mathbf{c}) = 2(\mathbf{M}\mathbf{c} - \mathbf{b})\,, \tag{6.2.10}$$

$$\mathrm{H}\,\Phi(\mathbf{c}) = 2\mathbf{M}\,. \quad \text{(independent of } \mathbf{c}\text{!)} \tag{6.2.11}$$

Since $\mathbf{M}$ is s.p.d., the unique solution of $\mathbf{grad}\,\Phi(\mathbf{c}) = \mathbf{M}\mathbf{c} - \mathbf{b} = 0$ yields the unique global minimizer of Φ; the Hessian $2\mathbf{M}$ is s.p.d. everywhere!

□

The unique q from Thm. 6.2.7 is called the best approximant of f in V.

Corollary 6.2.12. Best approximant by orthogonal projection

If q is the best approximant of f in V, then $f - q$ is orthogonal to every $p \in V$:

$$(f - q, p)_X = 0 \quad \forall p \in V \quad \Leftrightarrow \quad f - q \perp V\,.$$

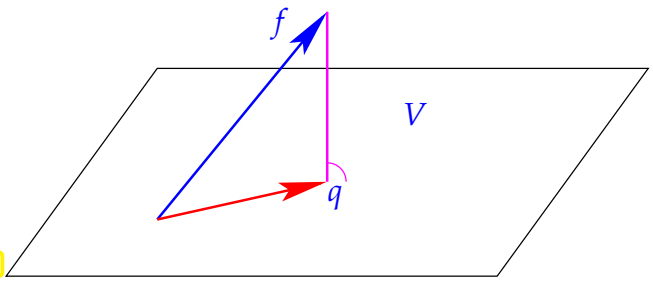


Fig. 233

The message of Cor. 6.2.12:

◁ the best approximation error $f - q$ for $f \in X$ in V is orthogonal to the subspace V.

See § 3.1.8 for related discussion in Euclidean space $\mathbb{K}^n$.

Remark 6.2.13 (Connetion with linear least squares problems Chapter 3)

In Section 3.1.1 we introduced the concept of least squares solutions of overdetermined linear systems of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $m > n$, see Def. 3.1.3. Thm. 3.1.10 taught that the normal equations $\mathbf{A}^\top\mathbf{A}\mathbf{X} = \mathbf{A}^\top\mathbf{b}$ give the least squares solution, if $\mathrm{rank}(\mathbf{A}) = n$.

In fact, Thm. 3.1.10 and the above Thm. 6.2.7 agree if $X = \mathbb{K}^n$ (Euclidean space) and $V = \mathrm{Span}\{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$, where $\mathbf{a}_j \in \mathbb{R}^m$ are the columns of $\mathbf{A}$ and $N = n$.

6.2.1.3 Orthonormal bases

In the setting of Section 6.2.1.2 we may ask: Which choice of basis $\mathfrak{B} = \{b_1, \ldots, b_N\}$ of $V \subset X$ renders the normal equations (6.2.8) particularly simple? Answer: A basis $\mathfrak{B}$, for which $(b_k, b_j)_X = \delta_{kj}$ (δ_{kj} the Kronecker symbol), because this will imply $\mathbf{M} = \mathbf{I}$ for the coefficient matrix of the normal equations.

Definition 6.2.14. Orthonormal basis

A subset $\{b_1, \ldots, b_N\}$ of an N-dimensional vector space V with inner product (→ Def. 6.2.1) $(\cdot,\cdot)_V$ is an orthonormal basis (ONB), if $(b_k, b_j)_V = \delta_{kj}$.

A basis of $\{b_1, \ldots, b_N\}$ of V is called orthogonal, if $(b_k, b_j)_V = 0$ for $k \neq j$.

Corollary 6.2.15. ONB representation of best approximant

If $\{b_1,\ldots,b_N\}$ is an orthonormal basis (→ Def. 6.2.14) of $V \subset X$, then the best approximant $q := \operatorname{argmin}_{p\in V}\|f-p\|_X$ of $f \in X$ has the representation

$$q = \sum_{j=1}^{N} (f,b_j)_X b_j \,. \tag{6.2.16}$$

(6.2.17) Gram-Schmidt orthonormalization

From Section 1.5.1 we already know how to compute orthonormal bases: The algorithm from § 1.5.1 can be run in the framework of any vector space V endowed with an inner product $(\cdot,\cdot)_V$ and induced mean square norm $\|\cdot\|_V$.

```
1:  b_1 := p_1/‖p_1‖_V    % 1st output vector
2:  for j = 2,...,k do
      {   % Orthogonal projection
3:      b_j := p_j
4:      for ℓ = 1,2,...,j−1 do
5:        { b_j ← b_j − (p_j, b_ℓ)_V b_ℓ }
6:      if ( b_j = 0 ) then STOP
7:      else { b_j ← b_j/‖b_j‖_V }
      }
```

(6.2.18)

Theorem 6.2.19. Gram-Schmidt orthonormalization

When supplied with $k \in \mathbb{N}$ linearly independent vectors $p_1,\ldots,p_k \in V$ in a vector space with inner product $(\cdot,\cdot)_V$, Algorithm (6.2.18) computes vectors $b_1,\ldots,b_k$ with

$$(b_\ell,b_j)_V = \delta_{\ell j}\,, \quad \ell,j \in \{1,\ldots,k\}\,,$$
$$\operatorname{Span}\{b_1,\ldots,b_\ell\} = \operatorname{Span}\{p_1,\ldots,p_\ell\}$$

for all $\ell \in \{1,\ldots,k\}$.

This suggests the following alternative approach to the computation of the mean square best approximant q in V of $f \in X$:

❶ Orthonormalize a basis $\{b_1,\ldots,b_N\}$ of V, $N := \dim V$, using Gram-Schmidt algorithm (6.2.18).

❷ Compute q according to (6.2.16).

Number of inner products to be evaluated: $O(N^2)$ for $N \to \infty$.

6.2.2 Polynomial mean square best approximation

Now we apply the results of Section 6.2.1 in the following setting:

X: function space $C^0([a,b])$, $-\infty < a < b < \infty$, of $\mathbb{R}$-values continuous functions,
V: space $\mathcal{P}_m$ of polynomials of degree $\le m$.

Remark 6.2.20 (Inner products on spaces $\mathcal{P}_m$ of polynomials)

To match the abstract framework of Section 6.2.1 we need to find (semi-)inner products on $C^0([a,b])$ that supply positive definite inner products on $\mathcal{P}_m$. The following options are commonly considered:

- On any interval $[a,b]$ we can use the $L^2([a,b])$-inner product $(\cdot,\cdot)_{L^2([a,b)]}$, defined in (6.2.5).
- Given a positive weight function $w : [a,b] \to \mathbb{R}$, $w(t) > 0$ for all $t \in [a,b]$, we can consider the weighted L^2-inner product on the interval $[a,b]$

$$(f,g)_{w,[a,b]} := \int_a^b w(\tau) f(\tau) \overline{g}(\tau) \,\mathrm{d}\tau \,. \tag{6.2.21}$$

- For $n \geq m$ and $n+1$ distinct points collected in the set $\mathcal{T} := \{t_0, t_1, \ldots, t_n\} \subset [a,b]$ we can use the discrete L^2-inner product

$$(f,g)_{\mathcal{T}} := \sum_{j=0}^{n} f(t_j)\overline{g(t_j)} \,. \tag{6.2.22}$$

  Since a polynomial of degree $\leq m$ must be zero everywhere, if it vanishes in at least $m+1$ distinct points, $(f,g)_{\mathcal{T}}$ is positive definite on $\mathcal{P}_m$.

For all these inner products on $\mathcal{P}_m$ holds

$$(\{t \mapsto tf(t)\}, g)_X = (f, \{t \mapsto tg(t)\})_X \,, \quad f,g \in C^0([a,b]) \,, \tag{6.2.23}$$

that is, multiplication with the independent variable can be shifted to the other function inside the inner product.

✎ notation: Note that we have to plug a function into the slots of the inner products; this is indicated by the notation $\{t \mapsto \ldots\}$.

Assumption 6.2.24. Self-adjointness of multiplication operator

We assume the inner product $(\cdot,\cdot)_X$ to satisfy (6.2.23).

The ideas of Section 6.2.1.3 that center around the use of orthonormal bases can also be applied to polynomials.

Definition 6.2.25. Orthonormal polynomials → Def. 6.2.14

Let $(\cdot,\cdot)_X$ be an inner product on $\mathcal{P}_m$. A sequence $r_0, r_1, \ldots, r_m$ provides orthonormal polynomials (ONPs) with respect to $(\cdot,\cdot)_X$, if

$$r_\ell \in \mathcal{P}_\ell \quad , \quad (r_k, r_\ell)_X = \delta_{k\ell} \,, \quad \ell,k \in \{0,\ldots,m\} \,. \tag{6.2.26}$$

The polynomials are just orthogonal, if $(r_k, r_\ell)_X = 0$ for $k \neq \ell$.

By virtue of Thm. 6.2.19 orthonormal polynomials can be generated by applying Gram-Schmidt orthonormalization from § 6.2.17 to the ordered basis of monomials $\{t \mapsto t^j\}_{j=0}^m$.

Lemma 6.2.27. Uniqueness of orthonormal polynomials

The sequence of orthonormal polynomials from Def. 6.2.25 is unique up to signs, supplies an $(\cdot,\cdot)_X$-orthonormal basis ($\rightarrow$ Def. 6.2.14) of $\mathcal{P}_m$, and satisfies

$$\operatorname{Span}\{r_0,\ldots,r_k\} = \mathcal{P}_k\,, \quad k \in \{0,\ldots,m\}\,. \tag{6.2.28}$$

Proof. Comparing Def. 6.2.14 and (6.2.26) the ONB-property of $\{r_0,\ldots,r_m\}$ is immediate. Then (6.2.26) follows from dimensional considerations.

r_0 must be a constant, which, up to sign, is fixed by the normalization condition $\|r_0\|_X = 1$.

$\mathcal{P}_{k-1} \subset \mathcal{P}_k$ has co-dimension 1 so that there a unit "vector" in $\mathcal{P}_k$, which is orthogonal to $\mathcal{P}_{k-1}$ and unique up to sign.

□

(6.2.29) Orthonormal polynomials by orthogonal projection

Let $r_0,\ldots,r_m \in \mathcal{T}_p$ be a sequence of orthonormal polynomials according to Def. 6.2.25. From (6.2.26) we conclude that $r_k \in \mathcal{P}_k$ has leading coefficient $\neq 0$.

Hence $s_k(t) := t\cdot r_k(t)$ is a polynomial of degree $k+1$ with leading coefficient $\neq 0$, that is $s_k \in \mathcal{P}_{k+1}\setminus\mathcal{P}_k$. Therefore, r_{k+1} can be obtain by *orthogonally projecting* s_k onto $\mathcal{P}_k$ plus normalization, *cf.* Lines 4-5 of Algorithm (6.2.18):

$$r_{k+1} = \pm\frac{\widetilde{r}_{k+1}}{\|\widetilde{r}_{k+1}\|_X}\,, \quad \widetilde{r}_{k+1} = s_k - \sum_{j=0}^{k}(s_k,r_j)_X r_j\,. \tag{6.2.30}$$

Straightforward computations confirm that $r_{k+1} \perp \mathcal{P}_k$.

The sum in (6.2.30) collapses to two terms! In fact, since $(r_k,q)_X = 0$ for all $q \in \mathcal{P}_{k-1}$, by Ass. 6.2.24

$$(s_k,r_j)_X = (\{t\mapsto tr_k(t)\},r_j)_X \overset{(6.2.23)}{=} (r_k,\{t\mapsto tr_j\})_X = 0\,, \quad \text{if } j < k-1\,,$$

because in this case $\{t\mapsto tr_j\} \in \mathcal{P}_{k-1}$. As a consequence (6.2.30) reduces to the 3-term recursion

$$\begin{aligned} r_{k+1} &= \pm\frac{\widetilde{r}_{k+1}}{\|\widetilde{r}_{k+1}\|_X}\,,\\ \widetilde{r}_{k+1} &= s_k - (\{t\mapsto tr_k\},r_k)_X r_k - (\{t\mapsto tr_k\},r_{k-1})_X r_{k-1} \end{aligned}\,, \quad k = 1,\ldots,m-1\,. \tag{6.2.31}$$

The recursion starts with $r_{-1} := 0$, $r_0 = \{t\mapsto 1/\|1\|_X\}$.

The 3-term recursion (6.2.31) can be recast in various ways. Forgoing normalization the next theorem presents one of them.

Theorem 6.2.32. 3-term recursion for orthogonal polynomials

*Given **any** inner product $(\cdot,\cdot)_X$ on $\mathcal{P}_m$, $m \in \mathbb{N}$, define $p_{-1} := 0$, $p_0 = 1$, and*

$$p_{k+1}(t) := (t-\alpha_{k+1})p_k(t) - \beta_k p_{k-1}(t)\,, \quad k = 0,1,\ldots,m-1\,,$$
$$\text{with} \quad \alpha_{k+1} := \frac{(\{t \mapsto tp_k(t)\}, p_k)_X}{\|p_k\|_X^2}\,, \quad \beta_k := \frac{\|p_k\|_X^2}{\|p_{k-1}\|_X^2}\,. \tag{6.2.33}$$

Then
- $p_k \in \mathcal{P}_k$ *has leading coefficient* $= 1$, *and*
- $\{p_0, p_1, \ldots, p_m\}$ *is an orthogonal basis of* $\mathcal{P}_m$.

Proof. (by rather straightforward induction) We first confirm, thanks to the definition of α_1,

$$(p_0,p_1)_X = (p_0, \{t \mapsto (t-\alpha_1)p_0(t)\})_X = (p_0, \{t \mapsto tp_0(t)\})_X - \alpha_1 (p_0,p_0)_X = 0\,.$$

For the induction step we assume that the assertion is true for $p_0,\ldots,p_k$ and observe that for p_{k+1} according to (6.2.33) we have

$$\begin{aligned}
(p_k,p_{k+1})_X &= (p_k, \{t \mapsto (t-\alpha_{k+1})p_k(t)\} - \beta_k p_{k-1})_X \\
&= (p_k, \{t \mapsto tp_k(t)\})_X - \underbrace{\alpha_{k+1}(p_k,p_k)_X}_{=(\{t\mapsto tp_k(t)\},p_k)_X} - \beta_k \underbrace{(p_k,p_{k-1})_X}_{=0} = 0\,, \\
(p_{k-1},p_{k+1})_X &= (p_{k-1}, \{t \mapsto (t-\alpha_{k+1})p_k(t)\} - \beta_k p_{k-1})_X \\
&= (p_{k-1}, \{t \mapsto tp_k(t)\})_X - \alpha_{k+1}(p_{k-1},p_k)_X - \beta_k (p_k,p_{k-1})_X = 0\,, \\
(p_\ell,p_{k+1})_X &= (p_\ell, \{t \mapsto (t-\alpha_{k+1})p_k(t)\} - \beta_k p_{k-1})_X \\
&= \underbrace{(p_\ell, \{t \mapsto tp_k(t)\})_X}_{=0} - \alpha_{k+1} \underbrace{(p_\ell,p_k)_X}_{0} - \beta_k \underbrace{(p_\ell,p_{k-1})_X}_{0} = 0\,, \quad \ell = 0,\ldots,k-2\,.
\end{aligned}$$

This amounts to the assertion of orthogonality for $k+1$. Above, several inner product vanish because of the induction hypothesis!

□

Remark 6.2.34 ($L^2([-1,1])$-orthogonal polynomials)

An important inner product on $C^0[-1,1]$ is the L^2-inner product, see (6.2.5)

$$(f,g)_{L^2([-1,1])} := \int_{-1}^{1} f(\tau)\overline{g}(\tau)\,\mathrm{d}\tau\,, \quad f,g, \in C^0([-1,1])\,.$$

It is a natural question what is the unique sequence of $L^2([-1,1])$-orthonormal polynomials. Their rather simple characterization will be discussed in the sequel.

The Legendre polynomials P_n can be defined by the 3-term recursion

$$P_{n+1}(t) := \frac{2n+1}{n+1} tP_n(t) - \frac{n}{n+1} P_{n-1}(t) , \tag{7.3.33}$$

$$P_0 := 1 , \quad P_1(t) := t .$$

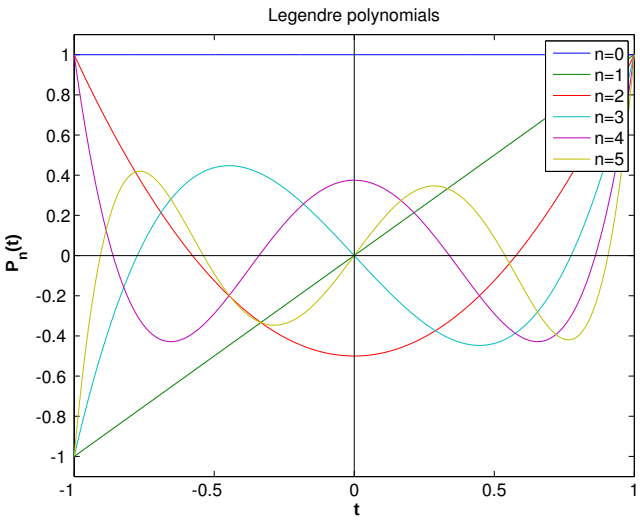


Fig. 234

$P_k \in \mathcal{P}_k$ is immediate and so is the parity

$$P_k(t) = P_k(-t) \quad \text{if } k \text{ is even}, \quad P_k(t) = -P_k(-t) \quad \text{if } k \text{ is odd} . \tag{6.2.35}$$

Orthogonality, $(P_k, P_m)_{L^2(\Omega)} = 0$ or $m \neq k$, as well as $\|\|]\|_{L^{P_k([-1,1)}}2^2 = {}^2/_{2n+1}$ can be proved by induction based on (6.2.35).This means implies that the $L^]([-1,1)2$-orthonormal polynomials are $r_k = \sqrt{k+{}^1/_2}P_k$ and their 3-term recursion from **??** reads

$$r_{n+1}(t) = \frac{\sqrt{4(n+1)^2-1}}{n+1} tr_n(t) - \frac{n}{n+1}\sqrt{\frac{4(n+1)^2-1}{4n^2-1}}\, u_{n-1}(t) , \tag{6.2.36}$$

$$r_{-1} := 0 , \quad r_0 = \tfrac{1}{2}\sqrt{2} . \tag{6.2.37}$$

(6.2.38) Discrete orthogonal polynomials

Since they involve integrals, weighted L^2-inner products (6.2.21) are not accessible computationally, unless one resigns to approximation, see Chapter 7 for corresponding theory and techniques.

Therefore, given a point set $\mathcal{T} := \{t_0, t_1, \ldots, t_n\}$, we focus on the associated discrete L^2-inner product

$$(f,g)_X := (f,g)_{\mathcal{T}} := \sum_{j=0}^{n} f(t_j)\overline{g(t_j)} , \quad f,g \in C^0([a,b]) , \tag{6.2.22}$$

which is positive definite on $\mathcal{P}_n$ and satisfies Ass. 6.2.24.

The polynomials p_k generated by the 3-term recursion (6.2.33) from Thm. 6.2.32 are then called discrete orthogonal polynomials. The following C++ code computes the recursion coefficients α_k and β_k, $k = 1, \ldots, n-1$.

C++11 code 6.2.39: Computation of weights in 3-term recursion for discrete orthogonal polynomials

```
2 // Computation of coefficients α, β from 6.2.32
3 // IN : t = points in the definition of the discrete L^2-inner product
```

```
4  // n = maximal index desired
5  // alpha, beta are used to save coefficients of recursion
6  void coeffortho(const VectorXd& t, const unsigned n, VectorXd& alpha,
       VectorXd& beta) {
7    const unsigned m = t.size(); // maximal degree of orthogonal
       polynomial
8    alpha = VectorXd( std::min(n-1, m-2) + 1 );
9    beta = VectorXd( std::min(n-1, m-2) + 1 );
10   alpha(0) = t.sum()/m;
11   // initialization of recursion; we store only the values of
12   // the polynomials a the points in T
13   VectorXd p0,
14            p1 = VectorXd::Ones(m),
15            p2 = t - alpha(0)*VectorXd::Ones(m);
16   for (unsigned k = 0; k < std::min(n-1, m-2); ++k) {
17     p0 = p1; p1 = p2;
18     // 3-term recursion (6.2.33)
19     alpha(k+1) = p1.dot(t.cwiseProduct(p1))/p1.squaredNorm();
20     beta(k) = p1.squaredNorm()/p0.squaredNorm();
21     p2 = (t-alpha(k+1)*VectorXd::Ones(m)).cwiseProduct(p1)-beta(k)*p0;
22   }
23 }
```

(6.2.40) Polynomial fitting

Given a point set $\mathcal{T} := \{t_0, t_1, \ldots, t_n\} \subset [a,b]$, and a function $f : [a,b] \to \mathbb{K}$, we may seek to approximate f by its polynomial best approximant with respect to the discrete L^2-norm $\|\cdot\|_{\mathcal{T}}$ induced by the discrete L^2-inner product (6.2.22).

Definition 6.2.41. Fitted polynomial

Given a point set $\mathcal{T} := \{t_0, t_1, \ldots, t_n\} \subset [a,b]$, and a function $f : [a,b] \to \mathbb{K}$ we call

$$q_k := \operatorname*{argmin}_{p \in \mathcal{P}_k} \|f - p\|_{\mathcal{T}}\,, \quad k \in \{0, \ldots, n\}\,,$$

the fitting polynomial to f on $\mathcal{T}$ of degree k.

The stable and efficient computation of fitting polynomials can rely on combining Thm. 6.2.32 with Cor. 6.2.15:

❶ (Pre-)compute the weights α_ℓ and β_ℓ for the 3-term recursion (6.2.33).

❷ (Pre-)compute the values of the orthogonal polynomials p_k at desired evaluation points $x_i \in \mathbb{R}$, $i = 1, \ldots, N$.

❸ Compute the inner product $(f, p_\ell)_X$, $\ell = 0, \ldots, k$, and use (6.2.16) to linearly combine the vectors $[p_\ell(x_i)]_{i=1}^N$, $\ell = 0, \ldots, k$.

This yields $[q(x_i)]_{i=1}^N$, $q \in \mathcal{P}_k$ the fitting polynomial.

Example 6.2.42 (Approximation by discrete polynomial fitting)

We use equidistant points $\mathcal{T} := \{t_k = -1 + k\frac{2}{m},\ k = 0,\ldots,m\} \subset [-1,1]$, $m \in \mathbb{N}$ to compute fitting polynomials ($\rightarrow$ Def. 6.2.41) for two different functions.

We monitor the L^2-norm and L^∞-norm of the approximation error, both norms approximated by sampling in $\xi_j = -1 + \frac{j}{500}$, $j = 0,\ldots,1000$.

① $f(t) = (1+(5t)^2)^{-1}$, $I = [-1,1]$ $\rightarrow$ Ex. 6.1.41, analytic in complex neighborhood of $[-1,1]$:

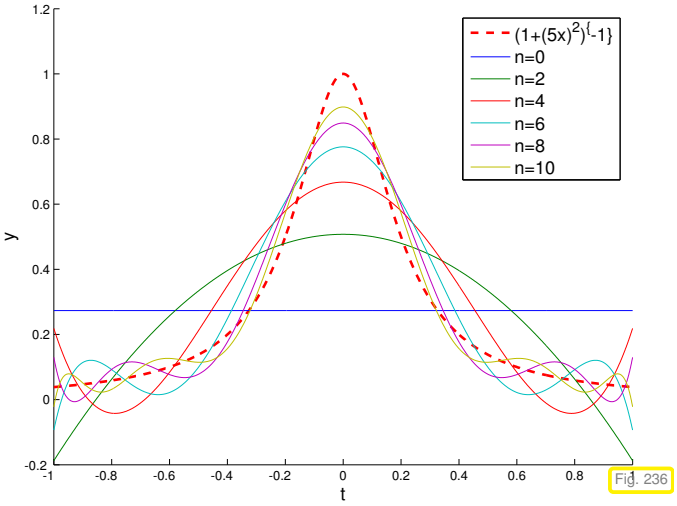


Fig. 235

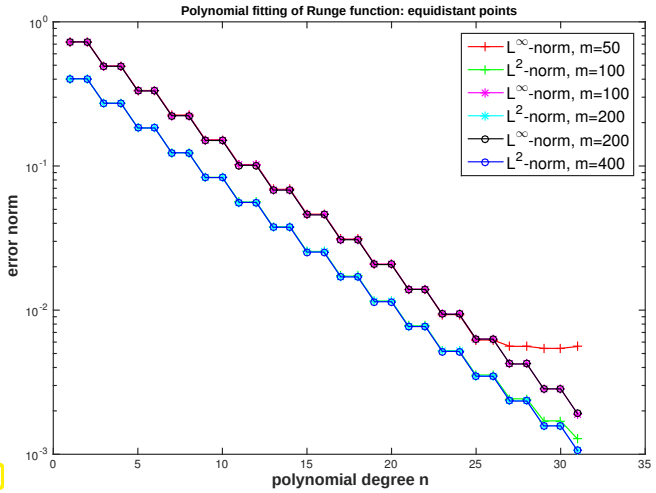


Fig. 236

➢ We observe exponential convergence ($\rightarrow$ Def. 6.1.38) in the polynomial degree n.

② $f(t) = \max\{0, 1 - 2 * |x + \frac{1}{4}|\}$, f only in $C^0([-1,1])$:

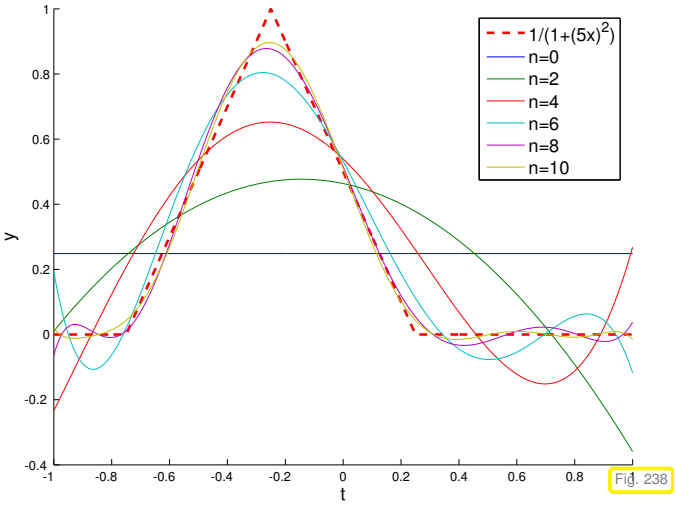


Fig. 237

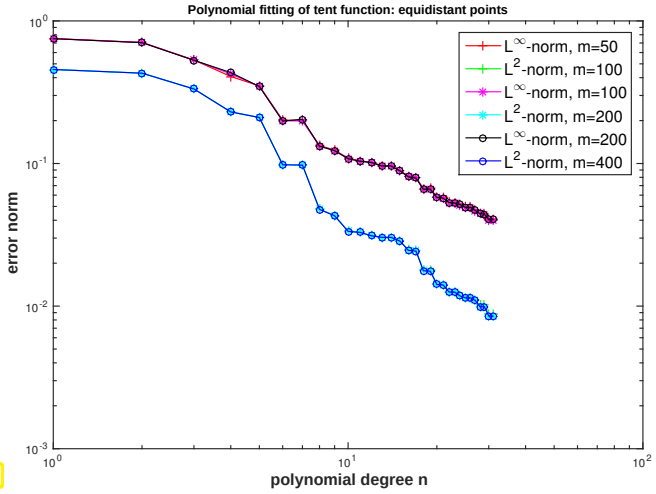


Fig. 238

➢ We observe only algebraic convergence (→ Def. 6.1.38 in the polynomial degree n (for $n \ll m$!).

③ "bump function"

$$f(t) = \max\{\cos(4\pi|t + \frac{1}{4}|), 0\} \ .$$

➢ Merely $f \in C^1([-1,1])$

Doubly logarithmic plot suggests "asymptotic" algebraic convergence.

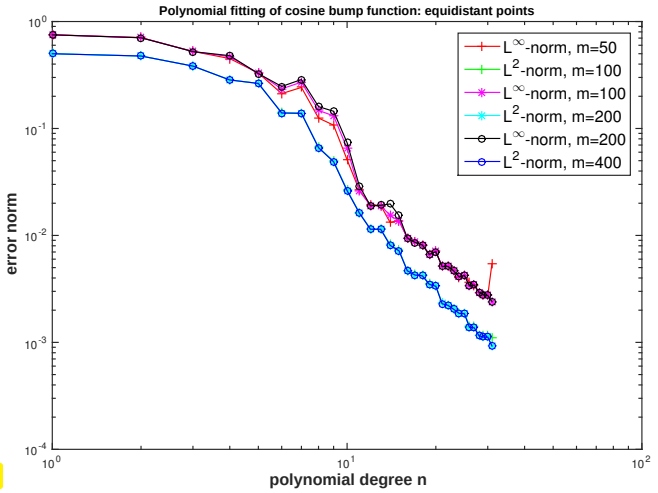


Fig. 239

6.3 Uniform Best Approximation

(6.3.1) The alternation theorem

Given an interval $[a,b]$ we seek a best approximant of a function $f \in C^0([a,b])$ in the space $\mathcal{P}_n$ of polynomials of degree $\leq n$ with respect to the supremum norm $\|\cdot\|_{L^\infty([a,b])}$:

$$q \in \operatorname*{argmin}_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty(I)} \ .$$

The results of Section 6.2.1 cannot be applied because the supremum norm is not induced by an inner product on $\mathcal{P}_n$.

Theory provides us with surprisingly precise necessary and sufficient conditions to be satisfied by the polynomial $L^\infty([a,b])$-best approximant q.

Theorem 6.3.2. Chebychev alternation theorem

Given $f \in C^0[a,b]$, $a < b$, and a polynomial degree $n \in \mathbb{N}$, a polynomial $q \in \mathcal{P}_n$ satisfies

$$q = \operatorname*{argmin}_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty(I)}$$

if and only if *there exist $n+2$ points $a \le \xi_0 < \xi_1 < \cdots < \xi_{n+1} \le b$ such that*

$$|e(\xi_j)| = \|e\|_{L^\infty([a,b])}, \quad j = 0, \ldots, n+1,$$
$$e(\xi_j) = -e(\xi_{j+1}), \quad j = 0, \ldots, n,$$

where $e := f - q$ denotes the approximation error.

Visualization of the behavior of the $L^\infty([a,b])$-best approximation error $e := f - q$ according to the Chebychev alternation theorem Thm. 6.3.2. ▷

The extrema of the approximation error are sometimes called alternants.

Compare with the shape of the Chebychev polynomials → Def. 6.1.76.

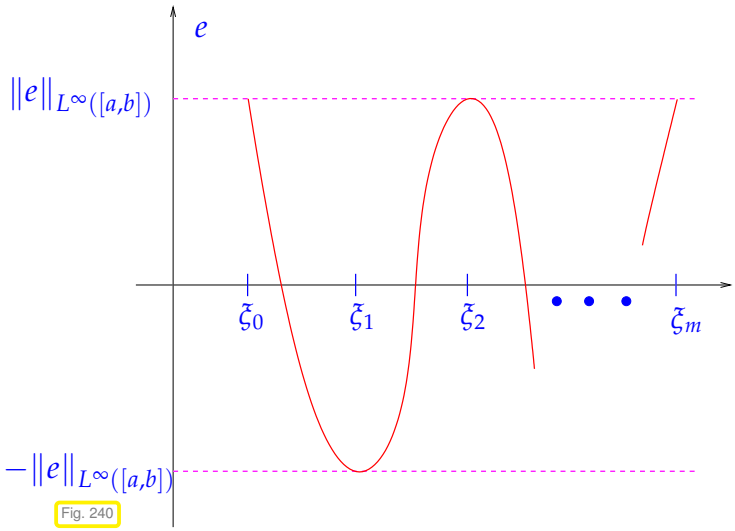


Fig. 240

(6.3.3) Remez algorithm

The widely used iterative algorithm (Remez algorithm) for finding an L^∞-best approximant is motivated by the alternation theorem. The idea is to determine successively better approximations of the set of alternants: $\mathcal{A}^{(0)} \to \mathcal{A}^{(1)} \to \ldots$, $\sharp\mathcal{A}^{(l)} = n+2$.

Key is the observation that, due to the alternation theorem, the polynomial $L^\infty([a,b])$-best approximant q will satisfy (one of the) interpolation conditions

$$q(\xi_k) \pm (-1)^k \delta = f(\xi_k), \quad k - 0, \ldots, n+1, \quad \delta := \|f - q\|_{L^\infty([a,b])}. \tag{6.3.4}$$

① Initial guess $\mathcal{A}^{(0)} := \{\xi_0^{(0)} < \xi_1^{(0)} < \cdots < \xi_n^{(0)} < \xi_{n+1}^{(0)}\} \subset [a,b]$ "arbitrary", for instance extremal points of the Chebychev polynomial T_{n+1}, → Def. 6.1.76, so so-called Chebychev alternants

$$\xi_j^{(0)} = \tfrac{1}{2}(a+b) + \tfrac{1}{2}(b-a)\cos\left(\frac{j}{n+1}\pi\right), \quad j = 0, \ldots, n+1. \tag{6.3.5}$$

② Given approximate alternants $\mathcal{A}^{(l)} := \{\xi_0^{(l)} < \xi_1^{(l)} < \cdots < \xi_n^{(l)} < \xi_{n+1}^{(l)}\} \subset [a,b]$ determine $q \in \mathcal{P}_n$ and a deviation $\delta \in \mathbb{R}$ satisfying the extended interpolation condition

$$q(\xi_k^{(l)}) + (-1)^k \delta = f(\xi_k^{(l)}), \quad k = 0, \ldots, n+1. \tag{6.3.6}$$

After choosing a basis for $\mathcal{P}_n$, this is $(n+2)\times(n+2)$ linear system of equations, *cf.* § 5.1.13.

③ Choose $\mathcal{A}^{(l+1)}$ as the set of extremal points of $f-q$, truncated in case more than $n+2$ of these exist.

These extreme can be located approximately by sampling on a fine grid covering $[a,b]$). If the derivative of $f \in C^1([a,b])$ is available, too, then search for zeros of $(f-p)'$ using the secant method from § 8.3.22.

④ If $\|f-q\|_{L^\infty([a,b])}| \leq \texttt{TOL}\cdot\|d\|_{L^\infty([a,b])}$ STOP, else GOTO ②.
(TOL is a prescribed relative tolerance.)

C++11 code 6.3.7: Remez algorithm for uniform polynomial approximation on an interval

```
function c = remes(f,f1,a,b,d,tol)
% f is a handle to the function, f1 to its derivative
% d = polynomial degree (positive integer)
% a,b = interval boundaries
% returns coefficients of polynomial in monomial basis
% (MATLAB convention, see Rem. 5.2.4).

n = 8*d;                    % n = number of sampling points
xtab=[a:(b-a)/(n-1):b]';    % Points of sampling grid
ftab = feval(f,xtab);       % Function values at sampling points
fsupn = max(abs(ftab));     % Approximate supremum norm of f
f1tab = feval(f1,xtab);     % Derivative values at sampling points
% The vector xe stores the current guess for the alternants; initial
% guess is Chebychev alternants (6.3.5).
h=pi/(d+1); xe=(a+b)/2 + (a-b)/2*cos(h*[0:d+1]');
fxe=feval(f,xe);

maxit = 10;
% Main iteration loop of Remez algorithm
for k=1:maxit
 % Interpolation at d+2 points xe with deviations ±δ
 % Algorithm uses monomial basis, which is not optimal
 V=vander(xe); A=[V(:,2:d+2), (-1).^[0:d+1]']; % LSE
 c=A\fxe;                   % Solve for coefficients of polynomial q
 c1=[d:-1:1]'.*c(1:d);      % Monomial doefficients of derivative q'

 % Find initial guesses for the inner extremes by sampling; track sign
 % changes of the derivative of the approximation error
 deltab = (polyval(c1,xtab) - f1tab);
 s=[deltab(1:n-1)].*[deltab(2:n)];
 ind=find(s<0); xx0=xtab(ind); % approximate zeros of e'
 nx = length(ind);             % number of approximate zeros
 % Too few extrema; bail out
 if (nx < d), error('Too few extrema'); end

 % Secant method to determine zeros of derivative
 % of approximation error
 F0 = polyval(c1,xx0) - feval(f1,xx0);
 % Initial guess from shifted sampling points
 xx1=xx0+(b-a)/(2*n);
 F1 = polyval(c1,xx1) - feval(f1,xx1);
```

```
% Main loop of secant method
while min(abs(F1)) > 1e-12,
  xx2=xx1-F1./(F1-F0).*(xx1-xx0);
  xx0=xx1; xx1=xx2; F0=F1;
  F1=polyval(c1,xx1) - feval(f1,xx1);
end;

% Determine new approximation for alternants; store in xe
% If too many zeros of the derivative (f-p)'
% have been found, select those, where the deviation is maximal.
if (nx == d)
  xe=[a;xx0;b];
elseif (nx == d+1)
  xmin = min(xx0); xmax = max(xx0);
  if ((xmin - a) > (b-xmax)), xe = [a;xx0];
  else xe = [xx0;b]; end
elseif (nx == d+2)
  xe = xx0;
else
  fx = feval(f,xx0);
  del = abs(polyval(c(1:d+1),xx0) - fx);
  [dummy,ind] = sort(del);
  xe = xx0(ind(end-d-1:end));
end

% Deviation in sampling points and approximate alternants
fxe=feval(f,xe);
del = [polyval(c(1:d+1),a) - ftab(1);...
    polyval(c(1:d+1),xe) - fxe;...
    polyval(c(1:d+1),b) - ftab(end)];
% Approximation of supremum norm of approximation error
dev = max(abs(del));
% Termination of Remez iteration
if ( dev < tol*fsupn), break; end
end;
c = c(1:d+1);
```

Experiment 6.3.8 (Convergence of Remez algorithm)

We examine the convergence of the Remez algorithm from Code 6.3.7 for two different functions:

- $f(t) = (1+t^2)^{-1}$, $I = [-5,5]$ → Bsp. 6.1.41
- $f(t) = \begin{cases} \frac{1}{2}(1+\cos(2\pi t)) & \text{, if } |t| < \frac{1}{2} \\ 0 & \text{else.} \end{cases}$, $I = [-1,1]$

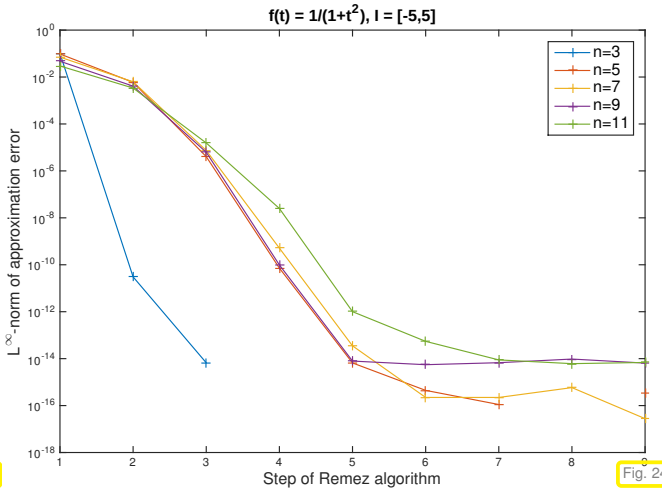


Fig. 241

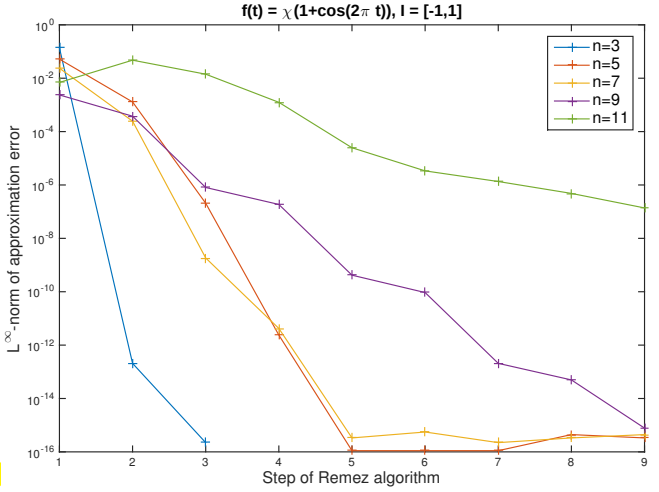


Fig. 242

Convergence in both cases; faster convergence observed for smooth function, for which machine precision is reached after a few steps.

6.4 Approximation by Trigonometric Polynomials

Task: approximation of a continuous 1-periodic function

$$f \in C^0(\mathbb{R}) \quad , \quad f(t+1) = f(t) \quad \forall t \in \mathbb{R} \, .$$

Dubious: approximation by global polynomials, because they are not 1-periodic
➢ the global polynomial Lagrange interpolant will lack an essential structural property.

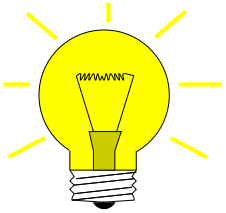
The natural space for approximating generic periodic functions is a space of trigonometric polynomials with the same period.

Remember from Def. 5.6.3: space of 1-periodic trigonometric polynomials of degree $2n$, $n \in \mathbb{N}$,

$$\begin{aligned} \mathcal{P}_{2n}^T &= \operatorname{Span}\Big\{ t \mapsto 1, t \mapsto \sin(2\pi t), t \mapsto \cos(2\pi t), t \mapsto \sin(4\pi t), t \mapsto \cos(4\pi t), \dots \\ &\qquad t \mapsto \sin(2\pi n t), t \mapsto \cos(2\pi n t) \Big\} \qquad (6.4.1a) \\ &= \operatorname{Span}\{ t \mapsto \exp(2\pi \imath k t) : k = -n \dots, n \} \, . \qquad (6.4.1b) \end{aligned}$$

Both sets of functions provide a basis of $\mathcal{P}_{2n}^T$ (when considered as a space of $\mathbb{C}$-valued functions on $\mathbb{R}$).

6.4.1 Approximation by Trigonometric Interpolation



Idea: Adapt the policy of approximation by interpolation from § 6.0.6

Now: Employ trigonometric interpolation from Section 5.6
into space $\mathcal{P}_{2n}^T$ of 1-periodic trigonometric polynomials → Def. 5.6.3.

Recall: Trigonometric interpolation → Section 5.6

Given nodes $t_0 < t_1 < \cdots < t_{2n}$, $t_k \in [0,1[$, and values $y_k \in \mathbb{R}$, $k = 0,\ldots,2n$ find

$$q \in \mathcal{P}_{2n}^T := \operatorname{Span}\{t \mapsto \cos(2\pi jt), t \mapsto \sin(2\pi jt)\}_{j=0}^{n}, \tag{6.4.3}$$

$$\text{with} \quad q(t_k) = y_k \quad \text{for all} \quad k = 0,\ldots,2n. \tag{6.4.4}$$

Terminology: $\mathcal{P}_{2n}^T \hat{=}$ space of trigonometric polynomials of degree $2n$.

From Section 5.6 remember a few more facts about trigonometric polynomials and trigonometric interpolation:

- Cor. 5.6.8: Dimension of the space of trigonometric polynomials: $\dim \mathcal{P}_{2n}^T = 2n+1$
- Trigonometric interpolation can be reduced to polynomial interpolation on the unit circle $S^1 \subset \mathbb{C}$ in the complex plane, see (5.6.7).
  ⟸ existence & uniqueness of trigonometric interpolant q satisfying (6.4.3) and (6.4.4)
- Code 5.6.15: Efficient FFT-based algorithms for trigonometric interpolation in equidistant nodes $t_k = \frac{k}{2n+1}$, $k = 0,\ldots,2n$.

The relationship of trigonometric interpolation and polynomial interpolation on the unit circle suggests a uniform distribution of nodes for general trigonometric interpolation.

Nodes for function approximation by trigonometric interpolation

Trigonometric approximation of generic *1-periodic* continuous functions $\in C^0(\mathbb{R})$ in $\mathcal{P}_{2n}^T$ usually relies on equidistant interpolation nodes $t_k = \frac{k}{2n+1}$, $k = 0,\ldots,2n$.

✎ notation: trigonometric interpolation operator in $2n+1$ equidistant nodes $t_k = \frac{k}{2n+1}$, $k = 0,\ldots,2n$

$$\mathsf{T}_n : C^0([0,1[) \to \mathcal{P}_{2n}^T \quad , \quad \mathsf{T}_n(f)(t_k) = f(t_k) \quad \forall k \in \{0,\ldots,2n\}. \tag{6.4.6}$$

Note that a function $f \in C^0([0,1[)$ can spawn a discontinuous 1-periodic function on $\mathbb{R}$

A prominent example is the "sawtooth function" ▷

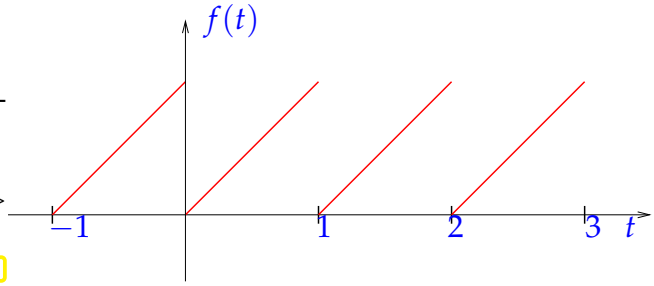


Fig. 243

6.4.2 Trigonometric interpolation error estimates

Our focus will be on the asymptotic behavior of

$$\|f - \mathsf{T}_n f\|_{L^\infty([0,1[)} \quad \text{and} \quad \|f - \mathsf{T}_n f\|_{L^2([0,1[)} \quad \text{as} \quad n \to \infty,$$

for functions $f : [0,1[\to \mathbb{C}$ with different smoothness properties. To begin with we perform an empiric study.

Experiment 6.4.7 (Interpolation error: trigonometric interpolation)

Now we study the asymptotic behavior of the error of equidistant trigonometric interpolation as $n \to \infty$ in a numerical experiment for functions with different smoothness properties.

#1 Step function: $f(t) = 0$ for $|t - \frac{1}{2}| > \frac{1}{4}$, $f(t) = 1$ for $|t - \frac{1}{2}| \leq \frac{1}{4}$

#2 C^∞ periodic function: $f(t) = \dfrac{1}{\sqrt{1 + \frac{1}{2}\sin(2\pi t)}}$.

#3 "wedge function": $f(t) = |t - \frac{1}{2}|$

Approximate computation of norms of interpolation errors on equidistant grid with 4096 points.

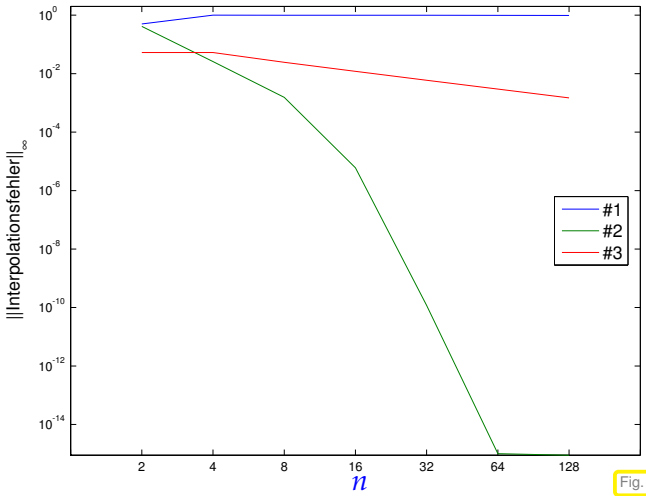


Fig. 244

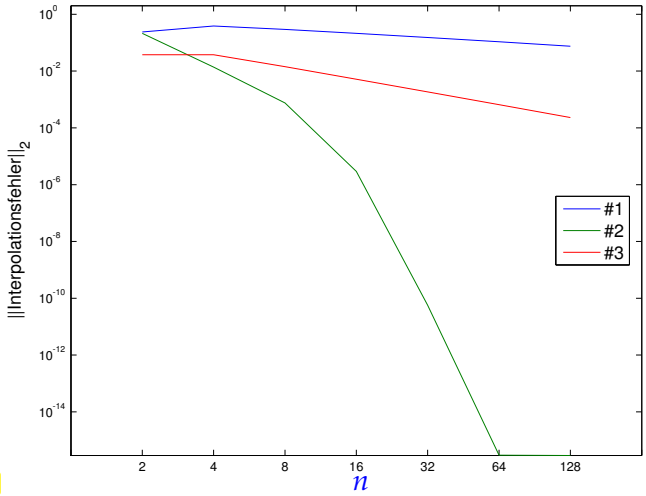


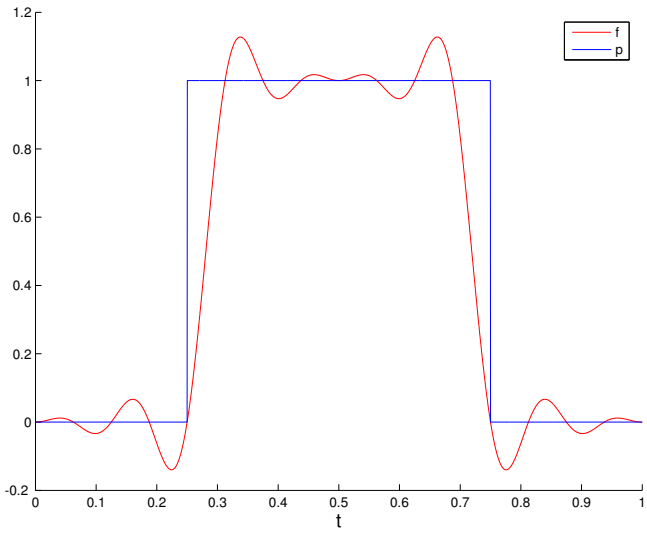
Fig. 245

Observation: Function #1: no convergence in L^∞-norm, algebraic convergence in L^2-norm
Function #3: algebraic convergence in both norms
Function #2: exponential convergence in both norms

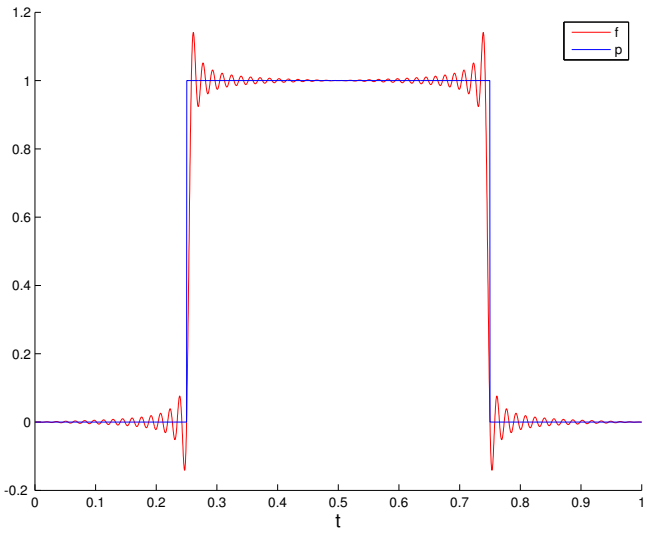
We conclude that in this experiment higher smoothness of f leads to faster convergence of the trigonometric interplant.

Experiment 6.4.8 (Gibbs phenomenon)

Of course the smooth trigonometric interpolants of step function fail to converge in L^∞-norm in Exp. 6.4.7. Moreover, they will not even converge "visually" to the step function, which becomes manifest by a closer inspection of the interpolants.



$n = 16$



$n = 128$

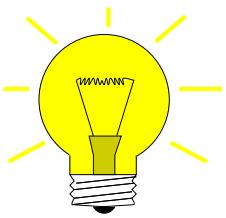
Observation: overshooting in neighborhood of discontinuity: Gibbs phenomenon

(6.4.9) Fourier series → (4.2.67)

From (6.4.1b) we learn that the space of trigonometric polynomials $\mathcal{P}_{2n}^T$ is spanned by the $2n+1$ Fourier modes $t \mapsto \exp(2\pi\imath kt)$ of "lowest frequency", that is, for $-n \le k \le n$. In Section 4.2.5 we learned that every function $f : [0,1[\to \mathbb{C}$ with finite $L^2([0,1])$-norm can be expanded in a Fourier series (→ Thm. 4.2.89)

$$f(t) = \sum_{k\in\mathbb{Z}} \widehat{f}_k \exp(-2\pi\imath kt) \quad \text{in } L^2([0,1]) \quad , \quad \widehat{f}_k := \int_0^1 f(t)\exp(2\pi\imath kt)\,\mathrm{d}t \; .$$

(A limit in $L^2([0,1])$ means that $\left\| f - \sum_{k=-M}^{M} \widehat{f}_k \exp(-2\pi\imath k\cdot) \right\|_{L^2([0,1])} \to 0$ for $M \to \infty$. Also remember the notation $\widehat{f}_k$ for the Fourier coefficients.)



Idea: Study trigonometric interpolation error for interpolands in Fourier series representation.

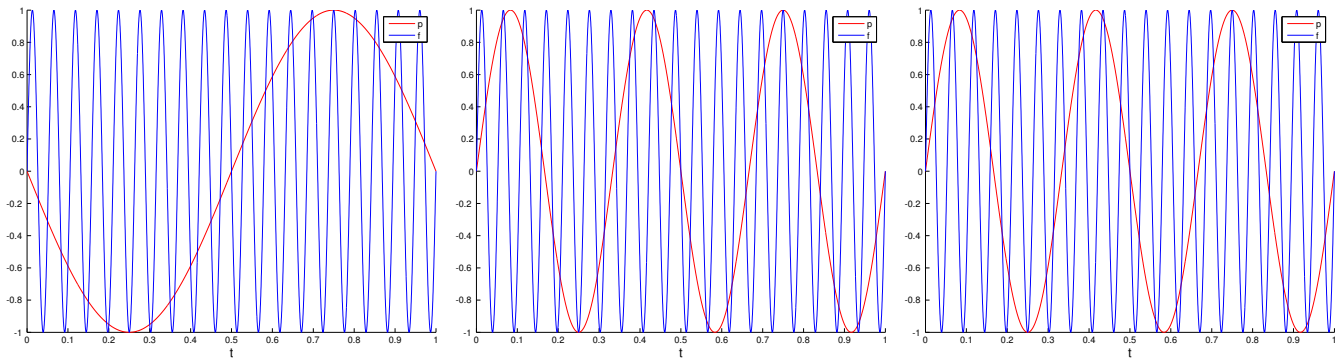
(6.4.10) Aliasing

We study the action of the trigonometric interpolation operator T_n from (6.4.6) on individual Fourier modes $\mu_k(t) := \exp(-2\pi k\imath t)$, $t \in \mathbb{R}$, $k \in \mathbb{Z}$. Due to the 1-periodicity of $t \mapsto \exp(2\pi\imath t)$ we find for every node $t_j := \frac{j}{2n+1}$, $j = 0,\ldots,2n$,

$$\mu_k(t_j) = \exp(-2\pi\imath k\tfrac{j}{2n+1}) = \exp(-2\pi\imath(k-\ell(2n+1))\tfrac{j}{2n+1}) = \mu_{k-\ell(2n+1)}(t_j) \quad \forall \ell \in \mathbb{Z} \; .$$

When sampled on the node set $\mathcal{T}_n := \{t_0,\ldots,t_{2n}\}$ all the Fourier modes $\mu_{k-\ell(2n+1)}$, $\ell \in \mathbb{Z}$, yield the same values. Thus the trigonometric cannot distinguish them! This phenomenon is called aliasing.

Aliasing demonstrated for $f(t) = \sin(2\pi\cdot 19t) = \mathrm{Im}(\exp(2\pi\imath 19t))$ for different node sets.



$n = 2$ $n = 4$ $n = 8$

The "low-frequency" sine waves plotted in red coincide with f on the node set $\mathcal{T}_n$.

Since $\mathsf{T}_n \mu_k = \mu_k$ for $k = -n, \ldots, n$, that is, for $\mu_k \in \mathcal{P}_{2n}^T$, the aliasing effect yields

$$\mathsf{T}_n \mu_k = \mu_{\widetilde{k}} \quad , \quad \widetilde{k} \in \{-n, \ldots, n\} \, , \quad k - \widetilde{k} \in (2n+1)\mathbb{Z} \quad [\, \widetilde{k} := k \mod (2n+1) \,] \, . \tag{6.4.11}$$

($\widetilde{n} = n$, $\widetilde{n+1} = -n$, $\widetilde{-n} = -n$, $\widetilde{-n-1} = n$, $\widetilde{2n} = -1$, etc.)

▬► Trigonometric interpolation by T_n folds all Fourier modes ("frequencies") to the finite range $\{-n, \ldots, n\}$.

From (6.4.11), by linearity of T_n, we obtain for $f : [0,1[\to \mathbb{C}$ in Fourier series representation

$$f(t) = \sum_{j=-\infty}^{\infty} \widehat{f}_j \mu_j(t) \quad ▬► \quad \mathsf{T}_n(f)(t) = \sum_{j=-n}^{n} \gamma_j \mu_j(t) \, , \quad \gamma_j = \sum_{\ell=-\infty}^{\ell} \widehat{f}_{j+\ell(2n+1)} \, . \tag{6.4.12}$$

We can read the trigonometric polynomial $\mathsf{T}_n f \in \mathcal{P}_{2n}^T$ as a Fourier series with non-zero coefficients only in the index range $\{-n, \ldots, n\}$. Thus, for the Fourier coefficients of the trigonometric interpolation error $E(t) := f(t) - \mathsf{T}_n f(t)$ we find from (6.4.12)

$$\widehat{E}_j = \begin{cases} -\sum\limits_{\ell \in \mathbb{Z} \setminus \{0\}} \widehat{f}_{j+\ell(2n+1)} & \text{, if } j \in \{-n, \ldots, n\} \, , \\ \widehat{f}_j & \text{, if } |j| > n \, , \end{cases} \quad j \in \mathbb{Z} \, . \tag{6.4.13}$$

Since we have (sufficient smoothness of f assumed)

$$E(t) := f(t) - \mathsf{T}_n f(t) = \sum_{j=-\infty}^{\infty} \widehat{E}_j e^{-2\pi \imath j t} \, , \tag{6.4.14}$$

we conclude, in particular from the isometry property asserted in Thm. 4.2.89,

$$\|f - \mathsf{T}_n f\|_{L^2(]0,1[)}^2 = \sum_{j=-n}^{n} \left| \sum_{\ell \in \mathbb{Z} \setminus \{0\}} \widehat{f}_{j+\ell(2n+1)} \right|^2 + \sum_{|j|>n} |\widehat{f}_j|^2 \, , \tag{6.4.15}$$

$$\|f - \mathsf{T}_n f\|_{L^\infty(]0,1[)} \leq \sum_{j=-n}^{n} \sum_{\ell \in \mathbb{Z} \setminus \{0\}} |\widehat{f}_{j+\ell(2n+1)}| + \sum_{|j|>n} |\widehat{f}_j| \, . \tag{6.4.16}$$

In order to estimate these norms of the trigonometric interpolation error we need quantitative information about the decay of the Fourier coefficients $\widehat{f}_j$ as $|j| \to \infty$.

(6.4.17) Fourier expansions of derivatives

For 1-periodic $c \in C^0(\mathbb{R})$ with integrable derivative $\dot{c} := \frac{d}{dt}c$ we find by integration by parts (boundary terms cancel due to periodicity)

$$\widehat{c}_j = \int_0^1 c(t)e^{2\pi\imath jt}\,\mathrm{d}t = -2\pi\imath j \cdot \int_0^1 \dot{c}(t)e^{2\pi\imath jt}\,\mathrm{d}t = (-2\pi\imath j)\,\widehat{\dot{c}}_j\,, \quad j \in \mathbb{Z}\,.$$

We can also arrive at this formula by (formal) term-wise differentiation of the Fourier series:

$$c(t) = \sum_{j=-\infty}^{\infty} \widehat{c}_j e^{-2\pi\imath jt} \quad \Longrightarrow \quad \dot{c}(t) = \sum_{j=-\infty}^{\infty} \underbrace{(-2\pi\imath j)\widehat{c}_j}_{=\widehat{\dot{c}}_j} e^{-2\pi\imath jt}\,. \tag{6.4.18}$$

Lemma 6.4.19. Fourier coefficients of derivatives

For the Fourier coefficients of the derivatives a 1-periodic function $f \in C^{k-1}(\mathbb{R})$, $k \in \mathbb{N}$, with integrable k-th derivative $f^{(k)}$ holds

$$\widehat{f^{(k)}}_j = (-2\pi\imath)^k\,\widehat{f}_j\,, \quad j \in \mathbb{Z}\,.$$

(6.4.20) Fourier coefficients and smoothness

From Lemma 6.4.19 and the trivial estimates ($|\exp(2\pi\imath t)| = 1$)

$$|\widehat{f}_j| \le \int_0^1 |f(t)|\,\mathrm{d}t \le \|f\|_{L^\infty(]0,1[)} \quad \forall j \in \mathbb{Z}\,, \tag{6.4.21}$$

we conclude that $\left((2\pi|j|)^m \widehat{f}_j\right)_{j\in\mathbb{Z}}$, $m \in \mathbb{N}$, is bounded, provided that $f \in C^{m-1}(\mathbb{R})$ with integrable m-th derivative

Lemma 6.4.22. Decay of Fourier coefficients

If $f \in C^{k-1}(\mathbb{R})$ with integrable k-th derivative, then $\widehat{f}_j = O(j^{-k})$ for $|j| \to \infty$

Decay of Fourier coefficients and smoothness

The smoother a periodic function the faster the decay of its Fourier coefficients

The isometry property of Thm. 4.2.89 also yields for $f \in C^{k-1}(\mathbb{R})$ with $f^{(k)} \in L^2(unitintv)$ that

$$\left\|f^{(k)}\right\|^2_{L^2(]0,1[)} = (2\pi)^k \sum_{j=-\infty}^{\infty} j^{2k}|\widehat{f}_j|^2\,. \tag{6.4.24}$$

We can now combine the identity (6.4.24) with (6.4.15) and obtain an interpolation error estimate in $L^2(]0,1[)$-norm.

Theorem 6.4.25. L^2-error estimate for trigonometric interpolation

If $f \in C^{k-1}(\mathbb{R})$, $k \in \mathbb{N}$, with square-integrable k-th derivative ($f^{(k)} \in L^2(]0,1[)$), then

$$\|f - \mathsf{T}_n f\|_{L^2(]0,1[)} \leq \sqrt{1+c_k}\, n^{-k} \left\| f^{(k)} \right\|_{L^2(]0,1[)} , \tag{6.4.26}$$

with $c_k = 2\sum_{\ell=1}^{\infty} (2\ell-1)^{-2k} < \infty$.

Thm. 6.4.25 confirms *algebraic convergence* of the L^2-norm of the trigonometric interpolation error for functions with *limited smoothness*. Higher rates can be expected for smoother functions, which we have also found in cases #1 and #3 in Exp. 6.4.7.

6.4.3 Trigonometric Interpolation of Analytic Periodic Functions

In § 6.1.62 we saw that we can expect exponential decay of the maximum norm of polynomial interpolation errors in the case of "very smooth" interpolands. To capture this property of functions we resorted to the notion of analytic functions, as defined in Def. 6.1.63. Since trigonometric interpolation is closely connected to polynomial interpolation (on the unit circle $\mathbb{S}^1$, see Section 5.6.2), it is not surprising that analyticity of interpolands will also involve exponential convergence of trigonometric interpolants. This result will be established in this section.

In case #2 of Exp. 6.4.7 we already say an instance of exponential convergence for an analytic interpoland. A more detailed study follows.

Experiment 6.4.27 (Trigonometric interpolation of analytic functions)

We study the convergence of equidistant trigonometric interpolation for the interpoland

$$f(t) = \frac{1}{\sqrt{1-\alpha\sin(2\pi t)}} \quad \text{on } I = [0,1] . \tag{6.4.28}$$

For $0 \leq \alpha < 1$ we have $f \in C^\infty(\mathbb{R})$, and f is even analytic ($\rightarrow$ Def. 6.1.63).

Approximative computations of error norms by "oversampling" in 4096 points. ▷

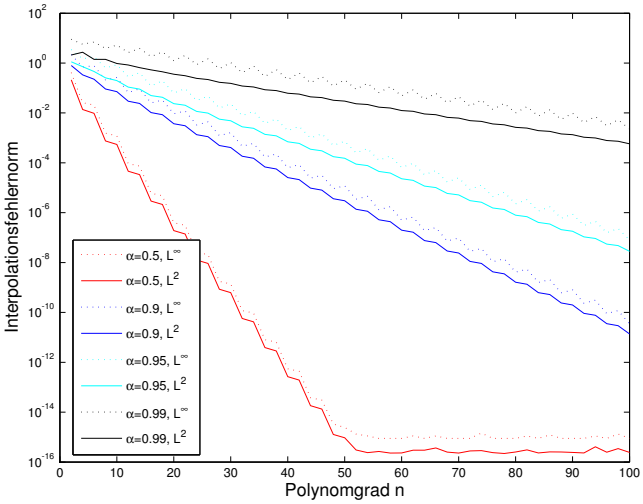


Fig. 246

➢ Observation: exponential convergence in n, faster for smaller α

(6.4.29) Decay of Fourier coefficients of analytic functions

Lemma 6.4.22 asserts algebraic decay of the Fourier coefficients of functions with limited smoothness. As analytic 1-periodic functions are "infinitely smooth", the will always belong to $C^\infty(\mathbb{R})$, we expect a stronger result in this case. In fact, we can conclude *exponential decay* of the Fourier coefficients.

Theorem 6.4.30. Exponential decay of Fourier coefficients of analytic functions

If $f : \mathbb{R} \to \mathbb{C}$ is 1-periodic and has an analytic extension to the strip

$$\bar{S} := \{z \in \mathbb{C}: \ -\eta \le \operatorname{Im} z \le \eta\} \ , \quad \text{for some } \eta > 0 \ ,$$

then its Fourier coefficients decay according to

$$|\widehat{f}_j| \le q^k \cdot \|f\|_{L^\infty(\bar{S})} \quad \text{with} \quad q := \exp(-2\pi\eta) \in]0,1[\ . \tag{6.4.31}$$

Proof.

Let $f : \mathbb{R} \mapsto \mathbb{C}$ be 1-periodic with analytic extension to the (closed) strip $S := \{z \in \mathbb{C}: \ -\eta \le \operatorname{Im}\{z\} \le \eta\}$, $\eta > 0$.

➢ $g_r(t) := f(t + \imath r)$ is 1-periodic and smooth for $-\eta \le r \le \eta$

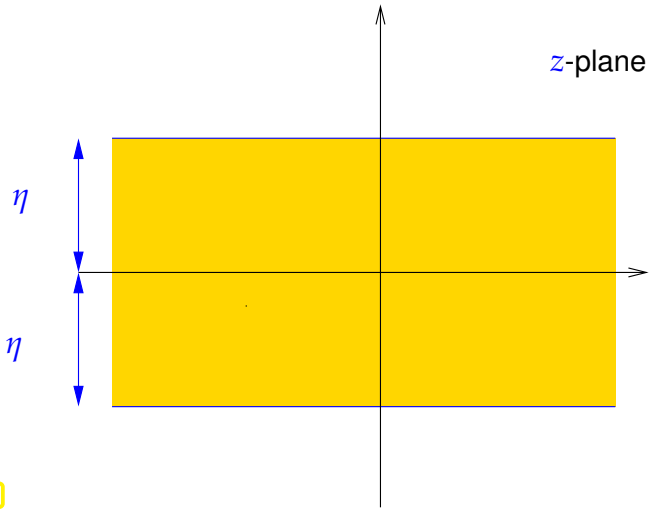


Fig. 247

We compute the Fourier coefficients of g_r

$$\widehat{g_r}_k = \int_0^1 f(t + ir)e^{-2\pi\imath kt}\,\mathrm{d}t = \int_0^1 f(t)e^{2\pi\imath k(t-ir)}\,\mathrm{d}t = e^{-2\pi rk}\widehat{f}_k \ ,$$

and arrive at the bound

$$|\widehat{f}_k| \le e^{-2\pi rk} \underbrace{\max\{|f(t + \ iur)|, 0 \le t \le 1\}}_{\|g_r\|_{L^\infty(]0,1[)}} \quad \forall k \in \mathbb{Z} \ , \quad \forall -\eta^- \le r \le \eta^+ \ .$$

The assertion of the theorem follows directly.

□

Knowing exponential decay of the Fourier coefficients, the geometric sum formula can be used to extract estimates from (6.4.15) and (6.4.16):

Lemma 6.4.32. Interpolation error estimates for exponentially decaying Fourier coefficients

If $f : \mathbb{R} \to \mathbb{C}$ is 1-periodic and $f \in L^2(]0,1[)$, then

$$\exists M > 0, \rho \in]0,1[: \quad |\widehat{f}(k)| \leq M\rho^{|k|} \quad \Rightarrow \quad \begin{aligned} \|f - \mathsf{T}_n(f)\|_{L^2(]0,1[)} &\leq M\frac{\rho^{n/2}}{1-\rho^n}\frac{2\sqrt{2}}{\sqrt{1-\rho^2}}\,, \\ \|f - \mathsf{T}_n(f)\|_{L^\infty(]0,1[)} &\leq 4M\frac{\rho^{n/2}}{1-\rho}\,. \end{aligned}$$

This estimate can be combined with the result of Thm. 6.4.30 and gives the main result of this section:

Theorem 6.4.33. Exponential convergence of trigonometric interpolation for analytic interpolands

If $f : \mathbb{R} \to \mathbb{C}$ is 1-periodic and possesses an analytic extension to the strip

$$\bar{S} := \{z \in \mathbb{C}: \ -\eta \leq \operatorname{Im} z \leq \eta\}\,, \quad \text{for some } \eta > 0\,,$$

then there is $C_\eta > 0$ depending only on η such that

$$\|f - \mathsf{T}_n f\|_* \leq C_\eta e^{-\pi\eta n}\|f\|_{L^\infty(\bar{S})}\,, \quad n \in \mathbb{N}\,, \quad (\,* = L^2(]0,1[), L^\infty(]0,1[)\,)\,. \tag{6.4.34}$$

The speed of exponential convergence clearly depends on the width η of the "strip of analyticity" $\bar{S}$.

(6.4.35) Convergence of trigonometric interpolation for analytic interpolands

Explanation of the observation made in Exp. 6.4.27, *cf.* Rem. 6.1.96:

Similar to Chebychev interpolants, also trigonometric interpolants converge exponentially fast, if the interpoland f is 1-periodic *analytic* ($\to$ Def. 6.1.63) in a strip around the real axis in $\mathbb{C}$, see Thm. 6.4.33 for details.

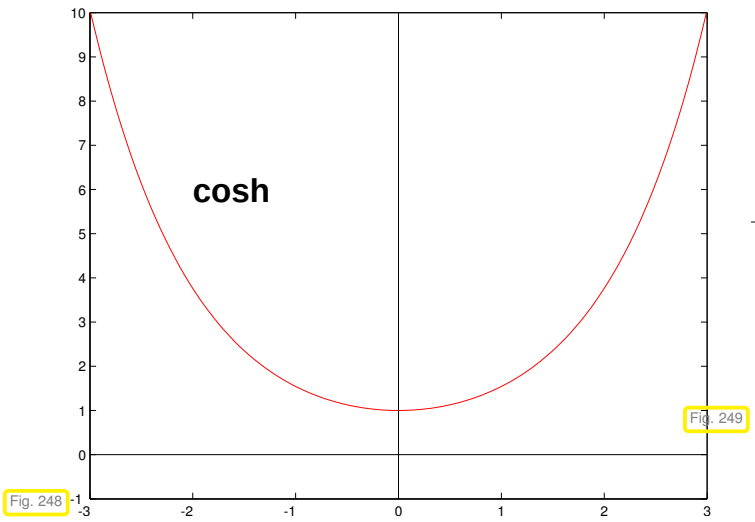
Note that f from (6.4.28) is holomorphic, where

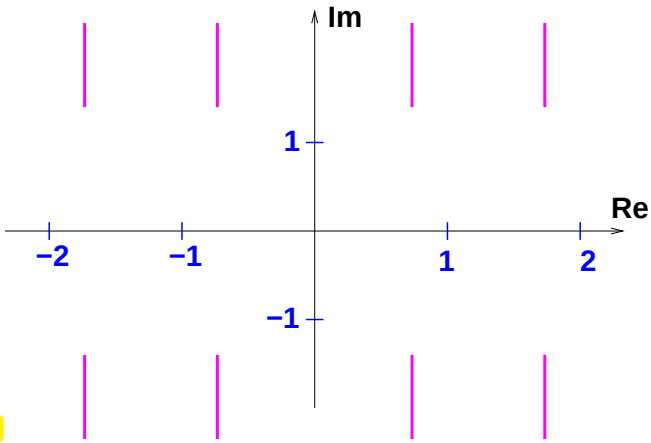
$$1 + \alpha\sin(2\pi z) \notin \mathbb{R}_0^- \Leftrightarrow \sin(2\pi z) = \sin(2\pi x)\cosh(2\pi y) + i\cos(2\pi x)\sinh(2\pi y) \notin]-\infty, -1 - \tfrac{1}{\alpha}]\,,$$

because the square root is holomorphic in $\mathbb{C} \setminus \mathbb{R}_0^-$.

Domain of analyticity of f:

$$\mathbb{C} \setminus \bigcup_{k\in\mathbb{Z}} (\tfrac{k}{2} + \tfrac{1}{4} + i(\mathbb{R}\setminus]-\zeta,\zeta[))\,, \quad \zeta \in \mathbb{R}^+\,, \ \cosh(2\pi\zeta) = 1 + \tfrac{1}{\alpha}\,.$$





➢ f analytic in strip $S := \{z \in \mathbb{C}: : -\zeta < \mathrm{Im}(z) < \zeta\}$.

➢ As α decreases the strip of analyticity becomes wider, since $x \to \cosh(x)$ is increasing for $x > 0$.

6.5 Approximation by piecewise polynomials

Recall some alternatives to interpolation by global polynomials discussed in Chapter 5:

- piecewise linear/quadratic interpolation → Section 5.3.2, Ex. 5.3.11,
- cubic Hermite interpolation → Section 5.4,
- (cubic) spline interpolation → Section 5.5.1.

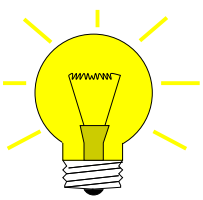
☞ All these interpolation schemes rely on piecewise polynomials (of different global smoothness)

Focus in this section: function approximation by *piecewise polynomial* interpolants

(6.5.1) Grid/mesh

The attribute "piecewise" refers to *partitioning of the interval* on which we aim to approximate. In the case of data interpolation the natural choice was to use intervales defined by interpolation nodes. Yet we already saw exceptions in the case of shape-preserving interpolation by means of quadratic splines, see Section 5.5.3.

In the case of function approximation based on an interpolation scheme the additional freedom to choose the interpolation nodes suggests that those be decoupled from the partitioning.



Idea: use piecewise polynomials with respect to a grid/mesh

$$\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\} \tag{6.5.2}$$

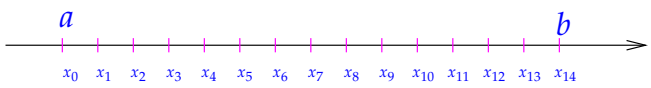
to approximate function $f : [a,b] \mapsto \mathbb{R}$, $a < b$.

Borrowing from terminology for splines, *cf.* Def. 5.5.1, the underlying mesh for piecewise polynomial

approximation is sometimes called the "knot set".

Terminology:

- $x_j \hat{=}$ nodes of the mesh $\mathcal{M}$,
- $[x_{j-1}, x_j[\hat{=}$ intervals/cells of the mesh,
- $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}| \hat{=}$ mesh width,
- If $x_j = a + jh \hat{=}$ equidistant (uniform) mesh with meshwidth $h > 0$



Remark 6.5.3 (Local approximation by piecewise polynomials)

We will see that most approximation schemes relying on piecewise polynomials are local in the sense that finding the approximant on a cell of the mesh relies only on a fixed number of function evaluations in a neighborhood of the cell.

➢ $O(1)$ computational effort to find interpolant on $[x_{j-1}, x_j]$ (independent of m)

➢ $O(m)$ computational effort to determine piecewise polynomial approximant for $m \to \infty$ ("fine meshes")

Contrast this with the computational cost of computing global polynomial interpolants, which will usually be $O(n^2)$ for polynomial degree $n \to \infty$.

6.5.1 Piecewise polynomial Lagrange interpolation

Given: interval $[a, b] \subset \mathbb{R}$ endowed with mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$.

Recall theory of polynomial interpolation → Section 5.2.2: $n+1$ data points needed to fix interpolating polynomial, see Thm. 5.2.14.

Approach to *local* Lagrange interpolation (→ (5.2.9)) of $f \in C(I)$ on mesh $\mathcal{M}$

General local Lagrange interpolation on a mesh

❶ Choose local degree $n_j \in \mathbb{N}_0$ for each cell of the mesh, $j = 1, \ldots, m$.

❷ Choose set of *local* interpolation nodes

$$\mathcal{T}^j := \{t_0^j, \ldots, t_{n_j}^j\} \subset I_j := [x_{j-1}, x_j] \,, \quad j = 1, \ldots, m \,,$$

for each mesh cell/grid interval I_j.

❸ Define piecewise polynomial interpolant $s : [x_0, x_m] \to \mathbb{K}$:

$$s_j := s_{|I_j} \in \mathcal{P}_{n_j} \quad \text{and} \quad s_j(t_i^j) = f(t_i^j) \quad i = 0, \ldots, n_j \,, \quad j = 1, \ldots, m \,. \tag{6.5.5}$$

Owing to Thm. 5.2.14, s_j is well defined.

Corollary 6.5.6. Piecewise polynomials Lagrange interpolation operator

The mapping $f \mapsto s$ defines a linear operator $\mathsf{I}_{\mathcal{M}} : C^0([a,b]) \mapsto C^0_{\mathcal{M},\mathrm{pw}}([a,b])$ in the sense of Def. 5.1.17.

Obviously, $\mathsf{I}_{\mathcal{M}}$ depends on $\mathcal{M}$, the local degrees n_j, and the sets $\mathcal{T}_j$ of local interpolation points (the latter two are suppressed in notation).

Corollary 6.5.7. Continuous local Lagrange interpolants

If the local degrees n_j are at least 1 and the local interpolation nodes t^j_k, $j = 1,\ldots,m$, $k = 0,\ldots,n_j$, for local Lagrange interpolation satisfy

$$t^j_{n_j} = t^{j+1}_0 \quad \forall j = 1,\ldots,m-1 \quad \Rightarrow \quad s \in C^0([a,b]) \,, \tag{6.5.8}$$

then the piecewise polynomial Lagrange interpolant according to (6.5.5) *is continuous on $[a,b]$: $s \in C^0([a,b])$.*

Focus: asymptotic behavior of (some norm of) interpolation error

$$\|f - \mathsf{I}_{\mathcal{M}} f\| \leq CT(N) \quad \text{for} \quad N \to \infty \,, \tag{6.5.9}$$

where $N := \sum_{j=1}^{m} (n_j + 1)$.

The decay of the bound $T(N)$ will characterize the type of convergence:

☛ algebraic convergence or exponential convergence, see Section 6.1.2, Def. 6.1.38.

But why do we choose this strange number N as parameter when investigating the approximation error?

Because, by Thm. 5.2.2, it agrees with the dimension of the space of discontinuous, piecewise polynomials functions

$$\{q : [a,b] \to \mathbb{R} \colon q_{|I_j} \in \mathcal{P}_{n_j} \; \forall j = 1,\ldots,m\} \quad !$$

This dimension tells us the number of real parameters we need to describe the interpolant s, that is, the "information cost" of s. N is also proportional to the number of interpolation conditions, which agrees with the number of f-evaluations needed to compute s (why only proportional in general?).

Special case: uniform polynomial degree $n_j = n$ for all $j = 1,\ldots,m$.

Then we may aim for estimates $\|f - \mathsf{I}_{\mathcal{M}} f\| \leq CT(h_{\mathcal{M}})$ for $h_{\mathcal{M}} \to 0$

in terms of meshwidth $h_{\mathcal{M}}$.

Terminology: investigations of this kind are called the study of h-convergence.

Example 6.5.10 (h-convergence of piecewise polynomial interpolation)

Compare Exp. 5.3.7:

$f(t) = \arctan t$, $I = [-5,5]$

Grid $\mathcal{M} := \{-5, -\frac{5}{2}, 0, \frac{5}{2}, 5\}$

Local interpolation nodes equidistant in I_j, endpoints included, (6.5.8) satisfied.

Plots of the piecewise linear, quadratic and cubic polynomial interpolants ▷

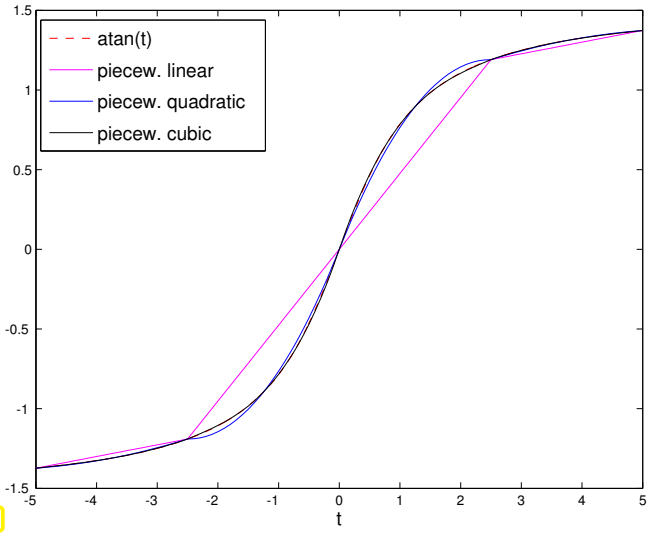


Fig. 250

- ✦ *Sequence* of (equidistant) meshes: $\mathcal{M}_i := \{-5 + j2^{-i}10\}_{j=0}^{2^i}$, $i = 1,\ldots,6$.
- ✦ *Equidistant* local interpolation nodes (endpoints of grid intervals included).

Monitored: interpolation error in (approximate) L^∞- and L^2-norms, see (6.1.95), (6.1.94)

$$\|g\|_{L^\infty([-5,5])} \approx \max_{j=0,\ldots,1000} |g(-5 + j/100)| ,$$

$$\|g\|_{L^2([-5,5])} \approx \sqrt{\tfrac{1}{1000}} \cdot \left(\tfrac{1}{2} g(-5)^2 + \sum_{j=1}^{999} |g(-5 + j/100)|^2 + \tfrac{1}{2} g(5)^2 \right)^{1/2} .$$

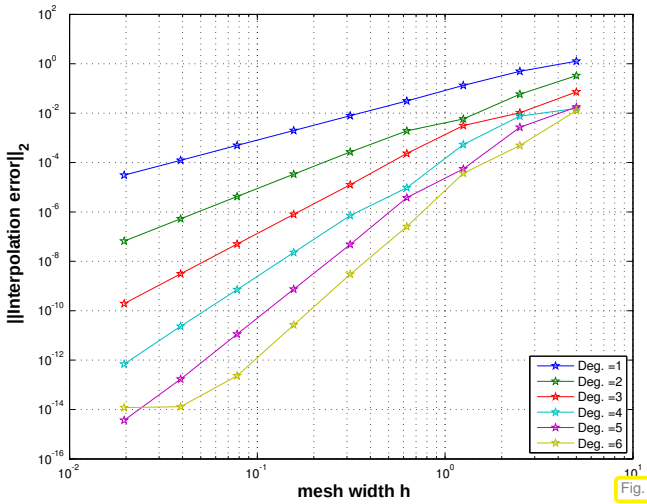


Fig. 251

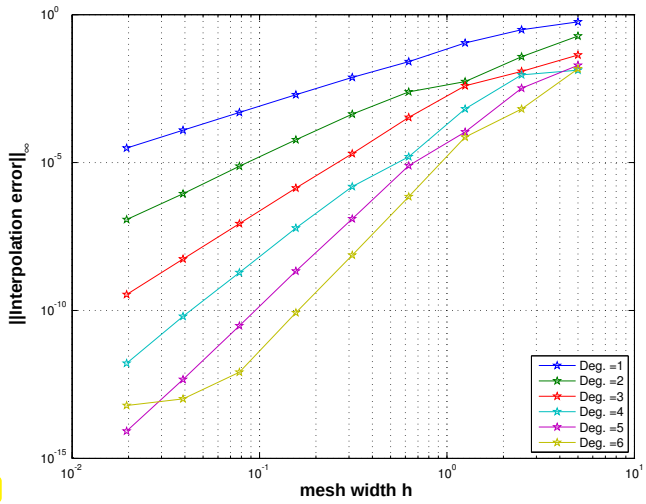


Fig. 252

Observation: Algebraic convergence (→ Def. 6.1.38) for meshwidth $h_\mathcal{M} \to 0$

(nearly linear error norm graphs in doubly logarithmic scale, see Rem. 6.1.40)

Observation: rate of algebraic convergence increases with polynomial degree n

Rates α of algebraic convergence $O(h_\mathcal{M}^\alpha)$ of norms of interpolation error:

n	1	2	3	4	5	6
w.r.t. L^2-norm	1.9957	2.9747	4.0256	4.8070	6.0013	5.2012
w.r.t. L^∞-norm	1.9529	2.8989	3.9712	4.7057	5.9801	4.9228

➢ Higher polynomial degree provides faster algebraic decrease of interpolation error norms. Empiric evidence for rates $\alpha = p + 1$

Here: rates estimated by linear regression (→ Ex. 3.1.5) based on MATLAB's `polyfit` and the interpolation errors for meshwidth $h \leq 10 \cdot 2^{-5}$. This was done in order to avoid erratic "preasymptotic", that is, for large meshwidth h, behavior of the error.

The bad rates for $n = 6$ are probably due to the impact of roundoff, because the norms of the interpolation error had dropped below machine precision, see Fig. 251, 252.

(6.5.11) Approximation error estimates for piecewise polynomial Lagrange interpolation

The observations made in Ex. 6.5.10 are easily explained by applying the polynomial interpolation error estimates of Section 6.1.2 *locally* on the mesh intervals $[x_{j-1}, x_j]$, $j = 1, \ldots, m$: for constant polynomial degree $n = n_j$, $j = 1, \ldots, m$, we get

$$\text{(6.1.50)} \quad \Rightarrow \quad \|f - s\|_{L^\infty([x_0,x_m])} \leq \frac{h_{\mathcal{M}}^{n+1}}{(n+1)!} \left\| f^{(n+1)} \right\|_{L^\infty([x_0,x_m])} , \tag{6.5.12}$$

with mesh width $h_{\mathcal{M}} := \max\{|x_j - x_{j-1}| : j = 1, \ldots, m\}$.

Another special case: fixed mesh $\mathcal{M}$, uniform polynomial degree n

Study estimates $\|f - \mathrm{I}_{\mathcal{M}} f\| \leq CT(n)$ for $n \to \infty$.

Terminology: investigation of p-convergence

Example 6.5.13 (p-convergence of piecewise polynomial interpolation)

We study p-convergence in the setting of Ex. 6.5.10.

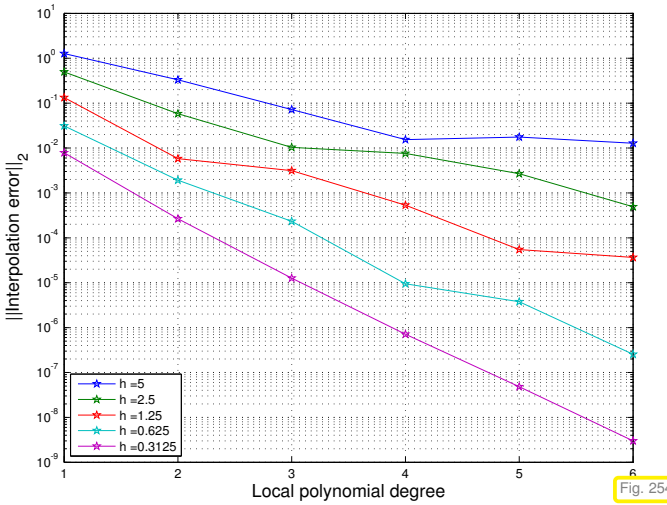


Fig. 253

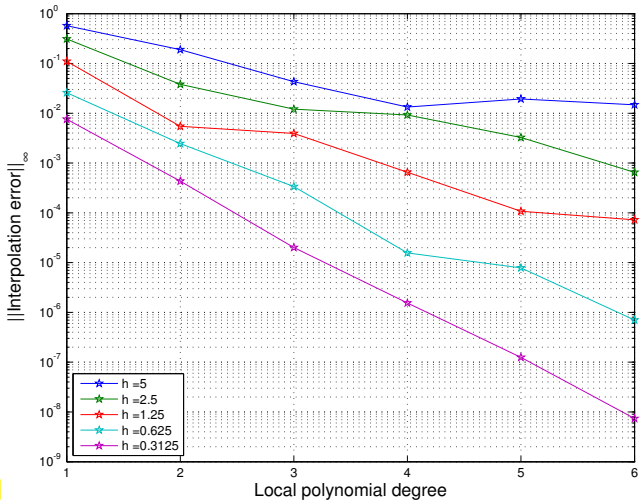


Fig. 254

Observation: (apparent) exponential convergence in polynomial degree

Note: in the case of p-convergence the situation is the same as for standard polynomial interpolation, see 6.1.2.

In this example we deal with an analytic function, see Rem. 6.1.96. Though equidistant local interpolation nodes are used *cf.* Ex. 6.1.41, the mesh intervals seems to be small enough that even in this case exponential convergence prevails.

6.5.2 Cubic Hermite interpolation: error estimates

See Section 5.4 for definition and algorithms for cubic Hermite interpolation of data points, with a focus on shape preservation, however. If the derivative f' of the interpoland f is available (in procedural form), then it can be used to fix local cubic polynomials by prescribing point values and derivative values in the endpoints of grid intervals.

Definition 6.5.14. Piecewise cubic Hermite interpolant (with exact slopes) → Def. 5.4.1

[Piecewise cubic Hermite interpolant (with exact slopes) Given $f \in C^1([a,b])$ and a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$ the piecewise cubic Hermite interpolant (with exact slopes) $s : [a,b] \to \mathbb{R}$ is defined as

$$s_{|[x_{j-1},x_j]} \in \mathcal{P}_3 , \quad j = 1,\ldots,m \quad , \quad s(x_j) = f(x_j) , \quad s'(x_j) = f'(x_j) , \quad j = 0,\ldots,m .$$

Clearly, the piecewise cubic Hermite interpolant is continuously differentiable: $s \in C^1([a,b])$, *cf.* Cor. 5.4.2.

Experiment 6.5.15 (Convergence of Hermite interpolation with exact slopes)

In this experiment we study the h-convergence of Cubic Hermite interpolation for a smooth function.

Piecewise cubic Hermite interpolation of

$$f(x) = \arctan(x) .$$

- ✦ domain: $I = (-5,5)$
- ✦ mesh $\mathcal{T} = \{-5 + hj\}_{j=0}^{n} \subset I, h = \frac{10}{n}$,
- ✦ exact slopes $c_i = f'(t_i), i = 0,\ldots,n$

➤ algebraic convergence $O(h^4)$

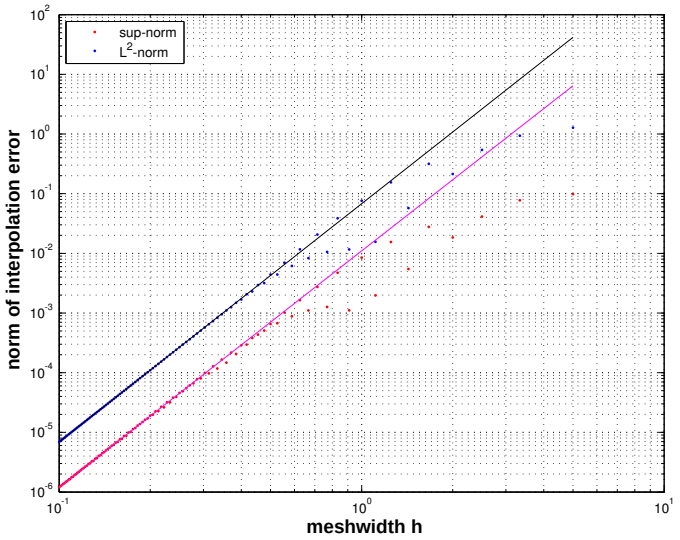


Fig. 255

Approximate computation of error norms analoguous to Ex. 6.5.10.

C++11 code 6.5.16: Hermite approximation and orders of convergence with exact slopes

```
2  void append(VectorXd&, const VectorXd&); // forward declaration of append()
3
4  // Investigation of interpolation error norms for cubic Hermite
       interpolation of f
5  // (passed through Functor f)
6  // on [a,b] with linearly averaged slopes according to (5.4.8).
7  // N gives the maximum number of mesh intervals
8  template <class Function, class Derivative>
9  void hermiteapprox(const Function& f, const Derivative& df,
10                    const double a, const double b, const unsigned N) {
11   std::vector<double> l2err, linferr, h; // save error and stepwidths in these vectors
12   for (unsigned j = 2; j <= N; ++j) {
13     // xx:  the fine mesh on which the error norms are computed
14     VectorXd xx(1); xx << a;
```

```
15    // val:  Hermite approximated values in points contained in xx
16    VectorXd val(1); val << f(a);
17
18    VectorXd t = VectorXd::LinSpaced(j, a, b); // mesh nodes
19    VectorXd y = feval(f, t); // feval returns f evaluated at t
20    VectorXd c = feval(df, t); // coefficients for Hermit polynomial
          representation
21
22    for (unsigned k = 0; k < j - 1; ++k) {
23      // local evaluation nodes in interval [t_k,t_{k+1}]
24      VectorXd vx = VectorXd::LinSpaced(100, t(k), t(k+1)),
25               locval;
26      // evaluate the hermite interpolant at the vx, using Code 5.4.6
27      hermloceval(vx, t(k), t(k+1), y(k), y(k+1), c(k), c(k+1), locval);
28      // do not append the first value as that one is already in the
            vector
29      append(xx, vx.tail(99));
30      append(val, locval.tail(99));
31    }
32
33    // difference between exact function and interpolant
34    VectorXd d = feval(f, xx) - val;
35    const double L2 = d.lpNorm<2>(), // L^2 norm
36                 Linf = d.lpNorm<Eigen::Infinity>(); // L^∞ norm
37    l2err.push_back(L2); // save L2 error
38    linferr.push_back(Linf); // save Linf error
39    h.push_back( (b - a)/j ); // save current meshwidth
40  }
41
42  // compute estimates for algebraic orderns of convergence
43  // using linear regression on half the data points
44
45  // L2Log contains the logarithmus of the last M l2err values,
46  // LinfLog and hLog the other according values
47  const unsigned M = unsigned( N / 2 );
48  VectorXd L2Log(M), LinfLog(M), hLog(M);
49  for (unsigned n = 0; n < M; ++n) {
50    // *(h.end() - n - 1) = h[end - n]
51    L2Log(M - n - 1) = std::log( *(l2err.end() - n - 1) );
52    LinfLog(M - n - 1) = std::log( *(linferr.end() - n - 1) );
53    hLog(M - n - 1) = std::log( *(h.end() - n - 1) );
54  }
55
56  // use linear regression
57  VectorXd pl = polyfit( hLog, LinfLog, 1 );
58  std::cout << "Algebraic convergence rate of Infinity norm: " << pl(0) << "\n";
59  VectorXd pL2 = polyfit( hLog, L2Log, 1 );
60  std::cout << "Algebraic convergence rate of L2 norm: " << pL2(0) << "\n";
61
62  mgl::Figure fig;
63  fig.setlog( true, true ); // double logarithmic plot
64  fig.title( "Hermite approximation" );
65  fig.xlabel( "Meshwidth h" );
66  fig.ylabel( "Norm of interpolation error" );
67  fig.legend();
68  fig.plot( h, l2err, " +r" ).label( "L^2 Error" );
69  fig.plot( h, linferr, " +b" ).label( "L^{\\infty} Error" );
70
71  // plot linear regression lines
72  fig.plot( std::vector<double>({h[0], h[N-2]}), std::vector<double>({std::pow(h[0],
        pL2(0))*std::exp(pL2(1)), std::pow(h[N-2], pL2(0))*std::exp(pL2(1))}), "m" );
73  fig.plot( std::vector<double>({h[0], h[N-2]}), std::vector<double>({std::pow(h[0],
        pl(0))*std::exp(pl(1)), std::pow(h[N-2], pl(0))*std::exp(pl(1))}), "k" );
74  fig.save( "hermiperravgsl" );
75 }
76
77 // Appends a Eigen::VectorXd to another Eigen::VectorXd
78 void append(VectorXd& x, const VectorXd& y) {
79   x.conservativeResize(x.size() + y.size());
80   x.tail(y.size()) = y;
81 }
```

The observation made in Exp. 6.5.15 matches the theoretical prediction of the rate of algebraic convergence for cubic Hermite interpolation with exact slopes for a smooth function.

Theorem 6.5.17. Convergence of approximation by cubic Hermite interpolation

Let s be the cubic Hermite interpolant of $f \in C^4([a,b])$ on a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$ according to Def. 6.5.14. Then

$$\|f - s\|_{L^\infty([a,b])} \leq \frac{1}{4!} h_{\mathcal{M}}^4 \left\| f^{(4)} \right\|_{L^\infty([a,b])} ,$$

with the meshwidth $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}|$.

In Section 5.4.2 we saw variants of cubic Hermite interpolation, for which the slopes $c_j = s'(x_j)$ were computed from the values y_j in preprocessing step. Now we study the use of such a scheme for approximation.

Experiment 6.5.18 (Convergence of Hermite interpolation with averaged slopes)

Piecewise cubic Hermite interpolation of

$$f(x) = \arctan(x) .$$

- ✦ domain: $I = (-5,5)$
- ✦ equidistant mesh $\mathcal{T}$ in I, see Exp. 6.5.15,
- ✦ averaged local slopes, see (5.4.8)

➤ algebraic convergence $O(h^3)$ in meshwidth

(see Code 6.5.19)

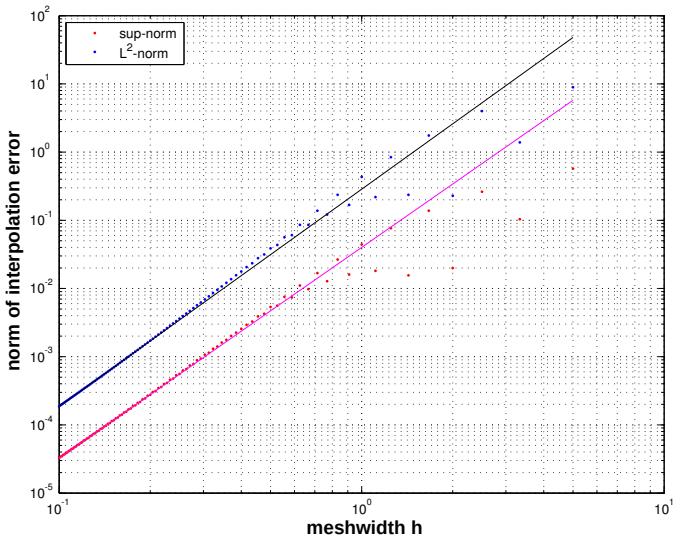


Fig. 256

We observe lower rate of algebraic convergence compared to the use of exact slopes due to averaging (5.4.8). From the plot we deduce $O(h^3)$ asymptotic decay of the L^2- and L^∞-norms of the approximation error for meshwidth $h \to 0$.

C++11 code 6.5.19: Hermite approximation and orders of convergence

```
2  VectorXd slopes(const VectorXd&, const VectorXd&); // forward declaration
3  void append(VectorXd&, const VectorXd&);
4
5  // Investigation of interpolation error norms for cubic Hermite
       interpolation of f (handle f)
6  // on [a,b] with linearly averaged slopes according to (5.4.8).
7  // N gives the maximum number of mesh intervals
8  template <class Function>
9  void hermiteapprox(const Function& f, const double a, const double b, const unsigned N) {
10   std::vector<double> l2err, linferr, h; // save error and stepwidth in these vectors
11   for (unsigned j = 2; j <= N; ++j) {
12     // xx is the fine mesh on which the error norms are computed
13     VectorXd xx(1); xx << a;
14     // val contains the hermite approximated values in xx
15     VectorXd val(1); val << f(a);
16
```

```
17    VectorXd t = VectorXd::LinSpaced(j, a, b); // mesh nodes
18    VectorXd y = feval(f, t); // feval returns f evaluated at t
19    VectorXd c = slopes(t, y); // coefficients for Hermit polynomial
          representation
20
21    for (unsigned k = 0; k < j - 1; ++k) {
22      // local evaluation nodes
23      VectorXd vx = VectorXd::LinSpaced(100, t(k), t(k+1)),
24               locval;
25      // evaluate the hermite interpolant at the vx, using Code 5.4.6
26      hermloceval(vx, t(k), t(k+1), y(k), y(k+1), c(k), c(k+1), locval);
27      // do not append the first value as that one is already in the
            vector
28      append(xx, vx.tail(99));
29      append(val, locval.tail(99));
30    }
31
32    // difference between exact function and interpolant
33    VectorXd d = feval(f, xx) - val;
34    const double L2 = d.lpNorm<2>(), // L^2 norm
35                 Linf = d.lpNorm<Eigen::Infinity>(); // L^∞ norm
36    l2err.push_back(L2); // save L2 error
37    linferr.push_back(Linf); // save Linf error
38    h.push_back( (b - a)/j ); // save current meshwidth
39  }
40
41  // compute estimates for algebraic orderns of convergence
42  // using linear regression on half the data points
43
44  // L2Log contains the logarithmus of the last M l2err values,
45  // LinfLog and hLog the other according values
46  const unsigned M = unsigned( N / 2 );
47  VectorXd L2Log(M), LinfLog(M), hLog(M);
48  for (unsigned n = 0; n < M; ++n) {
49    // *(h.end() - n - 1) = h[end - n]
50    L2Log(M - n - 1) = std::log( *(l2err.end() - n - 1) );
51    LinfLog(M - n - 1) = std::log( *(linferr.end() - n - 1) );
52    hLog(M - n - 1) = std::log( *(h.end() - n - 1) );
53  }
54
55  // use linear regression
56  VectorXd pl = polyfit( hLog, LinfLog, 1 );
57  std::cout << "Algebraic convergence rate of Infinity norm: " << pl(0) << "\n";
58  VectorXd pL2 = polyfit( hLog, L2Log, 1 );
59  std::cout << "Algebraic convergence rate of L2 norm: " << pL2(0) << "\n";
60
61  mgl::Figure fig;
62  fig.setlog( true, true ); // double logarithmic plot
63  fig.title( "Hermite approximation" );
64  fig.xlabel( "Meshwidth h" );
65  fig.ylabel( "Norm of interpolation error" );
66  fig.legend();
67  fig.plot( h, l2err, " +r" ).label( "L^2 Error" );
68  fig.plot( h, linferr, " +b" ).label( "L^{\\infty} Error" );
69
70  // plot linear regression lines
71  fig.plot( std::vector<double>({h[0], h[N-2]}), std::vector<double>({std::pow(h[0],
        pL2(0))*std::exp(pL2(1)), std::pow(h[N-2], pL2(0))*std::exp(pL2(1))}), "m" );
72  fig.plot( std::vector<double>({h[0], h[N-2]}), std::vector<double>({std::pow(h[0],
        pl(0))*std::exp(pl(1)), std::pow(h[N-2], pl(0))*std::exp(pl(1))}), "k" );
73
74  fig.save( "hermiperravgsl" );
75 }
76
77 // Computation of slopes of piecewise linear interpolation
78 // IN : t = nodes
79 // y = values at nodes t
80 // OUT: c = slopes of piecewise linear interpolation
81 VectorXd slopes(const VectorXd& t, const VectorXd& y) {
82   const unsigned n = t.size() - 1;
83   VectorXd h = t.tail(n) - t.head(n),
84            delta = (y.tail(n) - y.head(n)).cwiseQuotient(h),
85            c(n + 1);
86
87   c(0) = delta(0);
```

```
88    c(n) = delta(n-1);
89    for (unsigned i = 1; i < n; ++i) {
90      c(i) = (h(i)*delta(i - 1) + h(i - 1)*delta(i)) / (t(i + 1) - t(i - 1));
91    }
92    return c;
93  }
94
95  // Appends a Eigen::VectorXd to another Eigen::VectorXd
96  void append(VectorXd& x, const VectorXd& y) {
97    x.conservativeResize(x.size() + y.size());
98    x.tail(y.size()) = y;
99  }
```

6.5.3 Cubic spline interpolation: error estimates [?, Ch. 47]

Recall concept and algorithms for cubic spline interpolation from Section 5.5.1. As an interpolation scheme it can also serve as the foundation for an approximation scheme according to § 6.0.6: the mesh will double as know set, see Def. 5.5.1. Cubic spline interpolation is not local as we saw in § 5.5.19. Nevertheless, cubic spline interpolants can be computed with an effort of $O(m)$ as elaborated in § 5.5.5.

Experiment 6.5.20 (Approximation by complete cubic spline interpolants)

We take $I = [-1,1]$ and rely on an equidistant mesh (knot set) $\mathcal{M} := \{-1 + \frac{2}{n}j\}_{j=0}^{n}$, $n \in \mathbb{N}$ ➙ meshwidth $h = 2/n$.

We study h-convergence of complete (→ § 5.5.11) cubic spline interpolation, where the slopes at the endpoints of the interval are made to agree with the derivatives of the interpoland at these points. As interpolands we consider

$$f_1(t) = \frac{1}{1+e^{-2t}} \in C^\infty(I) \quad , \quad f_2(t) = \begin{cases} 0 & \text{, if } t < -\frac{2}{5} \text{ ,} \\ \frac{1}{2}(1+\cos(\pi(t-\frac{3}{5}))) & \text{, if } -\frac{2}{5} < t < \frac{3}{5} \text{ ,} \\ 1 & \text{otherwise.} \end{cases} \in C^1(I) \text{ .}$$

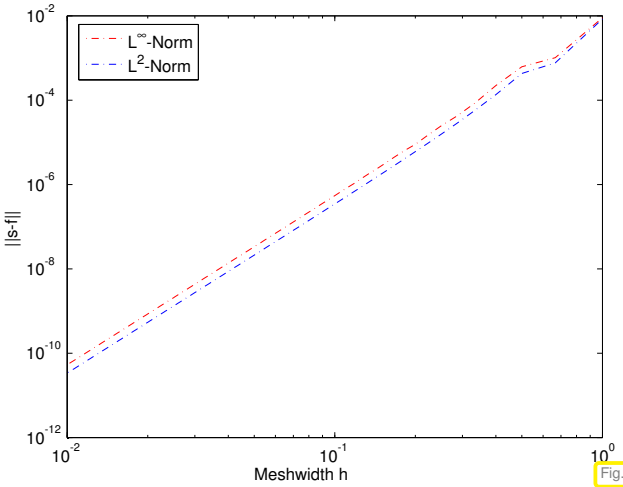


Fig. 257

$$\|f_1 - s\|_{L^\infty([-1,1])} = O(h^4)$$

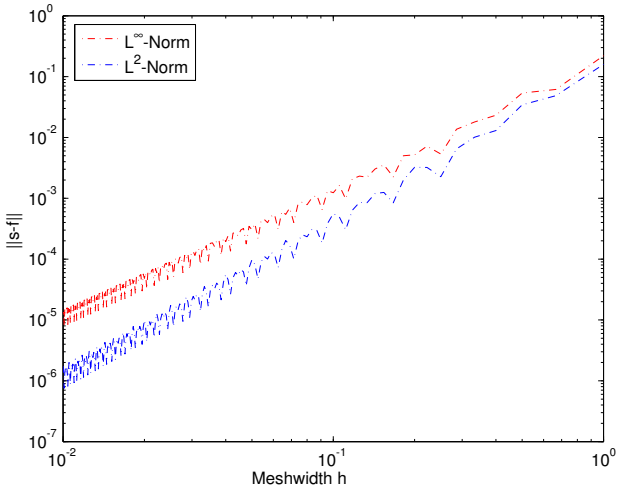


Fig. 258

$$\|f_2 - s\|_{L^\infty([-1,1])} = O(h^2)$$

We observe algebraic order of convergence in h with empiric rate approximately given by $\min\{1 +$ regularity of $f, 4\}$.

We remark that there is the following theoretical result [**?**], [**?**, Rem. 9.2]:

$$f \in C^4([t_0, t_n]) \qquad \|f - s\|_{L^\infty([t_0,t_n])} \leq \frac{5}{384} h^4 \left\| f^{(4)} \right\|_{L^\infty([t_0,t_n])} .$$

C++11 code 6.5.21: Computation and evaluation of complete cubic spline

```
// get coefficients for complete cubic spline interpolation
// IN : (t,y) = set of interpolation data
// c0, cn = slope at start and end point
// OUT: coefficients for complete cubic spline interpolation
VectorXd coeffsCSI(const VectorXd& t, const VectorXd& y, const double
    c0, const double cn)
{
  const long n = t.size() - 1;
  const VectorXd h = (t.tail(n).array() -
     t.head(n).array()).matrix(); // size:
     n
  const VectorXd b = (1./h.array()).matrix(); // size:  n
  const VectorXd a = 2*(b.head(n - 1).array() + b.tail(n -
     1).array()).matrix(); // size:  n -
     1

  // build rhs
  Eigen::VectorXd rhs(n - 1);
  for (long i = 0; i < n - 1; ++i){
    rhs(i) = 3*( (y(i + 1) - y(i))/(h(i)*h(i)) + (y(i + 2) - y(i +
       1))/(h(i + 1)*h(i + 1)) );
  }
  // modify according to complete cubic spline
  rhs(0) -= b(0)*c0;
  rhs(n - 2) -= b(n - 1)*cn;

  // build sparse system matrix
  typedef Eigen::Triplet<double> T;
  std::vector<T> triplets;
  triplets.reserve(3*n);
  for (int i = 0; i < n - 2; ++i){
    triplets.push_back( T(i, i, a(i)) );
    triplets.push_back( T(i, i + 1, b(i + 1)) );
    triplets.push_back( T(i + 1, i, b(i)) );
  }
  triplets.push_back( T(n - 2, n - 2, a(n - 2)) );
  SparseMatrix<double> A(n - 1,n - 1);
  A.setFromTriplets(triplets.begin(), triplets.end());

  // solve LSE and apply conditions
```

```
  Eigen::SparseLU<SparseMatrix<double>> solver;
  solver.compute(A);
  const VectorXd cpart = solver.solve(rhs);
  VectorXd c = Eigen::VectorXd(n + 1);
  c << c0, cpart, cn;
  return c;
}

// perform complete cubic spline interpolation of data (t_i, y_i)
// and evaluate at many points passed in vector x
VectorXd spline(const VectorXd& t, const VectorXd& y,
          const double c0, const double cn, const VectorXd& x) {
  const VectorXd c = coeffsCSI(t, y, c0, cn);
  return eval(x, t, y, c);
}
```

MATLAB-code 6.5.22: Spline approximation error

```
# include "polyfit.hpp" // provides polyfit(), see NumCSE/Utils
# include "feval.hpp" // provides feval(), see NumCSE/Utils
# include "completespline.hpp" // provides spline()
using Eigen::VectorXd;

// Study of interpolation error norms for complete cubic spline interpolation of f
// on equidistant knot set.
template <class Function, class Derivative>
void splineapprox(Function& f, Derivative& df, const double a, const double b, const unsigned N, const std::string plotname) {
  const unsigned K = int( (b-a)/0.00025 ); // no. of evaluation points for norm evaluation
  const VectorXd x = VectorXd::LinSpaced(K, a, b), // mesh for norm evaluation
                 fv = feval(f, x);

  std::vector<double> errL2, errInf, h;
  VectorXd v, // save result of spline approximation here
           t; // spline knots
  for (unsigned j = 3; j <= N; ++j) {
    t = VectorXd::LinSpaced(j, a, b); // spline knots
    VectorXd ft = feval(f, t);

    // compute complete spline imposing exact first derivative at the endpoints
    // spline: takes set of interpolation data (t, y) and slopes at endpoints,
    // and returns values at points x
    v = spline(t, ft, df(a), df(b), x);
    // compute error norms
    h.push_back( (b - a)/j ); // save current mesh width
    errL2.push_back( (fv - v).lpNorm<2>() );
    errInf.push_back( (fv - v).lpNorm<Eigen::Infinity>() );
  }

  // compute algebraic orders of convergence using polynomial fit
  const unsigned n = h.size();
  VectorXd hLog(n), errL2Log(n), errInfLog(n);
  for (unsigned i = 0; i < n; ++i) {
    hLog(i) = std::log(h[i]);
    errL2Log(i) = std::log(errL2[i]);
    errInfLog(i) = std::log(errInf[i]);
  }

  VectorXd p = polyfit(hLog, errL2Log, 1);
  std::cout << "L2 norm convergence rate: " << p(0) << "\n";
  p = polyfit(hLog, errInfLog, 1);
  std::cout << "Supremum norm convergence rate: " << p(0) << "\n";
```

```
45
46   // plot interpolation
47   mgl::Figure fig;
48   fig.title("Spline interpolation " + plotname);
49   fig.plot(t, feval(f, t), " m*").label("Data points");
50   fig.plot(x, fv, "b").label("f");
51   fig.plot(x, v, "r").label("Cubic spline interpolant");
52   fig.legend(1,0);
53   fig.save("interp_" + plotname);
54
55   // plot error
56   mgl::Figure err;
57   err.title("Spline approximation error " + plotname);
58   err.setlog(true, true);
59   err.plot(h, errL2, "r;").label("L^2 norm");
60   err.plot(h, errInf, "b;").label("L^\\infty norm");
61   fig.legend(1, 0);
62   err.save("approx_" + plotname);
63 }
```

Summary and Learning Outcomes

Chapter 7

Numerical Quadrature

Supplementary reading. Numerical quadrature is covered in [**?**, VII] and [**?**, Ch. 10].

Numerical quadrature deals with the *approximate* numerical evaluation of integrals $\int_\Omega f(\mathbf{x})\,\mathrm{d}\mathbf{x}$ for a given (closed) integration domain $\Omega \subset \mathbb{R}^d$. Thus, the underlying problem in the sense of § 1.5.67 is the mapping

$$I : \begin{cases} C^0(\Omega) & \to & \mathbb{R} \\ f & \mapsto & \int_\Omega f(\mathbf{x})\,\mathrm{d}\mathbf{x} \end{cases}, \tag{7.0.1}$$

with data space $X := C^0(\Omega)$ and result space $Y := \mathbb{R}$.

If f is complex-valued or vector-valued, then so is the integral. The methods presented in this chapter can immediately be generalized to this case by componentwise application.

(7.0.2) Integrands in procedural form

The integrand f, a continuos function $f : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$ should not be thought of as given by an analytic expression, but as given in procedural form, *cf.* Rem. 5.1.6,

- ☞ in MATLAB `function y = f(x)`
- ☞ in C++ as a data type providing an evaluation operator **double operator** `(Point &)` or a corresponding member function, see Code 5.1.7 for an example, or a lambda function

General methods for numerical quadrature should rely only on finitely many *point evaluations* of the integrand.

In this chapter the focus is on the special case $d = 1$: $\Omega = [a, b]$ (an interval)
(Multidimensional numerical quadrature is substantially more difficult, unless Ω is tensor-product domain, a multi-dimensional box. Multidimensional numerical quadrature will be treated in the course "Numerical methods for partial differential equations".)

Remark 7.0.3 (Importance of numerical quadrature)

☞ Numerical quadrature methods are key building blocks for so-called variational methods for the numerical treatment of partial differential equations. A prominent example is the finite element method.

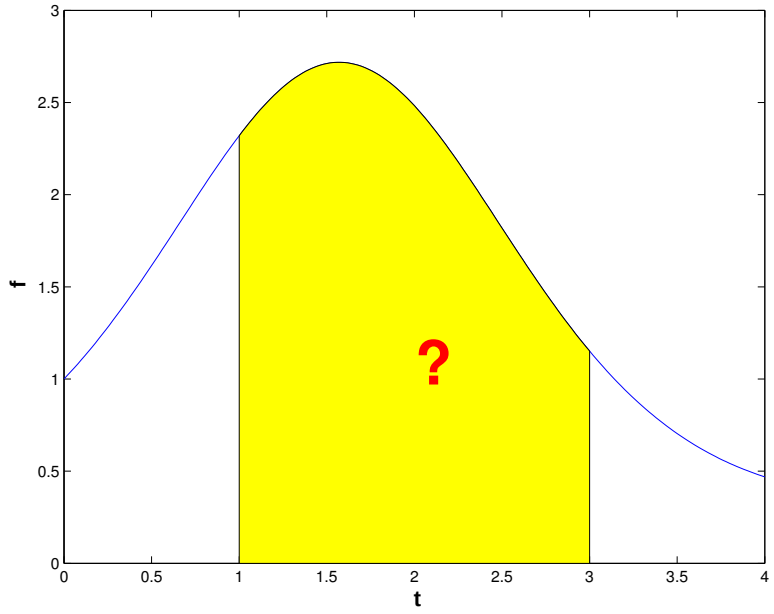


Fig. 259

For $d = 1$, from a geometric point of view methods for numerical quadrature aimed at computing

$$\int_a^b f(t)\,\mathrm{d}t$$

seek to *approximate* an area under the graph of the function f.

◁ area corresponding to value of an integral

Example 7.0.4 (Heating production in electrical circuits)

In Ex. 2.1.3 we learned about the nodal analysis of electrical circuits. Its application to a non-linear circuit will be discussed in Ex. 8.0.1, which will reveal that every computation of currents and voltages can be rather time-consuming. In this example we consider a non-linear circuit in quasi-stationary operation (capacities and inductances are ignored). Then the computation of branch currents and nodal voltages entails solving a non-linear system of equations.

Now assume time-harmonic periodic excitation with period $T > 0$.

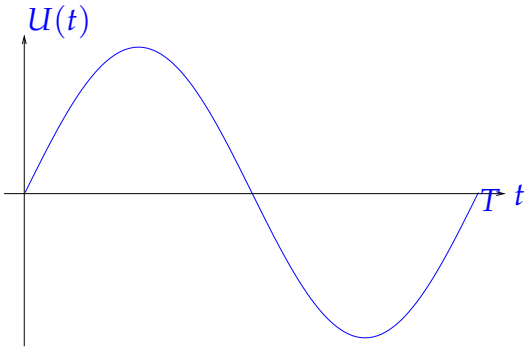


Fig. 260

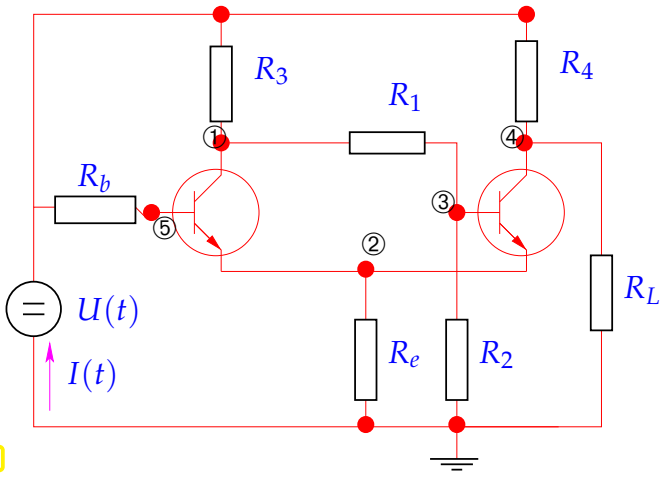


Fig. 261

The goal is to compute the energy dissipated by the circuit, which is equal to the energy injected by the voltage source. This energy can be obtained by integrating the power $P(t) = U(t)I(t)$ over period $[0, T]$:

$$W_{\text{therm}} = \int_0^T U(t)I(t)\,\mathrm{d}t\,, \quad \text{where} \quad I = I(U)\,.$$

double `I(double U)` involves solving non-linear system of equations, see Ex. 8.0.1!

This is a typical example where "point evaluation" by solving the non-linear circuit equations is the only way to gather information about the integrand.

Contents

7.1 Quadrature Formulas

Quadrature formulas realize the approximation of an integral through finitely many point evaluations of the integrand.

Definition 7.1.1. Quadrature formula/quadrature rule

An n-point quadrature formula/quadrature rule on $[a,b]$ provides an approximation of the value of an integral through a *weighted sum* of point values of the integrand:

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_n(f) := \sum_{j=1}^{n} w_j^n f(c_j^n)\,. \tag{7.1.2}$$

Terminology:
w_j^n : quadrature weights $\in \mathbb{R}$
c_j^n : quadrature nodes $\in [a,b]$

Obviously (7.1.2) is compatible with integrands f given in procedural form as `double f(double t)`, compare § 7.0.2.

C++-code 7.1.3: C++ template implementing generic quadrature formula

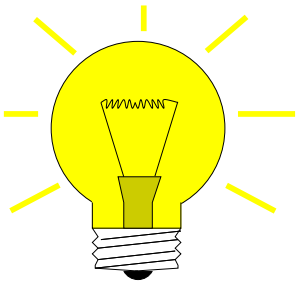
```
// Generic numerical quadrature routine implementing (7.1.2):
// f is a handle to a function, e.g. as lambda function
// c, w pass quadrature nodes c_j ∈ [a,b], and weights w_j ∈ ℝ
// in a Eigen::VectorXd
template <class Function>
double quadformula(Function& f, const VectorXd& c,const VectorXd& w) {
  const std::size_t n = c.size();
  double I = 0;
  for (std::size_t i = 0; i < n; ++i) I += w(i)*f(c(i));
  return I;
}
```

A single invocation costs n point evaluations of the integrand plus n additions and multiplications.

Remark 7.1.4 (Transformation of quadrature rules)

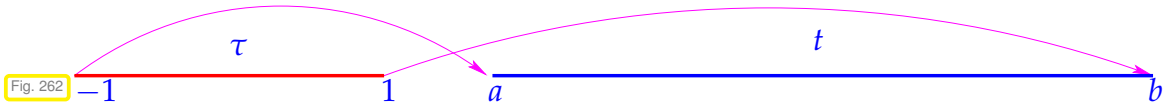
In the setting of function approximation by polynomials we learned in Rem. 6.1.18 that an approximation schemes for any interval could be obtained from an approximation scheme on a single reference interval ($[-1,1]$ in Rem. 6.1.18) by means of affine pullback, see (6.1.22). A similar affine transformation technique makes it possible to derive quadrature formula for an arbitrary interval from a single quadrature formula on a reference interval.

Given: quadrature formula $(\widehat{c}_j, \widehat{w}_j)_{j=1}^n$ on reference interval $[-1,1]$



Idea: transformation formula for integrals

$$\int_a^b f(t)\,\mathrm{d}t = \tfrac{1}{2}(b-a)\int_{-1}^1 \widehat{f}(\tau)\,\mathrm{d}\tau\,, \qquad \widehat{f}(\tau) := f(\tfrac{1}{2}(1-\tau)a + \tfrac{1}{2}(\tau+1)b)\,. \tag{7.1.5}$$



$$\tau \mapsto t := \Phi(\tau)t := \tfrac{1}{2}(1-\tau)a + \tfrac{1}{2}(\tau+1)b$$

Note that $\widehat{f}$ is the affine pullback $\Phi^* f$ of f to $[-1,1]$ as defined in Eq. (6.1.20).

▶ quadrature formula for general interval $[a,b]$, $a,b \in \mathbb{R}$:

$$\int_a^b f(t)\,\mathrm{d}t \approx \tfrac{1}{2}(b-a)\sum_{j=1}^n \widehat{w}_j \widehat{f}(\widehat{c}_j) = \sum_{j=1}^n w_j f(c_j) \quad \text{with} \quad \begin{aligned} c_j &= \tfrac{1}{2}(1-\widehat{c}_j)a + \tfrac{1}{2}(1+\widehat{c}_j)b\,, \\ w_j &= \tfrac{1}{2}(b-a)\widehat{w}_j\,. \end{aligned}$$

In words, the nodes are just mapped through the affine transformation $c_j = \Phi(\widehat{c}_j)$, the weights are scaled by the ratio of lengths of $[a,b]$ and $[-1,1]$.

▶ A 1D quadrature formula on arbitrary intervals can be specified by providing its weights $\widehat{w}_j$ /nodes $\widehat{c}_j$ for the integration domain $[-1,1]$ (reference interval). Then the above transformation is assumed.

Another common choice for the reference interval: $[0,1]$, pay attention!

Remark 7.1.6 (Tables of quadrature rules)

In many codes families of quadrature rules are used to control the quadrature error. Usually, suitable sequences of weights w_j^n and nodes c_j^n are precomputed and stored in tables up to sufficiently large values of n. A possible interface could be the following:

```
struct QuadTab {
  template <typename VecType>
    static void getrule(int n,VecType &c,VecType &w,
                        double a=-1.0,double b=1.0);
}
```

Calling the method `getrule()` fills the vectors `c` and `w` with the nodes and the weights for a desired n-point quadrature on $[a,b]$ with $[-1,1]$ being the default reference interval. For **VecType** we may assume the basic functionality of **Eigen::VectorXd**.

(7.1.7) Quadrature by approximation schemes

Every approximation scheme $\mathsf{A} : C^0([a,b]) \to V$, V a space of "simple functions" on $[a,b]$, see § 6.0.5, gives rise to a method for numerical quadrature according to

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_{\mathsf{A}}(f) := \int_a^b (\mathsf{A}f)(t)\,\mathrm{d}t\,. \tag{7.1.8}$$

As explained in § 6.0.6 every interpolation scheme $\mathsf{I}_{\mathcal{T}}$ based on the node set $\mathcal{T} = \{t_0, t_1, \ldots, t_n\} \subset [a,b]$ ($\to$ § 5.1.4) induces an approximation scheme, and, hence, also a quadrature scheme on $[a,b]$:

$$\int_a^b f(t)\,\mathrm{d}t \approx \int_a^b \mathsf{I}_{\mathcal{T}}[f(t_0),\ldots,f(t_n)]^\top(t)\,\mathrm{d}t\,. \tag{7.1.9}$$

Lemma 7.1.10. Quadrature formulas from linear interpolation schemes

Every linear *interpolation operator* $\mathsf{I}_{\mathcal{T}}$ *according to Def. 5.1.17 spawns a quadrature formula ($\to$ Def. 7.1.1) by* (7.1.9).

Proof. Writing $\mathbf{e}_j$ for the j-th unit vector of $\mathbb{R}^n$, we have by linearity

$$\int_a^b \mathsf{I}_{\mathcal{T}}[f(t_0),\ldots,f(t_n)]^\top\,\mathrm{d}t = \sum_{j=0}^{n} f(t_j) \underbrace{\int_a^b (\mathsf{I}_{\mathcal{T}}(\mathbf{e}_j))(t)\,\mathrm{d}t}_{\text{weight } w_j}\,. \tag{7.1.11}$$

Hence, we have arrived at an $n+1$-point quadrature formula with nodes t_j, whose weights are the integrals of the cardinal interpolants for the interpolation scheme $\mathcal{T}$.

□

Summing up, we have found

interpolation schemes $\longrightarrow$ approximation schemes $\longrightarrow$ quadrature schemes

(7.1.12) Convergence of numerical quadrature

In general the quadrature formula (7.1.2) will only provide an approximate value for the integral.

➢ For a generic integrand we will encounter a non-vanishing

quadrature error $$E_n(f) := \left| \int_a^b f(t)\,\mathrm{d}t - Q_n(f) \right|$$

As in the case of function approximation by interpolation Section 6.1.2, our focus will on the asymptotic behavior of the quadrature error as a function of the number n of point evaluations of the integrand.

Therefore consider families of quadrature rules $\{Q_n\}_n$ ($\to$ Def. 7.1.1) described by

- quadrature weights $\left\{w_j^n,\ j=1,\ldots,n\right\}_{n\in\mathbb{N}}$ and
- quadrature nodes $\left\{c_j^n,\ j=1,\ldots,n\right\}_{n\in\mathbb{N}}$.

We study the *asymptotic* behavior of the quadrature error $E(n)$ for $n\to\infty$

As in the case of interpolation errors in § 6.1.36 we make the usual qualitative distinction, see Def. 6.1.38:

▷ algebraic convergence $E(n)=O(n^{-p})$, rate $p>0$

▷ exponential convergence $E(n)=O(q^n)$, $0\le q<1$

Note that the number n of nodes agrees with the number of f-evaluations required for evaluation of the quadrature formula. This is usually used as a *measure for the cost* of computing $Q_n(f)$.

Therefore, in the sequel, we consider the quadrature error as a function of n.

(7.1.13) Quadrature error from approximation error

Bounds for the maximum norm of the approximation error of an approximation scheme directly translate into estimates of the quadrature error of the induced quadrature scheme (7.1.8):

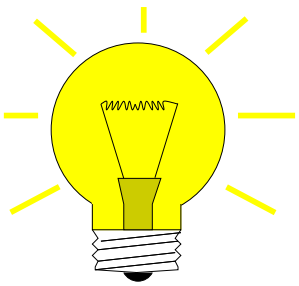
$$\left|\int_a^b f(t)\,\mathrm{d}t - Q\mathrm{A}(f)\right| \le \int_a^b |f(t)-\mathrm{A}(f)(t)|\,\mathrm{d}t \le |b-a|\|f-\mathrm{A}(f)\|_{L^\infty([a,b])}\,. \tag{7.1.14}$$

Hence, the various estimates derived in Section 6.1.2 and Section 6.1.3.2 give us quadrature error estimates "for free". More details will be given in the next section.

7.2 Polynomial Quadrature Formulas

Now we specialize the general recipe of § 7.1.7 for approximation schemes based on global polynomials, the Lagrange approximation scheme as introduced in Section 6.1, Def. 6.1.32.

Supplementary reading. This topic is discussed in [**?**, Sect. 10.2].



Idea: replace integrand f with $p_{n-1} := \mathrm{I}_{\mathcal{T}} f \in \mathcal{P}_{n-1}$ = polynomial Lagrange interpolant of f ($\to$ Cor. 5.2.15) for given node set $\mathcal{T} := \{t_0,\ldots,t_{n-1}\} \subset [a,b]$

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_n(f) := \int_a^b p_{n-1}(t)\,\mathrm{d}t\,. \tag{7.2.1}$$

The cardinal interpolants for Lagrange interpolation are the Lagrange polynomials (5.2.11)

$$L_i(t) := \prod_{\substack{j=0\\ j\neq i}}^{n-1} \frac{t-t_j}{t_i-t_j},\quad i=0,\ldots,n-1 \overset{(5.2.13)}{\Longrightarrow} p_{n-1}(t) = \sum_{i=0}^{n-1} f(t_i)L_i(t)\,.$$

Then (7.1.11) amounts to the n-point quadrature formula

$$\int_a^b p_{n-1}(t)\,\mathrm{d}t = \sum_{i=0}^{n-1} f(t_i) \int_a^b L_i(t)\,\mathrm{d}t \quad \blacktriangleright \quad \begin{array}{l} \text{nodes} \quad c_i = t_{i-1}\,, \\ \text{weights} \quad w_i := \int_a^b L_{i-1}(t)\,\mathrm{d}t\,. \end{array} \tag{7.2.2}$$

Example 7.2.3 (Midpoint rule)

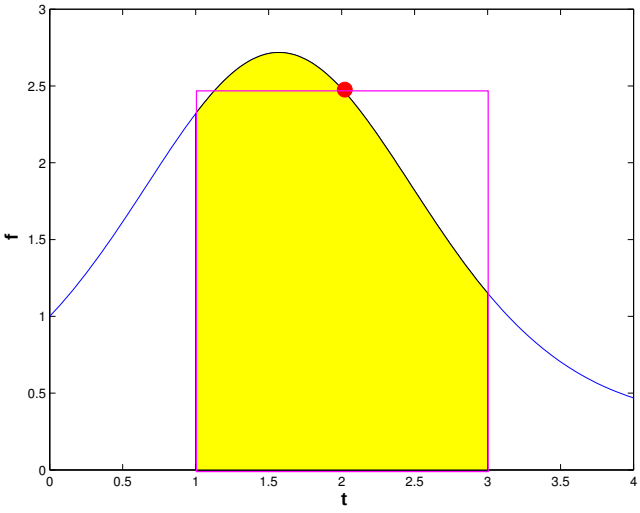


Fig. 263

The midpoint rule is (7.2.2) for $n = 1$ and $t_0 = \frac{1}{2}(a+b)$. It leads to the 1-point quadrature formula

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_{\mathrm{mp}}(f) = (b-a) f(\tfrac{1}{2}(a+b))\,.$$

"midpoint"

◁ the area under the graph of f is approximated by the area of a rectangle.

Example 7.2.4 (Newton-Cotes formulas → [?, Ch. 38])

The $n := m+1$-point Newton-Cotes formulas arise from Lagrange interpolation in equidistant nodes (6.1.34) in the integration interval $[a,b]$:

▶ Equidistant quadrature nodes $t_j := a + hj, \quad h := \dfrac{b-a}{m}, \quad j = 0,\ldots,m$:

The weights for the interval $[0,1]$ can be found, e.g., by symbolic computation using MAPLE: the following MAPLE function expects the polynomial degree as input argument.

```
> newtoncotes := m -> factor(int(interp([seq(i/n, i=0..m)],
                  [seq(f(i/n), i=0..m)], z),z=0..1)):
```

Weights on general intervals $[a,b]$ can then be deduced by the affine transformation rule as explained in Rem. 7.1.4.

• $n = 2$: Trapezoidal rule (integrate linear interpolant of integrand in endpoints)

```
> trapez := newtoncotes(1);
```

$$\widehat{Q}_{\mathrm{trp}}(f) := \frac{1}{2}(f(0)+f(1)) \qquad (7.2.5)$$

$$\left(\int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{2}(f(a)+f(b)) \right)$$

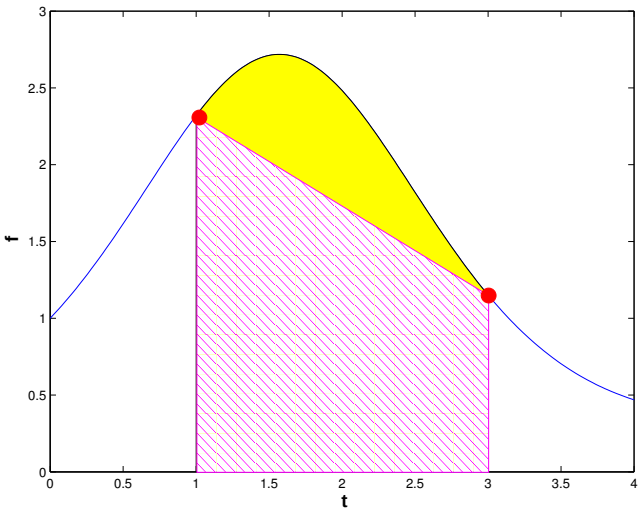


Fig. 264

- $n = 3$: Simpson rule

```
> simpson := newtoncotes(2);
```

$$\frac{h}{6}\left(f(0)+4f(\tfrac{1}{2})+f(1)\right) \quad \left(\int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right) \right) \qquad (7.2.6)$$

- $n = 5$: Milne rule

```
> milne := newtoncotes(4);
```

$$\frac{1}{90}h\left(7f(0)+32f(\tfrac{1}{4})+12f(\tfrac{1}{2})+32f(\tfrac{3}{4})+7f(1)\right)$$

$$\left(\frac{b-a}{90}(7f(a)+32f(a+(b-a)/4)+12f(a+(b-a)/2)+32f(a+3(b-a)/4)+7f(b))\right)$$

- $n = 7$: Weddle rule

```
> weddle := newtoncotes(6);
```

$$\frac{1}{840}h\,(41f(0)+216f(\tfrac{1}{6})+27f(\tfrac{1}{3})+272f(\tfrac{1}{2}) +27f(\tfrac{2}{3})+216f(\tfrac{5}{6})+41f(1))$$

- $n \geq 8$: quadrature formulas with *negative* weights

```
> newtoncotes(8);
```

$$\frac{1}{28350}h\,(989f(0)+5888f(\tfrac{1}{8})-928f(\tfrac{1}{4})+10496f(\tfrac{3}{8}) -4540f(\tfrac{1}{2})+10496f(\tfrac{5}{8})-928f(\tfrac{3}{4})+5888f(\tfrac{7}{8})+989f(1))$$

From Ex. 6.1.41 we know that the approximation error incurred by Lagrange interpolation in equidistant nodes can blow up even for analytic functions. This blow-up can also infect the quadrature error of Newton-Cotes formulas for large n, which renders them essentially useless. In addition they will be marred by large (in modulus) and negative weights, wich compromises numerical stability ($\rightarrow$ Def. 1.5.85)

Remark 7.2.7 (Clenshaw-Curtis quadrature rules [?])

The considerations of Section 6.1.3 confirmed the superiority of the "optimal" Chebychev nodes (6.1.84) for globally polynomial Lagrange interpolation. This suggests that we use these nodes also for numerical quadrature with weights given by (7.2.2). This yields the so-called Clenshaw-Curtis rules with the following rather desirable property:

Theorem 7.2.8. Positivity of Clenshaw-Curtis weights

The weights w_j^n, $j = 1, \ldots, n$, for every n-point Clenshaw-Curtis rule are positive.

The weights of any n-point Clenshaw-Curtis rule can be computed with a computational effort of $O(n \log n)$ using FFT.

(7.2.9) Error estimates for polynomial quadrature

As a concrete application of § 7.1.13, (7.1.14) we use the L^∞-bound (6.1.50) for Lagrange interpolation

$$\|f - \mathsf{L}_\mathcal{T} f\|_{L^\infty(I)} \leq \frac{\left\|f^{(n+1)}\right\|_{L^\infty(I)}}{(n+1)!} \max_{t \in I} |(t - t_0) \cdot \cdots \cdot (t - t_n)| \,. \tag{6.1.50}$$

to conclude for any n-point quadrature rule based on polynomial interpolation:

$$f \in C^n([a,b]) \quad \Rightarrow \quad \left| \int_a^b f(t)\,\mathrm{d}t - Q_n(f) \right| \leq \frac{1}{n!}(b-a)^{n+1} \left\|f^{(n)}\right\|_{L^\infty([a,b])} \,. \tag{7.2.10}$$

Much sharper estimates for Clenshaw-Curtis rules ($\rightarrow$ Rem. 7.2.7) can be inferred from the interpolation error estimate (6.1.88) for Chebychev interpolation. For functions with *limited smoothness* algebraic convergence of the quadrature error for Clenshaw-Curtis quadrature follows from (6.1.92). For integrands that possess an *analytic* extension to the complex plane in a neighborhood of $[a,b]$, we can conclude exponential convergence from (6.1.98).

7.3 Gauss Quadrature

Supplementary reading. Gauss quadrature is discussed in detail in [**?**, Ch. 40-41], [**?**, Sect.10.3]

How to gauge the "quality" of an n-point quadrature formula Q_n without testing it for specific integrands? The next definition gives an answer.

Definition 7.3.1. Order of a quadrature rule

The order of quadrature rule $Q_n : C^0([a,b]) \to \mathbb{R}$ is defined as

$$\text{order}(Q_n) := \max\{m \in \mathbb{N}_0: \quad Q_n(p) = \int_a^b p(t)\,\mathrm{d}t \quad \forall p \in \mathcal{P}_m\} + 1\,, \tag{7.3.2}$$

that is, as the maximal degree $+1$ of polynomials for which the quadrature rule is guaranteed to be exact.

(7.3.3) Order of polynomial quadrature rules

First we note a simple consequence of the invariance of the polynomial space $\mathcal{P}_n$ under affine pullback, see Lemma 6.1.21.

Corollary 7.3.4. Invariance of order under affine transformation

An affine transformation of a quadrature rule according to Rem. 7.1.4 does not change its order.

Further, by construction all polynomial n-point quadrature rules possess order at least n.

Theorem 7.3.5. Sufficient order conditions for quadrature rules

An n-point quadrature rule on $[a,b]$ ($\to$ Def. 7.1.1)

$$Q_n(f) := \sum_{j=1}^{n} w_j f(t_j)\,, \quad f \in C^0([a,b])\,,$$

with nodes $t_j \in [a,b]$ and weights $w_j \in \mathbb{R}$, $j = 1,\ldots,n$, has order $\geq n$, if and only if

$$w_j = \int_a^b L_{j-1}(t)\,dt\,, \quad j = 1,\ldots,n\,,$$

where L_k, $k = 0,\ldots,n-1$, is the k-th Lagrange polynomial (5.2.11) associated with the ordered node set $\{t_1, t_2, \ldots, t_n\}$.

Proof. The conclusion of the theorem is a direct consequence of the facts that

$$\mathcal{P}_{n-1} = \text{Span}\{L_0,\ldots,L_{n-1}\} \quad \text{and} \quad L_k(t_j) = \delta_{k+1,j}\,, \quad k+1, j \in \{1,\ldots,n\}\,:$$

just plug a Lagrange polynomial into the quadrature formula. □

By construction (7.2.2) polynomial n-point quadrature formulas (7.2.1) exact for $f \in \mathcal{P}_{n-1} \Rightarrow$ n-point polynomial quadrature formula has at least order n.

Remark 7.3.6 (Choice of (local) quadrature weights)

Thm. 7.3.5 provides a concrete formula for quadrature weights, which guaranteed order n for an n-point quadrature formula. Yet evaluating integrals of Lagrange polynomials may be cumbersome. Here we give a general recipe for finding the weights w_j according to Thm. 7.3.5 without dealing with Lagrange polynomials.

Given: arbitrary nodes $c_1, \ldots, c_n$ for n-point (local) quadrature formula on $[a,b]$

From Def. 7.3.1 we immediately conclude the following procedure: If $p_0, \ldots, p_{n-1}$ is a basis of $\mathcal{P}_n$, then, thanks to the linearity of the integral and quadrature formulas,

$$Q_n(p_j) = \int_a^b p_j(t)\,\mathrm{d}t \quad \forall j = 0, \ldots, n-1 \quad \Leftrightarrow \quad Q_n \text{ has order } \geq n\,. \tag{7.3.7}$$

➢ $n \times n$ linear system of equations, see (7.3.14) for an example:

$$\begin{bmatrix} p_0(c_1) & \ldots & p_0(c_n) \\ \vdots & & \vdots \\ p_{n-1}(c_1) & \ldots & p_{n-1}(c_n) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \int_a^b p_0(t)\,\mathrm{d}t \\ \vdots \\ \int_a^b p_{n-1}(t)\,\mathrm{d}t \end{bmatrix}. \tag{7.3.8}$$

For instance, for the computation of quadrature weights, one may choose the monomial basis $p_j(t) = t^j$.

Example 7.3.9 (Orders of simple polynomial quadrature formulas)

From the order rule for polynomial quadrature rule we immediately conclude the orders of simple representatives.

n		Order
1	midpoint rule	2
2	trapezoidal rule (7.2.5)	2
3	Simpson rule (7.2.6)	4
4	$\frac{3}{8}$-rule	4
5	Milne rule	6

The orders for even n surpass the predictions of Thm. 7.3.5 by 1, which can be verified by straightforward computations; following Def. 7.3.1 check the exactness of the quadrature rule on $[0,1]$ (this is sufficient $\rightarrow$ Cor. 7.3.4) for monomials $\{t \mapsto t^k\}$, $k = 0, \ldots, q-1$, which form a basis of $\mathcal{P}_q$, where q is the order that is to be confirmed: essentially one has to show

$$Q(\{t \mapsto t^k\}) = \sum_{j=1}^{n} w_j c_j^k = \frac{1}{k+1}\,, \quad k = 0, \ldots, q-1\,, \tag{7.3.10}$$

where $Q \mathrel{\hat{=}}$ quadrature rule on $[0,1]$ given by (7.1.2).

For the Simpson rule (7.2.6) we can also confirm order 4 with symbolic calculations in MAPLE:

```
> rule := 1/3*h*(f(2*h)+4*f(h)+f(0))
> err := taylor(rule - int(f(x),x=0..2*h),h=0,6);
```

$$err := \left(\frac{1}{90}\left(D^{(4)}\right)(f)(0)h^5 + O\left(h^6\right), h, 6\right)$$

➢ Composite Simpson rule possesses order 4, indeed **!**

(7.3.11) Maximal order of n-point quadrature rules

Natural question: Can an n-point quadrature formula achieve an order $> n$ **?**

A negative result limits the maximal order that can be achieved:

Theorem 7.3.12. Maximal order of n-point quadrature rule

The maximal order of an n-point quadrature rule is $2n$.

Proof. Consider a generic n-point quadrature rule according to Def. 7.1.1

$$Q_n(f) := \sum_{j=1}^{n} w_j^n f(c_j^n) , \tag{7.1.2}$$

We build a polynomial of degree $2n$ that cannot be integrated exactly by Q_n. We choose polynomial

$$q(t) := (t-c_1)^2 \cdot \cdots \cdot (t-c_n)^2 \in \mathcal{P}_{2n} .$$

On the one hand, q is strictly positive almost everywhere, which means

$$\int_a^b q(t)\,\mathrm{d}t > 0 .$$

On the other hand, we find a different value

$$Q_n(q) = \sum_{j=1}^{n} w_j^n \underbrace{q(c_j^n)}_{=0} = 0 .$$

□

Can we at least find n-point rules with order $2n$?

Heuristics: A quadrature formula has order $m \in \mathbb{N}$ already, if it is exact for m polynomials $\in \mathcal{P}_{m-1}$ that form a basis of $\mathcal{P}_{m-1}$ (recall Thm. 5.2.2).

$$\Updownarrow$$

An n-point quadrature formula has $2n$ "degrees of freedom" (n node positions, n weights).

$$\Downarrow$$

It might be possible to achieve order $2n = \dim \mathcal{P}_{2n-1}$
("No. of equations = No. of unknowns")

Example 7.3.13 (2-point quadrature rule of order 4)

Necessary & sufficient conditions for order 4, *cf.* (7.3.8), integrate the functions of the monomial basis of $\mathcal{P}_3$ exactly:

$$Q_n(p) = \int_a^b p(t)\,\mathrm{d}t \;\forall p \in \mathcal{P}_3 \quad \Leftrightarrow \quad Q_n(\{t \mapsto t^q\}) = \frac{1}{q+1}(b^{q+1} - a^{q+1})\,, \quad q = 0,1,2,3\,.$$

4 equations for weights w_j and nodes c_j, $j = 1,2$ ($a = -1, b = 1$), *cf.* Rem. 7.3.6

$$\begin{aligned} \int_{-1}^{1} 1\,\mathrm{d}t = 2 = 1w_1 + 1w_2 \quad , \quad \int_{-1}^{1} t\,\mathrm{d}t = 0 = c_1w_1 + c_2w_2 \\ \int_{-1}^{1} t^2\,\mathrm{d}t = \frac{2}{3} = c_1^2w_1 + c_2^2w_2 \quad , \quad \int_{-1}^{1} t^3\,\mathrm{d}t = 0 = c_1^3w_1 + c_2^3w_2\,. \end{aligned} \tag{7.3.14}$$

Solve using MAPLE:

```
> eqns := {seq(int(x^k, x=-1..1) = w[1]*xi[1]^k+w[2]*xi[2]^k,k=0..3)};
> sols := solve(eqns, indets(eqns, name)):
> convert(sols, radical);
```

➢ weights & nodes: $\left\{w_2 = 1, w_1 = 1, c_1 = 1/3\sqrt{3}, c_2 = -1/3\sqrt{3}\right\}$

quadrature formula (order 4): $$\int_{-1}^{1} f(x)\,dx \approx f\left(\frac{1}{\sqrt{3}}\right) + f\left(-\frac{1}{\sqrt{3}}\right) \tag{7.3.15}$$

(7.3.16) Construction of n-point quadrature rules with maximal order $2n$

First we search for *necessary conditions* that have to be met by the nodes, if an n-point quadrature rule has order $2n$.

Optimist's **assumption**: $\exists$ family of n-point quadrature formulas on $[-1,1]$

$$Q_n(f) := \sum_{j=1}^{n} w_j^n f(c_j^n) \approx \int_{-1}^{1} f(t)\,\mathrm{d}t\,, \quad w_j \in \mathbb{R}\,, n \in \mathbb{N}\,,$$

$$\text{of order } 2n \quad \Leftrightarrow \quad \text{exact for polynomials} \in \mathcal{P}_{2n-1}. \tag{7.3.17}$$

Define $\quad \bar{P}_n(t) := (t - c_1^n) \cdot \cdots \cdot (t - c_n^n)\,, \quad t \in \mathbb{R} \quad \Rightarrow \quad \bar{P}_n \in \mathcal{P}_n\,.$

Note: $\bar{P}_n$ has leading coefficient $= 1$.

By assumption on the order of Q_n: for any $q \in \mathcal{P}_{n-1}$

$$\int_{-1}^{1} \underbrace{q(t)\bar{P}_n(t)}_{\in \mathcal{P}_{2n-1}} \,\mathrm{d}t \overset{(7.3.17)}{=} \sum_{j=1}^{n} w_j^n q(c_j^n) \underbrace{\bar{P}_n(c_j^n)}_{=0} = 0 \,.$$

$$\Rightarrow \quad L^2([-1,1])\text{-orthogonality} \qquad \int_{-1}^{1} q(t)\bar{P}_n(t)\,\mathrm{d}t = 0 \quad \forall q \in \mathcal{P}_{n-1} \,. \tag{7.3.18}$$

$L^2(]-1,1[)$-inner product of q and $\bar{P}_n$, see (6.2.5).

Switching to a monomial representation of $\bar{P}_n$

$$\bar{P}_n(t) = t^n + \alpha_{n-1}t^{j-1} + \cdots + \alpha_1 t + \alpha_0 \,,$$

by linearity of the integral, (7.3.18) is equivalent to

$$\sum_{j=0}^{n-1} \alpha_j \int_{-1}^{1} t^j t^\ell \,\mathrm{d}t = -\int_{-1}^{1} t^n t^\ell \,\mathrm{d}t \,, \quad \ell = 0,\ldots,n-1 \,.$$

This is a linear system of equations $\mathbf{A}\left[\alpha_j\right]_{j=0}^{n-1} = \mathbf{b}$ with a symmetric, positive definite ($\rightarrow$ Def. 1.1.8) coefficient matrix $\mathbf{A} \in \mathbb{R}^{n,n}$. The $\mathbf{A}$ is positive definite can be concluded from

$$\mathbf{x}^\top \mathbf{A}\mathbf{x} = \int_{-1}^{1} \Big(\sum_{j=0}^{n-1} (\mathbf{x})_j t^j\Big)^2 \,\mathrm{d}t > 0 \quad \text{, if} \quad \mathbf{x} \neq 0 \,.$$

Hence, $\mathbf{A}$ is regular and the coefficients α_j are uniquely determined. Thus there is *only one* n-point quadrature rule of order n.

The nodes of an n-point quadrature formula of order $2n$, if it exists, must coincide with the unique zeros of the polynomials $\bar{P}_n \in \mathcal{P}_n \setminus \{0\}$ satisfying (7.3.18).

Remark 7.3.19 (Gram-Schmidt orthogonalization of polynomials)

Recall: $(f,g) \mapsto \int_a^b f(t)g(t)\,\mathrm{d}t$ is an inner product on $C^0([a,b])$, the L^2-inner product, see Rem. 6.2.20, [**?**, Sect. 4.4, Ex. 2], [**?**, Ex. 6.5]

➢ Treat space of polynomials $\mathcal{P}_n$ as a vector space equipped with an inner product.

➢ As we have seen in Section 6.2.2, abstract techniques for vector spaces with inner product can be applied to polynomials, for instance Gram-Schmidt orthogonalization, *cf.* § 6.2.17, [**?**, Thm. 4.8], [**?**, Alg. 6.1].

Now carry out the abstract Gram-Schmidt orthogonalization according to Algorithm (6.2.18) and recall Thm. 6.2.19: in a vector space V with inner product $(\cdot,\cdot)_V$ orthogonal vectors $q_0, q_1, \ldots$ spanning the same subspaces as the linearly independent vectors $v_0, v_1, \ldots$ are constructed recursively via

$$q_{n+1} := v_{n+1} - \sum_{k=0}^{n} \frac{(v_{n+1}, q_k)_V}{(q_k, q_k)_V} q_k \quad , \quad q_0 := v_0 \,. \tag{7.3.20}$$

➢ Construction of $\bar{P}_n$ by Gram-Schmidt orthogonalization of monomial basis $\{1, t, t^2, \ldots, t^{n-1}\}$ (the v_ks!) of $\mathcal{P}_{n-1}$ w.r.t. $L^2([-1,1])$-inner product:

$$\bar{P}_0(t) := 1 \,, \quad \bar{P}_{n+1}(t) = t^n - \sum_{k=0}^{n} \frac{\int_{-1}^{1} t^n \bar{P}_k(t)\,\mathrm{d}t}{\int_{-1}^{1} \bar{P}_k^2(t)\,\mathrm{d}t} \cdot \bar{P}_k(t) \tag{7.3.21}$$

Note: $\bar{P}_n$ has leading coefficient $= 1$ $\Rightarrow$ $\bar{P}_n$ uniquely defined (up to sign) by (7.3.21).

The considerations so far only reveal *necessary conditions* on the nodes of an n-point quadrature rule of order $2n$:

They do by no means confirm the existence of such rules, but offer a clear hint on how to construct them:

Theorem 7.3.22. Existence of n-point quadrature formulas of order $2n$

Let $\{\bar{P}_n\}_{n\in\mathbb{N}_0}$ be a family of non-zero polynomials that satisfies

- $\bar{P}_n \in \mathcal{P}_n$,
- $\int_{-1}^{1} q(t)\bar{P}_n(t)\,\mathrm{d}t = 0$ *for all* $q \in \mathcal{P}_{n-1}$ *($L^2([-1,1])$-orthogonality),*
- *The set $\{c_j^n\}_{j=1}^m$, $m \le n$, of real zeros of $\bar{P}_n$ is contained in $[-1,1]$.*

Then the quadrature rule ($\to$ Def. 7.1.1) $Q_n(f) := \sum_{j=1}^{m} w_j^n f(c_j^n)$

with weights chosen according to Thm. 7.3.5 provides a quadrature formula of order $2n$ on $[-1,1]$.

Proof. Conclude from the orthogonality of the $\bar{P}_n$ that $\{\bar{P}_k\}_{k=0}^n$ is a basis of $\mathcal{P}_n$ and

$$\int_{-1}^{1} h(t)\bar{P}_n(t)\,\mathrm{d}t = 0 \quad \forall h \in \mathcal{P}_{n-1}\,. \tag{7.3.23}$$

Recall division of polynomials with remainder (Euclid's algorithm $\to$ Course "Diskrete Mathematik"): for any $p \in \mathcal{P}_{2n-1}$

$$p(t) = h(t)\bar{P}_n(t) + r(t)\,, \quad \text{for some} \quad h \in \mathcal{P}_{n-1}, r \in \mathcal{P}_{n-1}\,. \tag{7.3.24}$$

Apply this representation to the integral:

$$\int_{-1}^{1} p(t)\,\mathrm{d}t = \underbrace{\int_{-1}^{1} h(t)\bar{P}_n(t)\,\mathrm{d}t}_{=0 \text{ by (7.3.23)}} + \int_{-1}^{1} r(t)\,\mathrm{d}t \overset{(*)}{=} \sum_{j=1}^{m} w_j^n r(c_j^n)\,, \tag{7.3.25}$$

$(*)$: by choice of weights according to Rem. 7.3.6 Q_n is exact for polynomials of degree $\le n-1$!

By choice of nodes as zeros of $\bar{P}_n$ using (7.3.23):

$$\sum_{j=1}^{m} w_j^n p(c_j^n) \overset{(7.3.24)}{=} \sum_{j=1}^{m} w_j^n h(c_j^n) \underbrace{\bar{P}_n(c_j^n)}_{=0} + \sum_{j=1}^{m} w_j^n r(c_j^n) \overset{(7.3.25)}{=} \int_{-1}^{1} p(t)\,\mathrm{d}t\,.$$

□

(7.3.26) Legendre polynomials and Gauss-Legendre quadrature

The family of polynomials $\{\bar{P}_n\}_{n\in\mathbb{N}_0}$ are so-called orthogonal polynomials w.r.t. the $L^2(]-1,1[)$-inner product, see Def. 6.2.25. We have made use of orthogonal polynomials already in Section 6.2.2. $L^2([-1,1])$-orthogonal polynomials play a key role in analysis.

The $L^2(]-1,1[)$-orthogonal are those already discussed in Rem. 6.2.34:

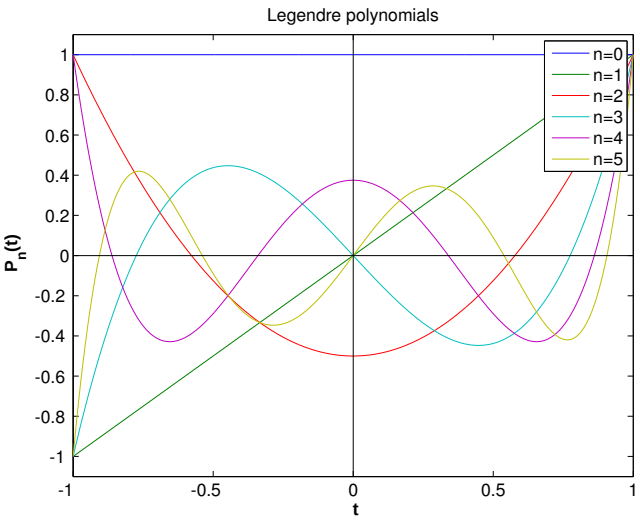
Definition 7.3.27. Legendre polynomials

The n-th Legendre polynomial P_n is defined by

- $P_n \in \mathcal{P}_n$,
- $\int_{-1}^{1} P_n(t)q(t)\,\mathrm{d}t = 0 \quad \forall q \in \mathcal{P}_{n-1}$,
- $P_n(1) = 1$.

Legendre polynomials $P_0, \ldots, P_5$ ➢

Fig. 265



Notice: the polynomials $\bar{P}_n$ defined by (7.3.21) and the Legendre polynomials P_n of Def. 7.3.27 (merely) *differ* by a constant factor!

◥ Gauss points ζ_j^n = zeros of Legendre polynomial P_n

Note: the above considerations, recall (7.3.18), show that the nodes of an n-point quadrature formula of order $2n$ on $[-1,1]$ must agree with the zeros of $L^2(]-1,1[)$-orthogonal polynomials.

◥ n-point quadrature formulas of order $2n$ are unique

This is not surprising in light of "$2n$ equations for $2n$ degrees of freedom".

We are not done yet: the zeros of $\bar{P}_n$ from (7.3.21) may lie outside $[-1,1]$.
In principle $\bar{P}_n$ could also have less than n real zeros.

The next lemma shows that all this cannot happen.

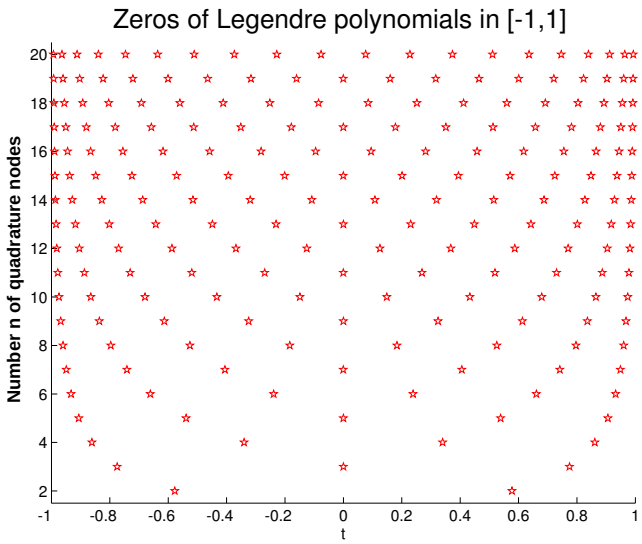


Fig. 266

◁ Obviously:

Lemma 7.3.28. Zeros of Legendre polynomials

P_n *has* n *distinct zeros in* $]-1,1[$.

Zeros of Legendre polynomials = Gauss points

Proof. *(indirect)* Assume that P_n has only $m < n$ zeros $\zeta_1, \ldots, \zeta_m$ in $]-1,1[$ *at which it changes sign.* Define

$$q(t) := \prod_{j=1}^{m}(t-\zeta_j) \quad \Rightarrow \quad qP_n \geq 0 \quad \text{or} \quad qP_n \leq 0\,.$$

$$\Rightarrow \quad \int_{-1}^{1} q(t)P_n(t)\,\mathrm{d}t \neq 0\,.$$

As $q \in \mathcal{P}_{n-1}$, this contradicts (7.3.23).

□

Definition 7.3.29. Gauss-Legendre quadrature formulas

The n-point Quadrature formulas whose nodes, the Gauss points, are given by the zeros of the n-th Legendre polynomial ($\rightarrow$ Def. 7.3.27), and whose weights are chosen according to Thm. 7.3.5, are called Gauss-Legendre quadrature formulas.

Obviously ▷

Lemma 7.3.30. Positivity of Gauss-Legendre quadrature weights

The weights of the Gauss-Legendre quadrature formulas are positive.

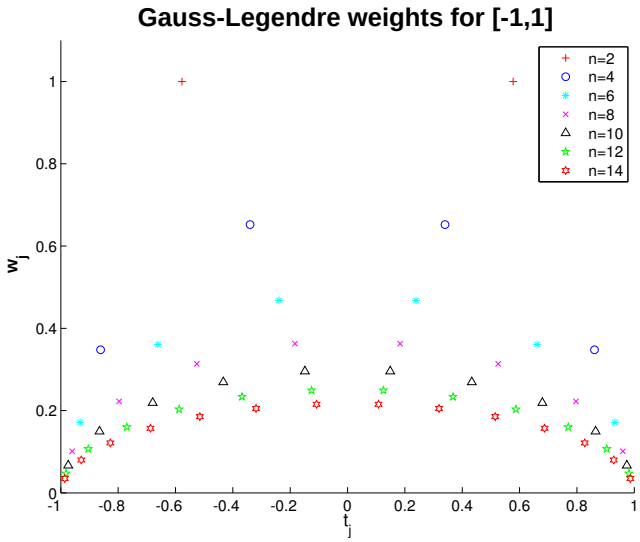


Fig. 267

Proof. Writing ξ_j^n, $j = 1, \ldots, n$, for the nodes (Gauss points) of the n-point Gauss-Legendre quadrature formula, $n \in \mathbb{N}$, we define

$$q_k(t) = \prod_{\substack{j=1\\ j\neq k}}^{n}(t-\xi_j^n)^2 \quad \Rightarrow \quad q_k \in \mathcal{P}_{2n-2}\,.$$

This polynomial is integrated exactly by the quadrature rule: since $q_k(\xi_j^n) = 0$ for $j \neq k$

$$0 < \int_{-1}^{1} q(t)\,\mathrm{d}t = w_k^n \underbrace{q(\xi_k^n)}_{>0}\,,$$

where w_j^n are the quadrature weights.

□

Remark 7.3.31 (3-Term recursion for Legendre polynomials)

From Thm. 6.2.32 we learn the orthogonal polynomials satisfy the 3-term recursion (6.2.33), see also (7.3.33). To keep this chapter self-contained we derive it independently for Legendre polynomials.

Note: the polynomials $\bar{P}_n$ from (7.3.21) are uniquely characterized by the two properties (try a proof!)

(i) $\bar{P}_n \in \mathcal{P}_n$ with leading coefficient 1: $\bar{P}(t) = t^n + \ldots$,

(ii) $\int_{-1}^{1} \bar{P}_k(t)\bar{P}_j(t)\,\mathrm{d}t = 0$, if $j \neq k$ ($L^2(]-1,1[)$-orthogonality).

➢ we get the same polynomials $\bar{P}_n$ by another Gram-Schmidt orthogonalization procedure, *cf.* (7.3.20) and § 6.2.29:

$$\bar{P}_{n+1}(t) = t\bar{P}_n(t) - \sum_{k=0}^{n} \frac{\int_{-1}^{1} \tau\bar{P}_n(\tau)\bar{P}_k(\tau)\,\mathrm{d}\tau}{\int_{-1}^{1} \bar{P}_k^2(\tau)\,\mathrm{d}\tau} \cdot \bar{P}_k(t)$$

By the orthogonality property (7.3.23) the sum collapses, since

$$\int_{-1}^{1} \tau\bar{P}_n(\tau)\bar{P}k(\tau)\,\mathrm{d}\tau = \int_{-1}^{1} \bar{P}_n(\tau)\underbrace{(\tau\bar{P}k(\tau))}_{\in\mathcal{P}_{k+1}}\,\mathrm{d}\tau = 0\,,$$

if $k+1 < n$:

$$\bar{P}_{n+1}(t) = t\bar{P}_n(t) - \frac{\int_{-1}^{1} \tau\bar{P}_n(\tau)\bar{P}_n(\tau)\,\mathrm{d}\tau}{\int_{-1}^{1} \bar{P}_n^2(\tau)\,\mathrm{d}\tau} \cdot \bar{P}_n(t) - \frac{\int_{-1}^{1} \tau\bar{P}_n(\tau)\bar{P}_{n-1}(\tau)\,\mathrm{d}\tau}{\int_{-1}^{1} \bar{P}_{n-1}^2(\tau)\,\mathrm{d}\tau} \cdot \bar{P}_{n-1}(t)\,. \tag{7.3.32}$$

After rescaling (tedious!): 3-term recursion for Legendre polynomials

$$P_{n+1}(t) := \frac{2n+1}{n+1}tP_n(t) - \frac{n}{n+1}P_{n-1}(t) \quad , \quad P_0 := 1\,, \quad P_1(t) := t\,. \tag{7.3.33}$$

Reminder (→ Section 6.1.3.1): we have a similar 3-term recursion (6.1.78) for Chebychev polynomials. Coincidence? Of course not, nothing in mathematics holds "by accident". By Thm. 6.2.32 3-term recursions are a distinguishing feature of so-called families of orthogonal polynomials, to which the Chebychev polynomials belong as well, spawned by Gram-Schmidt orthogonalization with respect to a weighted L^2-inner product, however, see [**?**, VI].

➤ Efficient and *stable* evaluation of Legendre polynomials by means of 3-term recursion (7.3.33), *cf.* the analoguous algorithm for Chebychev polynomials given in Code 6.1.79.

C++-code 7.3.34: computing Legende polynomials

```
2  // returns the values of the first n - 1 legendre polynomials
3  // in point x as columns of the matrix L
4  void legendre(const unsigned n, const VectorXd& x, MatrixXd& L) {
5    L = MatrixXd::Ones(n,n); // p_0(x) = 1
6    L.col(1) = x;            // p_1(x) = x
7    for (unsigned j = 1; j < n - 1; ++j) {
8      // p_{j+1}(x) = (2j+1)/(j+1) x p_j(x) - j/(j+1) p_{j-1}(x) Eq. (7.3.33)
9      V.col(j+1) = (2.*j+1)/(j+1.)*V.col(j-1).cwiseProduct(x)
10       -j/(j+1.)*V.col(j-1);
```

```
11     }
12 }
```

Comments on Code 7.3.34:

- return value: matrix $\mathbf{V}$ with $(\mathbf{V})_{ij} = P_i(x_j)$
- line 2: takes into account initialization of Legendre 3-term recursion (7.3.33)

Remark 7.3.35 (Computing Gauss nodes and weights)

There are several efficient ways to find the Gauss points. Here we discuss an intriguing connection with an eigenvalue problem.

Compute nodes/weights of Gaussian quadrature by solving an eigenvalue problem!

(Golub-Welsch algorithm [**?**, Sect. 3.5.4], [**?**, Sect. 1])

In codes Gauss nodes and weights are usually retrieved from tables, *cf.* Rem. 7.1.6.

C++-code 7.3.36: Golub-Welsch algorithm

```
2  struct QuadRule {
3      Eigen::VectorXd nodes, weights;
4  };
5
6  QuadRule gaussquad_(const unsigned n) {
7      QuadRule qr;
8      Eigen::MatrixXd M = Eigen::MatrixXd::Zero(n, n);
9      for (unsigned i = 1; i < n; ++i){
10         const double b = i/std::sqrt(4.*i*i - 1.);
11         M(i, i - 1) = b;
12         // line 15 is optional as the EV-Solver only references
13         // the lower triangular part of M
14         // M(i - 1, i) = b;
15     }
16     // using EigenSolver for symmetric matrices, exploiting the
       structure
17     Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> eig(M);
18
19     qr.nodes = eig.eigenvalues();
20     qr.weights = 2*eig.eigenvectors().topRows<1>().array().pow(2);
21
22     return qr;
23 }
```

Justification: rewrite 3-term recurrence (7.3.33) for scaled Legendre polynomials $\widetilde{P}_n = \frac{1}{\sqrt{n+1/2}} P_n$

$$t\widetilde{P}_n(t) = \underbrace{\frac{n}{\sqrt{4n^2-1}}}_{=:\beta_n} \widetilde{P}_{n-1}(t) + \underbrace{\frac{n+1}{\sqrt{4(n+1)^2-1}}}_{=:\beta_{n+1}} \widetilde{P}_{n+1}(t)\,. \tag{7.3.37}$$

For fixed $t \in \mathbb{R}$ (7.3.37) can be expressed as

$$t \underbrace{\begin{bmatrix} \widetilde{P}_0(t) \\ \widetilde{P}_1(t) \\ \vdots \\ \widetilde{P}_{n-1}(t) \end{bmatrix}}_{=:\mathbf{p}(t)\in\mathbb{R}^n} = \underbrace{\begin{bmatrix} 0 & \beta_1 & & & & \\ \beta_1 & 0 & \beta_2 & & & \\ & \beta_2 & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & & 0 & \beta_{n-1} \\ & & & & \beta_{n-1} & 0 \end{bmatrix}}_{=:\mathbf{J}_n\in\mathbb{R}^{n,n}} \begin{bmatrix} \widetilde{P}_0(t) \\ \widetilde{P}_1(t) \\ \vdots \\ \widetilde{P}_{n-1}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \beta_n \widetilde{P}_n(t) \end{bmatrix}$$

$$\blacktriangleright \quad \widetilde{P}_n(\xi) = 0 \;\Leftrightarrow\; \xi \mathbf{p}(\xi) = \mathbf{J}_n \mathbf{p}(\xi) \quad \text{(eigenvalue problem!)} .$$

$\blacktriangleright$ The zeros of P_n can be obtained as the n real eigenvalues of the symmetric tridiagonal matrix $\mathbf{J}_n \in \mathbb{R}^{n,n}$!

This matrix $\mathbf{J}_n$ is initialized in **??–??** of Code 7.3.36. The computation of the weights in **??** of Code 7.3.36 is explained in [**?**, Sect. 3.5.4].

(7.3.38) Quadrature error and best approximation error

The positivity of the weights w_j^n for all n-point Gauss-Legendre and Clenshaw-Curtis quadrature rules has important consequences.

Theorem 7.3.39. Quadrature error estimate for quadrature rules with positive weights

For every n-point quadrature rule Q_n as in (7.1.2) *of order $q \in \mathbb{N}$ with weights $w_j \geq 0$, $j = 1, \ldots, n$ the quadrature error satisfies*

$$E_n(f) := \left| \int_a^b f(t)\,dt - Q_n(f) \right| \leq 2|b-a| \underbrace{\inf_{p\in\mathcal{P}_{q-1}} \|f-p\|_{L^\infty([a,b])}}_{\text{best approximation error}} \quad \forall f \in C^0([a,b]) . \tag{7.3.40}$$

Proof. The proof runs parallel to the derivation of (6.1.61). Writing $E_n(f)$ for the quadrature error, the left hand side of (7.3.40), we find by the definition Def. 7.3.1 of the order of a quadrature rule

$$\begin{aligned} E_n(f) = E_n(f-p) &\leq \left| \int_a^b (f-p)(t)\,\mathrm{d}t \right| + \left| \sum_{j=1}^n w_j (f-p)(c_j) \right| \\ &\leq |b-a| \|f-p\|_{L^\infty([a,b])} + \Big(\sum_{j=1}^n |w_j|\Big) \|f-p\|_{L^\infty([a,b])} . \end{aligned} \tag{7.3.41}$$

Since the quadrature rule is exact for constants and $w_j \geq 0$

$$\sum_{j=1}^n |w_j| = \sum_{j=1}^n w_j = |b-a| ,$$

which finishes the proof. □

Drawing on best approximation estimates from Section 6.1.1 and Rem. 6.1.96, we immediately get results about the asymptotic decay of the quadrature error for n-point Gauss-Legendre and Clenshaw-Curtis quadrature as $n \to \infty$:

$f \in C^r([a,b]) \quad \Rightarrow \quad E_n(f) \to 0$ algebraically with rate r,
$f \in C^\infty([a,b])$ "analytically extensible" $\Rightarrow \quad E_n(f) \to 0$ exponentially,

as $n \to \infty$, see Def. 6.1.38 for type of convergence.

Appealing to Thm. 6.1.15 and Rem. 6.1.23, and (6.1.50), the dependence of the constants on the length of the integration interval can be quantified for integrands with limited smoothness.

Lemma 7.3.42. Quadrature error estimates for C^r-integrands

For every n-point quadrature rule Q_n as in (7.1.2) of order $q \in \mathbb{N}$ with weights $w_j \geq 0$, $j = 1,\ldots,n$ we find that the quadrature error $E_n(f)$ for and integrand $f \in C^r([a,b])$, $r \in \mathbb{N}_0$, satisfies

in the case $q \geq r$: $$E_n(f) \leq C\, q^{-r} |b-a|^{r+1} \left\| f^{(r)} \right\|_{L^\infty([a,b])} , \tag{7.3.43}$$

in the case $q < r$: $$E_n(f) \leq \frac{|b-a|^{q+1}}{q!} \left\| f^{(q)} \right\|_{L^\infty([a,b])} , \tag{7.3.44}$$

with a constant $C > 0$ independent of n, f, and $[a,b]$.

Please note the different estimates depending on whether the smoothness of f (as described by r) or the order of the quadrature rule is the "limiting factor".

Example 7.3.45 (Convergence of global quadrature rules)

We examine three families of global polynomial ($\to$ Thm. 7.3.5) quadrature rules: Newton-Cotes formulas, Gauss-Legendre rules, and Clenshaw-Curtis rules. We record the convergence of the quadrature errors for the interval $[0,1]$ and two different functions

1. $f_1(t) = \frac{1}{1+(5t)^2}$, an analytic function, see Rem. 6.1.72,
2. $f_2(t) = \sqrt{t}$, merely continuous, derivatives singular in $t = 0$.

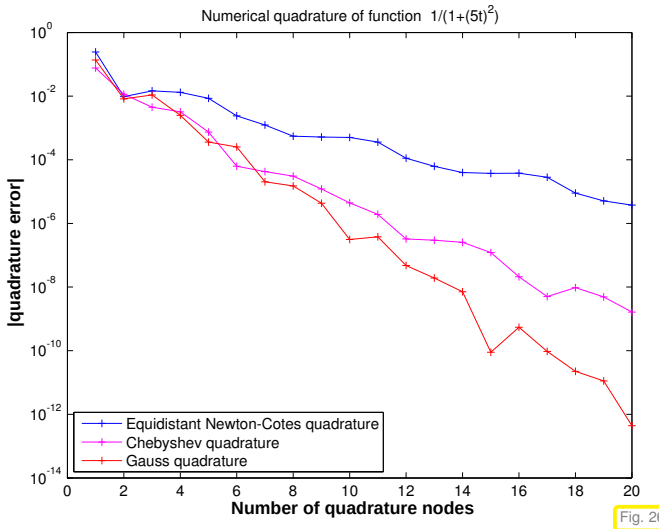


Fig. 268

quadrature error, $f_1(t) := \frac{1}{1+(5t)^2}$ on $[0,1]$

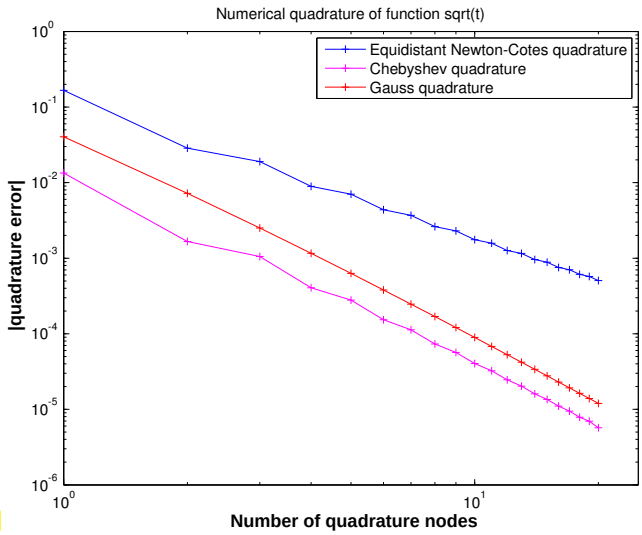


Fig. 269

quadrature error, $f_2(t) := \sqrt{t}$ on $[0,1]$

Asymptotic behavior of quadrature error $\epsilon_n := \left| \int_0^1 f(t)\,\mathrm{d}t - Q_n(f) \right|$ for "$n \to \infty$":

➢ exponential convergence $\epsilon_n \approx O(q^n)$, $0 < q < 1$, for C^∞-integrand f_1 $\rightsquigarrow$: Newton-Cotes quadrature : $q \approx 0.61$, Clenshaw-Curtis quadrature : $q \approx 0.40$, Gauss-Legendre quadrature : $q \approx 0.27$

➢ algebraic convergence $\epsilon_n \approx O(n^{-\alpha})$, $\alpha > 0$, for integrand f_2 with singularity at $t = 0$ $\rightsquigarrow$ Newton-Cotes quadrature : $\alpha \approx 1.8$, Clenshaw-Curtis quadrature : $\alpha \approx 2.5$, Gauss-Legendre quadrature : $\alpha \approx 2.7$

Remark 7.3.46 (Removing a singularity by transformation)

From Ex. 7.3.45 teaches us that a lack of smoothness of the integrand can thwart exponential convergence and severely limits the rate of algebraic convergence of a global quadrature rule for $n \to \infty$.

Idea: recover integral with smooth integrand by "analytic preprocessing"

Here is an example:

For a general but smooth $f \in C^\infty([0,b])$ compute $\int_0^b \sqrt{t} f(t)\,\mathrm{d}t$ via a quadrature rule, e.g., n-point Gauss-Legendre quadrature on $[0,b]$. Due to the presence of a square-root singularity at $t = 0$ the direct application of n-point Gauss-Legendre quadrature will result in a rather slow algebraic convergence of the quadrature error as $n \to \infty$, see Ex. 7.3.45.

Trick: Transformation of integrand by substitution rule:

$$\text{substitution } s = \sqrt{t}: \quad \int_0^b \sqrt{t} f(t)\,\mathrm{d}t = \int_0^{\sqrt{b}} \boxed{2s^2 f(s^2)}\,\mathrm{d}s\,. \tag{7.3.47}$$

Then: Apply Gauss-Legendre quadrature rule to smooth integrand

Remark 7.3.48 (The message of asymptotic estimates)

There is one blot on most n-asymptotic estimates obtained from Thm. 7.3.39: the bounds usually involve quantities like norms of higher derivatives of the interpoland that are elusive in general, in particular for integrands given only in procedural form, see § 7.0.2. Such unknown quantities are often hidden in "generic constants C". Can we extract useful information from estimates marred by the presence of such constants?

For fixed integrand f let us assume *sharp* algebraic convergence (in n) with rate $r \in \mathbb{N}$ of the quadrature error $E_n(f)$ for a family of n-point quadrature rules:

$$E_n(f) = O(n^{-r}) \overset{\text{sharp}}{\Longrightarrow} E_n(f) \approx Cn^{-r}\,, \tag{7.3.49}$$

with a "generic constant $C > 0$" independent of n.

Goal: Reduction of the quadrature error by a factor of $\rho > 1$

Which (minimal) increase in the number n of quadrature points accomplishes this?

$$\frac{Cn_{\text{old}}^{-r}}{Cn_{\text{new}}^{-r}} \overset{!}{=} \rho \quad \Leftrightarrow \quad \boxed{n_{\text{new}} : n_{\text{old}} = \sqrt[r]{\rho}}\,. \tag{7.3.50}$$

In the case of *algebraic convergence* with rate $r \in \mathbb{R}$ a reduction of the quadrature error by a factor of ρ is bought by an increase of the number of quadrature points by a factor of $\rho^{1/r}$.

(7.3.43) ➢ gains in accuracy are "cheaper" for smoother integrands!

Now assume *sharp* exponential convergence (in n) of the quadrature error $E_n(f)$ for a family of n-point quadrature rules, $0 \leq q < 1$:

$$E_n(f) = O(q^n) \overset{\text{sharp}}{\Longrightarrow} E_n(f) \approx Cq^n \,, \tag{7.3.51}$$

with a "generic constant $C > 0$" independent of n.

Error reduction by a factor $\rho > 1$ results from

$$\frac{Cq^{n_{\text{old}}}}{Cq^{n_{\text{new}}}} \overset{!}{=} \rho \quad \Leftrightarrow \quad \boxed{n_{\text{new}} - n_{\text{old}} = -\frac{\log \rho}{\log q}} \,.$$

In the case of *exponential convergence* (7.3.51) a fixed increase of the number of quadrature points by $-\log \rho : \log q$ results in a reduction of the quadrature error by a factor of $\rho > 1$.

7.4 Composite Quadrature

In 6, Section 6.5.1 we studied approximation by piecewise polynomial interpolants. A similar idea underlies the so-called composite quadrature rules on an interval $[a,b]$. Analogously to piecewise polynomial techniques they start from a grid/mesh

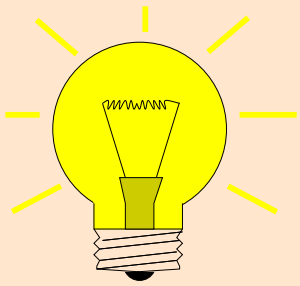
$$\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\} \tag{6.5.2}$$

and appeal to the trivial identity

$$\int_a^b f(t)\,\mathrm{d}t = \sum_{j=1}^{m} \int_{x_{j-1}}^{x_j} f(t)\,\mathrm{d}t \,. \tag{7.4.1}$$

On each mesh interval $[x_{j-1}, x_j]$ we then use a local quadrature rule, which may be one of the polynomial quadrature formulas from 7.2.

General construction of composite quadrature rules



Idea:
- Partition integration domain $[a,b]$ by a mesh/grid ($\rightarrow$ Section 6.5)
$$\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_m = b\}$$
- Apply quadrature formulas from Section 7.2, Section 7.3 locally on mesh intervals $I_j := [x_{j-1}, x_j]$, $j = 1, \ldots, m$, and sum up.

composite quadrature rule

Example 7.4.3 (Simple composite polynomial quadrature rules)

Composite trapezoidal rule, *cf.* (7.2.5)

$$\int_a^b f(t)\mathrm{d}t = \tfrac{1}{2}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \tfrac{1}{2}(x_{j+1} - x_{j-1})f(x_j) + \tfrac{1}{2}(x_m - x_{m-1})f(b)\ . \tag{7.4.4}$$

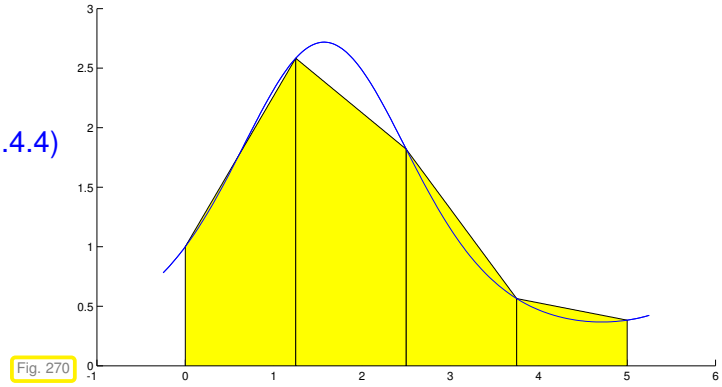


Fig. 270

➢ arising from piecewise linear interpolation of f.

Composite Simpson rule, *cf.* (7.2.6)

$$\int_a^b f(t)\mathrm{d}t = \tfrac{1}{6}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \tfrac{1}{6}(x_{j+1} - x_{j-1})f(x_j) + \sum_{j=1}^{m} \tfrac{2}{3}(x_j - x_{j-1})f(\tfrac{1}{2}(x_j + x_{j-1})) + \tfrac{1}{6}(x_m - x_{m-1})f(b)\ . \tag{7.4.5}$$

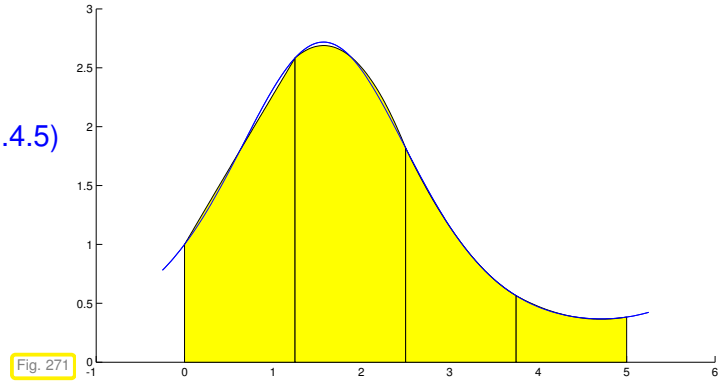


Fig. 271

➢

related to piecewise quadratic Lagrangian interpolation.

Formulas (7.4.4), (7.4.5) directly suggest efficient implementation with minimal number of f-evaluations.

C++-code 7.4.6: Equidistant composite trapezoidal rule (7.4.4)

```
1 template <class Function>
2 double trapezoidal(Function& f, const double a, const double b, const
    unsigned N) {
3   double I = 0;
4   const double h = (b - a)/N; // interval length
5
```

```
  for (unsigned i = 0; i < N; ++i) {
    // rule:  T = (b - a)/2 * (f(a) + f(b)),
    // apply on N intervals:  [a + i*h, a + (i+1)*h], i=0..(N-1)
    I += h/2*( f(a + i*h) + f(a + (i + 1)*h) );
  }
  return I;
}
```

C++-code 7.4.7: Equidistant composite Simpson rule (7.4.5)

```
template <class Function>
double simpson(Function& f, const double a, const double b, const
    unsigned N) {
  double I = 0;
  const double h = (b - a)/N; // intervall length

  for (unsigned i = 0; i < N; ++i) {
    // rule:  S = (b - a)/6*( f(a) + 4*f(0.5*(a + b)) + f(b) )
    // apply on [a + i*h, a + (i+1)*h]
    I += h/6*( f(a + i*h) + 4*f(a + (i+0.5)*h) + f(a + (i+1)*h) );
  }

  return I;
}
```

In both cases the function object passed in `f` must provide an evaluation operator **double operator** (**double**) **const**.

Remark 7.4.8 (Composite quadrature and piecewise polynomial interpolation)

Composite quadrature scheme based on local polynomial quadrature can usually be understood as "quadrature by approximation schemes" as explained in § 7.1.7. The underlying approximation schemes belong to the class of general local Lagrangian interpolation schemes introduced in Section 6.5.1.

In other words, many composite quadrature schemes arise from replacing the integrand by a piecewise interpolating polynomial, see Fig. 270 and Fig. 271 and compare with Fig. 250.

To see the main rationale behind the use of composite quadrature rules recall Lemma 7.3.42: for a polynomial quadrature rule (7.2.1) of order q with positive weights and $f \in C^r([a,b])$ the quadrature error shrinks with the $\min\{r,q\}+1$-st power of the length $|b-a|$ of the integration domain! Hence, applying polynomial quadrature rules to small mesh intervals should lead to a small overall quadrature error.

(7.4.9) Quadrature error estimate for composite polynomial quadrature rules

Assume a composite quadrature rule Q on $[x_0, x_m] = [a, b]$, $b > a$, based on n_j-point local quadrature rules $Q^j_{n_j}$ with *positive weights* (e.g. local Gauss-Legendre quadrature rules or local Clenshaw-Curtis quadrature rules) and of *fixed* orders $q_j \in \mathbb{N}$ on each mesh interval $[x_{j-1}, x_j]$. From Lemma 7.3.42 recall the estimate for $f \in C^r([x_{j-1}, x_j])$

$$\left| \int_{x_{j-1}}^{x_j} f(t)\,\mathrm{d}t - Q^j_{n_j}(f) \right| \leq C|x_j - x_{j-1}|^{\min\{r,q_j\}+1} \left\| f^{(\min\{r,q_j\})} \right\|_{L^\infty([x_{j-1},x_j])} . \tag{7.2.10}$$

with $C > 0$ independent of f and j.

For $f \in C^r([a, b])$, summing up these bounds we get for the global quadrature error

$$\left| \int_{x_0}^{x_m} f(t)\,\mathrm{d}t - Q(f) \right| \leq C \sum_{j=1}^{m} h_j^{\min\{r,q_j\}+1} \left\| f^{(\min\{r,q_j\})} \right\|_{L^\infty([x_{j-1},x_j])} ,$$

with local meshwidths $h_j = x_j - x_{j-1}$. If $q_j = q$, $q \in \mathbb{N}$, for all $j = 1, \ldots, m$, then, as $\sum_j h_j = b - a$,

$$\left| \int_{x_0}^{x_m} f(t)\,\mathrm{d}t - Q(f) \right| \leq C\, h_{\mathcal{M}}^{\min\{q,r\}}\, |b - a| \left\| f^{(\min\{q,r\})} \right\|_{L^\infty([a,b])} , \tag{7.4.10}$$

with (global) meshwidth $h_{\mathcal{M}} := \max_j h_j$.

(7.4.10) $\longleftrightarrow$ Algebraic convergence in no. of f-evaluations for $n \to \infty$

(7.4.11) Constructing families of composite quadrature rules

As with polynomial quadrature rules, we study the asymptotic behavior of the quadrature error for families of composite quadrature rules as a function on the total number n of function evaluations.

As in the case of $\mathcal{M}$-piecewise polynomial approximation of function ($\to$ Section 6.5.1) families of composite quadrature rules can be generated in two different ways:

(I) use a sequence of successively refined meshes $\left(\mathcal{M}_k = \{x_j^k\}_j\right)_{k\in\mathbb{N}}$ with $\sharp\mathcal{M} = m(k) + 1$, $m(k) \to \infty$ for $k \to \infty$, , combined with the *same* (transformed, $\to$ Rem. 7.1.4) local quadrature rule on all mesh intervals $[x_{j-1}^k, x_j^k]$. Examples are the composite trapezoidal rule and composite Simpson rule from Ex. 7.4.3 on sequences of equidistant meshes.

➢ h-convergence

(II) On a *fixed* mesh $\mathcal{M} = \{x_j\}_{j=0}^m$, on each cell use the same (transformed) local quadrature rule taken from a *sequence* of polynomial quadrature rules of increasing order.

➢ p-convergence

Example 7.4.12 (Quadrature errors for composite quadrature rules)

Composite quadrature rules based on

- trapezoidal rule (7.2.5) ➢ local order 2 (exact for linear functions, see Ex. 7.3.9),
- Simpson rule (7.2.6) ➢ local order 4 (exact for cubic polynomials, see Ex. 7.3.9)

on equidistant mesh $\mathcal{M} := \{jh\}_{j=0}^{n}$, $h = 1/n$, $n \in \mathbb{N}$.

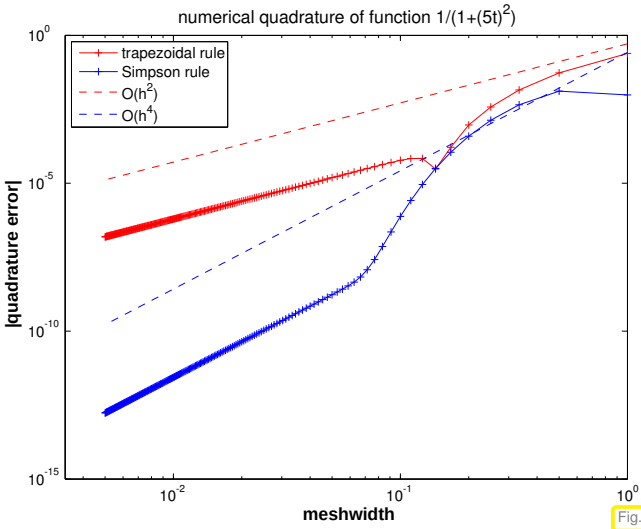


Fig. 272

quadrature error, $f_1(t) := \frac{1}{1+(5t)^2}$ on $[0,1]$

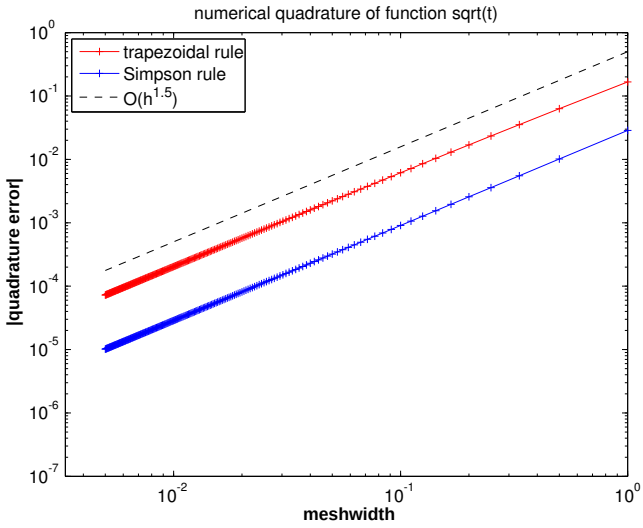


Fig. 273

quadrature error, $f_2(t) := \sqrt{t}$ on $[0,1]$

Asymptotic behavior of quadrature error $E(n) := \left| \int_0^1 f(t)\,\mathrm{d}t - Q_n(f) \right|$ for meshwidth "$h \to 0$"

☛ algebraic convergence $E(n) = O(h^\alpha)$ of order $\alpha > 0$, $n = h^{-1}$

➢ Sufficiently smooth integrand f_1: trapezoidal rule $\to \alpha = 2$, Simpson rule $\to \alpha = 4$!?

➢ singular integrand f_2: $\alpha = 3/2$ for trapezoidal rule **&** Simpson rule **!**

(lack of) smoothness of integrand limits convergence!

Remark 7.4.13 (Composite quadrature rules vs. global quadrature rules)

For a fixed integrand $f \in C^r([a,b])$ of limited smoothness on an interval $[a,b]$ we compare

- a family of composite quadrature rules based on single local ℓ-point rule (with positive weights) of order q on a sequence of equidistant meshes $\left(\mathcal{M}_k = \{x_j^k\}_j\right)_{k\in\mathbb{N}}$,
- the family of Gauss-Legendre quadrature rules from Def. 7.3.29.

We study the asymptotic dependence of the quadrature error on the number n of function evaluations.

For the composite quadrature rules we have $n \approx \ell \sharp \mathcal{M}_k \approx \ell h_{\mathcal{M}}^{-1}$. Combined with (7.4.10), we find for quadrature error $E_n^{\mathrm{comp}}(f)$ of the composite quadrature rules

$$E_n^{\mathrm{comp}}(f) \le C_1\, n^{-\min\{q,r\}}\,, \tag{7.4.14}$$

with $C_1 > 0$ independent of $\mathcal{M} = \mathcal{M}_k$.

The quadrature errors $E_n^{\mathrm{GL}}(f)$ of the n-point Gauss-Legendre quadrature rules are given in Lemma 7.3.42, (7.3.43):

$$E_n^{\mathrm{GL}}(f) \leq C_2\, n^{-r}\,, \tag{7.4.15}$$

with $C_2 > 0$ independent of n.

Gauss-Legendre quadrature converges *at least as fast* fixed order composite quadrature on equidistant meshes.

Moreover, Gauss-Legendre quadrature "automatically detects" the smoothness of the integrand, and enjoys fast exponential convergence for analytic integrands.

Use Gauss-Legendre quadrature instead of fixed order composite quadrature on equidistant meshes.

Experiment 7.4.16 (Empiric convergence of equidistant trapezoidal rule)

Sometimes there are surprises: Now we will witness a convergence behavior of a composite quadrature rule that is much better than predicted by the order of the local quadrature formula.

We consider the equidistant trapezoidal rule (order 2), see (7.4.4), Code 7.4.6

$$\int_a^b f(t)\,\mathrm{d}t \approx T_m(f) := h\left(\tfrac{1}{2}f(a) + \sum_{k=1}^{m-1} f(kh) + \tfrac{1}{2}f(b)\right)\,,\quad h := \frac{b-a}{m}\,. \tag{7.4.17}$$

and the 1-periodic smooth (analytic) integrand

$$f(t) = \frac{1}{\sqrt{1 - a\sin(2\pi t - 1)}}\,,\quad 0 < a < 1\,.$$

("exact value of integral": use T_{500})

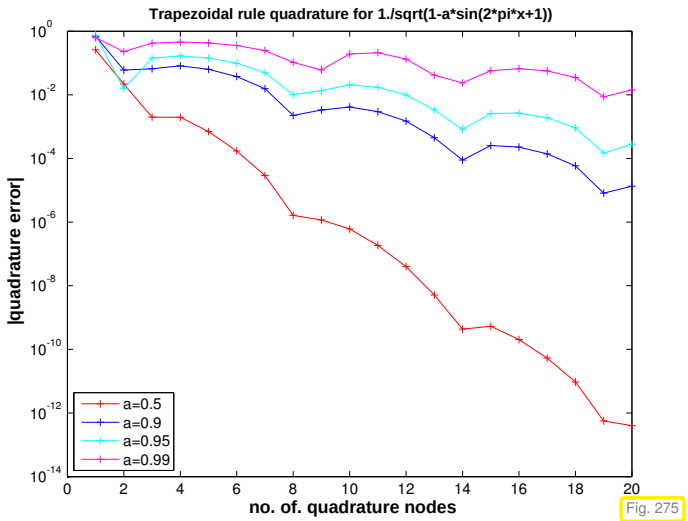


Fig. 274

quadrature error for $T_n(f)$ on $[0,1]$

exponential convergence !!

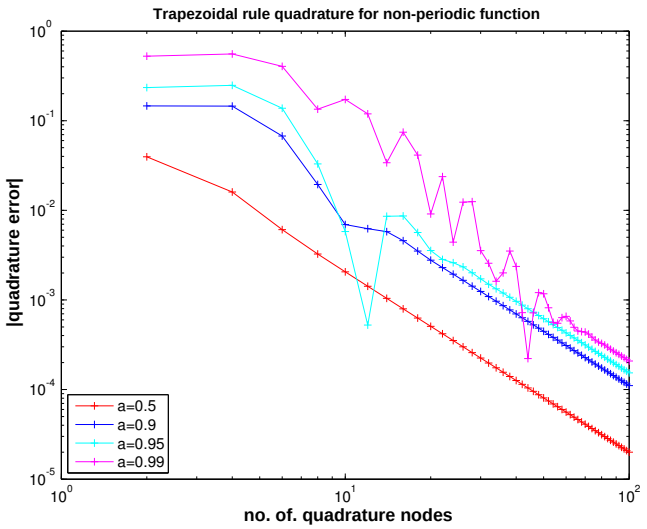


Fig. 275

quadrature error for $T_n(f)$ on $[0,\frac{1}{2}]$

merely algebraic convergence

(7.4.18) Equidistant trapezoidal rule (for periodic integrands)

In this § we use $I := [0,1[$ as a reference interval, *cf.* Exp. 7.4.16. We rely on similar techniques as in Section 5.6, Section 5.6.2. Again, a key tool will be the bijective mapping, see Fig. 203,

$$\Phi_{\mathbb{S}^1} : I \to \mathbb{S}^1 := \{z \in \mathbb{C} : |z| = 1\} \quad , \quad t \mapsto z := \exp(2\pi\imath t) \, , \tag{5.6.9}$$

which induces the general pullback, *c.f.* (6.1.20),

$$(\Phi_{\mathbb{S}^1}^{-1})^* : C^0([0,1[) \to C^0(\mathbb{S}^1) \quad , \quad ((\Phi_{\mathbb{S}^1}^{-1})^* f)(z) := f(\Phi_{\mathbb{S}^1}^{-1}(z)) \, , \quad z \in \mathbb{S}^1 \, .$$

If $f \in C^r(\mathbb{R})$ and 1-periodic, then $(\Phi_{\mathbb{S}^1}^{-1})^* f \in C^r(\mathbb{S}^1)$. Further, $\Phi_{\mathbb{S}^1}$ maps equidistant nodes on $I := [0,1]$ to equispaced nodes on $\mathbb{S}^1$, which are the roots of unity:

$$\Phi_{\mathbb{S}^1}(\tfrac{j}{n}) = \exp(2\pi\imath\tfrac{j}{n}) \quad [\, \exp(2\pi\imath\tfrac{j}{n})^n = 1; \,] \, . \tag{7.4.19}$$

Now consider an n-point polynomial quadrature rule on $\mathbb{S}^1$ based on the set of equidistant nodes $\mathcal{Z} := \{z_j := \exp(2\pi\imath\frac{j-1}{n}), j = 1,\ldots,n\}$ and defined as

$$Q_n^{\mathbb{S}^1}(g) := \int_{\mathbb{S}^1} (\mathsf{L}_{\mathcal{Z}} g)(\tau) \, \mathrm{d}S(\tau) = \sum_{j=1}^n w_j^{\mathbb{S}^1} g(z_j) \, , \tag{7.4.20}$$

where $\mathsf{L}_{\mathcal{Z}}$ is the Lagrange interpolation operator ($\to$ Def. 6.1.32). This means that the weights obey Thm. 7.3.5, where the definition (5.2.11) of Lagrange polynomials remains the same for complex nodes. By sheer *symmetry*, all the weights have to be the same, which, since the rule will be at least of order 1, means

$$w_j^{\mathbb{S}^1} = \frac{2\pi}{n} \, , \quad j = 1,\ldots,n \, .$$

Moreover, the quadrature rule $Q_n^{\mathbb{S}^1}$ will be of order n, see Def. 7.3.1, that is, it will *integrate polynomials of degree* $\leq n-1$, *exactly*.

By transformation ($\to$ Rem. 7.1.4 and pullback (7.4.18), $Q_n^{\mathbb{S}^1}$ induces a quadrature rule on $I := [0,1]$ by

$$Q_n^I(f) := \frac{1}{2\pi} Q_n^{\mathbb{S}^1}((\Phi_{\mathbb{S}^1}^{-1})^* f) = \frac{1}{2\pi} \sum_{j=1}^n w_j^{\mathbb{S}^1} f(\Phi^{-1}(z_j)) = \sum_{j=1}^n \frac{1}{n} f(\tfrac{j-1}{n}) \, . \tag{7.4.21}$$

This is exactly the equidistant trapezoidal rule(7.4.17), if f is 1-periodic, $f(0) = f(1)$: $Q_n^I = T_n$. Hence we arrive at the following estimate for the quadrature error

$$E_n(f) := \left| \int_0^1 f(t) \, \mathrm{d}t - T_n(f) \right| \leq 2\pi \max_{z \in \mathbb{S}^1} \left| ((\Phi_{\mathbb{S}^1}^{-1})^* f)(z) - (\mathsf{L}_{\mathcal{Z}}(\Phi_{\mathbb{S}^1}^{-1})^* f)(z) \right| \, .$$

Equivalently, one can show that T_n integrates trigonometric polynomials up to degree $2n-1$ exactly:

$$f(t) = e^{2\pi\imath k t} \quad \blacktriangleright \quad \begin{cases} \int_0^1 f(t) \, \mathrm{d}t = \begin{cases} 0 & \text{, if } k \neq 0 \, , \\ 1 & \text{, if } k = 0 \, . \end{cases} \\ T_n(f) = \frac{1}{m} \sum_{l=0}^{m-1} e^{\frac{2\pi\imath}{n} lk} \overset{(4.2.8)}{=} \begin{cases} 0 & \text{, if } k \notin n\mathbb{Z} \, , \\ 1 & \text{, if } k \in n\mathbb{Z} \, . \end{cases} \end{cases}$$

The equidistant trapezoidal rule T_n is exact for trigonometric polynomials of degree $< 2n$!

Remark 7.4.22 (Approximate computation of Fourier coefficients)

Recall from Section 4.2.5: recovery of signal $(y_k)_{k\in\mathbb{Z}}$ from its Fourier transform $c(t)$

$$y_j = \int_0^1 c(t)\exp(2\pi ijt)\,\mathrm{d}t\ . \tag{4.2.79}$$

Task: approximate computation of y_j

Recall: $c(t)$ obtained from $(y_k)_{k\in\mathbb{Z}}$ through Fourier series

$$c(t) = \sum_{k\in\mathbb{Z}} y_k \exp(-2\pi ikt)\ . \tag{4.2.67}$$

➢ $c(t)$ smooth & 1-periodic for finite/rapidly decaying $(y_k)_{k\in\mathbb{Z}}$.

Exp. 7.4.16 use equidistant trapezoidal rule (7.4.17) for approximate evaluation of integral in (4.2.79).

☞ Boils down to inverse DFT (4.2.20); hardly surprising in light of the derivation of (4.2.79) in Section 4.2.5.

C++-code 7.4.23: DFT-based approximate computation of Fourier coefficients

```
#include <iostream>
#include <vector>
#include <complex>
#include <unsupported/Eigen/FFT>

template <class Function>
void fourcoeffcomp(std::vector<std::complex<double>>& y, Function& c,
    const unsigned m, const unsigned ovsmpl = 2) {
  // Compute the Fourier coefficients y_{-m},...,y_m of the function
  // c : [0,1[ → C using an oversampling factor ovsmpl.
  // c must be a handle to a function @(t), e.g.  a lambda function
  const unsigned N = (2*m + 1)*ovsmpl; // number of quadrature points
  const double h = 1./N;

  // evaluate function in N points
  std::vector<std::complex<double>> c_eval(N);
  for (unsigned i = 0; i < N; ++i) {
    c_eval[i] = c(i*h);
  }

  // inverse discrete fourier transformation
  Eigen::FFT<double> fft;
  std::vector<std::complex<double>> z;
```

```
23    fft.inv(z, c_eval);
24
25    // Undo oversampling and wrapping of Fourier coefficient array
26    // -> y contains same values as z but in a different order:
27    // y = [z(N-m+1:N), z(1:m+1)]
28    y = std::vector<std::complex<double>>();
29    y.reserve(N);
30    for (unsigned i = N - m; i < N; ++i) {
31      y.push_back(z[i]);
32    }
33    for (unsigned j = 0; j < m + 1; ++j) {
34      y.push_back(z[j]);
35    }
36  }
```

7.5 Adaptive Quadrature

Hitherto, we have just "blindly" applied quadrature rules for the approximate evaluation of $\int_a^b f(t)\,\mathrm{d}t$, oblivious of any properties of the integrand f. This led us to the conclusion of Rem. 7.4.13 that Gauss-Legendre quadrature ($\rightarrow$ Def. 7.3.29) should be preferred to composite quadrature rules ($\rightarrow$ Section 7.4) in general. Now the composite quadrature rule will partly be rehabilitated, because they offer the flexibility to *adjust the quadrature rule to the integrand*, a policy known as adaptive quadrature.

Adaptive numerical quadrature

The policy of adaptive quadrature approximates $\int_a^b f(t)\,\mathrm{d}t$ by a quadrature formula (7.1.2), whose nodes c_j^n are chosen depending on the integrand f.

We distinguish

(I) a priori adaptive quadrature: the nodes a *fixed* before the evaluation of the quadrature formula, taking into account external information about f, and

(II) a posteriori adaptive quadrature: the node positions are chosen or improved based on information gleaned *during the computation* inside a loop. It terminates when sufficient accuracy has been reached.

In this section we will chiefly discuss a posteriori adaptive quadrature for composite quadrature rules ($\rightarrow$ Section 7.4) based on a single local quadrature rule (and its transformation).

Supplementary reading. [**?**, Sect. 9.7]

Example 7.5.2 (Rationale for adaptive quadrature)

This example presents an extreme case. We consider the composite trapezoidal rule (7.4.4) on a mesh $\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$ and for the integrand $f(t) = \frac{1}{10^{-4}+t^2}$ on $[-1,1]$.

f is a spike-like function ▷

Intuition: quadrature nodes should cluster around 0, whereas hardly any are needed close to the endpoints of the integration interval, where the function has very small (in modulus) values.

➢ Use locally refined mesh !

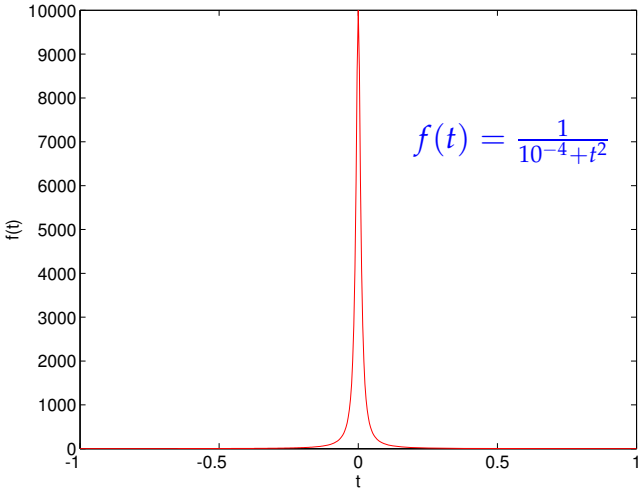


Fig. 276

A quantitative justification can appeal to (7.2.10) and the resulting bound for the local quadrature error (for $f \in C^2([a,b])$):

$$\int_{x_{k-1}}^{x_k} f(t)\,\mathrm{d}t - \frac{1}{2}(f(x_{k-1}) + f(x_k)) \le h_k^3 \|f''\|_{L^\infty([x_{k-1},x_k])}\,, \quad h_k := x_k - x_{k-1}\,. \tag{7.5.3}$$

➢ Suggests the use of small mesh intervals, where $|f''|$ is large **!**

(7.5.4) Goal: equidistribution of errors

The ultimate but elusive goal is to find a mesh with a minimal number of cells that just delivers a quadrature error below a prescribed threshold. A more practical goal is to adjust the local meshwidths $h_k := x_k - x_{k-1}$ in order to *achieve a minimal sum of local error bounds*. This leads to the constrained minimization problem:

$$\sum_{k=1}^m h_k^3 \|f''\|_{L^\infty([x_{k-1},x_k])} \to \min \quad \text{s.t.} \quad \sum_{k=1}^m h_k = b - a\,. \tag{7.5.5}$$

Lemma 7.5.6.

Let $f : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ be a convex function with $f(0) = 0$ and $x > 0$. Then the constrained minimization problem: seek $\zeta_1, \ldots, \zeta_m \in \mathbb{R}_0^+$ such that

$$\sum_{k=1}^m f(\zeta_k) \to \min \quad \textit{and} \quad \sum_{k=1}^m \zeta_k = x\,, \tag{7.5.7}$$

has the solution $\zeta_1 = \zeta_2 = \cdots = \zeta_m = \frac{x}{m}$.

This means that we should strive for equal bounds $h_k^3 \|f''\|_{L^\infty([x_{k-1},x_k])}$ for all mesh cells.

Error equidistribution principle

The mesh for *a posteriori adaptive composite numerical quadrature* should be chosen to achieve equal contributions of all mesh intervals to the quadrature error

---

A indicated above, guided by the equidistribution principle, the improvement of the mesh will be done gradually in an iteration. The change of the mesh in each step is called mesh adaptation and there are two fundamentally different ways to do it:

(I) by moving nodes, keeping their total number, but making them cluster where mesh intervals should be small, or

(II) by adding nodes, where mesh intervals should be small (mesh refinement).

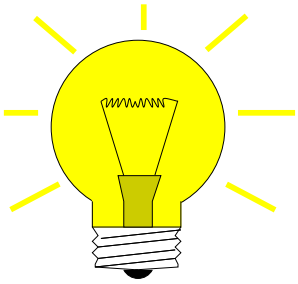
Algorithms for a posteriori adaptive quadrature based on mesh refinement usually have the following structure:

Adaptation loop for numerical quadrature

(1) ESTIMATE: based on available information compute an approximation for the quadrature error on every mesh interval.
(2) CHECK TERMINATION: if total error sufficient small $\rightarrow$ **STOP**
(3) MARK: single out mesh intervals with the largest or above average error contributions.
(4) REFINE: add node(s) inside the marked mesh intervals. GOTO (1)

(7.5.10) Adaptive multilevel quadrature

We now see a concrete algorithm based on the two composite quadrature rules introduced in Ex. 7.4.3.



Idea: local error estimation by comparing local results of two quadrature formulas Q_1, Q_2 of *different* order $\rightarrow$ local error estimates

heuristics: $\text{error}(Q_2) \ll \text{error}(Q_1) \quad \Rightarrow \quad \text{error}(Q_1) \approx Q_2(f) - Q_1(f)$.

Here: Q_1 = trapezoidal rule (order 2) $\leftrightarrow$ Q_2 = Simpson rule (order 4)

Given: initial mesh $\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$

❶ (Error estimation)

For $I_k = [x_{k-1}, x_k]$, $k = 1, \ldots, m$ (midpoints $p_k := \frac{1}{2}(x_{k-1} + x_k)$)

$$\text{EST}_k := \Big| \underbrace{\frac{h_k}{6}(f(x_{k-1}) + 4f(p_k) + f(x_k))}_{\text{Simpson rule}} - \underbrace{\frac{h_k}{4}(f(x_{k-1}) + 2f(p_k) + f(x_k))}_{\text{trapezoidal rule on split mesh interval}} \Big| . \tag{7.5.11}$$

❷ (Check termination)

Simpson rule on $\mathcal{M}$ $\Rightarrow$ intermediate approximation $I \approx \int_a^b f(t)\,\mathrm{d}t$

$$\text{If} \quad \sum_{k=1}^{m} \mathtt{EST}_k \leq \mathtt{RTOL} \cdot I \quad (RTOL := \text{prescribed relative tolerance}) \quad \Rightarrow \quad \textbf{STOP} \tag{7.5.12}$$

❸ (Marking)

$$\text{Marked intervals:} \quad \mathcal{S} := \{k \in \{1,\ldots,m\}\colon \mathtt{EST}_k \geq \eta \cdot \frac{1}{m} \sum_{j=1}^{m} \mathtt{EST}_j\}\,, \quad \eta \approx 0.9\,. \tag{7.5.13}$$

❹ (Local mesh refinement)

$$\text{new mesh:} \quad \mathcal{M}^* := \mathcal{M} \cup \{p_k := \frac{1}{2}(x_{k-1} + x_k)\colon k \in \mathcal{S}\}\,. \tag{7.5.14}$$

Then continue with step ❶ and mesh $\mathcal{M} \leftarrow \mathcal{M}^*$.

The following code give a non-optimal recursive MATLAB implementation:

C++-code 7.5.15: h-adaptive numerical quadrature

```
// Adaptive multilevel quadrature of a function passed in f.
// The vector M passes the positions of current quadrature nodes
template <class Function>
double adaptquad(Function& f, VectorXd& M,double rtol,double atol) {
  const std::size_t n = M.size(); // number of nodes
  VectorXd h = M.tail(n-1)-M.head(n-1), // distance of quadature nodes
           mp = 0.5*(M.head(n-1)+M.tail(n-1)); // midpoints
  // Values of integrand at nodes and midpoints
  VectorXd fx(n), fm(n - 1);
  for (unsigned i = 0; i < n; ++i) fx(i) = f(M(i));
  for (unsigned j = 0; j < n - 1; ++j) fm(j) = f(mp(j));
  // trapezoidal rule (7.4.4)
  const VectorXd trp_loc =
     1./4*h.cwiseProduct(fx.head(n-1)+2*fm+fx.tail(n-1));
  // Simpson rule (7.4.5)
  const VectorXd simp_loc =
     1./6*h.cwiseProduct(fx.head(n-1)+4*fm+fx.tail(n-1));

  // Simpson approximation for the integral value
  double I = simp_loc.sum();
  // local error estimate (7.5.11)
  const VectorXd est_loc = (simp_loc-trp_loc).cwiseAbs();
  // estimate for quadrature error
  const double err_tot = est_loc.sum();

  // STOP: Termination based on (7.5.12)
  if ( err_tot > rtol*std::abs(I) && err_tot > atol ) { //
    // find cells where error is large
    std::vector<double> new_cells;
    for (unsigned i = 0; i < est_loc.size(); ++i) {
      // MARK by criterion (7.5.13) & REFINE by (7.5.14)
      if (est_loc(i) > 0.9/(n-1)*err_tot) {
        // new quadrature point = midpoint of interval with large error
        new_cells.push_back(mp(i));
      }}

    // create new set of quadrature nodes
    // (necessary to convert std::vector to Eigen vector)
    Eigen::Map<VectorXd> tmp(new_cells.data(), new_cells.size());
    VectorXd new_M(M.size() + tmp.size());
    new_M << M, tmp; // concatenate old cells and new cells
    // nodes of a mesh are supposed to be sorted
    std::sort(new_M.data(),new_M.data()+new_M.size());
    I = adaptquad(f, new_M, rtol, atol); // recursion
  }
  return I;
}
```

Comments on Code 7.5.15:

- Arguments: `f` ≙ *handle* to function f, `M` ≙ initial mesh, `rtol` ≙ relative tolerance for termination, `atol` ≙ absolute tolerance for termination, necessary in case the exact integral value $= 0$, which renders a relative tolerance meaningless.
- Line 7: compute lengths of mesh-intervals $[x_{j-1}, x_j]$,
- Line 8: store positions of midpoints p_j,
- Line 9: evaluate function (vector arguments!),
- Line 13: local composite trapezoidal rule (7.4.4),
- Line 15: local simpson rule (7.2.6),
- Line 18: value obtained from composite simpson rule is used as intermediate approximation for integral value,
- Line 20: difference of values obtained from local composite trapezoidal rule ($\sim Q_1$) and local simpson rule ($\sim Q_2$) is used as an estimate for the local quadrature error.
- Line 22: estimate for global error by summing up moduli of local error contributions,
- Line 26: terminate, once the estimated total error is below the relative or absolute error threshold,
- Line 43 otherwise, add midpoints of mesh intervals with large error contributions according to (7.5.14) to the mesh and continue.

C++-code 7.5.16: Call of `adaptquad()`:

```
1  # include "./adaptquad.hpp"
2  # include <iostream>
3  # include <cmath>
4
5  int main () {
6    auto f =[](double x) { return std::exp(-x*x); };
7    VectorXd M(4);
8    M << -100, 0.1, 0.5, 100;
9    std::cout << "Sqrt(Pi) - Int_{-100}^{100} exp(-x*x) dx = ";
10   std::cout << adaptquad(f, M, 1e-10, 1e-12) - std::sqrt(M_PI) << "\n";
11   return 0;
12 }
```

Remark 7.5.17 (Estimation of "wrong quadrature error"?)

In Code 7.5.15 we use the higher order quadrature rule, the Simpson rule of order 4, to compute an approximate value for the integral. This is reasonable, because it would be foolish not to use this information after we have collected it for the sake of error estimation.

Yet, according to our heuristics, what `est_loc` and `est_tot` give us are estimates for the error of the second-order trapezoidal rule, which we do not use for the actual computations.

However, experience teaches that

`est_loc` gives useful (for the sake of mesh refinement) information about the *distribution* of the error of the Simpson rule, though it fails to capture its *size*.

Therefore, the termination criterion of Line 26 may not be appropriate!

Experiment 7.5.18 (h-adaptive numerical quadrature)

In this numerical test we investigate whether the adaptive technique from § 7.5.10 produces an appropriate distribution of integration nodes. We do this for different functions.

✦ approximate $\int_0^1 \exp(6\sin(2\pi t))\,\mathrm{d}t$, initial mesh $\mathcal{M}_0 = \{j/10\}_{j=0}^{10}$

Algorithm: adaptive quadrature, Code 7.5.15 with tolerances `rtol` $= 10^{-6}$, `abstol` $= 10^{-10}$

We monitor the distribution of quadrature points during the adaptive quadrature and the true and estimated quadrature errors. The "exact" value for the integral is computed by composite Simpson rule on an equidistant mesh with 10^7 intervals.

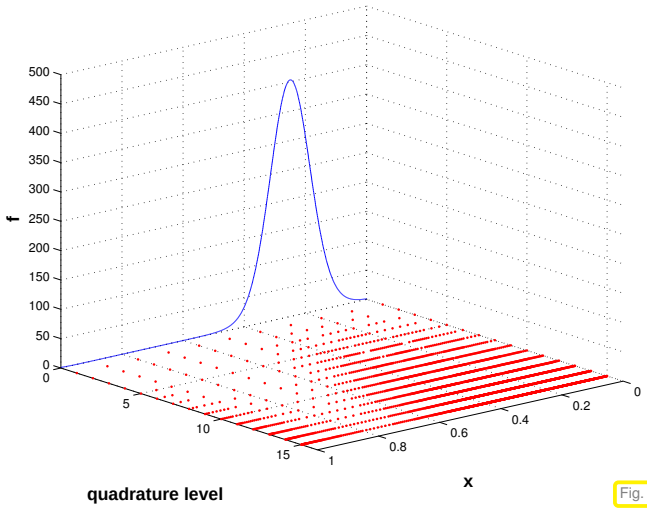


Fig. 277

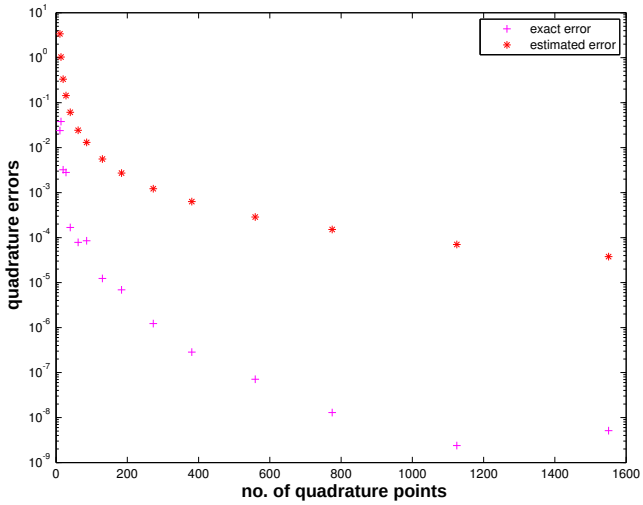


Fig. 278

✦ approximate $\int_0^1 \min\{\exp(6\sin(2\pi t)), 100\}\,\mathrm{d}t$, initial mesh as above

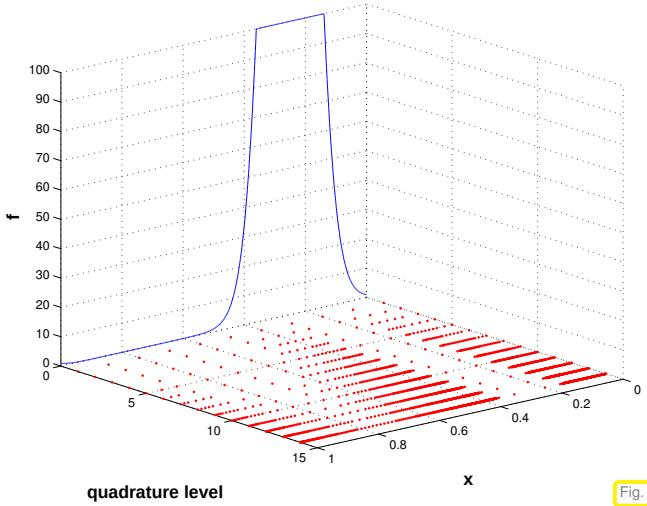


Fig. 279

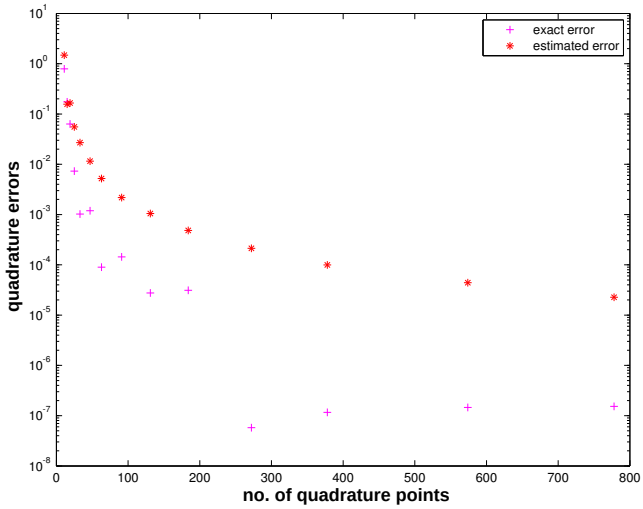


Fig. 280

Observation:

- Adaptive quadrature locally decreases meshwidth where integrand features variations or kinks.

- Trend for estimated error mirrors behavior of true error.
- Overestimation may be due to taking the modulus in (7.5.11)

However, the important piece of information we want to extract from EST_k is about the *distribution* of the quadrature error.

Remark 7.5.19 (Adaptive quadrature in MATLAB)

`q = quad(fun,a,b,tol)`:	adaptive multigrid quadrature (local low order quadrature formulas)
`q = quadl(fun,a,b,tol)`:	adaptive Gauss-Lobatto quadrature

Learning Outcomes

- You should know what is a quadrature formula and terminology connected with it,
- You should be able to transform quadrature formulas to arbitrary intervals.
- You should understand how a interpolation and approximation schemes spawn quadrature formulas and how quadrature errors are connected to interpolation/approximation errors.
- You should be able to compute the weights of polynomial quadrature formulas.
- You should know the concept of order of a quadrature rule and why it is invariant under (affine) transformation
- You should remember the maximal and minimal order of polynomial quadrature rules.
- You should know the order of the n-point Gauss-Legendre quadrature rule.
- You should understand why Gauss-Legendre quadrature converges exponentially for integrands that can be extended analytically and algebraically for integrands with limited smoothness.
- You should be apply to apply regularizing transformations to integrals with non-smooth integrands.
- You should know about asymptotic convergence of the h-version of composite quadrature.
- You should know the principles of adaptive composite quadrature.

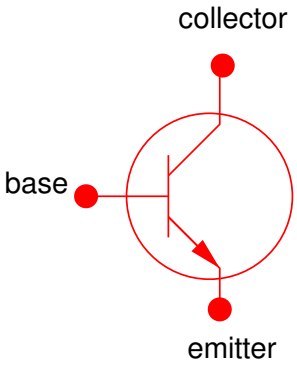
Chapter 8

Iterative Methods for Non-Linear Systems of Equations

Example 8.0.1 (Non-linear electric circuit)

Non-linear systems naturally arise in mathematical models of electrical circuits, once non-linear circuit elements are introduced. This generalizes Ex. 2.1.3, where the current-voltage relationship for all circuit elements was the simple proportionality (2.1.5) (of the complex amplitudes U and I).

As an example we consider the
Schmitt trigger circuit ▷
Its key non-linear circuit element is the NPN bipolar junction transistor:



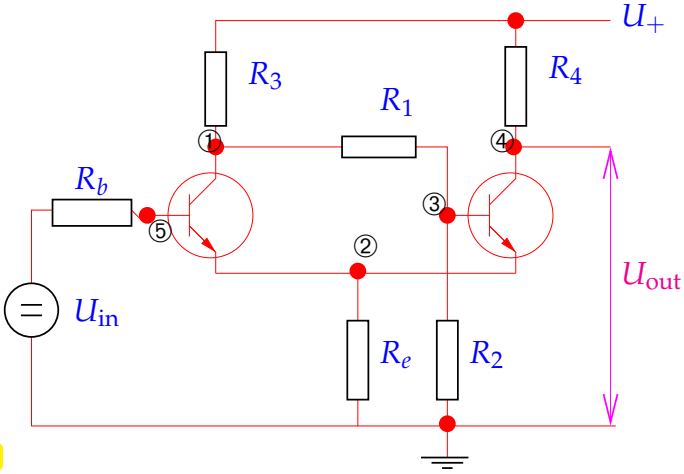


Fig. 281

A transistor has three ports: emitter, collector, and base. Transistor models give the port currents as functions of the applied voltages, for instance the Ebers-Moll model (large signal approximation):

$$\begin{aligned} I_{\mathrm{C}} &= I_{\mathrm{S}}\left(e^{\frac{U_{\mathrm{BE}}}{U_{\mathrm{T}}}} - e^{\frac{U_{\mathrm{BC}}}{U_{\mathrm{T}}}}\right) - \frac{I_{\mathrm{S}}}{\beta_R}\left(e^{\frac{U_{\mathrm{BC}}}{U_{\mathrm{T}}}} - 1\right) = I_{\mathrm{C}}(U_{\mathrm{BE}}, U_{\mathrm{BC}})\,, \\ I_{\mathrm{B}} &= \frac{I_{\mathrm{S}}}{\beta_F}\left(e^{\frac{U_{\mathrm{BE}}}{U_{\mathrm{T}}}} - 1\right) + \frac{I_{\mathrm{S}}}{\beta_R}\left(e^{\frac{U_{\mathrm{BC}}}{U_{\mathrm{T}}}} - 1\right) = I_{\mathrm{B}}(U_{\mathrm{BE}}, U_{\mathrm{BC}})\,, \\ I_{\mathrm{E}} &= I_{\mathrm{S}}\left(e^{\frac{U_{\mathrm{BE}}}{U_{\mathrm{T}}}} - e^{\frac{U_{\mathrm{BC}}}{U_{\mathrm{T}}}}\right) + \frac{I_{\mathrm{S}}}{\beta_F}\left(e^{\frac{U_{\mathrm{BE}}}{U_{\mathrm{T}}}} - 1\right) = I_{\mathrm{E}}(U_{\mathrm{BE}}, U_{\mathrm{BC}})\,. \end{aligned} \tag{8.0.2}$$

I_{C}, I_{B}, I_{E}: current in collector/base/emitter,
U_{BE}, U_{BC}: potential drop between base-emitter, base-collector.

The parameters have the following meanings: β_F is the forward common emitter current gain (20 to 500), β_R is the reverse common emitter current gain (0 to 20), I_S is the reverse saturation current (on the order of 10^{-15} to 10^{-12} amperes), U_T is the thermal voltage (approximately 26 mV at 300 K).

The circuit of Fig. 281 has 5 nodes ①–⑤ with unknown nodal potentials. Kirchhoffs law (2.1.4) plus the constitutive relations gives an equation for each of them.

Non-linear system of equations from nodal analysis, static case (→ Ex. 2.1.3):

$$\begin{aligned}
① &: \quad R_3(U_1 - U_+) + R_1(U_1 - U_3) + I_B(U_5 - U_1, U_5 - U_2) &&= 0\,,\\
② &: \quad R_e U_2 + I_E(U_5 - U_1, U_5 - U_2) + I_E(U_3 - U_4, U_3 - U_2) &&= 0\,,\\
③ &: \quad R_1(U_3 - U_1) + I_B(U_3 - U_4, U_3 - U_2) &&= 0\,,\\
④ &: \quad R_4(U_4 - U_+) + I_C(U_3 - U_4, U_3 - U_2) &&= 0\,,\\
⑤ &: \quad R_b(U_5 - U_{\text{in}}) + I_B(U_5 - U_1, U_5 - U_2) &&= 0\,.
\end{aligned} \tag{8.0.3}$$

5 equations ↔ 5 unknowns U_1, U_2, U_3, U_4, U_5

Formally: (8.0.3) ⟷ $F(\mathbf{u}) = 0$ with a function $F : \mathbb{R}^5 \to \mathbb{R}^5$

Remark 8.0.4 (*General* non-linear systems of equations)

A non-linear system of equations is a concept almost *too abstract to be useful*, because it covers an extremely wide variety of problems . Nevertheless in this chapter we will mainly look at "generic" methods for such systems. This means that every method discussed may take a good deal of fine-tuning before it will really perform satisfactorily for a given non-linear system of equations.

(8.0.5) Generic/general non-linear system of equations

Given: function $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$, $n \in \mathbb{N}$

⇕

Possible meanings: ☞ F is known as an analytic expression.

☞ F is merely available in procedural form allowing point evaluations.

Here, D is the domain of definition of the function F, which cannot be evaluated for $\mathbf{x} \notin D$.

Sought: solution(s) $\mathbf{x} \in D$ of non-linear equation $F(\mathbf{x}) = 0$

Note: $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ ↔ "same number of equations and unknowns"

In contrast to the situation for linear systems of equations (→ Thm. 2.2.4), the class of non-linear systems is far too big to allow a general theory:

There are no general results existence & uniqueness of solutions of $F(\mathbf{x}) = 0$

Contents

8.1 Iterative methods

Remark 8.1.1 (Necessity of iterative approximation)

Gaussian elimination (→ Section 2.3) provides an algorithm that, if carried out in exact arithmetic (no roundoff errors), computes the solution of a linear system of equations with a *finite* number of elementary operations. However, linear systems of equations represent an exceptional case, because it is hardly ever possible to solve general systems of non-linear equations using only finitely many elementary operations.

(8.1.2) Generic iterations

All methods for general non-lineare systems of equations are iterative in the sense that they will usually

yield only approximate solutions whenever they terminate after finite time.

An iterative method for (approximately) solving the non-linear equation $F(\mathbf{x}) = 0$ is an algorithm generating an arbitrarily long sequence $\left(\mathbf{x}^{(k)}\right)_k$ of approximate solutions.

$\mathbf{x}^{(k)} \triangleq k$-th iterate

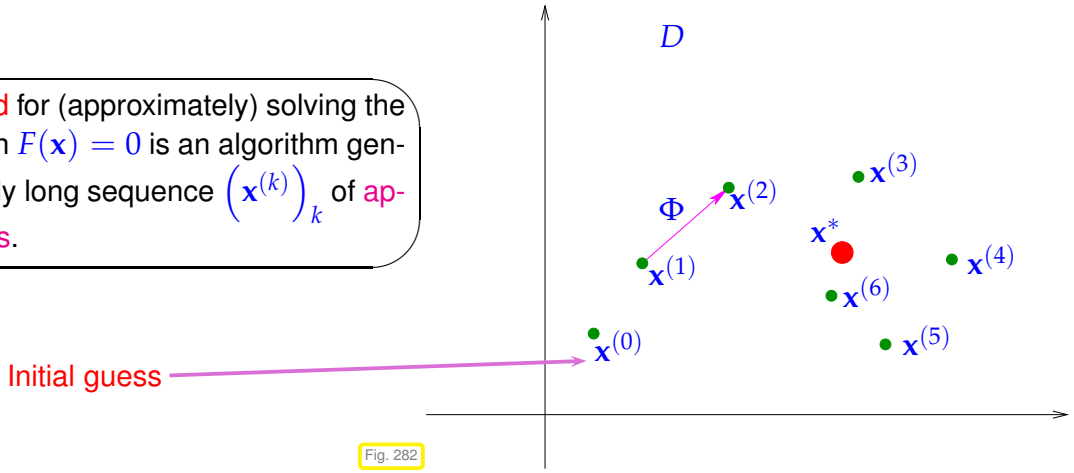


Fig. 282

(8.1.3) (Stationary) m-point iterative method

All the iterative methods discussed below fall in the class of (stationary) m-point, $m \in \mathbb{N}$, iterative methods, for which the iterate $\mathbf{x}^{(k)}$ depends on F and the m most recent iterates $\mathbf{x}^{(k-1)}, \ldots, \mathbf{x}^{(k-m)}$, e.g.,

$$\mathbf{x}^{(k)} = \underbrace{\Phi_F(\mathbf{x}^{(k-1)}, \ldots, \mathbf{x}^{(k-m)})}_{\text{iteration function for } m\text{-point method}} \tag{8.1.4}$$

Terminology: Φ_F is called the iteration function.

Note: The initial guess(es) $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(m-1)} \in \mathbb{R}^n$ have to be provided.

(8.1.5) Key issues with iterative methods

When applying an iterative method to solve a non-linear system of equations $F(\mathbf{x}) = 0$, the following issues arise:

- Convergence: Does the sequence $(\mathbf{x}^{(k)})_k$ converge to a limit: $\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}^*$?
- Consistency: Does the limit, if it exists, provide a solution of the non-linear system of equations: $F(\mathbf{x}^*) = 0$?
- Speed of convergence: How "fast" does $\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|$ ($\|\cdot\|$ a suitable norm on $\mathbb{R}^N$) decrease for increasing k?

More formal definitions can be given:

Definition 8.1.6. Convergence of iterative methods

An iterative method converges (for fixed initial guess(es)) $:\Leftrightarrow$ $\mathbf{x}^{(k)} \overset{k\to\infty}{\to} \mathbf{x}^*$ and $F(\mathbf{x}^*) = 0$.

Definition 8.1.7. Consistency of iterative methods

A statioanry m-point iterative method is consistent with the non-linear system of equations $F(\mathbf{x}) = 0$

$$:\Leftrightarrow \quad \Phi_F(\mathbf{x}^*,\dots,\mathbf{x}^*) = \mathbf{x}^* \quad \Leftrightarrow \quad F(\mathbf{x}^*) = 0$$

For a consistent stationary iterative method we can study the error of the iterates $\mathbf{x}^{(k)}$ defined as: $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}^*$

Unfortunately, convergence may critically depend on the choice of initial guesses. The property defined next weakens this dependence:

Definition 8.1.8. Local and global convergence $\rightarrow$ **[?, Def. 17.1]**

As stationary m-point iterative method converges locally to $\mathbf{x}^* \in \mathbb{R}^n$, if there is a neighborhood $U \subset D$ of $\mathbf{x}^*$, such that

$$\mathbf{x}^{(0)},\dots,\mathbf{x}^{(m-1)} \in U \quad \Rightarrow \quad \mathbf{x}^{(k)} \text{ well defined} \quad \wedge \quad \lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}^*$$

where $(\mathbf{x}^{(k)})_{k\in\mathbb{N}_0}$ is the (infinite) sequence of iterates.
If $U = D$, the iterative method is globally convergent.

Illustration of local convergence ▷

(Only initial guesses "sufficiently close" to $\mathbf{x}^*$ guarantee convergence.)

Unfortunately, the neighborhood U is rarely known a priori. It may also be very small.

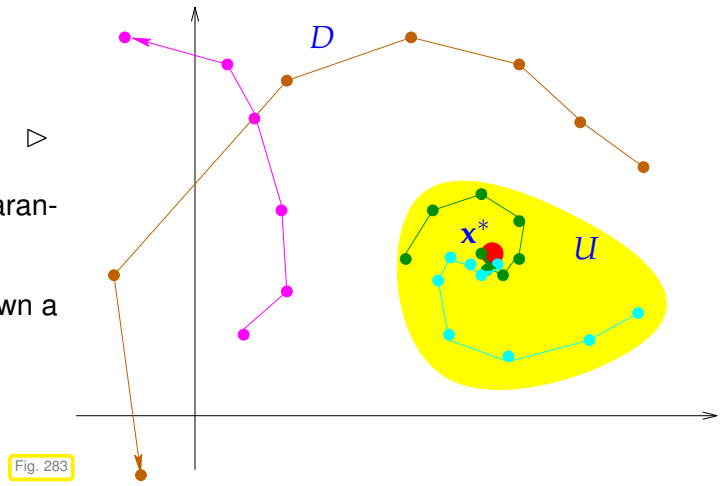


Fig. 283

Our goal: Given a non-linear system of equations, find iterative methods that converge (locally) to a solution of $F(\mathbf{x}) = 0$.

Two general questions: How to measure, describe, and predict the speed of convergence?
When to terminate the iteration?

8.1.1 Speed of convergence

Here and in the sequel, $\|\cdot\|$ designates a generic vector norm on $\mathbb{R}^n$, see Def. 1.5.70. Any occurring matrix norm is induced by this vector norm, see Def. 1.5.76.

It is important to be aware which statements depend on the choice of norm and which do not!

"*Speed of convergence*" $\leftrightarrow$ decrease of norm (see Def. 1.5.70) of iteration error

Definition 8.1.9. Linear convergence

A sequence $\mathbf{x}^{(k)}$, $k = 0, 1, 2, \ldots$, in $\mathbb{R}^n$ converges linearly to $\mathbf{x}^* \in \mathbb{R}^n$,

$$\exists 0 < L < 1: \quad \left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\| \leq L \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \quad \forall k \in \mathbb{N}_0 \, .$$

Terminology: least upper bound for L gives the rate of convergence

Remark 8.1.10 (Impact of choice of norm)

Fact of convergence of iteration is	independent	of choice of norm
Fact of linear convergence	depends	on choice of norm
Rate of linear convergence	depends	on choice of norm

The first statement is a consequence of the equivalence of all norms on the finite dimensional vector space $\mathbb{K}^n$:

Definition 8.1.11. Equivalence of norms

Two norms $\|\cdot\|_a$ and $\|\cdot\|_b$ on a vector space V are equivalent if

$$\exists \underline{C}, \overline{C} > 0: \quad \underline{C}\|v\|_a \leq \|v\|_b \leq \overline{C}\|v\|_a \quad \forall v \in V \, .$$

Theorem 8.1.12. Equivalence of all norms on finite dimensional vector spaces

If $\dim V < \infty$ all norms ($\rightarrow$ Def. 1.5.70) on V are equivalent ($\rightarrow$ Def. 8.1.11).

Remark 8.1.13 (Detecting linear convergence)

Often we will study the behavior of a consistent iterative method for a model problem in a numerical experiments and measure the norms of the iteration errors $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}^*$. How can we tell that the method enjoys linear convergence?

norms of iteration errors

$\updownarrow$

$\sim$ straight line in lin-log plot

$$\left\|\mathbf{e}^{(k)}\right\| \leq L^k \left\|\mathbf{e}^{(0)}\right\| \, ,$$

$$\log\left(\left\|\mathbf{e}^{(k)}\right\|\right) \leq k \log(L) + \log\left(\left\|\mathbf{e}^{(0)}\right\|\right) \, .$$

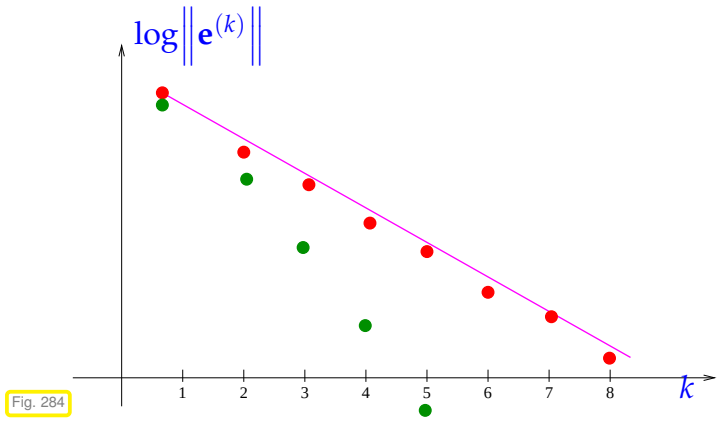


Fig. 284

Let us abbreviate the error norm in step k by $\epsilon_k := \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|$. In the case of linear convergence (see Def. 8.1.9) assume (with $0 < L < 1$)

$$\epsilon_{k+1} \approx L\epsilon_k \quad \Rightarrow \quad \log \epsilon_{k+1} \approx \log L + \log \epsilon_k \quad \Rightarrow \quad \log \epsilon_k \approx k \log L + \log \epsilon_0 \,. \tag{8.1.14}$$

We conclude that $\log L < 0$ determines the slope of the graph in lin-log error chart.

Related: guessing time complexity $O(n^\alpha)$ of an algorithm from measurements, see § 1.4.9.

Note the green dots • in Fig. 284: Any "faster" convergence also qualifies as linear convergence in the strict sense of the definition. However, whenever this term is used, we tacitly imply, that no "faster convergence" prevails.

Example 8.1.15 (Linearly convergent iteration)

C++11 code 8.1.16: Simple fixed point iteration in 1D

```
2  void fpit(double x0, VectorXd &rates,
       VectorXd &err)
3  {
4    const unsigned int N = 15;
5    double x = x0; // initial guess
6    VectorXd y(N);
7
8    for (int i=0; i<N; ++i) {
9      x = x + (cos(x)+1)/sin(x);
10     y(i) = x;
11   }
12   err.resize(N); rates.resize(N);
13   err = y-VectorXd::Constant(N,x);
14   rates =
       err.bottomRows(N-1).cwiseQuotient(err.topRo
15 }
```

We consider the iteration ($n = 1$):

$$x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)} + 1}{\sin x^{(k)}} \,.$$

In the C++ code (▷) x has to be initialized with the different values for x_0.

Note: The final iterate $x^{(15)}$ replaces the exact solution x^* in the computation of the rate of convergence.

k	$x^{(0)} = 0.4$		$x^{(0)} = 0.6$		$x^{(0)} = 1$	
	$x^{(k)}$	$\frac{\lvert x^{(k)} - x^{(15)}\rvert}{\lvert x^{(k-1)} - x^{(15)}\rvert}$	$x^{(k)}$	$\frac{\lvert x^{(k)} - x^{(15)}\rvert}{\lvert x^{(k-1)} - x^{(15)}\rvert}$	$x^{(k)}$	$\frac{\lvert x^{(k)} - x^{(15)}\rvert}{\lvert x^{(k-1)} - x^{(15)}\rvert}$
2	3.3887	0.1128	3.4727	0.4791	2.9873	0.4959
3	3.2645	0.4974	3.3056	0.4953	3.0646	0.4989
4	3.2030	0.4992	3.2234	0.4988	3.1031	0.4996
5	3.1723	0.4996	3.1825	0.4995	3.1224	0.4997
6	3.1569	0.4995	3.1620	0.4994	3.1320	0.4995
7	3.1493	0.4990	3.1518	0.4990	3.1368	0.4990
8	3.1454	0.4980	3.1467	0.4980	3.1392	0.4980

Rate of convergence ≈ 0.5

Plot of modulus of iteration errors for different initial guesses (see above table) ▷

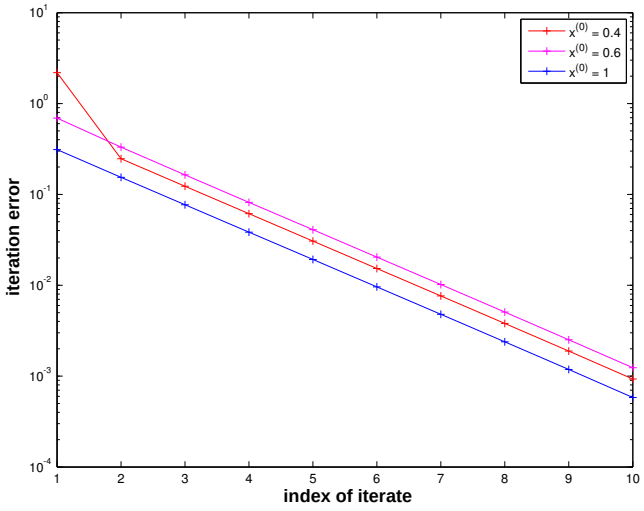


Fig. 285

Observation:

Linear convergence as in Def. 8.1.9

⇕

error graphs = straight lines in lin-log scale

→ Rem. 8.1.13

There are notions of convergence that guarantee a much faster (asymptotic) decay of norm of the iteration error than linear convergence from Def. 8.1.9.

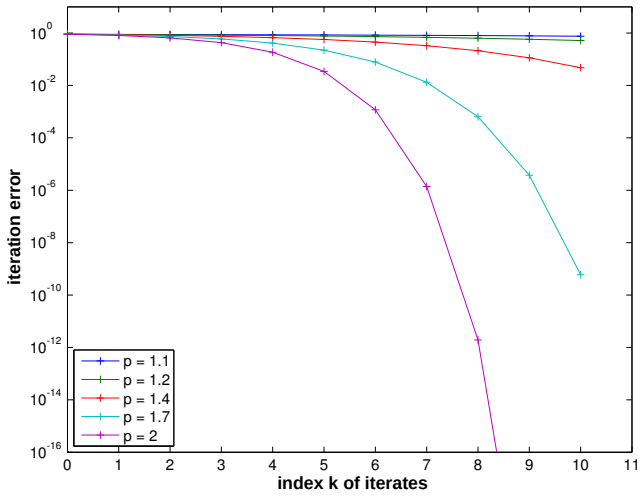
Definition 8.1.17. Order of convergence → [?, Sect. 17.2], [?, Def. 5.14], [?, Def. 6.1]

A **convergent** sequence $\mathbf{x}^{(k)}$, $k = 0, 1, 2, \ldots$, in $\mathbb{R}^n$ with limit $\mathbf{x}^* \in \mathbb{R}^n$ converges with order p, if

$$\exists C > 0: \quad \left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\| \leq C \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|^p \quad \forall k \in \mathbb{N}_0 \,, \tag{8.1.18}$$

and, in addition, $C < 1$ in the case $p = 1$ (linear convergence → Def. 8.1.9).

Of course, the order p of convergence of an iterative method refers to the largest possible p in the definition, that is, the error estimate will in general not hold, if p is replaced with $p + \epsilon$ for any $\epsilon > 0$, *cf.* Rem. 1.4.6.



◁ Qualitative error graphs for convergence of order p
(lin-log scale)

In the case of convergence of order p ($p > 1$) (see Def. 8.1.17):

$$\epsilon_{k+1} \approx C\epsilon_k^p \quad \Rightarrow \quad \log \epsilon_{k+1} = \log C + p \log \epsilon_k \quad \Rightarrow \quad \log \epsilon_{k+1} = \log C \sum_{l=0}^{k} p^l + p^{k+1} \log \epsilon_0$$

$$\Rightarrow \quad \log \epsilon_{k+1} = -\frac{\log C}{p-1} + \left(\frac{\log C}{p-1} + \log \epsilon_0 \right) p^{k+1} .$$

In this case, the error graph is a concave power curve (for sufficiently small ϵ_0 !)

Remark 8.1.19 (Detecting order of convergence)

How to guess the order of convergence ($\rightarrow$ Def. 8.1.17) from tabulated error norms measured in a numerical experiment?

Abbreviate $\epsilon_k := \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|$ (norm of iteration error):

$$\text{assume} \quad \epsilon_{k+1} \approx C\epsilon_k^p \quad \Rightarrow \quad \log \epsilon_{k+1} \approx \log C + p \log \epsilon_k \quad \Rightarrow \quad \frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}} \approx p .$$

➢ monitor the quotients $(\log \epsilon_{k+1} - \log \epsilon_k) / (\log \epsilon_k - \log \epsilon_{k-1})$ over several steps of the iteration.

Example 8.1.20 (quadratic convergence = convergence of order 2)

From your analysis course [**?**, Bsp. 3.3.2(iii)] recall the famous iteration for computing $\sqrt{a}$, $a > 0$:

$$x^{(k+1)} = \frac{1}{2}(x^{(k)} + \frac{a}{x^{(k)}}) \quad \Rightarrow \quad |x^{(k+1)} - \sqrt{a}| = \frac{1}{2x^{(k)}} |x^{(k)} - \sqrt{a}|^2 . \tag{8.1.21}$$

By the arithmetic-geometric mean inequality (AGM) $\sqrt{ab} \leq \frac{1}{2}(a+b)$ we conclude: $x^{(k)} > \sqrt{a}$ for $k \geq 1$. Therefore estimate from (8.1.21) means that the sequence from (8.1.21) converges with order 2 to $\sqrt{a}$.

Note: $x^{(k+1)} < x^{(k)}$ for all $k \geq 2$ ➢ $(x^{(k)})_{k\in\mathbb{N}_0}$ converges as a decreasing sequence that is bounded from below ($\rightarrow$ analysis course)

Numerical experiment: iterates for $a = 2$:

k	$x^{(k)}$	$e^{(k)} := x^{(k)} - \sqrt{2}$	$\log \frac{\lvert e^{(k)}\rvert}{\lvert e^{(k-1)}\rvert} : \log \frac{\lvert e^{(k-1)}\rvert}{\lvert e^{(k-2)}\rvert}$
0	2.00000000000000000	0.58578643762690485	
1	1.50000000000000000	0.08578643762690485	
2	1.41666666666666652	0.00245310429357137	1.850
3	1.41421568627450966	0.00000212390141452	1.984
4	1.41421356237468987	0.00000000000159472	2.000
5	1.41421356237309492	0.00000000000000022	0.630

Note the doubling of the number of significant digits in each step ! [impact of roundoff !]

The doubling of the number of significant digits for the iterates holds true for any quadratically convergent iteration:

Recall from Rem. 1.5.25 that the relative error ($\rightarrow$ Def. 1.5.24) tells the number of significant digits. Indeed, denoting the relative error in step k by δ_k, we have in the case of quadratic convergence.

$$\begin{aligned} x^{(k)} = x^*(1+\delta_k) \quad &\Rightarrow \quad x^{(k)} - x^* = \delta_k x^* \,. \\ \Rightarrow |x^*\delta_{k+1}| = |x^{(k+1)} - x^*| \leq C|x^{(k)} - x^*|^2 &= C|x^*\delta_k|^2 \\ \Rightarrow \quad |\delta_{k+1}| \leq C|x^*|\delta_k^2 \,. & \end{aligned} \tag{8.1.22}$$

Note: $\delta_k \approx 10^{-\ell}$ means that $\mathbf{x}^{(k)}$ has ℓ significant digits.

Also note that if $C \approx 1$, then $\delta_k = 10^{-\ell}$ and (8.1.20) implies $\delta_{k+1} \approx 10^{-2\ell}$.

8.1.2 Termination criteria

Supplementary reading. Also discussed in [**?**, Sect. 3.1, p. 42].

As remarked above, usually (even without roundoff errors) an iteration will never arrive at an/the exact solution $\mathbf{x}^*$ after finitely many steps. Thus, we can only hope to compute an *approximate* solution by accepting $\mathbf{x}^{(K)}$ as result for some $K \in \mathbb{N}_0$. Termination criteria (stopping rules) are used to determine a suitable value for K.

For the sake of efficiency ▷ stop iteration when iteration error is just "small enough"
("small enough" depends on the concrete problem and user demands.)

(8.1.23) Classification of termination criteria (stopping rules) for iterative solvers for non-linear systems of equations

A termination criterion (stopping rule) is an algorithm deciding in each step of an iterative method whether to **STOP** or to **CONTINUE**.

A priori termination	A posteriori termination
Decision to stop based on information about F and $\mathbf{x}^{(0)}$, made before starting iteration.	Beside $\mathbf{x}^{(0)}$ and F, also current and past iterates are used to decide about termination.

A termination criterion for a convergent iteration is deemed reliable, if it lets the iteration **CONTINUE**, until the iteration error $\mathbf{e}^{(k)} := \mathbf{x}^{(k)} - \mathbf{x}^*$, $\mathbf{x}^*$ the limit value, satisfies certain conditions (usually imposed before the start of the iteration).

(8.1.24) Ideal termination

Termination criteria are usually meant to ensure accuracy of the final iterate $\mathbf{x}^{(K)}$ in the following sense:

$$\left\|\mathbf{x}^{(K)} - \mathbf{x}^*\right\| \leq \tau_{\text{abs}}, \quad \tau_{\text{abs}} \;\hat{=}\; \text{prescribed (absolute) tolerance.}$$
$$\textbf{or}$$
$$\left\|\mathbf{x}^{(K)} - \mathbf{x}^*\right\| \leq \tau_{\text{rel}}\|\mathbf{x}^*\|, \quad \tau_{\text{rel}} \;\hat{=}\; \text{prescribed (relative) tolerance.}$$

it seems that the second criterion, asking that the relative ($\rightarrow$ Def. 1.5.24) iteration error be below a prescribed threshold, alone would suffice, but the absolute tolerance should be checked, if, by "accident", $\|\mathbf{x}^*\| = 0$ is possible. Otherwise, the iteration might fail to terminate at all.

Both criteria enter the "ideal (a posteriori) termination rule":

$$\text{STOP at step} \quad K = \operatorname{argmin}\{k \in \mathbb{N}_0 \colon \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \leq \begin{cases} \tau_{\text{abs}} \\ \textbf{or} \\ \tau_{\text{rel}}\|\mathbf{x}^*\| \end{cases}. \tag{8.1.25}$$

Obviously, (8.1.25) achieves the optimum in terms of efficiency and reliability. As obviously, this termination criterion is not practical, because $\mathbf{x}^*$ is not known.

(8.1.26) Practical termination criteria for iterations

The following termination criteria are commonly used in numerical codes:

① A priori termination: stop iteration after fixed number of steps (possibly depending on $\mathbf{x}^{(0)}$).

Drawback: hard to ensure prescribed accuracy!

(A priori $\hat{=}$ without actually taking into account the computed iterates, see § 8.1.23)

Invoking additional properties of either the non-linear system of equations $F(\mathbf{x}) = 0$ or the iteration it is sometimes possible to tell that for sure $\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \leq \tau$ for all $k \geq K$, though this K may be (significantly) larger than the optimal termination index from (8.1.25), see Rem. 8.1.28.

② Residual based termination: STOP convergent iteration $\{\mathbf{x}^{(k)}\}_{k\in\mathbb{N}_0}$, when

$$\left\|F(\mathbf{x}^{(k)})\right\| \leq \tau\,, \qquad \tau \hat{=} \text{ prescribed tolerance } > 0\,.$$

☹ no guaranteed accuracy

Consider the case $n = 1$. If $F : D \subset \mathbb{R} \to \mathbb{R}$ is "flat" in the neighborhood of a zero x^*, then a small value of $|F(x)|$ does not mean that x is close to x^*.

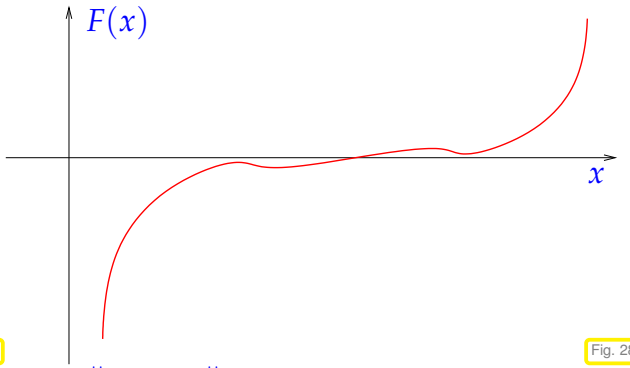


Fig. 286

$\left\|F(\mathbf{x}^{(k)})\right\|$ small $\not\Rightarrow |x - x^*|$ small

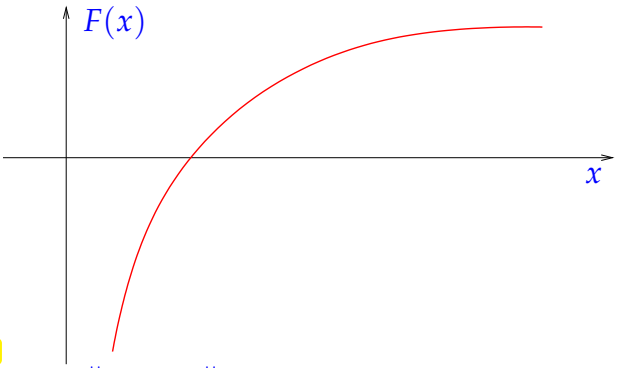


Fig. 287

$\left\|F(\mathbf{x}^{(k)})\right\|$ small $\Rightarrow |x - x^*|$ small

③ Correction based termination: STOP convergent iteration $\{\mathbf{x}^{(k)}\}_{k\in\mathbb{N}_0}$, when

$$\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right\| \leq \begin{cases} \tau_{\text{abs}} \\ \textbf{or} \\ \tau_{\text{rel}}\left\|\mathbf{x}^{(k+1)}\right\| , \end{cases} \qquad \begin{matrix}\tau_{\text{abs}} \\ \tau_{\text{rel}}\end{matrix} \text{ prescribed } \begin{matrix}\text{absolute} \\ \text{relative}\end{matrix} \text{ tolerances } > 0 .$$

Also for this criterion, we have no guarantee that (8.1.25) will be satisfied only remotely.

A special variant of correction based termination exploits that $\mathbb{M}$ is finite! (→ Section 1.5.3)

Wait until (convergent) iteration becomes stationary in the discrete set $\mathbb{M}$ of machine numbers!

☹ possibly grossly inefficient !
(always computes "up to machine precision")

C++11 code 8.1.27: Square root iteration → Ex. 8.1.20

```
2  double sqrtit(double a)
3  {
4    double x_old = -1;
5    double x = a;
6    while (x_old != x) {
7      x_old = x;
8      x = 0.5*(x+a/x);
9    }
10   return x;
11 }
```

Remark 8.1.28 (A posteriori termination criterion for linearly convergent iterations → [?, Lemma 5.17, 5.19])

Let us assume that we know that an iteration linearly convergent (→ Def. 8.1.9) with rate of convergence $0 < L < 1$:

The following simple manipulations give an a posteriori termination criterion (for linearly convergent iterations with rate of convergence $0 < L < 1$):

$$\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \overset{\triangle\text{-inequ.}}{\leq} \left\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right\| + \left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\| \leq \left\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right\| + L\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| .$$

Iterates satisfy: $$\boxed{\left\|\mathbf{x}^{(k+1)}-\mathbf{x}^{*}\right\| \leq \frac{L}{1-L}\left\|\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\right\|} \,. \tag{8.1.29}$$

This suggests that we take the right hand side of (8.1.29) as a posteriori error bound and use it instead of the inaccessible $\left\|\mathbf{x}^{(k+1)}-\mathbf{x}^{*}\right\|$ for checking absolute and relative accuracy in (8.1.25). The resulting termination criterium will be reliable (→ § 8.1.23), since we will certainly have achieved the desired accuracy when we stop the iteration.

Estimating the rate of convergence L might be difficult.

Pessimistic estimate for L will not compromise reliability.

(Using $\widetilde{L} > L$ in (8.1.29) still yields a valid *upper bound* for $\left\|\mathbf{x}^{(k)}-\mathbf{x}^{*}\right\|$.)

Example 8.1.30 (A posteriori error bound for linearly convergent iteration)

We revisit the iteration of Ex. 8.1.15:

$$x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)} + 1}{\sin x^{(k)}} \quad \Rightarrow x^{(k)} \to \pi \quad \text{for } x^{(0)} \text{ close to } \pi \,.$$

Observed rate of convergence: $L = 1/2$

Error and error bound for $x^{(0)} = 0.4$:

k	$\lvert x^{(k)} - \pi\rvert$	$\frac{L}{1-L}\lvert x^{(k)} - x^{(k-1)}\rvert$	slack of bound
1	2.191562221997101	4.933154875586894	2.741592653589793
2	0.247139097781070	1.944423124216031	1.697284026434961
3	0.122936737876834	0.124202359904236	0.001265622027401
4	0.061390835206217	0.061545902670618	0.000155067464401
5	0.030685773472263	0.030705061733954	0.000019288261691
6	0.015341682696235	0.015344090776028	0.000002408079792
7	0.007670690889185	0.007670991807050	0.000000300917864
8	0.003835326638666	0.003835364250520	0.000000037611854
9	0.001917660968637	0.001917665670029	0.000000004701392
10	0.000958830190489	0.000958830778147	0.000000000587658
11	0.000479415058549	0.000479415131941	0.000000000073392
12	0.000239707524646	0.000239707533903	0.000000000009257
13	0.000119853761949	0.000119853762696	0.000000000000747
14	0.000059926881308	0.000059926880641	0.000000000000667
15	0.000029963440745	0.000029963440563	0.000000000000181

Hence: the a posteriori error bound is highly accurate in this case!

8.2 Fixed Point Iterations

Supplementary reading. The contents of this section are also treated in [**?**, Sect. 5.3], [**?**, Sect. 6.3], [**?**, Sect .3.3]

As before we consider a non-linear system of equations $F(\mathbf{x}) = 0$, $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$.

1-point stationary iterative methods, see (8.1.4), for $F(\mathbf{x}) = 0$ are also called fixed point iterations.

A fixed point iteration is defined by iteration function $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$:

iteration function $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$
initial guess $\mathbf{x}^{(0)} \in U$

$\succ$ iterates $(\mathbf{x}^{(k)})_{k\in\mathbb{N}_0}$: $\mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)})$.

$\rightarrow$ 1-point method, *cf.* (8.1.4)

Here, U designates the domain of definition of the iteration function Φ.

Note that the sequence of iterates need not be well defined: $\mathbf{x}^{(k)} \notin U$ possible !

8.2.1 Consistent fixed point iterations

Next, we specialize Def. 8.1.7 for fixed point iterations:

Definition 8.2.1. Consistency of fixed point iterations, *c.f.* Def. 8.1.7

A fixed point iteration $\mathbf{x}^{(k+1)} = \Phi(\mathbf{x}^{(k)})$ is consistent with $F(\mathbf{x}) = 0$, if, for $\mathbf{x} \in U \cap D$,

$$F(\mathbf{x}) = 0 \quad \Leftrightarrow \quad \Phi(\mathbf{x}) = \mathbf{x} \,.$$

Note: iteration function Φ continuous **and** fixed point iteration (locally) convergent to $\mathbf{x}^* \in U$ $\Rightarrow$ $\mathbf{x}^*$ is a fixed point of Φ.

This is an immediate consequence that for a continuous function limits and function evaluations commute [**?**, Sect. 4.1].

General construction of fixed point iterations that is consistent with $F(\mathbf{x}) = 0$:

1. Rewrite equivalently $F(\mathbf{x}) = 0 \quad \Leftrightarrow \quad \Phi(\mathbf{x}) = \mathbf{x}$ and then
2. use the fixed point iteration

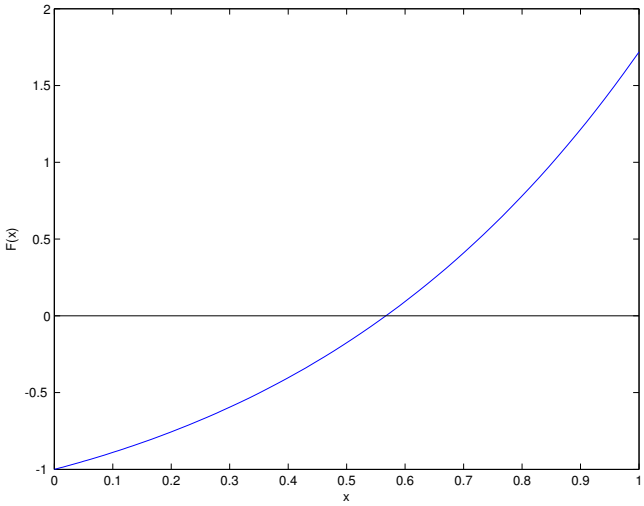
$$\mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)}) \,. \tag{8.2.2}$$

Note: there are *many ways* to transform $F(\mathbf{x}) = 0$ into a fixed point form **!**

Experiment 8.2.3 (Many choices for consistent fixed point iterations)

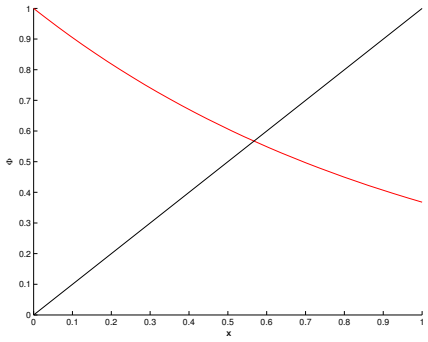
In this example we construct three different consistent fixed point iteration for a single scalar ($n = 1$) non-linear equation $F(x) = 0$. In numerical experiments we will see that they behave very differently.

$$F(x) = xe^x - 1\,, \quad x \in [0,1]\,.$$

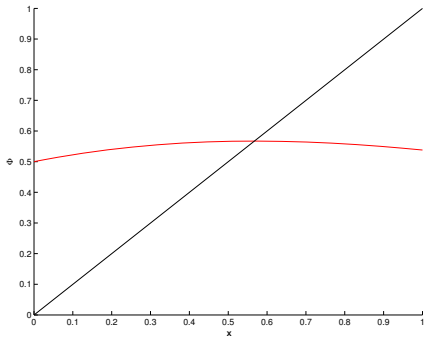


Different fixed point forms:

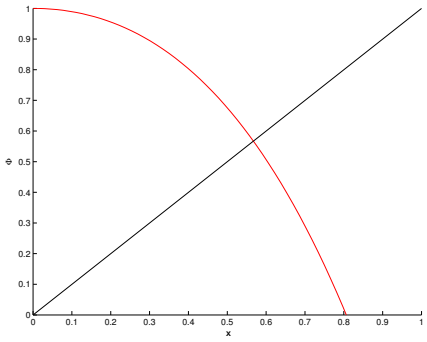
$$\begin{aligned}\Phi_1(x) &= e^{-x}\,,\\ \Phi_2(x) &= \frac{1+x}{1+e^x}\,,\\ \Phi_3(x) &= x+1-xe^x\,.\end{aligned}$$



function Φ_1



function Φ_2



function Φ_3

With the same intial guess $x^{(0)} = 0.5$ for all three fixed point iterations we obtain the following iterates:

k	$x^{(k+1)} := \Phi_1(x^{(k)})$	$x^{(k+1)} := \Phi_2(x^{(k)})$	$x^{(k+1)} := \Phi_3(x^{(k)})$
0	0.500000000000000	0.500000000000000	0.500000000000000
1	0.606530659712633	0.566311003197218	0.675639364649936
2	0.545239211892605	0.567143165034862	0.347812678511202
3	0.579703094878068	0.567143290409781	0.855321409174107
4	0.560064627938902	0.567143290409784	-0.156505955383169
5	0.571172148977215	0.567143290409784	0.977326422747719
6	0.564862946980323	0.567143290409784	-0.619764251895580
7	0.568438047570066	0.567143290409784	0.713713087416146
8	0.566409452746921	0.567143290409784	0.256626649129847
9	0.567559634262242	0.567143290409784	0.924920676910549
10	0.566907212935471	0.567143290409784	-0.407422405542253

We can also tabulate the modulus of the iteration error and mark correct digits with red:

k	$\lvert x_1^{(k+1)} - x^*\rvert$	$\lvert x_2^{(k+1)} - x^*\rvert$	$\lvert x_3^{(k+1)} - x^*\rvert$
0	0.067143290409784	0.067143290409784	0.067143290409784
1	0.039387369302849	0.000832287212566	0.108496074240152
2	0.021904078517179	0.000000125374922	0.219330611898582
3	0.012559804468284	0.000000000000003	0.288178118764323
4	0.007078662470882	0.000000000000000	0.723649245792953
5	0.004028858567431	0.000000000000000	0.410183132337935
6	0.002280343429460	0.000000000000000	1.186907542305364
7	0.001294757160282	0.000000000000000	0.146569797006362
8	0.000733837662863	0.000000000000000	0.310516641279937
9	0.000416343852458	0.000000000000000	0.357777386500765
10	0.000236077474313	0.000000000000000	0.974565695952037

Observed: linear convergence of $x_1^{(k)}$, quadratic convergence of $x_2^{(k)}$,
no convergence (erratic behavior of $x_3^{(k)}$) ($x_i^{(0)} = 0.5$ in all cases).

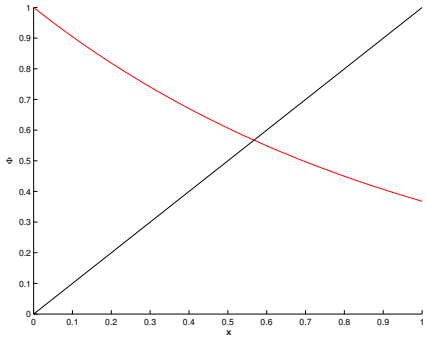
Question: Can we explain/forecast the behaviour of a fixed point iteration?

8.2.2 Convergence of fixed point iterations

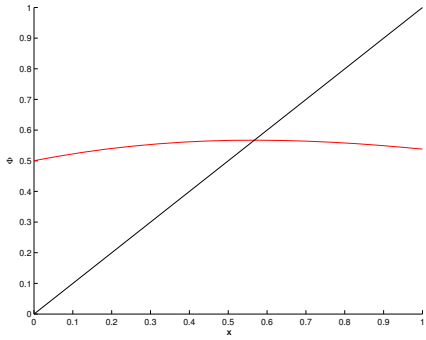
In this section we will try to find easily verifiable conditions that ensure convergence (of a certain order) of fixed point iterations. It will turn out that these conditions are surprisingly simple and general.

Experiment 8.2.4 (Exp. 8.2.3 revisited)

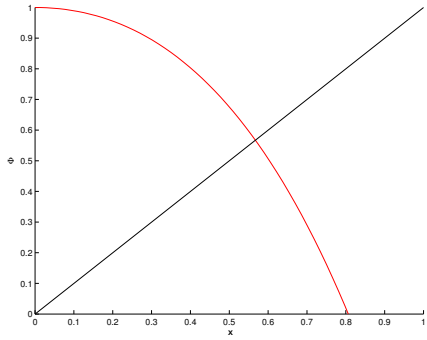
In Exp. 8.2.3 we observed vastly different behavior of different fixed point iterations for $n = 1$. Is it possible to predict this from the shape of the graph of the iteration functions?



Φ_1: linear convergence ?



Φ_2: quadratic convergence ?



function Φ_3: no convergence

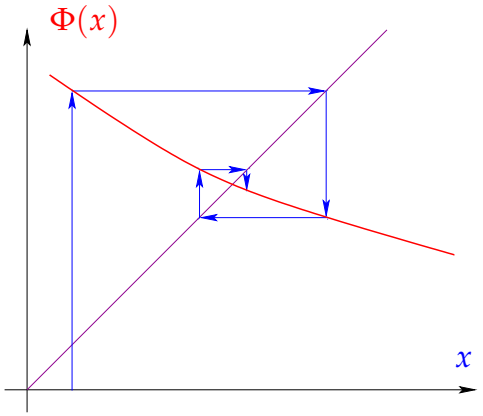
Remark 8.2.5 (Visualization of fixed point iterations in 1D)

1D setting ($n = 1$): $\Phi : \mathbb{R} \mapsto \mathbb{R}$ continuously differentiable, $\Phi(x^*) = x^*$

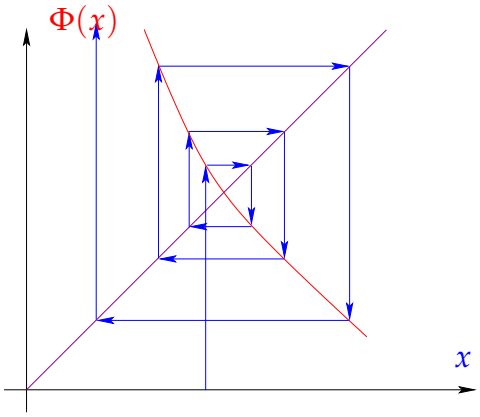
fixed point iteration: $x^{(k+1)} = \Phi(x^{(k)})$

In 1D it is possible to visualize the different convergence behavior of fixed point iterations: In order to construct $x^{(k+1)}$ from $x^{(k)}$ one moves vertically to $(x^{(k)}, x^{(k+1)} = \Phi(x^{(k)}))$, then horizontally to the

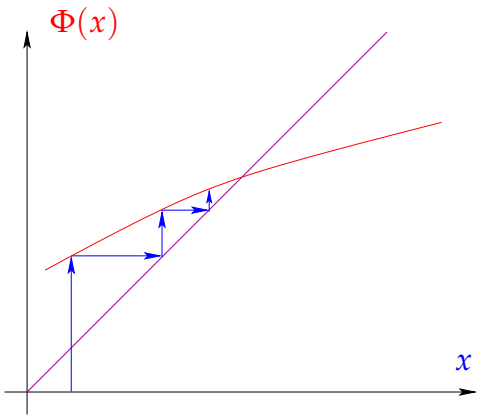
angular bisector of the first/third quadrant, that is, to the point $(x^{(k+1)}, x^{(k+1)})$. Returning vertically to the abscissa gives $x^{(k+1)}$.



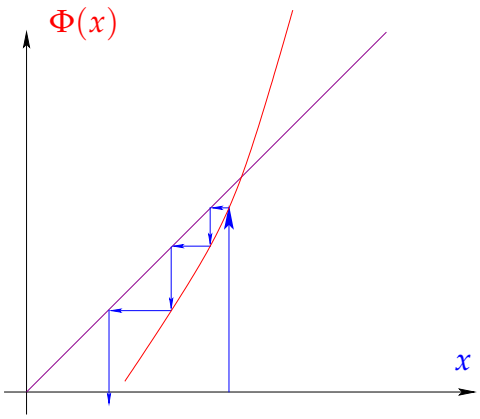
$-1 < \Phi'(x^*) \leq 0$ ➢ convergence



$\Phi'(x^*) < -1$ ➢ divergence



$0 \leq \Phi'(x^*) < 1$ ➢ convergence



$1 < \Phi'(x^*)$ ➢ divergence

Numerical examples for iteration functions ➢ Exp. 8.2.3, iteration functions Φ_1 and Φ_3

It seems that the slope of the iteration function Φ in the fixed point, that is, in the point where it intersects the bisector of the first/third quadrant, is crucial.

Now we investigate rigorously, when a fixed point iteration will lead to a convergent iteration with a particular qualitative kind of convergence according to Def. 8.1.17.

Definition 8.2.6. Contractive mapping

$\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ is contractive (w.r.t. norm $\|\cdot\|$ on $\mathbb{R}^n$), if

$$\exists L < 1: \quad \|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in U \, . \tag{8.2.7}$$

A simple consideration: if $\Phi(\mathbf{x}^*) = \mathbf{x}^*$ (fixed point), then a fixed point iteration induced by a contractive mapping Φ satisfies

$$\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\| = \left\|\Phi(\mathbf{x}^{(k)}) - \Phi(\mathbf{x}^*)\right\| \overset{(8.2.7)}{\leq} L\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| ,$$

that is, the iteration converges (at least) linearly ($\rightarrow$ Def. 8.1.9).

Note that Φ contractive $\Rightarrow$ Φ has *at most one* fixed point.

Remark 8.2.8 (Banach's fixed point theorem → [?, Satz 6.5.2],[?, Satz 5.8])

A key theorem in calculus (also functional analysis):

Theorem 8.2.9. Banach's fixed point theorem

If $D \subset \mathbb{K}^n$ ($\mathbb{K} = \mathbb{R}, \mathbb{C}$) closed and bounded and $\Phi : D \mapsto D$ satisfies

$$\exists L < 1: \quad \|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in D \,,$$

then there is a unique fixed point $\mathbf{x}^* \in D$, $\Phi(\mathbf{x}^*) = \mathbf{x}^*$, which is the limit of the sequence of iterates $\mathbf{x}^{(k+1)} := \Phi(x^{(k)})$ for any $\mathbf{x}^{(0)} \in D$.

Proof. Proof based on 1-point iteration $\mathbf{x}^{(k)} = \Phi(\mathbf{x}^{(k-1)})$, $\mathbf{x}^{(0)} \in D$:

$$\left\|\mathbf{x}^{(k+N)} - \mathbf{x}^{(k)}\right\| \leq \sum_{j=k}^{k+N-1} \left\|\mathbf{x}^{(j+1)} - \mathbf{x}^{(j)}\right\| \leq \sum_{j=k}^{k+N-1} L^j \left\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\right\|$$

$$\leq \frac{L^k}{1-L}\left\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\right\| \xrightarrow{k\to\infty} 0 \,.$$

$(\mathbf{x}^{(k)})_{k\in\mathbb{N}_0}$ Cauchy sequence ➤ convergent $\mathbf{x}^{(k)} \xrightarrow{k\to\infty} \mathbf{x}^*$.
Continuity of Φ ➤ $\Phi(\mathbf{x}^*) = \mathbf{x}^*$. Uniqueness of fixed point is evident.

□

A simple criterion for a differentiable Φ to be contractive:

Lemma 8.2.10. Sufficient condition for local linear convergence of fixed point iteration → [?, Thm. 17.2], [?, Cor. 5.12]

If $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$, $\Phi(\mathbf{x}^*) = \mathbf{x}^*$, Φ differentiable in $\mathbf{x}^*$, and $\|\mathrm{D}\Phi(\mathbf{x}^*)\| < 1$, then the fixed point iteration

$$\mathbf{x}^{(k+1)} := \Phi(\mathbf{x}^{(k)}) \,, \tag{8.2.2}$$

converges locally and at least linearly. *matrix norm, Def. 1.5.76 !*

✎ notation: $\mathrm{D}\Phi(\mathbf{x}) \hat{=}$ Jacobian (*ger.:* Jacobi-Matrix) of Φ at $\mathbf{x} \in D$ → [**?**, Sect. 7.6]

$$\mathrm{D}\Phi(\mathbf{x}) = \left[\frac{\partial \Phi_i}{\partial x_j}(\mathbf{x})\right]_{i,j=1}^{n} = \begin{bmatrix} \frac{\partial \Phi_1}{\partial x_1}(\mathbf{x}) & \frac{\partial \Phi_1}{\partial x_2}(\mathbf{x}) & \cdots & \cdots & \frac{\partial \Phi_1}{\partial x_n}(\mathbf{x}) \\ \frac{\partial \Phi_2}{\partial x_1}(\mathbf{x}) & & & & \frac{\partial \Phi_2}{\partial x_n}(\mathbf{x}) \\ \vdots & & & & \vdots \\ \frac{\partial \Phi_n}{\partial x_1}(\mathbf{x}) & \frac{\partial \Phi_n}{\partial x_2}(\mathbf{x}) & \cdots & \cdots & \frac{\partial \Phi_n}{\partial x_n}(\mathbf{x}) \end{bmatrix} . \tag{8.2.11}$$

"Visualization" of the statement of Lemma 8.2.10 in Rem. 8.2.5: The iteration converges *locally*, if Φ is flat in a neighborhood of x^*, it will diverge, if Φ is steep there.

Proof. (of Lemma 8.2.10) By definition of derivative

$$\|\Phi(\mathbf{y}) - \Phi(\mathbf{x}^*) - D\Phi(\mathbf{x}^*)(\mathbf{y} - \mathbf{x}^*)\| \leq \psi(\|\mathbf{y} - \mathbf{x}^*\|)\|\mathbf{y} - \mathbf{x}^*\| ,$$

with $\psi : \mathbb{R}_0^+ \mapsto \mathbb{R}_0^+$ satisfying $\lim_{t\to 0} \psi(t) = 0$.

Choose $\delta > 0$ such that

$$L := \psi(t) + \|D\Phi(\mathbf{x}^*)\| \leq \tfrac{1}{2}(1 + \|D\Phi(\mathbf{x}^*)\|) < 1 \quad \forall 0 \leq t < \delta .$$

By inverse triangle inequality we obtain for fixed point iteration

$$\begin{aligned} &\|\Phi(\mathbf{x}) - \mathbf{x}^*\| - \|D\Phi(\mathbf{x}^*)(\mathbf{x} - \mathbf{x}^*)\| \leq \psi(\|\mathbf{x} - \mathbf{x}^*\|)\|\mathbf{x} - \mathbf{x}^*\| \\ \blacktriangleright \quad &\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\| \leq (\psi(t) + \|D\Phi(\mathbf{x}^*)\|)\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \leq L\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| , \end{aligned}$$

if $\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| < \delta$. □

Lemma 8.2.12. Sufficient condition for linear convergence of fixed point iteration

Let U be convex and $\Phi : U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ be continuously differentiable with

$$L := \sup_{\mathbf{x}\in U} \|D\Phi(\mathbf{x})\| < 1 .$$

If $\Phi(\mathbf{x}^*) = \mathbf{x}^*$ for some interior point $\mathbf{x}^* \in U$, then the fixed point iteration $\mathbf{x}^{(k+1)} = \Phi(\mathbf{x}^{(k)})$ converges to $\mathbf{x}^*$ at least linearly with rate L.

Recall: $U \subset \mathbb{R}^n$ convex :⇔ $(t\mathbf{x} + (1-t)\mathbf{y}) \in U$ for all $\mathbf{x}, \mathbf{y} \in U$, $0 \leq t \leq 1$

Proof. (of Lemma 8.2.12) By the mean value theorem

$$\begin{aligned} \Phi(\mathbf{x}) - \Phi(\mathbf{y}) &= \int_0^1 D\Phi(\mathbf{x} + \tau(\mathbf{y} - \mathbf{x}))(\mathbf{y} - \mathbf{x})\,\mathrm{d}\tau \quad \forall \mathbf{x}, \mathbf{y} \in \mathrm{dom}(\Phi) . \\ \Rightarrow \quad &\|\Phi(\mathbf{x}) - \Phi(\mathbf{y})\| \leq L\|\mathbf{y} - \mathbf{x}\| , \\ \Rightarrow \quad &\left\|(\mathbf{x})^{(k+1)} - \mathbf{x}^*\right\| \leq L\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| . \end{aligned}$$

We find that Φ is contractive on U with unique fixed point $\mathbf{x}^*$, to which $\mathbf{x}^{(k)}$ converges linearly for $k \to \infty$. □

Remark 8.2.13 (Bound for *asymptotic* rate of linear convergence)

By asymptotic rate of a linearly converging iteration we mean the contraction factor for the norm of the iteration error that we can expect, when we are already very close to the limit $\mathbf{x}^*$.

If $0 < \|D\Phi(\mathbf{x}^*)\| < 1$, $\mathbf{x}^{(k)} \approx \mathbf{x}^*$ then the (worst) asymptotic rate of linear convergence is $L = \|D\Phi(x^*)\|$

Example 8.2.14 (Multidimensional fixed point iteration)

In this example we encounter the first genuine *system* of non-linear equations and apply Lemma 8.2.12 to it.

$$\begin{array}{ccc} \text{System of equations} & \text{in} & \text{fixed point form:} \\ \begin{cases} x_1 - c(\cos x_1 - \sin x_2) &= 0 \\ (x_1 - x_2) - c\sin x_2 &= 0 \end{cases} & \Rightarrow & \begin{cases} c(\cos x_1 - \sin x_2) &= x_1 \\ c(\cos x_1 - 2\sin x_2) &= x_2 \end{cases} . \end{array}$$

$$\text{Define:} \quad \Phi\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = c\begin{bmatrix} \cos x_1 - \sin x_2 \\ \cos x_1 - 2\sin x_2 \end{bmatrix} \quad \Rightarrow \quad D\Phi\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = -c\begin{bmatrix} \sin x_1 & \cos x_2 \\ \sin x_1 & 2\cos x_2 \end{bmatrix}.$$

Choose *appropriate* norm: $\|\cdot\|$ = ∞-norm $\|\cdot\|_\infty$ ($\rightarrow$ Example 1.5.78) ;

$$\text{if } c < \frac{1}{3} \quad \Rightarrow \quad \|D\Phi(\mathbf{x})\|_\infty < 1 \quad \forall \mathbf{x} \in \mathbb{R}^2 ,$$

$\succ$ (at least) linear convergence of the fixed point iteration. The existence of a fixed point is also guaranteed, because Φ maps into the closed set $[-3,3]^2$. Thus, the Banach fixed point theorem, Thm. 8.2.9, can be applied.

What about higher order convergence ($\rightarrow$ Def. 8.1.17, *cf.* Φ_2 in Ex. 8.2.3)? Also in this case we should study the derivatives of the iteration functions in the fixed point (limit point).

We give a refined convergence result only for $n = 1$ (scalar case, $\Phi : \mathrm{dom}(\Phi) \subset \mathbb{R} \mapsto \mathbb{R}$):

Theorem 8.2.15. Taylor's formula $\rightarrow$ [?, Sect. 5.5]

If $\Phi : U \subset \mathbb{R} \mapsto \mathbb{R}$, U interval, is $m+1$ times continuously differentiable, $x \in U$

$$\Phi(y) - \Phi(x) = \sum_{k=1}^{m} \frac{1}{k!}\Phi^{(k)}(x)(y-x)^k + O(|y-x|^{m+1}) \quad \forall y \in U . \tag{8.2.16}$$

Now apply Taylor expansion (8.2.16) to iteration function Φ:

If $\Phi(x^*) = x^*$ and $\Phi : \mathrm{dom}(\Phi) \subset \mathbb{R} \mapsto \mathbb{R}$ is "sufficiently smooth", it tells us that

$$x^{(k+1)} - x^* = \Phi(x^{(k)}) - \Phi(x^*) = \sum_{l=1}^{m} \frac{1}{l!}\Phi^{(l)}(x^*)(x^{(k)} - x^*)^l + O(|x^{(k)} - x^*|^{m+1}) . \tag{8.2.17}$$

Here we used the Landau symbol $O(\cdot)$ to describe the local behavior of a remainder term in the vicinity of x^*

Lemma 8.2.18. Higher order local convergence of fixed point iterations

If $\Phi : U \subset \mathbb{R} \mapsto \mathbb{R}$ is $m+1$ times continuously differentiable, $\Phi(x^*) = x^*$ for some x^* in the interior of U, and $\Phi^{(l)}(x^*) = 0$ for $l = 1, \ldots, m$, $m \geq 1$, then the fixed point iteration (8.2.2) *converges locally to x^* with order $\geq m+1$ ($\rightarrow$ Def. 8.1.17).*

Proof. For neighborhood $\mathcal{U}$ of x^*

$$(8.2.17) \quad \Rightarrow \quad \exists C > 0: \quad |\Phi(y) - \Phi(x^*)| \leq C\,|y - x^*|^{m+1} \quad \forall y \in \mathcal{U} .$$

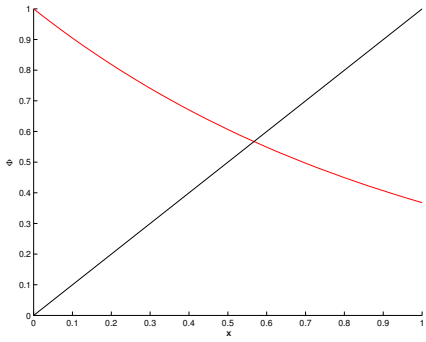
$$\delta^m C < 1/2 : \quad |x^{(0)} - x^*| < \delta \quad \Rightarrow \quad |x^{(k)} - x^*| < 2^{-k}\delta \quad \succ \text{ local convergence .}$$

Then appeal to (8.2.17)

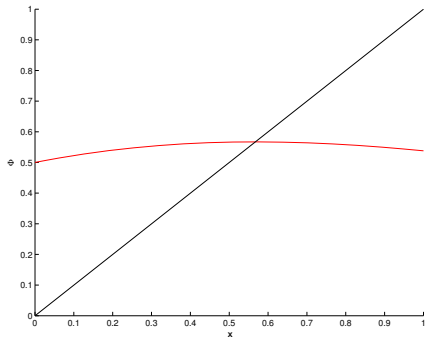
□

Experiment 8.2.19 (Exp. 8.2.4 continued)

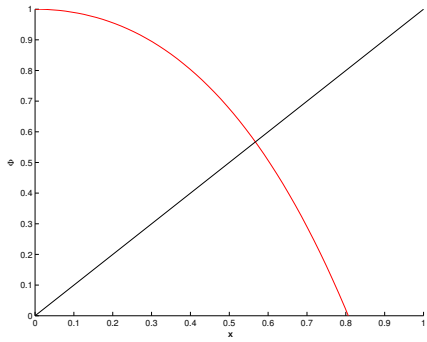
Now, Lemma 8.2.12 and Lemma 8.2.18 permit us a precise prediction of the (asymptotic) convergence we can expect from the different fixed point iterations studied in Exp. 8.2.3.



function Φ_1



function Φ_2



function Φ_3

$$\Phi_2'(x) = \frac{1 - xe^x}{(1+e^x)^2} = 0 \quad \text{, if} \quad xe^x - 1 = 0 \quad \text{hence quadratic convergence ! .}$$

Since $x^*e^{x^*} - 1 = 0$, simple computations yield

$$\Phi_1'(x) = -e^{-x} \quad \Rightarrow \quad \Phi_1'(x^*) = -x^* \approx -0.56 \quad \text{hence local linear convergence .}$$

$$\Phi_3'(x) = 1 - xe^x - e^x \quad \Rightarrow \quad \Phi_3'(x^*) = -\frac{1}{x^*} \approx -1.79 \quad \text{hence no convergence .}$$

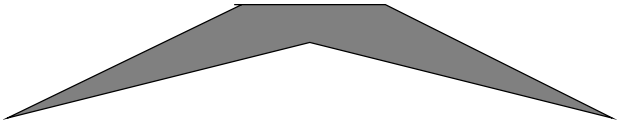
Remark 8.2.20 (Termination criterion for contractive fixed point iteration)

We recall the considerations of Rem. 8.1.28 about a termination criterion for contractive fixed point iteration (= linearly convergence fixed point iteration → Def. 8.1.9), *c.f.* (8.2.7), with contraction factor (= rate of convergence) $0 \le L < 1$:

$$\begin{aligned} \left\| \mathbf{x}^{(k+m)} - \mathbf{x}^{(k)} \right\| &\overset{\triangle\text{-ineq.}}{\le} \sum_{j=k}^{k+m-1} \left\| \mathbf{x}^{(j+1)} - \mathbf{x}^{(j)} \right\| \le \sum_{j=k}^{k+m-1} L^{j-k} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| \\ &= \frac{1-L^m}{1-L} \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} \right\| \le \frac{1-L^m}{1-L} L^{k-l} \left\| \mathbf{x}^{(l+1)} - \mathbf{x}^{(l)} \right\| . \end{aligned}$$

Hence, for $m \to \infty$, with $\mathbf{x}^* := \lim_{k\to\infty} \mathbf{x}^{(k)}$ we find the estimate

$$\left\| \mathbf{x}^* - \mathbf{x}^{(k)} \right\| \le \frac{L^{k-l}}{1-L} \left\| \mathbf{x}^{(l+1)} - \mathbf{x}^{(l)} \right\| . \tag{8.2.21}$$



Set $l = 0$ in (8.2.21)	Set $l = k-1$ in (8.2.21)
a priori termination criterion	a posteriori termination criterion
$\left\|\mathbf{x}^* - \mathbf{x}^{(k)}\right\| \leq \frac{L^k}{1-L}\left\|\mathbf{x}^{(1)} - \mathbf{x}^{(0)}\right\|$ (8.2.22)	$\left\|\mathbf{x}^* - \mathbf{x}^{(k)}\right\| \leq \frac{L}{1-L}\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\|$ (8.2.23)

With the same arguments as in Rem. 8.1.28 we see that overestimating L, that is, using a value for L that is larger than the true value, still gives reliable termination criteria.

However, whereas overestimating L in (8.2.23) will not lead to a severe deterioration of the bound, unless $L \approx 1$, using a pessimistic value for L in (8.2.22) will result in a bound way bigger than the true bound, if $k \gg 1$. Then the a priori termination criterion (8.2.22) will recommend termination many iterations after the accuracy requirements have already been met. This will thwart the efficiency of the method.

8.3 Finding Zeros of Scalar Functions

Supplementary reading. [**?**, Ch. 3] is also devoted to this topic. The algorithm of "bisection" discussed in the next subsection, is treated in [**?**, Sect. 5.5.1] and [**?**, Sect. 3.2].

Now, we focus on scalar case $n = 1$: $F : I \subset \mathbb{R} \mapsto \mathbb{R}$ **continuous**, I interval

Sought: $x^* \in I$: $F(x^*) = 0$

8.3.1 Bisection

Idea: use ordering of real numbers & intermediate value theorem [**?**, Sect. 4.6]

Input: $a, b \in I$ such that $F(a)F(b) < 0$ (different signs !)

$\Rightarrow$ $\exists\, x^* \in]\min\{a,b\}, \max\{a,b\}[$: $F(x^*) = 0$,

as we conclude from the intermediate value theorem.

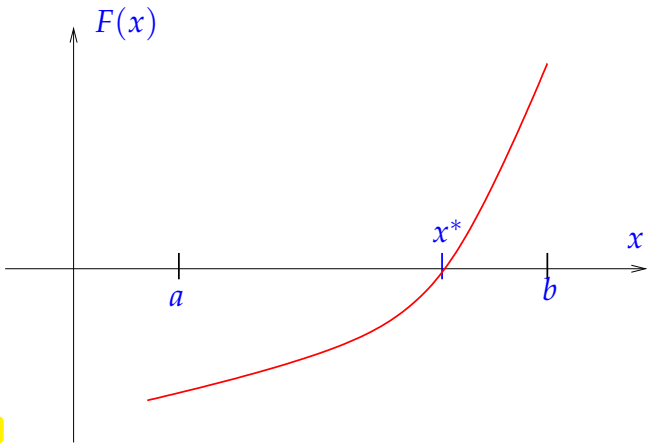


Fig. 288

Find a sequence of intervals with geometrically decreasing lengths, in each of which F will change sign.

Such a sequence can easily be found by testing the sign of F at the midpoint of the current interval, see Code 8.3.2.

(8.3.1) Bisection method

The following C++ code implements the bisection method for finding the zeros of a function passed through the function handle `F` in the interval $[a,b]$ with absolute tolerance `tol`.

C++11 code 8.3.2: Bisection method for solving $F(x) = 0$ on $[a,b]$

```
2  // Searching zero of F in [a,b] by bisection
3  template <typename Func, typename Scalar>
4  Scalar bisect(Func&& F, Scalar a, Scalar b, Scalar tol)
5  {
6    if (a > b) std::swap(a,b); // sort interval bounds
7    if (F(a)*F(b) > 0) throw "f(a) and f(b) have same sign";
8    static_assert(std::is_floating_point<Scalar>::value,
9                  "Scalar must be a floating point type");
10   int v=F(a) < 0 ? 1 : -1;
11   Scalar x = (a+b)/2; // determine midpoint
12   // termination, relies on machine arithmetic if tol = 0
13   while (b-a > tol) { //
14     assert(a<x && x<b); // assert invariant
15     // sgn(f(x)) = sgn(f(b)), then use x as next right boundary
16     if (v*F(x) > 0) b=x;
17     // sgn(f(x)) = sgn(f(a)), then use x as next left boundary
18     else a=x;
19     x = (a+b)/2; // determine next midpoint
20   }
21   return x;
22 }
```

Line 13: the test `((a<x) && (x<b))` offers a safeguard against an infinite loop in case `tol` $<$ resolution of $\mathbb{M}$ at zero x^* (*cf.* "$\mathbb{M}$-based termination criterion").

This is also an example for an algorithm that (in the case of `tol=0`) uses the properties of machine arithmetic to define an a posteriori termination criterion, see Section 8.1.2. The iteration will terminate, when, e.g., $a \tilde{+} \frac{1}{2}(b-a) = a$ ($\tilde{+}$ is the floating point realization of addition), which, by the Ass. 1.5.32 can only happen, when

$$|\tfrac{1}{2}(b-a)| \leq \mathrm{EPS} \cdot |a| \ .$$

Since the exact zero is located between a and b, this condition implies a relative error $\leq$ EPS of the computed zero.

Advantages:

- "foolproof", robust: will always terminate with a zero of requested accuracy,
- requires only point evaluations of F,
- works with *any* continuous function F, no derivatives needed.

Drawbacks:

Merely "linear-type" $(*)$ convergence: $|x^{(k)} - x^*| \leq 2^{-k}|b-a|$

$\log_2\left(\frac{|b-a|}{\mathtt{tol}}\right)$ steps necessary

$(*)$: the convergence of a bisection algorithm is not linear in the sense of Def. 8.1.9, because the condition $\left\|x^{(k+1)} - x^*\right\| \leq L\left\|x^{(k)} - x^*\right\|$ might be violated at any step of the iteration.

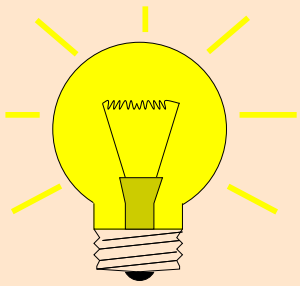
Remark 8.3.3 (Generalized bisection methods)

It is straightforward to combine the bisection idea with more elaborate "model function methods" as they will be discussed in the next section: Instead of stubbornly choosing the midpoint of the probing interval $[a,b]$ ($\rightarrow$ Code 8.3.2) as next iterate, one may use a refined guess for the location of a zero of F in $[a,b]$.

A method of this type is used by MATLAB's `fzero` function for root finding in 1D [**?**, Sect. 6.2.3].

8.3.2 Model function methods

$\hat{=}$ class of iterative methods for finding zeroes of F: iterate in step $k+1$ is computed according to the following idea:



Idea: Given recent iterates (approximate zeroes) $x^{(k)}, x^{(k-1)}, \ldots, x^{(k-m+1)}$, $m \in \mathbb{N}$

❶ replace F with a model function $\widetilde{F}_k$
(based on function values $F(x^{(k)}), F(x^{(k-1)}), \ldots, F(x^{(k-m+1)})$ and, possibly, derivative values $F'(x^{(k)}), F'(x^{(k-1)}), \ldots, F'(x^{(k-m+1)})$)

❷ $x^{(k+1)} :=$ zero of $\widetilde{F}_k$: $\widetilde{F}_k(x^{(k+1)}) = 0$
(has to be readily available $\leftrightarrow$ analytic formula)

Distinguish (see § 8.1.2 and (8.1.4)):

one-point methods : $x^{(k+1)} = \Phi_F(x^{(k)})$, $k \in \mathbb{N}$ (e.g., fixed point iteration $\rightarrow$ Section 8.2)
multi-point methods : $x^{(k+1)} = \Phi_F(x^{(k)}, x^{(k-1)}, \ldots, x^{(k-m)})$, $k \in \mathbb{N}$, $m = 2, 3, \ldots$.

8.3.2.1 Newton method in scalar case

Supplementary reading. Newton's method in 1D is discussed in [**?**, Sect. 18.1], [**?**, Sect. 5.5.2], [**?**, Sect. 3.4].

☞ Again we consider the problem of finding zeros of the function $F : I \subset \mathbb{R} \rightarrow \mathbb{R}$.
☞ Now we assume that $F : I \subset \mathbb{R} \mapsto \mathbb{R}$ is continuously differentiable.

Now model function := tangent at F in $x^{(k)}$:

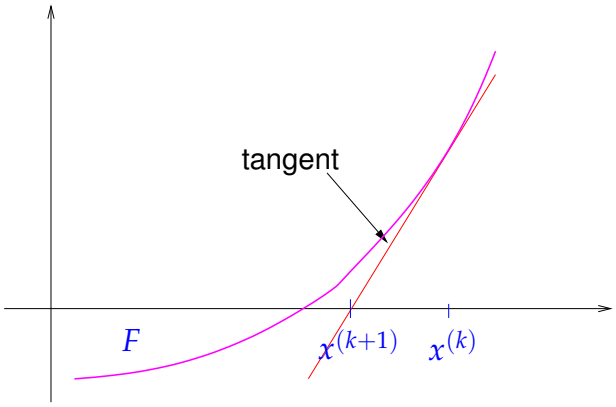
$$\widetilde{F}_k(x) := F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)})$$

take $x^{(k+1)}$:= zero of tangent

We obtain the Newton iteration

$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} , \qquad (8.3.4)$$

that requires $F'(x^{(k)}) \neq 0$.



C++11-code 8.3.5: Newton method in the scalar case $n = 1$

```
template <typename FuncType,typename DervType,typename Scalar>
Scalar newton1D(const FuncType &F,const DervType &DF,
                const Scalar &x0,double rtol,double atol)
{
  Scalar s,z = x0;
  do {
    s = F(z)/DF(z);  // compute Newton correction
    z -= s;          // compute next iterate
  }
  // correction based termination (relative and absolute)
  while ((std::abs(s) > rtol*std::abs(z)) && (std::abs(s) > atol));
  return (z);
}
```

Example 8.3.6 (Square root iteration as Newton's method)

In Ex. 8.1.20 we learned about the *quadratically convergent* fixed point iteration (8.1.21) for the approximate computation of the square root of a positive number. It can be derived as a Newton iteration (8.3.4)!

For $F(x) = x^2 - a$, $a > 0$, we find $F'(x) = 2x$, and, thus, the Newton iteration for finding zeros of F reads:

$$x^{(k+1)} = x^{(k)} - \frac{(x^{(k)})^2 - a}{2x^{(k)}} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right) ,$$

which is exactly (8.1.21). Thus, for this F Newton's method converges globally with order $p = 2$.

Example 8.3.7 (Newton method in 1D (→ Exp. 8.2.3))

Newton iterations for two different scalar non-linear equations $F(x) = 0$ with the same solution sets:

$$F(x) = xe^x - 1 \Rightarrow \quad F'(x) = e^x(1+x) \quad \Rightarrow \quad x^{(k+1)} = x^{(k)} - \frac{x^{(k)}e^{x^{(k)}} - 1}{e^{x^{(k)}}(1 + x^{(k)})} = \frac{(x^{(k)})^2 + e^{-x^{(k)}}}{1 + x^{(k)}}$$

$$F(x) = x - e^{-x} \quad \Rightarrow \quad F'(x) = 1 + e^{-x} \quad \Rightarrow \quad x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - e^{-x^{(k)}}}{1 + e^{-x^{(k)}}} = \frac{1 + x^{(k)}}{1 + e^{x^{(k)}}}$$

Exp. 8.2.3 confirms quadratic convergence in both cases! (→ Def. 8.1.17)

Note that for the computation of its zeros, the function F in this example can be recast in different forms!

In fact, based on Lemma 8.2.18 it is straightforward to show local quadratic convergence of Newton's method to a zero x^* of F, provided that $F'(x^*) \neq 0$:

Newton iteration (8.3.4) $\hat{=}$ fixed point iteration (→ Section 8.2) with iteration function

$$\Phi(x) = x - \frac{F(x)}{F'(x)} \quad \Rightarrow \quad \Phi'(x) = \frac{F(x)F''(x)}{(F'(x))^2} \quad \Rightarrow \quad \Phi'(x^*) = 0\,, \quad \text{if } F(x^*) = 0,\ F'(x^*) \neq 0\,.$$

Thus from Lemma 8.2.18 we conclude the following result:

Convergence of Newton's method in 1D

Newton's method locally quadratically converges (→ Def. 8.1.17) to a zero x^* of F, if $F'(x^*) \neq 0$

Example 8.3.9 (Implicit differentiation of F)

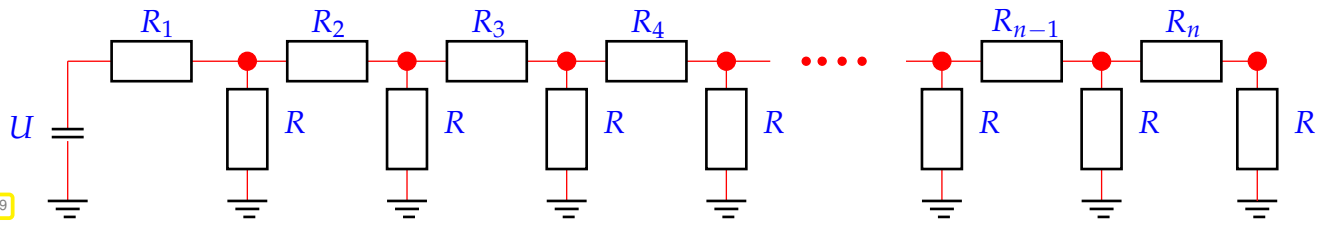


Fig. 289

How do we have to choose the leak resistance R in the linear circuit displayed in Fig. 289 in order to achieve a prescribed potential at one of the nodes?

Using nodal analysis of the circuit introduced in Ex. 2.1.3, this problem can be formulated as: find $x \in \mathbb{R}$, $x := R^{-1}$, such that

$$F(x) = 0 \quad \text{with} \quad F : \begin{cases} \mathbb{R} & \to & \mathbb{R} \\ x & \mapsto & \mathbf{w}^\top (\mathbf{A} + x\mathbf{I})^{-1}\mathbf{b} - 1 \end{cases}, \tag{8.3.10}$$

where $\mathbf{A} \in \mathbb{R}^{n,n}$ is a symmetric, tridiagonal, diagonally dominant matrix, $\mathbf{w} \in \mathbb{R}^n$ is a unit vector singling out the node of interest, and $\mathbf{b}$ takes into account the exciting voltage U.

In order to apply Newton's method to (8.3.10), we have to determine the derivative $F'(x)$ and so by implicit differentiation [**?**, Sect. 7.8], first rewriting ($\mathbf{u}(x) \hat{=}$ vector of nodal potentials as a function of $x = R^{-1}$)

$$F(x) = \mathbf{w}^\top \mathbf{u}(x) - 1 \quad , \quad (\mathbf{A} + x\mathbf{I})\mathbf{u}(x) = \mathbf{b}\,.$$

Then we differentiate the linear system of equations defining $\mathbf{u}(x)$ on both sides with respect to x using the product rule (8.4.10)

$$(\mathbf{A} + x\mathbf{I})\mathbf{u}(x) = \mathbf{b} \quad \xrightarrow{\frac{d}{dx}} \quad (\mathbf{A} + x\mathbf{I})\mathbf{u}'(x) + \mathbf{u}(x) = \mathbf{0}\,.$$

$$\blacktriangleright \quad \mathbf{u}'(x) = -(\mathbf{A}+x\mathbf{I})^{-1}\mathbf{u}(x)\ . \tag{8.3.11}$$

$$\blacktriangleright \quad F'(x) = \mathbf{w}^\top \mathbf{u}'(x) = -\mathbf{w}^\top(\mathbf{A}+x\mathbf{I})^{-1}\mathbf{u}(x)\ . \tag{8.3.12}$$

Thus, the Newton iteration for (8.3.10) reads:

$$\begin{aligned} x^{(k+1)} &= x^{(k)} + F'(x^{(k)})^{-1}F(x^{(k)}) \\ &= \frac{\mathbf{w}^\top \mathbf{u}(x^{(k)}) - 1}{\mathbf{w}^\top(\mathbf{A}+x^{(k)}\mathbf{I})^{-1}\mathbf{u}(x^{(k)})}\ , \quad (\mathbf{A}+x^{(k)}\mathbf{I})\mathbf{u}(x^{(k)}) = \mathbf{b}\ . \end{aligned} \tag{8.3.13}$$

In each step of the iteration we have to solve two linear systems of equations, which can be done with asymptotic effort $O(n)$ in this case, because $\mathbf{A}+x^{(k)}\mathbf{I}$ is tridiagonal.

Note that in a practical application one must demand $x > 0$, in addition, because the solution must provide a meaningful conductance (= inverse resistance.)

Also note that bisection ($\to$ 8.3.1) is a viable alternative to using Newton's method in this case.

8.3.2.2 Special one-point methods

Idea underlying other one-point methods: non-linear local approximation

Useful, if *a priori knowledge* about the structure of F (e.g. about F being a rational function, see below) is available. This is often the case, because many problems of 1D zero finding are posed for functions given in analytic form with a few parameters.

Prerequisite: Smoothness of F: $F \in C^m(I)$ for some $m > 1$

Example 8.3.14 (Halley's iteration $\to$ [?, Sect. 18.3])

This example demonstrates that non-polynomial model functions can offer excellent approximation of F. In this example the model function is chosen as a quotient of two linear function, that is, from the simplest class of true rational functions.

Of course, that this function provides a good model function is merely "a matter of luck", unless you have some more information about F. Such information might be available from the application context.

Given $x^{(k)} \in I$, next iterate := zero of model function: $h(x^{(k+1)}) = 0$, where

$$h(x) := \frac{a}{x+b} + c \quad \text{(rational function) such that } F^{(j)}(x^{(k)}) = h^{(j)}(x^{(k)})\ , \quad j = 0,1,2\ .$$

$$\Downarrow$$

$$\frac{a}{x^{(k)}+b} + c = F(x^{(k)})\ , \ -\frac{a}{(x^{(k)}+b)^2} = F'(x^{(k)})\ , \ \frac{2a}{(x^{(k)}+b)^3} = F''(x^{(k)})\ .$$

$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} \cdot \frac{1}{1 - \frac{1}{2}\frac{F(x^{(k)})F''(x^{(k)})}{F'(x^{(k)})^2}} \cdot$$

Halley's iteration for $F(x) = \frac{1}{(x+1)^2} + \frac{1}{(x+0.1)^2} - 1\,, \quad x > 0:$ and $x^{(0)} = 0$

k	$x^{(k)}$	$F(x^{(k)})$	$x^{(k)} - x^{(k-1)}$	$x^{(k)} - x^*$
1	0.19865959351191	10.90706835180178	-0.19865959351191	-0.84754290138257
2	0.69096314049024	0.94813655914799	-0.49230354697833	-0.35523935440424
3	1.02335017694603	0.03670912956750	-0.33238703645579	-0.02285231794846
4	1.04604398836483	0.00024757037430	-0.02269381141880	-0.00015850652965
5	1.04620248685303	0.00000001255745	-0.00015849848821	-0.00000000804145

Compare with Newton method (8.3.4) for the same problem:

k	$x^{(k)}$	$F(x^{(k)})$	$x^{(k)} - x^{(k-1)}$	$x^{(k)} - x^*$
1	0.04995004995005	44.38117504792020	-0.04995004995005	-0.99625244494443
2	0.12455117953073	19.62288236082625	-0.07460112958068	-0.92165131536375
3	0.23476467495811	8.57909346342925	-0.11021349542738	-0.81143781993637
4	0.39254785728080	3.63763326452917	-0.15778318232269	-0.65365463761368
5	0.60067545233191	1.42717892023773	-0.20812759505112	-0.44552704256257
6	0.82714994286833	0.46286007749125	-0.22647449053641	-0.21905255202615
7	0.99028203077844	0.09369191826377	-0.16313208791011	-0.05592046411604
8	1.04242438221432	0.00592723560279	-0.05214235143588	-0.00377811268016
9	1.04618505691071	0.00002723158211	-0.00376067469639	-0.00001743798377
10	1.04620249452271	0.00000000058056	-0.00001743761199	-0.00000000037178

Note that Halley's iteration is superior in this case, since F is a rational function.

! Newton method converges more slowly, but also needs less effort per step (→ Section 8.3.3)

In the previous example Newton's method performed rather poorly. Often its convergence can be boosted by converting the non-linear equation to an equivalent one (that is, one with the same solutions) for another function g, which is "closer to a linear function":

Assume $F \approx \widehat{F}$, where $\widehat{F}$ is invertible with an inverse $\widehat{F}^{-1}$ that can be evaluated with little effort.

$$\blacktriangleright \quad g(x) := \widehat{F}^{-1}(F(x)) \approx x\,.$$

Then apply Newton's method to $g(x)$, using the formula for the derivative of the inverse of a function

$$\frac{d}{dy}(\widehat{F}^{-1})(y) = \frac{1}{\widehat{F}'(\widehat{F}^{-1}(y))} \quad \Rightarrow \quad g'(x) = \frac{1}{\widehat{F}'(g(x))} \cdot F'(x)\,.$$

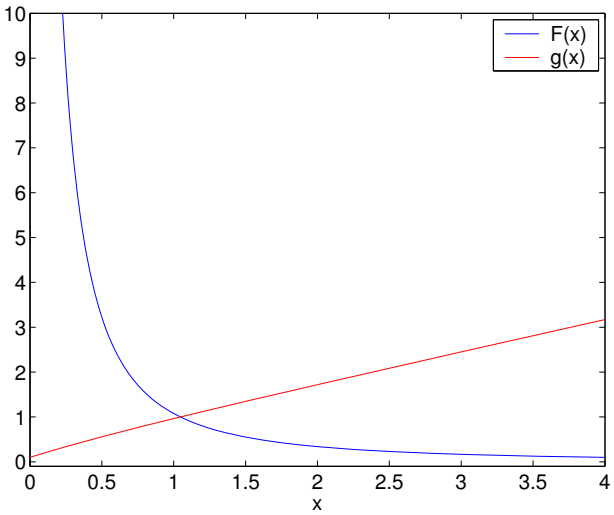
Example 8.3.15 (Adapted Newton method)

As in Ex. 8.3.14: $$F(x) = \frac{1}{(x+1)^2} + \frac{1}{(x+0.1)^2} - 1\,, \quad x > 0:$$

Observation:

$$F(x) + 1 \approx 2x^{-2} \text{ for } x \gg 1$$

and so $g(x) := \dfrac{1}{\sqrt{F(x)+1}}$ is "almost" linear for $x \gg 1$.



Idea: instead of $F(x) \stackrel{!}{=} 0$ tackle $g(x) \stackrel{!}{=} 1$ with Newton's method (8.3.4).

$$x^{(k+1)} = x^{(k)} - \frac{g(x^{(k)})-1}{g'(x^{(k)})} = x^{(k)} + \left(\frac{1}{\sqrt{F(x^{(k)})+1}} - 1\right)\frac{2(F(x^{(k)})+1)^{3/2}}{F'(x^{(k)})}$$

$$= x^{(k)} + \frac{2(F(x^{(k)})+1)(1-\sqrt{F(x^{(k)})+1})}{F'(x^{(k)})} \,.$$

Convergence recorded for $x^{(0)} = 0$:

k	$x^{(k)}$	$F(x^{(k)})$	$x^{(k)} - x^{(k-1)}$	$x^{(k)} - x^*$
1	0.91312431341979	0.24747993091128	0.91312431341979	-0.13307818147469
2	1.04517022155323	0.00161402574513	0.13204590813344	-0.00103227334125
3	1.04620244004116	0.00000008565847	0.00103221848793	-0.00000005485332
4	1.04620249489448	0.00000000000000	0.00000005485332	-0.00000000000000

For zero finding there is wealth of iterative methods that offer higher order of convergence. One class is discussed next.

(8.3.16) Modified Newton methods

Taking the cue from the iteration function of Newton's method (8.3.4), we extend it by introducing an extra function H:

$$\Phi(x) = x - \frac{F(x)}{F'(x)}H(x) \quad \text{with "proper" } H : I \mapsto \mathbb{R} \,.$$

Still, every zero of F is a fixed point of this Φ,that is, the fixed point iteration is still consistent ($\rightarrow$ Def. 8.2.1).

Aim: find H such that the method is of p-th order. The main tool is Lemma 8.2.18, which tells us that we have to ensure $\Phi^{(\ell)}(x^*) = 0$, $1 \le \ell \le p-1$, guarantees local convergence of order p.

Assume: F smooth "enough" and $\exists\, x^* \in I$: $F(x^*) = 0$, $F'(x^*) \neq 0$. Then we can compute the derivatives of Φ appealing to the product rule and quotient rule for derivatives.

$$\Phi = x - uH \quad , \quad \Phi' = 1 - u'H - uH' \quad , \quad \Phi'' = -u''H - 2u'H - uH'' \,,$$

$$\text{with } u = \frac{F}{F'} \quad \Rightarrow \quad u' = 1 - \frac{FF''}{(F')^2} \quad , \quad u'' = -\frac{F''}{F'} + 2\frac{F(F'')^2}{(F')^3} - \frac{FF'''}{(F')^2} \, .$$

$F(x^*) = 0 \quad \blacktriangleright \quad u(x^*) = 0, u'(x^*) = 1, u''(x^*) = -\frac{F''(x^*)}{F'(x^*)}$.

$$\blacktriangleright \quad \Phi'(x^*) = 1 - H(x^*) \quad , \quad \Phi''(x^*) = \frac{F''(x^*)}{F'(x^*)} H(x^*) - 2H'(x^*) \, . \tag{8.3.17}$$

Lemma 8.2.18 $\succ$ **Necessary** conditions for local convergence of order p:

$p = 2$ (quadratical convergence): $H(x^*) = 1$,

$p = 3$ (cubic convergence): $H(x^*) = 1 \quad \wedge \quad H'(x^*) = \frac{1}{2}\frac{F''(x^*)}{F'(x^*)}$.

Trial expression: $H(x) = G(1 - u'(x))$ with "appropriate" G

$$\blacktriangleright \quad \text{fixed point iteration} \quad x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} G\left(\frac{F(x^{(k)})F''(x^{(k)})}{(F'(x^{(k)}))^2} \right) . \tag{8.3.18}$$

Lemma 8.3.19. Cubic convergence of modified Newton methods

If $F \in C^2(I)$, $F(x^*) = 0$, $F'(x^*) \neq 0$, $G \in C^2(U)$ in a neighbourhood U of 0, $G(0) = 1$, $G'(0) = \frac{1}{2}$, then the fixed point iteration (8.3.18) *converge locally cubically to x^*.*

Proof. We apply Lemma 8.2.18, which tells us that both derivatives from (8.3.17) have to vanish. Using the definition of H we find.

$$H(x^*) = G(0) \quad , \quad H'(x^*) = -G'(0)u''(x^*) = G'(0)\frac{F''(x^*)}{F'(x^*)} \, .$$

Plugging these expressions into (8.3.17) finishes the proof. □

Experiment 8.3.20 (Application of modified Newton methods)

- $G(t) = \dfrac{1}{1 - \frac{1}{2}t}$ ➡ Halley's iteration ($\to$ Ex. 8.3.14)
- $G(t) = \dfrac{2}{1 + \sqrt{1 - 2t}}$ ➡ Euler's iteration
- $G(t) = 1 + \frac{1}{2}t$ ➡ quadratic inverse interpolation

Numerical experiment:

$F(x) = xe^x - 1$,
$x^{(0)} = 5$

k	$e^{(k)} := x^{(k)} - x^*$		
	Halley	Euler	Quad. Inv.
1	2.81548211105635	3.57571385244736	2.03843730027891
2	1.37597082614957	2.76924150041340	1.02137913293045
3	0.34002908011728	1.95675490333756	0.28835890388161
4	0.00951600547085	1.25252187565405	0.01497518178983
5	0.00000024995484	0.51609312477451	0.00000315361454
6		0.14709716035310	
7		0.00109463314926	
8		0.00000000107549	

8.3.2.3 Multi-point methods

Supplementary reading. The secant method is presented in [**?**, Sect. 18.2], [**?**, Sect. 5.5.3], [**?**, Sect. 3.4].

Construction of multi-point iterations in 1D

Idea: Replace F with interpolating polynomial

producing interpolatory model function methods

(8.3.22) The secant method

Simplest representative of model function multi-point methods:

secant method

$x^{(k+1)}$ = zero of secant (red line ▷)

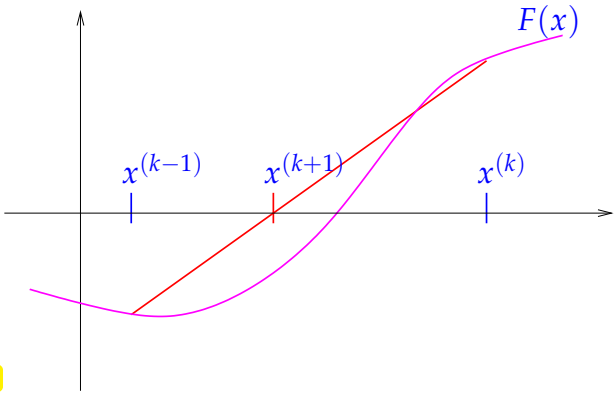


Fig. 290

The secant line is the graph of the function

$$s(x) = F(x^{(k)}) + \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}(x - x^{(k)}) , \tag{8.3.23}$$

$$\blacktriangleright \quad x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})(x^{(k)} - x^{(k-1)})}{F(x^{(k)}) - F(x^{(k-1)})} . \tag{8.3.24}$$

C++11 code 8.3.25: Secant method for 1D non-linear equaton

```
// Secand method for solving F(x) = 0 for F : D ⊂ ℝ → ℝ,
// initial guesses x_0, x_1,
// tolerances atol (absolute), rtol (relative)
template<typename Func>
double secant(double x0, double x1, Func&& F,
              double rtol, double atol, unsigned int maxIt)
{
  double fo = F(x0);
  for (unsigned int i=0; i<maxIt; ++i) {
      double fn = F(x1);
      double s = fn*(x1-x0)/(fn-fo); // secant correction
      x0 = x1; x1 = x1-s;
      // correction based termination (relative and absolute)
      if (abs(s) < max(atol, rtol*min(abs(x0),abs(x1))))
     return x1;
```

```
17          fo = fn;
18      }
19      return x1;
20  }
```

This code demonstrates several important features of the secant method:

- Only *one* function evaluation per step
- no derivatives required!
- 2-pojnt method: two initial guesses needed

Remember: $F(x)$ may only be available as output of a (complicated) procedure. In this case it is difficult to find a procedure that evaluates $F'(x)$. Thus the significance of methods that do not involve evaluations of derivatives.

Experiment 8.3.26 (Convergence of secant method)

Model problem: find zero of $F(x) = xe^x - 1$, using secant method of Code 8.3.25 with initial guesses $x^{(0)} = 0$, $x^{(1)} = 5$.

k	$x^{(k)}$	$F(x^{(k)})$	$e^{(k)} := x^{(k)} - x^*$	$\frac{\log\lvert e^{(k+1)}\rvert - \log\lvert e^{(k)}\rvert}{\log\lvert e^{(k)}\rvert - \log\lvert e^{(k-1)}\rvert}$
2	0.00673794699909	-0.99321649977589	-0.56040534341070	
3	0.01342122983571	-0.98639742654892	-0.55372206057408	24.43308649757745
4	0.98017620833821	1.61209684919288	0.41303291792843	2.70802321457994
5	0.38040476787948	-0.44351476841567	-0.18673852253030	1.48753625853887
6	0.50981028847430	-0.15117846201565	-0.05733300193548	1.51452723840131
7	0.57673091089295	0.02670169957932	0.00958762048317	1.70075240166256
8	0.56668541543431	-0.00126473620459	-0.00045787497547	1.59458505614449
9	0.56713970649585	-0.00000990312376	-0.00000358391394	1.62641838319117
10	0.56714329175406	0.00000000371452	0.00000000134427	
11	0.56714329040978	-0.00000000000001	-0.00000000000000	

Rem. 8.1.19: the rightmost column of the table provides an estimate for the order of convergence → Def. 8.1.17. For further explanations see Rem. 8.1.19.

A startling observation: the method seems to have a *fractional* (!) order of convergence, see Def. 8.1.17.

Remark 8.3.27 (Fractional order of convergence of secant method)

Indeed, a fractional order of convergence can be proved for the secant method, see [**?**, Sect. 18.2]. Here we give an asymptotic argument that holds, if the iterates are already very close to the zero x^* of F.

We can write the secant method in the form (8.1.4)

$$x^{(k+1)} = \Phi(x^{(k)}, x^{(k-1)}) \quad \text{with} \quad \Phi(x,y) = \Phi(x,y) := x - \frac{F(x)(x-y)}{F(x)-F(y)}\,. \tag{8.3.28}$$

Using Φ we find a recursion for the iteration error $e^{(k)} := x^{(k)} - x^*$:

$$e^{(k+1)} = \Phi(x^* + e^{(k)}, x^* + e^{(k-1)}) - x^* . \tag{8.3.29}$$

Thanks to the asymptotic perspective we may assume that $|e^{(k)}|, |e^{(k-1)}| \ll 1$ so that we can rely on two-dimensional Taylor expansion around $(x^*, x^{(*)})$, *cf.* [**?**, Satz 7.5.2]:

$$\begin{aligned} \Phi(x^*+h, x^*+k) = \Phi(x^*,x^*) + \frac{\partial\Phi}{\partial x}(x^*,x^*)h + \frac{\partial\Phi}{\partial y}(x^*,x^*)k + \\ \frac{1}{2}\frac{\partial^2\Phi}{\partial^2 x}(x^*,x^*)h^2 \frac{\partial^2\Phi}{\partial x\partial y}(x^*,x^*)hk \frac{1}{2}\frac{\partial^2\Phi}{\partial^2 y}(x^*,x^*)k^2 + R(x^*,h,k) , \\ \text{with} \quad |R| \le C(h^3 + h^2k + hk^2 + k^3) . \end{aligned} \tag{8.3.30}$$

Computations invoking the quotient rule and product rule and using $F(x^*) = 0$ show

$$\Phi(x^*,x^*) = x^* , \quad \frac{\partial\Phi}{\partial x}(x^*,x^*) = \frac{\partial\Phi}{\partial y}(x^*,x^*) = \frac{1}{2}\frac{\partial^2\Phi}{\partial^2 x}(x^*,x^*) = \frac{1}{2}\frac{\partial^2\Phi}{\partial^2 y}(x^*,x^*) = 0 .$$

We may also use MAPLE to find the Taylor expansion (assuming F sufficiently smooth):

```
> Phi := (x,y) -> x-F(x)*(x-y)/(F(x)-F(y));
> F(s) := 0;
> e2 = normal(mtaylor(Phi(s+e1,s+e0)-s,[e0,e1],4));
```

➢ truncated error propagation formula (products of three or more error terms ignored)

$$\boxed{e^{(k+1)} \doteq \frac{1}{2}\frac{F''(x^*)}{F'(x^*)} e^{(k)} e^{(k-1)} = C e^{(k)} e^{(k-1)}} . \tag{8.3.31}$$

How can we deduce the order of converge from this recursion formula? We try $e^{(k)} = K(e^{(k-1)})^p$ inspired by the estimate in Def. 8.1.17:

$$\Rightarrow \quad e^{(k+1)} = K^{p+1}(e^{(k-1)})^{p^2}$$

$$\Rightarrow \quad (e^{(k-1)})^{p^2-p-1} = K^{-p}C \quad \Rightarrow \quad p^2 - p - 1 = 0 \quad \Rightarrow \quad p = \frac{1}{2}(1 \pm \sqrt{5}) .$$

As $e^{(k)} \to 0$ for $k \to \infty$ we get the order of convergence $p = \frac{1}{2}(1+\sqrt{5}) \approx 1.62$ (see Exp. 8.3.26 !)

Example 8.3.32 (local convergence of the secant method)

Model problem: find zero of

$$F(x) = \arctan(x)$$

· $\hat{=}$ secant method converges for a pair $(x^{(0)}, x^{(1)}) \in \mathbb{R}^2_+$ of initial guesses. ▷

We observe that the secant method will converge only for initial guesses sufficiently close to 0 = local convergence → Def. 8.1.8

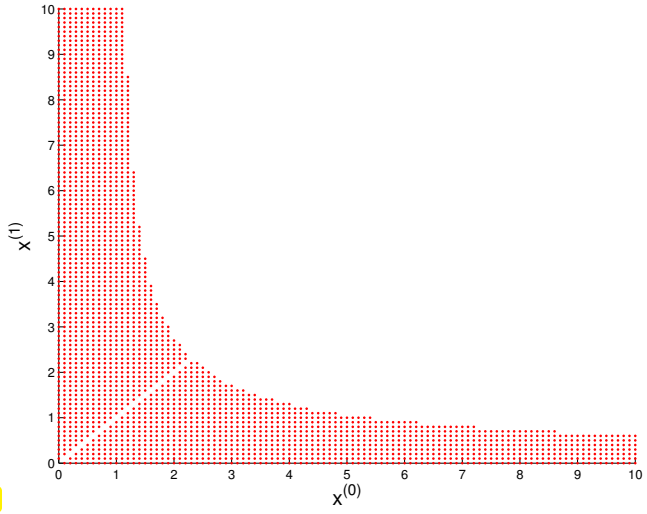
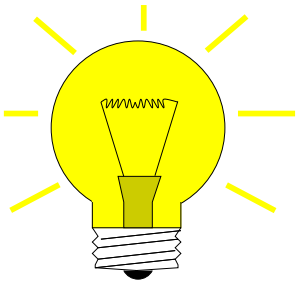


Fig. 291

(8.3.33) Inverse interpolation

Another class of multi-point methods: *inverse interpolation*



Assume: $F : I \subset \mathbb{R} \mapsto \mathbb{R}$ one-to-one (monotone)

$$F(x^*) = 0 \quad \Rightarrow \quad F^{-1}(0) = x^* .$$

- Interpolate F^{-1} by polynomial p of degree $m-1$ determined by

$$p(F(x^{(k-j)})) = x^{(k-j)} , \quad j = 0, \ldots, m-1 .$$

- New approximate zero $x^{(k+1)} := p(0)$

The graph of F^{-1} can be obtained by reflecting the graph of F at the angular bisector. ▷

$$F(x^*) = 0 \quad \Leftrightarrow \quad F^{-1}(0) = x^*$$

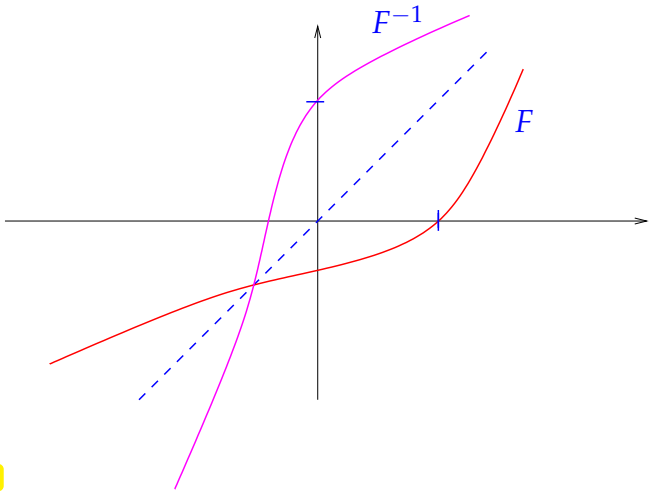


Fig. 292

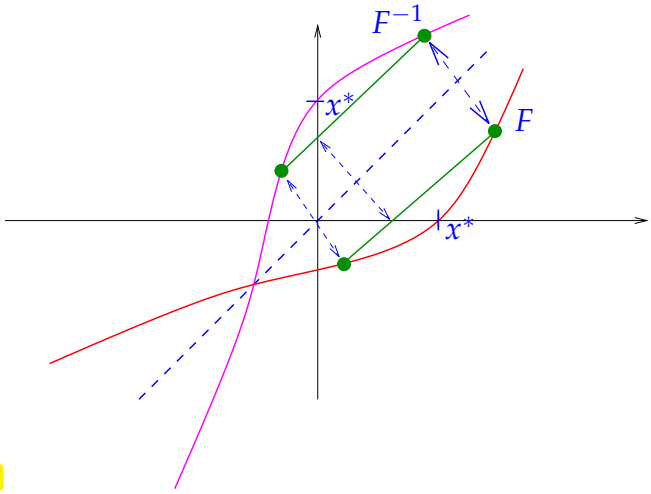


Fig. 293

Case $m = 2$ (2-point method)
➢ secant method

The interpolation polynomial is a line. In this case we do not get a new method, because the inverse function of a linear function (polynomial of degree 1) is again a polynomial of degree 1.

Case $m = 3$: quadratic inverse interpolation, a 3-point method, see [**?**, Sect. 4.5]

We interpolate the points $(F(x^{(k)}), x^{(k)}), (F(x^{(k-1)}), x^{(k-2)}), (F(x^{(k-1)}), x^{(k-2)})$ with a parabola (polynomial of degree 2). Note the importance of monotonicity of F, which ensures that $F(x^{(k)}), F(x^{(k-1)}), F(x^{(k-1)})$ are mutually different.

MAPLE code:
```
p := x-> a*x^2+b*x+c;
solve({p(f0)=x0,p(f1)=x1,p(f2)=x2},{a,b,c});
assign(%); p(0);
```

$$\blacktriangleright \quad x^{(k+1)} = \frac{F_0^2(F_1x_2 - F_2x_1) + F_1^2(F_2x_0 - F_0x_2) + F_2^2(F_0x_1 - F_1x_0)}{F_0^2(F_1 - F_2) + F_1^2(F_2 - F_0) + F_2^2(F_0 - F_1)} .$$

$$\left(F_0 := F(x^{(k-2)}), F_1 := F(x^{(k-1)}), F_2 := F(x^{(k)}), x_0 := x^{(k-2)}, x_1 := x^{(k-1)}, x_2 := x^{(k)} \right)$$

Experiment 8.3.34 (Convergence of quadratic inverse interpolation)

We test the method for the model problem/initial guesses $F(x) = xe^x - 1$, $x^{(0)} = 0$, $x^{(1)} = 2.5$, $x^{(2)} = 5$.

k	$x^{(k)}$	$F(x^{(k)})$	$e^{(k)} := x^{(k)} - x^*$	$\frac{\log\lvert e^{(k+1)}\rvert - \log\lvert e^{(k)}\rvert}{\log\lvert e^{(k)}\rvert - \log\lvert e^{(k-1)}\rvert}$
3	0.08520390058175	-0.90721814294134	-0.48193938982803	
4	0.16009252622586	-0.81211229637354	-0.40705076418392	3.33791154378839
5	0.79879381816390	0.77560534067946	0.23165052775411	2.28740488912208
6	0.63094636752843	0.18579323999999	0.06380307711864	1.82494667289715
7	0.56107750991028	-0.01667806436181	-0.00606578049951	1.87323264214217
8	0.56706941033107	-0.00020413476766	-0.00007388007872	1.79832936980454
9	0.56714331707092	0.00000007367067	0.00000002666114	1.84841261527097
10	0.56714329040980	0.00000000000003	0.00000000000001	

Also in this case the numerical experiment hints at a fractional rate of convergence $p \approx 1.8$, as in the case of the secant method, see Rem. 8.3.27.

8.3.3 Asymptotic efficiency of iterative methods for zero finding

Efficiency is measured by forming the ratio of gain and the effort required to achieve it. For iterative methods for solving $F(\mathbf{x}) = 0$, $F : D \subset \mathbb{R}^n \to \mathbb{R}^n$, this means the following:

Efficiency of an iterative method (for solving $F(\mathbf{x}) = 0$)	$\leftrightarrow$	computational effort to reach prescribed number of significant digits in result.

Ingredient ❶: $W \hat{=}$ computational effort per step

$$\left(\text{e.g.,} \quad W \approx \frac{\#\{\text{evaluations of } \mathrm{D}F\}}{\text{step}} + n \cdot \frac{\#\{\text{evaluations of } F'\}}{\text{step}} + \cdots \right)$$

Ingredient ❷: number of steps $k = k(\rho)$ to achieve *relative reduction of error* (= gain)

$$\left\| e^{(k)} \right\| \leq \rho \left\| e^{(0)} \right\| , \qquad \rho > 0 \text{ prescribed.} \tag{8.3.35}$$

Let us consider an iterative method of order $p \geq 1$ ($\rightarrow$ Def. 8.1.17). Its error recursion can be converted into expressions (8.3.36) and (8.3.37) that related the error norm $\left\|\mathbf{e}^{(k)}\right\|$ to $\left\|\mathbf{e}^{(0)}\right\|$ and lead to quantitative bounds for the number of steps to achieve (8.3.35):

$$\exists C > 0: \quad \left\|e^{(k)}\right\| \leq C\left\|e^{(k-1)}\right\|^p \quad \forall k \geq 1 \quad (C < 1 \text{ for } p = 1) .$$

Assuming $C\left\|e^{(0)}\right\|^{p-1} < 1$ (guarantees convergence!), we find the following minimum number of steps to achieve (8.3.35) for sure:

$$p = 1: \quad \left\|e^{(k)}\right\| \overset{!}{\leq} C^k\left\|e^{(0)}\right\| \quad \text{requires} \quad k \geq \frac{\log \rho}{\log C} , \tag{8.3.36}$$

$$p > 1: \quad \left\|e^{(k)}\right\| \overset{!}{\leq} C^{\frac{p^k-1}{p-1}}\left\|e^{(0)}\right\|^{p^k} \quad \text{requires} \quad p^k \geq 1 + \frac{\log \rho}{\log C/_{p-1} + \log(\left\|e^{(0)}\right\|)}$$

$$\Rightarrow \quad k \geq \log(1 + \frac{\log \rho}{\log L_0}) / \log p , \tag{8.3.37}$$

$$L_0 := C^{1/p-1}\left\|e^{(0)}\right\| < 1 .$$

Now we adopt an asymptotic perspective and ask for a large reduction of the error, that is $\rho \ll 1$.

If $\rho \ll 1$, then $\log(1 + \frac{\log \rho}{\log L_0}) \approx \log|\log \rho| - \log|\log L_0| \approx \log|\log \rho|$. This simplification will be made in the context of asymptotic considerations $\rho \to 0$ below.

Notice: $|\log \rho| \quad \leftrightarrow \quad$ Gain in no. of significant digits of $x^{(k)}$

Measure for efficiency:

$$\text{Efficiency} := \frac{\text{no. of digits gained}}{\text{total work required}} = \frac{|\log \rho|}{k(\rho) \cdot W} \tag{8.3.38}$$

asymptotic efficiency for $\rho \ll 1$ ➜ $|\log \rho| \to \infty$):

$$\text{Efficiency}_{|\rho \ll 1} = \begin{cases} -\dfrac{\log C}{W} & \text{, if } \quad p = 1 , \\ \dfrac{\log p}{W} \cdot \dfrac{|\log \rho|}{\log(|\log \rho|)} & \text{, if } \quad p > 1 . \end{cases} \tag{8.3.39}$$

We conclude that

- when requiring high accuracy, linearly convergent iterations should not be used, because their efficiency does not increase for $\rho \to 0$,
- for method of order $p > 1$, the factor $\frac{\log p}{W}$ offers a gauge for efficiency.

Example 8.3.40 (Efficiency of iterative methods)

We choose $\left\|e^{(0)}\right\| = 0.1$, $\rho = 10^{-8}$.

The plot displays the number of iteration steps according to (8.3.37).

Higher-order method require substantially fewer steps compared to low-order methods.

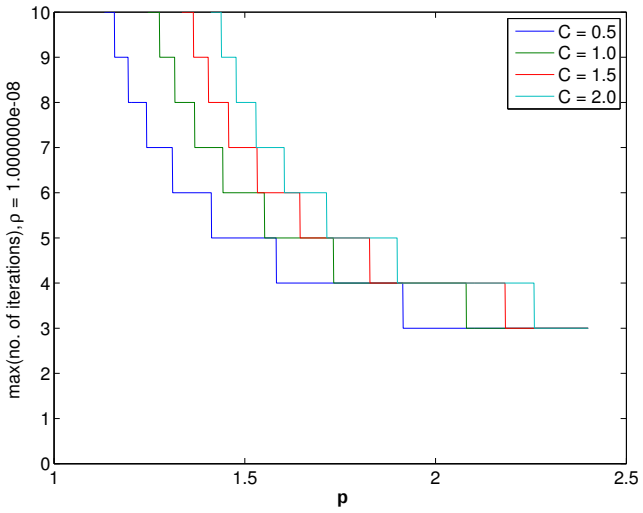


Fig. 294

We compare

Newton's method $\leftrightarrow$ secant method

in terms of number of steps required for a prescribed guaranteed error reduction, assuming $C = 1$ in both cases and for $\left\|e^{(0)}\right\| = 0.1$.

Newton's method requires only marginally fewer steps than the secant method.

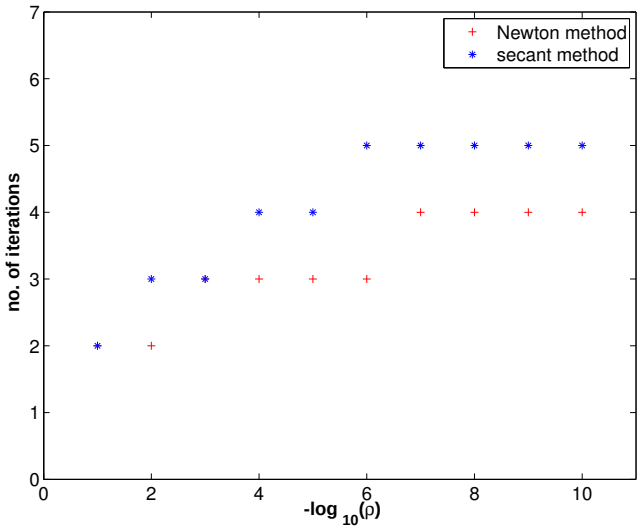


Fig. 295

We draw conclusions from the discussion above and (8.3.39):

$$\left.\begin{array}{l} W_{\text{Newton}} = 2W_{\text{secant}}\,, \\ p_{\text{Newton}} = 2,\ p_{\text{secant}} = 1.62 \end{array}\right\} \quad \Rightarrow \quad \frac{\log p_{\text{Newton}}}{W_{\text{Newton}}} : \frac{\log p_{\text{secant}}}{W_{\text{secant}}} = 0.71\,.$$

We set the effort for a step of Newton's method to twice that for a step of the secant method from Code 8.3.25, because we need an addition evaluation of F' in Newton's method.

➢ secant method is more efficient than Newton's method!

8.4 Newton's Method

Supplementary reading. A comprehensive monograph about all aspects of Newton's methods and generalizations in [**?**].

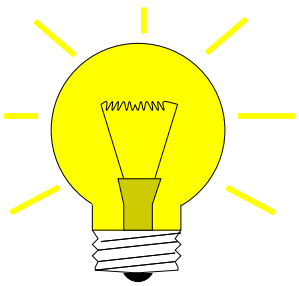
The multi-dimensional Newton method is also presented in [**?**, Sect. 19], [**?**, Sect. 5.6], [**?**, Sect. 9.1].

We consider a non-linear system of n equations with n unknowns:

for $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ find $\mathbf{x}^* \in D$: $F(\mathbf{x}^*) = 0$.

We assume: $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$ is continuously differentiable

8.4.1 The Newton iteration



Idea (→ Section 8.3.2.1): local linearization:

Given $\mathbf{x}^{(k)} \in D$ ➢ $\mathbf{x}^{(k+1)}$ as zero of affine linear model function

$$F(\mathbf{x}) \approx \widetilde{F}_k(\mathbf{x}) := F(\mathbf{x}^{(k)}) + \mathrm{D}F(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)}) ,$$

$\mathrm{D}F(\mathbf{x}) \in \mathbb{R}^{n,n}$ = Jacobian, $\mathrm{D}F(\mathbf{x}) := \left[\frac{\partial F_j}{\partial x_k}(\mathbf{x})\right]_{j,k=1}^{n}$.

◥ Newton iteration: (generalizes (8.3.4) to $n > 1$)

$$\boxed{\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - \mathrm{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)})} \quad , \quad [\text{ if } \mathrm{D}F(\mathbf{x}^{(k)}) \text{ regular }] \tag{8.4.1}$$

Terminology: $-\mathrm{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)})$ = Newton correction

Illustration of idea of Newton's method for $n = 2$: ▷

Sought: intersection point $\mathbf{x}^*$ of the curves $F_1(\mathbf{x}) = 0$ and $F_2(\mathbf{x}) = 0$.

Idea: $\mathbf{x}^{(k+1)} =$ the intersection of two straight lines (= zero sets of the components of the model function, *cf.* Ex. 2.2.15) that are approximations of the original curves

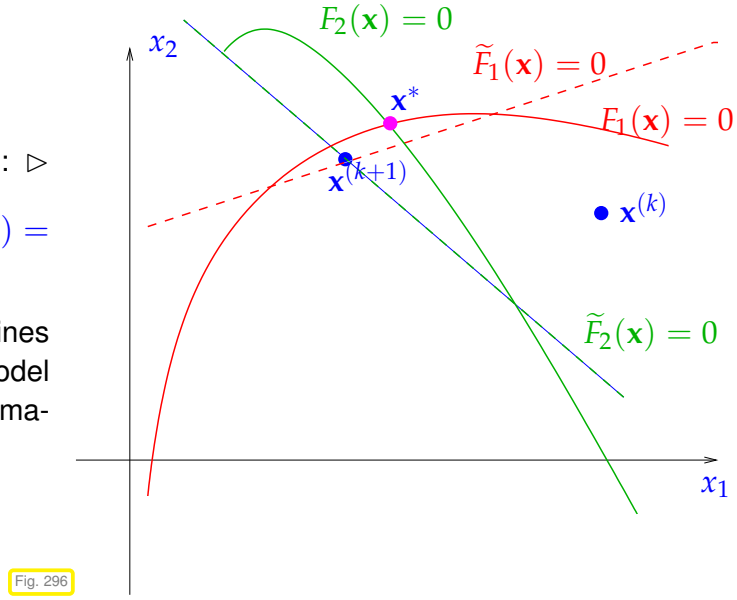


Fig. 296

MATLAB-code : Newton's method

MATLAB template for Newton method:

Solve linear system:

A\b = $\mathbf{A}^{-1}b$ → § 2.5.4

F, DF: function handles

A posteriori termination criterion

```
1 function x = newton(x,F,DF,rtol,atol)
2 for i=1:MAXIT
3   s = DF(x) \ F(x);
4   x = x-s;
5   if ((norm(s)<rtol*norm(x))||(norm(s)<atol))
6     return; end;
7 end
```

Here a correction based a posteriori termination criterion for the Newton iteration is used; it stops the iteration if the *relative* size of the Newton correction drops below the prescribed relative tolerance rtol.

If $\mathbf{x}^* \approx 0$ also the *absolute* size of the Newton correction has to be tested against an absolute tolerance `atol` in order to avoid non-termination despite convergence of the iteration,

C++11-code 8.4.2: Newton's method in C++

```
template <typename FuncType, typename JacType, typename Scalar,
          int N = Dynamic, typename CB=void_cb>
Vector<Scalar, N> newton(FuncType &F, JacType &DFinv,
                         const Vector<Scalar, N> x0, const double
                            rtol,
                         const double atol, CB callback=nullptr) {
  Vector<Scalar, N> x = x0;
  Vector<Scalar, N> s;

  do {
    s = DFinv(x,F(x));  // compute Newton correction
    x -= s;             // compute next iterate

    if (callback != nullptr)
        callback(x, s);
  }
  // correction based termination (relative and absolute)
  while ((s.norm() > rtol*x.norm()) && (s.norm() > atol));

  return x;
}
```

☞ Objects of type **FuncType** must feature

```
VecType operator(const VecType &x);
```

that evaluates $F(\mathbf{x})$ ($Vx \leftrightarrow$ `x`).

☞ Objects of type **JacType** must provide a method

```
VecType operator (const VecType &x,const VectType &f);
```

that computes the Newton correction, that is it returns the solution of a linear system with system matrix $DF(\mathbf{x})$ ($\mathbf{x} \leftrightarrow$ `x`) and right hand side $\mathbf{f} \leftrightarrow$ `f`.

☞ The argument `x` will be overwritten with the computed solution of the non-linear system.

The next code demonstrates the invocation of `newton` for a 2×2 non-linear system from a code relying on EIGEN. It also demonstrates the use of fixed size eigen matrices and vectors.

C++11-code 8.4.3: Calling `newton` with EIGEN data types

```
void newton2Ddriver(void)
{
  // Function F defined through lambda function
  auto F = [](const Eigen::Vector2d &x) {
    Eigen::Vector2d z;
    const double x1 = x(0),x2=x(1);
```

```
    z << x1*x1-2*x1-x2+1,x1*x1+x2*x2-1;
    return(z);
  };
  // JacType lambda function based on DF
  auto DF = [](const Eigen::Vector2d &x,const Eigen::Vector2d &f) {
    Eigen::Matrix2d J;
    const double x1 = x(0),x2=x(1);
    J << 2*x1-2,-1,2*x1,2*x2;
    Eigen::Vector2d s = J.lu().solve(f);
    return(s);
  };
  Eigen::Vector2d x; x << 2,3; % initial guess
  newton(F,DF,x,1E-6,1E-8);
  std::cout << "||F(x)|| = " << F(x).norm() << std::endl;
}
```

New aspect for $n \gg 1$ (compared to $n = 1$-dimensional case, Section 8.3.2.1):
Computation of the Newton correction may be expensive!
(because it involves the solution of a LSE, *cf.* Thm. 2.5.2)

Remark 8.4.4 (Affine invariance of Newton method)

An important property of the Newton iteration (8.4.1): affine invariance → [**?**, Sect .1.2.2]

set $G_{\mathbf{A}}(\mathbf{x}) := \mathbf{A}F(\mathbf{x})$ with regular $\mathbf{A} \in \mathbb{R}^{n,n}$ so that $F(\mathbf{x}^*) = 0 \Leftrightarrow G_{\mathbf{A}}(\mathbf{x}^*) = 0$.

Affine invariance: Newton iterations for $G_{\mathbf{A}}(\mathbf{x}) = 0$ are the same for all regular $\mathbf{A}$!

This is a simple computation:

$$\mathrm{D}\,G(\mathbf{x}) = \mathbf{A}\,\mathrm{D}\,F(\mathbf{x}) \quad \Rightarrow \quad \mathrm{D}\,G(\mathbf{x})^{-1}G(\mathbf{x}) = \mathrm{D}\,F(\mathbf{x})^{-1}\mathbf{A}^{-1}\mathbf{A}F(\mathbf{x}) = \mathrm{D}\,F(\mathbf{x})^{-1}F(\mathbf{x})\,.$$

Use affine invariance as guideline for

- convergence theory for Newton's method: assumptions and results should be affine invariant, too.
- modifying and extending Newton's method: resulting schemes should preserve affine invariance.

In particular, termination criteria for Newton's method should also be affine invariant in the sense that, when applied for $G_{\mathbf{A}}$ they STOP the iteration at exactly the same step for any choice of $\mathbf{A}$.

The function $F : \mathbb{R}^n \to \mathbb{R}^n$ defining the non-linear system of equations may be given in various formats, as explicit expression or rather implicitly. In most cases, $\mathrm{D}\,F$ has to be computed *symbolically* in order to obtain concrete formulas for the Newton iteration. We now learn how these symbolic computations can be carried out harnessing advanced techniques of multi-variate calculus.

(8.4.5) Derivatives and their role in Newton's method

The reader will probably agree that the derivative of a function $F : I \subset \mathbb{R} \to \mathbb{R}$ in $x \in I$ is a number $F'(x) \in \mathbb{R}$, the derivative of a function $F : D \subset \mathbb{R}^n \to \mathbb{R}^m$, in $\mathbf{x} \in D$ a matrix $\mathrm{D}F(\mathbf{x}) \in \mathbb{R}^{m,n}$. However, the nature of a derivative in a point is that of a linear mapping that approximates F locally up to second order:

Definition 8.4.6. Derivative of functions between vector spaces

Let V, W be finite dimensional vector spaces and $F : D \subset V \mapsto W$ a sufficiently smooth mapping. The derivative (differential) $\mathrm{D}F(\mathbf{x})$ of F in $\mathbf{x} \in V$ is the *unique* linear mapping $\mathrm{D}F(\mathbf{x}) : V \mapsto W$ such that there is a $\delta > 0$ and a function $\epsilon : [0, \delta] \to \mathbb{R}^+$ satisfying $\lim_{\xi \to 0} \epsilon(\xi) = 0$ such that

$$\|F(\mathbf{x} + \mathbf{h}) - F(\mathbf{x}) - \mathrm{D}F(\mathbf{x})\mathbf{h}\| = \epsilon(\|\mathbf{h}\|) \quad \forall \mathbf{h} \in V\,,\ \|\mathbf{h}\| < \delta\,. \tag{8.4.7}$$

The vector $\mathbf{h}$ in (8.4.7) may be called "direction of differentiation".

- ☞ Note that $\mathrm{D}F(\mathbf{x})\mathbf{h} \in W$ is the vector returned by the linear mapping $\mathrm{D}F(\mathbf{x})$ when applied to $\mathbf{h} \in V$.
- ☞ In Def. 8.4.6 $\|\cdot\|$ can be any norm on V ($\to$ Def. 1.5.70).
- ☞ A common shorthand notation for (8.4.7) relies on the "little-o" Landau symbol:

  $$\|F(\mathbf{x} + \mathbf{h}) - F(\mathbf{x}) - \mathrm{D}F(\mathbf{h})\mathbf{h}\| = o(\mathbf{h}) \quad \text{for} \quad \mathbf{h} \to 0\,,$$

  which designates a remainder term tending to 0 as its arguments tends to 0.
- ☞ Choosing bases of V and W, $\mathrm{D}F(\boldsymbol{x})$ can be described by the Jacobian (matrix) (8.2.11), because every linear mapping between finite dimensional vector spaces has a matrix representation after bases have been fixed. Thus, the derivative is usually written as a matrix-valued function on D.

In the context of the Newton iteration (8.4.1) the computation of the Newton correction $\mathbf{s}$ in the $k+1$-th step amounts to solving a linear system of equations:

$$\mathbf{s} = -\mathrm{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)}) \quad \Leftrightarrow \quad \mathrm{D}F(\mathbf{x}^{(k)})\mathbf{s} = -F(\mathbf{x}^{(k)})\,.$$

Matching this with Def. 8.4.6 we see that we need only determine expressions for $\mathrm{D}F(\mathbf{x}^{(k)})\mathbf{h}$, $\mathbf{h} \in V$, in order to state the LSE yielding the Newton correction. This will become important when applying the "compact" differentiation rules discussed next.

(8.4.8) Differentiation rules $\to$ Repetition of basic analysis skills

Statement of the Newton iteration (8.4.1) for $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ given as analytic expression entails computing the Jacobian $\mathrm{D}F$. The safe, but tedious way is to use the definition (8.2.11) directly and compute the partial derivatives.

To avoid cumbersome component-oriented considerations, it is sometimes useful to know the *rules of multidimensional differentiation*:

Immediate from Def. 8.4.6 are the following differentiation rules:

- For $F : V \mapsto W$ linear, we have $\mathrm{D}F(\mathbf{x}) = F$ for all $\mathbf{x} \in V$

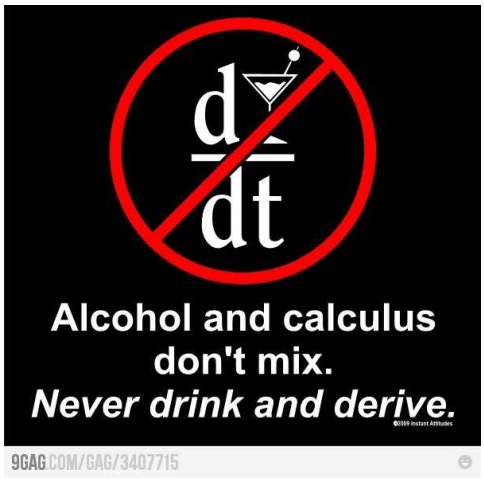
  (For instance, if $F : \mathbb{R}^n \to \mathbb{R}^m$, $F(\mathbf{x}) = \mathbf{A}\mathbf{x}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, then $\mathrm{D}F(\mathbf{x}) = \mathbf{A}$ for all $\mathbf{x} \in \mathbb{R}^n$.)
- Chain rule: For $F : V \mapsto W$, $G : W \mapsto U$ sufficiently smooth

$$\mathrm{D}(G \circ F)(\mathbf{x})\mathbf{h} = \mathrm{D}G(F(\mathbf{x}))(\mathrm{D}F(\mathbf{x}))\mathbf{h} \;, \quad \mathbf{h} \in V \,, \mathbf{x} \in D \;. \tag{8.4.9}$$

- Product rule: $F : D \subset V \mapsto W$, $G : D \subset V \mapsto U$ sufficiently smooth, $b : W \times U \mapsto Z$ bilinear, *ie.*, linear in each arguent:

$$T(\mathbf{x}) = b(F(\mathbf{x}), G(\mathbf{x})) \quad \Rightarrow \quad \mathrm{D}T(\mathbf{x})\mathbf{h} = b(\mathrm{D}F(\mathbf{x})\mathbf{h}, G(\mathbf{x})) + b(F(\mathbf{x}), \mathrm{D}G(\mathbf{x})\mathbf{h}) \;, \quad \mathbf{h} \in V, \mathbf{x} \in D \;. \tag{8.4.10}$$

Advice: If you do not feel comfortable with these rules of multidimensional differential calculus, please resort to detailed componentwise/entrywise calculations according to (8.2.11) ("pedestrian differentiation"), though they may be tedious.



The first and second derivatives of real-valued functions occur frequently and have special names, see [**?**, Def. 7.3.2] and [**?**, Satz 7.5.3].

Definition 8.4.11. Gradient and Hessian

For sufficiently smooth $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}$ the gradient $\mathbf{grad}\, F : D \mapsto \mathbb{R}^n$, and the Hessian (matrix) $\mathrm{H}F(\mathbf{x}) : D \mapsto \mathbb{R}^{n,n}$ are defined as

$$\left(\mathbf{grad}\, F(\mathbf{x})^T\right)\mathbf{h} := \mathrm{D}F(\mathbf{x})\mathbf{h} \;, \qquad \mathbf{h}_1^T \,\mathrm{H}F(\mathbf{x})\mathbf{h}_2 := \mathrm{D}(\mathrm{D}F(\mathbf{x})(\mathbf{h}_1))(\mathbf{h}_2) \;, \quad \mathbf{h}, \mathbf{h}_1, \mathbf{h}_2 \in V \;.$$

Example 8.4.12 (Derivative of a bilinear form)

A simple example: $\Psi : \mathbb{R}^n \mapsto \mathbb{R}$, $\Psi(\mathbf{x}) := \mathbf{x}^T\mathbf{A}\mathbf{x}$, with $\mathbf{A} \in \mathbb{R}^{n,n}$

This the general matrix representation of a bilinear form on $\mathbb{R}^n$.

"High level differentiation": We apply the product rule (8.4.10) with $F, G = Id$, which means $\mathrm{D}F(\mathbf{x}) = \mathrm{D}G(\mathbf{x}) = \mathbf{I}$, and the *bilinear form* $b(\mathbf{x}, \mathbf{y}) := \mathbf{x}^T\mathbf{A}\mathbf{y}$:

$$\blacktriangleright \quad \mathrm{D}\Psi(\mathbf{x})\mathbf{h} = \mathbf{h}^\top\mathbf{A}\mathbf{x} + \mathbf{x}^\top\mathbf{A}\mathbf{h} = \underbrace{\left(\mathbf{x}^\top\mathbf{A}^\top + \mathbf{x}^T\mathbf{A}\right)}_{=(\mathbf{grad}\,\Psi(\mathbf{x}))^\top}\mathbf{h} \;,$$

Hence, $\mathbf{grad}\,\Psi(\mathbf{x}) = (\mathbf{A}+\mathbf{A}^T)\mathbf{x}$ according to the definition of a gradient (→ Def. 8.4.11)

"Low level differentiation": Using the rules of matrix×vector multiplication, Ψ can be written in terms of the vector components x_i, $i = 1,\ldots,n$:

$$\Psi(\mathbf{x}) = \sum_{k=1}^{n}\sum_{j=1}^{n}(\mathbf{A})_{k,j}x_k x_j = (\mathbf{A})_{i,i}x_i^2 + \sum_{\substack{j=1\\ j\neq i}}^{n}(\mathbf{A})_{i,j}x_i x_j + \sum_{\substack{k=1\\ k\neq i}}^{n}(\mathbf{A})_{k,i}x_k x_i + \sum_{\substack{k=1\\ k\neq i}}^{n}\sum_{\substack{j=1\\ j\neq i}}^{n}(\mathbf{A})_{k,j}x_k x_j\,,$$

which leads to the partial derivatives

$$\begin{aligned}\frac{\partial\Psi(\mathbf{x})}{\partial x_i}(\mathbf{x}) &= 2(\mathbf{A})_{i,i}x_i + \sum_{\substack{j=1\\ j\neq i}}^{n}(\mathbf{A})_{i,j}x_j + \sum_{\substack{k=1\\ k\neq i}}^{n}(\mathbf{A})_{k,i}x_k = \sum_{j=1}^{n}(\mathbf{A})_{i,j}x_j + \sum_{k=1}^{n}(\mathbf{A})_{k,i}x_k\\ &= (\mathbf{A}\mathbf{x}+\mathbf{A}^\top\mathbf{x})_i\,.\end{aligned}$$

Of course, both results must agree!

Example 8.4.13 (Derivative of Euclidean norm)

We seek the derivative of the Euclidean norm, that is, of the function $F(\mathbf{x}) := \|\mathbf{x}\|_2$, $\mathbf{x}\in\mathbb{R}^n\setminus\{\mathbf{0}\}$ (F is defined but not differentiable in $\mathbf{x}=0$, just look at the case $n=1$!)

"High level differentiation": We can write F as the composition of two functions $F = G\circ H$ with

$$\begin{aligned}G &: \mathbb{R}_+ \to \mathbb{R}_+\,, \quad G(\xi) := \sqrt{\xi}\,,\\ H &: \mathbb{R}^n \to \mathbb{R}\,, \quad H(\mathbf{x}) := \mathbf{x}^\top\mathbf{x}\,.\end{aligned}$$

Using the rule for the differentiation of bilinear forms from Ex. 8.4.12 for the case $\mathbf{A}=\mathbf{I}$ and basic calculus, we find

$$\begin{aligned}\mathrm{D}\,H(\mathbf{x})\mathbf{h} &= 2\mathbf{x}^\top\mathbf{h}\,, \quad \mathbf{x},\mathbf{h}\in\mathbb{R}^n\,,\\ \mathrm{D}\,G(\xi)\zeta &= \frac{\zeta}{2\sqrt{\xi}}\,, \quad \xi>0,\ \zeta\in\mathbb{R}\,.\end{aligned}$$

Finally, the chain rule (8.4.9) gives

$$\mathrm{D}\,F(\mathbf{x})\mathbf{h} = \mathrm{D}\,G(H(\mathbf{x}))(\mathrm{D}\,H(\mathbf{x})\mathbf{h}) = \frac{2\mathbf{x}^\top\mathbf{h}}{2\sqrt{\mathbf{x}^\top\mathbf{x}}} = \frac{\mathbf{x}^\top}{\|\mathbf{x}\|_2}\cdot\mathbf{h}\,. \tag{8.4.14}$$

$$\text{Def. 8.4.11} \quad\Rightarrow\quad \mathbf{grad}\,F(\mathbf{x}) = \frac{\mathbf{x}}{\|\mathbf{x}\|_2}\,.$$

"Pedestrian differentiation": For a single component of $F(\mathbf{x})$ we have

$$\begin{aligned}&(F(\mathbf{x}))_i = \sqrt{x_1^2+x_2^2+\cdots+x_i^2+\cdots+x_n^2}\,,\\ \blacktriangleright\quad &\frac{\partial(F(\mathbf{x}))_i}{\partial x_i} = \frac{1}{2\sqrt{x_1^2+x_2^2+\cdots+x_i^2+\cdots+x_n^2}}\cdot 2x_i \quad \text{[chain rule]}\,.\\ \blacktriangleright\quad &\mathbf{grad}\,F(\mathbf{x}) = \frac{1}{\|\mathbf{x}\|_2}(x_1,x_2,\ldots,x_n)^\top\,.\end{aligned}$$

(8.4.15) Newton iteration via product rule

This paragraph explains the use of the general product rule (8.4.10) to derive the linear system solved by the Newton correction. It implements the insights from § 8.4.5.

We seek solutions of $F(\mathbf{x}) = \mathbf{0}$ with $F(\mathbf{x}) := b(G(\mathbf{x}), H(\mathbf{x}))$, where

- V, W are some vector spaces (finite- or even infinite-dimensional),
- $G : D \to V$, $H : D \to W$, $D \subset \mathbb{R}^n$, are continuously differentiable in the sense of Def. 8.4.6,
- $b : V \times W \mapsto \mathbb{R}^n$ is bilinear (linear in each argument).

According to the general product rule (8.4.10) we have

$$\mathrm{D}F(\mathbf{x})\mathbf{h} = b(\mathrm{D}G(\mathbf{x})\mathbf{h}, H(\mathbf{x})) + b(G(\mathbf{x}), \mathrm{D}H(\mathbf{x})\mathbf{h}) , \quad \mathbf{h} \in \mathbb{R}^n . \tag{8.4.16}$$

This already defines the linear system of equations to be solved to compute the Newton correction $\mathbf{s}$

$$b(\mathrm{D}G(\mathbf{x}^{(k)})\mathbf{s}, H(\mathbf{x}^{(k)})) + b(G(\mathbf{x}^{(k)}), \mathrm{D}H(\mathbf{x}^{(k)})\mathbf{s}) = -b(G(\mathbf{x}^{(k)}), H(\mathbf{x}^{(k)})) . \tag{8.4.17}$$

Since the left-hand side is linear in $\mathbf{s}$, this really represents a square linear system of n equations. The next example will present a concrete case.

(8.4.18) A "quasi-linear" system of equations

We call a quasilinear system of equations a non-linear equation of the form

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b} \quad \text{with } \mathbf{b} \in \mathbb{R}^n , \tag{8.4.19}$$

where $\mathbf{A} : D \subset \mathbb{R}^n \to \mathbb{R}^{n,n}$ is a matrix-valued function. On other words, a quasi-linear system is a "linear system of equations with solution-dependent system matrix". It incarnates a zero-finding problem for a function $F : \mathbb{R}^n \to \mathbb{R}^n$ with a bilinear structure as introduced in § 8.4.15.

For many quasi-linear systems, for which there exist solutions, the fixed point iteration (→ Section 8.2)

$$\mathbf{x}^{(k+1)} = \mathbf{A}(x^{(k)})^{-1}\mathbf{b} \quad \Leftrightarrow \quad \mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k+1)} = \mathbf{b} , \tag{8.4.20}$$

provides a convergent iteration, provided that a good initial guess is available.

We can also reformulate

$$(8.4.19) \quad \Leftrightarrow \quad F(\mathbf{x}) = \mathbf{0} \quad \text{with} \quad F(\mathbf{x}) = \mathbf{A}(\mathbf{x})\mathbf{x} - \mathbf{b} .$$

If $\mathbf{x} \mapsto \mathbf{A}(\mathbf{x})$ is differentiable, the product rule (8.4.10) yields

$$\mathrm{D}F(\mathbf{x})\mathbf{h} = (\mathrm{D}\mathbf{A}(\mathbf{x})\mathbf{h})\mathbf{x} + \mathbf{A}(\mathbf{x})\mathbf{h} , \quad \mathbf{h} \in \mathbb{R}^n . \tag{8.4.21}$$

Note that $\mathrm{D}\mathbf{A}(\mathbf{x}^{(k)})$ is a mapping from $\mathbb{R}^n$ into $\mathbb{R}^{n,n}$, which gets $\mathbf{h}$ as an argument. Then the Newton iteration reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{s} , \quad \mathrm{D}F(\mathbf{x})\mathbf{s} = (\mathrm{D}\mathbf{A}(\mathbf{x}^{(k)})\mathbf{s})\mathbf{x}^{(k)} + \mathbf{A}(\mathbf{x}^{(k)})\mathbf{s} = \mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b} . \tag{8.4.22}$$

The next example will examine a concrete quasi-linear system of equations.

Example 8.4.23 (A special quasi-linear system of equations)

We consider the quasi-linear system of equations

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b} \quad , \quad \mathbf{A}(\mathbf{x}) = \begin{bmatrix} \gamma(\mathbf{x}) & 1 & & & & \\ 1 & \gamma(\mathbf{x}) & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & \gamma(\mathbf{x}) & 1 \\ & & & & 1 & \gamma(\mathbf{x}) \end{bmatrix} \in \mathbb{R}^{n\times n} \, , \tag{8.4.24}$$

where $\gamma(\mathbf{x}) := 3 + \|\mathbf{x}\|_2$ (Euclidean vector norm), the right hand side vector $\mathbf{b} \in \mathbb{R}^n$ is given and $\mathbf{x} \in \mathbb{R}^n$ is unknown.

In order to compute the derivative of $F(\mathbf{x}) := \mathbf{A}(\mathbf{x})\mathbf{x} - \mathbf{b}$ it is advisable to rewrite

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{T}\mathbf{x} + \mathbf{x}\|\mathbf{x}\|_2 \, , \quad \mathbf{T} := \begin{bmatrix} 3 & 1 & & & & \\ 1 & 3 & 1 & & & \\ & \ddots & 3 & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 3 & 1 \\ & & & & 1 & 3 \end{bmatrix} .$$

The derivative of the first term is straightforward, because it is linear in $\mathbf{x}$, see the discussion following Def. 8.4.6.

The "pedestrian" approach to the second term starts with writing it explicitly in components as

$$(\mathbf{x}\|\mathbf{x}\|)_i = x_i\sqrt{x_1^2 + \cdots + x_n^2} \, , \quad i = 1, \ldots, n \, .$$

Then we can compute the Jacobian according to (8.2.11) by taking partial derivatives:

$$\frac{\partial}{\partial x_i}(\mathbf{x}\|\mathbf{x}\|)_i = \sqrt{x_1^2 + \cdots + x_n^2} + x_i\frac{x_i}{\sqrt{x_1^2 + \cdots + x_n^2}} \, ,$$

$$\frac{\partial}{\partial x_j}(\mathbf{x}\|\mathbf{x}\|)_i = x_i\frac{x_j}{\sqrt{x_1^2 + \cdots + x_n^2}} \, , \quad j \neq i \, .$$

For the "high level" treatment of the second term $\mathbf{x} \mapsto \mathbf{x}\|\mathbf{x}\|_2$ we apply the product rule (8.4.10), together with (8.4.14):

$$DF(\mathbf{x})\mathbf{h} = \mathbf{T}\mathbf{h} + \|\mathbf{x}\|_2\mathbf{h} + \mathbf{x}\frac{\mathbf{x}^\top\mathbf{h}}{\|\mathbf{x}\|_2} = \big(\mathbf{A}(\mathbf{x}) + \frac{\mathbf{x}\mathbf{x}^\top}{\|\mathbf{x}\|_2}\big)\mathbf{h} \, .$$

Thus, in concrete terms the Newton iteration (8.4.22) becomes

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \Big(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}\|_2}\Big)^{-1}(\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \, .$$

Note that that matrix of the linear system to be solved in each step is a rank-1-modification (2.6.17) of the symmetric positive definite tridiagonal matrix $\mathbf{A}(\mathbf{x}^{(k)})$, *cf.* Lemma 2.8.12. Thus the Sherman-Morrison-Woodbury formula from Lemma 2.6.22 can be used to solve it efficiently.

(8.4.25) Implicit differentiation of bilinear expressions

Given are

- a finite-dimensional vector space W,
- a continuously differentiable function $G : D \subset \mathbb{R}^n \to V$ into some vector space V,
- a bilinear mapping $b : V \times W \to W$.

Let $F : D \to W$ be *implicitly defined* by

$$F(\mathbf{x}): \qquad b(G(\mathbf{x}), F(\mathbf{x})) = \mathbf{b} \in W \,. \tag{8.4.26}$$

This relationship will provide a valid definition of F in a neighborhood of $\mathbf{x}_0 \in W$, if we assume that there is $\mathbf{x}_0, \mathbf{z}_0 \in W$ such that $b(G(\mathbf{x}_0), z_0) = \mathbf{b}$, and that the linear mapping $\mathbf{z} \mapsto b(G(\mathbf{x}_0), \mathbf{z})$ is *invertible*. Then, for $\mathbf{x}$ close to $\mathbf{x}_0$, $F(\mathbf{x})$ can be computed by solving a square linear system of equations in W. In Ex. 8.3.9 we already saw an example of an implicitly defined F for $W = \mathbb{R}$.

We want to solve $F(\mathbf{x}) = \mathbf{0}$ for this implicitly defined F by means of Newton's method. In order to determine the derivative of F we resort to implicit differentiation [**?**, Sect. 7.8] of the defining equation (8.4.26) by means of the general product rule (8.4.10). We formally differentiate both sides of (8.4.26):

$$b(\mathrm{D}\, G(\mathbf{x})\mathbf{h}, F(\mathbf{x})) + b(G(\mathbf{x}), \mathrm{D}\, F(\mathbf{x})\mathbf{h}) = \mathbf{0} \quad \forall \mathbf{h} \in W \,, \tag{8.4.27}$$

and find that the Newton correction $\mathbf{s}$ in the $k+1$-th Newton step can be computed as follows:

$$\begin{aligned} & \mathrm{D}\, F(\mathbf{x}^{(k)})\mathbf{s} = -F(\mathbf{x}^{(k)}) \quad \Rightarrow \quad b(G(\mathbf{x}^{(k)}), \mathrm{D}\, F(\mathbf{x}^{(k)})\mathbf{s}) = -b(G(\mathbf{x}^{(k)}), F(\mathbf{x}^{(k)})) \\ \overset{(8.4.27)}{\Rightarrow} \quad & b(\mathrm{D}\, G(\mathbf{x}^{(k)})\mathbf{s}, F(\mathbf{x}^{(k)})) = b(G(\mathbf{x}^{(k)}), F(\mathbf{x}^{(k)})) \,, \end{aligned}$$

which constitutes an $\dim W \times \dim W$ linear system of equations. The next example discusses a concrete application of implicit differentiation with $W = \mathbb{R}^{n,n}$.

Example 8.4.28 (Derivative of matrix inversion)

We consider matrix inversion as a mapping and (formally) compute its derivative, that is, the derivative of function

$$\mathrm{inv} : \begin{cases} \mathbb{R}^{n,n}_* & \to & \mathbb{R}^{n,n} \\ \mathbf{X} & \mapsto & \mathbf{X}^{-1} \end{cases} \,,$$

where $\mathbb{R}^{n,n}_*$ denotes the (open) set of invertible $n \times n$-matrices, $n \in \mathbb{N}$.

We apply the technique of implicit differentiation from § 8.4.25 to the equation

$$\mathrm{inv}(\mathbf{X}) \cdot \mathbf{X} = \mathbf{I} \,, \quad \mathbf{X} \in \mathbb{R}^{n,n}_* \,. \tag{8.4.29}$$

Differentiation on both sides of (8.4.29) by means of the product rule (8.4.10) yields

$$\mathrm{D}\,\mathrm{inv}(\mathbf{X})\mathbf{H}\cdot\mathbf{X}+\mathrm{inv}(\mathbf{X})\cdot\mathbf{H}=\mathbf{O}\;,\quad \mathbf{H}\in\mathbb{R}^{n,n}\;,$$

$$\Longrightarrow\quad \boxed{\mathrm{D}\,\mathrm{inv}(\mathbf{X})\mathbf{H}=-\mathbf{X}^{-1}\mathbf{H}\mathbf{X}^{-1}}\;,\quad \mathbf{H}\in\mathbb{R}^{n,n}\;. \tag{8.4.30}$$

For $n=1$ we get $\mathrm{D}\,\mathrm{inv}(x)h=-\frac{h}{x^2}$, which recovers the well-known derivative of the function $x\to x^{-1}$.

Example 8.4.31 (Matrix inversion by means of Newton's method → [?, ?])

Surprisingly, it is possible to obtain the inverse of a matrix as a the solution of a non-linear system of equations. Thus it can be computed using Newton's method.

Given a regular matrix $\mathbf{A}\in\mathbb{R}^{n,n}$, its inverse can be defined as the unique zero of a function:

$$\mathbf{X}=\mathbf{X}^{-1}\quad\Longleftrightarrow\quad F(\mathbf{X})=0\quad\text{for}\quad F:\left\{\begin{array}{rcl}\mathbb{R}^{n,n}_{*}&\to&\mathbb{R}^{n,n}\\ \mathbf{X}&\mapsto&\mathbf{A}-\mathbf{X}^{-1}\end{array}\right.\;.$$

Using (8.4.30) we find for the derivative of F in $\mathbf{X}\in\mathbb{R}^{n,n}_{*}$

$$\mathrm{D}\,F(\mathbf{X})\mathbf{H}=\mathbf{X}^{-1}\mathbf{H}\mathbf{X}^{-1}\;,\quad \mathbf{H}\in\mathbb{R}^{n,n}\;. \tag{8.4.32}$$

The abstract Newton iteration (8.4.1) for F reads

$$\mathbf{X}^{(k+1)}=\mathbf{X}^{(k)}-\mathbf{S}\;,\quad \mathbf{S}:=\mathrm{D}\,F(\mathbf{X}^{(k)})^{-1}F(\mathbf{X}^{(k)})\;. \tag{8.4.33}$$

The Newton correction $\mathbf{S}$ in the k-th step solves the *linear system of equations*

$$\mathrm{D}\,F(\mathbf{X}^{(k)})\mathbf{S}\overset{(8.4.32)}{=}\left(\mathbf{X}^{(k)}\right)^{-1}\mathbf{S}\left(\mathbf{X}^{(k)}\right)^{-1}=F(\mathbf{X}^{(k)})=\mathbf{A}-\left(\mathbf{X}^{(k)}\right)^{-1}\;.$$

$$\Longrightarrow\quad \mathbf{S}=\mathbf{X}^{(k)}(\mathbf{A}-\left(\mathbf{X}^{(k)}\right)^{-1})\mathbf{X}^{(k)}=\mathbf{X}^{(k)}\mathbf{A}\mathbf{X}^{(k)}-\mathbf{X}^{(k)}\;. \tag{8.4.34}$$

$$\overset{\text{in (8.4.33)}}{\Longrightarrow}\quad \boxed{\mathbf{X}^{(k+1)}=\mathbf{X}^{(k)}-\left(\mathbf{X}^{(k)}\mathbf{A}\mathbf{X}^{(k)}-\mathbf{X}^{(k)}\right)=\mathbf{X}^{(k)}\left(2\mathbf{I}-\mathbf{A}\mathbf{X}^{(k)}\right)}\;. \tag{8.4.35}$$

To study the convergence of this iteration we derive a recursion for the iteration errors $\mathbf{E}^{(k)}:=\mathbf{X}^{(k)}-\mathbf{A}^{-1}$:

$$\begin{aligned}\mathbf{E}^{(k+1)}&=\mathbf{X}^{(k+1)}-\mathbf{A}^{-1}\\ &\overset{(8.4.35)}{=}\mathbf{X}^{(k)}\left(2\mathbf{I}-\mathbf{A}\mathbf{X}^{(k)}\right)-\mathbf{A}^{-1}\\ &=(\mathbf{E}^{(k)}+\mathbf{A}^{-1})\left(2\mathbf{I}-\mathbf{A}(\mathbf{E}^{(k)}+\mathbf{A}^{-1})\right)-\mathbf{A}^{-1}\\ &=(\mathbf{E}^{(k)}+\mathbf{A}^{-1})(\mathbf{I}-\mathbf{A}\mathbf{E}^{(k)})-\mathbf{A}^{-1}=-\mathbf{E}^{(k)}\mathbf{A}\mathbf{E}^{(k)}\;.\end{aligned}$$

For the norm of the iteration error (a matrix norm → Def. 1.5.76) we conclude from submultiplicativity (1.5.77) a recursive estimate

$$\left\|\mathbf{E}^{(k+1)}\right\|\le\left\|\mathbf{E}^{(k)}\right\|^2\|\mathbf{A}\|\;. \tag{8.4.36}$$

This holds for any matrix norm according to Def. 1.5.76, which is induced by a vector norm. For the relative iteration error we obtain

$$\underbrace{\frac{\left\|\mathbf{E}^{(k+1)}\right\|}{\|\mathbf{A}\|}}_{\text{relative error}}\le\left(\underbrace{\frac{\left\|\mathbf{E}^{(k)}\right\|}{\|\mathbf{A}\|}}_{\text{relative error}}\right)^2\underbrace{\|\mathbf{A}\|\left\|\mathbf{A}^{-1}\right\|}_{=\mathrm{cond}(\mathbf{A})}\;, \tag{8.4.37}$$

where the condition number is defined in Def. 2.2.12.

From (8.4.36) we conclude that the iteration will converge ($\lim_{k\to\infty}\left\|\mathbf{E}^{(k)}\right\| = 0$), if

$$\left\|\mathbf{E}^{(0)}\mathbf{A}\right\| = \left\|\mathbf{X}^{(0)}\mathbf{A} - \mathbf{I}\right\| < 1\,, \tag{8.4.38}$$

which gives a condition on the initial guess $\mathbf{S}^{(0)}$. Now let us consider the Euclidean matrix norm $\|\cdot\|_2$, which can be expressed in terms of eigenvalues, see Cor. 1.5.82. Motivated by this relationship, we use the initial guess $\mathbf{X}^{(0)} = \alpha\mathbf{A}^\top$ with $a > 0$ still to be determined.

$$\left\|\mathbf{X}^{(0)}\mathbf{A} - \mathbf{I}\right\|_2 = \left\|\alpha\mathbf{A}^\top\mathbf{A} - \mathbf{I}\right\|_2 = \alpha\|\mathbf{A}\|_2^2 - 1 \overset{!}{<} 1 \quad\Leftrightarrow\quad \alpha < \frac{2}{\|\mathbf{A}\|_2^2}\,,$$

which is a sufficient condition for the initial guess $\mathbf{X}^{(0)} = \alpha\mathbf{A}^\top$, in order to make (8.4.35) converge. In this case we infer quadratic convergence from both (8.4.36) and (8.4.37).

Remark 8.4.39 (Simplified Newton method [?, Sect. 5.6.2])

C++11 code 8.4.40: Efficient implementation of simplified Newton method

```
// C++ template for simplified Newton method
template<typename Func, typename Jac, typename Vec>
void simpnewton(Vec& x, Func F, Jac DF, double rtol, double atol)
{
  auto lu = DF(x).lu(); // do LU decomposition once!
  Vec s;                // Newton correction
  double ns,nx;         // auxiliary variables for termination control
  do {
    s = lu.solve(F(x));
    x = x-s;            // new iterate
    ns = s.norm(); nx = x.norm();
  }
  // termination based on relative and absolute tolerance
  while((ns > rtol*nx) && (ns > atol));
}
```

Simplified Newton Method: ☞ use the same Jacobian $\mathrm{D}F(\mathbf{x}^{(k)})$ for all/several steps

Possible to reuse of LU-decomposition, *cf.* Rem. 2.5.10.

➢ (usually) merely linear convergence instead of quadratic convergence

Remark 8.4.41 (Numerical Differentiation for computation of Jacobian)

If $\mathrm{D}F(\mathbf{x})$ is not available (e.g. when $F(\mathbf{x})$ is given only as a procedure) we may resort to approximation by difference quotients:

Numerical Differentiation: $$\frac{\partial F_i}{\partial x_j}(\mathbf{x}) \approx \frac{F_i(\mathbf{x} + h\vec{e}_j) - F_i(\mathbf{x})}{h}\,.$$

Caution: Roundoff errors wreak havoc for small h → Ex. 1.5.45 **!** Therefore use $h \approx \sqrt{\mathrm{EPS}}$.

8.4.2 Convergence of Newton's method

Newton iteration (8.4.1) $\hat{=}$ fixed point iteration (→ Section 8.2) with iteration function

$$\Phi(\mathbf{x}) = \mathbf{x} - \mathrm{D}F(\mathbf{x})^{-1}F(\mathbf{x}) .$$

["product rule" (8.4.10) : $\mathrm{D}\Phi(\mathbf{x}) = \mathbf{I} - \mathrm{D}(\{\mathbf{x} \mapsto \mathrm{D}F(\mathbf{x})^{-1}\})F(\mathbf{x}) - \mathrm{D}F(\mathbf{x})^{-1}\,\mathrm{D}F(\mathbf{x})$]

$$F(\mathbf{x}^*) = 0 \quad \Rightarrow \quad \mathrm{D}\Phi(\mathbf{x}^*) = 0 ,$$

that is, the derivative (Jacobian) of the iteration function of the Newton fixed point iteration vanishes in the limit point. Thus from Lemma 8.2.18 we draw the same conclusion as in the scalar case $n = 1$, *cf.* Section 8.3.2.1.

Local quadratic convergence of Newton's method, if $\mathrm{D}F(\mathbf{x}^*)$ regular

Experiment 8.4.42 (Convergence of Newton's method in 2D)

We study the convergence of Newton's method empirically for $n = 2$ for

$$F(\mathbf{x}) = \begin{bmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2 \quad \text{with solution} \quad F(\begin{bmatrix} 1 \\ 1 \end{bmatrix}) = 0 . \tag{8.4.43}$$

Jacobian (analytic computation): $\mathrm{D}F(\mathbf{x}) = \begin{bmatrix} \partial_{x_1}F_1(x) & \partial_{x_2}F_1(x) \\ \partial_{x_1}F_2(x) & \partial_{x_2}F_2(x) \end{bmatrix} = \begin{bmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{bmatrix}$

Realization of Newton iteration (8.4.1):

1. Solve LSE
$$\begin{bmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{bmatrix} \Delta\mathbf{x}^{(k)} = -\begin{bmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{bmatrix},$$
where $\mathbf{x}^{(k)} = [x_1, x_2]^T$.
2. Set $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$.

MATLAB-code 8.4.44: Newton iteration for (8.4.43)

```
F=@(x) [x(1)^2-x(2)^4; x(1)-x(2)^3];
DF=@(x) [2*x(1),-4*x(2)^3;1,-3*x(2)^2];
x=[0.7;0.7]; x_ast=[1;1]; tol=1E-10;

res=[0,x',norm(x-x_ast)];
s = DF(x)\F(x); x = x-s;
res = [res; 1,x',norm(x-x_ast)]; k=2;
while (norm(s) > tol*norm(x))
  s = DF(x)\F(x); x = x-s;
  res = [res; k,x',norm(x-x_ast)];
```

```
11    k = k+1;
12  end
13
14  ld = diff(log(res(:,4))); %
15  rates = ld(2:end)./ld(1:end-1); %
```

Line 14, Line 15: estimation of order of convergence, see Rem. 8.1.19.

k	$\mathbf{x}^{(k)}$	$\epsilon_k := \|\mathbf{x}^* - \mathbf{x}^{(k)}\|_2$	$\frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}}$
0	$[0.7, \quad 0.7]^T$	4.24e-01	
1	$[0.87850000000000, \quad 1.064285714285714]^T$	1.37e-01	1.69
2	$[1.01815943274188, \quad 1.00914882463936]^T$	2.03e-02	2.23
3	$[1.00023355916300, \quad 1.00015913936075]^T$	2.83e-04	2.15
4	$[1.00000000583852, \quad 1.00000002726552]^T$	2.79e-08	1.77
5	$[0.999999999999998, \quad 1.000000000000000]^T$	2.11e-15	
6	$[1, \quad 1]^T$		

☞ (Some) evidence of quadratic convergence, see Rem. 8.1.19.

There is a sophisticated theory about the convergence of Newton's method. For example one can find the following theorem in [**?**, Thm. 4.10], [**?**, Sect. 2.1]):

Theorem 8.4.45. Local quadratic convergence of Newton's method

If:

(A) $D \subset \mathbb{R}^n$ *open and convex,*

(B) $F : D \mapsto \mathbb{R}^n$ *continuously differentiable,*

(C) $\mathrm{D}F(\mathbf{x})$ *regular* $\forall \mathbf{x} \in D$,

(D) $\exists L \geq 0$: $\left\| \mathrm{D}F(\mathbf{x})^{-1}(\mathrm{D}F(\mathbf{x}+\mathbf{v}) - \mathrm{D}F(\mathbf{x})) \right\|_2 \leq L\|\mathbf{v}\|_2 \quad \begin{array}{l} \forall \mathbf{v} \in \mathbb{R}^n, \mathbf{v}+\mathbf{x} \in D, \\ \forall \mathbf{x} \in D \end{array}$,

(E) $\exists \mathbf{x}^*$: $F(\mathbf{x}^*) = 0$ *(existence of solution in* D*)*

(F) *initial guess* $\mathbf{x}^{(0)} \in D$ *satisfies* $\rho := \left\| \mathbf{x}^* - \mathbf{x}^{(0)} \right\|_2 < \frac{2}{L} \quad \wedge \quad B_\rho(\mathbf{x}^*) \subset D$.

then *the Newton iteration* (8.4.1) *satisfies:*

(i) $\mathbf{x}^{(k)} \in B_\rho(\mathbf{x}^*) := \{\mathbf{y} \in \mathbb{R}^n, \|\mathbf{y} - \mathbf{x}^*\| < \rho\}$ *for all* $k \in \mathbb{N}$,

(ii) $\lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}^*$,

(iii) $\left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\|_2 \leq \frac{L}{2} \left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|_2^2$ *(local quadratic convergence) .*

✎ notation: ball $B_\rho(\mathbf{z}) := \{\mathbf{x} \in \mathbb{R}^n : \|\mathbf{x} - \mathbf{z}\|_2 \leq \rho\}$

Terminology: (D) $\hat{=}$ affine invariant Lipschitz condition

Usually, it is hardly possible to verify the assumptions of the theorem for a concrete non-linear system of equations, because neither L nor x^* are known.

In general: a priori estimates as in Thm. 8.4.45 are of little practical relevance.

8.4.3 Termination of Newton iteration

An abstract discussion of ways to stop iterations for solving $F(\mathbf{x}) = 0$ was presented in Section 8.1.2, with "ideal termination" ($\rightarrow$ § 8.1.24) as ultimate, but unfeasible, goal.

Yet, in 8.4.2 we saw that Newton's method enjoys (asymptotic) quadratic convergence, which means rapid decrease of the relative error of the iterates, once we are close to the solution, which is exactly the point, when we want to **STOP**. As a consequence, asymptotically, the Newton correction (difference of two consecutive iterates) yields rather precise information about the size of the error:

$$\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\| \ll \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \quad \Rightarrow \quad \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\| \approx \left\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right\| . \tag{8.4.46}$$

This suggests the following correction based termination criterion:

$$\begin{aligned} &\textbf{STOP}, \text{ as soon as } \quad \left\|\Delta\mathbf{x}^{(k)}\right\| \leq \tau_{\mathrm{rel}}\left\|\mathbf{x}^{(k)}\right\| \quad \textbf{or} \quad \left\|\Delta\mathbf{x}^{(k)}\right\| \leq \tau_{\mathrm{abs}} , \\ &\text{with} \quad \text{Newton correction} \quad \Delta\mathbf{x}^{(k)} := \mathrm{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)}). \end{aligned} \tag{8.4.47}$$

Here, $\|\cdot\|$ can be any suitable vector norm, $\tau_{\mathrm{rel}} \mathrel{\hat{=}}$ relative tolerance, $\tau_{\mathrm{abs}} \mathrel{\hat{=}}$ absolute tolerance, see § 8.1.24.

➢ quit iterating as soon as $\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\right\| = \left\|\mathrm{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)})\right\| < \tau\left\|\mathbf{x}^{(k)}\right\|$,
with τ = tolerance

$\rightarrow$ uneconomical: one needless update, because $\mathbf{x}^{(k)}$ would already be accurate enough.

Remark 8.4.48 (Newton's iteration; computational effort and termination)

Some facts about the Newton method for solving large ($n \gg 1$) non-linear systems of equations:

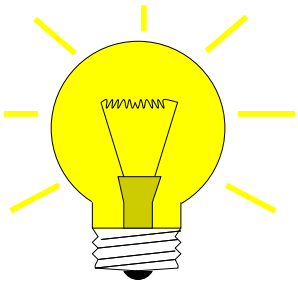
- ☛ Solving the linear system to compute the Newton correction may be expensive (asymptotic computational effort $O(n^3)$ for direct elimination $\rightarrow$ § 2.3.5) and accounts for the bulk of numerical cost of a single step of the iteration.
- ☛ In applications only very few steps of the iteration will be needed to achieve the desired accuracy due to fast quadratic convergence.

▷ The termination criterion (8.4.47) computes the last Newton correction $\Delta\mathbf{x}^{(k)}$ needlessly, because $\mathbf{x}^{(k)}$ already accurate enough**!**

Therefore we would like to use an a-posteriori termination criterion that dispenses with computing (and "inverting") another Jacobian $\mathrm{D}F(\mathbf{x}^{(k)})$ just to tell us that $\mathbf{x}^{(k)}$ is already accurate enough.

(8.4.49) Termination of Newton iteration based on simplified Newton correction

Due to fast asymptotic quadratic convergence, we can expect $\mathrm{D}F(\mathbf{x}^{(k-1)}) \approx \mathrm{D}F(\mathbf{x}^{(k)})$ during the final steps of the iteration.



Idea: Replace $\mathrm{D}F(\mathbf{x}^{(k)})$ with $\mathrm{D}F(\mathbf{x}^{(k-1)})$ in any correction based termination criterion.

Rationale: LU-decomposition of $\mathrm{D}F(\mathbf{x}^{(k-1)})$ is already available ➤ less effort.

Terminology: $\Delta\bar{\mathbf{x}}^{(k)} := \mathrm{D}F(\mathbf{x}^{(k-1)})^{-1}F(\mathbf{x}^{(k)}) \hat{=}$ simplified Newton correction

➤ Economical correction based termination criterion for Newton's method:

STOP, as soon as $\left\|\Delta\bar{\mathbf{x}}^{(k)}\right\| \leq \tau_{\mathrm{rel}}\left\|\mathbf{x}^{(k)}\right\|$ **or** $\left\|\Delta\bar{\mathbf{x}}^{(k)}\right\| \leq \tau_{\mathrm{abs}}$, (8.4.50)

with simplfied Newton correction $\Delta\bar{\mathbf{x}}^{(k)} := \mathrm{D}F(\mathbf{x}^{(k-1)})^{-1}F(\mathbf{x}^{(k)})$.

Note that (8.4.50) is affine invariant → Rem. 8.4.4.

Effort: Reuse of LU-factorization (→ Rem. 2.5.10) of $\mathrm{D}F(\mathbf{x}^{(k-1)})$ ➤ $\Delta\bar{\mathbf{x}}^{(k)}$ available with $O(n^2)$ operations

C++11 code 8.4.51: Generic Newton iteration with termination criterion (8.4.50)

```
template <typename FuncType,typename JacType,typename VecType>
void newton_stc(const FuncType &F,const JacType &DF,
                VecType &x,double rtol,double atol)
{
  using scalar_t = typename VecType::Scalar;
  scalar_t sn;
  do {
    auto jacfac = DF(x).lu(); // LU-factorize Jacobian ]
    x -= jacfac.solve(F(x));  // Compute next iterate
    // Compute norm of simplified Newton correction
    sn = jacfac.solve(F(x)).norm();
  }
  // Termination based on simplified Newton correction
  while ((sn > rtol*x.norm()) && (sn > atol));
}
```

Remark 8.4.52 (Residual based termination of Newton's method)

If we used the residual based termination criterion

$$\left\|F(\mathbf{x}^{(k)})\right\| \leq \tau ,$$

then the resulting algorithm would not be affine invariant, because for $F(\mathbf{x}) = 0$ and $\mathbf{A}F(\mathbf{x}) = 0$, $\mathbf{A} \in \mathbb{R}^{n,n}$ regular, the Newton iteration might terminate with different iterates.

Summary: Newton's method


converges *asymptotically* very fast: doubling of number of significant digits in each step


often a very small region of convergence, which requires an initial guess rather close to the solution.

8.4.4 Damped Newton method

Potentially big problem: Newton method converges quadratically, but only *locally*, which may render it useless, if convergence is guaranteed only for initial guesses very close to exact solution, see also Ex. 8.3.32.

In this section we study a method to enlarge the region of convergence, at the expense of quadratic convergence, of course.

Example 8.4.54 (Local convergence of Newton's method)

The dark side of local convergence (→ Def. 8.1.8): for many initial guesses $\mathbf{x}^{(0)}$ Newton's method will not converge!

In 1D two main causes can be identified:

❶ "Wrong direction" of Newton correction:

$$F(x) = xe^x - 1 \quad \Rightarrow \quad F'(-1) = 0$$

$$x^{(0)} < -1 \quad \Rightarrow \quad x^{(k)} \to -\infty\,,$$
$$x^{(0)} > -1 \quad \Rightarrow \quad x^{(k)} \to x^*\,,$$

because all Newton corrections for $x^{(k)} < -1$ make the iterates decrease even further.

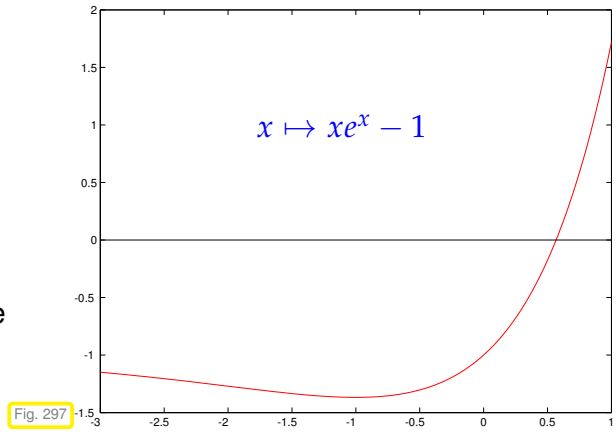


Fig. 297

❷ Newton correction is too large:

$$F(x) = \arctan(ax)\,, \quad a > 0, x \in \mathbb{R}$$

with zero $x^* = 0$.

If $x^{(k)}$ is located where the function is "flat", the intersection of the tangents with the x-axis is "far out", see Fig. 299.

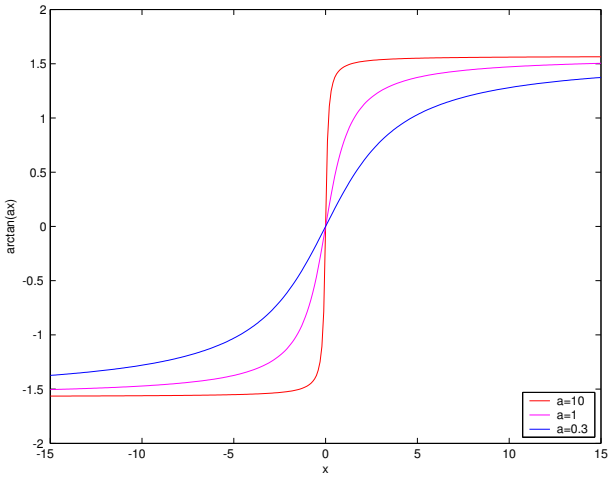
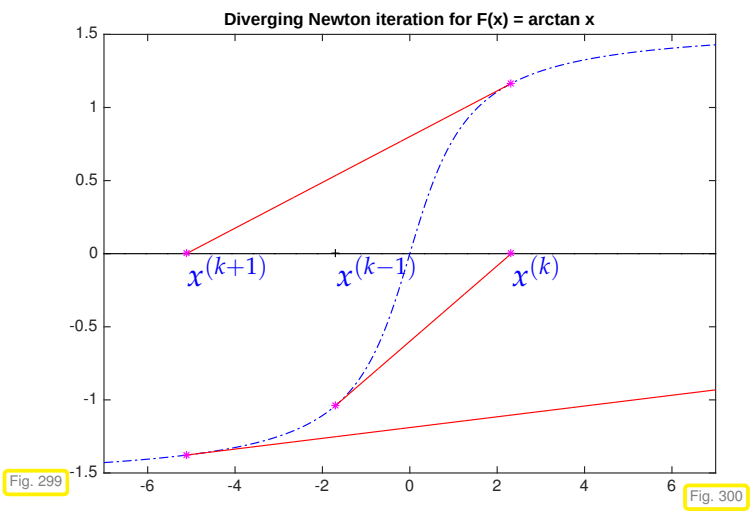
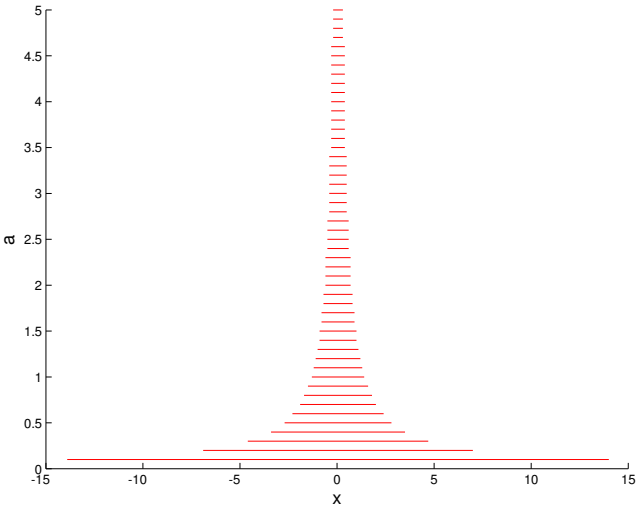


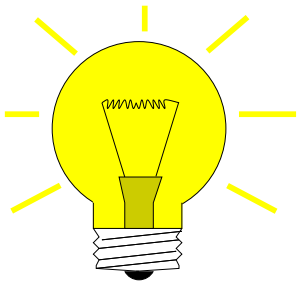
Fig. 298





In Fig. 300 the red zone = $\{x^{(0)} \in \mathbb{R}, x^{(k)} \to 0\}$, domain of initial guesses for which Newton's method converges.

If the Newton correction points in the wrong direction (Item ❶), no general remedy is available. If the Newton correction is too large (Item ❷), there is an effective cure:



we observe "overshooting" of Newton correction

Idea: damping of Newton correction:

With $\lambda^{(k)} > 0$: $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - \lambda^{(k)} \mathrm{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)})$. (8.4.55)

Terminology: $\lambda^{(k)}$ = damping factor

Affine invariant damping strategy

Choice of damping factor: affine invariant natural monotonicity test [?, Ch. 3]:

choose "maximal" $0 < \lambda^{(k)} \leq 1$: $\left\| \Delta\overline{\mathbf{x}}(\lambda^{(k)}) \right\| \leq (1 - \frac{\lambda^{(k)}}{2}) \left\| \Delta\mathbf{x}^{(k)} \right\|_2$ (8.4.57)

where

$\Delta\mathbf{x}^{(k)} := \mathrm{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)})$ → current Newton correction ,

$\Delta\overline{\mathbf{x}}(\lambda^{(k)}) := \mathrm{D}F(\mathbf{x}^{(k)})^{-1}F(\mathbf{x}^{(k)} + \lambda^{(k)}\Delta\mathbf{x}^{(k)})$ → tentative simplified Newton correction .

Heuristics behind control of damping:

- When the method converges ⇔ size of Newton correction decreases ⇔ (8.4.57) satisfied.
- In the case of strong damping ($\lambda^{(k)} \ll 1$) the size of the Newton correction cannot be expected to shrink significantly, since iterates do not change much ➢ factor $(1 - \frac{1}{2}\lambda^{(k)})$ in (8.4.57).

Note: As before, reuse of LU-factorization in the computation of $\Delta\mathbf{x}^{(k)}$ and $\Delta\overline{\mathbf{x}}(\lambda^{(k)})$.

MATLAB-code Damped Newton method

```
1  function [x,cvg] =
     dampnewton(x,F,DF,rtol,atol)
2  [L,U] = lu(DF(x)); s = U\(L\F(x));
3  xn = x-s; lambda = 1; cvg = 0;
4  f = F(xn); st = U\(L\f); stn = norm(st);
5  while ((stn>rtol*norm(xn)) && (stn > atol))
6    while (norm(st) > (1-lambda/2)*norm(s))
7      lambda = lambda/2;
8      if (lambda < LMIN), cvg = -1; return; end
9      xn = x-lambda*s; f = F(xn);
10     st =  U\(L\f);
11   end
12   x = xn; [L,U] = lu(DF(x)); s = U\(L\f);
13   lambda = min(2*lambda,1);
14   xn = x-lambda*s; f = F(xn); st = U\(L\f);
15 end
16 x = xn;
```

Reuse of LU-factorization, see Rem. 2.5.10

a-posteriori termination criterion (based on simplified Newton correction, *cf.* Section 8.4.3)

Natural monotonicity test (8.4.57)

Reduce damping factor λ

Note: LU-factorization of Jacobi matrix $\mathrm{D}F(\mathbf{x}^{(k)})$ is done once per *successful* iteration step (Line 12 of the above code) and reused for the computation of the simplified Newton correction in Line 10, Line 14 of the above MATLAB code.

Policy: Reduce damping factor by a factor $q \in]0,1[$ (usually $q = \frac{1}{2}$) until the affine invariant natural monotonicity test (8.4.57) passed, see Line 13 in the above MATLAB code.

C++11 code 8.4.58: Generic damped Newton method based on natural monotonicity test

```
1  template <typename FuncType,typename JacType,typename VecType>
2  void dampnewton(const FuncType &F,const JacType &DF,
3                  VecType &x,double rtol,double atol)
4  {
5    using index_t = typename VecType::Index;
6    using scalar_t = typename VecType::Scalar;
7    const index_t n = x.size();
8    const scalar_t lmin = 1E-3; // Minimal damping factor
9    scalar_t lambda = 1.0; // Initial and actual damping factor
10   VecType s(n),st(n); // Newton corrections
11   VecType xn(n);      // Tentative new iterate
12   scalar_t sn,stn;    // Norms of Newton corrections
13
14   do {
15     auto jacfac = DF(x).lu(); // LU-factorize Jacobian
16     s = jacfac.solve(F(x));   // Newton correction
17     sn = s.norm();            // Norm of Newton correction
18     lambda *= 2.0;
19     do {
20       lambda /= 2;
21       if (lambda < lmin) throw "No convergence: lambda -> 0";
22       xn = x-lambda*s;          // Tentative next iterate
23       st = jacfac.solve(F(xn)); // Simplified Newton correction
```

```
24          stn = st.norm();
25        }
26        while (stn > (1-lambda/2)*sn); // Natural monotonicity test
27        x = xn; // Now: xn accepted as new iterate
28        lambda = std::min(2.0*lambda,1.0); // Try to mitigate damping
29      }
30      // Termination based on simplified Newton correction
31      while ((stn > rtol*x.norm()) && (stn > atol));
32    }
```

The arguments for Code 8.4.58 are the same as for Code 8.4.51. As termination criterion is uses (8.4.50). Note that all calls to **solve** boil down to forward/backward elimination for triangular matrices and incur cost of $O(n^2)$ only.

Experiment 8.4.59 (Damped Newton method)

We test the damped Newton method for Item ❷ of Ex. 8.4.54, where excessive Newton corrections made Newton's method fail.

$F(x) = \arctan(x)$,
- $x^{(0)} = 20$
- $q = \frac{1}{2}$
- `LMIN = 0.001`

We observe that damping is effective and asymptotic quadratic convergence is recovered.

k	$\lambda^{(k)}$	$x^{(k)}$	$F(x^{(k)})$
1	0.03125	0.94199967624205	0.75554074974604
2	0.06250	0.85287592931991	0.70616132170387
3	0.12500	0.70039827977515	0.61099321623952
4	0.25000	0.47271811131169	0.44158487422833
5	0.50000	0.20258686348037	0.19988168667351
6	1.00000	-0.00549825489514	-0.00549819949059
7	1.00000	0.00000011081045	0.00000011081045
8	1.00000	-0.00000000000001	-0.00000000000001

Experiment 8.4.60 (Failure of damped Newton method)

We examine the effect of damping in the case of Item ❶ of Ex. 8.4.54.

- As in Ex. 8.4.54: $F(x) = xe^x - 1$,
- Initial guess for damped Newton method $x^{(0)} = -1.5$

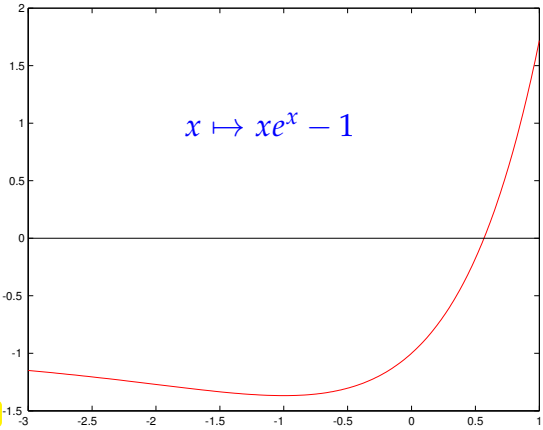


Fig. 301

Observation:

Newton correction pointing in "wrong direction"

➤ no convergence despite damping

k	$\lambda^{(k)}$	$x^{(k)}$	$F(x^{(k)})$
1	0.25000	-4.4908445351690	-1.0503476286303
2	0.06250	-6.1682249558799	-1.0129221310944
3	0.01562	-7.6300006580712	-1.0037055902301
4	0.00390	-8.8476436930246	-1.0012715832278
5	0.00195	-10.5815494437311	-1.0002685596314
Bailed out because of `lambda < LMIN` !			

8.4.5 Quasi-Newton Method

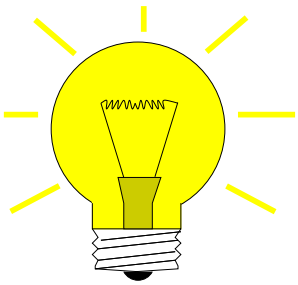
Supplementary reading. For related expositions refer to [**?**, Sect. 7.1.4], [**?**, 2.3.2].

How can we solve $F(\mathbf{x}) = 0$ iteratively, in case $\mathrm{D}F(\mathbf{x})$ is not available and numerical differentiation (see Rem. 8.4.41) is too expensive**?**

In 1D ($n = 1$ we can choose among many derivative-free methods that rely on F-evaluations alone, for instance the secant method (8.3.24) from Section 8.3.2.3:

$$x^{(k+1)} = x^{(k)} - \frac{F(x^{(k)})(x^{(k)} - x^{(k-1)})}{F(x^{(k)}) - F(x^{(k-1)})} \,. \tag{8.3.24}$$

Recall that the secant method converges locally with order $p \approx 1.6$ and beats Newton's method in terms of efficiency ($\rightarrow$ Section 8.3.3).

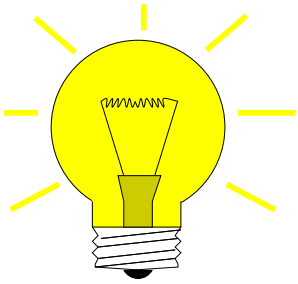


Comparing with (8.3.4) we realize that this iteration amounts to a "Newton-type iteration" with the approximation

$$F'(x^{(k)}) \approx \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}} \quad \text{"difference quotient"} \tag{8.4.61}$$

already computed **!** $\rightarrow$ cheap

Not clear how to generalize the secant method to $n > 1$ **?**



Idea: rewrite (8.4.61) as a secant condition for an approximation $\mathbf{J}_k \approx \mathrm{D}F(\mathbf{x}^{(k)})$, $\mathbf{x}^{(k)} \hat{=}$ iterate:

$$\boxed{\mathbf{J}_k(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) = F(\mathbf{x}^{(k)}) - F(\mathbf{x}^{(k-1)})} \,. \tag{8.4.62}$$

➤ Iteration: $\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} - \mathbf{J}_k^{-1} F(\mathbf{x}^{(k)})$. (8.4.63)

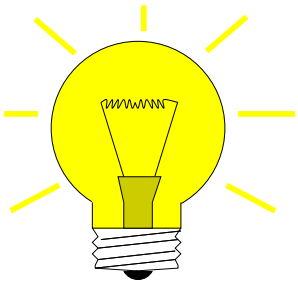
However, many matrices $\mathbf{J}_k$ fulfill (8.4.62)!

➢ We need extra conditions to fix $\mathbf{J}_k \in \mathbb{R}^{n,n}$.

Reasoning: If we assume that $\mathbf{J}_k$ is a good approximation of $\mathrm{D}F(\mathbf{x}^{(k)})$, then it would be foolish not to use the information contained in $\mathbf{J}_k$ for the construction of $\mathbf{J}_k$.

Guideline: obtain $\mathbf{J}_k$ through a "small" modification of $\mathbf{J}_{k-1}$ compliant with (8.4.62)

What can "small modification" mean: Demand that $\mathbf{J}_k$ acts like $\mathbf{J}_{k-1}$ on a complement of the span of $\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}$!



Broyden's conditions: $\mathbf{J}_k\mathbf{z} = \mathbf{J}_{k-1}\mathbf{z} \quad \forall \mathbf{z}: \mathbf{z} \perp (\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)})$. (8.4.64)

i.e.:
$$\boxed{\mathbf{J}_k := \mathbf{J}_{k-1} + \frac{F(\mathbf{x}^{(k)})(\mathbf{x}^{(k)}-\mathbf{x}^{(k-1)})^\top}{\|\mathbf{x}^{(k)}-\mathbf{x}^{(k-1)}\|_2^2}} \tag{8.4.65}$$

- The conditons (8.4.62) and (8.4.64) uniquely define $\mathbf{J}_k$
- The update formula (8.4.65) means that $\mathbf{J}_k$ is spawned by a rank-1-modification of $\mathbf{J}_{k-1}$.

Final form of Broyden's quasi-Newton method for solving $F(\mathbf{x}) = 0$:

$$\begin{aligned} \mathbf{x}^{(k+1)} &:= \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)} \quad , \quad \Delta\mathbf{x}^{(k)} := -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k)}) \, , \\ \mathbf{J}_{k+1} &:= \mathbf{J}_k + \frac{F(\mathbf{x}^{(k+1)})(\Delta\mathbf{x}^{(k)})^\top}{\|\Delta\mathbf{x}^{(k)}\|_2^2} \, . \end{aligned} \tag{8.4.66}$$

To start the iteration we have to initialize $\mathbf{J}_0$, e.g. with the exact Jacobi matrix $\mathrm{D}F(\mathbf{x}^{(0)})$.

Remark 8.4.67 (Minimality property of Broyden's rank-1-modification)

in another sense $\mathbf{J}_l$ is closest to $\mathbf{J}_{k-1}$ under the constraint of the secant condition (8.4.62):

Let $\mathbf{x}^{(k)}$ and $\mathbf{J}_k$ be the iterates and matrices, respectively, from Broyden's method (8.4.66), and let $\mathbf{J} \in \mathbb{R}^{n,n}$ satisfy the same secant condition (8.4.62) as $\mathbf{J}_{k+1}$:

$$\mathbf{J}(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = F(\mathbf{x}^{(k+1)}) - F(\mathbf{x}^{(k)}) \, . \tag{8.4.68}$$

Then from $\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)} - -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k)})$ we obtain

$$(\mathbf{I} - \mathbf{J}_k^{-1}\mathbf{J})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}) = -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k)}) - \mathbf{J}_k^{-1}(F(\mathbf{x}^{(k+1)}) - F(\mathbf{x}^{(k)})) = -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k+1)}) \, . \tag{8.4.69}$$

From this we get the identity

$$\begin{aligned} \mathbf{I} - \mathbf{J}_k^{-1}\mathbf{J}_{k+1} &= \mathbf{I} - \mathbf{J}_k^{-1}\left(\mathbf{J}_k + \frac{F(\mathbf{x}^{(k+1)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2^2}\right) \\ &= -\mathbf{J}_k^{-1}F(\mathbf{x}^{(k+1)})\frac{(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2^2} = \\ &\overset{(8.4.69)}{=} (\mathbf{I} - \mathbf{J}_k^{-1}\mathbf{J})\frac{(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})(\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}\|_2^2} \, . \end{aligned}$$

Using the submultiplicative property (1.5.77) of the Euclidean matrix norm, we conclude

$$\left\|\mathbf{I}-\mathbf{J}_k^{-1}\mathbf{J}_{k+1}\right\| \leq \left\|\mathbf{I}-\mathbf{J}_k^{-1}\mathbf{J}\right\| \text{ , because } \left\|\frac{(\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)})(\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k+1)}-\mathbf{x}^{(k)}\|_2^2}\right\|_2 \leq 1 \text{ ,}$$

which we saw in Ex. 1.5.86. This estimate holds for *all* matrices $\mathbf{J}$ satisfying (8.4.68).

We may read this as follows: (8.4.65) gives the $\|\cdot\|_2$-minimal relative correction of $\mathbf{J}_{k-1}$, such that the secant condition (8.4.62) holds.

Experiment 8.4.70 (Broydens quasi-Newton method: Convergence)

We revisit the 2×2 non-linear system of the Exp. 8.4.42 and take $\mathbf{x}^{(0)} = [0.7, 0.7]^T$. As starting value for the matrix iteration we use $\mathbf{J}_0 = \mathrm{D}F(\mathbf{x}^{(0)})$.

The numerical example shows that, in terms of convergence, the method is:

- slower than Newton method (8.4.1),
- faster than the simplified Newton method (see Rem. 8.4.39)

▷

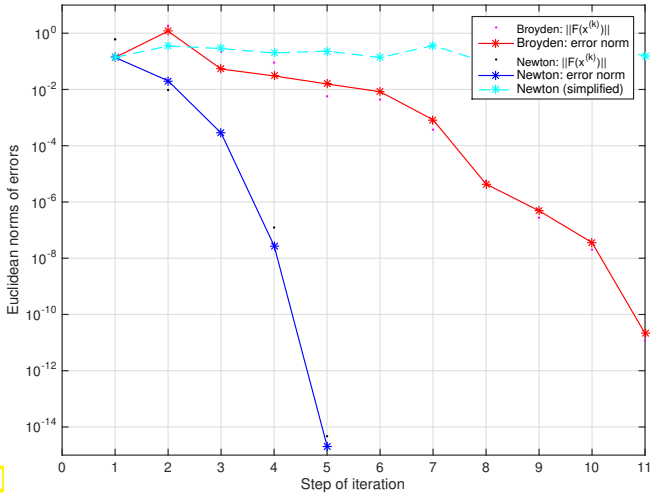


Fig. 302

Remark 8.4.71 (Convergence monitors)

In general, any iterative methods for non-linear systems of equations convergence can fail, that is it may stall or even diverge.

Demand on good numerical software: Algorithms should warn users of impending failure. For iterative methods this is the task of convergence monitors, that is, conditions, *cheaply verifiable* a posteriori during the iteration, that indicate stalled convergence or divergence.

For the damped Newton's method this role can be played by the natural monotonicity test, see Code 8.4.58; if it fails repeatedly, then the iteration should terminate with an error status.

For Broyden's quasi-Newton method, a similar strategy can rely on the relative size of the "simplified Broyden correction" $\mathbf{J}_k F(\mathbf{x}^{(k+1)})$:

$$\text{Convergence monitor for (8.4.66) : } \quad \mu := \frac{\left\|\mathbf{J}_{k-1}^{-1}F(\mathbf{x}^{(k)})\right\|}{\left\|\Delta\mathbf{x}^{(k-1)}\right\|} < 1 \text{ ?} \tag{8.4.72}$$

Experiment 8.4.73 (Monitoring convergence for Broyden's quasi-Newton method)

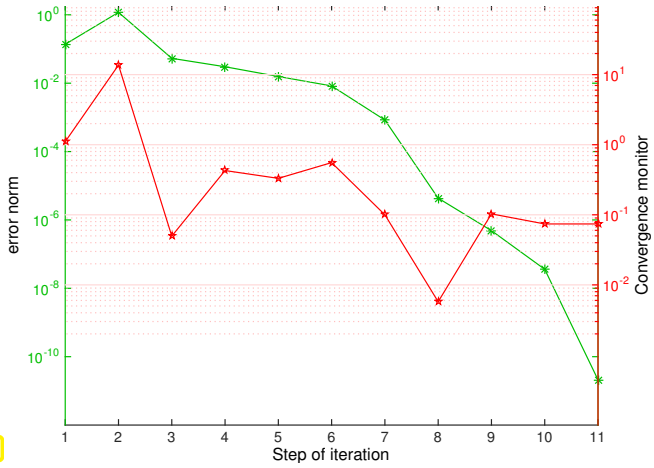


Fig. 303

We rely on the setting of Exp. 8.4.70.

We track
1. the Euclidean norm of the iteration error,
2. and the value of the convergence monitor from (8.4.72).

◁ Decay of (norm of) iteration error and μ are well correlated.

Remark 8.4.74 (Damped Broyden method)

Option to improve robustness (increase region of local convergence):

damped Broyden method (*cf.* same idea for Newton's method, Section 8.4.4)

(8.4.75) Implementation of Broyden's quasi-Nerwton method

As remarked, (8.4.66) represents a rank-1-update as already discussed in § 2.6.13.

Idea: use Sherman-Morrison-Woodbury update-formula from Lemma 2.6.22, which yields

$$\mathbf{J}_{k+1}^{-1} = \left(\mathbf{I} - \frac{\mathbf{J}_k^{-1}F(\mathbf{x}^{(k+1)})(\Delta\mathbf{x}^{(k)})^T}{\|\Delta\mathbf{x}^{(k)}\|_2^2 + \Delta\mathbf{x}^{(k)}\cdot\mathbf{J}_k^{-1}F(\mathbf{x}^{(k+1)})}\right)\mathbf{J}_k^{-1} = \left(\mathbf{I} + \frac{\Delta\mathbf{x}^{(k+1)}(\Delta\mathbf{x}^{(k)})^T}{\|\Delta\mathbf{x}^{(k)}\|_2^2}\right)\mathbf{J}_k^{-1}\,. \tag{8.4.76}$$

This gives a well defined $\mathbf{J}_{k+1}$, if

$$\left\|\mathbf{J}_k^{-1}F(\mathbf{x}^{(k+1)})\right\|_2 < \left\|\Delta\mathbf{x}^{(k)}\right\|_2\,. \tag{8.4.77}$$

"simplified Quasi-Newton correction"

Note that the condition (8.4.77) is checked by the convergence monitor (8.4.72).

Iterated application of (8.4.76) pays off, if iteration terminates after only a few steps. For large $n \gg 1$ it is not advisable to form the matrices $\mathbf{J}_k^{-1}$ (which will usually be dense in contrast to $\mathbf{J}_k$), but we employ fast successive multiplications with rank-1-matrices ($\rightarrow$ Ex. 1.4.11) to apply $\mathbf{J}_k^{-1}$ to a vector. This is implemented in the following code.

MATLAB-code : Broyden method (8.4.66)

```
function x = broyden(F,x,J,tol)
k = 1;
[L,U] = lu(J);
s = U\(L\F(x)); sn = dot(s,s);
dx = [s]; dxn = [sn];
x = x - s; f = F(x);

while (sqrt(sn) > tol), k=k+1
  w = U\(L\f);
  for l=2:k-1
    w = w+dx(:,l)*(dx(:,l-1)'*w) ...
        /dxn(l-1);
  end
  if (norm(w)>=sn)
    warning('Dubious step %d!',k);
  end
  z = s'*w; s = (1+z/(sn-z))*w; sn=s'*s;
  dx = [dx,s]; dxn = [dxn,sn];
  x = x - s; f = F(x);
end
```

unique LU-decomposition !

store $\Delta\mathbf{x}^{(k)}$, $\left\|\Delta\mathbf{x}^{(k)}\right\|_2^2$ (see (8.4.76))

solve two SLEs

Termination, Section 8.4.2

construct $\mathbf{w} := \mathbf{J}_k^{-1}F(\mathbf{x}^{(k)})$ ($\rightarrow$ recursion (8.4.76))

convergence monitor $\frac{\|\mathbf{J}_{k-1}^{-1}F(\mathbf{x}^{(k)})\|}{\|\Delta\mathbf{x}^{(k-1)}\|} < 1$?

correction $\mathbf{s} = \mathbf{J}_k^{-1}F(\mathbf{x}^{(k)})$

Computational cost : N steps

- $O(N^2 \cdot n)$ operations with vectors, (Level I)
- 1 LU-decomposition of $\mathtt{J}$, $N\times$ solutions of SLEs, see Section 2.3.2
- N evaluations of F !

Memory cost : N steps

- LU-factors of $\mathtt{J}$ + auxiliary vectors $\in \mathbb{R}^n$
- N vectors $\mathbf{x}^{(k)} \in \mathbb{R}^n$

Experiment 8.4.78 (Broyden method for a large non-linear system)

$$F(\mathbf{x}) = \begin{cases} \mathbb{R}^n & \mapsto \mathbb{R}^n \\ \mathbf{x} & \mapsto \operatorname{diag}(\mathbf{x})\mathbf{A}\mathbf{x} - \mathbf{b} \end{cases},$$
$$\mathbf{b} = [1,2,\ldots,n] \in \mathbb{R}^n,$$
$$\mathbf{A} = \mathbf{I} + \mathbf{a}\mathbf{a}^T \in \mathbb{R}^{n,n},$$
$$\mathbf{a} = \frac{1}{\sqrt{\mathbf{1}\cdot\mathbf{b} - \mathbf{1}}}(\mathbf{b} - \mathbf{1}).$$

Initial guess: `h = 2/n; x0 = (2:h:4-h)';`

The results resemble those of Exp. 8.4.70 ▷

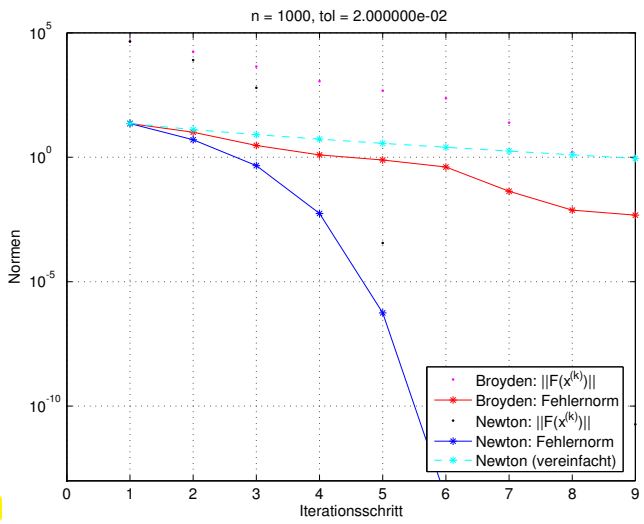


Fig. 304

Efficiency comparison: Broyden method $\longleftrightarrow$ Newton method:
(in case of dimension n use tolerance $\mathtt{tol} = 2n \cdot 10^{-5}$, `h = 2/n; x0 = (2:h:4-h)';`)

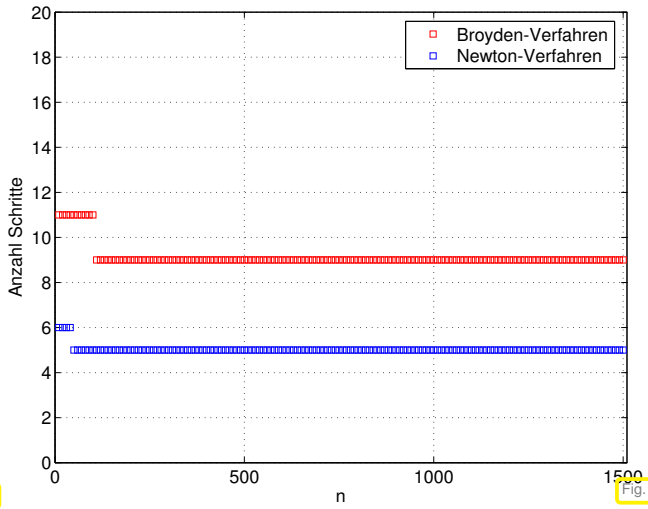


Fig. 305

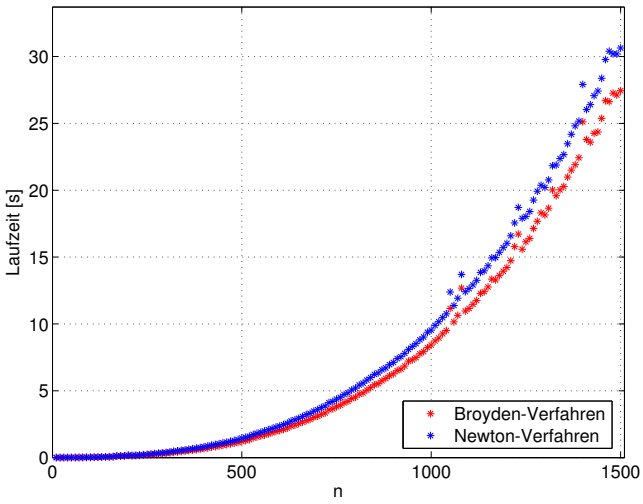


Fig. 306

☞ In conclusion,
the Broyden method is worthwhile for dimensions $n \gg 1$ and low accuracy requirements.

8.5 Unconstrained Optimization

8.5.1 Minima and minimizers: Some theory

8.5.2 Newton's method

8.5.3 Descent methods

8.5.4 Quasi-Newton methods

Learning Outcomes

- Knowledge about concepts related to the speed of convergence of an iteration for solving a non-linear system of equations.
- Ability to estimate type and orders of convergence from empiric data.
- Ability to predict asymptotic linear, quadratic and cubic convergence by inspection of the iteration function.
- Familiarity with (damped) Newton's method for general non-linear systems of equations and with the secant method in 1D.
- Ability to derive the Newton iteration for an (implicitly) given non-linear system of equations.
- Knowledge about quasi-Newton method as multi-dimensional generalizations of the secant method.

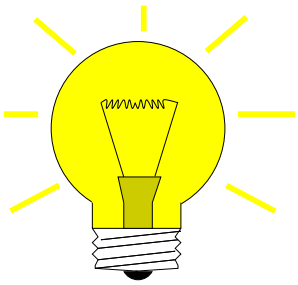
8.6 Non-linear Least Squares [?, Ch. 6]

Example 8.6.1 (Least squares data fitting)

In Section 5.1 we discussed the reconstruction of a *parameterized function* $f(x_1,\ldots,x_n;\cdot) : D \subset \mathbb{R} \mapsto \mathbb{R}$ from data points (t_i, y_i), $i = 1,\ldots,n$, by imposing interpolation conditions. Necessary was that the number n of parameters agreed with number of data points. The interpolation approach is justified in the case of *highly accurate data*.

Frequently encountered: *inaccurate data* (due to *measurement errors*)

➢ interpolation approach dubious (impact of "outliers"!)



Mitigate impact of data uncertainty by
choosing fewer parameters than data points

(measurement errors can "average out")

Non-linear least squares fitting problem: [**?**, Sect. 6.1]

Given:
- ✦ data points (t_i, y_i), $i = 1,\ldots,m$
- ✦ (symbolic formula) for parameterized function $f(x_1,\ldots,x_n;\cdot) : D \subset \mathbb{R} \mapsto \mathbb{R}$, $n < m$

Sought: parameter values $x_1^*,\ldots,x_n^* \in \mathbb{R}$ such that

$$(x_1^*,\ldots,x_n^*) = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} \sum_{i=1}^{m} |f(x_1,\ldots,x_n;t_i) - y_i|^2 \,. \tag{8.6.2}$$

Example 8.6.3 (Non-linear data fitting (parametric statistics) → Ex. 8.6.1 revisited)

Given: data points (t_i, y_i), $i = 1,\ldots,m$ with measurement errors.

Known: $y = f(\mathbf{x}, t)$ through a function $f : \mathbb{R}^n \times \mathbb{R} \mapsto \mathbb{R}$ depending non-linearly and smoothly on parameters $\mathbf{x} \in \mathbb{R}^n$.

Example: $f(t) = x_1 + x_2 \exp(-x_3 t)$, $n = 3$.

Determine parameters by non-linear least squares data fitting:

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} \sum_{i=1}^{m} |f(\mathbf{x},t_i) - y_i|^2 = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} \tfrac{1}{2}\|F(\mathbf{x})\|_2^2 \,, \tag{8.6.4}$$

$$\text{with} \quad F(\mathbf{x}) = \begin{bmatrix} f(\mathbf{x},t_1) - y_1 \\ \vdots \\ f(\mathbf{x},t_m) - y_m \end{bmatrix}.$$

Non-linear least squares problem

Given: $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}^m, \quad m,n \in \mathbb{N},\ m > n.$

Find: $\mathbf{x}^* \in D$: $\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x}\in D} \Phi(\mathbf{x})$, $\Phi(\mathbf{x}) := \frac{1}{2}\|F(\mathbf{x})\|_2^2$. (8.6.5)

Terminology: $D \hat{=}$ parameter space, $x_1, \ldots, x_n \hat{=}$ parameter.

As in the case of linear least squares problems (→ Section 3.1.1): a non-linear least squares problem is related to an overdetermined non-linear system of equations $F(\mathbf{x}) = 0$.

As for non-linear systems of equations (→ Chapter 8): existence and uniqueness of $\mathbf{x}^*$ in (8.6.5) has to be established in each concrete case!

We require "independence for each parameter": → Rem. 3.1.27

$\exists$ neighbourhood $\mathcal{U}(\mathbf{x}^*)$ such that $DF(\mathbf{x})$ has full rank $n \quad \forall\, \mathbf{x} \in \mathcal{U}(\mathbf{x}^*)$. (8.6.6)

(It means: the columns of the Jacobi matrix $DF(\mathbf{x})$ are linearly independent.)

If (8.6.6) is not satisfied, then the parameters are redundant in the sense that fewer parameters would be enough to model the same dependence (locally at $\mathbf{x}^*$), *cf.* Rem. 3.1.27.

8.6.1 (Damped) Newton method

$$\Phi(\mathbf{x}^*) = \min \quad \Rightarrow \quad \mathbf{grad}\,\Phi(\mathbf{x}) = 0, \quad \mathbf{grad}\,\Phi(\mathbf{x}) := (\tfrac{\partial\Phi}{\partial x_1}(\mathbf{x}), \ldots, \tfrac{\partial\Phi}{\partial x_n}(\mathbf{x}))^T \in \mathbb{R}^n.$$

Simple idea: use Newton's method (→ Section 8.4) to determine a zero of $\mathbf{grad}\,\Phi : D \subset \mathbb{R}^n \mapsto \mathbb{R}^n$.

Newton iteration (8.4.1) for non-linear system of equations $\mathbf{grad}\,\Phi(\mathbf{x}) = 0$

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - H\Phi(\mathbf{x}^{(k)})^{-1}\,\mathbf{grad}\,\Phi(\mathbf{x}^{(k)}) \ , \quad (H\Phi(\mathbf{x}) = \text{Hessian matrix}) \ . \tag{8.6.7}$$

Expressed in terms of $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ from (8.6.5):

chain rule (8.4.9) ➤ $\mathbf{grad}\,\Phi(\mathbf{x}) = DF(\mathbf{x})^T F(\mathbf{x})$,

product rule (8.4.10) ➤ $H\Phi(\mathbf{x}) := D(\mathbf{grad}\,\Phi)(\mathbf{x}) = DF(\mathbf{x})^T DF(\mathbf{x}) + \sum_{j=1}^{m} F_j(\mathbf{x}) D^2 F_j(\mathbf{x})$,

$$\Updownarrow$$

$$(H\Phi(\mathbf{x}))_{i,k} = \sum_{j=1}^{n} \frac{\partial^2 F_j}{\partial x_i \partial x_k}(\mathbf{x}) F_j(\mathbf{x}) + \frac{\partial F_j}{\partial x_k}(\mathbf{x}) \frac{\partial F_j}{\partial x_i}(\mathbf{x}) \ .$$

Recommendation, *cf.* § 8.4.8: when in doubt, differentiate components of matrices and vectors!

The above derivative formulas allow to rewrite (8.6.7) in concrete terms:

For Newton iterate $\mathbf{x}^{(k)}$: Newton correction $\mathbf{s} \in \mathbb{R}^n$ from LSE

$$\underbrace{\left(DF(\mathbf{x}^{(k)})^T DF(\mathbf{x}^{(k)}) + \sum_{j=1}^{m} F_j(\mathbf{x}^{(k)}) D^2 F_j(\mathbf{x}^{(k)}) \right)}_{= H\Phi(\mathbf{x}^{(k)})} \mathbf{s} = - \underbrace{DF(\mathbf{x}^{(k)})^T F(\mathbf{x}^{(k)})}_{= \mathbf{grad}\, \Phi(\mathbf{x}^{(k)})} . \tag{8.6.8}$$

Remark 8.6.9 (Newton method and minimization of quadratic functional)

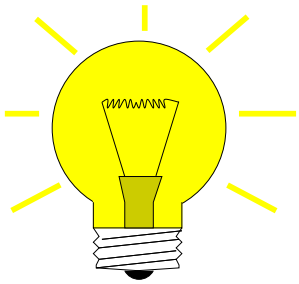
Newton's method (8.6.7) for (8.6.5) can be read as *successive minimization* of a local quadratic approximation of Φ:

$$\Phi(\mathbf{x}) \approx Q(\mathbf{s}) := \Phi(\mathbf{x}^{(k)}) + \mathbf{grad}\, \Phi(\mathbf{x}^{(k)})^T \mathbf{s} + \frac{1}{2} \mathbf{s}^T H\Phi(\mathbf{x}^{(k)}) \mathbf{s} , \tag{8.6.10}$$

$$\mathbf{grad}\, Q(\mathbf{s}) = 0 \quad \Leftrightarrow \quad H\Phi(\mathbf{x}^{(k)})\mathbf{s} + \mathbf{grad}\, \Phi(\mathbf{x}^{(k)}) = 0 \quad \Leftrightarrow \quad (8.6.8) .$$

➢ So we deal with yet another model function method (→ Section 8.3.2) with quadratic model function for Q.

8.6.2 Gauss-Newton method



Idea: local linearization of F: $F(\mathbf{x}) \approx F(\mathbf{y}) + DF(\mathbf{y})(\mathbf{x} - \mathbf{y})$

➢ sequence of *linear* least squares problems

$$\operatorname*{argmin}_{\mathbf{x} \in \mathbb{R}^n} \|F(\mathbf{x})\|_2 \quad \text{is approximated by} \quad \underbrace{\operatorname*{argmin}_{\mathbf{x} \in \mathbb{R}^n} \|F(\mathbf{x}_0) + DF(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)\|_2}_{(\spadesuit)} ,$$

where $\mathbf{x}_0$ is an approximation of the solution $\mathbf{x}^*$ of (8.6.5).

$$(\spadesuit) \quad \Leftrightarrow \quad \operatorname*{argmin}_{\mathbf{x} \in \mathbb{R}^n} \|\mathbf{A}\mathbf{x} - \mathbf{b}\| \quad \text{with} \quad \mathbf{A} := DF(\mathbf{x}_0) \in \mathbb{R}^{m,n} , \quad \mathbf{b} := -F(\mathbf{x}_0) + DF(\mathbf{x}_0)\mathbf{x}_0 \in \mathbb{R}^m .$$

This is a linear least squares problem of the form (3.1.38).

Note: (8.6.6) ⇒ $\mathbf{A}$ has full rank, if $\mathbf{x}_0$ sufficiently close to $\mathbf{x}^*$.

Note: This approach is different from local quadratic approximation of Φ underlying Newton's method for (8.6.5), see Section 8.6.1, Rem. 8.6.9.

Gauss-Newton iteration (under assumption (8.6.6))

Initial guess $\mathbf{x}^{(0)} \in D$

$$\mathbf{x}^{(k+1)} := \operatorname*{argmin}_{\mathbf{x} \in \mathbb{R}^n} \left\| F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})(\mathbf{x} - \mathbf{x}^{(k)}) \right\|_2 . \tag{8.6.11}$$

linear least squares problem

MATLAB-\ used to solve linear least squares problem in each step:

for $\mathbf{A} \in \mathbb{R}^{m,n}$

```
x = A\b
```
$\updownarrow$
$\mathbf{x}$ minimizer of $\|\mathbf{Ax}-\mathbf{b}\|_2$
with minimal 2-norm

C++-code 8.6.12: template for Gauss-Newton method

```
1  #include <Eigen/Dense>
2  #include <Eigen/QR>
3  using Eigen::VectorXd;
4  using Eigen::MatrixXd;
5
6  template <class Function, class Jacobian>
7  VectorXd gn(const VectorXd& init, const Function& F, const
       Jacobian& J, const double tol) {
8    VectorXd x = init;
9    VectorXd s = J(x).householderQr().solve(F(x)); //
10   x = x - s;
11   while (s.norm() > tol * x.norm()) { //
12     s = J(x).householderQr().solve(F(x)); //
13     x = x - s;
14   }
15
16   return x;
17 }
```

Comments on Code 8.6.12:

- ☞ Argument `x` passes initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$, argument `F` must be a *handle* to a function $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, argument `J` provides the Jacobian of F, namely $DF : \mathbb{R}^n \mapsto \mathbb{R}^{m,n}$, argument `tol` specifies the tolerance for termination
- ☞ Line 11: iteration terminates if relative norm of correction is below threshold specified in `tol`.

Note: Code 8.6.12 also implements Newton's method (→ Section 8.4.1) in the case $m = n$!

Summary:

Advantage of the Gauss-Newton method : second derivative of F not needed.
Drawback of the Gauss-Newton method : no local quadratic convergence.

Example 8.6.13 (Non-linear data fitting (II) → **Ex. 8.6.3)**

Non-linear data fitting problem (8.6.4) for $f(t) = x_1 + x_2 \exp(-x_3 t)$.

$$F(\mathbf{x}) = \begin{bmatrix} x_1 + x_2\exp(-x_3 t_1) - y_1 \\ \vdots \\ x_1 + x_2\exp(-x_3 t_m) - y_m \end{bmatrix} : \mathbb{R}^3 \mapsto \mathbb{R}^m \,,\; DF(\mathbf{x}) = \begin{bmatrix} 1 & e^{-x_3 t_1} & -x_2 t_1 e^{-x_3 t_1} \\ \vdots & \vdots & \vdots \\ 1 & e^{-x_3 t_m} & -x_2 t_m e^{-x_3 t_m} \end{bmatrix}$$

Numerical experiment:

convergence of the Newton method, damped Newton method (→ Section 8.4.4) and Gauss-Newton method for different initial values

C++-code 8.6.14:

```
#include <Eigen/Dense>
using Eigen::VectorXd;

VectorXd gnrandinit(const VectorXd& x) {
  std::srand((unsigned int) time(0));

  auto t = VectorXd::LinSpaced((7.0 - 1.0) /
      0.3 - 1, 1.0, 7.0);
  auto y = x(0) +
      x(1)*((-x(2)*t).array().exp());
  return y + 0.1 *
      (VectorXd::Random(y.size()).array()-0.5);
}
```

✦ initial value $(1.8, 1.8, 0.1)^T$ (red curves, blue curves)
✦ initial value $(1.5, 1.5, 0.1)^T$ (cyan curves, green curves)

First experiment (→ Section 8.6.1): iterative solution of non-linear least squares data fitting problem by means of the Newton method (8.6.8) and the damped Newton method from Section 8.4.4

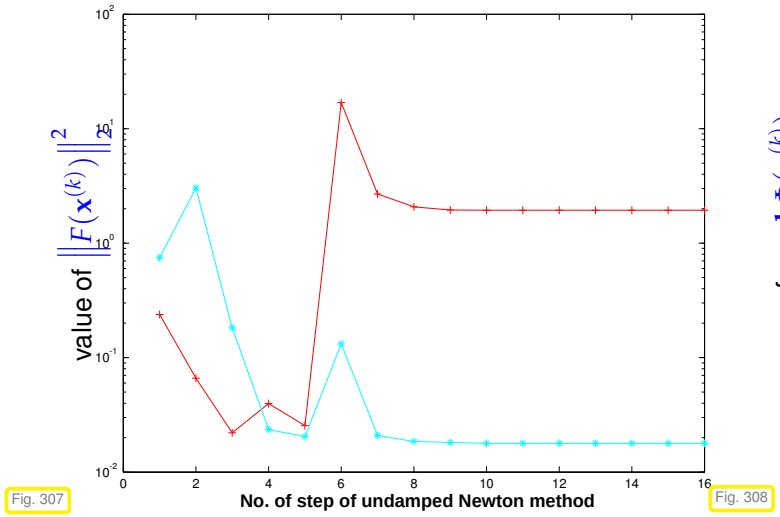


Fig. 307

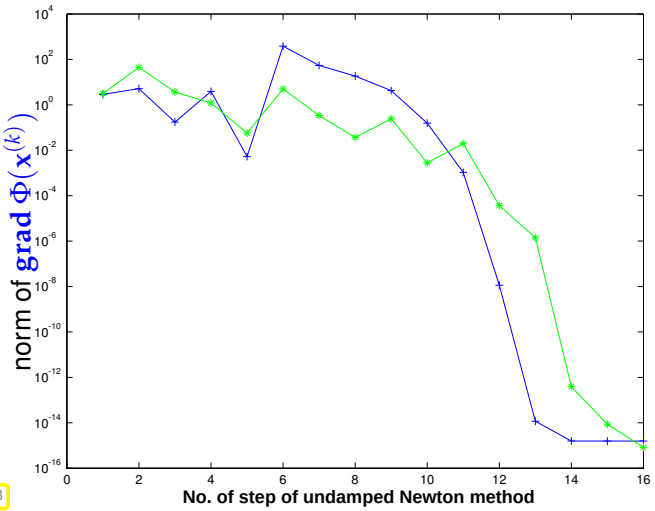


Fig. 308

Convergence behaviour of plain Newton method:

initial value $(1.8, 1.8, 0.1)^T$ (red curve) ➤ Newton method caught in local minimum,
initial value $(1.5, 1.5, 0.1)^T$ (cyan curve) ➤ fast (locally quadratic) convergence.

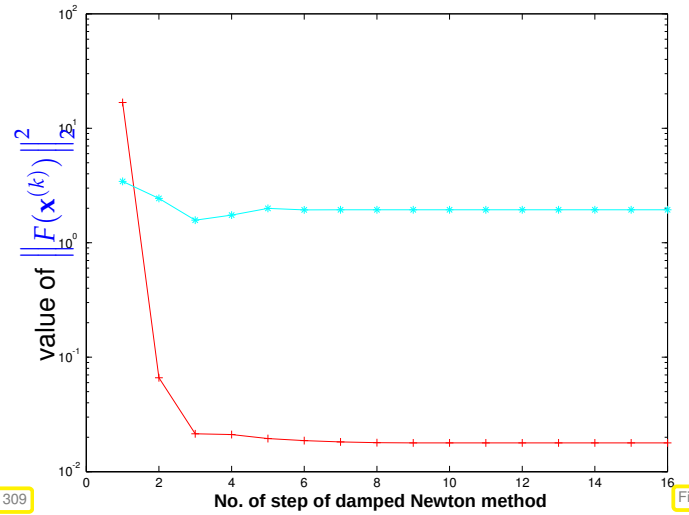


Fig. 309

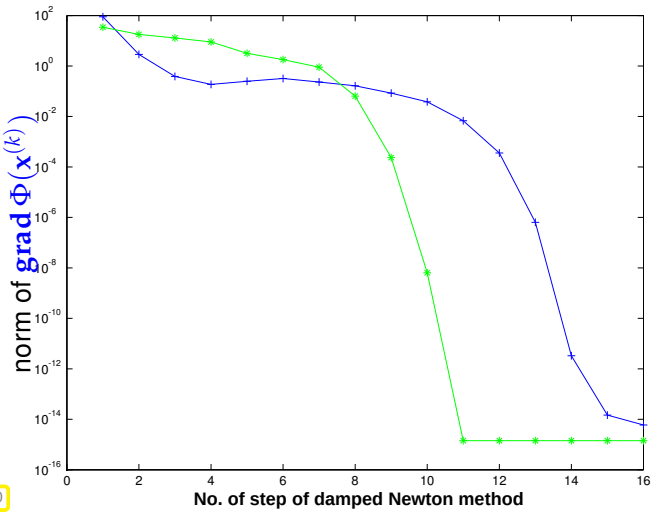


Fig. 310

Convergence behavior of damped Newton method:

initial value $(1.8, 1.8, 0.1)^T$ (red curve) ➤ fast (locally quadratic) convergence,
initial value $(1.5, 1.5, 0.1)^T$ (cyan curve) ➤ Newton method caught in local minimum.

Second experiment: iterative solution of non-linear least squares data fitting problem by means of the Gauss-Newton method (8.6.11), see Code 8.6.12.

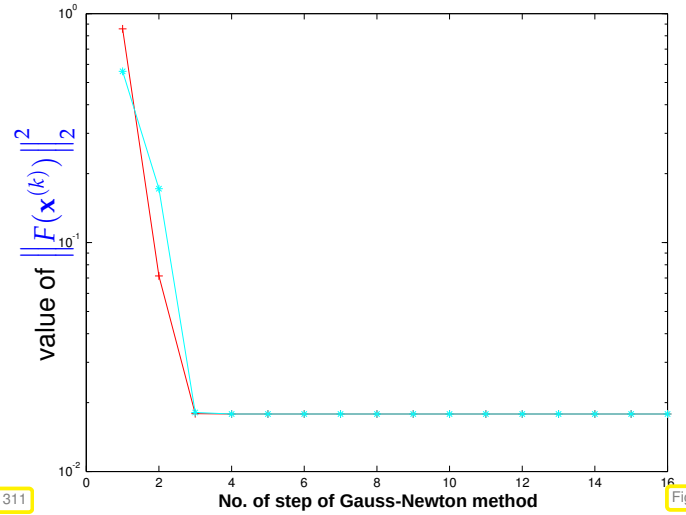


Fig. 311

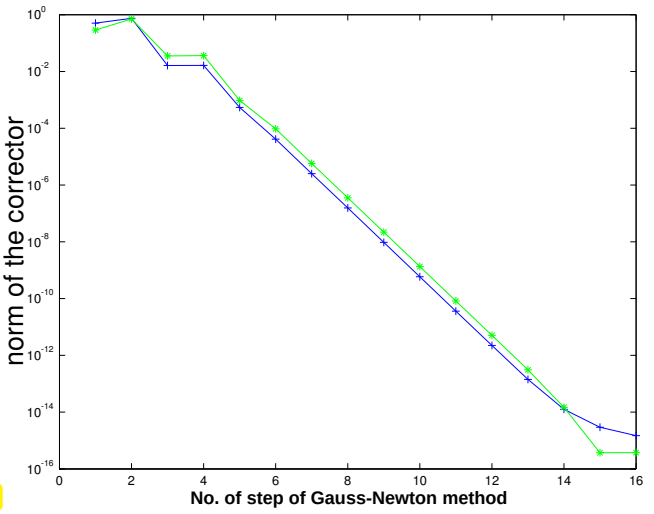


Fig. 312

We observe: linear convergence for all initial values, *cf.* Def. 8.1.9, Rem. 8.1.13.

8.6.3 Trust region method (Levenberg-Marquardt method)

As in the case of Newton's method for non-linear systems of equations, see Section 8.4.4: often overshooting of Gauss-Newton corrections occurs.

Remedy as in the case of Newton's method: damping.

Idea: damping of the Gauss-Newton correction in (8.6.11) using a penalty term

$$\text{instead of} \quad \left\|F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})\mathbf{s}\right\|^2 \quad \text{minimize} \quad \left\|F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})\mathbf{s}\right\|^2 + \lambda\|\mathbf{s}\|_2^2 \,.$$

$\lambda > 0 \triangleq$ penalty parameter (how to choose it **?** $\rightarrow$ heuristic)

$$\lambda = \gamma \left\| F(\mathbf{x}^{(k)}) \right\|_2 \quad , \quad \gamma := \begin{cases} 10 & \text{, if } \left\| F(\mathbf{x}^{(k)}) \right\|_2 \geq 10 \text{ ,} \\ 1 & \text{, if } 1 < \left\| F(\mathbf{x}^{(k)}) \right\|_2 < 10 \text{ ,} \\ 0.01 & \text{, if } \left\| F(\mathbf{x}^{(k)}) \right\|_2 \leq 1 \text{ .} \end{cases}$$

Modified (regularized) equation for the corrector $\mathbf{s}$:

$$\left(DF(\mathbf{x}^{(k)})^T DF(\mathbf{x}^{(k)}) + \lambda \mathbf{I} \right) \mathbf{s} = -DF(\mathbf{x}^{(k)}) F(\mathbf{x}^{(k)}) \text{ .} \tag{8.6.15}$$

Chapter 9

Eigenvalues

Supplementary reading. [**?**] offers comprehensive presentation of numerical methods for the solution of eigenvalue problems from an algorithmic point of view.

Example 9.0.1 (Resonances of linear electric circuits)

Simple electric circuit, *cf.* Ex. 2.1.3 ▷

- ✦ linear components (resistors, coils, capacitors) only,
- ✦ time-harmonic excitation (alternating voltage/current)
- ✦ "frequency domain" circuit model

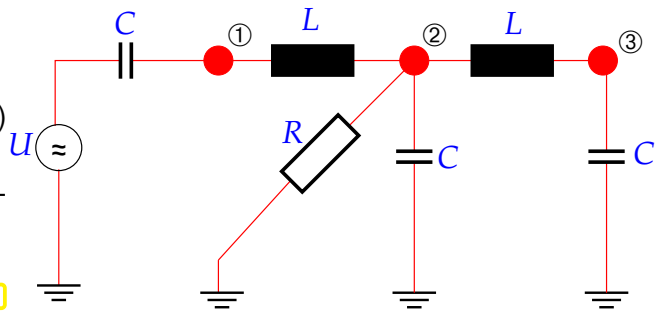


Fig. 313

Ex. 2.1.3: nodal analysis of linear (↔ composed of resistors, inductors, capacitors) electric circuit in frequency domain (at angular frequency $\omega > 0$) , see (2.1.6)

➢ linear system of equations for nodal potentials with complex system matrix $\mathbf{A}$

For circuit of Code 9.0.3: three unknown nodal potentials

➢ system matrix from nodal analysis at angular frequency $\omega > 0$:

$$\mathbf{A} = \begin{bmatrix} \imath\omega C + \frac{1}{\imath\omega L} & -\frac{1}{\imath\omega L} & 0 \\ -\frac{1}{\imath\omega L} & \imath\omega C + \frac{1}{R} + \frac{2}{\imath\omega L} & -\frac{1}{\imath\omega L} \\ 0 & -\frac{1}{\imath\omega L} & \imath\omega C + \frac{1}{\imath\omega L} \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{1}{R} & 0 \\ 0 & 0 & 0 \end{bmatrix} + \imath\omega \begin{bmatrix} C & 0 & 0 \\ 0 & C & 0 \\ 0 & 0 & C \end{bmatrix} + {}^1\!/\!_{\imath\omega} \begin{bmatrix} \frac{1}{L} & -\frac{1}{L} & 0 \\ -\frac{1}{L} & \frac{2}{L} & -\frac{1}{L} \\ 0 & -\frac{1}{L} & \frac{1}{L} \end{bmatrix} .$$

$$\mathbf{A}(\omega) := \mathbf{W} + i\omega\mathbf{C} - i\omega^{-1}\mathbf{S} \quad , \quad \mathbf{W}, \mathbf{C}, \mathbf{S} \in \mathbb{R}^{n,n} \text{ symmetric} . \tag{9.0.2}$$

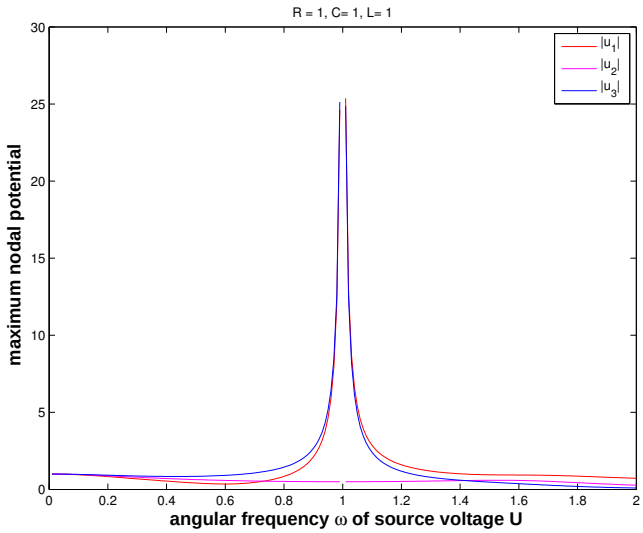


Fig. 314

◁ plot of $|u_i(U)|$, $i = 1,2,3$ for $R = L = C = 1$ (scaled model)

Blow-up of some nodal potentials for certain ω !

MATLAB-code 9.0.3: Computation of nodal potential for circuit of Code 9.0.3

```
function rescirc(R,L,C)
% Ex. ??:  Numerical nodal analysis of the resonant circuit
% R, L, C ≙ network component parameters

Z = 1/R; K = 1/L;

% Matrices W, C, S for nodal analysis of circuit
Wmat = [0 0 0; 0 Z 0; 0 0 0];
Cmat = [C 0 0; 0 C 0; 0 0 C];
Smat = [K -K 0; -K 2*K -K; 0 -K K];
% System matrix from nodal analysis
Amat = @(w) (Wmat+i*w*Cmat+Smat/(i*w));

% Scanning source currents
res = [];
for w=0.01:0.01:2
  res = [res; w, abs(Amat(w)\[C;0;0])'];
end

figure('name','resonant circuit');
plot(res(:,1),res(:,2),'r-',res(:,1),res(:,3),'m-',res(:,1),res(:,4),'b-'
xlabel('{\bf angular frequency \omega of source voltage
   U}','fontsize',14);
ylabel('{\bf maximum nodal potential}','fontsize',14);
title(sprintf('R = %d, C= %d, L= %d',R,L,C));
legend('|u_1|','|u_2|','|u_3|');

print -depsc2 '../PICTURES/rescircpot.eps'

% Solving generalized eigenvalue problem (9.0.5)
Zmat = zeros(3,3); Imat = eye(3,3);
% Assemble 6×6-matrices M and B
Mmat = [Wmat,Smat; Imat, Zmat];
Bmat = [-i*Cmat, Zmat; Zmat , i*Imat];
```

```
% Solve generalized eigenvalue problem, cf. (9.0.6)
omega = eig(Mmat,Bmat);

figure('name','resonances');
plot(real(omega),imag(omega),'r*'); hold on;
ax = axis;
plot([ax(1) ax(2)],[0 0],'k-');
plot([ 0 0],[ax(3) ax(4)],'k-');
grid on;
xlabel('{\bf Re(\omega)}','fontsize',14);
ylabel('{\bf Im(\omega)}','fontsize',14);
title(sprintf('R = %d, C= %d, L= %d',R,L,C));
legend('\omega');

print -depsc2 '../PICTURES/rescircomega.eps'
```

resonant frequencies = $\omega \in \{\omega \in \mathbb{R}: \mathbf{A}(\omega) \text{ singular}\}$

If the circuit is operated at a real resonant frequency, the circuit equations will not possess a solution. Of course, the real circuit will always behave in a well-defined way, but the linear model will break down due to extremely large currents and voltages. In an experiment this breakdown manifests itself as a rather explosive meltdown of circuits components. Hence, it is vital to determine resonant frequencies of circuits in order to avoid their destruction.

➥ relevance of numerical methods for solving:

$$\text{Find} \quad \omega \in \mathbb{C} \setminus \{0\}: \quad \mathbf{W} + \imath\omega\mathbf{C} + \frac{1}{\imath\omega}\mathbf{S} \quad \text{singular .}$$

This is a quadratic eigenvalue problem: find $\mathbf{x} \neq 0$, $\omega \in \mathbb{C} \setminus \{0\}$,

$$\mathbf{A}(\omega)\mathbf{x} = (\mathbf{W} + \imath\omega\mathbf{C} + \frac{1}{\imath\omega}\mathbf{S})\mathbf{x} = 0\,. \tag{9.0.4}$$

Substitution: $\mathbf{y} = \frac{1}{\imath\omega}\mathbf{x} \leftrightarrow \mathbf{x} = \imath\omega\mathbf{y}$ [**?**, Sect. 3.4]:

$$(9.0.4) \quad \Leftrightarrow \quad \underbrace{\begin{bmatrix}\mathbf{W} & \mathbf{S}\\ \mathbf{I} & 0\end{bmatrix}}_{:=\mathbf{M}} \underbrace{\begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix}}_{:=\mathbf{z}} = \omega \underbrace{\begin{bmatrix}-\imath\mathbf{C} & 0\\ 0 & -\imath\mathbf{I}\end{bmatrix}}_{:=\mathbf{B}} \begin{bmatrix}\mathbf{x}\\ \mathbf{y}\end{bmatrix} \tag{9.0.5}$$

➢ generalized linear eigenvalue problem of the form: find $\omega \in \mathbb{C}$, $\mathbf{z} \in \mathbb{C}^{2n} \setminus \{0\}$ such that

$$\mathbf{M}\mathbf{z} = \omega\mathbf{B}\mathbf{z}\,. \tag{9.0.6}$$

In this example one is mainly interested in the eigenvalues ω, whereas the eigenvectors $\mathbf{z}$ usually need not be computed.

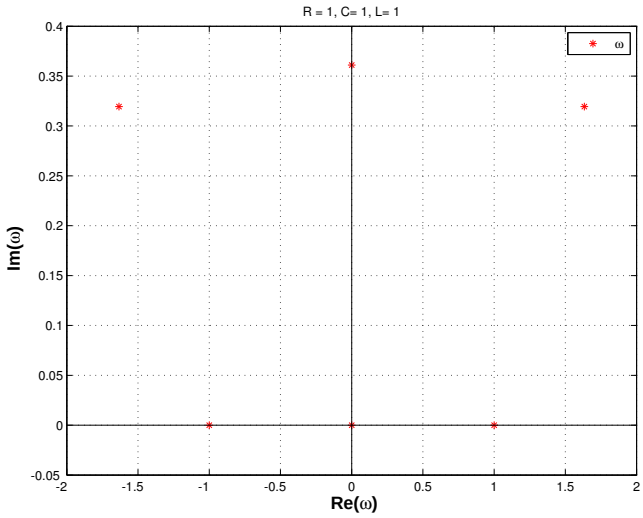


Fig. 315

◁ resonant frequencies for circuit from Code 9.0.3 (including decaying modes with $\mathrm{Im}(\omega) > 0$)

Example 9.0.7 (Analytic solution of homogeneous linear ordinary differential equations → [?, Remark 5.6.1], [?, Sect. 10.1],[?, Sect. 8.1], [?, Ex. 7.3])

Autonomous homogeneous linear ordinary differential equation (ODE):

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \quad , \quad \mathbf{A} \in \mathbb{C}^{n,n} \, . \tag{9.0.8}$$

$$\mathbf{A} = \mathbf{S} \underbrace{\begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}}_{=:\mathbf{D}} \mathbf{S}^{-1} \, , \quad \mathbf{S} \in \mathbb{C}^{n,n} \text{ regular} \quad \Longrightarrow \quad \left(\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \overset{\mathbf{z}=\mathbf{S}^{-1}\mathbf{y}}{\longleftrightarrow} \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} \right) .$$

➢ solution of initial value problem:

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} \, , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{C}^n \quad \Rightarrow \quad \mathbf{y}(t) = \mathbf{S}\mathbf{z}(t) \, , \quad \dot{\mathbf{z}} = \mathbf{D}\mathbf{z} \, , \quad \mathbf{z}(0) = \mathbf{S}^{-1}\mathbf{y}_0 \, .$$

The initial value problem for the *decoupled* homogeneous linear ODE $\dot{\mathbf{z}} = \mathbf{D}\mathbf{z}$ has a simple analytic solution

$$\mathbf{z}_i(t) = \exp(\lambda_i t)(\mathbf{z}_0)_i = \exp(\lambda_i t)\left((\mathbf{S}^{-1})^T_{i,:} \mathbf{y}_0 \right) .$$

In light of Rem. 1.3.3:

$$\mathbf{A} = \mathbf{S} \begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix} \mathbf{S}^{-1} \quad \Leftrightarrow \quad \mathbf{A}((\mathbf{S})_{:,i}) = \lambda_i((\mathbf{S})_{:,i}) \quad i = 1, \ldots, n \, . \tag{9.0.9}$$

In order to find the transformation matrix $\mathbf{S}$ all non-zero solution vectors (= eigenvectors) $\mathbf{x} \in \mathbb{C}^n$ of the linear eigenvalue problem

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

have to be found.

Contents

9.1 Theory of eigenvalue problems

Supplementary reading. [?, Ch. 7], [?, Ch. 9], [?, Sect. 1.7]

Definition 9.1.1. Eigenvalues and eigenvectors → [?, Sects. 7.1,7.2], [?, Sect. 9.1]

- $\lambda \in \mathbb{C}$ eigenvalue (*ger.:* Eigenwert) of $\mathbf{A} \in \mathbb{K}^{n,n}$:⇔ $\underbrace{\det(\lambda\mathbf{I} - \mathbf{A})}_{\text{characteristic polynomial } \chi(\lambda)} = 0$
- spectrum of $\mathbf{A} \in \mathbb{K}^{n,n}$: $\sigma(\mathbf{A}) := \{\lambda \in \mathbb{C}: \lambda \text{ eigenvalue of } \mathbf{A}\}$
- eigenspace (*ger.:* Eigenraum) associated with eigenvalue $\lambda \in \sigma(\mathbf{A})$:
$$\mathbf{EigA}\lambda := \mathcal{N}\lambda\mathbf{I} - \mathbf{A}$$
- $\mathbf{x} \in \mathbf{EigA}\lambda \setminus \{0\}$ ⇒ $\mathbf{x}$ is eigenvector
- Geometric multiplicity (*ger.:* Vielfachheit) of an eigenvalue $\lambda \in \sigma(\mathbf{A})$:
$$m(\lambda) := \dim \mathbf{EigA}\lambda$$

Two simple facts:

$$\lambda \in \sigma(\mathbf{A}) \quad \Rightarrow \quad \dim \mathbf{EigA}\lambda > 0\,, \tag{9.1.2}$$

$$\det(\mathbf{A}) = \det(\mathbf{A}^T) \quad \forall \mathbf{A} \in \mathbb{K}^{n,n} \quad \Rightarrow \quad \sigma(\mathbf{A}) = \sigma(\mathbf{A}^T)\,. \tag{9.1.3}$$

✎ notation: $\rho(\mathbf{A}) := \max\{|\lambda|: \lambda \in \sigma(\mathbf{A})\} \triangleq$ spectral radius of $\mathbf{A} \in \mathbb{K}^{n,n}$

Theorem 9.1.4. Bound for spectral radius

For any matrix norm $\|\cdot\|$ induced by a vector norm (→ Def. 1.5.76)

$$\rho(\mathbf{A}) \le \|\mathbf{A}\|\,.$$

Proof. Let $\mathbf{z} \in \mathbb{C}^n \setminus \{0\}$ be an eigenvector to the largest (in modulus) eigenvalue λ of $\mathbf{A} \in \mathbb{C}^{n,n}$. Then

$$\|\mathbf{A}\| := \sup_{\mathbf{x} \in \mathbb{C}^{n,n} \setminus \{0\}} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} \ge \frac{\|\mathbf{Az}\|}{\|\mathbf{z}\|} = |\lambda| = \rho(\mathbf{A})\,.$$

□

Lemma 9.1.5. Gershgorin circle theorem → [?, Thm. 7.13], [?, Thm. 32.1], [?, Sect. 5.1]

For any $\mathbf{A} \in \mathbb{K}^{n,n}$ *holds true*

$$\sigma(\mathbf{A}) \subset \bigcup_{j=1}^{n} \left\{ z \in \mathbb{C} \colon |z - a_{jj}| \leq \sum_{i \neq j} |a_{ji}| \right\} .$$

Lemma 9.1.6. Similarity and spectrum → [?, Thm. 9.7], [?, Lemma 7.6], [?, Thm. 7.2]

The spectrum of a matrix is invariant with respect to similarity transformations:

$$\forall \mathbf{A} \in \mathbb{K}^{n,n} \colon \quad \sigma(\mathbf{S}^{-1}\mathbf{A}\mathbf{S}) = \sigma(\mathbf{A}) \quad \forall \text{ regular } \mathbf{S} \in \mathbb{K}^{n,n} .$$

Lemma 9.1.7.

Existence of a one-dimensional invariant subspace

$$\forall \mathbf{C} \in \mathbb{C}^{n,n} \colon \quad \exists \mathbf{u} \in \mathbb{C}^{n} \colon \quad \mathbf{C}(\mathrm{Span}\{\mathbf{u}\}) \subset \mathrm{Span}\{\mathbf{u}\} .$$

Theorem 9.1.8. Schur normal form → [?, Thm .2.8.1]

$$\forall \mathbf{A} \in \mathbb{K}^{n,n} \colon \quad \exists \mathbf{U} \in \mathbb{C}^{n,n} \text{ unitary} \colon \quad \mathbf{U}^H \mathbf{A} \mathbf{U} = \mathbf{T} \quad \text{with } \mathbf{T} \in \mathbb{C}^{n,n} \text{ upper triangular} .$$

Corollary 9.1.9. Principal axis transformation

$$\mathbf{A} \in \mathbb{K}^{n,n}, \mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A} \colon \quad \exists \mathbf{U} \in \mathbb{C}^{n,n} \text{ unitary} \colon \quad \mathbf{U}^H \mathbf{A} \mathbf{U} = \mathrm{diag}(\lambda_1, \ldots, \lambda_n) , \quad \lambda_i \in \mathbb{C} .$$

A matrix $\mathbf{A} \in \mathbb{K}^{n,n}$ with $\mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A}$ is called normal.

Examples of normal matrices are
- Hermitian matrices: $\mathbf{A}^H = \mathbf{A}$ ➤ $\sigma(\mathbf{A}) \subset \mathbb{R}$
- unitary matrices: $\mathbf{A}^H = \mathbf{A}^{-1}$ ➤ $|\sigma(\mathbf{A})| = 1$
- skew-Hermitian matrices: $\mathbf{A} = -\mathbf{A}^H$ ➤ $\sigma(\mathbf{A}) \subset i\mathbb{R}$

➤

Normal matrices can be diagonalized by *unitary* similarity transformations

Symmetric real matrices can be diagonalized by *orthogonal* similarity transformations

In Cor. 9.1.9:
- $\lambda_1, \ldots, \lambda_n$ = eigenvalues of $\mathbf{A}$
- Columns of $\mathbf{U}$ = orthonormal basis of eigenvectors of $\mathbf{A}$

Eigenvalue problems: (EVPs)
❶ Given $\mathbf{A} \in \mathbb{K}^{n,n}$ find **all** eigenvalues (= spectrum of $\mathbf{A}$).
❷ Given $\mathbf{A} \in \mathbb{K}^{n,n}$ find $\sigma(\mathbf{A})$ plus **all** eigenvectors.
❸ Given $\mathbf{A} \in \mathbb{K}^{n,n}$ find **a few** eigenvalues and associated eigenvectors

(Linear) generalized eigenvalue problem:

Given $\mathbf{A} \in \mathbb{C}^{n,n}$, regular $\mathbf{B} \in \mathbb{C}^{n,n}$, seek $\mathbf{x} \neq 0$, $\lambda \in \mathbb{C}$

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \mathbf{B}^{-1}\mathbf{A}\mathbf{x} = \lambda \mathbf{x} \,. \tag{9.1.10}$$

$\mathbf{x} \hat{=}$ generalized eigenvector, $\lambda \hat{=}$ generalized eigenvalue

Obviously every generalized eigenvalue problem is equivalent to a standard eigenvalue problem

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \mathbf{B}^{-1}\mathbf{A} = \lambda \mathbf{x} \,.$$

However, usually it is not advisable to use this equivalence for numerical purposes!

Remark 9.1.11 (Generalized eigenvalue problems and Cholesky factorization)

If $\mathbf{B} = \mathbf{B}^H$ s.p.d. ($\rightarrow$ Def. 1.1.8) with Cholesky factorization $\mathbf{B} = \mathbf{R}^H\mathbf{R}$

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{B}\mathbf{x} \quad \Leftrightarrow \quad \widetilde{\mathbf{A}}\mathbf{y} = \lambda \mathbf{y} \quad \text{where } \widetilde{\mathbf{A}} := \mathbf{R}^{-H}\mathbf{A}\mathbf{R}^{-1},\ \mathbf{y} := \mathbf{R}\mathbf{x} \,.$$

➞ This transformation can be used for efficient computations.

9.2 "Direct" Eigensolvers

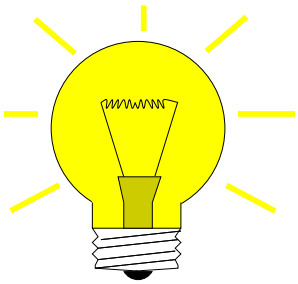
Purpose: solution of eigenvalue problems ❶, ❷ for dense matrices "up to machine precision"

MATLAB-function: `eig`

`d = eig(A)` : computes spectrum $\sigma(\mathbf{A}) = \{d_1, \ldots, d_n\}$ of $\mathbf{A} \in \mathbb{C}^{n,n}$
`[V,D] = eig(A)` : computes $\mathbf{V} \in \mathbb{C}^{n,n}$, *diagonal* $\mathbf{D} \in \mathbb{C}^{n,n}$ such that $\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{D}$

Remark 9.2.1 (QR-Algorithm → [?, Sect. 7.5], [?, Sect. 10.3],[?, Ch. 26],[?, Sect. 5.5-5.7])

Note: All "direct" eigensolvers are iterative methods



Idea: Iteration based on successive **unitary** similarity transformations

$$\mathbf{A} = \mathbf{A}^{(0)} \longrightarrow \mathbf{A}^{(1)} \longrightarrow \ldots \longrightarrow \begin{cases} \text{diagonal matrix} \\ \text{upper triangular matrix} \\ (\rightarrow \text{Thm. 9.1.8}) \end{cases}$$

(superior stability of unitary transformations, see **??**)

QR-algorithm (with shift)

✦ in general: quadratic convergence
✦ cubic convergence for normal matrices
(→ [?, Sect. 7.5,8.2])

MATLAB-code 9.2.2: QR-algorithm with shift

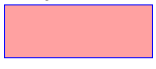
```
function d = eigqr(A,tol)
n = size(A,1);
while (norm(tril(A,-1)) > tol*norm(A))
% shift by eigenvalue of lower right 2×2 block closest to (A)_{n,n}
  sc = eig(A(n-1:n,n-1:n));
  [dummy,si] = min(abs(sc-A(n,n)));
  shift = sc(si);
  [Q,R] = qr( A - shift * eye(n));
  A = Q'*A*Q;
end
d = diag(A);
```

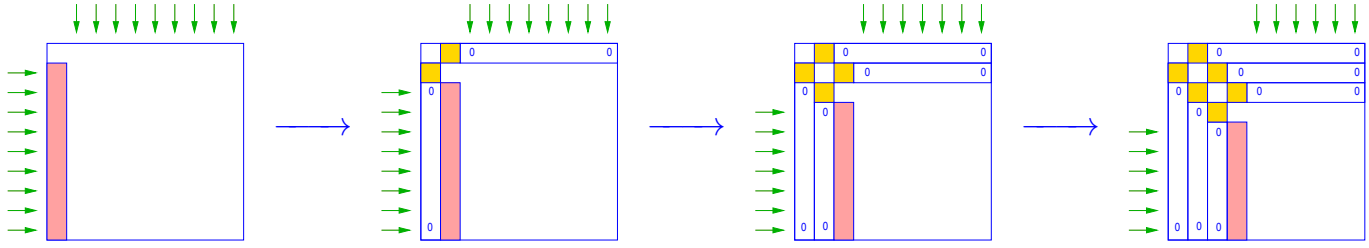
Computational cost: $O(n^3)$ operations per step of the QR-algorithm

Library implementations of the QR-algorithm provide *numerically stable* eigensolvers (→ Def.1.5.85)

Remark 9.2.3 (Unitary similarity transformation to tridiagonal form)

Successive Householder similarity transformations of $\mathbf{A} = \mathbf{A}^H$:

(→ ≙ affected rows/columns,  ≙ targeted vector)



 transformation to tridiagonal form **!** (for general matrices a similar strategy can achieve a similarity transformation to upper Hessenberg form)

this transformation is used as a preprocessing step for QR-algorithm ➢ `eig`.

Similar functionality for generalized EVP $\mathbf{Ax} = \lambda\mathbf{Bx}$, $\mathbf{A},\mathbf{B} \in \mathbb{C}^{n,n}$

`d = eig(A,B)`	:	computes all generalized eigenvalues
`[V,D] = eig(A,B)`	:	computes $\mathbf{V} \in \mathbb{C}^{n,n}$, *diagonal* $\mathbf{D} \in \mathbb{C}^{n,n}$ such that $\mathbf{AV} = \mathbf{BVD}$

Note: (Generalized) eigenvectors can be recovered as columns of $\mathbf{V}$:

$$\mathbf{AV} = \mathbf{VD} \quad \Leftrightarrow \quad \mathbf{A}(\mathbf{V})_{:,i} = (\mathbf{D})_{i,i}\mathbf{V}_{:,i}\,,$$

if $\mathbf{D} = \operatorname{diag}(d_1,\ldots,d_n)$.

Remark 9.2.4 (Computational effort for eigenvalue computations)

Computational effort (#elementary operations) for `eig()`:

eigenvalues & eigenvectors of $\mathbf{A} \in \mathbb{K}^{n,n}$	$\sim 25n^3 + O(n^2)$
only eigenvalues of $\mathbf{A} \in \mathbb{K}^{n,n}$	$\sim 10n^3 + O(n^2)$
eigenvalues and eigenvectors $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim 9n^3 + O(n^2)$
only eigenvalues of $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim \frac{4}{3}n^3 + O(n^2)$
only eigenvalues of tridiagonal $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim 30n^2 + O(n)$

$O(n^3)$!

Note: `eig` not available for sparse matrix arguments

Exception: `d=eig(A)` for sparse *Hermitian* matrices

Example 9.2.5 (Runtimes of `eig`)

MATLAB-code 9.2.6:

```
A = rand(500,500); B = A'*A; C = gallery('tridiag',500,1,3,1);
```

➢
- ✦ $\mathbf{A}$ generic dense matrix
- ✦ $\mathbf{B}$ symmetric (s.p.d. → Def. 1.1.8) matrix
- ✦ $\mathbf{C}$ s.p.d. *tridiagonal* matrix

MATLAB-code 9.2.7: measuring runtimes of `eig`

```
function eigtiming

A = rand(500,500); B = A'*A;
C = gallery('tridiag',500,1,3,1);
times = [];
for n=5:5:500
  An = A(1:n,1:n); Bn = B(1:n,1:n); Cn = C(1:n,1:n);
  t1 = 1000; for k=1:3, tic; d = eig(An); t1 = min(t1,toc); end
  t2 = 1000; for k=1:3, tic; [V,D] = eig(An); t2 = min(t2,toc);
```

```
    end
  t3 = 1000; for k=1:3, tic; d = eig(Bn); t3 = min(t3,toc); end
  t4 = 1000; for k=1:3, tic; [V,D] = eig(Bn); t4 = min(t4,toc);
    end
  t5 = 1000; for k=1:3, tic; d = eig(Cn); t5 = min(t5,toc); end
  times = [times; n t1 t2 t3 t4 t5];
end

figure;
loglog(times(:,1),times(:,2),'r+', times(:,1),times(:,3),'m*',...
       times(:,1),times(:,4),'cp', times(:,1),times(:,5),'b^',...
       times(:,1),times(:,6),'k.');
xlabel('{\bf matrix size n}','fontsize',14);
ylabel('{\bf time [s]}','fontsize',14);
title('eig runtimes');
legend('d = eig(A)','[V,D] = eig(A)','d = eig(B)','[V,D] =
  eig(B)','d = eig(C)',...
       'location','northwest');

print -depsc2 '../PICTURES/eigtimingall.eps'

figure;
loglog(times(:,1),times(:,2),'r+', times(:,1),times(:,3),'m*',...
     times(:,1),(times(:,1).^3)/(times(1,1)^3)*times(1,2),'k-');
xlabel('{\bf matrix size n}','fontsize',14);
ylabel('{\bf time [s]}','fontsize',14);
title('nxn random matrix');
legend('d = eig(A)','[V,D] =
  eig(A)','O(n^3)','location','northwest');

print -depsc2 '../PICTURES/eigtimingA.eps'

figure;
loglog(times(:,1),times(:,4),'r+', times(:,1),times(:,5),'m*',...
     times(:,1),(times(:,1).^3)/(times(1,1)^3)*times(1,2),'k-');
xlabel('{\bf matrix size n}','fontsize',14);
ylabel('{\bf time [s]}','fontsize',14);
title('nxn random Hermitian matrix');
legend('d = eig(A)','[V,D] =
  eig(A)','O(n^3)','location','northwest');

print -depsc2 '../PICTURES/eigtimingB.eps'

figure;
loglog(times(:,1),times(:,6),'r*',...
     times(:,1),(times(:,1).^2)/(times(1,1)^2)*times(1,2),'k-');
xlabel('{\bf matrix size n}','fontsize',14);
ylabel('{\bf time [s]}','fontsize',14);
title('nxn tridiagonel Hermitian matrix');
legend('d = eig(A)','O(n^2)','location','northwest');
```

```
55
56 print -depsc2 '../PICTURES/eigtimingC.eps'
```

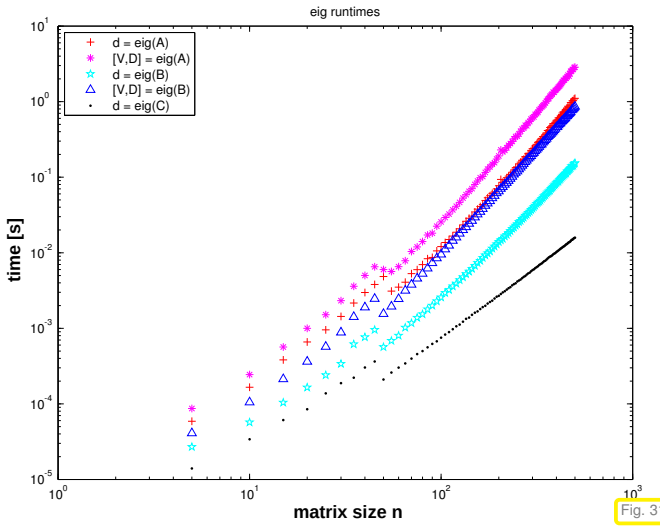


Fig. 316

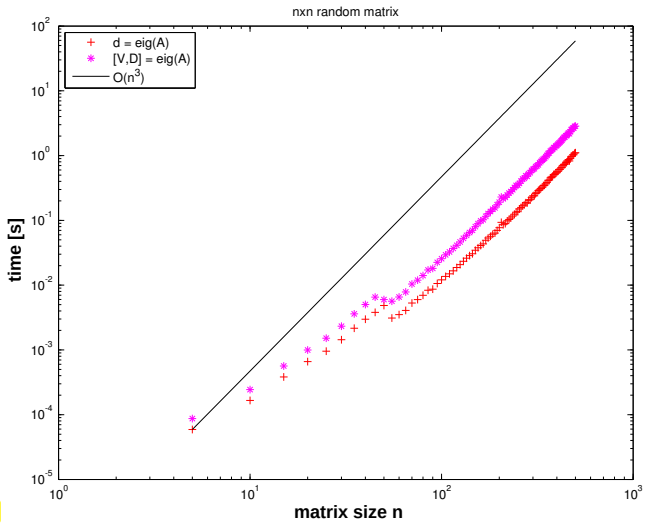


Fig. 317

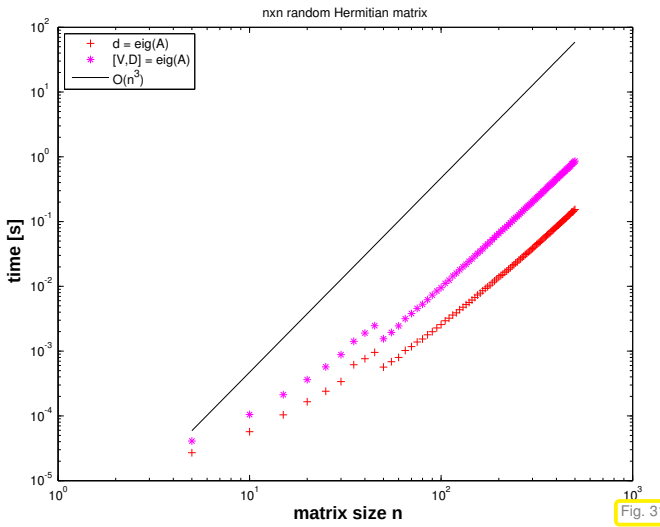


Fig. 318

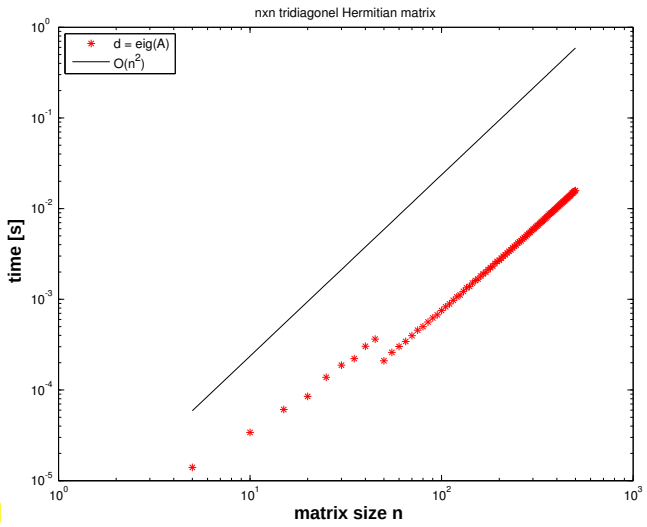


Fig. 319

☛ For the sake of efficiency: think which information you really need when computing eigenvalues/eigenvectors of dense matrices

Potentially more efficient methods for *sparse matrices* will be introduced below in Section 9.3, 9.4.

9.3 Power Methods

9.3.1 Direct power method

Supplementary reading. [**?**, Sect. 7.5], [**?**, Sect. 5.3.1], [**?**, Sect. 5.3]

Example 9.3.1 ((Simplified) Page rank algorithm → [?, ?])

Model: Random surfer visits a web page, stays there for fixed time Δt, and then
❶ either follows each of ℓ links on a page with probabilty $1/\ell$.
❷ or resumes surfing at a randomly (with equal probability) selected page

Option ❷ is chosen with probability d, $0 \leq d \leq 1$, option ❶ with probability $1-d$.

Stationary Markov chain, state space $\hat{=}$ set of all web pages

Question: Fraction of time spent by random surfer on i-th page (= page rank $x_i \in [0,1]$)

This number $\in]0,1[$ can be used to gauge the "importance" of a web page, which, in turns, offers a way to sort the hits resulting from a keyword query: the GOOGLE idea.

Method: Stochastic simulation ▷

MATLAB-code 9.3.2: stochastic page rank simulation

```
1  function prstochsim(Nhops)
2  % Load web graph data stored in N×N-matrix G
3  load harvard500.mat;
4  N = size(G,1); d = 0.15;
5  count = zeros(1,N); cp = 1;
6  figure('position',[0 0 1200 1000]); pause;
7  for n=1:Nhops
8    % Find links from current page cp
9    idx = find(G(:,cp)); l = size(idx,1); rn = rand(); %
10   % If no links, jump to any other pages with equal probability
11   if (isempty(idx)), cp = floor(rn*N)+1;
12   % With probabilty d jump to any other page
13   elseif (rn < d), cp = floor(rn/d*N)+1;
14   % Follow outgoing links with equal probabilty
15   else cp = idx(floor((rn-d)/(1-d)*l)+1,1);
16   end
17   count(cp) = count(cp) + 1;
18   plot(1:N,count/n,'r.'); axis([0 N+1 0 0.1]);
19   xlabel('{\bf harvard500: no. of page}','fontsize',14);
20   ylabel('{\bf page rank}','fontsize',14);
21   title(sprintf('{\\bf page rank, harvard500: %d
       hops}',n),'fontsize',14);
22   drawnow;
23 end
```

Explanations Code 9.3.2:

- Line 9: `rand` generates uniformly distributed pseudo-random numbers $\in [0,1[$
- Web graph encoded in $\mathbf{G} \in \{0,1\}^{N,N}$:

$$(\mathbf{G})_{ij} = 1 \quad \Rightarrow \quad \text{link } j \to i\,,$$

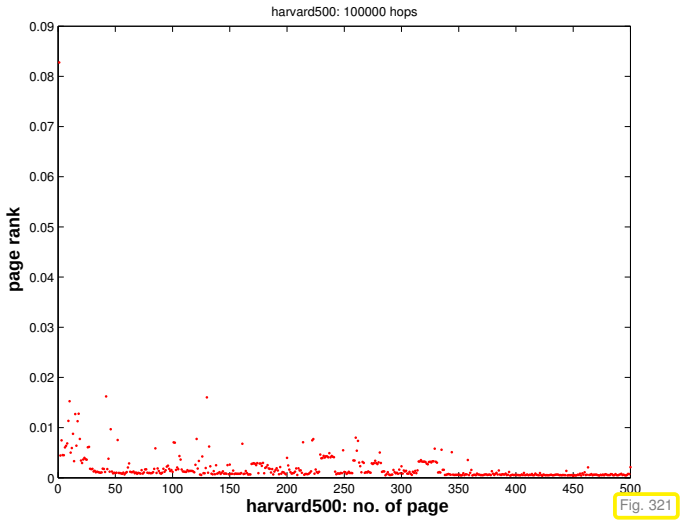


Fig. 320

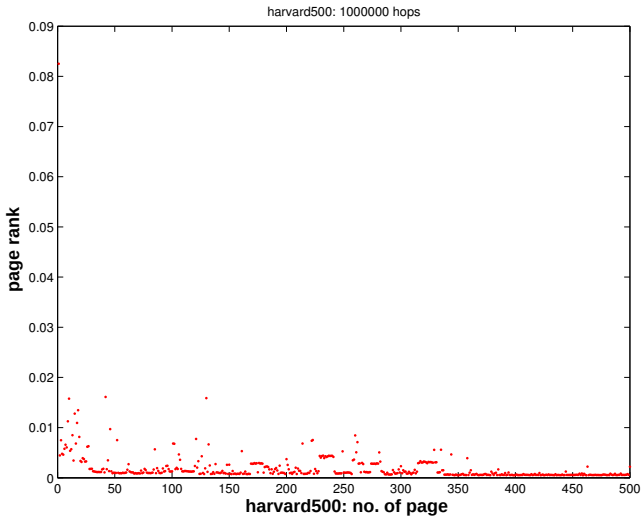


Fig. 321

Observation: relative visit times stabilize as the number of hops in the stochastic simulation $\to \infty$.

The limit distribution is called stationary distribution/invariant measure of the Markov chain. This is what we seek.

✦ Numbering of pages $1, \ldots, N$, $\ell_i \mathrel{\hat{=}}$ number of links from page i

✦ $N \times N$-matrix of transition probabilities page $j \to$ page i: $\mathbf{A} = (a_{ij})_{i,j=1}^{N} \in \mathbb{R}^{N,N}$

$$a_{ij} \in [0,1] \quad \mathrel{\hat{=}} \text{probability to jump from page } j \text{ to page } i.$$

$$\Rightarrow \quad \sum_{i=1}^{N} a_{ij} = 1 \,. \tag{9.3.3}$$

A matrix $\mathbf{A} \in [0,1]^{N,N}$ with the property (9.3.3) is called a (column) stochastic matrix.

"Meaning" of $\mathbf{A}$: given $\mathbf{x} \in [0,1]^N$, $\|\mathbf{x}\|_1 = 1$, where x_i is the probability of the surfer to visit page i, $i = 1, \ldots, N$, at an instance t in time, $\mathbf{y} = \mathbf{A}\mathbf{x}$ satisfies

$$y_j \geq 0 \quad , \quad \sum_{j=1}^{N} y_j = \sum_{j=1}^{N} \sum_{i=1}^{N} a_{ji} x_i = \sum_{i=1}^{N} x_i \underbrace{\sum_{j=1}^{N} a_{ij}}_{=1} = \sum_{i=1}^{N} x_i = 1 \,.$$

$y_j \mathrel{\hat{=}}$ probability for visiting page j at time $t + \Delta t$.

Transition probability matrix for page rank computation

$$(\mathbf{A})_{ij} = \begin{cases} \frac{1}{N} & \text{, if } (\mathbf{G})_{ij} = 0 \;\forall i = 1, \ldots, N \,, \\ d/N + (1-d)\dfrac{(\mathbf{G})_{ij}}{\ell_j} & \text{else.} \end{cases} \tag{9.3.4}$$

random jump to any other page — follow link

MATLAB-code 9.3.5: transition probability matrix for page rank

```
function A = prbuildA(G,d)
N = size(G,1);
l = full(sum(G)); idx = find(l>0);

s = zeros(N,1); s(idx) = 1./l(idx);
ds = ones(N,1)/N; ds(idx) = d/N;
A = ones(N,1)*ones(1,N)*diag(ds) +
   (1-d)*G*diag(s);
```

Note: special treatment of zero columns of $\mathbf{G}$, *cf.* (9.3.4)!

Stochastic simulation based on a single surfer is *slow*. Alternatives?

Thought experiment: Instead of a single random surfer we may consider $m \in \mathbb{N}$, $m \gg 1$, of them who visit pages independently. The fraction of time $m \cdot T$ they all together spend on page i will obviously be the same for $T \to \infty$ as that for a single random surfer.

Instead of counting the surfers we watch the proportions of them visiting particular web pages at an instance of time. Thus, after the k-th hop we can assign a number $x_i^{(k)} \in [0,1]$ to web page i, which gives the proportion of surfers currently on that page: $x_i^{(k)} := \frac{n_i^{(k)}}{m}$, where $n_i^{(k)} \in \mathbb{N}_0$ designates the number of surfers on page i after the k-th hop.

Now consider $m \to \infty$. The law of law of large numbers suggests that the ("infinitely many") surfers visiting page j will move on to other pages proportional to the transistion probabilities a_{ij}: in terms of proportions, for $m \to \infty$ the stochastic evolution becomes a deterministic discrete dynamical system and we find

$$x_i^{(k+1)} = \sum_{j=1}^{N} a_{ij} x_j^{(k)} \,, \tag{9.3.6}$$

that is, the proportion of surfers ending up on page i equals the sum of the proportions on the "source pages" weighted with the transition probabilities.

Notice that (9.3.6) amounts to matrix×vector. Thus, writing $\mathbf{x}^{(0)} \in [0,1]^N$, $\left\|\mathbf{x}^{(0)}\right\| = 1$ for the initial distribution of the surfers on the net we find

$$\mathbf{x}^{(k)} = \mathbf{A}^k \mathbf{x}^{(0)}$$

will be their mass distribution after k hops. If the limit exists, the i-th component of $\mathbf{x}^* := \lim_{k\to\infty} \mathbf{x}^{(k)}$ tells us which fraction of the (infinitely many) surfers will be visiting page i most of the time. Thus, $\mathbf{x}^*$ yields the stationary distribution of the Markov chain.

MATLAB-code 9.3.7: tracking fractions of many surfers

```
function prpowitsim(d,Nsteps)
% MATLAB way of specifying Default arguments
if (nargin < 2), Nsteps = 5; end
if (nargin < 1), d = 0.15; end
% load connectivity matrix and build transition matrix
```

```
load harvard500.mat; A = prbuildA(G,d);
N = size(A,1); x = ones(N,1)/N;

figure('position',[0 0 1200 1000]);
plot(1:N,x,'r+'); axis([0 N+1 0 0.1]);
% Plain power iteration for stochastic matrix A
for l=1:Nsteps
  pause; x = A*x; plot(1:N,x,'r+'); axis([0 N+1 0 0.1]);
  title(sprintf('{\\bf step %d}',l),'fontsize',14);
  xlabel('{\bf harvard500: no. of page}','fontsize',14);
  ylabel('{\bf page rank}','fontsize',14); drawnow;
end
```

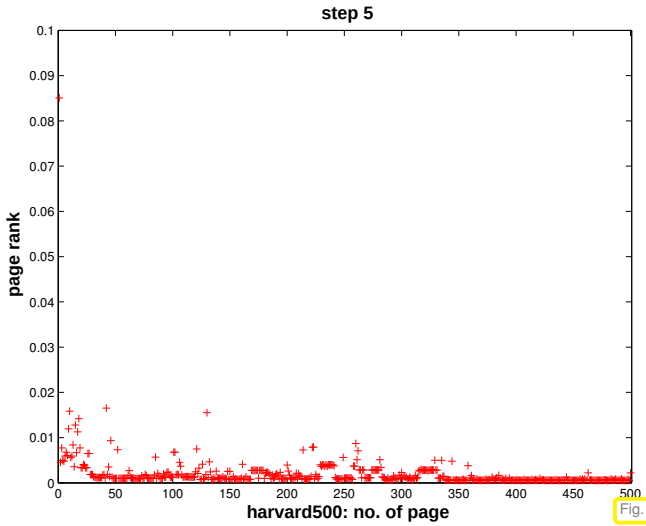


Fig. 322

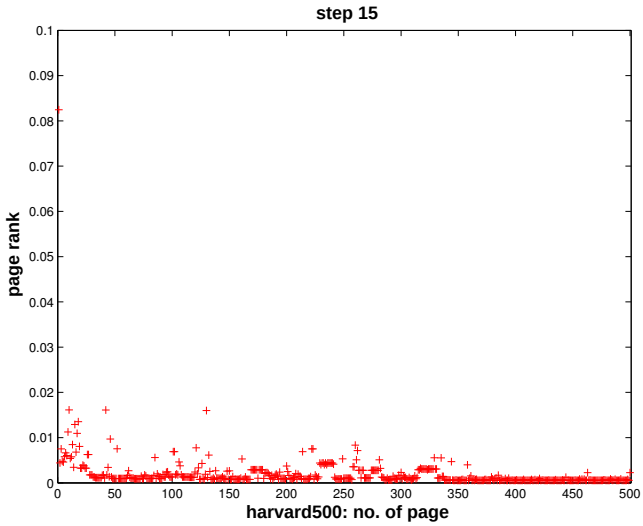


Fig. 323

Comparison:

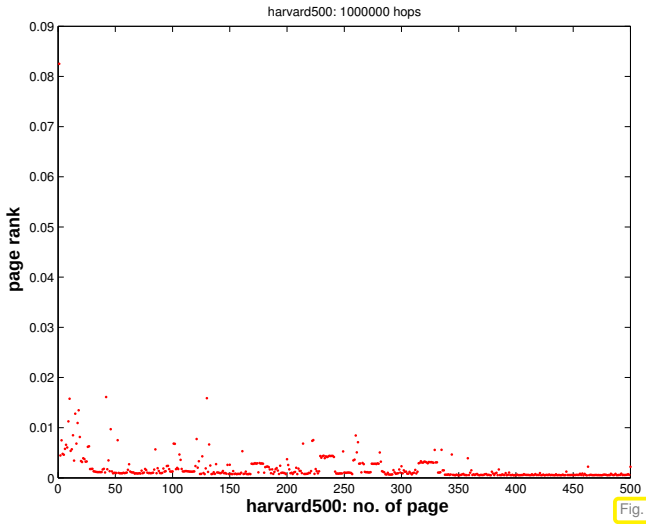


Fig. 324

Single surfer stochastic simulation

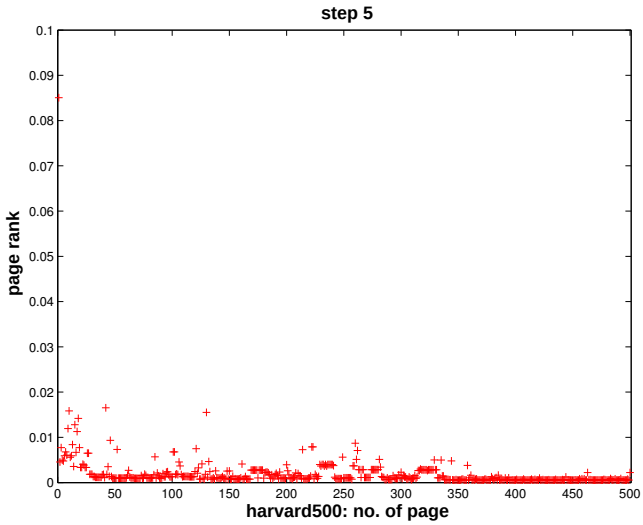


Fig. 325

Power method, Code 9.3.7

Observation: Convergence of the $\mathbf{x}^{(k)} \to \mathbf{x}^*$, and the limit must be a fixed point of the iteration function:

$$\succ \quad \mathbf{A}\mathbf{x}^* = \mathbf{x}^* \quad \Rightarrow \quad \mathbf{x}^* \in \mathbf{EigA}1 \,.$$

Does $\mathbf{A}$ possess an eigenvalue $= 1$? Does the associated eigenvector really provide a probability distribution (after scaling), that is, are all of its entries non-negative? Is this probability distribution unique? To answer these questions we have to study the matrix $\mathbf{A}$:

For every stochastic matrix $\mathbf{A}$, by definition (9.3.3)

$$\mathbf{A}^T\mathbf{1} = \mathbf{1} \overset{(9.1.3)}{\Rightarrow} 1 \in \sigma(\mathbf{A}) ,$$

$$\text{(1.5.80)} \Rightarrow \|\mathbf{A}\|_1 = 1 \overset{\text{Thm. 9.1.4}}{\Rightarrow} \rho(\mathbf{A}) = 1 ,$$

where $\rho(\mathbf{A})$ is the spectral radius of the matrix $\mathbf{A}$, see Section 9.1.

For $\mathbf{r} \in \mathbf{Eig}\mathbf{A}1$, that is, $\mathbf{A}\mathbf{r} = \mathbf{r}$, denote by $|\mathbf{r}|$ the vector $(|r_i|)_{i=1}^N$. Since all entries of $\mathbf{A}$ are non-negative, we conclude by the triangle inequality that $\|\mathbf{A}\mathbf{r}\|_1 \leq \|\mathbf{A}|\mathbf{r}|\|_1$

$$\Rightarrow \quad 1 = \|\mathbf{A}\|_1 = \sup_{\mathbf{x}\in\mathbb{R}^N} \frac{\|\mathbf{A}\mathbf{x}\|_1}{\|\mathbf{x}\|_1} \geq \frac{\|\mathbf{A}|\mathbf{r}|\|_1}{\||\mathbf{r}|\|_1} \geq \frac{\|\mathbf{A}\mathbf{r}\|_1}{\|\mathbf{r}\|_1} = 1 .$$

$$\Rightarrow \quad \|\mathbf{A}|\mathbf{r}|\|_1 = \|\mathbf{A}\mathbf{r}\|_1 \overset{\text{if } a_{ij}>0}{\Rightarrow} |\mathbf{r}| = \pm\mathbf{r} .$$

Hence, different components of $\mathbf{r}$ cannot have opposite sign, which means, that $\mathbf{r}$ *can be chosen to have non-negative entries*, if the entries of $\mathbf{A}$ are strictly positive, which is the case for $\mathbf{A}$ from (9.3.4). After normalization $\|\mathbf{r}\|_1 = 1$ the eigenvector can be regarded as a probability distribution on $\{1,\ldots,N\}$.

If $\mathbf{A}\mathbf{r} = \mathbf{r}$ and $\mathbf{A}\mathbf{s} = \mathbf{s}$ with $(\mathbf{r})_i \geq 0$, $(\mathbf{s})_i \geq 0$, $\|\mathbf{r}\|_1 = \|\mathbf{s}\|_1 = 1$, then $\mathbf{A}(\mathbf{r}-\mathbf{s}) = \mathbf{r}-\mathbf{s}$. Hence, by the above considerations, also all the entries of $\mathbf{r}-\mathbf{s}$ are either non-negative or non-positive. By the assumptions on $\mathbf{r}$ and $\mathbf{s}$ this is only possible, if $\mathbf{r}-\mathbf{s} = 0$. We conclude that

$$\mathbf{A} \in]0,1]^{N,N} \text{ stochastic} \quad \Rightarrow \quad \dim \mathbf{Eig}\mathbf{A}1 = 1 . \tag{9.3.8}$$

Sorting the pages according to the size of the corresponding entries in $\mathbf{r}$ yields the famous "page rank".

MATLAB-code 9.3.9: computing page rank vector $\mathbf{r}$ via `eig`

```
function prevp
load harvard500.mat; d = 0.15;
[V,D] = eig(prbuildA(G,d));

figure; r = V(:,1); N = length(r);
plot(1:N,r/sum(r),'m.'); axis([0 N+1 0 0.1]);
xlabel('{\bf harvard500: no. of page}','fontsize',14);
ylabel('{\bf entry of r-vector}','fontsize',14);
title('harvard 500: Perron-Frobenius vector');
print -depsc2 '../PICTURES/prevp.eps';
```

Plot of entries of *unique* vector $\mathbf{r} \in \mathbb{R}^N$ with

$$0 \leq (\mathbf{r})_i \leq 1 ,$$
$$\|\mathbf{r}\|_1 = 1 ,$$
$$\boxed{\mathbf{A}\mathbf{r} = \mathbf{r}} .$$

Inefficient implementation!

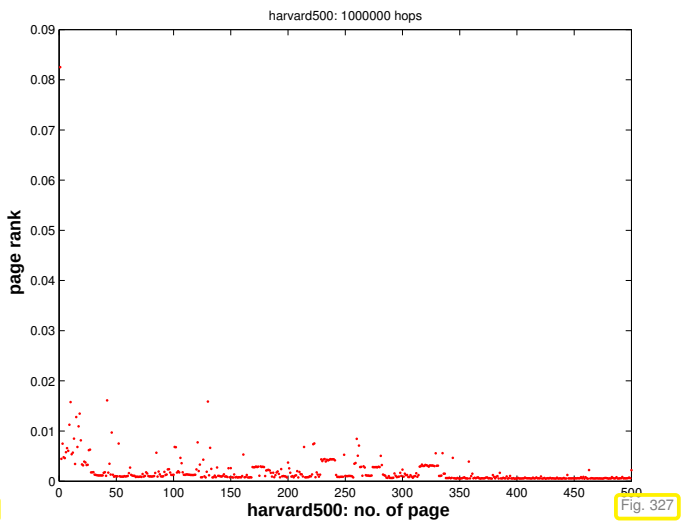


Fig. 326

stochastic simulation

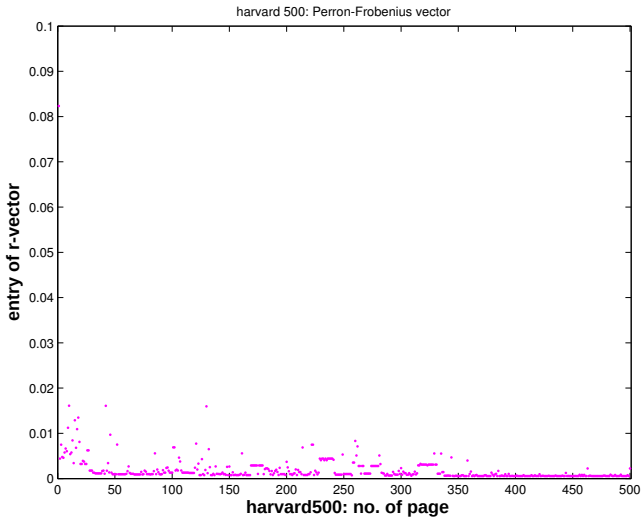


Fig. 327

eigenvector computation

The possibility to compute the stationary probability distribution of a Markov chain through an eigenvector of the transition probability matrix is due to a property of stationary Markov chains called ergodicity.

$\mathbf{A} \hat{=}$ page rank transition probability matrix, see Code 9.3.5, $d = 0.15$, harvard500 example.

Errors: ▷

$$\left\| \mathbf{A}^k \mathbf{x}_0 - \mathbf{r} \right\|_1 ,$$

with $\mathbf{x}_0 = \mathbf{1}/N$, $N = 500$.
We observe linear convergence! (→ Def. 8.1.9, iteration error vs. iteration count ≈ straight line lin-log plot)

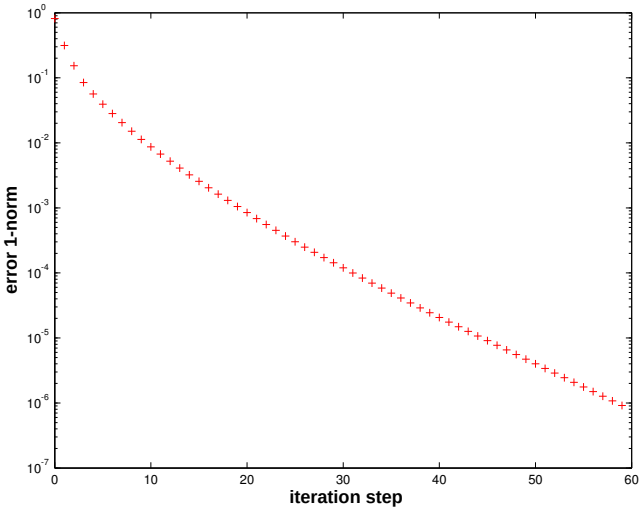
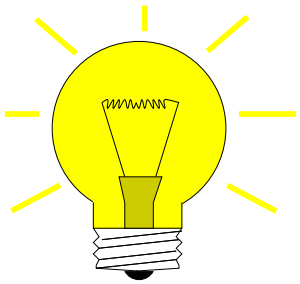


Fig. 328

The computation of page rank amounts to finding the eigenvector of the matrix $\mathbf{A}$ of transition probabilities that belongs to its *largest* eigenvalue 1. This is addressed by an important class of practical eigenvalue problems:

Task:	given $\mathbf{A} \in \mathbb{K}^{n,n}$, find largest (in modulus) eigenvalue of $\mathbf{A}$ and (an) associated eigenvector.



Idea: (suggested by page rank computation, Code 9.3.7)
Iteration: $\mathbf{z}^{(k+1)} = \mathbf{A}\mathbf{z}^{(k)}$, $\mathbf{z}^{(0)}$ arbitrary

Example 9.3.10 (Power iteration → Ex. 9.3.1)

Try the above iteration for general 10×10-matrix, largest eigenvalue 10, algebraic multiplicity 1.

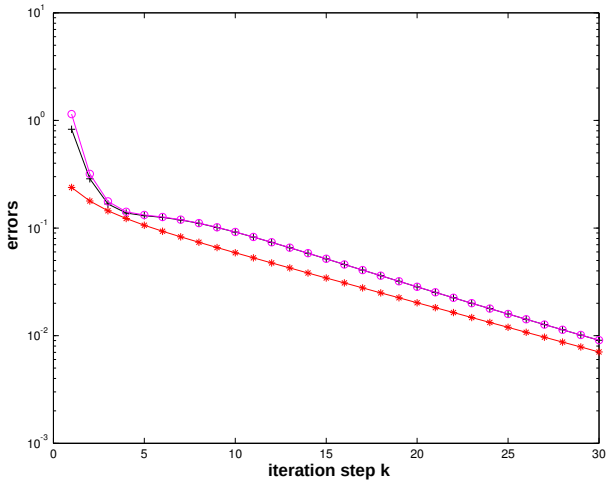


Fig. 329

MATLAB-code 9.3.11:

```
d = (1:10)'; n = length(d);
S = triu(diag(n:-1:1,0)+...
    ones(n,n));
A = S*diag(d,0)*inv(S);
```

◁ error norm $\left\| \frac{\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|} - (\mathbf{S})_{:,10} \right\|$
(Note: $(\mathbf{S})_{:,10} \hat{=}$ eigenvector for eigenvalue 10)

$\mathbf{z}^{(0)}$ = random vector

Observation: linear convergence of (normalized) eigenvectors!

Suggests direct power method (*ger.:* Potenzmethode): iterative method (→ Section 8.1)

$$\begin{aligned} &\text{initial guess:} \quad \mathbf{z}^{(0)} \text{ "arbitrary"}, \\ &\text{next iterate:} \quad \mathbf{w} := \mathbf{A}\mathbf{z}^{(k-1)}, \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, \quad k = 1,2,\ldots. \end{aligned} \tag{9.3.12}$$

Note: the "normalization" of the iterates in (9.3.12) does not change anything (in exact arithmetic) and helps *avoid overflow* in floating point arithmetic.

Computational effort: $1\times$ matrix$\times$vector per step ➢ inexpensive for sparse matrices

A persuasive theoretical justification for the direct power method:

Assume $\mathbf{A} \in \mathbb{K}^{n,n}$ to be diagonalizable:

$$\Leftrightarrow \quad \exists \text{ basis } \{\mathbf{v}_1,\ldots,\mathbf{v}_n\} \text{ of eigenvectors of } \mathbf{A}: \quad \mathbf{A}\mathbf{v}_j = \lambda_j \mathbf{v}_j,\ \lambda_j \in \mathbb{C}.$$

Assume

$$|\lambda_1| \le |\lambda_2| \le \cdots \le |\lambda_{n-1}| < |\lambda_n| \quad , \quad \|\mathbf{v}_j\|_2 = 1 \,. \tag{9.3.13}$$

Key observations for power iteration (9.3.12)

$$\mathbf{z}^{(k)} = \mathbf{A}^k \mathbf{z}^{(0)} \qquad (\rightarrow \text{name "power method"}) \tag{9.3.14}$$

$$\mathbf{z}^{(0)} = \sum_{j=1}^{n} \zeta_j \mathbf{v}_j \quad \Rightarrow \quad \mathbf{z}^{(k)} = \sum_{j=1}^{n} \zeta_j \lambda_j^k \mathbf{v}_j \,. \tag{9.3.15}$$

Due to (9.3.13) for large $k \gg 1$ ($\Rightarrow |\lambda_n^k| \gg |\lambda_j^k|$ for $j \neq n$) the contribution of $\mathbf{v}_n$ (size $\zeta_n \lambda_n^k$) in the *eigenvector expansion* (9.3.15) will be much larger than the contribution (size $\zeta_n \lambda_j^k$) of any other eigenvector (, if $\zeta_n \neq 0$): the eigenvector for λ_n will swamp all other for $k \to \infty$.

Further (9.3.15) nutures expectation: $\mathbf{v}_n$ will become dominant in $\mathbf{z}^{(k)}$ the faster, the better $|\lambda_n|$ is separated from $|\lambda_{n-1}|$, see Thm. 9.3.21 for rigorous statement.

$\mathbf{z}^{(k)} \to$ eigenvector, but how do we get the associated eigenvalue λ_n **?**

When (9.3.12) has converged, two common ways to recover $\lambda_{\max}$ → [**?**, Alg. 7.20]

❶ $\mathbf{A}\mathbf{z}^{(k)} \approx \lambda_{\max}\mathbf{z}^{(k)}$ ➢ $|\lambda_n| \approx \frac{\|\mathbf{A}\mathbf{z}^{(k)}\|}{\|\mathbf{z}^{(k)}\|}$ (modulus only!)

❷ $\lambda_{\max} \approx \operatorname{argmin}_{\theta\in\mathbb{R}} \left\|\mathbf{A}\mathbf{z}^{(k)} - \theta\mathbf{z}^{(k)}\right\|_2^2$ ➤ $\lambda_{\max} \approx \frac{(\mathbf{z}^{(k)})^H \mathbf{A}\mathbf{z}^{(k)}}{\|\mathbf{z}^{(k)}\|_2^2}$.

This latter formula is extremely useful, which has earned it a special name:

Definition 9.3.16.

For $\mathbf{A} \in \mathbb{K}^{n,n}$, $\mathbf{u} \in \mathbb{K}^n$ the Rayleigh quotient is defined by

$$\rho_{\mathbf{A}}(\mathbf{u}) := \frac{\mathbf{u}^H \mathbf{A}\mathbf{u}}{\mathbf{u}^H \mathbf{u}} .$$

An immediate consequence of the definitions:

$$\lambda \in \sigma(\mathbf{A}) \quad , \quad \mathbf{z} \in \mathbf{Eig}\lambda\mathbf{A} \quad \Rightarrow \quad \rho_{\mathbf{A}}(\mathbf{z}) = \lambda . \tag{9.3.17}$$

Example 9.3.18 (Direct power method → Ex. 9.3.18 cnt'd)

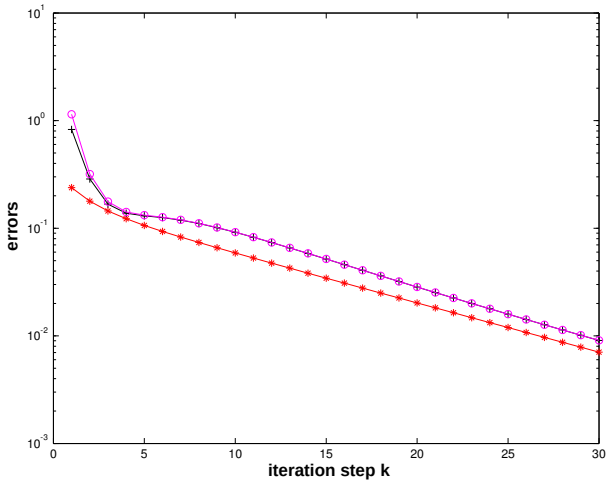


Fig. 330

MATLAB-code 9.3.19:
n = length(d); S = triu(diag(n:-1:1,0)+... ones(n,n)); A = S*diag(d,0)*inv(S);

```
d = (1:10)';
```

o : error $|\lambda_n - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$

◁ * : error norm $\left\| \mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n} \right\|$

+ : $\left| \lambda_n - \frac{\left\| \mathbf{A}\mathbf{z}^{(k-1)} \right\|_2}{\left\| \mathbf{z}^{(k-1)} \right\|_2} \right|$

$\mathbf{z}^{(0)}$ = random vector

Test matrices:

① `d=(1:10)';` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.9$
② `d = [ones(9,1); 2];` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.5$
③ `d = 1-2.^(-(1:0.5:5)');` ➢ $|\lambda_{n-1}| : |\lambda_n| = 0.9866$

MATLAB-code 9.3.20: Investigating convergence of direct power method

```
% Demonstration of direct power method for Ex. 9.3.18
maxit = 30; d = [1:10]'; n = length(d);
% Initialize the matrix A
S = triu(diag(n:-1:1,0)+ones(n,n));
A = S*diag(d,0)*inv(S);
% This calculates the exact eigenvalues (for error calculation)
[V,D] = eig(A);

k = find(d == max(abs(d)));
if (length(k) > 1), error('No single largest EV'); end;
ev = X(:,k(1)); ev = ev/norm(ev); ev
ew = d(k(1)); ew
if (ew < 0), sgn = -1; else sgn = 1; end

z = rand(n,1); z = z/norm(z);
s = 1;
```

```
res = [];

% Actual direct power iteration
for i=1:maxit
  w = A*z; l = norm(w); rq = real(dot(w,z)); z = w/l;
  res = [res;i,l,norm(s*z-ev),abs(l-abs(ew)),abs(sgn*rq-ew)];
  s = s * sgn;
end

% Plot the result
semilogy(res(:,1),res(:,3),'r-*',res(:,1),res(:,4),'k-+',res(:,1),res(:,5),'m-o')
xlabel('{\bf iteration step k}','FontSize',14);
ylabel('{\bf errors}','FontSize',14);
print -deps2c '../PICTURES/pm1.eps';
```

$$\rho_{\mathrm{EV}}^{(k)} := \frac{\left\|\mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n}\right\|}{\left\|\mathbf{z}^{(k-1)} - \mathbf{s}_{\cdot,n}\right\|}\,,$$

$$\rho_{\mathrm{EW}}^{(k)} := \frac{|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_n|}{|\rho_{\mathbf{A}}(\mathbf{z}^{(k-1)}) - \lambda_n|}\,.$$

	①		②		③	
k	$\rho_{\mathrm{EV}}^{(k)}$	$\rho_{\mathrm{EW}}^{(k)}$	$\rho_{\mathrm{EV}}^{(k)}$	$\rho_{\mathrm{EW}}^{(k)}$	$\rho_{\mathrm{EV}}^{(k)}$	$\rho_{\mathrm{EW}}^{(k)}$
22	0.9102	0.9007	0.5000	0.5000	0.9900	0.9781
23	0.9092	0.9004	0.5000	0.5000	0.9900	0.9791
24	0.9083	0.9001	0.5000	0.5000	0.9901	0.9800
25	0.9075	0.9000	0.5000	0.5000	0.9901	0.9809
26	0.9068	0.8998	0.5000	0.5000	0.9901	0.9817
27	0.9061	0.8997	0.5000	0.5000	0.9901	0.9825
28	0.9055	0.8997	0.5000	0.5000	0.9901	0.9832
29	0.9049	0.8996	0.5000	0.5000	0.9901	0.9839
30	0.9045	0.8996	0.5000	0.5000	0.9901	0.9844

Observation: linear convergence ($\rightarrow$ **??**)

Theorem 9.3.21. Convergence of direct power method $\rightarrow$ **[?, Thm. 25.1]**

Let $\lambda_n > 0$ be the largest (in modulus) eigenvalue of $\mathbf{A} \in \mathbb{K}^{n,n}$ and have (algebraic) multiplicity 1. Let $\mathbf{v}, \mathbf{y}$ be the left and right eigenvectors of $\mathbf{A}$ for λ_n normalized according to $\|\mathbf{y}\|_2 = \|\mathbf{v}\|_2 = 1$. Then there is convergence

$$\left\|\mathbf{A}\mathbf{z}^{(k)}\right\|_2 \to \lambda_n \quad , \quad \mathbf{z}^{(k)} \to \pm\mathbf{v} \quad \textit{linearly with rate} \quad \frac{|\lambda_{n-1}|}{|\lambda_n|}\,,$$

where $\mathbf{z}^{(k)}$ are the iterates of the direct power iteration and $\mathbf{y}^H\mathbf{z}^{(0)} \neq 0$ is assumed.

Remark 9.3.22 (Initial guess for power iteration)

roundoff errors ➤ $\mathbf{y}^H\mathbf{z}^{(0)} \neq 0$ always satisfied in practical computations

Usual (not the best!) choice for $\mathbf{x}^{(0)}$ = random vector

Remark 9.3.23 (Termination criterion for direct power iteration)

(→ Section 8.1.2)

Adaptation of a posteriori termination criterion (8.2.23)

"relative change" $\leq$ tol:

$$\begin{cases} \min\left\| \mathbf{z}^{(k)} \pm \mathbf{z}^{(k-1)} \right\| \leq (1/L-1)\texttt{tol}\,, \\ \left| \frac{\|\mathbf{A}\mathbf{z}^{(k)}\|}{\|\mathbf{z}^{(k)}\|} - \frac{\|\mathbf{A}\mathbf{z}^{(k-1)}\|}{\|\mathbf{z}^{(k-1)}\|} \right| \leq (1/L-1)\texttt{tol} \quad \text{see (8.1.29)}\,. \end{cases}$$

Estimated rate of convergence

9.3.2 Inverse Iteration [?, Sect. 7.6], [?, Sect. 5.3.2]

Example 9.3.24 (Image segmentation)

Given: gray-scale image: intensity matrix $\mathbf{P} \in \{0,\ldots,255\}^{m,n}$, $m,n \in \mathbb{N}$
($(\mathbf{P})_{ij} \leftrightarrow$ pixel, $0 \hat{=}$ black, $255 \hat{=}$ white)

Loading and displaying images in MATLAB ▷

MATLAB-code 9.3.25: loading and displaying an image

```
M = imread('eth.pbm');
[m,n] = size(M);
fprintf('%dx%d grey scale pixel
   image\n',m,n);
figure; image(M); title('ETH view');
col = [0:1/215:1]'*[1,1,1]; colormap(col);
```

(Fuzzy) task: Local segmentation

Find connected patches of image of the same shade/color

More general segmentation problem (non-local): identify parts of the image, not necessarily connected, with the same texture.

Next: Statement of (rigorously defined) problem, *cf.* **??**:

Preparation: Numbering of pixels $1\ldots,mn$, e.g, lexicographic numbering:
- ✦ pixel set $\mathcal{V} := \{1\ldots,nm\}$
- ✦ indexing: index(pixel$_{i,j}$) $= (i-1)n+j$

✎ notation: $p_k := (\mathbf{P})_{ij}$, if $k = \text{index}(\text{pixel}_{i,j}) = (i-1)n+j$, $k = 1,\ldots,N := mn$

Local similarity matrix:

$$\mathbf{W} \in \mathbb{R}^{N,N}, \quad N := mn, \qquad (9.3.26)$$

$$(\mathbf{W})_{ij} = \begin{cases} 0 & \text{, if pixels } i, j \text{ not adjacent,} \\ 0 & \text{, if } i = j, \\ \sigma(p_i, p_j) & \text{, if pixels } i, j \text{ adjacent.} \end{cases}$$

$\leftrightarrow \hat{=}$ adjacent pixels ▷

Similarity function, e.g., with $\alpha > 0$

$$\sigma(x,y) := \exp(-\alpha(x-y)^2), \quad x, y \in \mathbb{R}.$$

Lexicographic numbering ▷

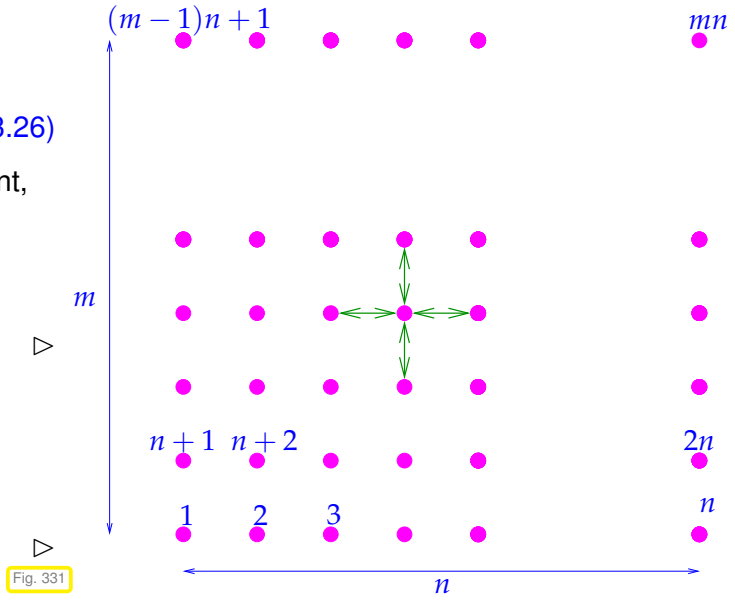


Fig. 331

The entries of the matrix $\mathbf{W}$ measure the "similarity" of neighboring pixels: if $(\mathbf{W})_{ij}$ is large, they encode (almost) the same intensity, if $(\mathbf{W})_{ij}$ is close to zero, then they belong to parts of the picture with very different brightness. In the latter case, the boundary of the segment may separate the two pixels.

Definition 9.3.27. Normalized cut ($\rightarrow$ [?, Sect. 2])

For $\mathcal{X} \subset \mathcal{V}$ define the normalized cut as

$$\mathrm{Ncut}(\mathcal{X}) := \frac{\mathrm{cut}(\mathcal{X})}{\mathrm{weight}(\mathcal{X})} + \frac{\mathrm{cut}(\mathcal{X})}{\mathrm{weight}(\mathcal{V} \setminus \mathcal{X})},$$

$$\text{with} \quad \mathrm{cut}(\mathcal{X}) := \sum_{i \in \mathcal{X}, j \notin \mathcal{X}} w_{ij}, \quad \mathrm{weight}(\mathcal{X}) := \sum_{i \in \mathcal{X}, j \in \mathcal{X}} w_{ij}.$$

In light of local similarity relationship:

- $\mathrm{cut}(\mathcal{X})$ big ➢ substantial similarity of pixels across interface between $\mathcal{X}$ and $\mathcal{V} \setminus \mathcal{X}$.
- $\mathrm{weight}(\mathcal{X})$ big ➢ a lot of similarity of adjacent pixels inside $\mathcal{X}$.

➢ Segmentation problem (rigorous statement):

$$\text{find} \quad \mathcal{X}^* \subset \mathcal{V}: \quad \mathcal{X}^* = \operatorname*{argmin}_{\mathcal{X} \subset \mathcal{V}} \mathrm{Ncut}(\mathcal{X}). \qquad (9.3.28)$$

NP-hard combinatorial optimization problem !

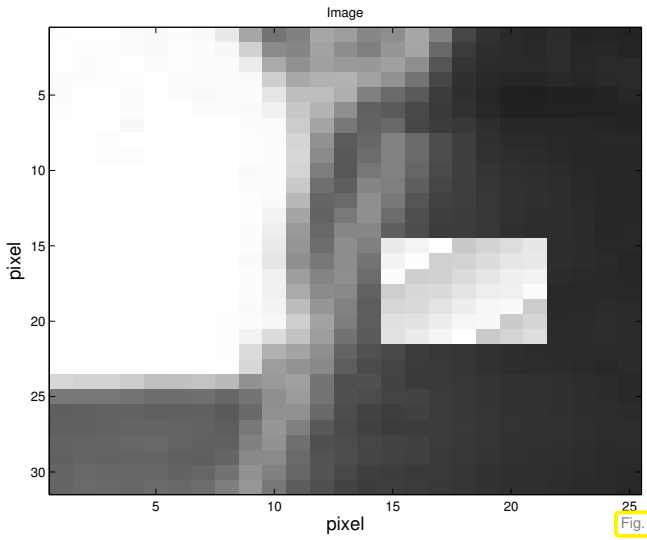


Fig. 332

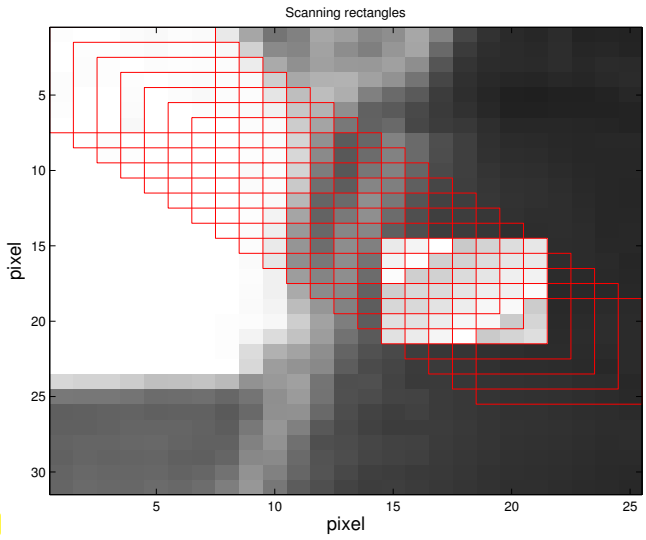


Fig. 333

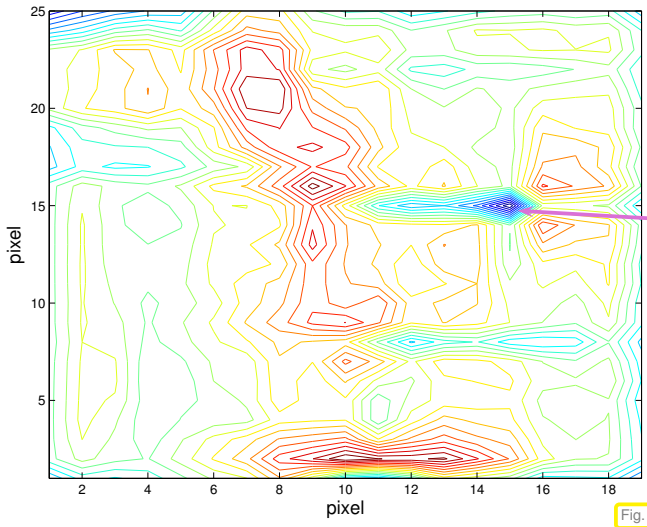


Fig. 334

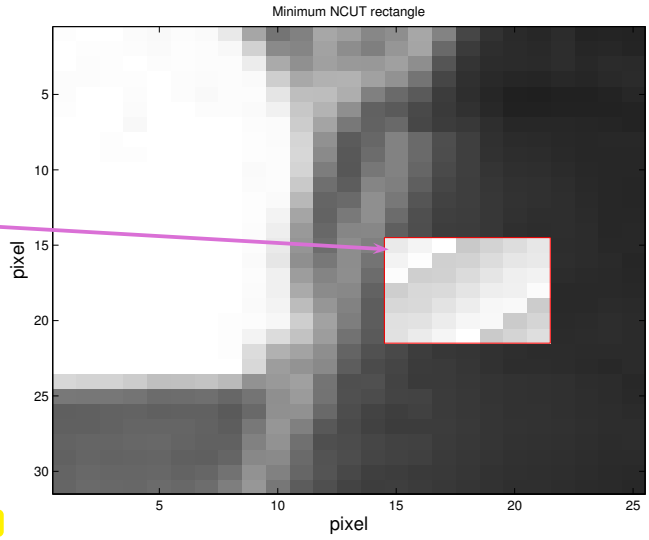


Fig. 335

△ $\mathrm{Ncut}(\mathcal{X})$ for pixel subsets $\mathcal{X}$ defined by sliding rectangles, see Fig. 333.

Equivalent reformulation:

$$\text{indicator function:} \quad z : \{1,\ldots,N\} \mapsto \{-1,1\}\,, \quad z_i := z(i) = \begin{cases} 1 & \text{, if } i \in \mathcal{X}\,, \\ -1 & \text{, if } i \notin \mathcal{X}\,. \end{cases} \tag{9.3.29}$$

$$\mathrm{Ncut}(\mathcal{X}) = \frac{\sum\limits_{z_i>0, z_j<0} -w_{ij} z_i z_j}{\sum\limits_{z_i>0} d_i} + \frac{\sum\limits_{z_i>0, z_j<0} -w_{ij} z_i z_j}{\sum\limits_{z_i<0} d_i}\,, \tag{9.3.30}$$

$$d_i = \sum_{j\in\mathcal{V}} w_{ij} = \mathrm{weight}(\{i\})\,. \tag{9.3.31}$$

Sparse matrices:

$$\mathbf{D} := \mathrm{diag}(d_1,\ldots,d_N) \in \mathbb{R}^{N,N}\,, \quad \mathbf{A} := \mathbf{D} - \mathbf{W} = \mathbf{A}^\top\,. \tag{9.3.32}$$

Summary: (obvious) properties of these matrices

- $\mathbf{A}$ has positive diagonal and non-positive off-diagonal entries.

- $\mathbf{A}$ is diagonally dominant ($\rightarrow$ Def. 2.8.8) $\succ$ $\mathbf{A}$ is positive semidefinite by Lemma 2.8.12.
- $\mathbf{A}$ has row sums $= 0$:

$$\mathbf{1}^\top \mathbf{A} = \mathbf{A}\mathbf{1} = 0\,. \tag{9.3.33}$$

MATLAB-code 9.3.34: assembly of $\mathbf{A}$, $\mathbf{D}$

```
function [A,D] = imgsegmat(P)
P = double(P); [n,m] = size(P);
spdata = zeros(4*n*m,1); spidx  = zeros(4*n*m,2);
k = 1;
for ni=1:n
  for mi=1:m
    mni = (mi-1)*n+ni;
    if (ni-1>0), spidx(k,:) = [mni,mni-1];
      spdata(k) = Sfun(P(ni,mi),P(ni-1,mi));
      k = k + 1;
    end
    if (ni+1<=n), spidx(k,:) = [mni,mni+1];
      spdata(k) = Sfun(P(ni,mi),P(ni+1,mi));
      k = k + 1;
    end
    if (mi-1>0), spidx(k,:) = [mni,mni-n];
      spdata(k) = Sfun(P(ni,mi),P(ni,mi-1));
      k = k + 1;
    end
    if (mi+1<=m), spidx(k,:) = [mni,mni+n];
      spdata(k) = Sfun(P(ni,mi),P(ni,mi+1));
      k = k + 1;
    end
  end
end
% Efficient initialization, see Sect. 2.7.2, Ex. 2.7.13
W = sparse(spidx(1:k-1,1),spidx(1:k-1,2),spdata(1:k-1),n*m,n*m);
D = spdiags(full(sum(W')'),[0],n*m,n*m);
A = D-W;
```

Lemma 9.3.35. Ncut and Rayleigh quotient ($\rightarrow$ [?, Sect. 2])

With $\mathbf{z} \in \{-1,1\}^N$ according to (9.3.29) *there holds*

$$\mathrm{Ncut}(\mathcal{X}) = \frac{\mathbf{y}^\top \mathbf{A}\mathbf{y}}{\mathbf{y}^\top \mathbf{D}\mathbf{y}} \quad , \quad \mathbf{y} := (\mathbf{1}+\mathbf{z}) - \beta(\mathbf{1}-\mathbf{z}) \, , \quad \beta := \frac{\sum\limits_{z_i>0} d_i}{\sum\limits_{z_i<0} d_i} \, .$$

generalized Rayleigh quotient $\rho_{\mathbf{A},\mathbf{D}}(\mathbf{y})$

Proof. Note that by (9.3.29) $(\mathbf{1}-\mathbf{z})_i = 0 \;\Leftrightarrow\; i \in \mathcal{X}$, $(\mathbf{1}+\mathbf{z})_i = 0 \;\Leftrightarrow\; i \notin \mathcal{X}$. Hence, since

$(\mathbf{1}+\mathbf{z})^\top \mathbf{D}(\mathbf{1}-\mathbf{z}) = 0$,

$$4\,\mathrm{Ncut}(\mathcal{X}) = (\mathbf{1}+\mathbf{z})^\top \mathbf{A}(\mathbf{1}+\mathbf{z})\left(\frac{1}{\kappa \mathbf{1}^\top \mathbf{D}\mathbf{1}} + \frac{1}{(1-\kappa)\mathbf{1}^\top \mathbf{D}\mathbf{1}}\right) = \frac{\mathbf{y}^\top \mathbf{A}\mathbf{y}}{\beta \mathbf{1}^\top \mathbf{D}\mathbf{1}}\,,$$

where $\kappa := \sum_{z_i>0} d_i / \sum_i d_i = \frac{\beta}{1+\beta}$. Also observe

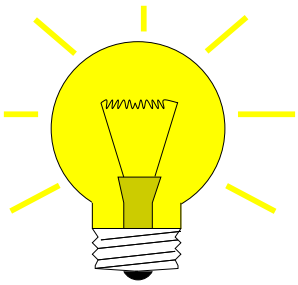
$$\mathbf{y}^\top \mathbf{D}\mathbf{y} = (\mathbf{1}+\mathbf{z})^\top \mathbf{D}(\mathbf{1}+\mathbf{z}) + \beta^2(\mathbf{1}-\mathbf{z})^\top \mathbf{D}(\mathbf{1}-\mathbf{z}) = 4(\sum_{z_i>0} d_i + \beta^2 \sum_{z_i<0} d_i) = 4\beta \sum_i d_i = 4\beta \mathbf{1}^\top \mathbf{D}\mathbf{1}\,.$$

This finishes the proof. □

- ✦ (9.3.33) $\Rightarrow$ $\mathbf{1} \in \mathbf{EigA}0$
- ✦ Lemma 2.8.12: $\mathbf{A}$ diagonally dominant $\Longrightarrow$ $\mathbf{A}$ is *positive semidefinite* (→ Def. 1.1.8)

➤ $\mathrm{Ncut}(\mathcal{X}) \geq 0$ and 0 is the smallest eigenvalue of $\mathbf{A}$.

However, we are by no means interested in a minimizer $\mathbf{y} \in \mathrm{Span}\{\mathbf{1}\}$ (with constant entries) that does not provide a meaningful segmentation.



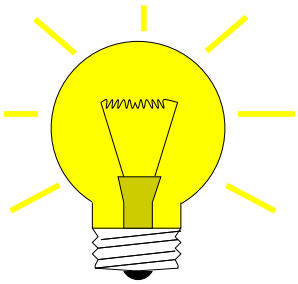
Idea: weed out undesirable constant minimizers by imposing orthogonality constraint (orthogonality w.r.t. inner product induced by $\mathbf{D}$, *cf.* Section 10.1)

$$\mathbf{y} \perp \mathbf{D}\mathbf{1} \quad \Leftrightarrow \quad \mathbf{1}^\top \mathbf{D}\mathbf{y} = 0\,. \tag{9.3.36}$$

➤ segmentation problem (9.3.28) $\Leftrightarrow$ $\operatorname{argmin}_{\mathbf{y}\in\{2,-2\beta\}^N,\, \mathbf{1}^\top\mathbf{D}\mathbf{y}=0} \rho_{\mathbf{A},\mathbf{D}}(\mathbf{y})$. (9.3.37)

still NP-hard

➢ Minimizing $\mathrm{Ncut}(\mathcal{X})$ amounts to minimizing a (generalized) Rayleigh quotient (→ Def. 9.3.16) over a *discrete* set of vectors, which is still an NP-hard problem.



Idea: Relaxation

Discrete optimization problem → continuous optimization problem

(9.3.37) → $\operatorname{argmin}_{\mathbf{y}\in\mathbb{R}^N,\, \mathbf{y}\neq 0,\, \mathbf{1}^\top\mathbf{D}\mathbf{y}=0} \rho_{\mathbf{A},\mathbf{D}}(\mathbf{y})$. (9.3.38)

Task: (9.3.38) $\Leftrightarrow$ Find minimizer of (generalized) Rayleigh quotient under linear constraint

Here: linear constraint on $\mathbf{y}$: $\mathbf{1}^\top \mathbf{D}\mathbf{y} = 0$

The next theorem establishes a link between argument vectors that render the Rayleigh quotient extremal and eigenspace for extremal eigenvalues.

Theorem 9.3.39. Rayleigh quotient theorem

Let $\lambda_1 < \lambda_2 < \cdots < \lambda_m$, $m \le n$, be the sorted sequence of all (real!) eigenvalues of $\mathbf{A} = \mathbf{A}^{\mathrm{H}} \in \mathbb{C}^{n,n}$. Then

$$\mathbf{EigA}\lambda_1 = \operatorname*{argmin}_{\mathbf{y}\in\mathbb{C}^{n.n}\setminus\{0\}} \rho_{\mathbf{A}}(\mathbf{y}) \quad \textit{and} \quad \mathbf{EigA}\lambda_m = \operatorname*{argmax}_{\mathbf{y}\in\mathbb{C}^{n.n}\setminus\{0\}} \rho_{\mathbf{A}}(\mathbf{y}) \,.$$

Remark 9.3.40 (Min-max theorem)

Thm. 9.3.39 is a an immediate consequence of the following more general and fundamentally important result.

Theorem 9.3.41. Courant-Fischer min-max theorem → [?, Thm. 8.1.2]

Let $\lambda_1 < \lambda_2 < \cdots < \lambda_m$, $m \le n$, be the sorted sequence of the (real!) eigenvalues of $\mathbf{A} = \mathbf{A}^{\mathrm{H}} \in \mathbb{C}^{n,n}$. Write

$$U_0 = \{0\}\,, \quad U_\ell := \sum_{j=1}^{\ell} \mathbf{EigA}\lambda_j\,, \ell = 1,\ldots,m \quad \textit{and} \quad U_\ell^\perp := \{\mathbf{x}\in\mathbb{C}^n \colon \mathbf{u}^{\mathrm{H}}\mathbf{x} = 0\ \forall \mathbf{u}\in U_\ell\}\,.$$

Then

$$\min_{\mathbf{y}\in U_{\ell-1}^\perp\setminus\{0\}} \rho_{\mathbf{A}}(\mathbf{y}) = \lambda_\ell\,, \quad 1\le\ell\le m\,, \quad \operatorname*{argmin}_{\mathbf{y}\in U_{\ell-1}^\perp\setminus\{0\}} \rho_{\mathbf{A}}(\mathbf{y}) \subset \mathbf{EigA}\lambda_\ell\,.$$

Proof. For diagonal $\mathbf{A}\in\mathbb{R}^{n,n}$ the assertion of the theorem is obvious. Thus, Cor. 9.1.9 settles everything. □

A simple conclusion from Thm. 9.3.41: If $\mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n}$ with eigenvalues $\lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$, then

$$\lambda_1 = \min_{\mathbf{z}\in\mathbb{R}^n} \rho_{\mathbf{A}}(\mathbf{z}) \quad , \quad \lambda_2 = \min_{\mathbf{z}\in\mathbb{R}^n, \mathbf{z}\perp\mathbf{v}_1} \rho_{\mathbf{A}}(\mathbf{z})\,, \tag{9.3.42}$$

where $\mathbf{v}_1 \in \mathbf{EigA}\lambda_1 \setminus \{0\}$.

Well, in Lemma 9.3.35 we encounter a generalized Rayleigh quotient $\rho_{\mathbf{A},\mathbf{D}}(\mathbf{y})$! How can Thm. 9.3.39 be applied to it?

▬► Transformation idea: $\rho_{\mathbf{A},\mathbf{D}}(\mathbf{D}^{-1/2}\mathbf{z}) = \rho_{\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}}(\mathbf{z})\,, \quad \mathbf{y}\in\mathbb{R}^n\,.$ (9.3.43)

▬► Apply Thm. 9.3.41 to transformed matrix $\widetilde{\mathbf{A}} := \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$. *Elementary* manipulations show

$$\begin{aligned} \text{(9.3.38)} \quad \Leftrightarrow \quad \operatorname*{argmin}_{\mathbf{1}^\top\mathbf{D}\mathbf{y}=0} \rho_{\mathbf{A},\mathbf{D}}(\mathbf{y}) &\overset{\mathbf{z}=\mathbf{D}^{1/2}\mathbf{y}}{=} \operatorname*{argmin}_{\mathbf{1}^\top\mathbf{D}^{1/2}\mathbf{z}=0} \rho_{\mathbf{A},\mathbf{D}}(\mathbf{D}^{-1/2}\mathbf{z}) \\ &= \operatorname*{argmin}_{\mathbf{1}^\top\mathbf{D}^{1/2}\mathbf{z}=0} \rho_{\widetilde{\mathbf{A}}}(\mathbf{z}) \quad \text{with} \quad \widetilde{\mathbf{A}} := \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}\,. \end{aligned} \tag{9.3.44}$$

Related: transformation of a generalized eigenvalue problem into a standard eigenvalue problem according to

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x} \quad \overset{\mathbf{z}=\mathbf{B}^{1/2}\mathbf{x}}{\Longrightarrow} \quad \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\mathbf{z} = \lambda\mathbf{z}\ . \tag{9.3.45}$$

$\mathbf{B}^{1/2} \hat{=}$ square root of s.p.d. matrix $\mathbf{B}$ $\rightarrow$ Rem. 10.3.2.

For segmentation problem: $\mathbf{B} = \mathbf{D}$ diagonal with positive diagonal entries, see (9.3.32)

➥ $\mathbf{D}^{-1/2} = \operatorname{diag}(d_1^{-1/2}, \ldots, d_N^{-1/2})$ and $\widetilde{\mathbf{A}} := \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ can easily be computed.

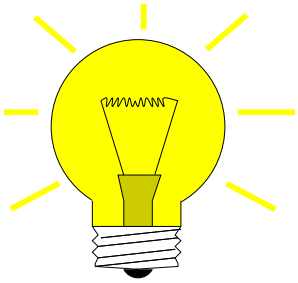
► In the sequel consider minimization problem/related eigenvalue problem

$$\boxed{\mathbf{z}^* = \operatorname*{argmin}_{\mathbf{1}^\top\mathbf{D}^{1/2}\mathbf{z}=0} \rho_{\widetilde{\mathbf{A}}}(\mathbf{z})} \quad \longleftrightarrow \quad \widetilde{\mathbf{A}}\mathbf{z} = \lambda\mathbf{z}\ . \tag{9.3.46}$$

Recover solution $\mathbf{y}^*$ of (9.3.38) as $\mathbf{y}^* = \mathbf{D}^{-1/2}\mathbf{z}^*$.

Still, Thm. 9.3.39 cannot be applied to (9.3.46):

How to deal with *constraint* $\mathbf{1}^\top\mathbf{D}^{1/2}\mathbf{z} = 0$?



Idea: Penalization

Add term $P(\mathbf{z})$ to $\rho_{\widetilde{\mathbf{A}}}(\mathbf{z})$ that becomes "sufficiently large" in case the constraint is violated.

► $\mathbf{z}^*$ can only be a minimizer of $\rho_{\widetilde{\mathbf{A}}}(\mathbf{z}) + P(\mathbf{z})$, if $P(\mathbf{z}) = 0$.

How to choose the penalty function $P(\mathbf{z})$ for the segmentation problem ?

$$\left\{ \mathbf{1}^\top\mathbf{D}^{1/2}\mathbf{z} \neq 0 \quad \Rightarrow \quad P(\mathbf{z}) > 0 \right\} \text{ satisfied for } \boxed{P(\mathbf{z}) = \mu\frac{|\mathbf{1}^\top\mathbf{D}^{1/2}\mathbf{z}|^2}{\|\mathbf{z}\|_2^2}}\ ,$$

with penalty parameter $\mu > 0$.

► penalized minimization problem

dense rank-1 matrix

$$\begin{aligned}\mathbf{z}^* &= \operatorname*{argmin}_{\mathbf{z}\in\mathbb{R}^N\setminus\{0\}} \rho_{\widetilde{\mathbf{A}}}(\mathbf{z}) + P(\mathbf{z}) = \operatorname*{argmin}_{\mathbf{z}\in\mathbb{R}^N\setminus\{0\}} \rho_{\widetilde{\mathbf{A}}}(\mathbf{z}) + \frac{\mathbf{z}^\top(\mathbf{D}^{1/2}\mathbf{1}\mathbf{1}^\top\mathbf{D}^{1/2})\mathbf{z}}{\mathbf{z}^\top\mathbf{z}} \\ &= \operatorname*{argmin}_{\mathbf{z}\in\mathbb{R}^N\setminus\{0\}} \rho_{\widehat{A}}(\mathbf{z}) \quad \text{with} \quad \widehat{\mathbf{A}} := \widetilde{\mathbf{A}} + \mathbf{D}^{1/2}\mathbf{1}\mathbf{1}^\top\mathbf{D}^{1/2}\ .\end{aligned} \tag{9.3.47}$$

How to choose the penalty parameter μ ?

In general: finding a "suitable" value for μ may be difficult or even impossible!

Here we are lucky:

$$\text{(9.3.33)} \quad \Rightarrow \quad \mathbf{A}\mathbf{1} = 0 \quad \Rightarrow \quad \widetilde{\mathbf{A}}(\mathbf{D}^{1/2}\mathbf{1}) = 0 \quad \Leftrightarrow \quad \mathbf{D}^{1/2}\mathbf{1} \in \mathbf{Eig}\widetilde{\mathbf{A}}0\,.$$

Constraint in (9.3.46) means

Minimize over the orthogonal complement of an *eigenvector*. (9.3.48)

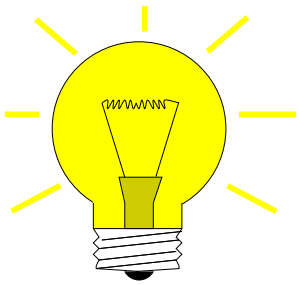
Cor. 9.1.9 ➤ The orthogonal complement of an eigenvector of a *symmetric* matrix is spanned by the other eigenvectors (orthonormalization of eigenvectors belonging to the same eigenvalue is assumed).

(9.3.48) The minimizer of (9.3.46) will be one of the other eigenvectors of $\widetilde{\mathbf{A}}$ that belongs to the smallest eigenvalue.

Note: This eigenvector $\mathbf{z}^*$ will be orthogonal to $\mathbf{D}^{1/2}\mathbf{1}$, it satisfies the constraint, and, thus, $P(\mathbf{z}^*) = 0$!

Note: eigenspaces of $\widetilde{\mathbf{A}}$ and $\widehat{\mathbf{A}}$ agree.

Note: Lemma 2.8.12 $\Longrightarrow$ $\widetilde{\mathbf{A}}$ is *positive semidefinite* ($\rightarrow$ Def. 1.1.8) with smallest eigenvalue 0.n



Idea: Choose penalization parameter μ in (9.3.47) such that $\mathbf{D}^{1/2}\mathbf{1}$ is guaranteed not to be an eigenvector belonging to the smallest eigenvalue of $\widehat{\mathbf{A}}$.

Safe choice: choose μ such that $\mathbf{D}^{1/2}\mathbf{1}$ will belong to the largest eigenvalue of $\widehat{\mathbf{A}}$: Thm. 9.1.4 suggests

$$\mu = \left\|\widetilde{\mathbf{A}}\right\|_\infty \overset{(1.5.79)}{=} 2\,. \tag{9.3.49}$$

$$\mathbf{z}^* = \operatorname*{argmin}_{\mathbf{1}^\top\mathbf{D}^{1/2}\mathbf{z}=0} \rho_{\widetilde{\mathbf{A}}}(\mathbf{z}) = \operatorname*{argmin}_{\mathbf{z}\neq 0} \rho_{\widehat{\mathbf{A}}}(\mathbf{z})\,. \tag{9.3.50}$$

By Thm. 9.3.39:

$$\begin{array}{c} \mathbf{z}^* = \text{eigenvector belonging to minimal eigenvalue of } \widehat{\mathbf{A}}, \\ \Updownarrow \\ \mathbf{z}^* = \text{eigenvector} \perp \mathbf{D}^{1/2}\mathbf{1} \text{ belonging to minimal eigenvalue of } \widetilde{\mathbf{A}}, \\ \Updownarrow \\ \mathbf{D}^{-1/2}\mathbf{z}^* = \text{minimizer for (9.3.38)}. \end{array}$$

(9.3.51) Algorithm outline: Binary grayscale image segmentation

❶ Given similarity function σ compute (sparse!) matrices $\mathbf{W}, \mathbf{D}, \mathbf{A} \in \mathbb{R}^{N,N}$, see (9.3.26), (9.3.32).

❷ Compute $\mathbf{y}^*$, $\|\mathbf{y}^*\|_2 = 1$, as eigenvector belonging to the *smallest eigenvalue* of $\widehat{\mathbf{A}} := \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} + 2(\mathbf{D}^{1/2}\mathbf{1})(\mathbf{D}^{1/2}\mathbf{1})^\top$.

❸ Set $\mathbf{x}^* = \mathbf{D}^{-1/2}\mathbf{y}^*$ and define the image segment as pixel set

$$\mathcal{X} := \{i \in \{1,\ldots,N\}\colon x_i^* > \tfrac{1}{N}\sum_{i=1}^N x_i^*\}\,. \tag{9.3.52}$$

mean value of entries of $\mathbf{x}^*$

MATLAB-code 9.3.53: 1st stage of segmentation of grayscale image

```
% Read image and build matrices, see Code 9.3.34 and (9.3.32)
P = imread('image.pbm'); [m,n] = size(P); [A,D] = imgsegmat(P);
% Build scaling matrics
N = size(A,1); dv = sqrt(spdiags(A,0));
Dm = spdiags(1./dv,[0],N,N); % D^{-1/2}
Dp = spdiags(dv,[0],N,N);    % D^{-1/2}
% Build (densely populated !) matrix Â
c = Dp*ones(N,1); Ah = Dm*A*Dm + 2*c*c';
% Compute and sort eigenvalues; grossly inefficient !
[W,E] = eig(full(Ah)); [ev,idx] = sort(diag(E)); W(:,idx) = W;
% Obtain eigenvector x* belonging to 2nd smallest generalized
% eigenvalue of A and D
x = W(:,1); x = Dm*v;
% Extract segmented image
xs = reshape(x,m,n); Xidx = find(xs>(sum(sum(xs))/(n*m)));
```

1st-stage of segmentation of 31×25 grayscale pixel image (`root.pbm`, red pixels $\hat{=} \mathcal{X}$, $\sigma(x,y) = \exp(-(x-y/10)^2)$)

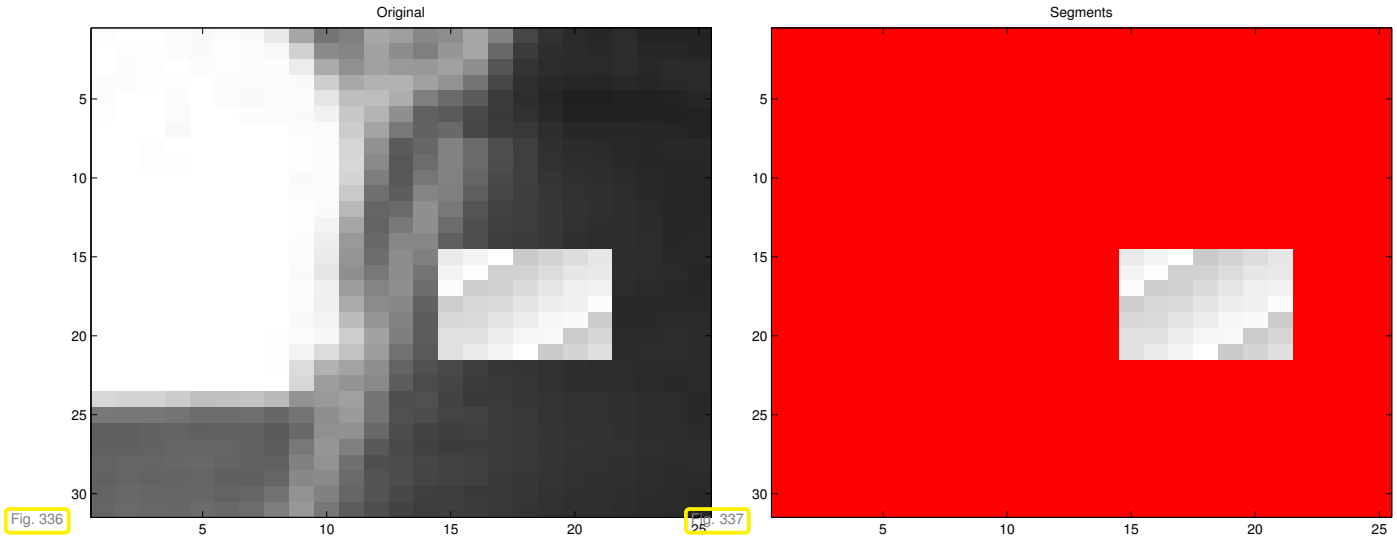


Fig. 336 Fig. 337

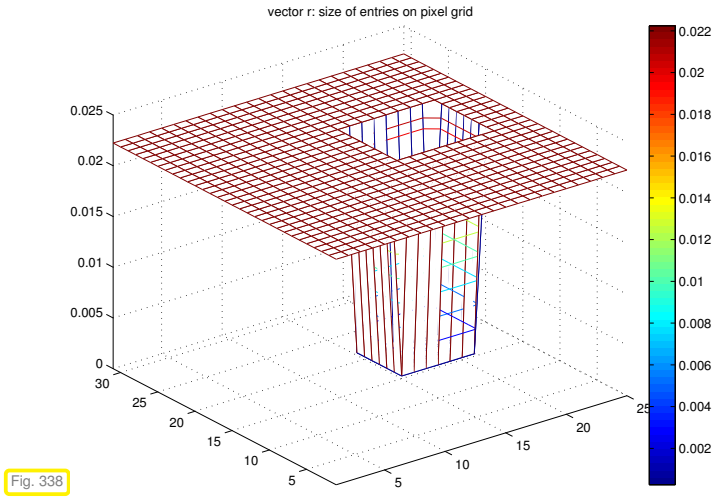


Fig. 338

Image from Fig. 336:

◁ eigenvector $\mathbf{x}^*$ plotted on pixel grid

To identify more segments, the same algorithm is *recursively applied* to segment parts of the image

already determined.

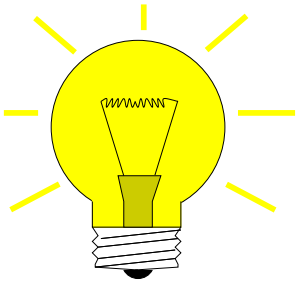
Practical segmentation algorithms rely on many more steps of which the above algorithm is only one, preceeded by substantial preprocessing. Moreover, they dispense with the strictly local perspective adopted above and take into account more distant connections between image parts, often in a randomized fashion [**?**].

The image segmentation problem falls into the wider class of graph partitioning problems. Methods based on (a few of) the eigenvector of the connectivity matrix belonging to the smallest eigenvalues are known as spectral partitioning methods. The eigenvector belonging to the smallest non-zero eigenvalue that we computed above is usually called the Fiedler vector of the graph, see [**?**, **?**].

The solution of the image segmentation problem by means of `eig` in Code 9.3.53 amounts a tremendous waste of computational resources: we compute *all* eigenvalues/eigenvectors of dense matrices, though only a single eigenvector associated with the smallest eigenvalue is of interest.

This motivates the quest to find *efficient* numerical methods for the following task.

Task:	given $\mathbf{A} \in \mathbb{K}^{n,n}$, find smallest (in modulus) eigenvalue of regular $\mathbf{A} \in \mathbb{K}^{n,n}$ and (an) associated eigenvector.



If $\mathbf{A} \in \mathbb{K}^{n,n}$ regular:

Smallest (in modulus) EV of $\mathbf{A}$ = $\left(\text{Largest (in modulus) EV of } \mathbf{A}^{-1}\right)^{-1}$

Direct power method ($\rightarrow$ Section 9.3.1) for $\mathbf{A}^{-1}$ = inverse iteration

MATLAB-code 9.3.54: inverse iteration for computing $\lambda_{\min}(\mathbf{A})$ and associated eigenvector

```
function [lmin,y] = invit(A,tol)
[L,U] = lu(A); % single intial LU-factorization, see Rem. 2.5.10
n = size(A,1); x = rand(n,1); x = x/norm(x); % random initial guess
y = U\(L\x); lmin = 1/norm(y); y = y*lmin; lold = 0;
while(abs(lmin-lold) > tol*lmin) % termination, if small relative
    change
  lold = lmin; x = y;
  y = U\(L\x); % core iteration: y = A^{-1}x,
  lmin = 1/norm(y); % new approxmation of λ_min(A)
  y = y*lmin;  % normalization y := y/‖y‖_2
end
```

Note: reuse of LU-factorization, see Rem. 2.5.10

Remark 9.3.55 (Shifted inverse iteration)

More general task:

$$\text{For } \alpha \in \mathbb{C} \text{ find } \lambda \in \sigma(\mathbf{A}) \text{ such that } |\alpha - \lambda| = \min\{|\alpha - \mu|, \mu \in \sigma(\mathbf{A})\}$$

➤ Shifted inverse iteration: [?, Alg .7.24]

$$\mathbf{z}^{(0)} \text{ arbitrary}, \quad \mathbf{w} = (\mathbf{A} - \alpha\mathbf{I})^{-1}\mathbf{z}^{(k-1)} \quad , \quad \mathbf{z}^{(k)} := \frac{\mathbf{w}}{\|\mathbf{w}\|_2}, \quad k = 1, 2, \ldots , \tag{9.3.56}$$

where: $(\mathbf{A} - \alpha\mathbf{I})^{-1}\mathbf{z}^{(k-1)} \triangleq$ solve $(\mathbf{A} - \alpha\mathbf{I})\mathbf{w} = \mathbf{z}^{(k-1)}$ based on Gaussian elimination ($\leftrightarrow$ a single LU-factorization of $\mathbf{A} - \alpha\mathbf{I}$ as in Code 9.3.54).

Remark 9.3.57 ((Nearly) singular LSE in shifted inverse iteration)

What if "by accident" $\alpha \in \sigma(\mathbf{A})$ ($\Leftrightarrow \mathbf{A} - \alpha\mathbf{I}$ singular) **?**

Stability of Gaussian elimination/LU-factorization ($\rightarrow$ **??**) will ensure that "$\mathbf{w}$ from (9.3.56) points in the right direction"

In other words, roundoff errors may badly affect the length of the solution $\mathbf{w}$, but not its direction.

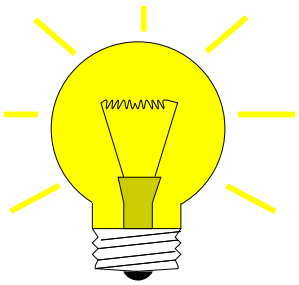
Practice [**?**]: If, in the course of Gaussian elimination/LU-factorization a zero pivot element is really encountered, then we just *replace it with* `eps`, in order to avoid `inf` values!

Thm. 9.3.21 ➢ Convergence of shifted inverse iteration for $\mathbf{A}^H = \mathbf{A}$:

Asymptotic linear convergence, Rayleigh quotient $\rightarrow \lambda_j$ with rate

$$\frac{|\lambda_j - \alpha|}{\min\{|\lambda_i - \alpha|, i \neq j\}} \quad \text{with} \quad \lambda_j \in \sigma(\mathbf{A}), \quad |\alpha - \lambda_j| \leq |\alpha - \lambda| \quad \forall \lambda \in \sigma(\mathbf{A}) .$$

➤ Extremely fast for $\alpha \approx \lambda_j$ **!**



Idea: A posteriori adaptation of shift

Use $\alpha := \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})$ in k-th step of inverse iteration.

(9.3.58) Rayleigh quotient iteration $\rightarrow$ **[?, Alg. 25.2]**

MATLAB-code 9.3.59: Rayleigh quotient iteration (for *normal* $\mathbf{A} \in \mathbb{R}^{n,n}$)

```
function [z,lmin] = rqui(A,tol,maxit)
alpha = 0; n = size(A,1);
z = rand(size(A,1),1); z = z/norm(z); % z^(0)
for i=1:maxit
```

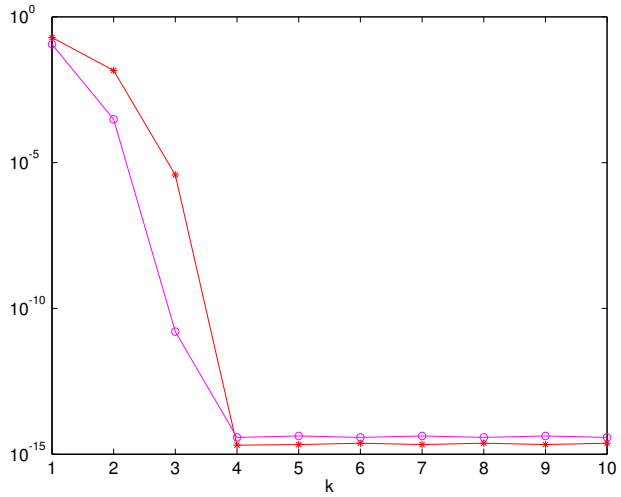
```
5    z = (A-alpha*speye(n))\z; % z^(k+1) = (A - ρ_A(z^(k))I)^(-1) x^(k)
6    z = z/norm(z); lmin=dot(A*z,z); % Computation of ρ_A(z^(k+1))
7    if (abs(alpha-lmin) < tol*lmin) % Desired relative accuracy
       reached ?
8      break; end;
9    alpha = lmin;
10 end
```

Line 5: note use of `speye` to preserve spare matrix data format!

✦ Drawback compared with Code 9.3.54: reuse of LU-factorization no longer possible.
✦ Even if LSE nearly singular, stability of Gaussian elimination guarantees correct direction of $\mathbf{z}$, see discussion in Rem. 9.3.57.

Example 9.3.60 (Rayleigh quotient iteration)

Monitored: iterates of Rayleigh quotient iteration (9.3.59) for s.p.d. $\mathbf{A} \in \mathbb{R}^{n,n}$



MATLAB-code 9.3.61:

```
d = (1:10)';
n = length(d);
Z = diag(sqrt(1:n),0)+ones(n,n);
[Q,R] = qr(Z);
A = Q*diag(d,0)*Q';
```

o : $|\lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$
* : $\left\|\mathbf{z}^{(k)} - \mathbf{x}_j\right\|$, $\lambda_{\min} = \lambda_j$, $\mathbf{x}_j \in \mathbf{EigA}\lambda_j$,
: $\left\|\mathbf{x}_j\right\|_2 = 1$

k	$\lvert\lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})\rvert$	$\left\|\mathbf{z}^{(k)} - \mathbf{x}_j\right\|$
1	0.09381702342056	0.20748822490698
2	0.00029035607981	0.01530829569530
3	0.00000000001783	0.00000411928759
4	0.00000000000000	0.00000000000000
5	0.00000000000000	0.00000000000000

Theorem 9.3.62. → **[?, Thm. 25.4]**

If $\mathbf{A} = \mathbf{A}^H$, then $\rho_{\mathbf{A}}(\mathbf{z}^{(k)})$ converges locally of order 3 *(→ Def. 8.1.17) to the smallest eigenvalue (in modulus), when $\mathbf{z}^{(k)}$ are generated by the Rayleigh quotient iteration* (9.3.59).

9.3.3 Preconditioned inverse iteration (PINVIT)

Task:	given $\mathbf{A} \in \mathbb{K}^{n,n}$, find smallest (in modulus) eigenvalue of regular $\mathbf{A} \in \mathbb{K}^{n,n}$ and (an) associated eigenvector.

Options: inverse iteration (→ Code 9.3.54) and Rayleigh quotient iteration (9.3.59).

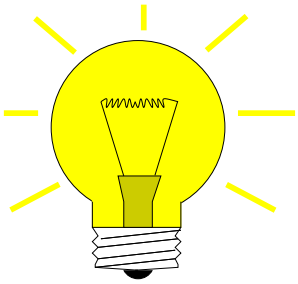
? What if direct solution of $\mathbf{Ax} = \mathbf{b}$ not feasible ?

This can happen, in case

- for large sparse $\mathbf{A}$ the amount of fill-in exhausts memory, despite sparse elimination techniques (→ Section 2.7.5),
- $\mathbf{A}$ is available only through a routine `evalA(x)` providing $\mathbf{A}\times$vector.

We expect that an approximate solution of the linear systems of equations encountered during inverse iteration should be sufficient, because we are dealing with approximate eigenvectors anyway.

Thus, iterative solvers for solving $\mathbf{Aw} = \mathbf{z}^{(k-1)}$ may be considered, see Chapter 10. However, the required accuracy is not clear a priori. Here we examine an approach that completely dispenses with an iterative solver and uses a *preconditioner* (→ Notion 10.3.3) instead.

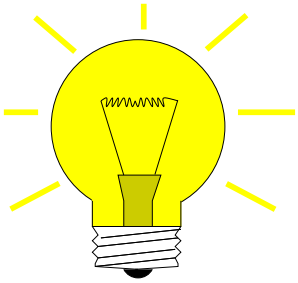


Idea: (for inverse iteration without shift, $\mathbf{A} = \mathbf{A}^H$ s.p.d.)
Instead of solving $\mathbf{Aw} = \mathbf{z}^{(k-1)}$ compute $\mathbf{w} = \mathbf{B}^{-1}\mathbf{z}^{(k-1)}$ with "inexpensive" s.p.d. approximate inverse $\mathbf{B}^{-1} \approx \mathbf{A}^{-1}$

➢ $\mathbf{B} \hat{=}$ Preconditioner for $\mathbf{A}$, see Notion 10.3.3

Possible to replace $\mathbf{A}^{-1}$ with $\mathbf{B}^{-1}$ in inverse iteration **?**

NO, because we are not interested in smallest eigenvalue of $\mathbf{B}$!



Replacement $\mathbf{A}^{-1} \to \mathbf{B}^{-1}$ possible only when applied to residual quantity

residual quantity = quantity that $\to 0$ in the case of convergence to exact solution

Natural residual quantity for eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{x}$:

$$\mathbf{r} := \mathbf{Az} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z} \quad , \quad \rho_{\mathbf{A}}(\mathbf{z}) = \text{Rayleigh quotient} \to \text{Def. 9.3.16 .}$$

Note: only *direction* of $\mathbf{A}^{-1}\mathbf{z}$ matters in inverse iteration (9.3.56)

$$(\mathbf{A}^{-1}\mathbf{z}) \parallel (\mathbf{z} - \mathbf{A}^{-1}(\mathbf{Az} - \rho_{\mathbf{A}}(\mathbf{z})\mathbf{z})) \quad \Rightarrow \quad \text{defines same next iterate!}$$

[Preconditioned inverse iteration (PINVIT) for s.p.d. $\mathbf{A}$]

$\mathbf{z}^{(0)}$ arbitrary,

$$\begin{aligned} \mathbf{w} &= \mathbf{z}^{(k-1)} - \mathbf{B}^{-1}(\mathbf{Az}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}) \,, \\ \mathbf{z}^{(k)} &= \frac{\mathbf{w}}{\|\mathbf{w}\|_2} \,, \end{aligned} \qquad k = 1, 2, \ldots \,. \tag{9.3.63}$$

MATLAB-code 9.3.64: preconditioned inverse iteration (9.3.63)

```
function [lmin,z,res] =
   pinvit(evalA,n,invB,tol,maxit)
% invB ≙ handle to function implementing
   preconditioner B^{-1}
z = (1:n)'; z = z/norm(z); % initial guess
res = []; rho = 0;
for i=1:maxit
  v = evalA(z); rhon = dot(v,z); % Rayleigh
     quotient
  r = v - rhon*z;     % residual
  z = z - invB(r);    % iteration according to
     (9.3.63)
  z = z/norm(z);      % normalization
  res = [res; rhon]; % tracking iteration
  if (abs(rho-rhon) < tol*abs(rhon)), break;
  else rho = rhon; end
end
lmin =  dot(evalA(z),z); res = [res; lmin],
```

Computational effort:

1 matrix×vector
1 evaluation of preconditioner
A few AXPY-operations

Example 9.3.65 (Convergence of PINVIT)

S.p.d. matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, tridiagonal preconditioner, see Ex. 10.3.11

MATLAB-code 9.3.66:

```
A = spdiags(repmat([1/n,-1,2*(1+1/n),-1,1/n],n,1),
   [-n/2,-1,0,1,n/2],n,n);
evalA = @(x) A*x;
% inverse iteration
invB = @(x) A\x;
% tridiagonal preconditioning
B = spdiags(spdiags(A,[-1,0,1]),[-1,0,1],n,n); invB  = @(x) B\x;
```

Monitored: error decay during iteration of Code 9.3.64: $|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_{\min}(\mathbf{A})|$

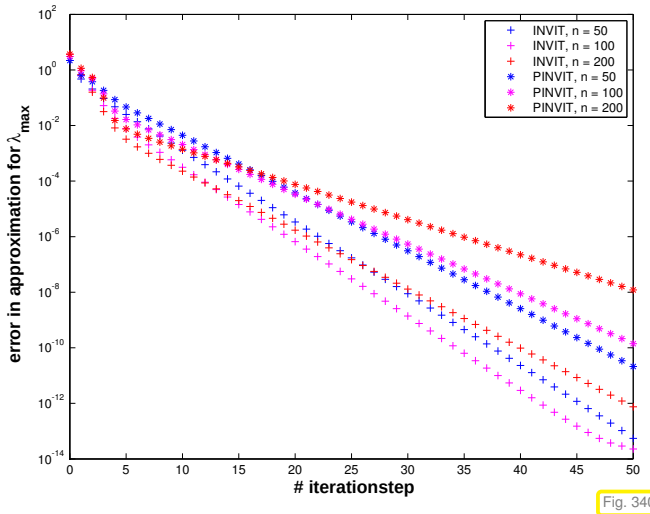


Fig. 339

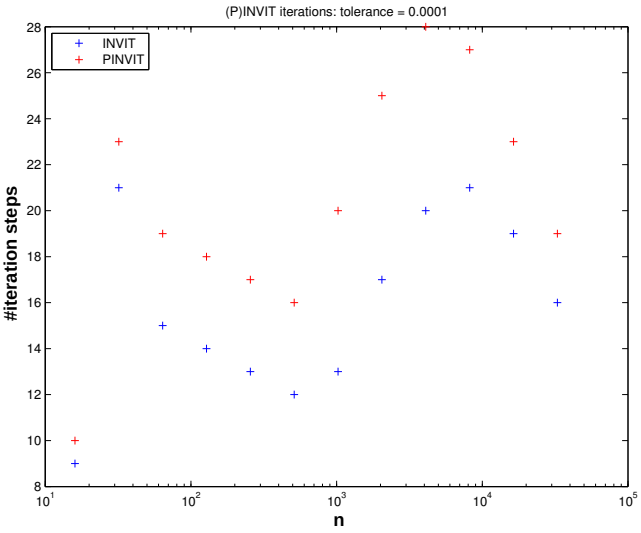


Fig. 340

Observation: linear convergence of eigenvectors also for PINVIT.

Theory [**?**, **?**]:
- linear convergence of (9.3.63)
- fast convergence, if spectral condition number $\kappa(\mathbf{B}^{-1}\mathbf{A})$ small

The theory of PINVIT [**?**, **?**] is based on the identity

$$\mathbf{w} = \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)} + (\mathbf{I} - \mathbf{B}^{-1}\mathbf{A})(\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{A}^{-1}\mathbf{z}^{(k-1)}) \,. \tag{9.3.67}$$

For small residual $\mathbf{A}\mathbf{z}^{(k-1)} - \rho_{\mathbf{A}}(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}$ PINVIT almost agrees with the regular inverse iteration.

9.3.4 Subspace iterations

Remark 9.3.68 (Excited resonances)

Consider the non-autonomous ODE (excited harmonic oscillator)

$$\ddot{y} + \lambda^2 y = \cos(\omega t) \,, \tag{9.3.69}$$

with general solution

$$y(t) = \begin{cases} \dfrac{1}{\lambda^2 - \omega^2}\cos(\omega t) + A\cos(\lambda t) + B\sin(\lambda t) & \text{, if } \lambda \neq \omega \,, \\ \dfrac{t}{2\omega}\sin(\omega t) + A\cos(\lambda t) + B\sin(\lambda t) & \text{, if } \lambda = \omega \,. \end{cases} \tag{9.3.70}$$

growing solutions possible in resonance case $\lambda = \omega$!

Now consider harmonically excited vibration modelled by ODE

$$\ddot{\mathbf{y}} + \mathbf{A}\mathbf{y} = \mathbf{b}\cos(\omega t) \,, \tag{9.3.71}$$

with *symmetric*, *positive (semi)definite* matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$. By Cor. 9.1.9 there is an *orthogonal* matrix $\mathbf{Q} \in \mathbb{R}^{n,n}$ such that

$$\mathbf{Q}^\top \mathbf{A}\mathbf{Q} = \mathbf{D} := \operatorname{diag}(\lambda_1, \ldots, \lambda_n) .$$

where the $0 \leq \lambda_1 < \lambda_2 < \cdots < \lambda_n$ are the eigenvalues of $\mathbf{A}$.

► Transform ODE as in Ex. 9.0.7: with $\mathbf{z} = \mathbf{Q}^\top \mathbf{y}$

$$\text{(9.3.71)} \quad \blacktriangleright \quad \ddot{\mathbf{z}} + \mathbf{D}\mathbf{z} = \mathbf{Q}^\top \mathbf{b} \cos(\omega t) .$$

We have obtained decoupled linear 2nd-order scalar ODEs of the type (9.3.69).

► (9.3.71) can have growing (with time) solutions, if $\omega = \sqrt{\lambda_i}$ for some $i = 1, \ldots, n$

If $\omega = \sqrt{\lambda_j}$ for one $j \in \{1, \ldots, n\}$, then the solution for the initial value problem for (9.3.71) with $\mathbf{y}(0) = \dot{\mathbf{y}}(0) = 0$ ($\leftrightarrow \mathbf{z}(0) = \dot{\mathbf{z}}(0) = 0$) is

$$\mathbf{z}(t) \sim \frac{t}{2\omega} \sin(\omega t)\mathbf{e}_j + \text{bounded oscillations}$$

$$\Updownarrow$$

$$\mathbf{y}(t) \sim \frac{t}{2\omega} \sin(\omega t)(\mathbf{Q})_{:,j} + \text{bounded oscillations} .$$

j-th eigenvector of $\mathbf{A}$

► Eigenvectors of $\mathbf{A}$ $\leftrightarrow$ excitable states

---

Example 9.3.72 (Vibrations of a truss structure *cf.* [?, Sect. 3], MATLAB's `truss` demo)

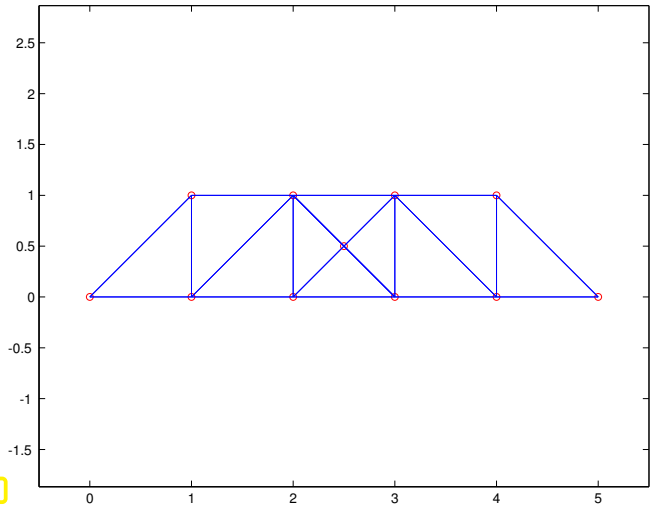


Fig. 341

◁ *a "bridge" truss]*
A truss is a structure composed of (massless) rods and point masses; we consider in-plane (2D) trusses.

Encoding: positions of masses + (sparse) connectivity matrix

MATLAB-code 9.3.73: Data for "bridge truss"

```
% Data for truss structure "bridge"
pos = [ 0 0; 1 0;2 0;3 0;4 0;5 0;1 1;2 1;3 1;4 1;2.5 0.5];
con = [1 2;2 3;3 4;4 5;5 6;1 7;2 7;3 8;2 8;4 8;5 9;5 10;6 10;7
   8;8 9;9 10;8 11 ...
       ; 9 11;3 11;4 11;4 9];
n = size(pos,1);
```

```
top = sparse(con(:,1),con(:,2),ones(size(con,1),1),n,n);
top = sign(top+top');
```

Assumptions:
- ✦ Truss in static equilibrium (perfect balance of forces at each point mass).
- ✦ Rods are *perfectly elastic* (*i.e.*, frictionless).

► Hook's law holds for force in the direction of a rod:

$$F = \alpha \frac{\Delta l}{l} , \tag{9.3.74}$$

where
- ✦ l is the equilibrium length of the rod,
- ✦ Δl is the elongation of the rod effected by the force F in the direction of the rod
- ✦ α is a material coefficient (Young's modulus).

n point masses are numbered $1,\ldots,n$: $\mathbf{p}^i \in \mathbb{R}^2 \triangleq$ position of i-th mass

We consider a swaying truss: description by time-dependent displacements $\mathbf{u}^i(t) \in \mathbb{R}^2$ of point masses:

$$\text{position of } i\text{-th mass at time } t \quad = \quad \mathbf{p}^i + \mathbf{u}^i(t);.$$

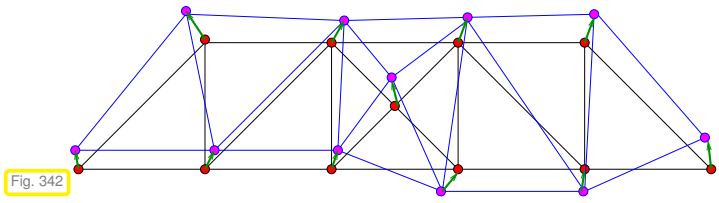


Fig. 342

◁ deformed truss:
- • $\triangleq$ point masses at positions $\mathbf{p}^i$
- $\rightarrow \triangleq$ displacement vectors $\mathbf{u}^i$
- • $\triangleq$ shifted masses at $\mathbf{p}^i + \mathbf{u}^i$

Equilibrium length and (time-dependent) elongation of rod connecting point masses i and j, $i \neq j$:

$$l_{ij} := \left\| \Delta \mathbf{p}^{ji} \right\|_2 , \quad \Delta \mathbf{p}^{ji} := \mathbf{p}^j - \mathbf{p}^i , \tag{9.3.75}$$

$$\Delta l_{ij}(t) := \left\| \Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t) \right\|_2 - l_{ij} , \quad \Delta \mathbf{u}^{ji}(t) := \mathbf{u}^j(t) - \mathbf{u}^i(t) . \tag{9.3.76}$$

► Extra (reaction) force on masses i and j:

$$F_{ij}(t) = -\alpha_{ij} \frac{\Delta l_{ij}}{l_{ij}} \cdot \frac{\Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t)}{\left\| \Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t) \right\|_2} . \tag{9.3.77}$$

Assumption:	Small displacements

► Possibility of linearization by neglecting terms of order $\left\| \mathbf{u}^i \right\|_2^2$

(9.3.75) (9.3.76) ► $$F_{ij}(t) = \alpha_{ij} \left(\frac{1}{\left\| \Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t) \right\|_2} - \frac{1}{\left\| \Delta \mathbf{p}^{ji} \right\|} \right) \cdot \left(\Delta \mathbf{p}^{ji} + \Delta \mathbf{u}^{ji}(t) \right) . \tag{9.3.78}$$

Lemma 9.3.79. Taylor expansion of inverse distance function

For $\mathbf{x} \in \mathbb{R}^d \setminus \{0\}$, $\mathbf{y} \in \mathbb{R}^d$, $\|\mathbf{y}\|_2 < \|\mathbf{x}\|_2$ *holds for* $\mathbf{y} \to 0$

$$\frac{1}{\|\mathbf{x}+\mathbf{y}\|_2} = \frac{1}{\|\mathbf{x}\|_2} - \frac{\mathbf{x}\cdot\mathbf{y}}{\|\mathbf{x}\|_2^3} + O(\|\mathbf{y}\|_2^2) \,.$$

Proof. Simple Taylor expansion up to linear term for $f(\mathbf{x}) = (x_1^2 + \cdots + x_d^2)^{-1/2}$ and $f(\mathbf{x}+\mathbf{y}) = f(\mathbf{x}) + \mathbf{grad}\, f(\mathbf{x})\cdot\mathbf{y} + O(\|\mathbf{y}\|_2^2)$. □

Linearization of force: apply Lemma 9.3.79 to (9.3.78) and drop terms $O(\|\Delta\mathbf{u}^{ji}\|_2^2)$:

$$\begin{aligned} F_{ij}(t) &\approx -\alpha_{ij}\frac{\Delta\mathbf{p}^{ji}\cdot\Delta\mathbf{u}^{ji}(t)}{l_{ij}^3}\cdot(\Delta\mathbf{p}^{ji}+\Delta\mathbf{u}^{ji}(t)) \\ &\approx -\alpha_{ij}\frac{\Delta\mathbf{p}^{ji}\cdot\Delta\mathbf{u}^{ji}(t)}{l_{ij}^3}\cdot\Delta\mathbf{p}^{ji} \,. \end{aligned} \tag{9.3.80}$$

Newton's second law of motion: ($F_i \mathrel{\hat{=}}$ total force acting on i-th mass)

$$\boxed{m_i\frac{d^2}{dt^2}\mathbf{u}^i(t) = F_i = \sum_{\substack{j=1\\ j\neq i}}^n -F_{ij}(t)} \,, \tag{9.3.81}$$

$m_i \mathrel{\hat{=}}$ mass of point mass i.

$$m_i\frac{d^2}{dt^2}\mathbf{u}^i(t) = \sum_{\substack{j=1\\ j\neq i}}^n \alpha_{ij}\frac{1}{l_{ij}^3}\left(\Delta\mathbf{p}^{ji}(\Delta\mathbf{p}^{ji})^\top\right)(\mathbf{u}^j(t)-\mathbf{u}^i(t)) \,. \tag{9.3.82}$$

Compact notation: collect all displacements into one vector $\mathbf{u}(t) = \left(\mathbf{u}^i(t)\right)_{i=1}^n \in \mathbb{R}^{2n}$

(9.3.82) $\Longrightarrow$ $$\boxed{\mathbf{M}\frac{d\mathbf{u}}{dt^2}(t) + \mathbf{A}\mathbf{u}(t) = \mathbf{f}(t)} \,. \tag{9.3.83}$$

with mass matrix $\mathbf{M} = \operatorname{diag}(m_1, m_1, \ldots, m_n, m_n)$

and stiffness matrix $\mathbf{A} \in \mathbb{R}^{2n,2n}$ with 2×2-blocks

$$\begin{aligned} (\mathbf{A})_{2i-1:2i,2i-1,2i} &= \sum_{\substack{j=1\\ j\neq i}}^n \alpha_{ij}\frac{1}{l_{ij}^3}\left(\Delta\mathbf{p}^{ji}(\Delta\mathbf{p}^{ji})^\top\right) \,, \quad i = 1,\ldots,n \,, \\ (\mathbf{A})_{2i-1:2i,2j-1:2j} &= -\alpha_{ij}\frac{1}{l_{ij}^3}\left(\Delta\mathbf{p}^{ji}(\Delta\mathbf{p}^{ji})^\top\right) \,, \quad i \neq j \,. \end{aligned} \tag{9.3.84}$$

and external forces $\mathbf{f}(t) = \left(\mathbf{f}^i(t)\right)_{i=1}^n$.

Note: stiffness matrix $\mathbf{A}$ is *symmetric, positive semidefinite* (→ Def. 1.1.8).

Rem. 9.3.68: if periodic external forces $\mathbf{f}(t) = \cos(\omega t)\mathbf{f}$, $\mathbf{f} \in \mathbb{R}^{2n}$, (wind, earthquake) act on the truss they can excite vibrations of (linearly in time) growing amplitude, if ω coincides with $\sqrt{\lambda_j}$ for an eigenvalue λ_j of $\mathbf{A}$.

Excited vibrations can lead to the collapse of a truss structure, *cf.* the notorious Tacoma-Narrows bridge disaster.

It is essential to know whether eigenvalues of a truss structure fall into a range that can be excited by external forces.

These will typically$^{(*)}$ be the low modes $\leftrightarrow$ a few of the smallest eigenvalues.

($(*)$ Reason: fast oscillations will quickly be damped due to friction, which was neglected in our model.)

MATLAB-code 9.3.85: Computing resonant frequencies and modes of elastic truss

```
function [lambda,V] = trussvib(pos,top)
% Computes vibration modes of a truss structure, see Ex. 9.3.72. Mass point
% positions passed in the n×2-matrix poss and the connectivity encoded in
% the sparse symmetric matrix top. In addition top(i,j) also stores the
% Young's moduli α_ij.
% The 2n resonant frequencies are returned in the vector lambda, the
% eigenmodes in the column of V, where entries at odd positions contain the
% x_1-coordinates, entries at even positions the x_2-coordinates
n = size(pos,1); % no. of point masses
% Assembly of stiffness matrix according to (9.3.84)
A = zeros(2*n,2*n);
[Iidx,Jidx] = find(top); idx = [Iidx,Jidx]; % Find connected masses
for ij = idx'
  i = ij(1); j = ij(2);
  dp = [pos(j,1);pos(j,2)] - [pos(i,1);pos(i,2)]; % Δp^ji
  lij = norm(dp);                                 % l_ij
  A(2*i-1:2*i,2*j-1:2*j) = -(dp*dp')/(lij^3);
end
% Set Young's moduli α_ij (stored in top matrix)
A = A.*full(kron(top,[1 1;1 1]));
% Set 2×2 diagonal blocks
for i=1:n
  A(2*i-1:2*i,2*i-1) = -sum(A(2*i-1:2*i,1:2:end)')';
  A(2*i-1:2*i,2*i)   = -sum(A(2*i-1:2*i,2:2:end)')';
end
% Compute eigenvalues and eigenmodes
[V,D] = eig(A); lambda = diag(D);
```

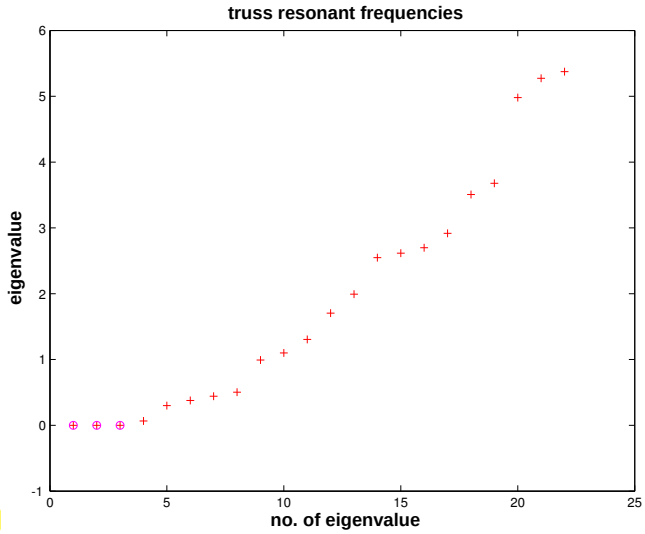


Fig. 343

◁ resonant frequencies of bridge truss from Fig. 341. The stiffness matrix will always possess three zero eigenvalues corresponding to rigid body modes (= displacements without change of length of the rods)

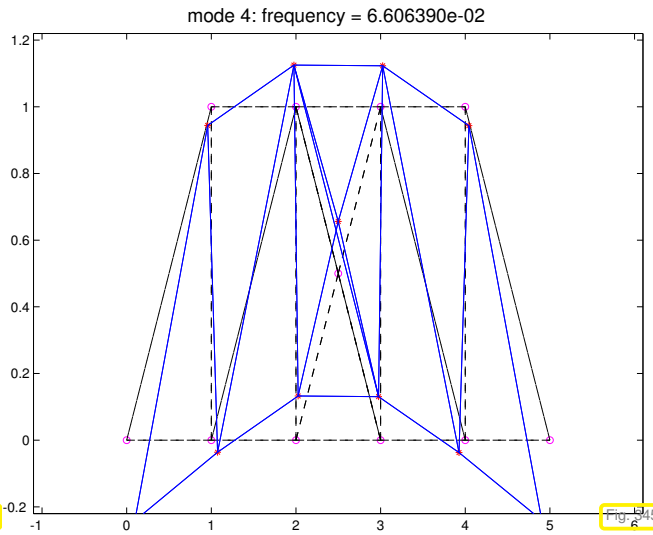


Fig. 344

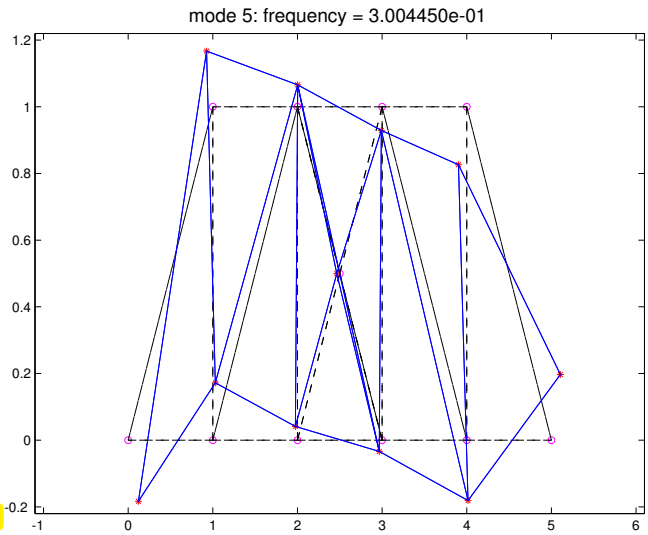


Fig. 345

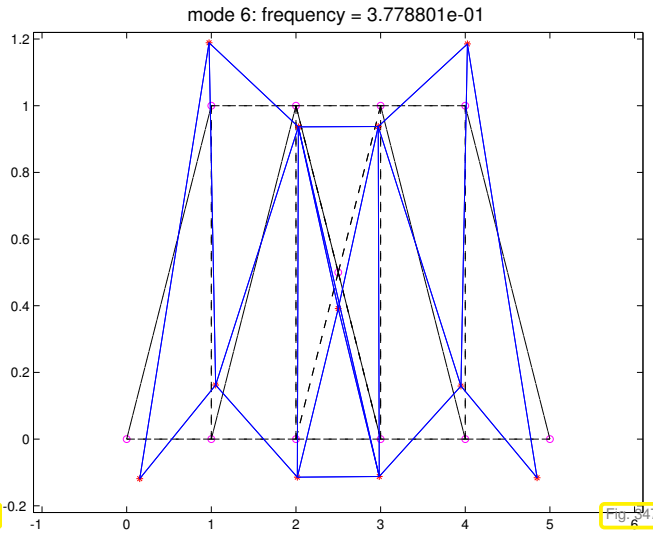


Fig. 346

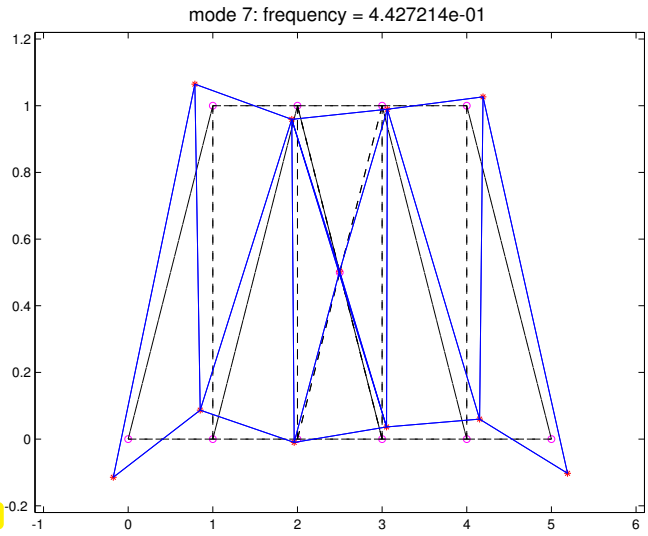


Fig. 347

To compute *a few* of a truss's lowest resonant frequencies and excitable mode, we need efficient numerical methods for the following tasks. Obviously, Code 9.3.85 cannot be used for large trusses, because `eig` invariable operates on dense matrices and will be prohibitively slow and gobble up huge amounts of memory, also recall the discussion of Code 9.3.53.

Task:	Compute m, $m \ll n$, of the smallest/largest (in modulus) eigenvalues of $\mathbf{A} = \mathbf{A}^{\mathrm{H}} \in \mathbb{C}^{n,n}$ and associated eigenvectors.

Of course, we aim to tackle this task by iterative methods generalizing power iteration (→ Section 9.3.1) and inverse iteration (→ Section 9.3.2).

9.3.4.1 Orthogonalization

Preliminary considerations (in $\mathbb{R}$, $m = 2$):

According to Cor. 9.1.9: For $\mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n}$ there is a factorization $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{U}^\top$ with $\mathbf{D} = \operatorname{diag}(\lambda_1, \ldots, \lambda_n)$, $\lambda_j \in \mathbb{R}$, $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$, and $\mathbf{U}$ orthogonal. Thus, $\mathbf{u}_j := (\mathbf{U})_{:,j}$, $j = 1, \ldots, n$, are (mutually orthogonal) eigenvectors of $\mathbf{A}$.

Assume $0 \leq \lambda_1 \leq \cdots \leq \lambda_{n-2} < \lambda_{n-1} < \lambda_n$ (largest eigenvalues are simple).

If we just carry out the direct power iteration (9.3.12) for two vectors both sequences will converge to the largest (in modulus) eigenvector. However, we recall that all eigenvectors are mutually orthogonal. This suggests that we orthogonalize the iterates of the second power iteration (that is to yield the eigenvector for the second largest eigenvalue) with respect to those of the first. This idea spawns the following iteration, *cf.* Gram-Schmidt orthogonalization in (10.2.11):

MATLAB-code 9.3.86: one step of subspace power iteration, $m = 2$

```
1 function [v,w] = sspowitstep(A,v,w)
2 v = A*v; w = A*w; % "power iteration", cf. (9.3.12)
3 % orthogonalization, cf. Gram-Schmidt orthogonalization (10.2.11)
4 v = v/norm(v); w = w - dot(v,w)*v; w = w/norm(w); % now w ⊥ v
```

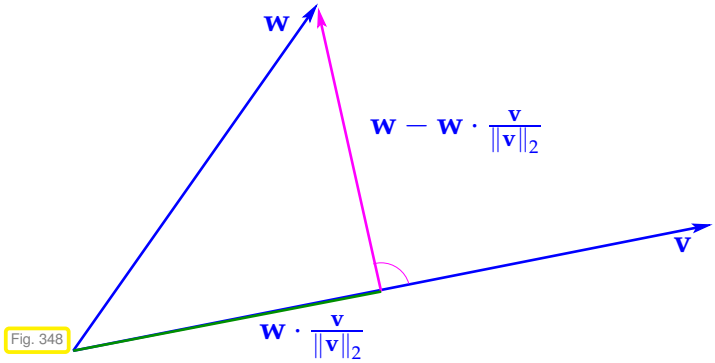


Fig. 348

◁ Orthogonalization of two vectors (see Line 4 of Code 9.3.86)

Analysis through eigenvector expansions ($\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$, $\|\mathbf{v}\|_2 = \|\mathbf{w}\|_2 = 1$)

$$\mathbf{v} = \sum_{i=1}^{n} \alpha_j \mathbf{u}_j \quad , \quad \mathbf{w} = \sum_{i=1}^{n} \beta_j \mathbf{u}_j \, ,$$

$$\Rightarrow \quad \mathbf{A}\mathbf{v} = \sum_{i=1}^{n} \lambda_j \alpha_j \mathbf{u}_j \quad , \quad \mathbf{A}\mathbf{w} = \sum_{i=1}^{n} \lambda_j \beta_j \mathbf{u}_j \, ,$$

$$\mathbf{v}_0 := \frac{\mathbf{v}}{\|\mathbf{v}\|_2} = \Big(\sum_{i=1}^{n} \lambda_j^2 \alpha_j^2\Big)^{-1/2} \sum_{i=1}^{n} \lambda_j \alpha_j \mathbf{u}_j \, ,$$

$$\mathbf{A}\mathbf{w} - (\mathbf{v}_0^\top \mathbf{A}\mathbf{w})\mathbf{v}_0 = \sum_{i=1}^{n} \left(\beta_j - \Big(\sum_{i=1}^{n} \lambda_j^2 \alpha_j \beta_j \Big/ \sum_{i=1}^{n} \lambda_j^2 \alpha_j^2 \Big) \alpha_j \right) \lambda_j \mathbf{u}_j \, .$$

We notice that $\mathbf{v}$ is just mapped to the next iterate in the regular direct power iteration (9.3.12). After many steps, it will be very close to $\mathbf{u}_n$, and, therefore, we may now assume $\mathbf{v} = \mathbf{u}_n \Leftrightarrow \alpha_j = \delta_{j,n}$ (Kronecker symbol).

$$\mathbf{z} := \mathbf{A}\mathbf{w} - (\mathbf{v}_0^\top \mathbf{A}\mathbf{w})\mathbf{v}_0 = 0 \cdot \mathbf{u}_n + \sum_{i=1}^{n-1} \lambda_j \beta_j \mathbf{u}_j \,,$$

$$\mathbf{w}^{(\mathrm{new})} := \frac{\mathbf{z}}{\|\mathbf{z}\|_2} = \Big(\sum_{i=1}^{n-1} \lambda_j^2 \beta_j^2\Big)^{-1/2} \sum_{i=1}^{n-1} \lambda_j \beta_j \mathbf{u}_j \,.$$

The sequence $\mathbf{w}^{(k)}$ produced by repeated application of the mapping given by Code 9.3.86 asymptotically (that is, when $\mathbf{v}^{(k)}$ has already converged to $\mathbf{u}_n$) agrees with the sequence produced by the direct power method for $\widetilde{\mathbf{A}} := \mathbf{U}\,\mathrm{diag}(\lambda_1, \ldots, \lambda_{n-1}, 0)$. Its convergence will be governed by the relative gap $\lambda_{n-2}/\lambda_{n-1}$, see Thm. 9.3.21.

However: if $\mathbf{v}^{(k)}$ itself converges slowly, this reasoning does not apply.

Example 9.3.87 (Subspace power iteration with orthogonal projection)

☞ construction of matrix $\mathbf{A} = \mathbf{A}^\top$ as in Ex. 9.3.60

MATLAB-code 9.3.88: power iteration with orthogonal projection for two vectors

```
function sppowitdriver(d,maxit)
% monitor power iteration with orthogonal projection for finding
% the two largest (in modulus) eigenvalues and associated eigenvectors
% of a symmetric matrix with prescribed eigenvalues passed in d
if (nargin < 10), maxit = 20; end
if (nargin < 1), d = (1:10)'; end
% Generate matrix
n = length(d);
Z = diag(sqrt(1:n),0) + ones(n,n);
[Q,R] = qr(Z);    % generate orthogonal matrix
A = Q*diag(d,0)*Q'; % "synthetic" A = A^T with spectrum σ(A) = {d_1,...,d_n}
% Compute "exact" eigenvectors and eigenvalues
[V,D] = eig(A); [d,idx] = sort(diag(D)),
v_ex = V(:,idx(n)); w_ex = V(:,idx(n-1));
lv_ex = d(n); lw_ex = d(n-1);

v = ones(n,1); w = (-1).^v; % (Arbitrary) initial guess for
   eigenvectors
v = v/norm(v); w = w/norm(w);
result = [];
for k=1:maxit
  v_new = A*v; w_new = A*w; % "power iteration", cf. (9.3.12)
  % Rayleigh quotients provide approximate eigenvalues
  lv = dot(v_new,v); lw = dot(w_new,w);
  % orthogonalization, cf. Gram-Schmidt orthogonalization (10.2.11):
     w ⊥ v
  v = v_new/norm(v_new); w = w_new - dot(v,w_new)*v; w =
     w/norm(w);
  % Record errors in eigenvalue and eigenvector approximations. Note
     that the
  % direction of the eigenvectors is not specified.
  result = [result; k, abs(lv-lv_ex), abs(lw-lw_ex), ...
```

```
            min(norm(v-v_ex),norm(v+v_ex)),
                min(norm(w-w_ex),norm(w+w_ex))];
end

figure('name','sspowit');
semilogy(result(:,1),result(:,2),'m-+',...
     result(:,1),result(:,3),'r-*',...
     result(:,1),result(:,4),'k-^',...
     result(:,1),result(:,5),'b-p');
title('d = [0.5*(1:8),9.5,10]');
xlabel('{\bf power iteration step}','fontsize',14);
ylabel('{\bf error}','fontsize',14);
legend('error in \lambda_n','error in \lambda_n-1','error in
   v','error in w','location','northeast');
print -depsc2 '../PICTURES/sspowitcvg1.eps';

rates = result(2:end,2:end)./result(1:end-1,2:end);
figure('name','rates');
plot(result(2:end,1),rates(:,1),'m-+',...
     result(2:end,1),rates(:,2),'r-*',...
     result(2:end,1),rates(:,3),'k-^',...
     result(2:end,1),rates(:,4),'b-p');
axis([0 maxit 0.5 1]);
title('d = [0.5*(1:8),9.5,10]');
xlabel('{\bf power iteration step}','fontsize',14);
ylabel('{\bf error quotient}','fontsize',14);
legend('error in \lambda_n','error in \lambda_n-1','error in
   v','error in w','location','southeast');
print -depsc2 '../PICTURES/sspowitcvgrates1.eps';
```

$\sigma(\mathbf{A}) = \{1,2,\ldots,10\}$:

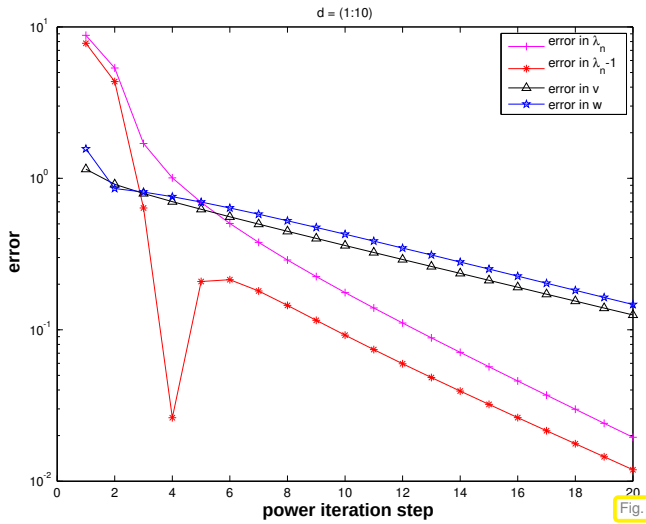


Fig. 349

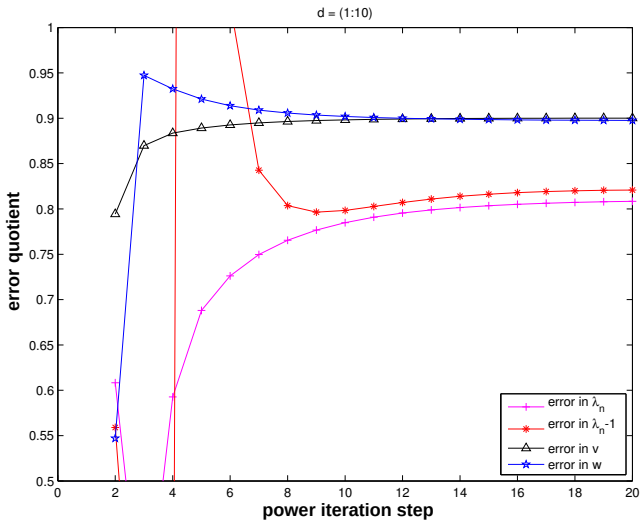


Fig. 350

$\sigma(\mathbf{A}) = \{0.5,1,\ldots,4,9.5,10\}$:

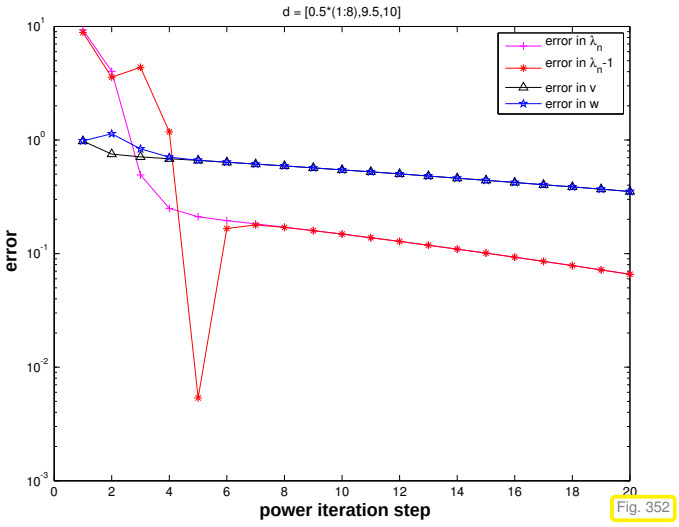


Fig. 351

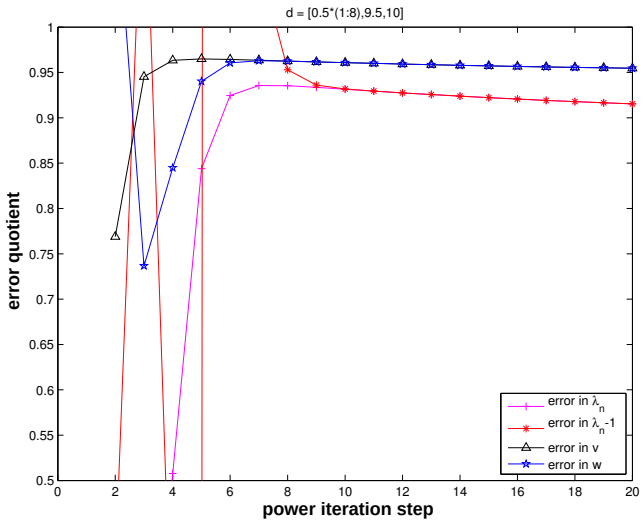


Fig. 352

Issue: generalization of orthogonalization idea to subspaces of dimension > 2

Nothing new:

Gram-Schmidt orthonormalization

(→ [?, Thm. 4.8], [?, Alg. 6.1], [?, Sect. 3.4.3])

Given: linearly independent vectors $\mathbf{v}_1, \ldots, \mathbf{v}_m \in \mathbb{R}^n$, $m \in \mathbb{N}$

Sought: vectors $\mathbf{q}_1, \ldots, \mathbf{q}_m \in \mathbb{R}^n$ such that

❷ $\mathbf{q}_l^\top \mathbf{q}_k = \delta_{lk}$ (orthonormality) , (9.3.89)

❷ $\mathrm{Span}\{\mathbf{q}_1, \ldots, \mathbf{q}_k\} = \mathrm{Span}\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ for all $k = 1, \ldots, m$. (9.3.90)

Constructive proof & algorithm for Gram-Schmidt orthonormalization:

$$\begin{aligned} \mathbf{z}_1 &= \mathbf{v}_1 , \\ \mathbf{z}_2 &= \mathbf{v}_2 - \frac{\mathbf{v}_2^\top \mathbf{z}_1}{\mathbf{z}_1^\top \mathbf{z}_1}\mathbf{z}_1 , \\ \mathbf{z}_3 &= \mathbf{v}_3 - \frac{\mathbf{v}_3^\top \mathbf{z}_1}{\mathbf{z}_1^\top \mathbf{z}_1}\mathbf{z}_1 - \frac{\mathbf{v}_3^\top \mathbf{z}_2}{\mathbf{z}_2^\top \mathbf{z}_2}\mathbf{z}_2 , \\ &\vdots \end{aligned} \tag{9.3.91}$$

+ normalization $\mathbf{q}_k = \dfrac{\mathbf{z}_k}{\|\mathbf{z}_k\|_2}$, $k = 1, \ldots, m$. (9.3.92)

Easy computation: the vectors $\mathbf{q}_1, \ldots, \mathbf{q}_m$ produced by (9.3.91) satisfy (9.3.89) and (9.3.90).

MATLAB-code 9.3.93: Gram-Schmidt orthonormalization (do not use, unstable algorithm!)

```
function Q = gso(V)
% Gram-Schmidt orthonormalization of the columns of V ∈ R^{n,m}, see
% (9.3.91). The vectors q_1,...,q_m are returned as the columns of
% the orthogonal matrix Q.
m = size(V,2);
Q = V(:,1)/norm(V(:,1)); % normalization
for l=2:m
  q = V(:,l);
  % orthogonalization
  for k=1:l-1
```

```
    q = q - dot(Q(:,k),V(:,l))*Q(:,k);
  end
  Q = [Q,q/norm(q)]; % normalization
end
```

Warning! Code 9.3.93 provides an unstable implementation of Gram-Schmidt orthonormalization: for large n, m impact of round-off will destroy the orthogonality of the columns of $\mathbf{Q}$.

A stable implementation of Gram-Schmidt orthogonalization of the columns of a matrix $\mathbf{V} \in \mathbb{K}^{n,m}$, $m \leq n$, is provided by the following MATLAB command:

```
[Q,~] = qr(V,0)
```

(Asymptotic computational cost: $O(m^2 n)$)

dummy return value (for our purposes)

dummy argument

Detailed description of the algorithm behind `qr` and meaning of the return value `R` → Section 3.3.3.

Example 9.3.94 (`qr` based orthogonalization, $m = 2$)

The following two MATLAB code snippets perform the same function, *cf.* Code 9.3.86:

MATLAB-code 9.3.95:

```
v = v/norm(v);
w = w - dot(v,w)*v;
w = w/norm(w);
```

MATLAB-code 9.3.96:

```
[Q,R] = qr([v,w],0);
v = Q(:,1);
w = Q(:,2);
```

Explanation ➢ discussion of Gram-Schmidt orthonormalization.

MATLAB-code 9.3.97: General subspace power iteration step with `qr` based orthonormalization

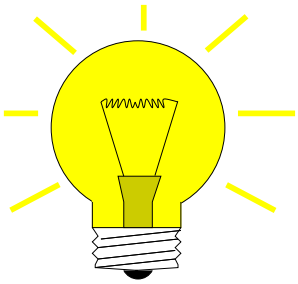
```
function V = sspowitstep(A,V)
% power iteration with orthonormalization for A = A^T.
% columns of matrix V span subspace for power iteration.
V = A*V;          % actual power iteration on individual columns
[V,R] = qr(V,0); % Gram-Schmidt orthonormalization (9.3.91)
```

9.3.4.2 Ritz projection

Observations on Code 9.3.86:

- the first column of $\mathbf{V}$, $(\mathbf{V})_{:,1}$, is a sequence of vectors created by the standard direct power method (9.3.12).

✦ reasoning: the other columns of $\mathbf{V}$, after each multiplication with $\mathbf{A}$ can be expected to contain a significant component in the direction of the eigenvector associated with the eigenvalue of largest modulus.



Idea: use information in $(\mathbf{V})_{:,2},\ldots,(\mathbf{V})_{:,\text{end}}$ to accelerate convergence of $(\mathbf{V})_{:,1}$.

Since the columns of $\mathbf{V}$ span a *subspace* of $\mathbb{R}^n$, this idea can be recast as the following task:

Task:	given $\mathbf{v}_1,\ldots,\mathbf{v}_k \in \mathbb{K}^n$, $k \ll n$, extract (good approximations of) eigenvectors of $\mathbf{A} = \mathbf{A}^{\mathrm{H}} \in \mathbb{K}^{n,n}$ contained in $\mathrm{Span}\{\mathbf{v}_1,\ldots,\mathbf{v}_m\}$.

We take for granted that $\{\mathbf{v}_1,\ldots,\mathbf{v}_m\}$ is linearly independent.

Assumption: $\mathbf{Eig}\mathbf{A}\lambda \cap V := \mathrm{Span}\{\mathbf{v}_1,\ldots,\mathbf{v}_m\} \neq \{0\}$

$\Leftrightarrow$ V contains an eigenvector of $\mathbf{A}$

$$\begin{aligned} &\Leftrightarrow \quad \exists \mathbf{w} \in V \setminus \{0\}: \quad \mathbf{A}\mathbf{w} = \lambda \mathbf{w} \\ &\Leftrightarrow \quad \exists \mathbf{u} \in \mathbb{K}^m \setminus \{0\}: \quad \mathbf{A}\mathbf{V}\mathbf{u} = \lambda \mathbf{V}\mathbf{u} \\ &\Rightarrow \quad \exists \mathbf{u} \in \mathbb{K}^m \setminus \{0\}: \quad \mathbf{V}^{\mathrm{H}}\mathbf{A}\mathbf{V}\mathbf{u} = \lambda \mathbf{V}^{\mathrm{H}}\mathbf{V}\mathbf{u}\,, \end{aligned} \tag{9.3.98}$$

where $\mathbf{V} := (\mathbf{v}_1,\ldots,\mathbf{v}_m) \in \mathbb{K}^{n,m}$ and we used

$$V = \{\mathbf{V}\mathbf{u} : \mathbf{u} \in \mathbb{K}^m\} \quad \text{(linear combinations of the } \mathbf{v}_i\text{)}.$$

(9.3.98) ➢ $\mathbf{u} \in \mathbb{K}^k \setminus \{0\}$ solves $m \times m$ generalized eigenvalue problem

$$(\mathbf{V}^{\mathrm{H}}\mathbf{A}\mathbf{V})\mathbf{u} = \lambda(\mathbf{V}^{\mathrm{H}}\mathbf{V})\mathbf{u}\,. \tag{9.3.99}$$

Note: $\{\mathbf{v}_1,\ldots,\mathbf{v}_m\}$ linearly independent

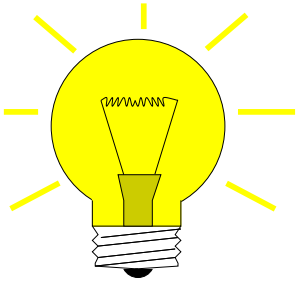
$\Updownarrow$

$\mathbf{V}$ has full rank m ($\to$ Def. 2.2.3)

$\Updownarrow$

$\mathbf{V}^{\mathrm{H}}\mathbf{V}$ is symmetric positive definite ($\to$ Def. 1.1.8)

If our initial assumption holds true and $\mathbf{u}$ solves (9.3.99) and is a simple eigenvalue, a corresponding $\mathbf{x} \in \mathbf{Eig}\mathbf{A}\lambda$ can be recovered as $\mathbf{x} = \mathbf{V}\mathbf{u}$.



Idea: Given a subspace $V = \mathrm{Im}(\mathbf{V}) \subset \mathbb{K}^n$, $\mathbf{V} \in \mathbb{K}^{n,m}$, $\dim(V) = m$, obtain *approximations* of (a few) eigenvalues and eigenvectors $\mathbf{x}_1,\ldots,\mathbf{x}_m$ of $\mathbf{A}$ by

❶ solving the generalized eigenvalue problem (9.3.99)

➔ eigenvectors $\mathbf{u}_1,\ldots,\mathbf{u}_k \in \mathbb{K}^m$ (linearly independent),

❷ and transforming them back according to $\mathbf{x}_k = \mathbf{V}\mathbf{u}_k$, $k = 1, \ldots, m$.

Terminology: (9.3.99) is called the Ritz projection of EVP $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ onto V

Terminology: $\sigma(\mathbf{V}^H\mathbf{A}\mathbf{V}) \hat{=}$ Ritz values,
eigenvectors of $\mathbf{V}^H\mathbf{A}\mathbf{V} \hat{=}$ Ritz vectors

Example: Ritz projection of $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ onto $\mathrm{Span}\{\mathbf{v}, \mathbf{w}\}$:

$$(\mathbf{v}, \mathbf{w})^H \mathbf{A} (\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \lambda (\mathbf{v}, \mathbf{w})^H (\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} .$$

Note: If $\mathbf{V}$ is *unitary* ($\rightarrow$ Def. 6.2.2), then the generalized eigenvalue problem (9.3.99) will become a standard linear eigenvalue problem.

Remark 9.3.100 (Justification of Ritz projection by min-max theorem)

We revisit $m = 2$, see Code 9.3.86. Recall that by the min-max theorem Thm. 9.3.41

$$\mathbf{u}_n = \mathrm{argmax}_{\mathbf{x}\in\mathbb{R}^n} \rho_{\mathbf{A}}(\mathbf{x}) \quad , \quad \mathbf{u}_{n-1} = \mathrm{argmax}_{\mathbf{x}\in\mathbb{R}^n, \mathbf{x}\perp\mathbf{u}_n} \rho_{\mathbf{A}}(\mathbf{x}) . \tag{9.3.101}$$

Idea: maximize Rayleigh quotient over $\mathrm{Span}\{\mathbf{v}, \mathbf{w}\}$, where $\mathbf{v}$, $\mathbf{w}$ are output by Code 9.3.86. This leads to the optimization problem

$$(\alpha^*, \beta^*) := \operatorname*{argmax}_{\alpha,\beta\in\mathbb{R}, \alpha^2+\beta^2=1} \rho_{\mathbf{A}}(\alpha\mathbf{v} + \beta\mathbf{w}) = \operatorname*{argmax}_{\alpha,\beta\in\mathbb{R}, \alpha^2+\beta^2=1} \rho_{(\mathbf{v},\mathbf{w})^\top \mathbf{A} (\mathbf{v},\mathbf{w})}\left(\begin{pmatrix} \alpha \\ \beta \end{pmatrix}\right) . \tag{9.3.102}$$

Then a better approximation for the eigenvector to the largest eigenvalue is

$$\mathbf{v}^* := \alpha^*\mathbf{v} + \beta^*\mathbf{w} .$$

Note that $\|\mathbf{v}^*\|_2 = 1$, if both $\mathbf{v}$ and $\mathbf{w}$ are normalized, which is guaranteed in Code 9.3.86.

Then, orthogonalizing $\mathbf{w}$ w.r.t $\mathbf{v}^*$ will produce a new iterate $\mathbf{w}^*$.

Again the min-max theorem Thm. 9.3.41 tells us that we can find $(\alpha^*, \beta^*)^\top$ as eigenvector to the largest eigenvalue of

$$(\mathbf{v}, \mathbf{w})^\top \mathbf{A} (\mathbf{v}, \mathbf{w}) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \lambda \begin{pmatrix} \alpha \\ \beta \end{pmatrix} . \tag{9.3.103}$$

Since eigenvectors of symmetric matrices are mutually orthogonal, we find $\mathbf{w}^* = \alpha_2\mathbf{v} + \beta_2\mathbf{w}$, where $(\alpha_2, \beta_2)^\top$ is the eigenvector of (9.3.103) belonging to the smallest eigenvalue. This assumes orthonormal vectors $\mathbf{v}$, $\mathbf{w}$.

MATLAB-code 9.3.104: one step of subspace power iteration with Ritz projection, matrix version

```
function V = sspowitsteprp(A,V)
V = A*V;              % power iteration applied to columns of V
[Q,R] = qr(V,0);      % orthonormalization, see Section 9.3.4.1
[U,D] = eig(Q'*A*Q);  % Solve Ritz projected m × m eigenvalue problem
V = Q*U;              % recover approximate eigenvectors
```

```
6 ev = diag(D);        % approximate eigenvalues
```

Note that he orthogonalization step in Code 9.3.104 is actually redundant, if exact arithmetic could be employed, because the Ritz projection could also be realized by solving the generalized eigenvalue problem.

However, prior orthogonalization is essential for numerical stability (→ Def. 1.5.85), *cf.* the discussion in Section 3.3.3.

Example 9.3.105 (Power iteration with Ritz projection)

Listing 9.1: Main loop: power iteration with Ritz projection for two eigenvectors

```
% See Code 9.3.88 for generation of matrix A and output
for k=1:maxit
  v_new = A*v; w_new = A*w;      % "power iteration", cf. (9.3.12)
  [Q,R] = qr([v_new,w_new],0); % orthogonalization, see Sect. 9.3.4.1
  [U,D] = eig(Q'*A*Q);          % Solve Ritz projected eigenvalue problem
  [ev,idx] = sort(abs(diag(D))), % Sort eigenvalues
  w = Q*U(:,idx(1)); v = Q*U(:,idx(2)); % Recover approximate
     eigenvectors

  % Record errors in eigenvalue and eigenvector approximations.  Note that
     the
  % direction of the eigenvectors is not specified.
  result = [result; k, abs(ev(2)-lv_ex), abs(ev(1)-lw_ex), ...
            min(norm(v-v_ex),norm(v+v_ex)),
               min(norm(w-w_ex),norm(w+w_ex))];
end
```

Matrix as in Ex. 9.3.87, $\sigma(\mathbf{A}) = \{0.5, 1, \ldots, 4, 9.5, 10\}$:

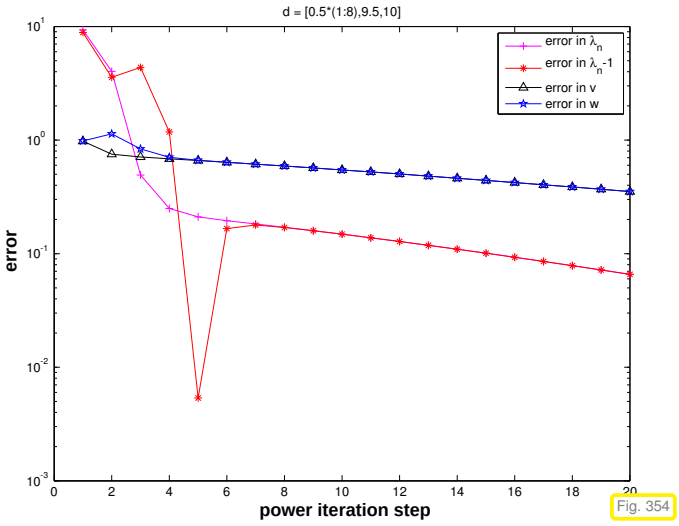


Fig. 353

simple orthonormalization, Ex. 9.3.87

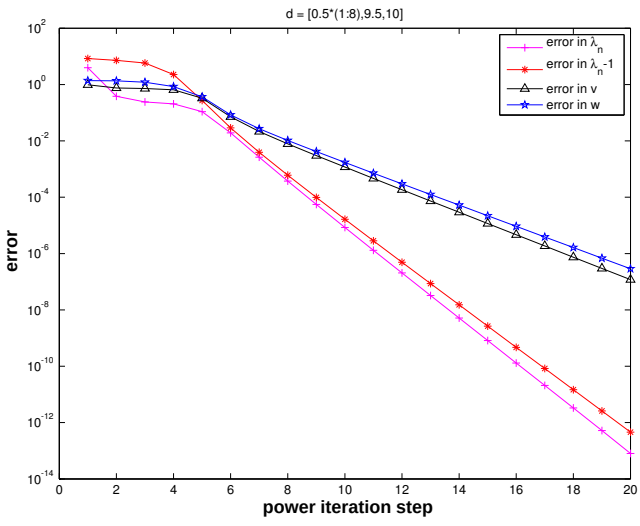


Fig. 354

with Ritz projection, Code 9.3.104

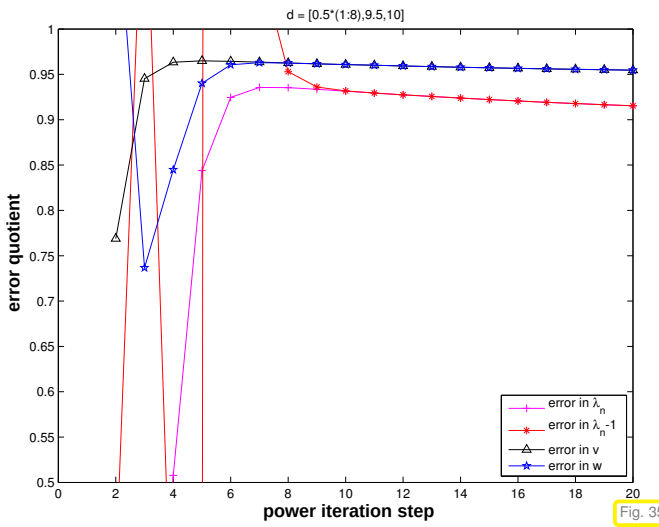


Fig. 355

simple orthonormalization, Ex. 9.3.87

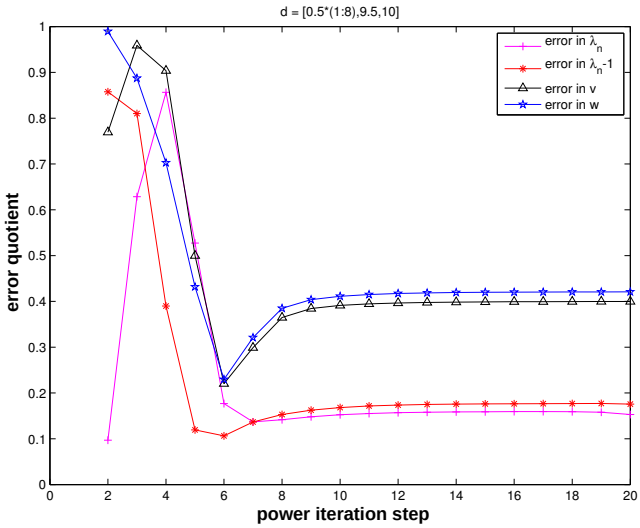


Fig. 356

with Ritz projection, Code 9.3.104

Observation: tremendous acceleration of power iteration through Ritz projection, convergence still linear but with much better rates.

In Code 9.3.104: diagonal entries of $\mathbf{D}$ provide approximations of eigenvalues. Their (relative) changes can be used as a termination criterion.

(9.3.106) Subspace variant of direct power method with Ritz projection

MATLAB-code 9.3.107: Subspace power iteration with Ritz projection

```
function [ev,V] = sspowitrp(A,k,m,tol,maxit)
% Power iteration with Ritz projection for matrix A = A^T ∈ R^{n,n}:
% Subspace of dimension m ≤ n is used to compute the k ≤ m largest
% eigenvalues of A and associated eigenvectors.
n = size(A,1); V = eye(n,m); d = zeros(m,1); % (Arbitrary) initial eigenvectors
% The approximate eigenvectors are stored in the columns of V ∈ R^{n,m}
for i=1:maxit
  [Q,R] = qr(A*V,0); % Power iteration and orthonormalization
  [U,D] = eig(Q'*A*Q); % Small m × m eigenvalue problem for Ritz projection
  [ev,idx] = sort(diag(D)); % eigenvalue approximations in diagonal of D
  V = Q*U; % 2nd part of Ritz projection
  if (abs(ev-d) < tol*max(abs(ev))), break; end
  d = ev;
end
ev = ev(m-k+1:end);
V = V(:,idx(m-k+1:end));
```

Example 9.3.108 (Convergence of subspace variant of direct power method)

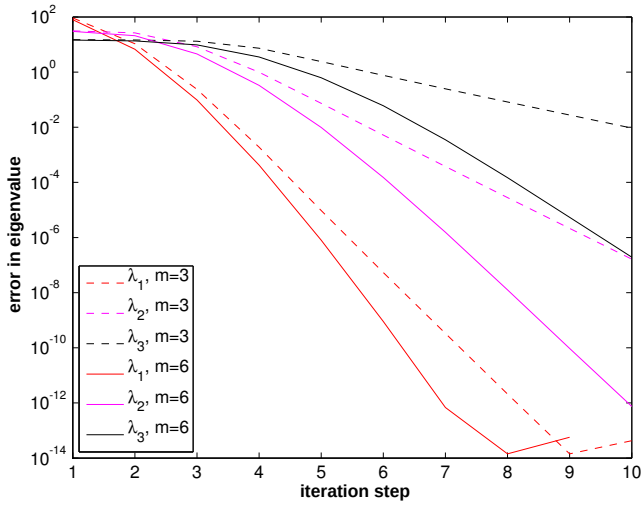


Fig. 357

S.p.d. test matrix: $a_{ij} := \min\{\frac{i}{j}, \frac{j}{i}\}$

```
n=200; A = gallery('lehmer',n);
```

"Initial eigenvector guesses":

```
V = eye(n,m);
```

- Observation: linear convergence of eigenvalues
- choice $m > k$ boosts convergence of eigenvalues

Remark 9.3.109 (Subspace power methods)

Analoguous to § 9.3.106: construction of subspace variants of inverse iteration (→ Code 9.3.54), PINVIT (9.3.63), and Rayleigh quotient iteration (9.3.59).

9.4 Krylov Subspace Methods

Supplementary reading. [**?**, Sect. 30]

All power methods (→ Section 9.3) for the eigenvalue problem (EVP) $\mathbf{Ax} = \lambda\mathbf{x}$ only rely on the last iterate to determine the next one (1-point methods, *cf.* (8.1.4))

➢ NO MEMORY, *cf.* discussion in the beginning of Section 10.2.

"Memory for power iterations": pursue same idea that led from the gradient method, § 10.1.11, to the conjugate gradient method, § 10.2.17: use information from previous iterates to achieve efficient minimization over larger and larger subspaces.

Min-max theorem, Thm. 9.3.41	:	$\mathbf{A} = \mathbf{A}^H$	$\Rightarrow$	EVPs	$\Leftrightarrow$	Finding extrema/stationary points of Rayleigh quotient (→ Def. 9.3.16)

Setting: EVP $\mathbf{Ax} = \lambda\mathbf{x}$ for real s.p.d. (→ Def. 1.1.8) matrix $\mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}$

notations used below: $0 < \lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$: eigenvalues of $\mathbf{A}$, counted with multiplicity, see Def. 9.1.1,

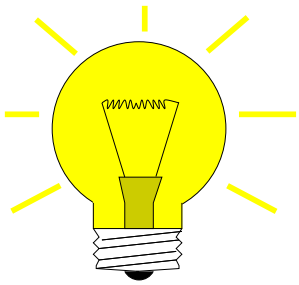
$\mathbf{u}_1, \ldots, \mathbf{u}_n \hat{=}$ corresponding orthonormal eigenvectors, *cf.* Cor. 9.1.9.

$$\mathbf{AU} = \mathbf{DU} \quad , \quad \mathbf{U} = (\mathbf{u}_1, \ldots, \mathbf{u}_n) \in \mathbb{R}^{n,n} \, , \quad \mathbf{D} = \operatorname{diag}(\lambda_1, \ldots, \lambda_n) \, .$$

We recall

- the direct power method (9.3.12) from Section 9.3.1
- and the inverse iteration from Section 9.3.2

and how they produce sequences $(\mathbf{z}^{(k)})_{k\in\mathbb{N}_0}$ of vectors that are supposed to converge to a vector $\in \mathbf{EigA}\lambda_1$ or $\in \mathbf{EigA}\lambda_n$, respectively.



Idea: Better $\mathbf{z}^{(k)}$ from Ritz projection onto $V := \mathrm{Span}\{\mathbf{z}^{(0)},\ldots,\mathbf{z}^{(k)}\}$
(= space spanned by previous iterates)

Recall (→ Code 9.3.104) Ritz projection of an EVP $\mathbf{Ax} = \lambda\mathbf{x}$ onto a subspace $V := \mathrm{Span}\{\mathbf{v}_1,\ldots,\mathbf{v}_m\}$, $m < n$ ➡ smaller $m\times m$ generalized EVP

$$\underbrace{\mathbf{V}^T\mathbf{AV}}_{:=\mathbf{H}}\mathbf{x} = \lambda\mathbf{V}^T\mathbf{Vx} \quad , \quad \mathbf{V} := (\mathbf{v}_1,\ldots,\mathbf{v}_m) \in \mathbb{R}^{n,m} \, . \tag{9.4.1}$$

From Rayleigh quotient Thm. 9.3.39 and considerations in Section 9.3.4.2:

$$\begin{aligned} \mathbf{u}_n \in V \quad &\Rightarrow \quad \text{largest eigenvalue of (9.4.1)} = \lambda_{\max}(\mathbf{A}) \, , \\ \mathbf{u}_1 \in V \quad &\Rightarrow \quad \text{smallest eigenvalue of (9.4.1)} = \lambda_{\min}(\mathbf{A}) \, . \end{aligned}$$

Intuition: If $\mathbf{u}_n(\mathbf{u}_1)$ "well captured" by V (that is, the angle between the vector and the space V is small), then we can expect that the largest (smallest) eigenvalue of (9.4.1) is a good approximation for $\lambda_{\max}(\mathbf{A})(\lambda_{\min}(\mathbf{A}))$, and that, assuming normalization

$$\mathbf{Vw} \approx \mathbf{u}_1 \quad (\text{or } \mathbf{Vw} \approx \mathbf{u}_n) \, ,$$

where $\mathbf{w}$ is the corresponding eigenvector of (9.4.1).

For direct power method (9.3.12): $\mathbf{z}^{(k)} || \mathbf{A}^k\mathbf{z}^{(0)}$

$$V = \mathrm{Span}\{\mathbf{z}^{(0)}, \mathbf{Az}^{(0)}, \ldots, \mathbf{A}^{(k)}\mathbf{z}^{(0)}\} = \mathcal{K}_{k+1}(\mathbf{A}, \mathbf{z}^{(0)}) \quad \text{a Krylov space, → Def. 10.2.6 .} \tag{9.4.2}$$

MATLAB-code 9.4.3: Ritz projections onto Krylov space (9.4.2)

```
function [V,D] = kryleig(A,m)
% Ritz projection onto Krylov subspace. An
    orthonormal basis of K_m(A,1) is assembled
    into the columns of V.
n = size(A,1); V = (1:n)'; V =
   V/norm(V);
for l=1:m-1
  V = [V,A*V(:,end)]; [Q,R] = qr(V,0);
  [W,D] = eig(Q'*A*Q); V = Q*W;
end
```

◁ direct power method with Ritz projection onto Krylov space from (9.4.2), *cf.* § 9.3.106.

Note: implementation for demonstration purposes only (inefficient for sparse matrix $\mathbf{A}$!)

Example 9.4.4 (Ritz projections onto Krylov space)

MATLAB-code 9.4.5:

```
n=100;
M=gallery('tridiag',-0.5*ones(n-1,1),2*ones(n,1),-1.5*ones(n-1,1));
[Q,R]=qr(M); A=Q'*diag(1:n)*Q; % synthetic matrix,
    σ(A) = {1,2,3,...,100}
```

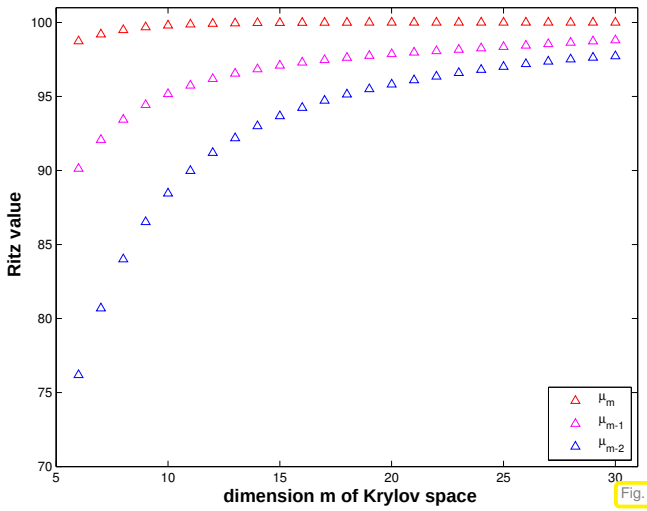


Fig. 358

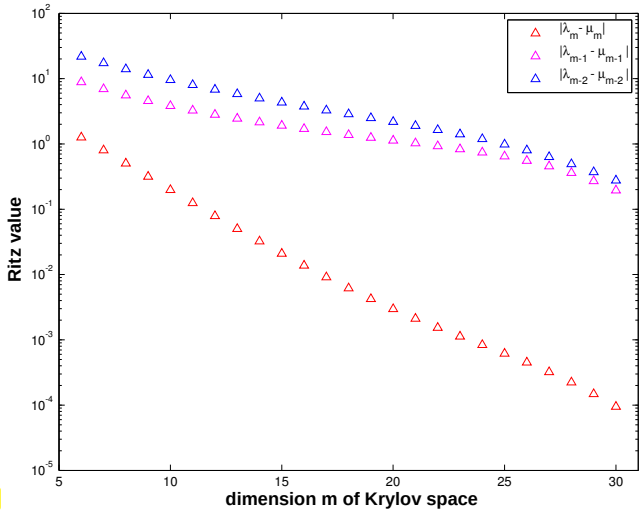


Fig. 359

Observation: "vaguely linear" convergence of largest Ritz values (notation μ_i) to largest eigenvalues. Fastest convergence of largest Ritz value $\rightarrow$ largest eigenvalue of $\mathbf{A}$

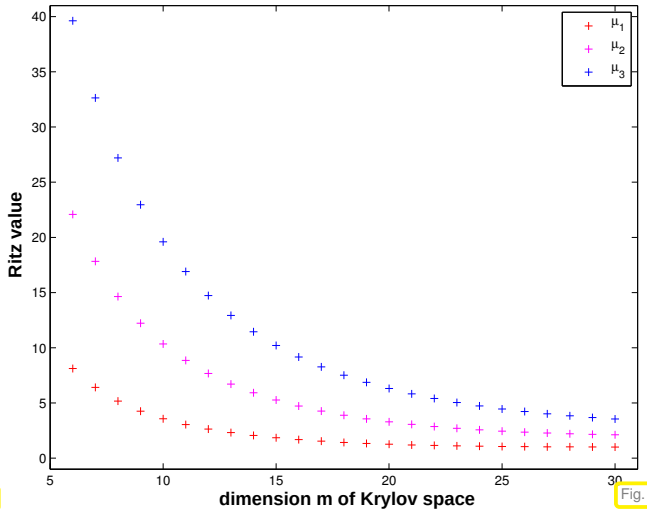


Fig. 360

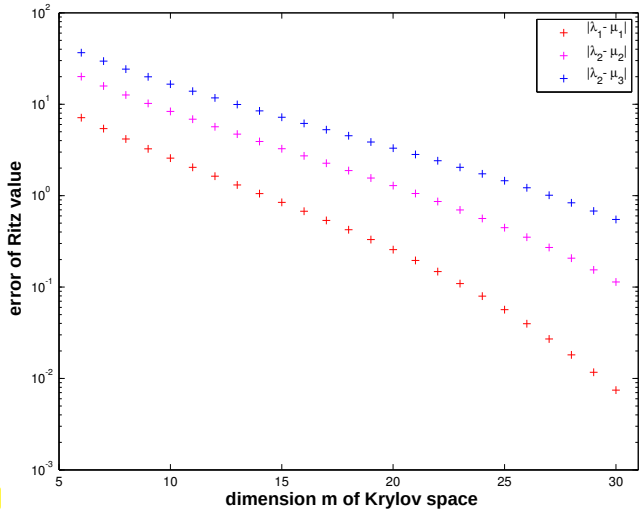


Fig. 361

Observation: *Also the smallest Ritz values converge* "vaguely linearly" to the smallest eigenvalues of $\mathbf{A}$. Fastest convergence of smallest Ritz value $\rightarrow$ smallest eigenvalue of $\mathbf{A}$.

? Why do smallest Ritz values converge to smallest eigenvalues of $\mathbf{A}$?

Consider direct power method (9.3.12) for $\widetilde{\mathbf{A}} := \nu\mathbf{I} - \mathbf{A}$, $\nu > \lambda_{\max}(\mathbf{A})$:

$$\mathbf{z}^{(0)} \text{ arbitrary}, \quad \widetilde{\mathbf{z}}^{(k+1)} = \frac{(\nu\mathbf{I} - \mathbf{A})\widetilde{\mathbf{z}}^{(k)}}{\left\|(\nu\mathbf{I} - \mathbf{A})\widetilde{\mathbf{z}}^{(k)}\right\|_2} \tag{9.4.6}$$

As $\sigma(\nu\mathbf{I} - \mathbf{A}) = \nu - \sigma(\mathbf{A})$ and eigenspaces agree, we infer from Thm. 9.3.21

$$\lambda_1 < \lambda_2 \quad \Rightarrow \quad \mathbf{z}^{(k)} \overset{k\to\infty}{\longrightarrow} \mathbf{u}_1 \quad \& \quad \rho_{\mathbf{A}}(\mathbf{z}^{(k)}) \overset{k\to\infty}{\longrightarrow} \lambda_1 \quad \text{linearly}. \tag{9.4.7}$$

By the binomial theorem (also applies to matrices, if they commute)

$$(\nu\mathbf{I}-\mathbf{A})^k = \sum_{j=0}^{k}\binom{k}{j}\nu^{k-j}\mathbf{A}^j \quad \Rightarrow \quad (\nu\mathbf{I}-\mathbf{A})^k\widetilde{\mathbf{z}}^{(0)} \in \mathcal{K}_k(\mathbf{A},\mathbf{z}^{(0)}) ,$$

$$\boxed{\mathcal{K}_k(\nu\mathbf{I}-\mathbf{A},\mathbf{x}) = \mathcal{K}_k(\mathbf{A},\mathbf{x})} . \tag{9.4.8}$$

➢ $\mathbf{u}_1$ can also be expected to be "well captured" by $\mathcal{K}_k(\mathbf{A},\mathbf{x})$ and the smallest Ritz value should provide a good aproximation for $\lambda_{\min}(\mathbf{A})$.

Recall from Section 10.2.2 Lemma 10.2.12 :

Residuals $\mathbf{r}_0,\ldots,\mathbf{r}_{m-1}$ generated in CG iteration, § 10.2.17 applied to $\mathbf{Ax}=\mathbf{z}$ with $\mathbf{x}^{(0)}=0$, provide *orthogonal basis* for $\mathcal{K}_m(\mathbf{A},\mathbf{z})$ (, if $\mathbf{r}_k \neq 0$).

Inexpensive Ritz projection of $\mathbf{Ax}=\lambda\mathbf{x}$ onto $\mathcal{K}_m(\mathbf{A},\mathbf{z})$: orthogonal matrix

$$\mathbf{V}_m^T\mathbf{A}\mathbf{V}_m\mathbf{x} = \lambda\mathbf{x} , \quad \mathbf{V}_m := \left(\frac{\mathbf{r}_0}{\|\mathbf{r}_0\|},\ldots,\frac{\mathbf{r}_{m-1}}{\|\mathbf{r}_{m-1}\|}\right) \in \mathbb{R}^{n,m} . \tag{9.4.9}$$

recall: residuals generated by *short recursions*, see § 10.2.17

Lemma 9.4.10. Tridiagonal Ritz projection from CG residuals

$\mathbf{V}_m^T\mathbf{A}\mathbf{V}_m$ *is a tridiagonal matrix.*

Proof. Lemma 10.2.12: $\{\mathbf{r}_0,\ldots,\mathbf{r}_{\ell-1}\}$ is an orthogonal basis of $\mathcal{K}_\ell(\mathbf{A},\mathbf{r}_0)$, if all the residuals are non-zero. As $\mathbf{A}\mathcal{K}_{\ell-1}(\mathbf{A},\mathbf{r}_0) \subset \mathcal{K}_\ell(\mathbf{A},\mathbf{r}_0)$, we conclude the orthogonality $\mathbf{r}_m^T\mathbf{A}\mathbf{r}_j$ for all $j=0,\ldots,m-2$. Since

$$\left(\mathbf{V}_m^T\mathbf{A}\mathbf{V}_m\right)_{ij} = \mathbf{r}_{i-1}^T\mathbf{A}\mathbf{r}_{j-1} , \quad 1\le i,j\le m ,$$

the assertion of the theorem follows. □

$$\mathbf{V}_l^H\mathbf{A}\mathbf{V}_l = \begin{bmatrix} \alpha_1 & \beta_1 & & & & & \\ \beta_1 & \alpha_2 & \beta_2 & & & & \\ & \beta_2 & \alpha_3 & \ddots & & & \\ & & \ddots & \ddots & & & \\ & & & & \ddots & & \\ & & & \ddots & \ddots & \beta_{k-1} & \\ & & & & \beta_{k-1} & \alpha_k & \end{bmatrix} =: \mathbf{T}_l \in \mathbb{K}^{k,k} \quad \text{[tridiagonal matrix]} \tag{9.4.11}$$

Algorithm for computing $\mathbf{V}_l$ and $\mathbf{T}_l$:
Lanczos process
Computational effort/step:

1×	$\mathbf{A}\times$vector
2	dot products
2	AXPY-operations
1	division

Closely related to CG iteration, § 10.2.17, Code 10.2.18.

MATLAB-code 9.4.12: Lanczos process, *cf.* Code 10.2.18

```
function [V,alph,bet] = lanczos(A,k,z0)
% Note: this implementation of the Lanczos
    process also records the orthonormal CG
    residuals in the columns of the matrix V,
    which is not needed when only eigenvalue
    approximations are desired.
V = z0/norm(z0);
% Vectors storing entries of tridiagonal matrix
    (9.4.11)
alph=zeros(k,1); bet = zeros(k,1);
for j=1:k
  q = A*V(:,j); alph(j) = dot(q,V(:,j));
  w = q - alph(j)*V(:,j);
  if (j > 1), w = w - bet(j-1)*V(:,j-1);
    end;
  bet(j) = norm(w); V = [V,w/bet(j)];
end
bet = bet(1:end-1);
```

Total computational effort for l steps of Lanczos process, if $\mathbf{A}$ has at most k non-zero entries per row: $O(nkl)$

Note: Code 9.4.12 assumes that no residual vanishes. This could happen, if $\mathbf{z}_0$ exactly belonged to the span of a few eigenvectors. However, in practical computations inevitable round-off errors will always ensure that the iterates do not stay in an invariant subspace of $\mathbf{A}$, *cf.* Rem. 9.3.22.

Convergence (what we expect from the above considerations) → [**?**, Sect. 8.5])

$$\text{In } l\text{-th step: } \lambda_n \approx \mu_l^{(l)}, \lambda_{n-1} \approx \mu_{l-1}^{(l)}, \ldots, \lambda_1 \approx \mu_1^{(l)},$$
$$\sigma(\mathbf{T}_l) = \{\mu_1^{(l)}, \ldots, \mu_l^{(l)}\}, \quad \mu_1^{(l)} \le \mu_2^{(l)} \le \cdots \le \mu_l^{(l)}.$$

Example 9.4.13 (Lanczos process for eigenvalue computation)

$\mathbf{A}$ from Ex. 9.4.4

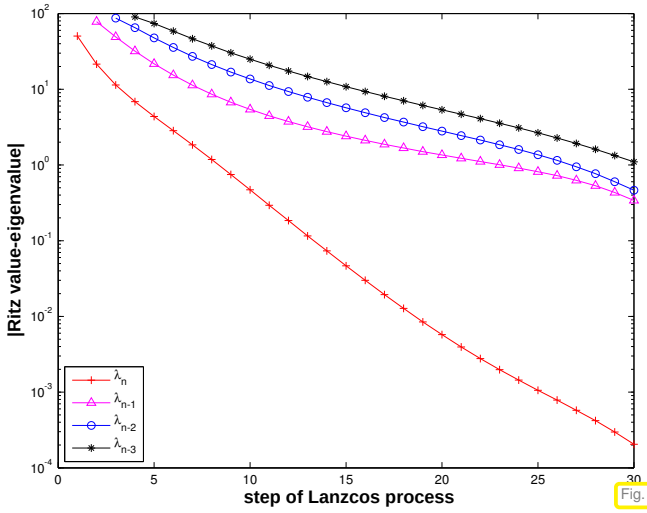


Fig. 362

```
A = gallery('minij',100);
```

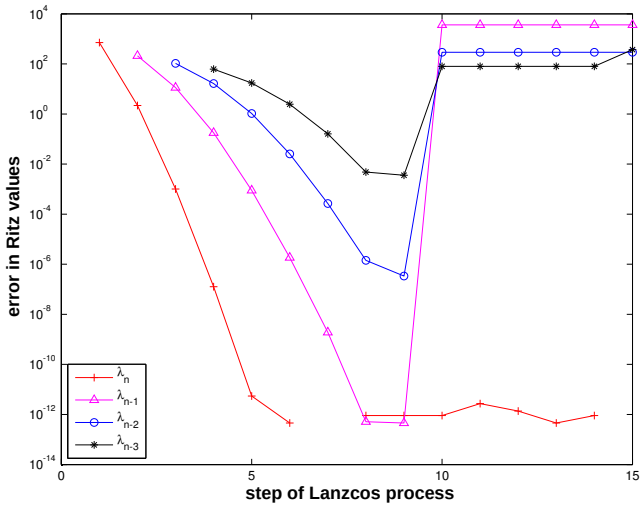


Fig. 363

Observation: same as in Ex. 9.4.4, linear convergence of Ritz values to eigenvalues.

However for $\mathbf{A} \in \mathbb{R}^{10,10}$, $a_{ij} = \min\{i,j\}$ good initial convergence, but sudden "jump" of Ritz values off eigenvalues!

Conjecture: Impact of roundoff errors, *cf.* Ex. 10.2.21

Example 9.4.14 (Impact of roundoff on Lanczos process)

$\mathbf{A} \in \mathbb{R}^{10,10}$, $a_{ij} = \min\{i,j\}$.

```
A = gallery('minij',10);
```

Computed by `[V,alpha,beta] = lanczos(A,n,ones(n,1));`, see Code 9.4.12:

$$\mathbf{T} = \begin{pmatrix}
38.500000 & 14.813845 & & & & & & & & \\
14.813845 & 9.642857 & 2.062955 & & & & & & & \\
 & 2.062955 & 2.720779 & 0.776284 & & & & & & \\
 & & 0.776284 & 1.336364 & 0.385013 & & & & & \\
 & & & 0.385013 & 0.826316 & 0.215431 & & & & \\
 & & & & 0.215431 & 0.582380 & 0.126781 & & & \\
 & & & & & 0.126781 & 0.446860 & 0.074650 & & \\
 & & & & & & 0.074650 & 0.363803 & 0.043121 & \\
 & & & & & & & 0.043121 & 3.820888 & 11.991094 \\
 & & & & & & & & 11.991094 & 41.254286
\end{pmatrix}$$

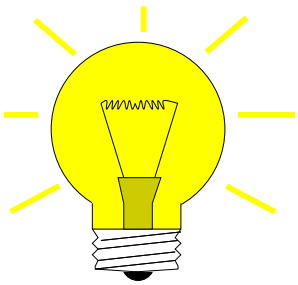
$$\sigma(\mathbf{A}) = \{0.255680,0.273787,0.307979,0.366209,0.465233,0.643104,1.000000,1.873023,5.048917,44.766069\}$$
$$\sigma(\mathbf{T}) = \{0.263867,0.303001,0.365376,0.465199,0.643104,1.000000,1.873023,5.048917,44.765976,44.766069\}$$

► Uncanny cluster of computed eigenvalues of $\mathbf{T}$ ("ghost eigenvalues", [**?**, Sect. 9.2.5])

$$\mathbf{V}^H\mathbf{V} = \begin{pmatrix}
1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000251 & 0.258801 & 0.883711 \\
0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000106 & 0.109470 & 0.373799 \\
0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000005 & 0.005373 & 0.018347 \\
0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000096 & 0.000328 \\
0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000001 & 0.000003 \\
0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 \\
0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 \\
0.000251 & 0.000106 & 0.000005 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 \\
0.258801 & 0.109470 & 0.005373 & 0.000096 & 0.000001 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 \\
0.883711 & 0.373799 & 0.018347 & 0.000328 & 0.000003 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000
\end{pmatrix}$$

► Loss of orthogonality of residual vectors due to roundoff
(compare: impact of roundoff on CG iteration, Ex. 10.2.21

l	$\sigma(\mathbf{T}_l)$									
1										38.500000
2									3.392123	44.750734
3								1.117692	4.979881	44.766064
4							0.597664	1.788008	5.048259	44.766069
5						0.415715	0.925441	1.870175	5.048916	44.766069
6					0.336507	0.588906	0.995299	1.872997	5.048917	44.766069
7				0.297303	0.431779	0.638542	0.999922	1.873023	5.048917	44.766069
8			0.276160	0.349724	0.462449	0.643016	1.000000	1.873023	5.048917	44.766069
9		0.276035	0.349451	0.462320	0.643006	1.000000	1.873023	3.821426	5.048917	44.766069
10	0.263867	0.303001	0.365376	0.465199	0.643104	1.000000	1.873023	5.048917	44.765976	44.766069



Idea:

✦ do not rely on orthogonality relations of Lemma 10.2.12

✦ use explicit Gram-Schmidt orthogonalization [**?**, Thm. 4.8], [**?**, Alg .6.1]

Details: inductive approach: given $\{\mathbf{v}_1,\ldots,\mathbf{v}_l\}$ ONB of $\mathcal{K}_l(\mathbf{A},\mathbf{z})$

$$\widetilde{\mathbf{v}}_{l+1} := \mathbf{A}\mathbf{v}_l - \sum_{j=1}^{l}(\mathbf{v}_j^H\mathbf{A}\mathbf{v}_l)\,\mathbf{v}_j\,,\quad \mathbf{v}_{l+1} := \frac{\widetilde{\mathbf{v}}_{l+1}}{\|\widetilde{\mathbf{v}}_{l+1}\|_2} \;\Rightarrow\; \mathbf{v}_{l+1}\perp\mathcal{K}_l(\mathbf{A},\mathbf{z})\,. \tag{9.4.15}$$

(Gram-Schmidt, *cf.* (10.2.11)) orthogonal

Arnoldi process: In step l:

$1\times$	$\mathbf{A}\times$vector
$l+1$	dot products
l	AXPY-operations
n	divisions

➢ Computational cost for l steps, if at most k non-zero entries in each row of $\mathbf{A}$: $O(nkl^2)$

MATLAB-code 9.4.16: Arnoldi process

```
function [V,H] = arnoldi(A,k,v0)
% Columns of V store orthonormal basis of Krylov spaces K_l(A,v_0).
% H returns Hessenberg matrix, see Lemma 9.4.17.
V = [v0/norm(v0)];
H = zeros(k+1,k);
for l=1:k
  vt = A*V(:,l); % "power iteration", next basis vector
  for j=1:l
    % Gram-Schmidt orthogonalization, cf. Sect. 9.3.4.1
    H(j,l) = dot(V(:,j),vt);
    vt = vt - H(j,l)*V(:,j);
  end
  H(l+1,l) = norm(vt);
  if (H(l+1,l) == 0), break; end % "theoretical" termination
  V = [V, vt/H(l+1,l)];
```

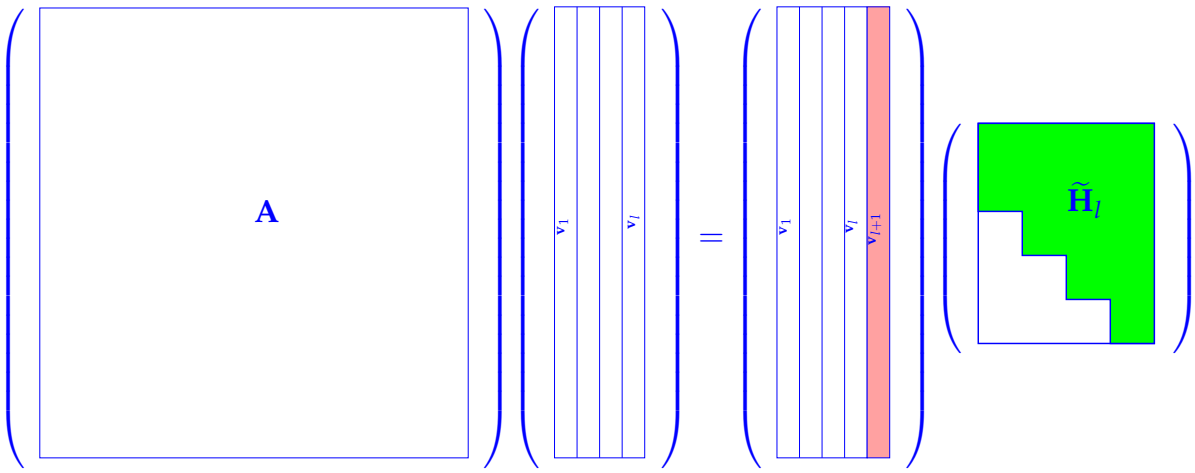
```
16 end
```

If it does not stop prematurely, the Arnoldi process of Code 9.4.16 will yield an *orthonormal basis* (ONB) of $\mathcal{K}_{k+1}(\mathbf{A},\mathbf{v}_0)$ for a **general** $\mathbf{A} \in \mathbb{C}^{n,n}$.

Algebraic view of the Arnoldi process of Code 9.4.16, meaning of output `H`:

$$\mathbf{V}_l = [\mathbf{v}_1,\ldots,\mathbf{v}_l]: \quad \mathbf{A}\mathbf{V}_l = \mathbf{V}_{l+1}\widetilde{\mathbf{H}}_l \quad , \quad \widetilde{\mathbf{H}}_l \in \mathbb{K}^{l+1,l} \text{ mit } \widetilde{h}_{ij} = \begin{cases} \mathbf{v}_i^H \mathbf{A}\mathbf{v}_j & \text{, if } i \le j \text{ ,} \\ \|\widetilde{\mathbf{v}}_i\|_2 & \text{, if } i = j+1 \text{ ,} \\ 0 & \text{else.} \end{cases}$$

➡ $\widetilde{\mathbf{H}}_l$ = non-square upper Hessenberg matrices



Translate Code 9.4.16 to matrix calculus:

Lemma 9.4.17. Theory of Arnoldi process

For the matrices $\mathbf{V}_l \in \mathbb{K}^{n,l}$, $\widetilde{\mathbf{H}}_l \in \mathbb{K}^{l+1,l}$ arising in the l-th step, $l \le n$, of the Arnoldi process holds

(i) $\mathbf{V}_l^H \mathbf{V}_l = \mathbf{I}$ *(unitary matrix),*

(ii) $\mathbf{A}\mathbf{V}_l = \mathbf{V}_{l+1}\widetilde{\mathbf{H}}_l$, $\widetilde{\mathbf{H}}_l$ *is non-square upper Hessenberg matrix,*

(iii) $\mathbf{V}_l^H \mathbf{A}\mathbf{V}_l = \mathbf{H}_l \in \mathbb{K}^{l,l}$, $h_{ij} = \widetilde{h}_{ij}$ *for* $1 \le i,j \le l$,

(iv) If $\mathbf{A} = \mathbf{A}^H$ *then* $\mathbf{H}_l$ *is tridiagonal (➢ Lanczos process)*

Proof. Direct from Gram-Schmidt orthogonalization and inspection of Code 9.4.16. □

Remark 9.4.18 (Arnoldi process and Ritz projection)

Interpretation of Lemma 9.4.17 (iii) & (i):

$\mathbf{H}_l\mathbf{x} = \lambda\mathbf{x}$ is a (generalized) Ritz projection of EVP $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$, *cf.* Section 9.3.4.2.

Eigenvalue approximation for general EVP $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$ by Arnoldi process:

MATLAB-code 9.4.19: Arnoldi eigenvalue approximation

```
function [dn,V,Ht] =
   arnoldieig(A,v0,k,tol)
n = size(A,1); V = [v0/norm(v0)];
H = zeros(1,0); dn = zeros(k,1);
for l=1:n
  d = dn;
  Ht = [Ht, zeros(l,1); zeros(1,l)];
  vt = A*V(:,l);
  for j=1:l
    Ht(j,l) = dot(V(:,j),vt);
    vt = vt - Ht(j,l)*V(:,j);
  end
  ev = sort(eig(Ht(1:l,1:l)));
  dn(1:min(l,k)) =
     ev(end:-1:end-min(l,k)+1);
  if (norm(d-dn) <
     tol*norm(dn))&\pnode{SPOWITx}&,
     break; end;
  Ht(l+1,l) = norm(vt);
  V = [V, vt/Ht(l+1,l)];
end
```

MATLAB-code 9.4.20: MATLAB-CODE Arnoldi eigenvalue approximation

```
function [dn,V,Ht] = arnoldieig(A,v0,k,tol)
n = size(A,1); V = [v0/norm(v0)];
H = zeros(1,0); dn = zeros(k,1);
for l=1:n
  d = dn;
  Ht = [Ht, zeros(l,1); zeros(1,l)];
  vt = A*V(:,l);
  for j=1:l
    Ht(j,l) = dot(V(:,j),vt);
    vt = vt - Ht(j,l)*V(:,j);
  end
  ev = sort(eig(Ht(1:l,1:l)));
```

Arnoldi process for computing the k largest (in modulus) eigenvalues of $\mathbf{A} \in \mathbb{C}^{n,n}$

1 $\mathbf{A}\times$vector per step (➢ attractive for sparse matrices)

However: required storage increases with number of steps, *cf.* situation with GMRES, Section 10.4.1.

Heuristic termination criterion

Example 9.4.21 (Stabilty of Arnoldi process)

$\mathbf{A} \in \mathbb{R}^{100,100}$, $a_{ij} = \min\{i,j\}$. `A = gallery('minij',100);`

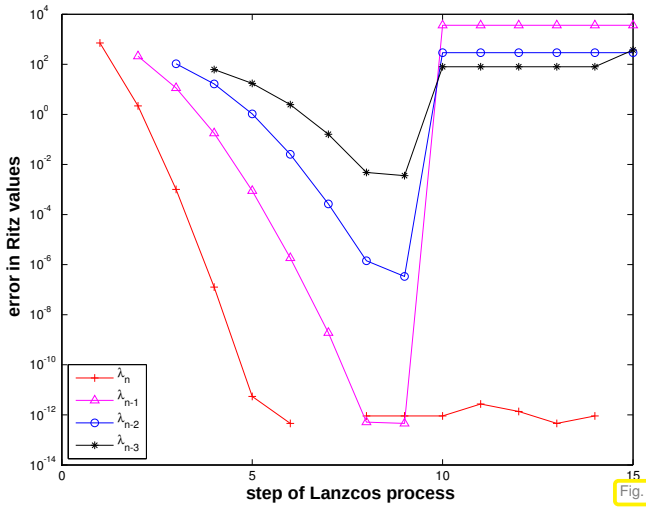


Fig. 364

Lanczos process: Ritz values

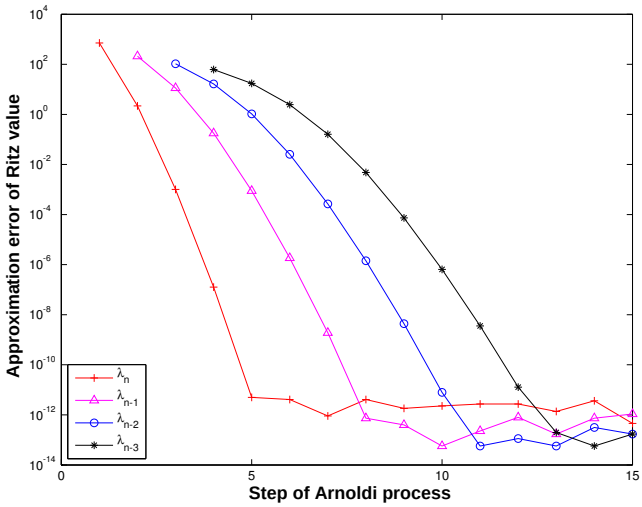


Fig. 365

Arnoldi process: Ritz values

Ritz values during Arnoldi process for `A = gallery('minij',10);` ↔ Ex. 9.4.13

l	$\sigma(\mathbf{H}_l)$									
1										38.500000
2									3.392123	44.750734
3								1.117692	4.979881	44.766064
4							0.597664	1.788008	5.048259	44.766069
5						0.415715	0.925441	1.870175	5.048916	44.766069
6					0.336507	0.588906	0.995299	1.872997	5.048917	44.766069
7				0.297303	0.431779	0.638542	0.999922	1.873023	5.048917	44.766069
8			0.276159	0.349722	0.462449	0.643016	1.000000	1.873023	5.048917	44.766069
9		0.263872	0.303009	0.365379	0.465199	0.643104	1.000000	1.873023	5.048917	44.766069
10	0.255680	0.273787	0.307979	0.366209	0.465233	0.643104	1.000000	1.873023	5.048917	44.766069

Observation: (almost perfect approximation of spectrum of $\mathbf{A}$)

For the above examples both the Arnoldi process and the Lanczos process are *algebraically equivalent*, because they are applied to a symmetric matrix $\mathbf{A} = \mathbf{A}^T$. However, they behave strikingly differently, which indicates that they are *not numerically equivalent*.

The Arnoldi process is much less affected by roundoff than the Lanczos process, because it does not take for granted orthogonality of the "residual vector sequence". Hence, the Arnoldi process enjoys superior numerical stability (→ **??**, Def. 1.5.85) compared to the Lanczos process.

Example 9.4.22 (Eigenvalue computation with Arnoldi process)

Eigenvalue approximation from Arnoldi process for *non-symmetric* $\mathbf{A}$, initial vector `ones(100,1);`

MATLAB-code 9.4.23:

```
n=100;
M=full(gallery('tridiag',-0.5*ones(n-1,1),2*ones(n,1),-1.5*ones(n-1,1)));
A=M*diag(1:n)*inv(M);
```

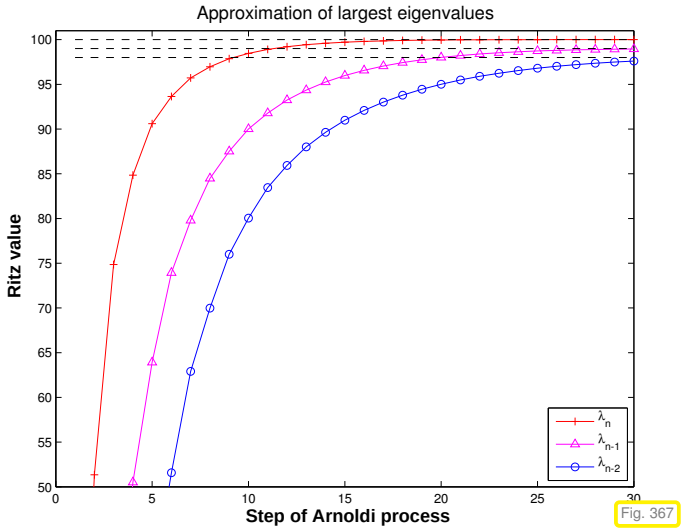


Fig. 366

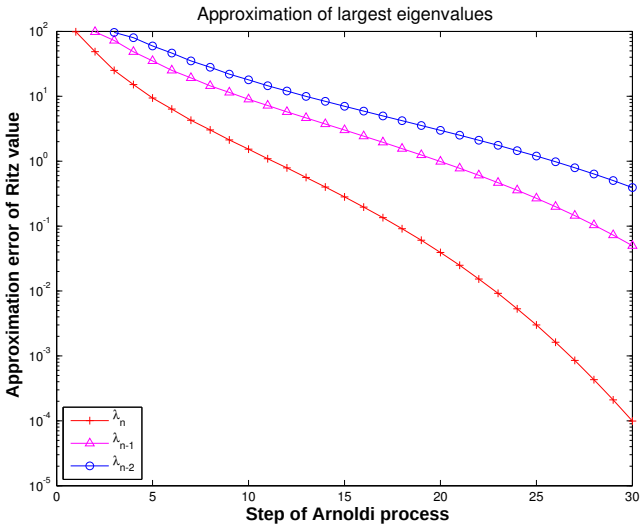


Fig. 367

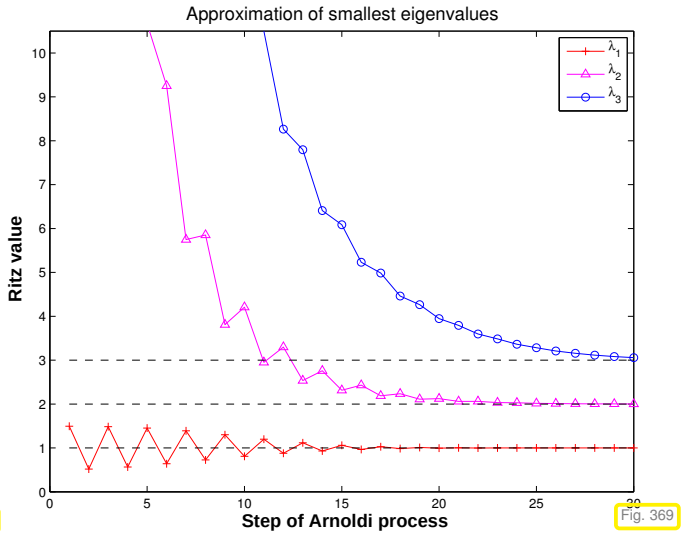


Fig. 368

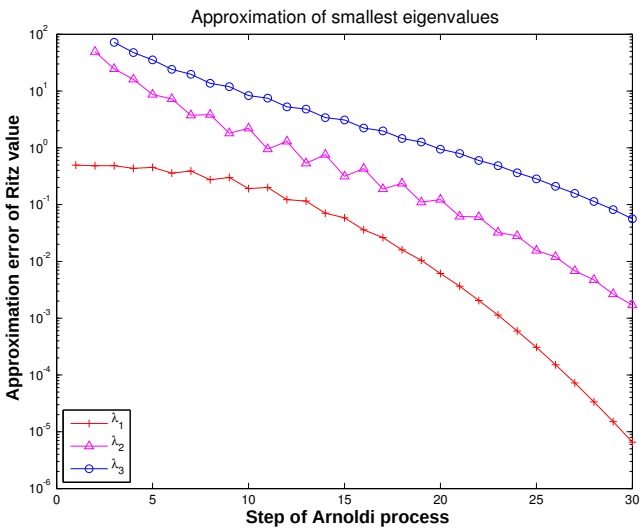


Fig. 369

Observation: "vaguely linear" convergence of largest and smallest eigenvalues, *cf.* Ex. 9.4.4.

Krylov subspace iteration methods (= Arnoldi process, Lanczos process) attractive for computing *a few* of the largest/smallest eigenvalues and associated eigenvectors of *large sparse matrices*.

Remark 9.4.24 (Krylov subspace methods for generalized EVP)

Adaptation of Krylov subspace iterative eigensolvers to generalized EVP: $\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x}$, $\mathbf{B}$ s.p.d.: replace Euclidean inner product with "$\mathbf{B}$-inner product" $(\mathbf{x},\mathbf{y}) \mapsto \mathbf{x}^H\mathbf{B}\mathbf{y}$.

MATLAB-functions:

`d = eigs(A,k,sigma)`	: k largest/smallest eigenvalues of $\mathbf{A}$
`d = eigs(A,B,k,sigma)`	: k largest/smallest eigenvalues for generalized EVP $\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x}$,$\mathbf{B}$ s.p.d.
`d = eigs(Afun,n,k)`	: `Afun` = handle to function providing matrix×vector for $\mathbf{A}/\mathbf{A}^{-1}/\mathbf{A} - \alpha\mathbf{I}/(\mathbf{A} - \alpha\mathbf{B})^{-1}$. (Use flags to tell `eigs` about special properties of matrix behind `Afun`.)

`eigs` just calls routines of the open source ARPACK numerical library.

Chapter 10

Krylov Methods for Linear Systems of Equations

Supplementary reading. There is a wealth of literature on iterative methods for the solution of linear systems of equations: The two books [**?**] and [**?**] offer a comprehensive treatment of the topic (the latter is available online for ETH students and staff).

Concise presentations can be found in [**?**, Ch. 4] and [**?**, Ch. 13].

Learning outcomes:

- Understanding when and why iterative solution of linear systems of equations may be preferred to direct solvers based on Gaussian elimination.
-

= A class of iterative methods ($\rightarrow$ Section 8.1) for approximate solution of large linear systems of equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{K}^{n,n}$.

BUT, we have reliable *direct* methods (Gauss elimination $\rightarrow$ Section 2.3, LU-factorization $\rightarrow$ § 2.3.30, QR-factorization $\rightarrow$ **??**) that provide an (apart from roundoff errors) exact solution with a *finite* number of elementary operations!

Alas, direct elimination may not be feasible, or may be grossly inefficient, because

- it may be too expensive (e.g. for $\mathbf{A}$ too large, sparse), $\rightarrow$ (2.3.25),
- inevitable fill-in may exhaust main memory,
- the system matrix may be available only as procedure `y=evalA(x)` $\leftrightarrow \mathbf{y} = \mathbf{Ax}$

Contents

10.1 Descent Methods [?, Sect. 4.3.3]

Focus:

Linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$, $n \in \mathbb{N}$ given,

with symmetric positive definite (s.p.d., → Def. 1.1.8) system matrix $\mathbf{A}$

➡

$\mathbf{A}$-inner product $(\mathbf{x}, \mathbf{y}) \mapsto \mathbf{x}^\top \mathbf{A}\mathbf{y}$ ⇒ "$\mathbf{A}$-geometry"

Definition 10.1.1. Energy norm → [?, Def. 9.1]

A s.p.d. matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ induces an energy norm

$$\|\mathbf{x}\|_A := (\mathbf{x}^\top \mathbf{A}\mathbf{x})^{1/2}, \quad \mathbf{x} \in \mathbb{R}^n .$$

Remark 10.1.2 (Krylov methods for complex s.p.d. system matrices)

In this chapter, for the sake of simplicity, we restrict ourselves to $\mathbb{K} = \mathbb{R}$.

However, the (conjugate) gradient methods introduced below also work for LSE $\mathbf{A}\mathbf{x} = \mathbf{b}$ with $\mathbf{A} \in \mathbb{C}^{n,n}$, $\mathbf{A} = \mathbf{A}^H$ s.p.d. when $^\top$ is replaced with ^{H} (Hermitian transposed). Then, all theoretical statements remain valid unaltered for $\mathbb{K} = \mathbb{C}$.

10.1.1 Quadratic minimization context

Lemma 10.1.3. S.p.d. LSE and quadratic minimization problem [?, (13.37)]

A LSE with $\mathbf{A} \in \mathbb{R}^{n,n}$ s.p.d. and $\mathbf{b} \in \mathbb{R}^n$ is equivalent to a minimization problem:

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad \Leftrightarrow \quad \mathbf{x} = \arg\min_{\mathbf{y} \in \mathbb{R}^n} J(\mathbf{y}), \quad J(\mathbf{y}) := \tfrac{1}{2}\mathbf{y}^\top \mathbf{A}\mathbf{y} - \mathbf{b}^\top \mathbf{y} . \tag{10.1.4}$$

A quadratic functional

Proof. If $\mathbf{x}^* := \mathbf{A}^{-1}\mathbf{b}$ a straightforward computation using $\mathbf{A} = \mathbf{A}^T$ shows

$$\begin{aligned} J(\mathbf{x}) - J(\mathbf{x}^*) &= \tfrac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} - \tfrac{1}{2}(\mathbf{x}^*)^T\mathbf{A}\mathbf{x}^* + \mathbf{b}^T\mathbf{x}^* \\ &\overset{\mathbf{b}=\mathbf{A}\mathbf{x}^*}{=} \tfrac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - (\mathbf{x}^*)^T\mathbf{A}\mathbf{x} + \tfrac{1}{2}(\mathbf{x}^*)^T\mathbf{A}\mathbf{x}^* \\ &= \tfrac{1}{2}\|\mathbf{x} - \mathbf{x}^*\|_A^2 . \end{aligned} \tag{10.1.5}$$

Then the assertion follows from the properties of the energy norm.

□

Example 10.1.6 (Quadratic functional in 2D)

Plot of J from (10.1.4) for $\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

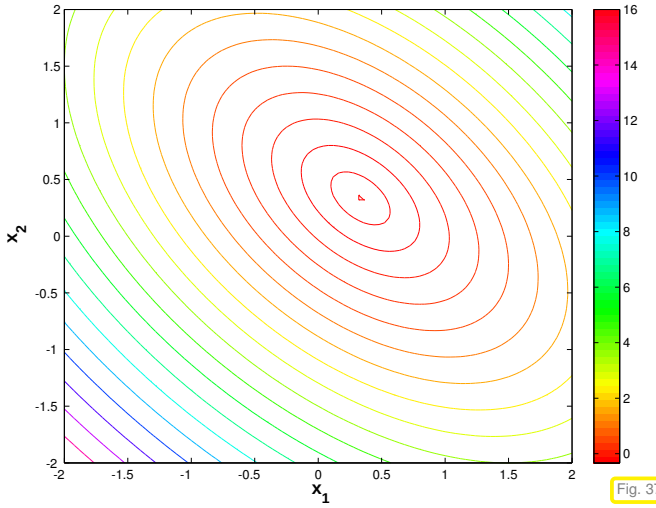


Fig. 370

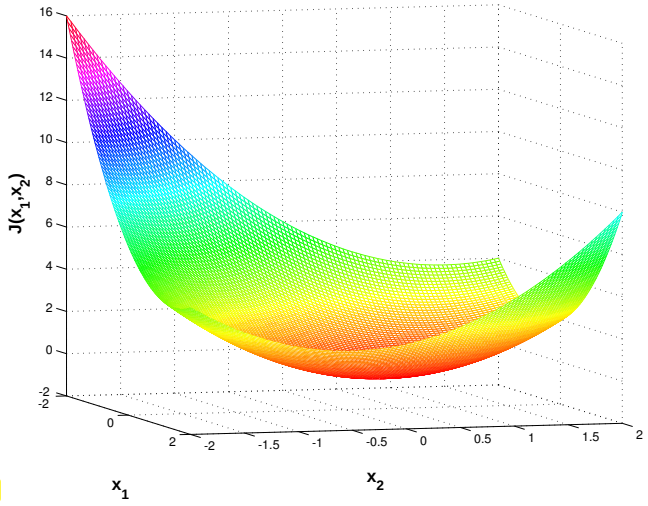
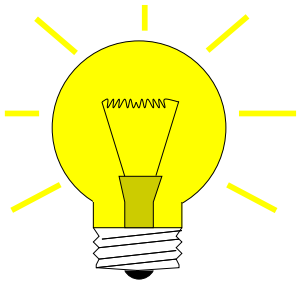


Fig. 371

Level lines of quadratic functionals with s.p.d. $\mathbf{A}$ are (hyper)ellipses



Algorithmic idea: (Lemma 10.1.3 ➢) Solve $\mathbf{Ax} = \mathbf{b}$ iteratively by successive solution of *simpler* minimization problems

10.1.2 Abstract steepest descent

Task:	Given continuously differentiable $F : D \subset \mathbb{R}^n \mapsto \mathbb{R}$,
	find minimizer $\mathbf{x}^* \in D$: $\mathbf{x}^* = \operatorname{argmin}_{\mathbf{x} \in D} F(\mathbf{x})$

Note that a minimizer need not exist, if F is not bounded from below (e.g., $F(x) = x^3$, $x \in \mathbb{R}$, or $F(x) = \log x$, $x > 0$), or if D is open (e.g., $F(x) = \sqrt{x}$, $x > 0$).

The existence of a minimizer is guaranteed if F is bounded from below and D is closed ($\rightarrow$ Analysis).

The most natural iteration:

(10.1.7) Steepest descent (*ger.:* steilster Abstieg)

Initial guess $\mathbf{x}^{(0)} \in D$, $k = 0$
repeat
$\mathbf{d}_k := -\mathbf{grad}\, F(\mathbf{x}^{(k)})$
$t^* := \operatorname{argmin}_{t\in\mathbb{R}} F(\mathbf{x}^{(k)} + t\mathbf{d}_k)$ (line search)
$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + t^*\mathbf{d}_k$
$k := k + 1$
until $\left(\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\| \le \tau_{\text{rel}} \left\|\mathbf{x}^{(k)}\right\|$ **or** $\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\| \le \tau_{\text{abs}} \right)$

- $\mathbf{d}_k \hat{=}$ *direction of steepest descent*
- linear search $\hat{=}$ 1D minimization: use Newton's method (→ Section 8.3.2.1) on derivative
- correction based a posteriori termination criterion, see Section 8.1.2 for a discussion.
($\tau \hat{=}$ prescribed tolerance)

The gradient (→ [**?**, Kapitel 7])

$$\mathbf{grad}\, F(\mathbf{x}) = \begin{bmatrix} \frac{\partial F}{\partial x_i} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n \qquad (10.1.8)$$

provides the direction of local steepest ascent/descent of F

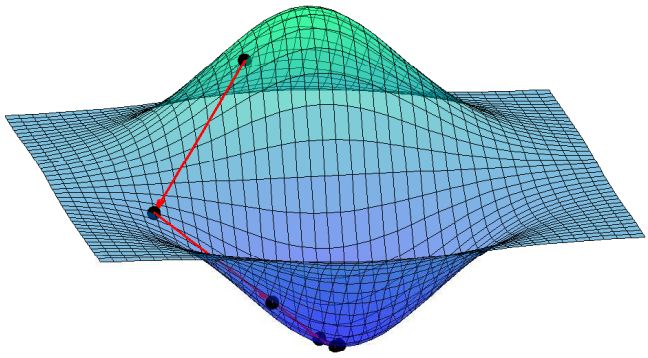


Fig. 372

Of course this very algorithm can encounter plenty of difficulties:

- iteration may get stuck in a *local minimum*,
- iteration may diverge or lead out of D,
- line search may not be feasible.

10.1.3 Gradient method for s.p.d. linear system of equations

However, for the quadratic minimization problem (10.1.4) § 10.1.7 will converge:

("Geometric intuition", see Fig. 370: quadratic functional J with s.p.d. $\mathbf{A}$ has unique global minimum, $\mathbf{grad}\, J \neq 0$ away from minimum, pointing towards it.)

Adaptation: steepest descent algorithm § 10.1.7 for quadratic minimization problem (10.1.4), see [**?**, Sect. 7.2.4]:

$$F(\mathbf{x}) := J(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^\top\mathbf{A}\mathbf{x} - \mathbf{b}^\top\mathbf{x} \quad \Rightarrow \quad \mathbf{grad}\, J(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}\,. \qquad (10.1.9)$$

This follows from $\mathbf{A} = \mathbf{A}^\top$, the componentwise expression

$$J(\mathbf{x}) = \tfrac{1}{2}\sum_{i,j=1}^{n} a_{ij}x_ix_j - \sum_{i=1}^{n} b_ix_i$$

and the definition (10.1.8) of the gradient.

➢ For the descent direction in § 10.1.7 applied to the minimization of J from (10.1.4) holds

$$\mathbf{d}_k = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} =: \mathbf{r}_k \quad \text{the residual } (\rightarrow \text{Def. 2.4.1}) \text{ for } \mathbf{x}^{(k-1)}.$$

§ 10.1.7 for $F = J$ from (10.1.4): function to be minimized in line search step:

$$\varphi(t) := J(\mathbf{x}^{(k)} + t\mathbf{d}_k) = J(\mathbf{x}^{(k)}) + t\mathbf{d}_k^\top(\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b}) + \tfrac{1}{2}t^2\mathbf{d}_k^\top\mathbf{A}\mathbf{d}_k \quad \rightarrowtail \text{ a parabola !}.$$

$$\frac{d\varphi}{dt}(t^*) = 0 \quad \Leftrightarrow \quad \boxed{t^* = \frac{\mathbf{d}_k^\top\mathbf{d}_k}{\mathbf{d}_k^\top\mathbf{A}\mathbf{d}_k}} \quad \text{(unique minimizer)}. \tag{10.1.10}$$

Note: $\mathbf{d}_k = 0 \quad \Leftrightarrow \quad \mathbf{A}\mathbf{x}^{(k)} = \mathbf{b}$ (solution found !)

Note: $\mathbf{A}$ s.p.d. ($\rightarrow$ Def. 1.1.8) $\quad \Rightarrow \quad \mathbf{d}_k^\top\mathbf{A}\mathbf{d}_k > 0$, if $\mathbf{d}_k \neq 0$

◥ $\varphi(t)$ is a parabola that is bounded from below (upward opening)

Based on (10.1.9) and (10.1.10) we obtain the following steepest descent method for the minimization problem (10.1.4):

Steepest descent iteration = gradient method for LSE $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$ s.p.d., $\mathbf{b} \in \mathbb{R}^n$:

(10.1.11) Gradient method for s.p.d. LSE

Initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$, $k = 0$
$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
repeat
$t^* := \dfrac{\mathbf{r}_k^\top\mathbf{r}_k}{\mathbf{r}_k^\top\mathbf{A}\mathbf{r}_k}$
$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + t^*\mathbf{r}_k$
$\mathbf{r}_{k+1} := \mathbf{r}_k - t^*\mathbf{A}\mathbf{r}_k$
$k := k + 1$
until $\left(\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\| \leq \tau_{\text{rel}}\left\|\mathbf{x}^{(k)}\right\| \text{ or } \left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\| \leq \tau_{\text{abs}}\right)$

MATLAB-code 10.1.12: gradient method for $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A}$ s.p.d.

```
1  function x =
     gradit(A,b,x,rtol,atol,maxit)
2  r = b-A*x; % residual → Def. 2.4.1
3  for k=1:maxit
4    p = A*r;
5    ts = (r'*r)/(r'*p); % cf.
       (10.1.10)
6    x = x + ts*r;
7    cn = (abs(ts)*norm(r); % norm of
       correction
8    if ((cn < tol*norm(x)) ||
9        (cn < atol))
10     return; end
11   r = r - ts*p; %
12 end
```

Recursion for residuals, see Line 11 of Code 10.1.12:

$$\mathbf{r}_{k+1} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k+1)} = \mathbf{b} - \mathbf{A}(\mathbf{x}^{(k)} + t^*\mathbf{r}_k) = \mathbf{r}_k - t^*\mathbf{A}\mathbf{r}_k\,. \tag{10.1.13}$$

One step of gradient method involves

- ✦ *A single* matrix×vector product with $\mathbf{A}$,
- ✦ 2 AXPY-operations ($\rightarrow$ Section 1.3.2) on vectors of length n,
- ✦ 2 dot products in $\mathbb{R}^n$.

Computational cost (per step) = cost(matrix×vector) + $O(n)$

➢ If $\mathbf{A} \in \mathbb{R}^{n,n}$ is a sparse matrix ($\rightarrow$ **??**) with "$O(n)$ nonzero entries", and the data structures allow to perform the matrix×vector product with a computational effort $O(n)$, then a single step of the gradient method costs $O(n)$ elementary operations.

➢ Gradient method of § 10.1.11 only needs $\mathbf{A}\times$vector in procedural form `y = evalA(x)`.

10.1.4 Convergence of the gradient method

Example 10.1.14 (Gradient method in 2D)

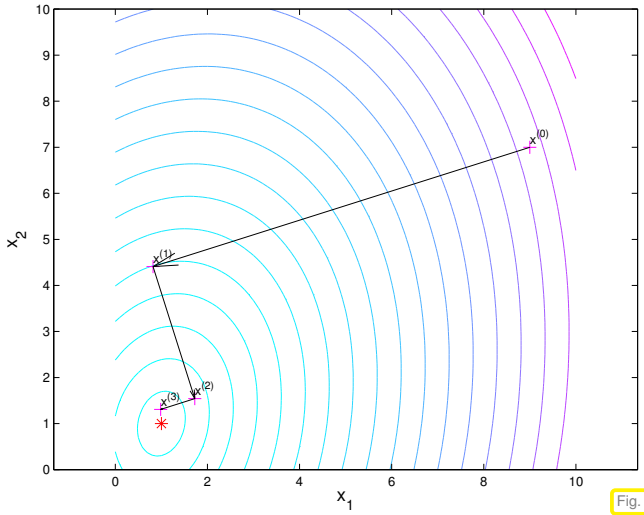
S.p.d. matrices $\in \mathbb{R}^{2,2}$:

$$\mathbf{A}_1 = \begin{bmatrix} 1.9412 & -0.2353 \\ -0.2353 & 1.0588 \end{bmatrix} \quad , \quad \mathbf{A}_2 = \begin{bmatrix} 7.5353 & -1.8588 \\ -1.8588 & 0.5647 \end{bmatrix}$$

Eigenvalues: $\sigma(\mathbf{A}_1) = \{1,2\}$, $\sigma(\mathbf{A}_2) = \{0.1,8\}$

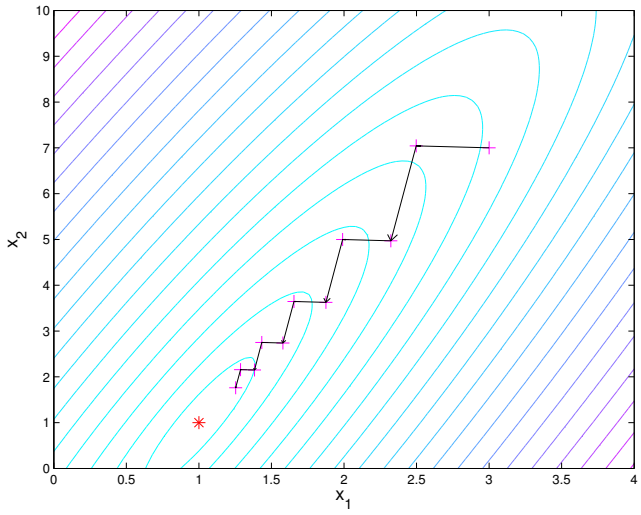
✎ notation: spectrum of a matrix $\in \mathbb{K}^{n,n}$ $\sigma(\mathbf{M}) := \{\lambda \in \mathbb{C}: \lambda \text{ is eigenvalue of } \mathbf{M}\}$

Fig. 373



iterates of § 10.1.11 for $\mathbf{A}_1$

Fig. 374



iterates of § 10.1.11 for $\mathbf{A}_2$

Recall theorem on principal axis transformation: every real *symmetric* matrix can be diagonalized by *orthogonal* similarity transformations, see Cor. 9.1.9, [**?**, Thm. 7.8], [**?**, Satz 9.15],

$$\mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n} \quad \Rightarrow \quad \exists \mathbf{Q} \in \mathbb{R}^{n,n} \text{ orthogonal: } \mathbf{A} = \mathbf{Q}\mathbf{D}\mathbf{Q}^\top \quad , \quad \mathbf{D} = \operatorname{diag}(d_1,\ldots,d_n) \in \mathbb{R}^{n,n} \text{ diagonal .} \tag{10.1.15}$$

$$J(\mathbf{Q}\widehat{\mathbf{y}}) = \tfrac{1}{2}\widehat{\mathbf{y}}^\top \mathbf{D}\widehat{\mathbf{y}} - \underbrace{(\mathbf{Q}^\top \mathbf{b})^\top}_{=:\widehat{\mathbf{b}}^\top}\widehat{\mathbf{y}} = \tfrac{1}{2}\sum_{i=1}^{n} d_i \widehat{y}_i^2 - \widehat{b}_i \widehat{y}_i \, .$$

Hence, a rigid transformation (rotation, reflection) maps the level surfaces of J from (10.1.4) to ellipses with principal axes d_i. As $\mathbf{A}$ s.p.d. $d_i > 0$ is guaranteed.

Observations:

- Larger spread of spectrum leads to more elongated ellipses as level lines ➢ slower convergence of gradient method, see Fig. 374.

- Orthogonality of successive residuals $\mathbf{r}_k$, $\mathbf{r}_{k+1}$.

Clear from definition of § 10.1.11:

$$\mathbf{r}_k^\top \mathbf{r}_{k+1} = \mathbf{r}_k^\top \mathbf{r}_k - \mathbf{r}_k^\top \frac{\mathbf{r}_k^\top \mathbf{r}_k}{\mathbf{r}_k^\top \mathbf{A}\mathbf{r}_k} \mathbf{A}\mathbf{r}_k = 0 \,. \tag{10.1.16}$$

Example 10.1.17 (Convergence of gradient method)

Convergence of gradient method for diagonal matrices, $\mathbf{x}^* = [1,\ldots,1]^\top$, $\mathbf{x}^{(0)} = 0$:

```
d = 1:0.01:2;   A1 = diag(d);
d = 1:0.1:11;   A2 = diag(d);
d = 1:1:101;    A3 = diag(d);
```

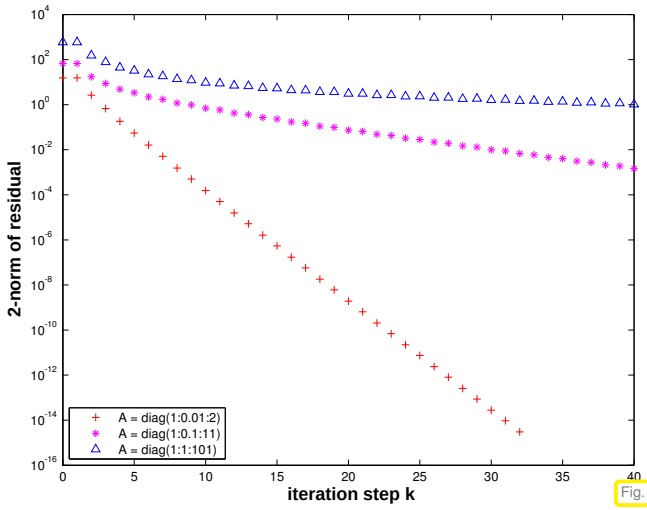


Fig. 375

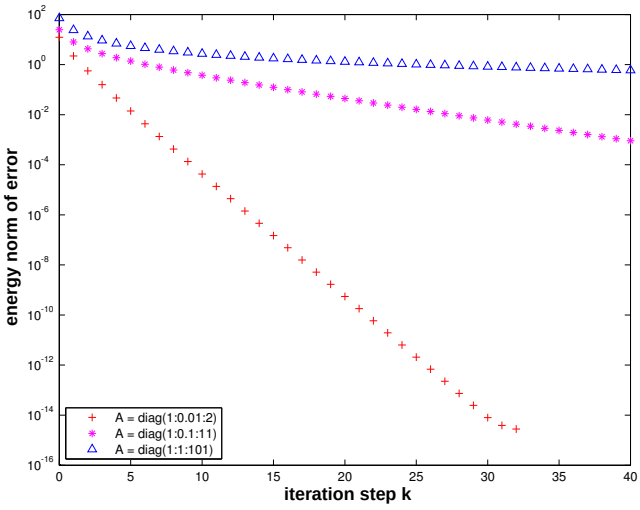


Fig. 376

Note: To study convergence it is *sufficient to consider diagonal matrices*, because

1. (10.1.15): for every $\mathbf{A} \in \mathbb{R}^{n,n}$ with $\mathbf{A}^\top = \mathbf{A}$ there is an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{n.n}$ such that $\mathbf{A} = \mathbf{Q}^\top \mathbf{D}\mathbf{Q}$ with a diagonal matrix $\mathbf{D}$ (principal axis transformation), → Cor. 9.1.9, [**?**, Thm. 7.8], [**?**, Satz 9.15],
2. when applying the gradient method § 10.1.11 to both $\mathbf{A}\mathbf{x} = \mathbf{b}$ and $\mathbf{D}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}} := \mathbf{Q}\mathbf{b}$, then the iterates $\mathbf{x}^{(k)}$ and $\widetilde{\mathbf{x}}^{(k)}$ are related by $\mathbf{Q}\mathbf{x}^{(k)} = \widetilde{\mathbf{x}}^{(k)}$.

With $\widetilde{\mathbf{r}}_k := \mathbf{Q}\mathbf{r}_k$, $\widetilde{\mathbf{x}}^{(k)} := \mathbf{Q}\mathbf{x}^{(k)}$, using $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$:

Initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$, $k = 0$
$\mathbf{r}_0 := \mathbf{b} - \mathbf{Q}^\top\mathbf{D}\mathbf{Q}\mathbf{x}^{(0)}$
repeat
$t^* := \dfrac{\mathbf{r}_k^\top \mathbf{Q}^\top \mathbf{Q}\mathbf{r}_k}{\mathbf{r}_k^\top \mathbf{Q}^\top \mathbf{D}\mathbf{Q}\mathbf{r}_k}$
$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + t^*\mathbf{r}_k$
$\mathbf{r}_{k+1} := \mathbf{r}_k - t^*\mathbf{Q}^\top\mathbf{D}\mathbf{Q}\mathbf{r}_k$
$k := k+1$
until $\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\| \le \tau \left\|\mathbf{x}^{(k)}\right\|$

Initial guess $\widetilde{\mathbf{x}}^{(0)} \in \mathbb{R}^n$, $k = 0$
$\widetilde{\mathbf{r}}_0 := \widetilde{\mathbf{b}} - \mathbf{D}\widetilde{\mathbf{x}}^{(0)}$
repeat
$t^* := \dfrac{\widetilde{\mathbf{r}}_k^\top \widetilde{\mathbf{r}}_k}{\widetilde{\mathbf{r}}_k^\top \mathbf{D}\widetilde{\mathbf{r}}_k}$
$\widetilde{\mathbf{x}}^{(k+1)} := \widetilde{\mathbf{x}}^{(k)} + t^*\widetilde{\mathbf{r}}_k$
$\widetilde{\mathbf{r}}_{k+1} := \widetilde{\mathbf{r}}_k - t^*\mathbf{D}\widetilde{\mathbf{r}}_k$
$k := k+1$
until $\left\|\widetilde{\mathbf{x}}^{(k)} - \widetilde{\mathbf{x}}^{(k-1)}\right\| \le \tau \left\|\widetilde{\mathbf{x}}^{(k)}\right\|$

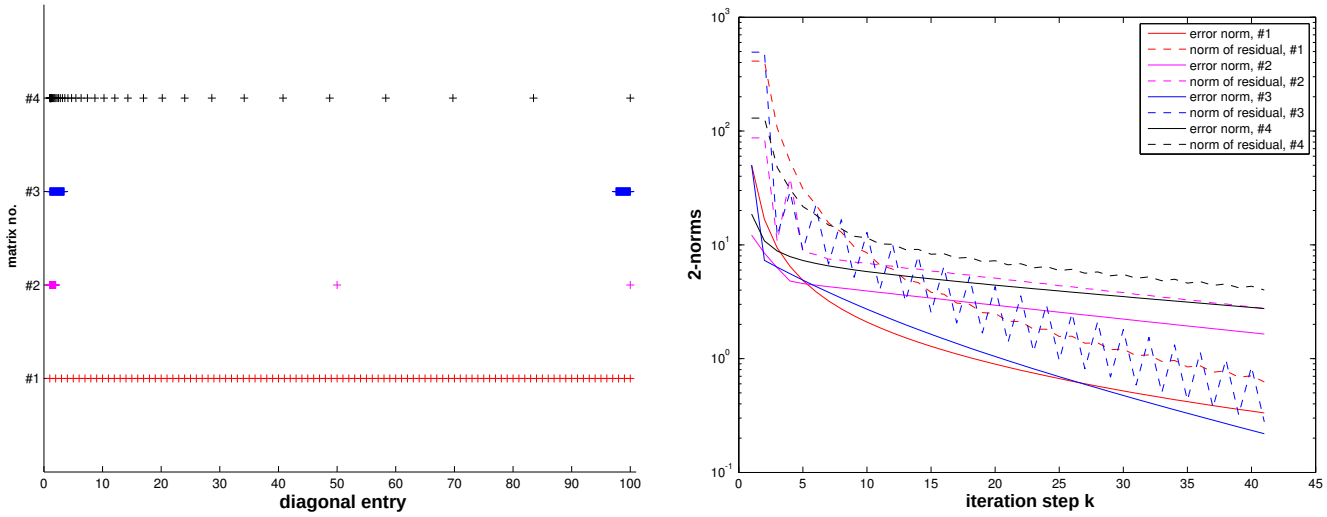
Observation:

- ✦ linear convergence (→ Def. 8.1.9), see also Rem. 8.1.13
- ✦ rate of convergence increases (↔ speed of convergence decreases) with spread of spectrum of $\mathbf{A}$

Impact of distribution of diagonal entries (↔ eigenvalues) of (diagonal matrix) $\mathbf{A}$
($\mathbf{b} = \mathbf{x}^* = 0$, `x0 = cos((1:n)');`)

Test matrix #1: `A=diag(d); d = (1:100);`
Test matrix #2: `A=diag(d); d = [1+(0:97)/97 , 50 , 100];`
Test matrix #3: `A=diag(d); d = [1+(0:49)*0.05, 100-(0:49)*0.05];`
Test matrix #4: eigenvalues exponentially dense at 1

Fig. 377



Observation: Matrices #1, #2 & #4 ➢ little impact of distribution of eigenvalues on *asymptotic* convergence (exception: matrix #2)

Theory [**?**, Sect. 9.2.2], [**?**, Sect.7.2.4]:

Theorem 10.1.18. Convergence of gradient method/steepest descent

The iterates of the gradient method of § 10.1.11 satisfy

$$\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\|_A \leq L\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|_A , \quad L := \frac{\operatorname{cond}_2(\mathbf{A}) - 1}{\operatorname{cond}_2(\mathbf{A}) + 1} ,$$

that is, the iteration converges at least linearly (→ Def. 8.1.9) w.r.t. energy norm (→ Def. 10.1.1).

✎ notation: $\operatorname{cond}_2(\mathbf{A})$ ≙ condition number (→ Def. 2.2.12) of $\mathbf{A}$ induced by 2-norm

Remark 10.1.19 (2-norm from eigenvalues → [?, Sect. 10.6], [?, Sect. 7.4])

$$\mathbf{A} = \mathbf{A}^\top \quad \Rightarrow \quad \begin{aligned} \|\mathbf{A}\|_2 &= \max(|\sigma(\mathbf{A})|) , \\ \left\|\mathbf{A}^{-1}\right\|_2 &= \min(|\sigma(\mathbf{A})|)^{-1} \end{aligned} \quad \text{, if } \mathbf{A} \text{ regular.} \tag{10.1.20}$$

$$\mathbf{A} = \mathbf{A}^\top \quad \Rightarrow \quad \operatorname{cond}_2(\mathbf{A}) = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})} , \quad \text{where} \quad \begin{aligned} \lambda_{\max}(\mathbf{A}) &:= \max(|\sigma(\mathbf{A})|) , \\ \lambda_{\min}(\mathbf{A}) &:= \min(|\sigma(\mathbf{A})|) . \end{aligned} \tag{10.1.21}$$

✎ other notation $\kappa(\mathbf{A}) := \dfrac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$ $\hat{=}$ spectral condition number of $\mathbf{A}$

(for general $\mathbf{A}$: $\lambda_{\max}(\mathbf{A})/\lambda_{\min}(\mathbf{A})$ largest/smallest eigenvalue *in modulus*)

These results are an immediate consequence of the fact that

$$\forall \mathbf{A} \in \mathbb{R}^{n,n},\ \mathbf{A}^\top = \mathbf{A}\ \ \exists \mathbf{U} \in \mathbb{R}^{n,n},\ \mathbf{U}^{-1} = \mathbf{U}^\top:\ \ \mathbf{U}^\top \mathbf{A}\mathbf{U} \text{ is diagonal,}$$

see (10.1.15), Cor. 9.1.9, [**?**, Thm. 7.8], [**?**, Satz 9.15].

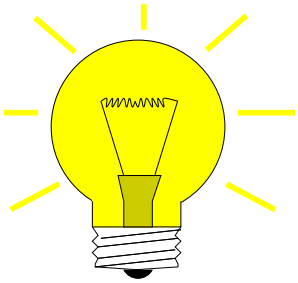
Please note that for general regular $\mathbf{M} \in \mathbb{R}^{n,n}$ we *cannot* expect $\operatorname{cond}_2(\mathbf{M}) = \kappa(\mathbf{M})$.

10.2 Conjugate gradient method (CG) [?, Ch. 9], [?, Sect. 13.4], [?, Sect. 4.3.4]

Again we consider a linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ with s.p.d. ($\to$ Def. 1.1.8) system matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ and given $\mathbf{b} \in \mathbb{R}^n$.

Liability of gradient method of Section 10.1.3:	NO MEMORY

1D line search in § 10.1.11 is oblivious of former line searches, which rules out reuse of information gained in previous steps of the iteration. This is a typical drawback of 1-point iterative methods.



Idea: Replace linear search with subspace correction

Given:
- initial guess $\mathbf{x}^{(0)}$
- nested subspaces $U_1 \subset U_2 \subset U_3 \subset \cdots \subset U_n = \mathbb{R}^n$, $\dim U_k = k$

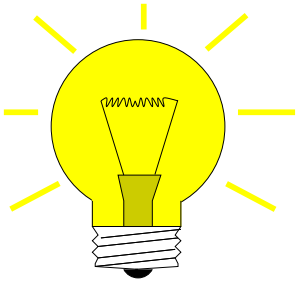
$$\mathbf{x}^{(k)} := \operatorname*{argmin}_{\mathbf{x} \in U_k + \mathbf{x}^{(0)}} J(x)\,, \tag{10.2.1}$$

quadratic functional from (10.1.4)

Note: Once the subspaces U_k and $\mathbf{x}^{(0)}$ are fixed, the iteration (10.2.1) is well defined, because $J_{|U_k+\mathbf{x}^{(0)}}$ always possess a unique minimizer.

Obvious (from Lemma 10.1.3): $\mathbf{x}^{(n)} = \mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$

Thanks to (10.1.5), definition (10.2.1) ensures: $\left\|\mathbf{x}^{(k+1)} - \mathbf{x}^*\right\|_A \leq \left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|_A$



How to find suitable subspaces U_k **?**

Idea: $U_{k+1} \leftarrow U_k +$ " local steepest descent direction"

given by $-\mathbf{grad}\, J(\mathbf{x}^{(k)}) = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} = \mathbf{r}_k$ (residual $\to$ Def. 2.4.1)

$$U_{k+1} = \operatorname{Span}\{U_k, \mathbf{r}_k\} \quad , \quad \mathbf{x}^{(k)} \text{ from (10.2.1).} \tag{10.2.2}$$

Obvious: $\mathbf{r}_k = 0 \quad \Rightarrow \quad \mathbf{x}^{(k)} = \mathbf{x}^* := \mathbf{A}^{-1}\mathbf{b}$ done ✔

Lemma 10.2.3. $\mathbf{r}_k \perp U_k$

With $\mathbf{x}^{(k)}$ according to (10.2.1), *U_k from* (10.2.2) *the residual $\mathbf{r}_k := \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ satisfies*

$$\mathbf{r}_k^\top \mathbf{u} = 0 \quad \forall \mathbf{u} \in U_k \quad (\text{"}\mathbf{r}_k \perp U_k\text{"}).$$

Geometric consideration: since $\mathbf{x}^{(k)}$ is the minimizer of J over the affine space $U_k + \mathbf{x}^{(0)}$, the projection of the steepest descent direction $\mathbf{grad}\, J(\mathbf{x}^{(k)})$ onto U_k has to vanish:

$$\mathbf{x}^{(k)} := \operatorname*{argmin}_{\mathbf{x} \in U_k + \mathbf{x}^{(0)}} J(x) \quad \Rightarrow \quad \mathbf{grad}\, J(\mathbf{x}^{(k)}) \perp U_k \,. \tag{10.2.4}$$

Proof. Consider

$$\psi(t) = J(\mathbf{x}^{(k)} + t\mathbf{u}) \,, \quad \mathbf{u} \in U_k, \quad t \in \mathbb{R} \,.$$

By (10.2.1), $t \mapsto \psi(t)$ has a global minimum in $t = 0$, which implies

$$\frac{d\psi}{dt}(0) = \mathbf{grad}\, J(\mathbf{x}^{(k)})^\top \mathbf{u} = (\mathbf{A}\mathbf{x}^{(k)} - \mathbf{b})^\top \mathbf{u} = 0 \,.$$

Since $\mathbf{u} \in U_k$ was arbitrary, the lemma is proved. □

Corollary 10.2.5.

If $\mathbf{r}_l \neq 0$ for $l = 0, \ldots, k$, $k \leq n$, then $\{\mathbf{r}_0, \ldots, \mathbf{r}_k\}$ is an orthogonal basis of U_k.

Lemma 10.2.3 also implies that, if $U_0 = \{0\}$, then $\dim U_k = k$ as long as $\mathbf{x}^{(k)} \neq \mathbf{x}^*$, that is, before we have converged to the exact solution.

(10.2.1) and (10.2.2) define the conjugate gradient method (CG) for the iterative solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$
(hailed as a "top ten algorithm" of the 20th century, SIAM News, 33(4))

10.2.1 Krylov spaces

Definition 10.2.6. Krylov space

For $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{z} \in \mathbb{R}^n$, $\mathbf{z} \neq 0$, the l-th Krylov space is defined as

$$\mathcal{K}_l(\mathbf{A}, \mathbf{z}) := \operatorname{Span}\{\mathbf{z}, \mathbf{A}\mathbf{z}, \ldots, \mathbf{A}^{l-1}\mathbf{z}\} \,.$$

Equivalent definition: $\mathcal{K}_l(\mathbf{A},\mathbf{z}) = \{p(\mathbf{A})\mathbf{z}\text{: } p \text{ polynomial of degree } \leq l\}$

Lemma 10.2.7.

The subspaces $U_k \subset \mathbb{R}^n$, $k \geq 1$, defined by (10.2.1) *and* (10.2.2) *satisfy*

$$U_k = \mathrm{Span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \ldots, \mathbf{A}^{k-1}\mathbf{r}_0\} = \mathcal{K}_k(\mathbf{A},\mathbf{r}_0)\,,$$

where $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$ is the initial residual.

Proof. (by induction) Obviously $\mathbf{A}\mathcal{K}_k(\mathbf{A},\mathbf{r}_0) \subset \mathcal{K}_{k+1}(\mathbf{A},\mathbf{r}_0)$. In addition

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}(\mathbf{x}^{(0)} + \mathbf{z}) \quad \text{for some } \mathbf{z} \in U_k \quad \Rightarrow \quad \mathbf{r}_k = \underbrace{\mathbf{r}_0}_{\in\mathcal{K}_{k+1}(\mathbf{A},\mathbf{r}_0)} - \underbrace{\mathbf{A}\mathbf{z}}_{\in\mathcal{K}_{k+1}(\mathbf{A},\mathbf{r}_0)}\,.$$

Since $U_{k+1} = \mathrm{Span}\{U_k, \mathbf{r}_k\}$, we obtain $U_{k+1} \subset \mathcal{K}_{k+1}(\mathbf{A},\mathbf{r}_0)$. Dimensional considerations based on Lemma 10.2.3 finish the proof. □

10.2.2 Implementation of CG

Assume: basis $\{\mathbf{p}_1,\ldots,\mathbf{p}_l\}$, $l = 1,\ldots,n$, of $\mathcal{K}_l(\mathbf{A},\mathbf{r})$ available

$$\mathbf{x}^{(l)} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A},\mathbf{r}_0) \;\succ\; \text{set} \quad \mathbf{x}^{(l)} = \mathbf{x}^{(0)} + \gamma_1\mathbf{p}_1 + \cdots + \gamma_l\mathbf{p}_l\,.$$

For $\psi(\gamma_1,\ldots,\gamma_l) := J(\mathbf{x}^{(0)} + \gamma_1\mathbf{p}_1 + \cdots + \gamma_l\mathbf{p}_l)$ holds

$$(10.2.1) \quad \Leftrightarrow \quad \frac{\partial\psi}{\partial\gamma_j} = 0\,, \quad j = 1,\ldots,l\,.$$

This leads to a linear system of equations by which the coefficients γ_j can be computed:

$$\begin{bmatrix} \mathbf{p}_1^\top\mathbf{A}\mathbf{p}_1 & \cdots & \mathbf{p}_1^\top\mathbf{A}\mathbf{p}_l \\ \vdots & & \vdots \\ \mathbf{p}_l^\top\mathbf{A}\mathbf{p}_1 & \cdots & \mathbf{p}_l^\top\mathbf{A}\mathbf{p}_l \end{bmatrix} \begin{bmatrix} \gamma_1 \\ \vdots \\ \gamma_l \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1^\top\mathbf{r} \\ \vdots \\ \mathbf{p}_l^\top\mathbf{r} \end{bmatrix}, \quad \mathbf{r} := \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}\,. \tag{10.2.8}$$

Great simplification, if $\{\mathbf{p}_1,\ldots,\mathbf{p}_l\}$ $\mathbf{A}$-orthogonal basis of $\mathcal{K}_l(\mathbf{A},\mathbf{r})$: $\mathbf{p}_j^\top\mathbf{A}\mathbf{p}_i = 0$ for $i \neq j$.

Recall: s.p.d. $\mathbf{A}$ induces an inner product $\succ$ concept of orthogonality [**?**, Sect. 4.4], [**?**, Sect. 6.2]. "$\mathbf{A}$-geometry" like standard Euclidean space.

Assume: $\mathbf{A}$-orthogonal basis $\{\mathbf{p}_1,\ldots,\mathbf{p}_n\}$ of $\mathbb{R}^n$ available, such that

$$\mathrm{Span}\{\mathbf{p}_1,\ldots,\mathbf{p}_l\} = \mathcal{K}_l(\mathbf{A},\mathbf{r})\,.$$

(Efficient) successive computation of $\mathbf{x}^{(l)}$ becomes possible, see [**?**, Lemma 13.24]
(LSE (10.2.8) becomes diagonal !)

Input: : initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$
Given: : $\mathbf{A}$-orthogonal bases $\{\mathbf{p}_1, \ldots, \mathbf{p}_l\}$ of $\mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)$, $l = 1, \ldots, n$
Output: : approximate solution $\mathbf{x}^{(l)} \in \mathbb{R}^n$ of $\mathbf{Ax} = \mathbf{b}$

$$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)};$$

$$\textbf{for } j = 1 \textbf{ to } l \textbf{ do } \{ \quad \mathbf{x}^{(j)} := \mathbf{x}^{(j-1)} + \frac{\mathbf{p}_j^\top \mathbf{r}_0}{\mathbf{p}_j^\top \mathbf{A}\mathbf{p}_j} \mathbf{p}_j \quad \} \tag{10.2.9}$$

Task: Efficient computation of $\mathbf{A}$-orthogonal vectors $\{\mathbf{p}_1, \ldots, \mathbf{p}_l\}$ spanning $\mathcal{K}_l(\mathbf{A}, \mathbf{r}_0)$ *during the CG iteration*.

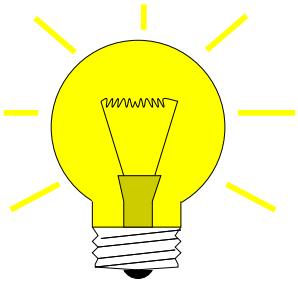
$\mathbf{A}$-orthogonalities/orthogonalities ➤ short recursions

Lemma 10.2.3 implies orthogonality $\mathbf{p}_j \perp \mathbf{r}_m := \mathbf{b} - \mathbf{A}\mathbf{x}^{(m)}$, $1 \le j \le m \le l$. Also by $\mathbf{A}$-orthogonality of the $\mathbf{p}_k$

$$\mathbf{p}_j^T(\mathbf{b} - \mathbf{A}\mathbf{x}^{(m)}) = \mathbf{p}_j^\top \left(\underbrace{\mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}}_{=\mathbf{r}_0} - \sum_{k=1}^{m} \frac{\mathbf{p}_k^\top \mathbf{r}_0}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k} \mathbf{A}\mathbf{p}_k \right) = 0 \,. \tag{10.2.10}$$

From linear algebra we already know a way to construct orthogonal basis vectors:

(10.2.10) ⇒ Idea:



Gram-Schmidt orthogonalization [**?**, Thm. 4.8], [**?**, Alg. 6.1], of residuals $\mathbf{r}_j := \mathbf{b} - \mathbf{A}\mathbf{x}^{(j)}$ w.r.t. $\mathbf{A}$-inner product:

$$\mathbf{p}_1 := \mathbf{r}_0 \,,\ \mathbf{p}_{j+1} := \underbrace{(\mathbf{b} - \mathbf{A}\mathbf{x}^{(j)})}_{=:\mathbf{r}_j} - \underbrace{\sum_{k=1}^{j} \frac{\mathbf{p}_k^\top \mathbf{A}\mathbf{r}_j}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k} \mathbf{p}_k}_{(*)} \,, \quad j = 1, \ldots, l-1 \quad . \tag{10.2.11}$$

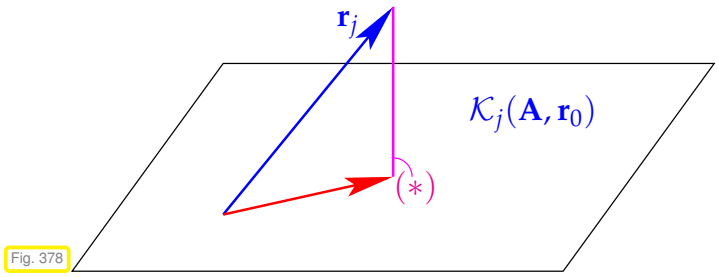


Fig. 378

Geometric interpretation of (10.2.11):

$(*) \mathrel{\hat{=}}$ orthogonal projection of $\mathbf{r}_j$ on the subspace $\mathrm{Span}\{\mathbf{p}_1, \ldots, \mathbf{p}_j\}$

Lemma 10.2.12. Bases for Krylov spaces in CG

If they do not vanish, the vectors $\mathbf{p}_j$, $1 \le j \le l$, and $\mathbf{r}_j := \mathbf{b} - \mathbf{A}\mathbf{x}^{(j)}$, $0 \le j \le l$, from (10.2.9), (10.2.11) *satisfy*

(i) $\{\mathbf{p}_1, \ldots, \mathbf{p}_j\}$ is $\mathbf{A}$-orthogonal basis von $\mathcal{K}_j(\mathbf{A}, \mathbf{r}_0)$,
(ii) $\{\mathbf{r}_0, \ldots, \mathbf{r}_{j-1}\}$ is orthogonal basis of $\mathcal{K}_j(\mathbf{A}, \mathbf{r}_0)$, cf. Cor. 10.2.5

Proof. $\mathbf{A}$-orthogonality of $\mathbf{p}_j$ by construction, study (10.2.11).

$$\text{(10.2.9) \& (10.2.11)} \quad \Rightarrow \quad \mathbf{p}_{j+1} = \mathbf{r}_0 - \sum_{k=1}^{j} \frac{\mathbf{p}_k^\top \mathbf{r}_0}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k} \mathbf{A}\mathbf{p}_k - \sum_{k=1}^{j} \frac{\mathbf{p}_k^\top \mathbf{A}\mathbf{r}_j}{\mathbf{p}_k^\top \mathbf{A}\mathbf{p}_k} \mathbf{p}_k \,.$$

$$\Rightarrow \quad \mathbf{p}_{j+1} \in \mathrm{Span}\{\mathbf{r}_0, \mathbf{p}_1, \ldots, \mathbf{p}_j, \mathbf{A}\mathbf{p}_1, \ldots, \mathbf{A}\mathbf{p}_j\} \,.$$

A simple induction argument confirms (i)

$$(10.2.11) \quad \Rightarrow \quad \mathbf{r}_j \in \mathrm{Span}\{\mathbf{p}_1,\ldots,\mathbf{p}_{j+1}\} \quad \& \quad \mathbf{p}_j \in \mathrm{Span}\{\mathbf{r}_0,\ldots,\mathbf{r}_{j-1}\}\,. \tag{10.2.13}$$

$$\blacktriangleright \quad \boxed{\mathrm{Span}\{\mathbf{p}_1,\ldots,\mathbf{p}_j\} = \mathrm{Span}\{\mathbf{r}_0,\ldots,\mathbf{r}_{j-1}\} = \mathcal{K}_l(\mathbf{A},\mathbf{r}_0)}\,. \tag{10.2.14}$$

$$(10.2.10) \quad \Rightarrow \quad \mathbf{r}_j \perp \mathrm{Span}\{\mathbf{p}_1,\ldots,\mathbf{p}_j\} = \mathrm{Span}\{\mathbf{r}_0,\ldots,\mathbf{r}_{j-1}\}\,. \tag{10.2.15}$$

□

Orthogonalities from Lemma 10.2.12 ➤ short recursions for $\mathbf{p}_k$, $\mathbf{r}_k$, $\mathbf{x}^{(k)}$!

$$(10.2.10) \quad \Rightarrow \quad (10.2.11) \text{ collapses to } \quad \mathbf{p}_{j+1} := \mathbf{r}_j - \frac{\mathbf{p}_j^\top \mathbf{A}\mathbf{r}_j}{\mathbf{p}_j^\top \mathbf{A}\mathbf{p}_j}\,\mathbf{p}_j\,, \quad j = 1,\ldots,l\,.$$

recursion for residuals:

$$(10.2.9) \quad \blacktriangleright \quad \mathbf{r}_j = \mathbf{r}_{j-1} - \frac{\mathbf{p}_j^\top \mathbf{r}_0}{\mathbf{p}_j^\top \mathbf{A}\mathbf{p}_j}\,\mathbf{A}\mathbf{p}_j\,.$$

$$\text{Lemma 10.2.12, (i)} \quad \blacktriangleright \quad \mathbf{r}_{j-1}^H \mathbf{p}_j = \left(\mathbf{r}_0 + \sum_{k=1}^{m-1} \frac{\mathbf{r}_0^\top \mathbf{p}_k}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k}\,\mathbf{A}\mathbf{p}_k\right)^T \mathbf{p}_j = \mathbf{r}_0^\top \mathbf{p}_j\,. \tag{10.2.16}$$

The orthogonality (10.2.16) together with (10.2.15) permits us to replace $\mathbf{r}_0$ with $\mathbf{r}_{j-1}$ in the actual implementation.

(10.2.17) CG method for solving $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A}$ s.p.d. → [?, Alg. 13.27]

Input : initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$
Output : approximate solution $\mathbf{x}^{(l)} \in \mathbb{R}^n$

$\mathbf{p}_1 = \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$;
for $j = 1$ **to** l **do** {

$$\mathbf{x}^{(j)} := \mathbf{x}^{(j-1)} + \frac{\mathbf{p}_j^T \mathbf{r}_{j-1}}{\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j}\,\mathbf{p}_j;$$

$$\mathbf{r}_j = \mathbf{r}_{j-1} - \frac{\mathbf{p}_j^T \mathbf{r}_{j-1}}{\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j}\mathbf{A}\mathbf{p}_j;$$

$$\mathbf{p}_{j+1} = \mathbf{r}_j - \frac{(\mathbf{A}\mathbf{p}_j)^T \mathbf{r}_j}{\mathbf{p}_j^T \mathbf{A}\mathbf{p}_j}\,\mathbf{p}_j;$$

}

Input: initial guess $\mathbf{x} \hat{=} \mathbf{x}^{(0)} \in \mathbb{R}^n$
tolerance $\tau > 0$
Output: approximate solution $\mathbf{x} \hat{=} \mathbf{x}^{(l)}$

$\mathbf{p} := \mathbf{r}_0 := \mathbf{r} := \mathbf{b} - \mathbf{A}\mathbf{x}$;
for $j = 1$ **to** $l_{\max}$ **do** {
$\beta := \mathbf{r}^T\mathbf{r}$;
$\mathbf{h} := \mathbf{A}\mathbf{p}$;
$\alpha := \frac{\beta}{\mathbf{p}^T\mathbf{h}}$;
$\mathbf{x} := \mathbf{x} + \alpha\mathbf{p}$;
$\mathbf{r} := \mathbf{r} - \alpha\mathbf{h}$;
if $\|\mathbf{r}\| \leq \tau\|\mathbf{r}_0\|$ **then stop**;
$\beta := \frac{\mathbf{r}^T\mathbf{r}}{\beta}$;
$\mathbf{p} := \mathbf{r} + \beta\mathbf{p}$;
}

In CG algorithm: $\mathbf{r}_j = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$ agrees with the residual associated with the current iterate (in exact arithmetic, *cf.* Ex. 10.2.21), but computation through short recursion is more efficient.

➢ We find that the CG method possesses all the algorithmic advantages of the gradient method, *cf.* the discussion in Section 10.1.3.

1 matrix×vector product, 3 dot products, 3 AXPY-operations per step:
If $\mathbf{A}$ sparse, $\mathrm{nnz}(\mathbf{A}) \sim n$ ➤ computational effort $O(n)$ per step

MATLAB-code 10.2.18: basic CG iteration for solving $\mathbf{Ax} = \mathbf{b}$, § 10.2.17

```
function x = cg(evalA,b,x,tol,maxit)
% x supplies initial guess, maxit maximal number of CG steps
% evalA must pass a handle to a MATLAB function realizing A*x
r = b - evalA(x); rho = 1; n0 = norm(r);
for i = 1 : maxit
   rho1 = rho; rho = r' * r;
   if (i == 1), p = r;
   else  beta = rho/rho1; p = r + beta * p; end
   q = evalA(p); alpha = rho/(p' * q);
   x = x + alpha * p;     % update of approximate solution
   if (norm(b-A*x) <= tol*n0) return; end % termination, see
      Rem. 10.2.19
   r = r - alpha * q;       % update of residual
end
```

MATLAB-function:

`x=pcg(A,b,tol,maxit,[],[],x0)` : Solve $\mathbf{Ax} = \mathbf{b}$ with at most `maxit` CG steps: stop, when $\|\mathbf{r}_l\| : \|\mathbf{r}_0\| < \mathrm{tol}$.
`x=pcg(Afun,b,tol,maxit,[],[],x0)` : `Afun` = handle to function for computing $\mathbf{A}\times$vector.
`[x,flag,relr,it,resv] = pcg(...)` : diagnostic information about iteration

Remark 10.2.19 (A posteriori termination criterion for plain CG)

For any vector norm and associated matrix norm (→ Def. 1.5.76) hold (with residual $\mathbf{r}_l := \mathbf{b} - \mathbf{Ax}^{(l)}$)

$$\frac{1}{\mathrm{cond}(\mathbf{A})}\frac{\|\mathbf{r}_l\|}{\|\mathbf{r}_0\|} \leq \frac{\|\mathbf{x}^{(l)}-\mathbf{x}^*\|}{\|\mathbf{x}^{(0)}-\mathbf{x}^*\|} \leq \mathrm{cond}(\mathbf{A})\,\frac{\|\mathbf{r}_l\|}{\|\mathbf{r}_0\|}\,. \tag{10.2.20}$$

relative decrease of iteration error

(10.2.20) can easily be deduced from the error equation $\mathbf{A}(\mathbf{x}^{(k)} - \mathbf{x}^*) = \mathbf{r}_k$, see Def. 2.4.1 and (2.4.12).

10.2.3 Convergence of CG

Note: CG is a *direct solver*, because (in exact arithmetic) $\mathbf{x}^{(k)} = \mathbf{x}^*$ for some $k \leq n$

Example 10.2.21 (Impact of roundoff errors on CG → [?, Rem. 4.3])

Numerical experiment: `A=hilb(20)`, $\mathbf{x}^{(0)} = 0$, $\mathbf{b} = [1,\ldots,1]^\top$

Hilbert-Matrix: extremely ill-conditioned

residual norms during CG iteration $\mathbf{R} = \left[\mathbf{r}_0,\ldots,\mathbf{r}^{(10)}\right]$

▷:

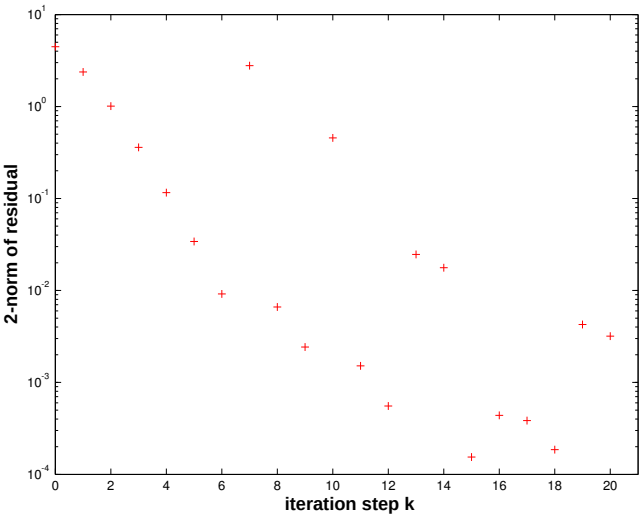


Fig. 379

$\mathbf{R}^\top\mathbf{R} =$

1.000000	−0.000000	0.000000	−0.000000	0.000000	−0.000000	0.016019	−0.795816	−0.430569	0.348133
−0.000000	1.000000	−0.000000	0.000000	−0.000000	0.000000	−0.012075	0.600068	−0.520610	0.420903
0.000000	−0.000000	1.000000	−0.000000	0.000000	−0.000000	0.001582	−0.078664	0.384453	−0.310577
−0.000000	0.000000	−0.000000	1.000000	−0.000000	0.000000	−0.000024	0.001218	−0.024115	0.019394
0.000000	−0.000000	0.000000	−0.000000	1.000000	−0.000000	0.000000	−0.000002	0.000151	−0.000118
−0.000000	0.000000	−0.000000	0.000000	−0.000000	1.000000	−0.000000	0.000000	−0.000000	0.000000
0.016019	−0.012075	0.001582	−0.000024	0.000000	−0.000000	1.000000	−0.000000	−0.000000	0.000000
−0.795816	0.600068	−0.078664	0.001218	−0.000002	0.000000	−0.000000	1.000000	0.000000	−0.000000
−0.430569	−0.520610	0.384453	−0.024115	0.000151	−0.000000	−0.000000	0.000000	1.000000	0.000000
0.348133	0.420903	−0.310577	0.019394	−0.000118	0.000000	0.000000	−0.000000	0.000000	1.000000

➢ Roundoff
- ✦ destroys orthogonality of residuals
- ✦ prevents computation of exact solution after n steps.

➢ Numerical instability (→ Def. 1.5.85) ➢ pointless to (try to) use CG as direct solver!

Practice: CG used for large n as *iterative solver*: $\mathbf{x}^{(k)}$ for some $k \ll n$ is expected to provide good approximation for $\mathbf{x}^*$

Example 10.2.22 (Convergence of CG as iterative solver)

CG (Code 10.2.18) & gradient method (Code 10.1.12) for LSE with sparse s.p.d. "Poisson matrix"

```
A = gallery('poisson',m); x0 = (1:n)'; b = zeros(n,1);
```

➢ $\mathbf{A} \in \mathbb{R}^{m^2,m^2}$

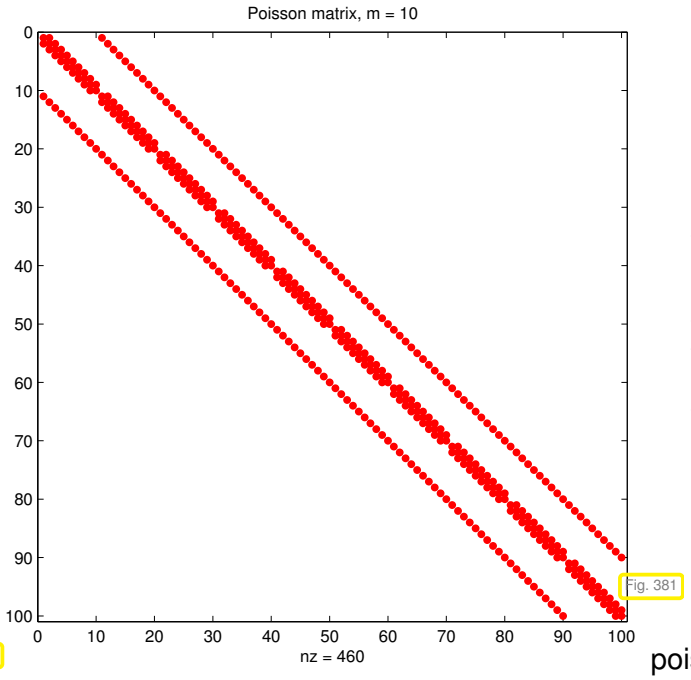


Fig. 380 poissonspy

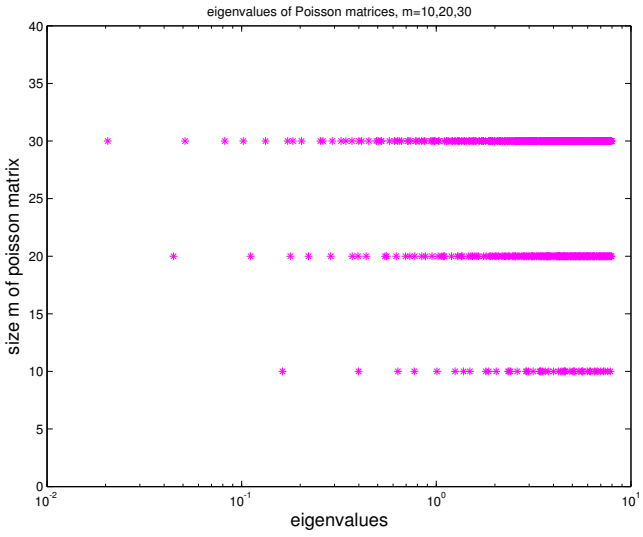


Fig. 381 poissoneig

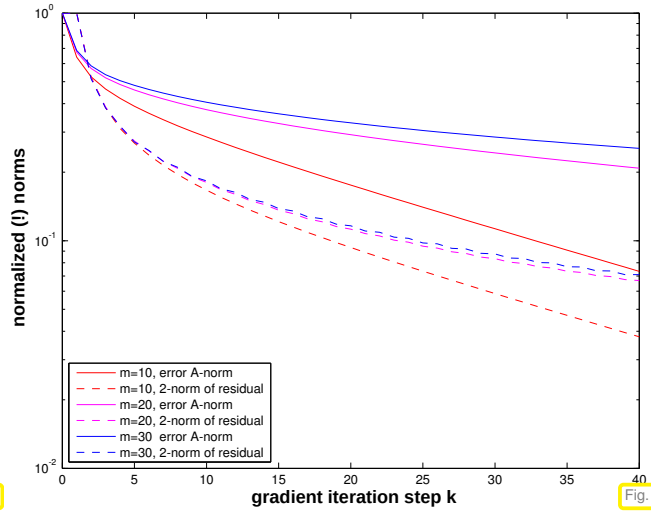


Fig. 382

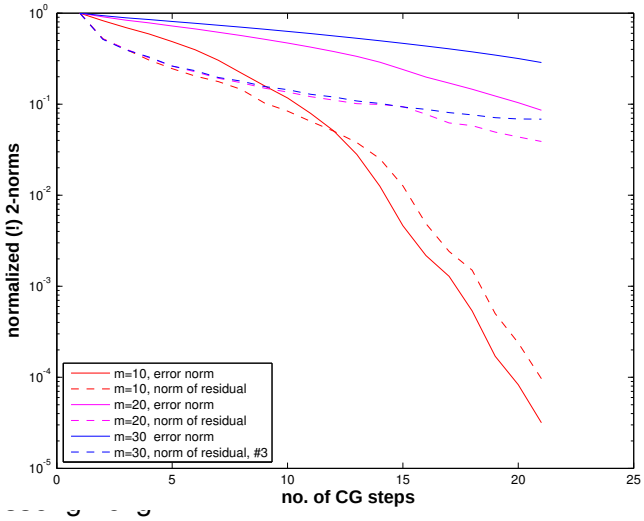


Fig. 383 cgcvgpoiss

Observations:

- CG much faster than gradient method (as expected, because it has "memory")
- Both, CG and gradient method converge more slowly for larger sizes of Poisson matrices.

Convergence theory: [**?**, Sect. 9.4.3]

A simple consequence of (10.1.5) and (10.2.1):

Corollary 10.2.23. "Optimality" of CG iterates

Writing $\mathbf{x}^* \in \mathbb{R}^n$ for the exact solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ the CG iterates satisfy

$$\left\| \mathbf{x}^* - \mathbf{x}^{(l)} \right\|_A = \min\{ \|\mathbf{y} - \mathbf{x}^*\|_A : \mathbf{y} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A}, \mathbf{r}_0) \} \quad , \quad \mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)} \, .$$

This paves the way for a quantitative convergence estimate:

$$\mathbf{y} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A},\mathbf{r}) \quad \Leftrightarrow \quad \mathbf{y} = \mathbf{x}^{(0)} + \mathbf{A}\, p(\mathbf{A})(\mathbf{x}-\mathbf{x}^{(0)}) \;, \quad p \text{ = polynomial of degree } \leq l-1 \;.$$

$$\blacktriangleright \quad \mathbf{x}-\mathbf{y} = q(\mathbf{A})(\mathbf{x}-\mathbf{x}^{(0)}), \quad q \text{ = polynomial of degree } \leq l \;, \quad q(0)=1 \;.$$

$$\blacktriangledown$$

$$\left\|\mathbf{x}-\mathbf{x}^{(l)}\right\|_A \leq \boxed{\min\{\max_{\lambda\in\sigma(\mathbf{A})} |q(\lambda)| \text{: } q \text{ polynomial of degree } \leq l \;, \quad q(0)=1\}} \cdot \left\|\mathbf{x}-\mathbf{x}^{(0)}\right\|_A \;. \tag{10.2.24}$$

Bound this minimum for $\lambda \in [\lambda_{\min}(\mathbf{A}), \lambda_{\max}(\mathbf{A})]$ by using suitable "polynomial candidates"

Tool: Chebychev polynomials (→ Section 6.1.3.1) ➢ lead to the following estimate [**?**, Satz 9.4.2], [**?**, Satz 13.29]

Theorem 10.2.25. Convergence of CG method

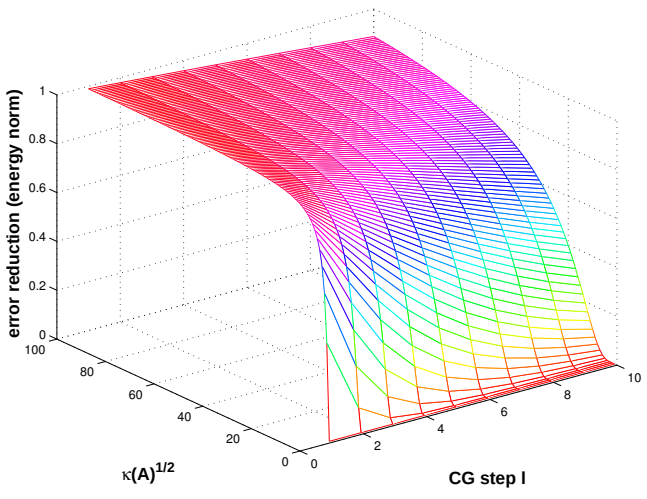
The iterates of the CG method for solving $\mathbf{Ax}=\mathbf{b}$ (see Code 10.2.18) with $\mathbf{A}=\mathbf{A}^\top$ s.p.d. satisfy

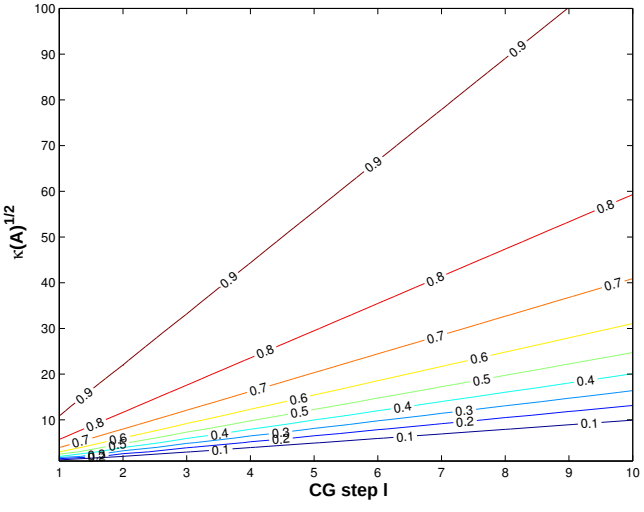
$$\begin{aligned}\left\|\mathbf{x}-\mathbf{x}^{(l)}\right\|_A &\leq \frac{2\left(1-\frac{1}{\sqrt{\kappa(\mathbf{A})}}\right)^l}{\left(1+\frac{1}{\sqrt{\kappa(\mathbf{A})}}\right)^{2l}+\left(1-\frac{1}{\sqrt{\kappa(\mathbf{A})}}\right)^{2l}}\left\|\mathbf{x}-\mathbf{x}^{(0)}\right\|_A \\ &\leq 2\left(\frac{\sqrt{\kappa(\mathbf{A})}-1}{\sqrt{\kappa(\mathbf{A})}+1}\right)^l \left\|\mathbf{x}-\mathbf{x}^{(0)}\right\|_A \;.\end{aligned}$$

(recall: $\kappa(\mathbf{A})$ = spectral condition number of $\mathbf{A}$, $\kappa(\mathbf{A}) = \mathrm{cond}_2(\mathbf{A})$)

The estimate of this theorem confirms *asymptotic linear convergence* of the CG method (→ Def. 8.1.9) with a rate of $\dfrac{\sqrt{\kappa(\mathbf{A})}-1}{\sqrt{\kappa(\mathbf{A})}+1}$

Plots of bounds for error reduction (in energy norm) during CG iteration from Thm. 10.2.25:





MATLAB-code 10.2.26: lotting theoretical bounds for CG convergence rate

```
function plottheorate
% Python:  ../PYTHON/ConjugateGradient.py plottheorate()

[X,Y] = meshgrid(1:10,1:100); R = zeros(100,10);
for I=1:100
  t = 1/I;
  for j=1:10
    R(I,j) = 2*(1-t)^j/((1+t)^(2*j)+(1-t)^(2*j));
  end
end

figure; view([-45,28]); mesh(X,Y,R); colormap hsv;
xlabel('{\bf CG step l}','Fontsize',14);
ylabel('{\bf \kappa(A)^{1/2}}','Fontsize',14);
zlabel('{\bf error reduction (energy norm)}','Fontsize',14);

print -depsc2 '../PICTURES/theorate1.eps';

figure; [C,h] = contour(X,Y,R); clabel(C,h);
xlabel('{\bf CG step l}','Fontsize',14);
ylabel('{\bf \kappa(A)^{1/2}}','Fontsize',14);

print -depsc2 '../PICTURES/theorate2.eps';
```

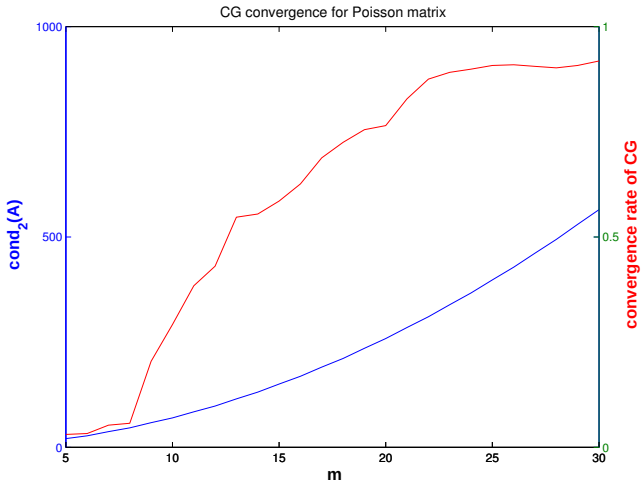
Example 10.2.27 (Convergence rates for CG method)

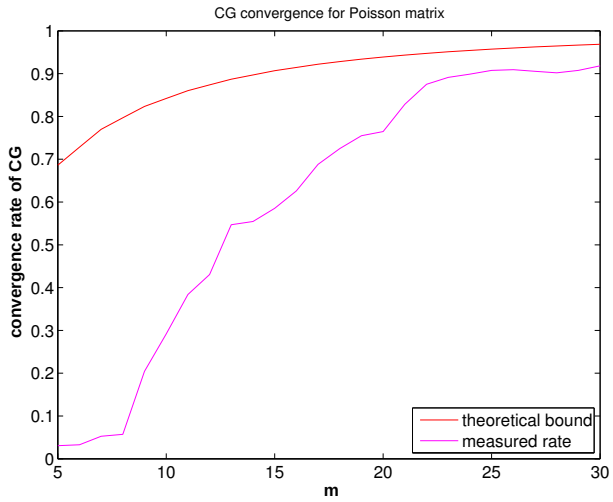
MATLAB-code 10.2.28: CG for Poisson matrix

```
A = gallery('poisson',m); n =
   size(A,1);
x0 = (1:n)'; b = ones(n,1); maxit =
   30; tol =0;
[x,flag,relres,iter,resvec] =
   pcg(A,b,tol,maxit,[],[],x0);
```

Measurement of rate of (linear) convergence:

$$\text{rate} \approx \sqrt[10]{\frac{\|\mathbf{r}_{30}\|_2}{\|\mathbf{r}_{20}\|_2}}\,. \qquad (10.2.29)$$





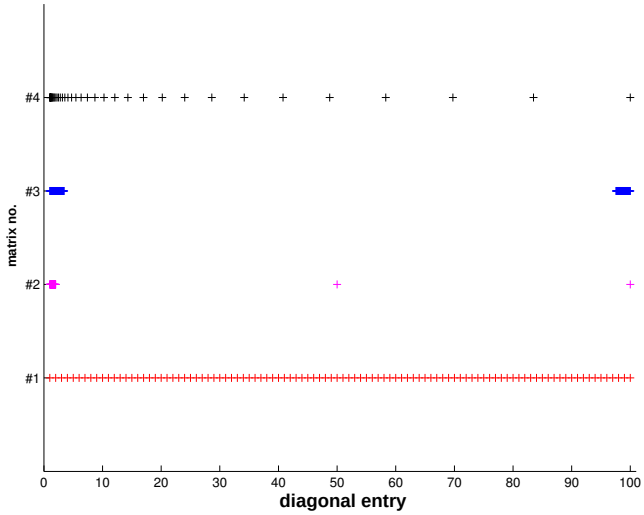
Justification for estimating the rate of linear convergence (→ Def. 8.1.9) of $\|\mathbf{r}_k\|_2 \to 0$:

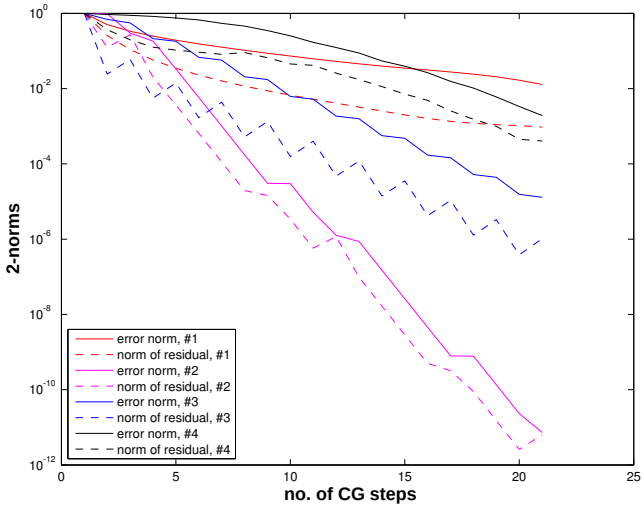
$$\|\mathbf{r}_{k+1}\|_2 \approx L\|\mathbf{r}_k\|_2 \quad \Rightarrow \quad \|\mathbf{r}_{k+m}\|_2 \approx L^m\|\mathbf{r}_k\|_2 \,.$$

Example 10.2.30 (CG convergence and spectrum → Ex. 10.1.17)

Test matrix #1: `A=diag(d); d = (1:100);`
Test matrix #2: `A=diag(d); d = [1+(0:97)/97 , 50 , 100];`
Test matrix #3: `A=diag(d); d = [1+(0:49)*0.05, 100-(0:49)*0.05];`
Test matrix #4: eigenvalues exponentially dense at 1

```
x0 = cos((1:n)'); b = zeros(n,1);
```





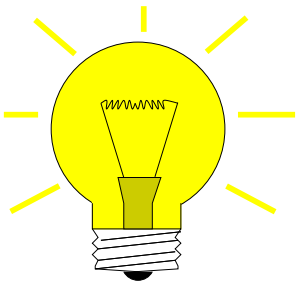
Observations: Distribution of eigenvalues has crucial impact on convergence of CG
(This is clear from the convergence theory, because detailed information about the spectrum allows a much better choice of "candidate polynomial" in (10.2.24) than merely using Chebychev polynomials)

➢ Clustering of eigenvalues leads to faster convergence of CG
(in stark contrast to the behavior of the gradient method, see Ex. 10.1.17)

CG convergence boosted by clustering of eigenvalues

10.3 Preconditioning [?, Sect. 13.5], [?, Ch. 10], [?, Sect. 4.3.5]

Thm. 10.2.25 ➢ (Potentially) slow convergence of CG in case $\kappa(\mathbf{A}) \gg 1$.



Idea: Preconditioning

Apply CG method to transformed linear system

$$\widetilde{\mathbf{A}}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}} \quad , \quad \widetilde{\mathbf{A}} := \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2} \, , \, \widetilde{\mathbf{x}} := \mathbf{B}^{1/2}\mathbf{x} \, , \, \widetilde{\mathbf{b}} := \mathbf{B}^{-1/2}\mathbf{b} \, , \tag{10.3.1}$$

with "small" $\kappa(\widetilde{\mathbf{A}})$, $\mathbf{B} = \mathbf{B}^\top \in \mathbb{R}^{N,N}$ s.p.d. $\hat{=}$ preconditioner.

Remark 10.3.2 (Square root of a s.p.d. matrix)

What is meant by the "square root" $\mathbf{B}^{1/2}$ of a s.p.d. matrix $\mathbf{B}$?

Recall (10.1.15) : for every $\mathbf{B} \in \mathbb{R}^{n,n}$ with $\mathbf{B}^\top = \mathbf{B}$ there is an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{n.n}$ such that $\mathbf{B} = \mathbf{Q}^\top\mathbf{D}\mathbf{Q}$ with a diagonal matrix $\mathbf{D}$ ($\rightarrow$ Cor. 9.1.9, [?, Thm. 7.8], [?, Satz 9.15]). If $\mathbf{B}$ is s.p.d. the (diagonal) entries of $\mathbf{D}$ are strictly positive and we can define

$$\mathbf{D} = \operatorname{diag}(\lambda_1, \ldots, \lambda_n), \quad \lambda_i > 0 \quad \Rightarrow \quad \mathbf{D}^{1/2} := \operatorname{diag}(\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_n}) \, .$$

This is generalized to

$$\mathbf{B}^{1/2} := \mathbf{Q}^\top\mathbf{D}^{1/2}\mathbf{Q} \, ,$$

and one easily verifies, using $\mathbf{Q}^\top = \mathbf{Q}^{-1}$, that $(\mathbf{B}^{1/2})^2 = \mathbf{B}$ and that $\mathbf{B}^{1/2}$ is s.p.d. In fact, these two requirements already determine $\mathbf{B}^{1/2}$ uniquely:

$\mathbf{B}^{1/2}$ is the unique s.p.d. matrix such that $(\mathbf{B}^{1/2})^2 = \mathbf{B}$.

Notion 10.3.3. Preconditioner

A s.p.d. matrix $\mathbf{B} \in \mathbb{R}^{n,n}$ is called a preconditioner (*ger.:* Vorkonditionierer) for the s.p.d. matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, if

1. $\kappa(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})$ is "small" and
2. the evaluation of $\mathbf{B}^{-1}\mathbf{x}$ is about as expensive (in terms of elementary operations) as the matrix×vector multiplication $\mathbf{A}\mathbf{x}$, $\mathbf{x} \in \mathbb{R}^n$.

Recall: spectral condition number $\kappa(\mathbf{A}) := \dfrac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}$, see (10.1.21)

There are several equivalent ways to express that $\kappa(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})$ is "small":

- $\kappa(\mathbf{B}^{-1}\mathbf{A})$ is "small",

  because spectra agree $\sigma(\mathbf{B}^{-1}\mathbf{A}) = \sigma(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})$ due to similarity ($\rightarrow$ Lemma 9.1.6)
- $\exists 0 < \gamma < \Gamma$, Γ/γ "small": $\gamma\,(\mathbf{x}^\top\mathbf{B}\mathbf{x}) \le \mathbf{x}^\top\mathbf{A}\mathbf{x} \le \Gamma\,(\mathbf{x}^\top\mathbf{B}\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{R}^n$,

  where equivalence is seen by transforming $\mathbf{y} := \mathbf{B}^{-1/2}\mathbf{x}$ and appealing to the min-max Thm. 9.3.41.

"Reader's digest" version of Notion 10.3.3:

S.p.d. $\mathbf{B}$ preconditioner $:\Leftrightarrow$ $\mathbf{B}^{-1}$ = cheap approximate inverse of $\mathbf{A}$

Problem: $\mathbf{B}^{1/2}$, which occurs prominently in (10.3.1) is usually not available with acceptable computational costs.

However, if one formally applies § 10.2.17 to the transformed system

$$\widetilde{\mathbf{A}}\widetilde{\mathbf{x}} := \left(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\right)(\mathbf{B}^{1/2}\mathbf{x}) = \widetilde{\mathbf{b}} := \mathbf{B}^{-1/2}\mathbf{b}\,,$$

from (10.3.1), it becomes apparent that, after suitable transformation of the iteration variables $\mathbf{p}_j$ and $\mathbf{r}_j$, $\mathbf{B}^{1/2}$ and $\mathbf{B}^{-1/2}$ invariably occur in products $\mathbf{B}^{-1/2}\mathbf{B}^{-1/2} = \mathbf{B}^{-1}$ and $\mathbf{B}^{1/2}\mathbf{B}^{-1/2} = \mathbf{I}$. Thus, thanks to this intrinsic transformation square roots of $\mathbf{B}$ are not required for the implementation!

CG for $\widetilde{\mathbf{A}}\widetilde{\mathbf{x}} = \widetilde{\mathbf{b}}$

Input : initial guess $\widetilde{\mathbf{x}}^{(0)} \in \mathbb{R}^n$
Output : approximate solution $\widetilde{\mathbf{x}}^{(l)} \in \mathbb{R}^n$

$\widetilde{\mathbf{p}}_1 := \widetilde{\mathbf{r}}_0 := \widetilde{\mathbf{b}} - \mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\widetilde{\mathbf{x}}^{(0)}$;
for $j = 1$ **to** l **do** {

$$\alpha := \frac{\widetilde{\mathbf{p}}_j^T\widetilde{\mathbf{r}}_{j-1}}{\widetilde{\mathbf{p}}_j^T\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j}$$
$\widetilde{\mathbf{x}}^{(j)} := \widetilde{\mathbf{x}}^{(j-1)} + \alpha\,\widetilde{\mathbf{p}}_j$;

$\widetilde{\mathbf{r}}_j = \widetilde{\mathbf{r}}_{j-1} - \alpha\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{1/2}\widetilde{\mathbf{p}}_j$;

$$\widetilde{\mathbf{p}}_{j+1} = \widetilde{\mathbf{r}}_j - \frac{(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j)^T\widetilde{\mathbf{r}}_j}{\widetilde{\mathbf{p}}_j^T\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j}\,\widetilde{\mathbf{p}}_j;$$

}

Equivalent CG with transformed variables

Input : initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$
Output : approximate solution $\mathbf{x}^{(l)} \in \mathbb{R}^n$

$\mathbf{B}^{1/2}\widetilde{\mathbf{r}}_0 := \mathbf{B}^{1/2}\widetilde{\mathbf{b}} - \mathbf{A}\mathbf{B}^{-1/2}\widetilde{\mathbf{x}}^{(0)}$;
$\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_1 := \mathbf{B}^{-1}(\mathbf{B}^{1/2}\widetilde{\mathbf{r}}_0)$;
for $j = 1$ **to** l **do** {

$$\alpha := \frac{(\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j)^T\mathbf{B}^{1/2}\widetilde{\mathbf{r}}_{j-1}}{(\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j)^T\mathbf{A}\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j}$$
$\mathbf{B}^{-1/2}\widetilde{\mathbf{x}}^{(j)} := \mathbf{B}^{-1/2}\widetilde{\mathbf{x}}^{(j-1)} + \alpha\,\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j$;

$\mathbf{B}^{1/2}\widetilde{\mathbf{r}}_j = \mathbf{B}^{1/2}\widetilde{\mathbf{r}}_{j-1} - \alpha\mathbf{A}\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j$;

$$\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_{j+1} = \mathbf{B}^{-1}(\mathbf{B}^{-1/2}\widetilde{\mathbf{r}}_j) - \frac{(\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j)^T\mathbf{A}\mathbf{B}^{-1}(\mathbf{B}^{1/2}\widetilde{\mathbf{r}}_j)}{(\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j)^T\mathbf{A}\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j}\,\mathbf{B}^{-1/2}\widetilde{\mathbf{p}}_j;$$

}

with the transformations:

$$\widetilde{\mathbf{x}}^{(k)} = \mathbf{B}^{1/2}\mathbf{x}^{(k)} \quad , \quad \widetilde{\mathbf{r}}_k = \mathbf{B}^{-1/2}\mathbf{r}_k \quad , \quad \widetilde{\mathbf{p}}_k = \mathbf{B}^{-1/2}\mathbf{r}_k\,. \tag{10.3.4}$$

(10.3.5) Preconditioned CG method (PCG) **[?, Alg. 13.32], [?, Alg. 10.1]**

Input: initial guess $\mathbf{x} \in \mathbb{R}^n \triangleq \mathbf{x}^{(0)} \in \mathbb{R}^n$, tolerance $\tau > 0$
Output: approximate solution $\mathbf{x} \triangleq \mathbf{x}^{(l)}$

$$
\begin{aligned}
&\mathbf{p} := \mathbf{r} := \mathbf{b} - \mathbf{A}\mathbf{x}; \quad \mathbf{p} := \mathbf{B}^{-1}\mathbf{r}; \mathbf{q} := \mathbf{p}; \tau_0 := \mathbf{p}^\top \mathbf{r};\\
&\textbf{for } l = 1 \textbf{ to } l_{\max} \textbf{ do } \{\\
&\quad \beta := \mathbf{r}^\top \mathbf{q}; \quad \mathbf{h} := \mathbf{A}\mathbf{p}; \quad \alpha := \frac{\beta}{\mathbf{p}^\top \mathbf{h}};\\
&\quad \mathbf{x} := \mathbf{x} + \alpha \mathbf{p};\\
&\quad \mathbf{r} := \mathbf{r} - \alpha \mathbf{h};\\
&\quad \mathbf{q} := \mathbf{B}^{-1}\mathbf{r}; \quad \beta := \frac{\mathbf{r}^\top \mathbf{q}}{\beta};\\
&\quad \textbf{if } |\mathbf{q}^\top \mathbf{r}| \leq \tau \cdot \tau_0 \textbf{ then stop};\\
&\quad \mathbf{p} := \mathbf{q} + \beta \mathbf{p};\\
&\quad \}
\end{aligned}
\tag{10.3.6}
$$

MATLAB-code 10.3.7: simple implementation of PCG algorithm § 10.3.5

```
function [x,rn,xk] = pcgbase(evalA,b,tol,maxit,invB,x)
% evalA must pass a handle to a function implementing A*x
% invB is to be a handle to a function providing the action of the
% preconditioner on a vector.  The other arguments like for MATLAB's
   pcg.
r = b - evalA(x); rho = 1; rn = [];
if (nargout > 2), xk = x; end
for i = 1 : maxit
  y = invB(r);
  rho_old = rho; rho = r' * y; rn = [rn,rho];
  if (i == 1), p = y; rho0 = rho;
  elseif (rho < rho0*tol), return;
  else beta = rho/rho_old; p = y+beta*p; end
  q = evalA(p); alpha = rho /(p' * q);
  x = x + alpha * p;
  r = r - alpha * q;
  if (nargout > 2), xk = [xk,x]; end
end
```

Computational effort per step: 1 evaluation $\mathbf{A}\times$vector,
1 evaluation $\mathbf{B}^{-1}\times$vector,
3 dot products, 3 AXPY-operations

Remark 10.3.8 (Convergence theory for PCG)

Assertions of Thm. 10.2.25 remain valid with $\kappa(\mathbf{A})$ replaced with $\kappa(\mathbf{B}^{-1}\mathbf{A})$ and energy norm based on $\widetilde{\mathbf{A}}$ instead of $\mathbf{A}$.

Example 10.3.9 (Simple preconditioners)

$\mathbf{B}$ = easily invertible "part" of $\mathbf{A}$

- $\mathbf{B} = \mathtt{diag}(\mathbf{A})$: Jacobi preconditioner (diagonal scaling)
- $(\mathbf{B})_{ij} = \begin{cases} (\mathbf{A})_{ij} & \text{, if } |i-j| \le k \text{ ,} \\ 0 & \text{else,} \end{cases}$ for some $k \ll n$.
- Symmetric Gauss-Seidel preconditioner

Idea: Solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ approximately in two stages:

① Approximation $\mathbf{A}^{-1} \approx \mathtt{tril}(\mathbf{A})$ (lower triangular part): $\widetilde{\mathbf{x}} = \mathtt{tril}(\mathbf{A})^{-1}\mathbf{b}$

② Approximation $\mathbf{A}^{-1} \approx \mathtt{triu}(\mathbf{A})$ (upper triangular part) and use this to approximately "solve" the error equation $\mathbf{A}(\mathbf{x} - \widetilde{\mathbf{x}}) = \mathbf{r}$, with residual $\mathbf{r} := \mathbf{b} - \mathbf{A}\widetilde{\mathbf{x}}$:

$$\mathbf{x} = \widetilde{\mathbf{x}} + \mathtt{triu}(\mathbf{A})^{-1}(\mathbf{b} - \mathbf{A}\widetilde{\mathbf{x}}) \,.$$

With $\mathbf{L}_A := \mathtt{tril}(\mathbf{A})$, $\mathbf{U}_A := \mathtt{triu}(\mathbf{A})$ one finds

$$\mathbf{x} = (\mathbf{L}_A^{-1} + \mathbf{U}_A^{-1} - \mathbf{U}_A^{-1}\mathbf{A}\mathbf{L}_A^{-1})\mathbf{b} \quad \blacktriangleright \quad \mathbf{B}^{-1} = \mathbf{L}_A^{-1} + \mathbf{U}_A^{-1} - \mathbf{U}_A^{-1}\mathbf{A}\mathbf{L}_A^{-1} \,. \tag{10.3.10}$$

For all these approaches the evaluation of $\mathbf{B}^{-1}\mathbf{r}$ can be done with effort of $O(n)$ in the case of a sparse matrix $\mathbf{A}$ (e.g. with $O(1)$ non-zero entries per row). However, there is absolutely no guarantee that $\kappa(\mathbf{B}^{-1}\mathbf{A})$ will be reasonably small. It will crucially depend on $\mathbf{A}$, if this can be expected.

More complicated preconditioning strategies:

- Incomplete Cholesky factorization, MATLAB-`ichol`, [**?**, Sect. 13.5]]
- Sparse approximate inverse preconditioner (SPAI)

Example 10.3.11 (Tridiagonal preconditioning)

Efficacy of preconditioning of sparse LSE with tridiagonal part:

MATLAB-code 10.3.12: LSE for Ex. 10.3.11

```
1 A =
     spdiags(repmat([1/n,-1,2+2/n,-1,1/n],n,1),[-n/2,-1,0,1,n/2],n,n);
2 b = ones(n,1); x0 = ones(n,1); tol = 1.0E-4; maxit = 1000;
3 evalA = @(x) A*x;
4
5 % no preconditioning, see Code 10.3.7
6 invB = @(x) x; [x,rn] = pcgbase(evalA,b,tol,maxit,invB,x0);
7
8 % tridiagonal preconditioning, see Code 10.3.7
9 B = spdiags(spdiags(A,[-1,0,1]),[-1,0,1],n,n);
```

```
10 invB  = @(x) B\x; [x,rnpc] = pcgbase(evalA,b,tol,maxit,invB,x0);
     %
```

The Code 10.3.12 highlights the use of a preconditioner in the context of the PCG method; it only takes a function that realizes the application of $\mathbf{B}^{-1}$ to a vector. In Line 10 of the code this function is passed as function handle `invB`.

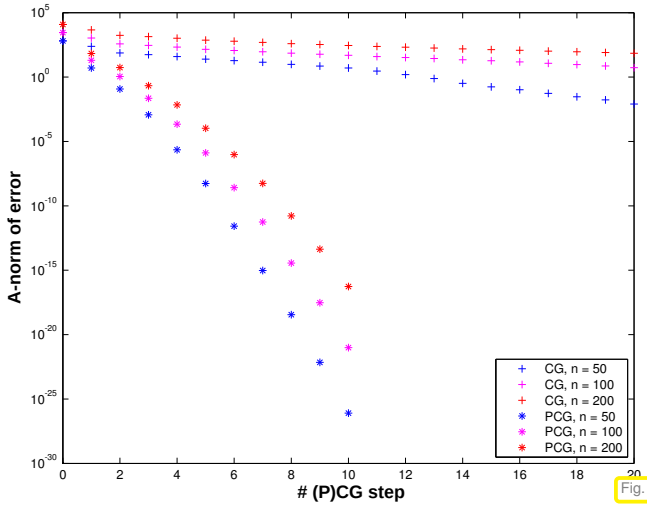


Fig. 384

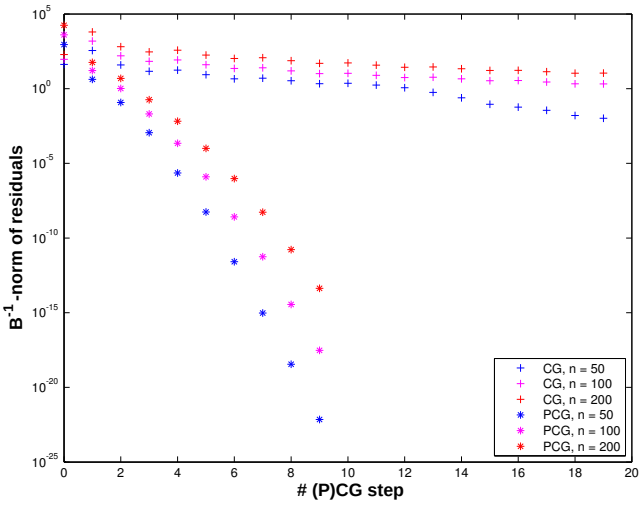


Fig. 385

n	# CG steps	# PCG steps
16	8	3
32	16	3
64	25	4
128	38	4
256	66	4
512	106	4
1024	149	4
2048	211	4
4096	298	3
8192	421	3
16384	595	3
32768	841	3

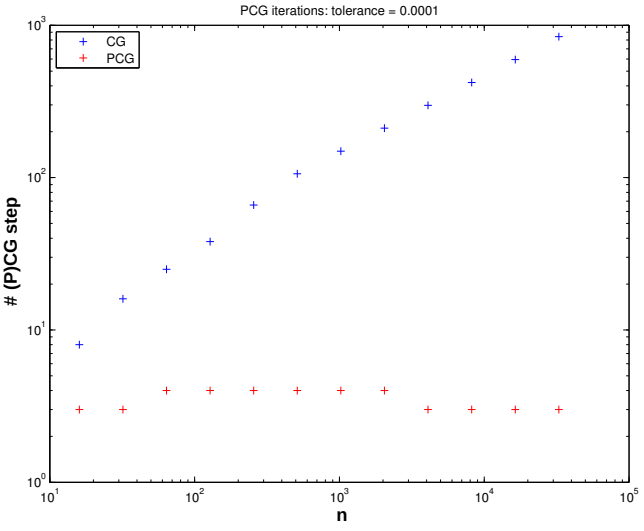


Fig. 386

Clearly in this example the tridiagonal part of the matrix is dominant for large n. In addition, its condition number grows $\sim n^2$ as is revealed by a closer inspection of the spectrum.

Preconditioning with the tridiagonal part manages to suppress this growth of the condition number of $\mathbf{B}^{-1}\mathbf{A}$ and ensures fast convergence of the preconditioned CG method

Remark 10.3.13 (Termination of PCG)

Recall Rem. 10.2.19, (10.2.20):

$$\frac{1}{\operatorname{cond}(\mathbf{A})}\frac{\|\mathbf{r}_l\|}{\|\mathbf{r}_0\|} \leq \frac{\|\mathbf{x}^{(l)}-\mathbf{x}^*\|}{\|\mathbf{x}^{(0)}-\mathbf{x}^*\|} \leq \operatorname{cond}(\mathbf{A})\,\frac{\|\mathbf{r}_l\|}{\|\mathbf{r}_0\|}\,. \tag{10.2.20}$$

$\mathbf{B}$ good preconditioner ➤ $\operatorname{cond}_2(\mathbf{B}^{-1/2}\mathbf{A}\mathbf{B}^{-1/2})$ small (→ Notion 10.3.3)

Idea: consider (10.2.20) for
- Euclidean norm $\|\cdot\| = \|\cdot\|_2 \quad \leftrightarrow \quad \mathrm{cond}_2$
- transformed quantities $\widetilde{\mathbf{x}}$, $\widetilde{\mathbf{r}}$, see (10.3.1), (10.3.4)

Monitor 2-norm of transformed residual:

$$\widetilde{\mathbf{r}} = \widetilde{\mathbf{b}} - \widetilde{\mathbf{A}}\widetilde{\mathbf{x}} = \mathbf{B}^{-1/2}\mathbf{r} \quad \Rightarrow \quad \|\widetilde{\mathbf{r}}\|_2^2 = \mathbf{r}^\top \mathbf{B}^{-1}\mathbf{r}\,.$$

(10.2.20) ➤ estimate for 2-norm of transformed iteration errors: $\left\|\widetilde{\mathbf{e}}^{(l)}\right\|_2^2 = (\mathbf{e}^{(l)})^\top \mathbf{B}\mathbf{e}^{(l)}$

Analogous to (10.2.20), estimates for energy norm (→ Def. 10.1.1) of error $\mathbf{e}^{(l)} := \mathbf{x} - \mathbf{x}^{(l)}$, $\mathbf{x}^* := \mathbf{A}^{-1}\mathbf{b}$:

Use error equation $\mathbf{A}\mathbf{e}^{(l)} = \mathbf{r}_l$:

$$\mathbf{r}_l^\top \mathbf{B}^{-1}\mathbf{r}_l = (\mathbf{B}^{-1}\mathbf{A}\mathbf{e}^{(l)})^\top \mathbf{A}\mathbf{e}^{(l)} \le \lambda_{\max}(\mathbf{B}^{-1}\mathbf{A})\left\|\mathbf{e}^{(l)}\right\|_A^2\,,$$

$$\left\|\mathbf{e}^{(l)}\right\|_A^2 = (\mathbf{A}\mathbf{e}^{(l)})^\top \mathbf{e}^{(l)} = \mathbf{r}_l^\top \mathbf{A}^{-1}\mathbf{r}_l = \mathbf{B}^{-1}\mathbf{r}^\top \mathbf{B}\mathbf{A}^{-1}\mathbf{r}_l \le \lambda_{\max}(\mathbf{B}\mathbf{A}^{-1})\,(\mathbf{B}^{-1}\mathbf{r}_l)^\top \mathbf{r}_l\,.$$

available during PCG iteration (10.3.6)

$$\frac{1}{\kappa(\mathbf{B}^{-1}\mathbf{A})}\frac{\left\|\mathbf{e}^{(l)}\right\|_A^2}{\left\|\mathbf{e}^{(0)}\right\|_A^2} \le \frac{(\mathbf{B}^{-1}\mathbf{r}_l)^\top \mathbf{r}_l}{(\mathbf{B}^{-1}\mathbf{r}_0)^\top \mathbf{r}_0} \le \kappa(\mathbf{B}^{-1}\mathbf{A})\frac{\left\|\mathbf{e}^{(l)}\right\|_A^2}{\left\|\mathbf{e}^{(0)}\right\|_A^2} \tag{10.3.14}$$

$\kappa(\mathbf{B}^{-1}\mathbf{A})$ "small" ➤ $\mathbf{B}^{-1}$-energy norm of residual $\approx$ $\mathbf{A}$-norm of error !
($\mathbf{r}_l \cdot \mathbf{B}^{-1}\mathbf{r}_l = \mathbf{q}^\top \mathbf{r}$ in Algorithm (10.3.6))

MATLAB-function: `[x,flag,relr,it,rv] = pcg(A,b,tol,maxit,B,[],x0);`
(`A`, `B` may be handles to functions providing $\mathbf{A}\mathbf{x}$ and $\mathbf{B}^{-1}\mathbf{x}$, resp.)

Remark 10.3.15 (Termination criterion in MATLAB-`pcg` → [?, Sect. 4.6])

Implementation (skeleton) of MATLAB built-in `pcg`:

Listing 10.1: MATLAB-code : PCG algorithm

```
function x = pcg(Afun,b,tol,maxit,Binvfun,x0)
x = x0; r = b - feval(Afun,x); rho = 1;
for i = 1 : maxit
  y = feval(Binvfun,r);
  rho1 = rho; rho = r' * y;
  if (i == 1)
    p = y;
  else
    beta = rho / rho1;
    p = y + beta * p;
  end
  q = feval(Afun,p);
  alpha = rho /(p' * q);
  x = x + alpha * p;
  if (norm(b - evalf(Afun,x)) <= tolb*norm(b)), return; end
  r = r - alpha * q;
end
```

Dubious termination criterion !

10.4 Survey of Krylov Subspace Methods

10.4.1 Minimal residual methods

Idea: Replace Euclidean inner product in CG with $\mathbf{A}$-inner product

$$\left\|\mathbf{x}^{(l)}-\mathbf{x}\right\|_{\mathbf{A}} \quad \text{replaced with} \quad \left\|\mathbf{A}(\mathbf{x}^{(l)}-\mathbf{x})\right\|_2 = \|\mathbf{r}_l\|_2$$

► MINRES method [**?**, Sect. 9.5.2] (for *any* symmetric matrix **!**)

Theorem 10.4.1.

For $\mathbf{A}=\mathbf{A}^H \in \mathbb{R}^{n,n}$ the residuals $\mathbf{r}_l$ generated in the MINRES iteration satisfy

$$\|\mathbf{r}_l\|_2 = \min\{\|\mathbf{A}\mathbf{y}-\mathbf{b}\|_2 \colon \mathbf{y} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A},\mathbf{r}_0)\}$$

$$\|\mathbf{r}_l\|_2 \le \frac{2\left(1-\frac{1}{\kappa(\mathbf{A})}\right)^l}{\left(1+\frac{1}{\kappa(\mathbf{A})}\right)^{2l}+\left(1-\frac{1}{\kappa(\mathbf{A})}\right)^{2l}}\|\mathbf{r}_0\|_2 \,.$$

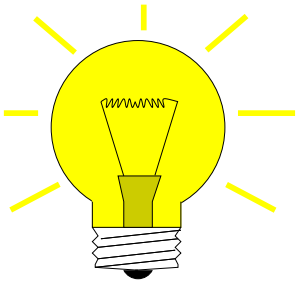
Note: similar formula for (linear) rate of convergence as for CG, see Thm. 10.2.25, but with $\sqrt{\kappa(\mathbf{A})}$ replaced with $\kappa(\mathbf{A})$ **!**

Iterative solver for $\mathbf{Ax} = \mathbf{b}$ with *symmetric* system matrix $\mathbf{A}$:

MATLAB-functions:
- `[x,flg,res,it,resv] = minres(A,b,tol,maxit,B,[],x0);`
- `[...] = minres(Afun,b,tol,maxit,Binvfun,[],x0);`

Computational costs : 1 $\mathbf{A}\times$vector, 1 $\mathbf{B}^{-1}\times$vector per step, a few dot products & SAXPYs
Memory requirement: a few vectors $\in \mathbb{R}^n$

Extension to general regular $\mathbf{A} \in \mathbb{R}^{n,n}$:



Idea: Solver overdetermined linear system of equations

$$\mathbf{x}^{(l)} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A},\mathbf{r}_0): \quad \mathbf{A}\mathbf{x}^{(l)} = \mathbf{b}$$

in *least squares sense*, → Chapter 3.

$$\mathbf{x}^{(l)} = \operatorname{argmin}\{\|\mathbf{A}\mathbf{y}-\mathbf{b}\|_2 : \mathbf{y} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A},\mathbf{r}_0)\}\ .$$

➤ GMRES method for general matrices $\mathbf{A} \in \mathbb{R}^{n,n}$ → [?, Ch. 16], [?, Sect. 4.4.2]

MATLAB-function:
- `[x,flag,relr,it,rv] = gmres(A,b,rs,tol,maxit,B,[],x0);`
- `[...] = gmres(Afun,b,rs,tol,maxit,Binvfun,[],x0);`

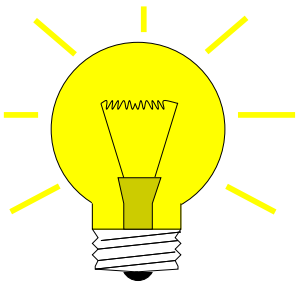
Computational costs : 1 $\mathbf{A}\times$vector, 1 $\mathbf{B}^{-1}\times$vector per step,
: $O(l)$ dot products & SAXPYs in l-th step
Memory requirements: $O(l)$ vectors $\in \mathbb{K}^n$ in l-th step

Remark 10.4.2 (Restarted GMRES)

After many steps of GMRES we face considerable computational costs and memory requirements for every further step. Thus, the iteration may be *restarted* with the current iterate $\mathbf{x}^{(l)}$ as initial guess → `rs`-parameter triggers restart after every `rs` steps (Danger: failure to converge).

10.4.2 Iterations with short recursions [?, Sect. 4.5]

Iterative methods for *general* regular system matrix $\mathbf{A}$:



Idea: Given $\mathbf{x}^{(0)} \in \mathbb{R}^n$ determine (better) approximation $\mathbf{x}^{(l)}$ through Petrov-Galerkin condition

$$\mathbf{x}^{(l)} \in \mathbf{x}^{(0)} + \mathcal{K}_l(\mathbf{A},\mathbf{r}_0): \quad \mathbf{p}^H(\mathbf{b}-\mathbf{A}\mathbf{x}^{(l)}) = 0 \quad \forall \mathbf{p} \in W_l\ ,$$

with suitable test space W_l, $\dim W_l = l$, e.g. $W_l := \mathcal{K}_l(\mathbf{A}^H,\mathbf{r}_0)$ (→ bi-conjugate gradients, BiCG)

➤ Zoo of methods with short recursions (i.e. constant effort per step)

MATLAB-function:
- `[x,flag,r,it,rv] = bicgstab(A,b,tol,maxit,B,[],x0)`
- `[...] = bicgstab(Afun,b,tol,maxit,Binvfun,[],x0);`

Computational costs : 2 $\mathbf{A}\times$vector, 2 $\mathbf{B}^{-1}\times$vector, 4 dot products, 6 SAXPYs per step
Memory requirements: 8 vectors $\in \mathbb{R}^n$

MATLAB-function:
- `[x,flag,r,it,rv] = qmr(A,b,tol,maxit,B,[],x0)`
- `[...] = qmr(Afun,b,tol,maxit,Binvfun,[],x0);`

Computational costs : 2 $\mathbf{A}\times$vector, 2 $\mathbf{B}^{-1}\times$vector, 2 dot products, 12 SAXPYs per step
Memory requirements: 10 vectors $\in \mathbb{R}^n$

- little (useful) convergence theory available
- stagnation & "breakdowns" commonly occur

Example 10.4.3 (Failure of Krylov iterative solvers)

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & \cdots & & & \cdots & 0 \\ 0 & 0 & 1 & 0 & & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & & \\ & & & & & & \ddots & \vdots \\ \vdots & & & & & \ddots & \ddots & 0 \\ 0 & & & & & & 0 & 1 \\ 1 & 0 & \cdots & & & & \cdots & 0 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 0 \\ \vdots \\ \\ \\ \vdots \\ 0 \\ 1 \end{bmatrix} \quad \blacktriangleright \quad \mathbf{x} = \mathbf{e}_1 .$$

$$\mathbf{x}^{(0)} = 0 \;\succ\; \mathbf{r}_0 = \mathbf{e}_n \;\succ\; \mathcal{K}_l(\mathbf{A},\mathbf{r}_0) = \operatorname{Span}\{\mathbf{e}_n, \mathbf{e}_{n-1}, \ldots, \mathbf{e}_{n-l+1}\}$$

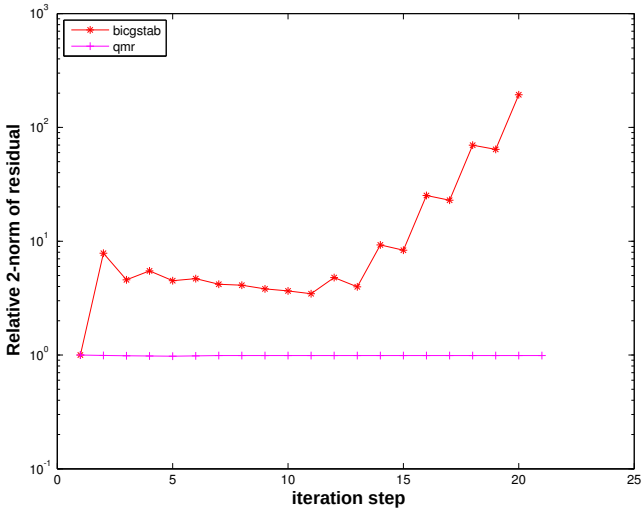
$$\blacktriangleright \quad \min\{\|\mathbf{y}-\mathbf{x}\|_2 \colon \mathbf{y} \in \mathcal{K}_l(\mathbf{A},\mathbf{r}_0)\} = \begin{cases} 1 & \text{, if } l \le n \text{ ,} \\ 0 & \text{, for } l = n \text{ .} \end{cases}$$

TRY & PRAY

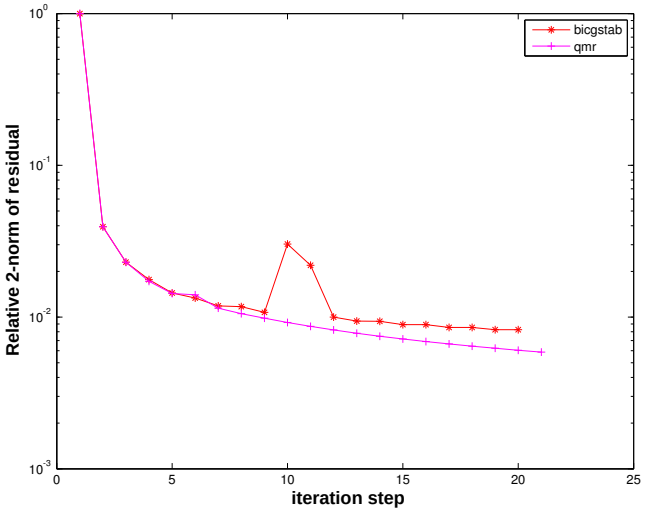
Example 10.4.4 (Convergence of Krylov subspace methods for non-symmetric system matrix)

```
A = gallery('tridiag',-0.5*ones(n-1,1),2*ones(n,1),-1.5*ones(n-1,1));
B = gallery('tridiag',0.5*ones(n-1,1),2*ones(n,1),1.5*ones(n-1,1));
```

Plotted: $\|\mathbf{r}_l\|_2 : \|\mathbf{r}_0\|_2$:



tridiagonal matrix $\mathbf{A}$



tridiagonal matrix $\mathbf{B}$

Summary:

Advantages of Krylov methods vs. direct elimination (**IF** they converge at all/sufficiently fast).

- They require system matrix $\mathbf{A}$ in procedural form `y=evalA(x)` $\leftrightarrow \mathbf{y} = \mathbf{A}\mathbf{x}$ only.
- They can perfectly exploit sparsity of system matrix.
- They can cash in on low accuracy requirements (**IF** viable termination criterion available).
- They can benefit from a good initial guess.

Chapter 11

Numerical Integration – Single Step Methods

Contents

11.1 Initial value problems (IVP) for ODEs

Acronym: **ODE** = **o**rdinary **d**ifferential **e**quation

(11.1.1) Terminology and notations related to ordinary differential equations

In our parlance, a (first-order) ordinary differential equation (ODE) is an equation of the form

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \ , \tag{11.1.2}$$

with

- ☞ a (continuous) right hand side function (r.h.s) $f : I \times D \to \mathbb{R}^d$ of time $t \in \mathbb{R}$ and state $\mathbf{y} \in \mathbb{R}^d$
- ☞ defined on a (finite) time interval $I \subset \mathbb{R}$, and state space $D \subset \mathbb{R}^d$.

In the context of mathematical modeling the state vector $\mathbf{y} \in \mathbb{R}^d$ is supposed to provide a complete (in the sense of the model) description of a system. Then (11.1.2) models a finite-dimensional dynamical system.

Examples will be provided below.

For $d > 1$ $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ can be viewed as a system of ordinary differential equations:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad \Longleftrightarrow \quad \begin{bmatrix} \dot{y}_1 \\ \vdots \\ \dot{y}_d \end{bmatrix} = \begin{bmatrix} f_1(t, y_1, \ldots, y_d) \\ \vdots \\ f_d(t, y_1, \ldots, y_d) \end{bmatrix} .$$

✎ Notation (Newton): dot $\dot{}$ $\hat{=}$ (total) derivative with respect to time t

Definition 11.1.3. Solution of an ordinary differential equation

A solution of the ODE $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ with continuous right hand side function $\mathbf{f}$ is a continuously differentiable *function* "of time t" $\mathbf{y} : J \subset I \to D$, defined on an open interval J, for which $\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t))$ holds for all $t \in J$.

A solution describes a continuous trajectory in state space, a one-parameter family of states, parameterized by time.

It goes without saying that smoothness of the right hand side function $\mathbf{f}$ is inherited by solutions of the ODE:

Lemma 11.1.4. Smoothness of solutions of ODEs

Let $\mathbf{y} : I \subset \mathbb{R} \to D$ be a solution of the ODE $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ on the time interval I.

If $\mathbf{f} : I \times D \to \mathbb{R}^d$ is r-times continuously differentiable with respect to both arguments, $r \in \mathbb{N}_0$, then the trajectory $t \mapsto \mathbf{y}(t)$ is $r+1$-times continuously differentiable in the interior of I.

Supplementary reading. Some grasp of the meaning and theory of ordinary differential equations (ODEs) is indispensable for understanding the construction and properties of numerical methods. Relevant information can be found in [**?**, Sect. 5.6, 5.7, 6.5].

Books dedicated to numerical methods for ordinary differential equations:

- [**?**] excellent textbook, but geared to the needs of students of mathematics.
- [**?**] and [**?**] : *the* standard reference.
- [**?**]: wonderful book conveying deep insight, with emphasis on mathematical concepts.

11.1.1 Modeling with ordinary differential equations: Examples

Example 11.1.5 (Growth with limited resources [?, Sect. 1.1], [?, Ch. 60])

This is an example from population dyanmics with a one-dimensional state space $D = \mathbb{R}_0^+$, $d = 1$:

$y : [0, T] \mapsto \mathbb{R}$: bacterial population density as a function of time

ODE-based model: autonomous logistic differential equations [**?**, Ex. 5.7.2]

$$\dot{y} = f(y) := (\alpha - \beta y)\, y \tag{11.1.6}$$

- $y \mathrel{\hat{=}}$ population density, $[y] = \frac{1}{\mathrm{m}^2}$
  ➢ $\dot{y} \mathrel{\hat{=}}$ instantaneous change (growth/decay) of population density
- growth rate $\alpha - \beta y$ with growth coefficients $\alpha, \beta > 0$, $[\alpha] = \frac{1}{\mathrm{s}}$, $[\beta] = \frac{\mathrm{m}^2}{\mathrm{s}}$: decreases due to more fierce competition as population density increases.

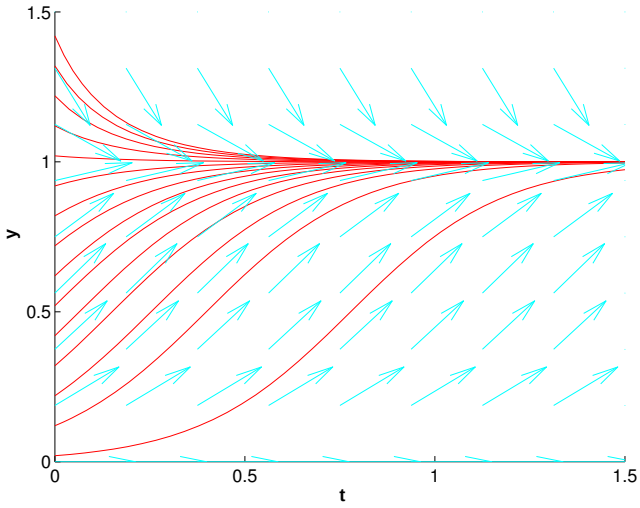


Fig. 387

Solution for different $y(0)$ ($\alpha, \beta = 5$)

By separation of variables we can compute a *family of solutions* of (11.1.6) parameterized by the initial value $y(0) = y_0 > 0$:

$$y(t) = \frac{\alpha y_0}{\beta y_0 + (\alpha - \beta y_0)\exp(-\alpha t)}\,, \tag{11.1.7}$$

for all $t \in \mathbb{R}$.

Note: $f(y^*) = 0$ for $y^* \in \{0, \alpha/\beta\}$, which are the stationary points for the ODE (11.1.6). If $y(0) = y^*$ the solution will be constant in time.

Note that by fixing the initial value $y(0)$ we can single out a *unique* representative from the family of solutions. This will turn out to be a general principle, see Section 11.1.2.

Definition 11.1.8. Autonomous ODE

An ODE of the from $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, that is, with a right hand side function that does not depend on time, but only on state, is called autonomous.

For an autonomous ODE the right hand side function defines a vector field ("velocity field") $\mathbf{y} \mapsto \mathbf{f}(\mathbf{y})$ on state space.

Example 11.1.9 (Predator-prey model [?, Sect. 1.1],[?, Sect. 1.1.1],[?, Ch. 60], [?, Ex. 11.3])

Predators and prey coexist in an ecosystem. Without predators the population of prey would be governed by a simple exponential growth law. However, the growth rate of prey will decrease with increasing numbers of predators and, eventually, become negative. Similar considerations apply to the predator population and lead to an ODE model.

ODE-based model: autonomous Lotka-Volterra ODE:

$$\begin{aligned} \dot{u} &= (\alpha - \beta v)u \\ \dot{v} &= (\delta u - \gamma)v \end{aligned} \quad \leftrightarrow \quad \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \text{with} \quad \mathbf{y} = \begin{bmatrix} u \\ v \end{bmatrix}, \quad \mathbf{f}(\mathbf{y}) = \begin{bmatrix} (\alpha - \beta v)u \\ (\delta u - \gamma)v \end{bmatrix}, \tag{11.1.10}$$

with positive model parameters $\alpha, \beta, \gamma, \delta > 0$.

population densities:

$u(t) \to$ density of prey at time t,
$v(t) \to$ density of predators at time t

Right hand side vector field $\mathbf{f}$ for Lotka-Volterra ODE ▷

Solution curves are trajectories of particles carried along by velocity field $\mathbf{f}$.

(Parameter values for Fig. 388: $\alpha = 2, \beta = 1, \delta = 1, \gamma = 1$.)

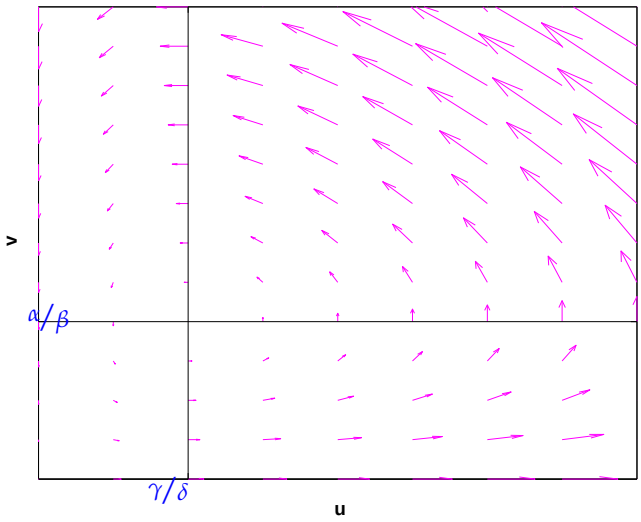


Fig. 388

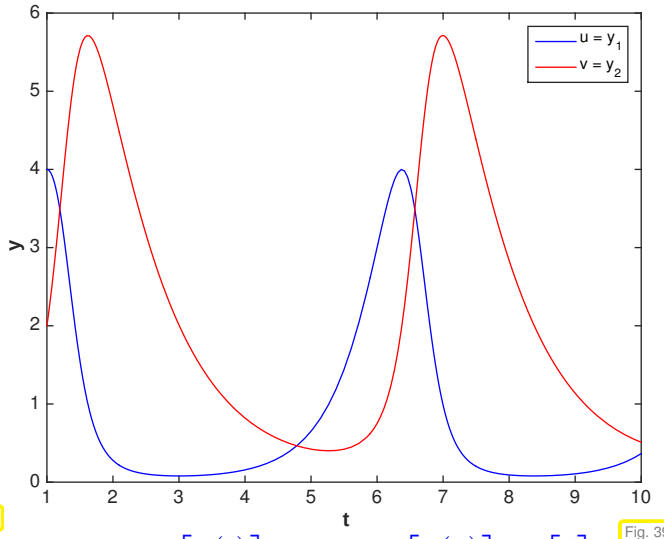


Fig. 389

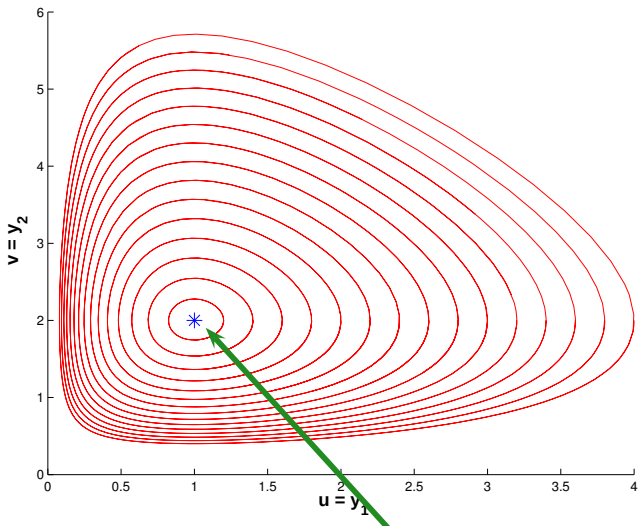


Fig. 390

Solution $\begin{bmatrix} u(t) \\ v(t) \end{bmatrix}$ for $\mathbf{y}_0 := \begin{bmatrix} u(0) \\ v(0) \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$

Solution curves for (11.1.10)

Parameter values for Fig. 390, 389: $\alpha = 2, \beta = 1, \delta = 1, \gamma = 1$

stationary point

Example 11.1.11 (Heartbeat model → [?, p. 655])

This example centers around a *phenomenological model* from physiology.

State of heart described by quantities:

$$\begin{aligned} l = l(t) &\triangleq \text{length of muscle fiber} \\ p = p(t) &\triangleq \text{electro-chemical potential} \end{aligned}$$

Phenomenological model:

$$\begin{aligned} \dot{l} &= -(l^3 - \alpha l + p)\,, \\ \dot{p} &= \beta l\,, \end{aligned} \tag{11.1.12}$$

with parameters:

$$\begin{aligned} \alpha &\triangleq \text{pre-tension of muscle fiber} \\ \beta &\triangleq \text{(phenomenological) feedback parameter} \end{aligned}$$

This is the so-called Zeeman model: it is a phenomenological model entirely based on macroscopic observations without relying on knowledge about the underlying molecular mechanisms.

Vector fields and solutions for different choices of parameters:

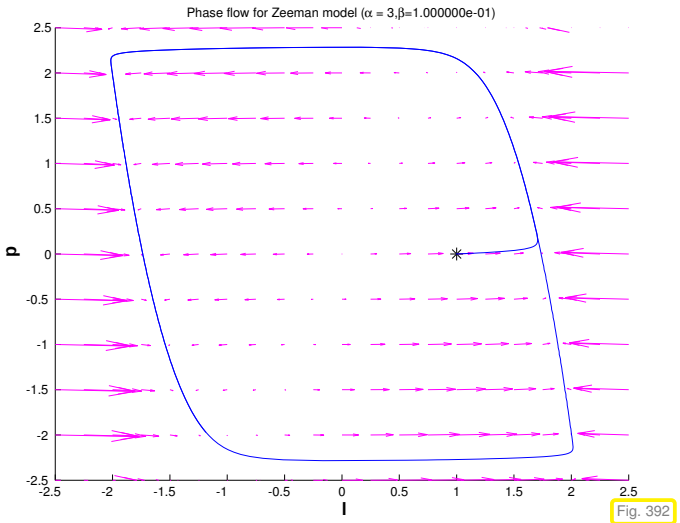


Fig. 391

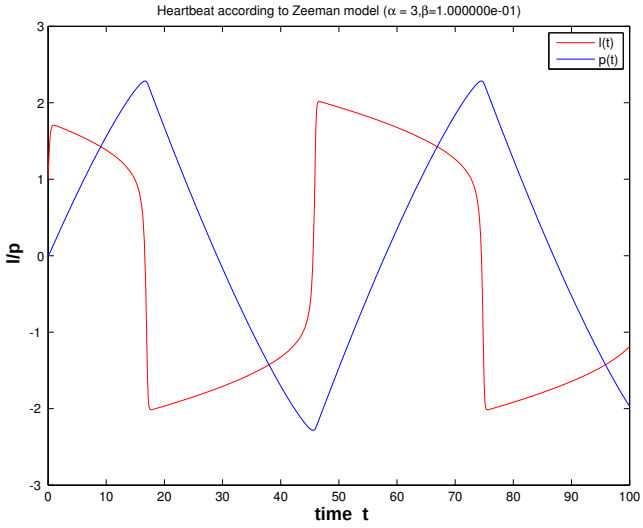


Fig. 392

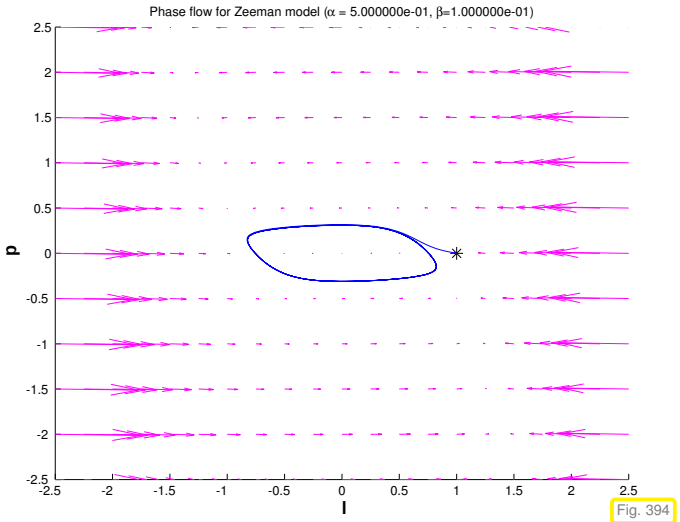


Fig. 393

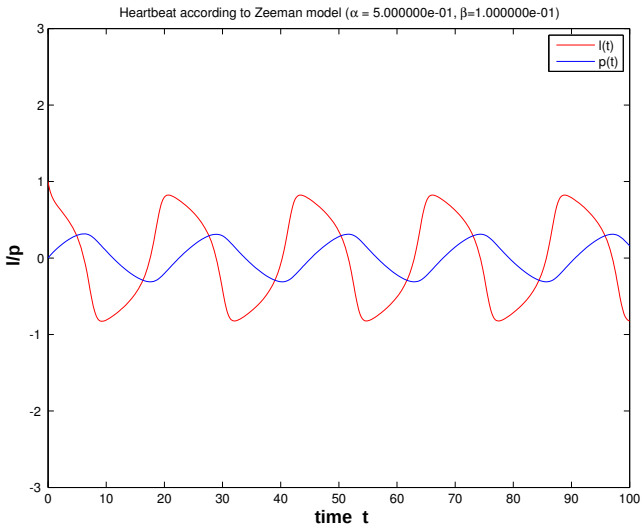


Fig. 394

Observation: $\alpha \ll 1$ ➤ ventricular fibrillation, a life-threatening condition.

Example 11.1.13 (Transient circuit simulation [?, Ch. 64])

In Chapter 1 and Chapter 8 we discussed circuit analysis as a source of linear and non-linear systems of equations, see Ex. 2.1.3 and Ex. 8.0.1. In the former example we admitted time-dependent currents and potentials, but dependence on time was confined to be "sinusoidal". This enabled us to switch to frequency domain, see (2.1.6), which gave us a complex linear system of equations for the complex nodal potentials. Yet, this trick is only possible for *linear* circuits. In the general case, circuits have to be modelled by ODEs connecting time-dependent potentials and currents. This will be briefly explained now.

The approach is transient nodal analysis, *cf.* Ex. 2.1.3, based on the Kirchhoff current law, *cf.*

$$i_R(t) - i_L(t) - i_C(t) = 0\,. \qquad (11.1.14)$$

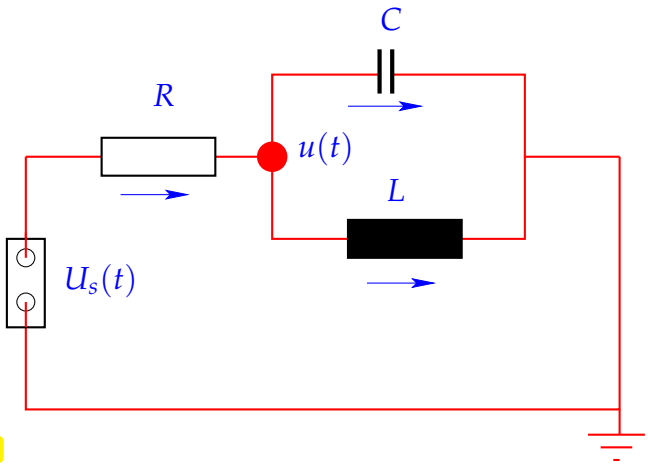


Fig. 395

Transient constitutive relations for basic linear circuit elements:

resistor: $i_R(t) = R^{-1}u_R(t)\,,$ (11.1.15)

capacitor: $i_C(t) = C\dfrac{du_C}{dt}(t)\,,$ (11.1.16)

coil: $u_L(t) = L\dfrac{di_L}{dt}(t)\,.$ (11.1.17)

Given: source voltage $U_s(t)$

To apply nodal analysis to the circuit of Fig. 395 we differentiate (11.1.14) w.r.t. t

$$\frac{di_R}{dt}(t) - \frac{di_L}{dt}(t) - \frac{di_C}{dt}(t) = 0\,,$$

and plug in the above constitutive relations for circuit elements:

$$R^{-1}\frac{du_R}{dt}(t) - L^{-1}u_L(t) - C\frac{d^2u_C}{dt^2}(t) = 0\,.$$

We continue following the policy of nodal analysis and express all voltages by potential differences between nodes of the circuit.

$$u_R(t) = U_s(t) - u(t)\,, \quad u_C(t) = u(t) - 0\,, \quad u_L(t) = u(t) - 0\,.$$

For this simple circuit there is only one node with unknown potential, see Fig. 395. Its time-dependent potential will be denoted by $u(t)$ and this is the unknown of the model, a function of time obeying the ordinary differential equation

$$R^{-1}(\dot{U}_s(t) - \dot{u}(t)) - L^{-1}u(t) - C\frac{d^2u}{dt^2}(t) = 0\,.$$

This is an autonomous 2nd-order ordinary differential equation:

$$C\ddot{u} + R^{-1}\dot{u} + L^{-1}u = R^{-1}\dot{U}_s\,. \qquad (11.1.18)$$

The attribute "2nd-order" refers to the occurrence of a second derivative with respect to time.

11.1.2 Theory of initial value problems

Supplementary reading. [**?**, Sect. 11.1], [**?**, Sect. 11.3]

(11.1.19) Initial value problems

Wit start with an abstract mathematical description

A generic Initial value problem (IVP) for a first-order ordinary differential equation (ODE) ($\rightarrow$ [?, Sect. 5.6], [?, Sect. 11.1]) can be stated as: find a function $\mathbf{y} : I \to D$ that satisfies, *cf.* Def. 11.1.3,

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \, . \tag{11.1.20}$$

- $\mathbf{f} : I \times D \mapsto \mathbb{R}^d \triangleq$ right hand side (r.h.s.) ($d \in \mathbb{N}$),
- $I \subset \mathbb{R} \triangleq$ (time)interval $\leftrightarrow$ "time variable" t,
- $D \subset \mathbb{R}^d \triangleq$ state space/phase space $\leftrightarrow$ "state variable" $\mathbf{y}$,
- $\Omega := I \times D \triangleq$ extended state space (of tupels $(t, \mathbf{y})$),
- $t_0 \in I \triangleq$ initial time, $\mathbf{y}_0 \in D \triangleq$ initial state $\succ$ initial conditions.

(11.1.21) IVPs for autonomous ODEs

Recall Def. 11.1.8: For an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, that is the right hand side $\mathbf{f}$ does not depend on time t.

Hence, for autonomous ODEs we have $I = \mathbb{R}$ and the right hand side function $\mathbf{y} \mapsto \mathbf{f}(\mathbf{y})$ can be regarded as a stationary vector field (velocity field), see Fig. 388 or Fig. 391.

An important observation: If $t \mapsto \mathbf{y}(t)$ is a solution of an autonomous ODE, then, for any $\tau \in \mathbb{R}$, also the shifted function $t \mapsto \mathbf{y}(t - \tau)$ is a solution.

$\succ$ For initial value problems for autonomous ODEs the initial time is irrelevant and therefore we can always make the "canonical choice $t_0 = 0$.

Autonomous ODEs naturally arise when modeling time-invariant systems or phenomena. All examples for Section 11.1.1 belong to this class.

(11.1.22) Autonomization: Conversion into autonomous ODE

In fact, autonomous ODEs already represent the general case, because every ODE can be converted into an autonomous one:

Idea: include time as an extra $d+1$-st component of an extended state vector.

This solution component has to grow linearly $\Leftrightarrow$ temporal derivative $= 1$

$$\mathbf{z}(t) := \begin{bmatrix} \mathbf{y}(t) \\ t \end{bmatrix} = \begin{bmatrix} \mathbf{z}' \\ z_{d+1} \end{bmatrix} : \quad \dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad \leftrightarrow \quad \dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}) \, , \quad \mathbf{g}(\mathbf{z}) := \begin{bmatrix} \mathbf{f}(z_{d+1}, \mathbf{z}') \\ 1 \end{bmatrix} .$$

$\succ$ We restrict ourselves to autonomous ODEs in the remainder of this chapter.

Remark 11.1.23 (From higher order ODEs to first order systems [?, Sect. 11.2])

An ordinary differential equation of order $n \in \mathbb{N}$ has the form

$$\mathbf{y}^{(n)} = \mathbf{f}(t, \mathbf{y}, \dot{\mathbf{y}}, \ldots, \mathbf{y}^{(n-1)}) \,. \tag{11.1.24}$$

✎ Notation: superscript $^{(n)} \hat{=}$ n-th temporal derivative t: $\frac{d^n}{dt^n}$

No special treatment of higher order ODEs is necessary, because (11.1.24) can be turned into a 1st-order ODE (a system of size nd) by adding all derivatives up to order $n-1$ as additional components to the state vector. This extended state vector $\mathbf{z}(t) \in \mathbb{R}^{nd}$ is defined as

$$\mathbf{z}(t) := \begin{bmatrix} \mathbf{y}(t) \\ \mathbf{y}^{(1)}(t) \\ \vdots \\ \mathbf{y}^{(n-1)}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \end{bmatrix} \in \mathbb{R}^{dn}: \quad (11.1.24) \quad \leftrightarrow \quad \dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}) \,, \quad \mathbf{g}(\mathbf{z}) := \begin{bmatrix} \mathbf{z}_2 \\ \mathbf{z}_3 \\ \vdots \\ \mathbf{z}_n \\ \mathbf{f}(t, \mathbf{z}_1, \ldots, \mathbf{z}_n) \end{bmatrix} . \tag{11.1.25}$$

Note that the extended system requires initial values $\mathbf{y}(t_0), \dot{\mathbf{y}}(t_0), \ldots, \mathbf{y}^{(n-1)}(t_0)$: for ODEs of order $n \in \mathbb{N}$ well-posed initial value problems need to specify initial values for the first $n-1$ derivatives.

Now we review results about existence and uniqueness of solutions of initial value problems for first-order ODEs. These are surprisingly general and do not impose severe constraints on right hand side functions.

Definition 11.1.26. Lipschitz continuous function ($\rightarrow$ [?, Def. 4.1.4])

Let $\Theta := I \times D$, $I \subset \mathbb{R}$ an interval, $D \subset \mathbb{R}^d$ an open domain. A function $\mathbf{f} : \Theta \mapsto \mathbb{R}^d$ is Lipschitz continuous (in the second argument) on Θ, if

$$\exists L > 0: \quad \|\mathbf{f}(t, \mathbf{w}) - \mathbf{f}(t, \mathbf{z})\| \leq L \|\mathbf{w} - \mathbf{z}\| \quad \forall (t, \mathbf{w}), (t, \mathbf{z}) \in \Theta \,. \tag{11.1.27}$$

Definition 11.1.28. Local Lipschitz continuity ($\rightarrow$ [?, Def. 4.1.5])

Let $\Omega := I \times D$, $I \subset \mathbb{R}$ an interval, $D \subset \mathbb{R}^d$ an open domain. A functions $\mathbf{f} : \Omega \mapsto \mathbb{R}^d$ is locally Lipschitz continuous, if for every $(t, \mathbf{y}) \in \Omega$ there is a closed box B with $(t, \mathbf{y}) \in B$ such that $\mathbf{f}$ is Lipschitz continuous on B:

$$\begin{aligned} \forall (t, \mathbf{y}) \in \Omega: \quad & \exists \delta > 0, L > 0: \\ & \|\mathbf{f}(\tau, \mathbf{z}) - \mathbf{f}(\tau, \mathbf{w})\| \leq L \|\mathbf{z} - \mathbf{w}\| \\ & \forall \mathbf{z}, \mathbf{w} \in D: \|\mathbf{z} - \mathbf{y}\| \leq \delta, \ \|\mathbf{w} - \mathbf{y}\| \leq \delta, \forall \tau \in I: |t - \tau| \leq \delta \,. \end{aligned} \tag{11.1.29}$$

The property of local Lipschitz continuity means that the function $(t, \mathbf{y}) \mapsto \mathbf{f}(t, \mathbf{y})$ has "locally finite slope" in $\mathbf{y}$.

Example 11.1.30 (A function that is not locally Lipschitz continuous)

The meaning of local Lipschitz continuity is best explained by giving an example of a function that fails to possess this property.

Consider the square root function $t \mapsto \sqrt{t}$ on the *closed* interval $[0,1]$. Its slope in $t = 0$ is infinite and so it is not locally Lipschitz continuous on $[0,1]$.

However, if we consider the square root on the *open* interval $]0,1[$, then it is locally Lipschitz continuous there.

✎ Notation: $D_{\mathbf{y}}\mathbf{f} \triangleq$ derivative of $\mathbf{f}$ w.r.t. state variable, a Jacobian $\in \mathbb{R}^{d,d}$ as defined in (8.2.11).

The next lemma gives a simple criterion for local Lipschitz continuity, which can be proved by the mean value theorem, *cf.* the proof of Lemma 8.2.12.

Lemma 11.1.31. Criterion for local Liptschitz continuity

If $\mathbf{f}$ and $D_{\mathbf{y}}\mathbf{f}$ are continuous on the extended state space Ω, then $\mathbf{f}$ is locally Lipschitz continuous ($\rightarrow$ Def. 11.1.28).

Theorem 11.1.32. Theorem of Peano & Picard-Lindelöf [?, Satz II(7.6)], [?, Satz 6.5.1], [?, Thm. 11.10], [?, Thm. 73.1]

If the right hand side function $\mathbf{f} : \hat{\Omega} \mapsto \mathbb{R}^d$ is locally Lipschitz continuous ($\rightarrow$ Def. 11.1.28) then for all initial conditions $(t_0, \mathbf{y}_0) \in \hat{\Omega}$ the IVP (11.1.20) *has a solution $\mathbf{y} \in C^1(J(t_0,\mathbf{y}_0), \mathbb{R}^d)$ with maximal (temporal) domain of definition $J(t_0,\mathbf{y}_0) \subset \mathbb{R}$.*

In light of § 11.1.22 and Thm. 11.1.32 henceforth we mainly consider

$$\text{autonomous IVPs:} \quad \boxed{\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad , \quad \mathbf{y}(0) = \mathbf{y}_0} \,, \tag{11.1.33}$$

with locally Lipschitz continuous ($\rightarrow$ Def. 11.1.28) right hand side $\mathbf{f}$.

(11.1.34) Domain of definition of solutions of IVPs

Solutions of an IVP have an *intrinsic* maximal domain of definition

! domain of definition/domain of existence $J(t_0,\mathbf{y}_0)$ usually depends on $(t_0,\mathbf{y}_0)$!

Terminology: if $J(t_0,\mathbf{y}_0) = I$ ➥ solution $\mathbf{y} : I \mapsto \mathbb{R}^d$ is global.

Notation: for autonomous ODE we always have $t_0 = 0$, and therefore we write $J(\mathbf{y}_0) := J(0,\mathbf{y}_0)$.

Example 11.1.35 (Finite-time blow-up)

Let us explain the still mysterious "maximal domain of definition" in statement of Thm. 11.1.32. I is related to the fact that every solution of an initial value problem (11.1.33) has its own largest possible time interval $J(\mathbf{y}_0) \subset \mathbb{R}$ on which it is defined naturally.

As an example we consider the autonomous scalar ($d = 1$) initial value problem, modeling "explosive growth" with a growth rate increasing linearly with the density:

$$\dot{y} = y^2 \quad , \quad y(0) = y_0 \in \mathbb{R} \, . \tag{11.1.36}$$

We choose $I = D = \mathbb{R}$. Clearly, $y \mapsto y^2$ is locally Lipschitz-continuous, but only locally! Why not globally?

We find the solutions

$$y(t) = \begin{cases} \frac{1}{y_0^{-1} - t} & \text{, if } y_0 \neq 0 \, , \\ 0 & \text{, if } y_0 = 0 \, , \end{cases} \tag{11.1.37}$$

with domains of definition

$$J(y_0) = \begin{cases}]-\infty, y_0^{-1}[& \text{, if } y_0 > 0 \, , \\ \mathbb{R} & \text{, if } y_0 = 0 \, , \\ ]y_0^{-1}, \infty[& \text{, if } y_0 < 0 \, . \end{cases}$$

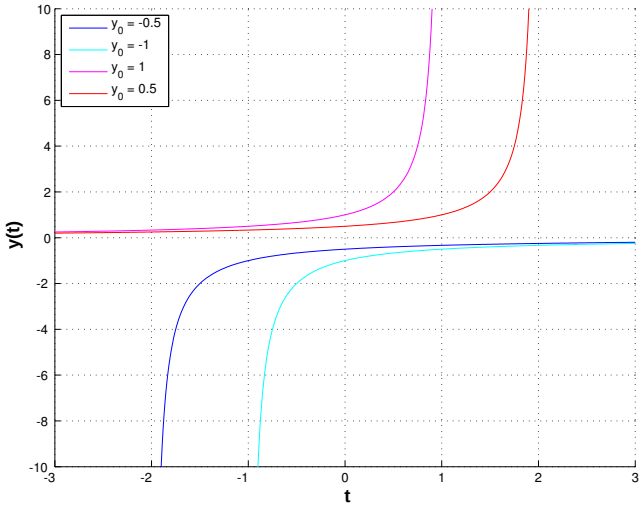


Fig. 396

In this example, for $y_0 > 0$ the solution experiences a blow-up in finite time and ceases to exists afterwards.

11.1.3 Evolution operators

For the sake of simplicity we restrict the discussion to autonomous IVPs (11.1.33) with locally Lipschitz continuous right hand side and make the following assumption. A more general treatment is given in [**?**].

Assumption 11.1.38. Global solutions

All solutions of (11.1.33) are global: $J(\mathbf{y}_0) = \mathbb{R}$ for all $\mathbf{y}_0 \in D$.

Now we return to the study of a generic ODE (11.1.2) instead of an IVP (11.1.20). We do this by temporarily changing the perspective: we fix a "time of interest" $t \in \mathbb{R} \setminus \{0\}$ and follow all trajectories for the duration t. This induces a mapping of points in state space:

$$\succ \quad \text{mapping} \quad \boldsymbol{\Phi}^t : \begin{cases} D & \mapsto & D \\ \mathbf{y}_0 & \mapsto & \mathbf{y}(t) \end{cases} \quad , \quad t \mapsto \mathbf{y}(t) \text{ solution of IVP (11.1.33)} \, ,$$

This is a well-defined mapping of the state space into itself, by Thm. 11.1.32 and Ass. 11.1.38.

Now, we may also let t vary, which spawns a *family* of mappings $\{\boldsymbol{\Phi}^t\}$ of the state space into itself. However, it can also be viewed as a mapping with two arguments, a duration t and an initial state value $\mathbf{y}_0$!

Definition 11.1.39. Evolution operator/mapping

Under Ass. 11.1.38 the mapping

$$\boldsymbol{\Phi} : \begin{cases} \mathbb{R} \times D & \mapsto & D \\ (t, \mathbf{y}_0) & \mapsto & \boldsymbol{\Phi}^t \mathbf{y}_0 := \mathbf{y}(t) \end{cases} ,$$

where $t \mapsto \mathbf{y}(t) \in C^1(\mathbb{R}, \mathbb{R}^d)$ is the unique (global) solution of the IVP $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$, is the evolution operator/mapping for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

Note that $t \mapsto \boldsymbol{\Phi}^t \mathbf{y}_0$ describes the solution of $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ for $\mathbf{y}(0) = \mathbf{y}_0$ (a trajectory). Therefore, by virtue of definition, we have

$$\frac{\partial \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathbf{f}(\boldsymbol{\Phi}^t \mathbf{y}) .$$

Example 11.1.40 (Evolution operator for Lotka-Volterra ODE (11.1.10))

For $d = 2$ the action of an evolution operator can be visualized by tracking the movement of point sets in state space. Here this is done for the Lotka-Volterra ODE (11.1.10):

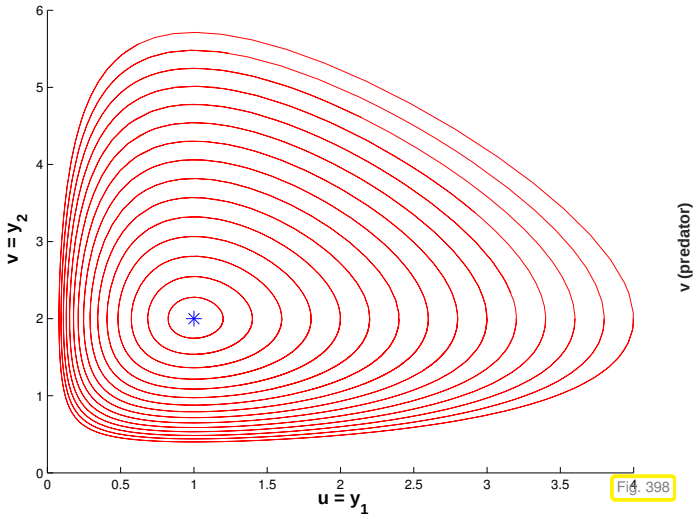


Fig. 397

trajectories $t \mapsto \boldsymbol{\Phi}^t \mathbf{y}_0$

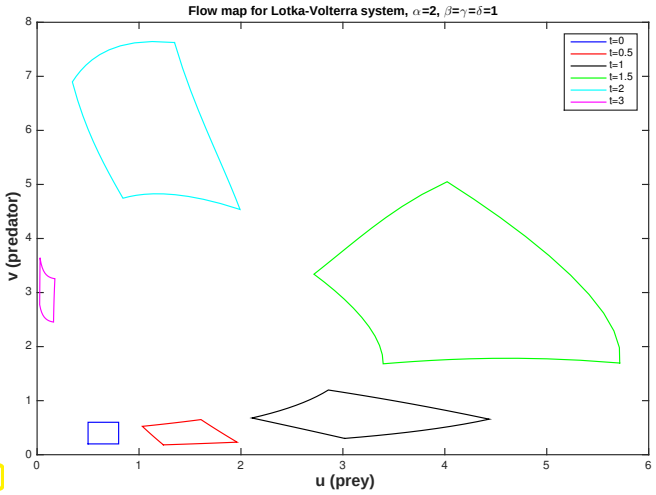


Fig. 398

state mapping $\mathbf{y} \mapsto \Phi^t \mathbf{y}$

Remark 11.1.41 (Group property of autonomous evolutions)

Under Ass. 11.1.38 the evolution operator gives rise to a group of mappings $D \mapsto D$:

$$\boldsymbol{\Phi}^s \circ \boldsymbol{\Phi}^t = \boldsymbol{\Phi}^{s+t} \quad , \quad \boldsymbol{\Phi}^{-t} \circ \boldsymbol{\Phi}^t = Id \quad \forall t \in \mathbb{R} . \tag{11.1.42}$$

This is a consequence of the uniqueness theorem Thm. 11.1.32. It is also intuitive: following an evolution up to time t and then for some more time s leads us to the same final state as observing it for the whole time $s + t$.

11.2 Introduction: Polygonal Approximation Methods

We target an initial value problem (11.1.20) for a first-order ordinary differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \, . \tag{11.1.20}$$

As usual, the right hand side function $\mathbf{f}$ may be given only in procedural form, in MATLAB as

```
function v = f(t,y),
```

or in a C++ code as an object providing an evaluation operator, see Rem. 5.1.6. An evaluation of $\mathbf{f}$ may involve costly computations.

(11.2.1) Objectives of numerical integration

Two basic tasks can be identified in the field of numerical integration = approximate solution of initial value problems for ODEs (Please distinguish from "numerical quadrature", see Chapter 7.):

(I) Given initial time t_0, final time T, and initial state $\mathbf{y}_0$ compute an approximation of $\mathbf{y}(T)$, where $t \mapsto \mathbf{y}(t)$ is the solution of (11.1.20). A corresponding function in C++ could look like

```
State solveivp(double t0,double T,State y0);
```

Here **State** is a type providing a fixed size or variable size vector $\in \mathbb{R}^d$:

```
using State = Eigen::Matrix<double,statedim,1>;
```

Here `statedim` is the dimension d of the state space that has to be known at compile time.

(II) Output an *approximate* solution $t \to \mathbf{y}_h(t)$ of (11.1.20) on $[t_0,T]$ up to final time $T \neq t_0$ for "all times" $t \in [t_0,T]$ (actually for many times $t_0 = \tau_0 < \tau_1 < \tau_2 < \cdots < \tau_{m-1} < \tau_m = T$ consecutively): "plot solution"!

```
std::vector<State>
  solveivp(State y0,const std::vector<double> &tauvec);
```

This section presents three methods that provide a piecewise linear, that is, "polygonal" approximation of solution trajectories $t \mapsto \mathbf{y}(t)$, *cf.* Ex. 5.1.10 for $d = 1$.

(11.2.2) Temporal mesh

As in Section 6.5.1 the polygonal approximation in this section will be based on a (temporal) mesh (→ § 6.5.1)

$$\mathcal{M} := \{t_0 < t_1 < t_2 < \cdots < t_{N-1} < t_N := T\} \subset [t_0,T] \, , \tag{11.2.3}$$

covering the time interval of interest between initial time t_0 and final time $T > t_0$. We assume that the interval of interest is contained in the domain of definition of the solution of the IVP: $[t_0,T] \subset J(t_0,\mathbf{y}_0)$.

11.2.1 Explicit Euler method

Example 11.2.4 (Tangent field and solution curves)

For $d = 1$ polygonal methods can be constructed by geometric considerations in the $t - y$ plane, a model for the extended state space. We explain this for the Riccati differential equation, a scalar ODE:

$$\dot{y} = y^2 + t^2 \quad \blacktriangleright \quad d = 1, \quad I, D = \mathbb{R}^+ . \tag{11.2.5}$$

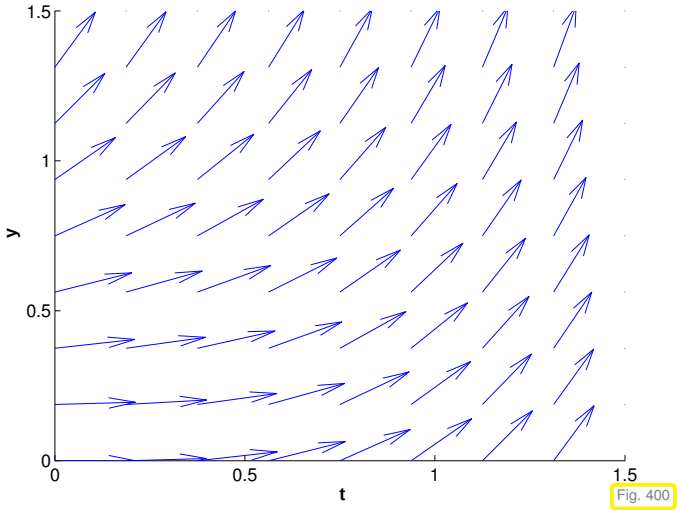


Fig. 399

tangent field

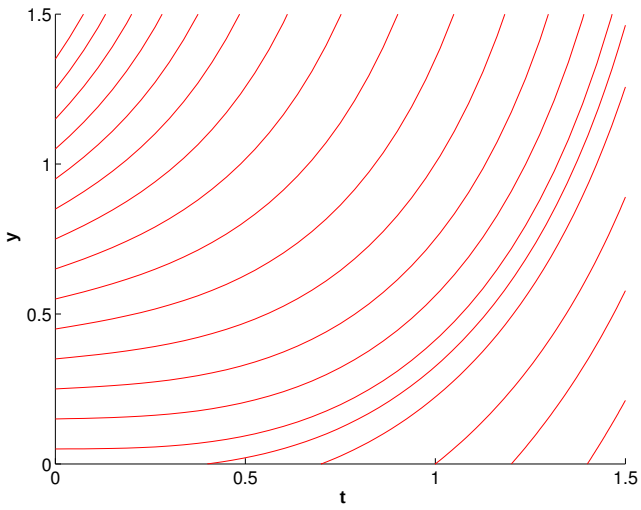
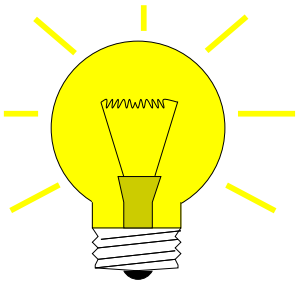


Fig. 400

solution curves

The solution curves run tangentially to the tangent field in each point of the extended state space.



Idea: "follow the tangents over short periods of time"

❶ timestepping: successive approximation of evolution on *mesh intervals* $[t_{k-1}, t_k]$, $k = 1, \ldots, N$, $t_N := T$,

❷ approximation of solution on $[t_{k-1}, t_k]$ by tangent line to solution trajectory through $(t_{k-1}, \mathbf{y}_{k-1})$.

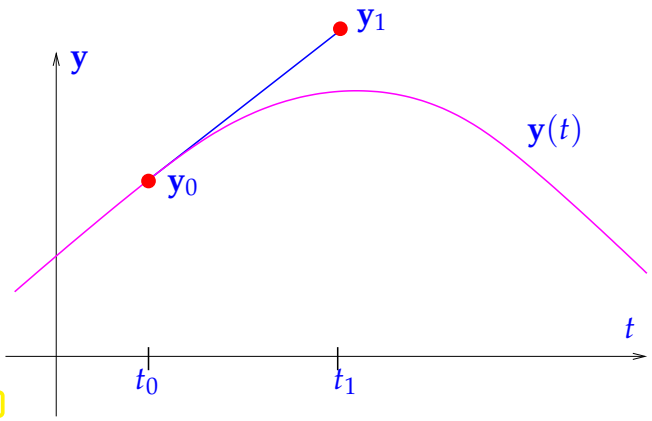


Fig. 401

explicit Euler method (Euler 1768)

◁ First step of explicit Euler method ($d = 1$):

Slope of tangent $= f(t_0, \mathbf{y}_0)$

$\mathbf{y}_1$ serves as initial value for next step!

See also [?, Ch. 74], [?, Alg. 11.4]

Example 11.2.6 (Visualization of explicit Euler method)

Temporal mesh

$$\mathcal{M} := \{t_j := j/5 : j = 0, \ldots, 5\} .$$

IVP for Riccati differential equation, see Ex. 11.2.4

$$\dot{y} = y^2 + t^2 . \qquad (11.2.5)$$

Here: $y_0 = \frac{1}{2}$, $t_0 = 0$, $T = 1$, ▷

— ≙ "Euler polygon" for uniform timestep $h = 0.2$

↦ ≙ tangent field of Riccati ODE

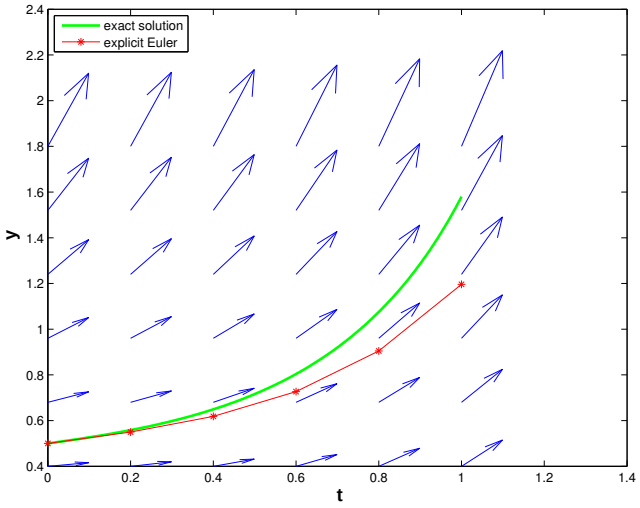


Fig. 402

Formula: When applied to a general IVP of the from (11.1.20) the explicit Euler method generates a *sequence* $(\mathbf{y}_k)_{k=0}^N$ by the recursion

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_k, \mathbf{y}_k) , \quad k = 0, \ldots, N-1 \quad ,} \qquad (11.2.7)$$

with local (size of) timestep (stepsize) $h_k := t_{k+1} - t_k$.

Remark 11.2.8 (Explicit Euler method as difference scheme)

One can obtain (11.2.7) by approximating the derivative $\frac{d}{dt}$ by a forward difference quotient on the (temporal) mesh $\mathcal{M} := \{t_0, t_1, \ldots, t_N\}$:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = \mathbf{f}(t_k, \mathbf{y}_h(t_k)) , \quad k = 0, \ldots, N-1 . \qquad (11.2.9)$$

Difference schemes follow a simple policy for the *discretization* of differential equations: replace all derivatives by difference quotients connecting solution values on a set of discrete points (the mesh).

Remark 11.2.10 (Output of explicit Euler method)

To begin with, the explicit Euler recursion (11.2.7) produces a sequence $\mathbf{y}_0, \ldots, \mathbf{y}_N$ of states. How does it deliver on the task (I) and (II) stated in § 11.2.1? By "geometric insight" we expect

$$\boxed{\mathbf{y}_k \approx \mathbf{y}(t_k)} .$$

(As usual, we use the notation $t \mapsto \mathbf{y}(t)$ for the exact solution of an IVP.)

Task (I): Easy, because $\mathbf{y}_N$ already provides an approximation of $\mathbf{y}(T)$.

Task (II): The trajectory $t \mapsto \mathbf{y}(t)$ is approximated by the piecewise linear function ('Euler polygon")

$$\mathbf{y}_h : [t_0, t_N] \to \mathbb{R}^d \quad , \quad \mathbf{y}_h(t) := \mathbf{y}_k \frac{t_{k+1} - t}{t_{k+1} - t_k} + \mathbf{y}_{k+1} \frac{t - t_k}{t_{k+1} - t_k} \quad \text{for} \quad t \in [t_k, t_{k+1}] , \qquad (11.2.11)$$

see Fig. 402. This function can easily be sampled on any grid of $[t_0, t_N]$. In fact, it is the $\mathcal{M}$-piecewise linear interpolant of the data points $(t_k, \mathbf{y}_k)$, $k = 0, \ldots, N$, see Section 5.3.2).

The same considerations apply to the methods discussed in the next two sections and will not be repeated there.

11.2.2 Implicit Euler method

Why forward difference quotient and not backward difference quotient? Let's try!

On (temporal) mesh $\mathcal{M} := \{t_0, t_1, \ldots, t_N\}$ we obtain

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = f(t_{k+1}, \mathbf{y}_h(t_{k+1})) \,, \quad k = 0, \ldots, N-1 \,. \tag{11.2.12}$$

backward difference quotient

This leads to another simple timestepping scheme analoguous to (11.2.7):

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) \,, \quad k = 0, \ldots, N-1} \,, \tag{11.2.13}$$

with local timestep (stepsize) $h_k := t_{k+1} - t_k$.

(11.2.13) = implicit Euler method

Note: (11.2.13) requires solving a (possibly non-linear) system of equations to obtain $\mathbf{y}_{k+1}$!
(➤ Terminology "implicit")

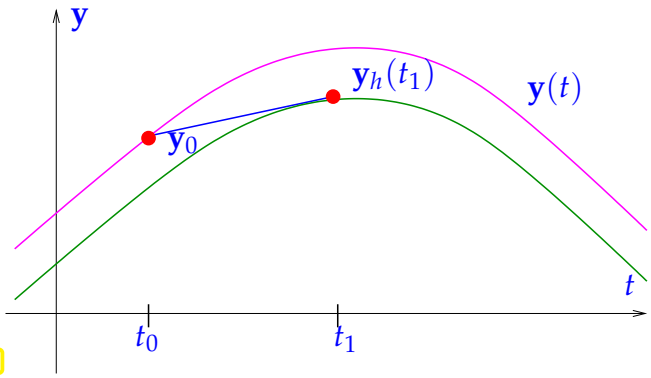


Fig. 403

Geometry of implicit Euler method:

Approximate solution through (t_0, y_0) on $[t_0, t_1]$ by
- straight line through (t_0, y_0)
- with slope $f(t_1, y_1)$

◁ — $\hat{=}$ trajectory through (t_0, y_0),
— $\hat{=}$ trajectory through (t_1, y_1),
— $\hat{=}$ tangent at — in (t_1, y_1).

Remark 11.2.14 (Feasibility of implicit Euler timestepping)

Issue: Is (11.2.13) well defined, that is, can we solve it for $\mathbf{y}_{k+1}$ and is this solution unique?

Intuition: for small timesteps $h > 0$ the right hand side of (11.2.13) is a "small perturbation of the identity".

Formal: Consider an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, assume a continuously differentiable right hand side function $\mathbf{f}$, $\mathbf{f} \in C^1(D, \mathbb{R}^d)$, and regard (11.2.13) as an h-dependent non-linear system of equations:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(t_{k+1}, \mathbf{y}_{k+1}) \quad \Leftrightarrow \quad G(h, \mathbf{y}_{k+1}) = 0 \quad \text{with} \quad G(h, \mathbf{z}) := \mathbf{z} - h\mathbf{f}(t_{k+1}, \mathbf{z}) - \mathbf{y}_k \,.$$

To investigate the solvability of this non-linear equation we start with an observation about a partial derivative of G:

$$\frac{dG}{d\mathbf{z}}(h, \mathbf{z}) = \mathbf{I} - h\, \mathrm{D}_{\mathbf{y}} \mathbf{f}(t_{k+1}, \mathbf{z}) \quad \Rightarrow \quad \frac{dG}{d\mathbf{z}}(0, \mathbf{z}) = \mathbf{I} \,.$$

In addition, $G(0, y_k) = \mathbf{0}$. Next, recall the implicit function theorem [**?**, Thm. 7.8.1]:

Theorem 11.2.15. Implicit function theorem

Let $G = G(\mathbf{x}, \mathbf{y})$ a continuously differentiable function of $\mathbf{x} \in \mathbb{R}^k$ and $\mathbf{y} \in \mathbb{R}^\ell$, defined on the open set $\Omega \subset \mathbb{R}^k \times \mathbb{R}^\ell$ with values in $\mathbb{R}^\ell$: $G : \Omega \subset \mathbb{R}^k \times \mathbb{R}^\ell \to \mathbb{R}^\ell$.

Assume that G has a zero in $\mathbf{z}_0 := \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{y}_0 \end{bmatrix} \in \Omega$, $\mathbf{x}_0 \in \mathbb{R}^k$, $\mathbf{y}_0 \in \mathbb{R}^\ell$: $G(\mathbf{z}_0) = 0$.

If the Jacobian $\frac{\partial G}{\partial \mathbf{y}}(\mathbf{p}_0) \in \mathbb{R}^{\ell,\ell}$ is invertible, then there is an open neighborhood U of $\mathbf{x}_0 \in \mathbb{R}^k$ and a continuously differentiable function $\mathbf{g} : U \to \mathbb{R}^l$ such that

$$\mathbf{g}(\mathbf{x}_0) = \mathbf{y}_0 \quad \textit{and} \quad G(\mathbf{x}, \mathbf{g}(\mathbf{x})) = 0 \quad \forall \mathbf{x} \in U \,.$$

For *sufficiently small* $|h|$ it permits us to conclude that the equation $G(h, \mathbf{z}) = 0$ defines a continuous function $\mathbf{g} = \mathbf{g}(h)$ with $\mathbf{g}(0) = \mathbf{y}_k$.

➢ for *sufficiently small* $h > 0$ the equation (11.2.13) has a unique solution $\mathbf{y}_{k+1}$.

11.2.3 Implicit midpoint method

Beside using forward or backward difference quotients, the derivative $\dot{\mathbf{y}}$ can also be approximated by the symmetric difference quotient, see also (5.2.44),

$$\dot{\mathbf{y}}(t) \approx \frac{\mathbf{y}(t+h) - \mathbf{y}(t-h)}{2h} \,. \tag{11.2.16}$$

The idea is to apply this formula in $t = \frac{1}{2}(t_k + t_{k+1})$ with $h = h_k/2$, which transforms the ODE into

$$\dot{\mathbf{y}} = f(t, \mathbf{y}) \quad \longleftrightarrow \quad \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h_k} = f\big(\tfrac{1}{2}(t_k + t_{k+1}), \mathbf{y}_h(\tfrac{1}{2}(t_{k+1} + t_{k+1}))\big) \,, \quad k = 0, \ldots, N-1 \,. \tag{11.2.17}$$

The trouble is that the value $\mathbf{y}_h(\frac{1}{2}(t_{k+1} + t_{k+1}))$ does not seem to be available, unless we recall that the approximate trajectory $t \mapsto \mathbf{y}_h(t)$ is supposed to be piecewise linear, which implies $\mathbf{y}_h(\frac{1}{2}(t_{k+1} + t_{k+1})) =$

$\frac{1}{2}(\mathbf{y}_h(t_k) + \mathbf{y}_h(t_{k+1}))$. This gives the recursion formula for the implicit midpoint method in analogy to (11.2.7) and (11.2.13):

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}\left(\tfrac{1}{2}(t_k + t_{k+1}), \tfrac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})\right) , \quad k = 0, \ldots, N-1} , \tag{11.2.18}$$

with local timestep (stepsize) $h_k := t_{k+1} - t_k$.

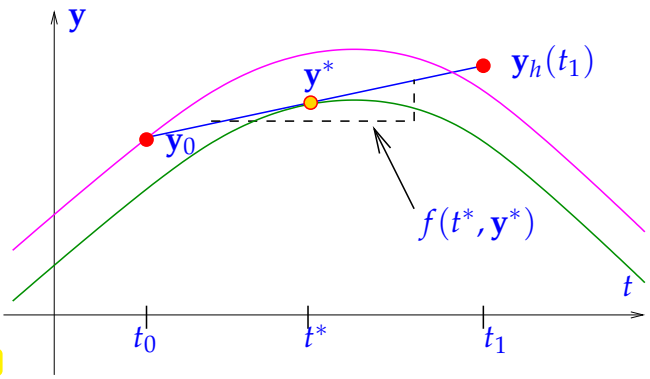


Fig. 404

Implicit midpoint method: a geometric view:

Approximaate trajectory through $(t_0, \mathbf{y}_0)$ on $[t_0, t_1]$ by

- straight line through $(t_0, \mathbf{y}_0)$
- with slope $f(t^*, \mathbf{y}^*)$, where $t^* := \frac{1}{2}(t_0 + t_1)$, $\mathbf{y}^* = \frac{1}{2}(\mathbf{y}_0 + \mathbf{y}_1)$

◁ — $\hat{=}$ trajectory through (t_0, y_0),
— $\hat{=}$ trajectory through (t^*, y^*),
— $\hat{=}$ tangent at — in (t^*, y^*).

As in the case of (11.2.13), also (11.2.18) entails solving a (non-linear) system of equations in order to obtain $\mathbf{y}_{k+1}$. Rem. 11.2.14 also holds true in this case: for sufficiently small h (11.2.18) will have a unique solution $\mathbf{y}_{k+1}$, which renders the recursion well defined.

11.3 General single step methods

Now we fit the numerical schemes introduced in the previous section into a more general class of methods for the solution of (autonomous) initial value problems (11.1.33) for ODEs. Throughout we assume that all times considered belong to the domain of definition of the unique solution $t \to \mathbf{y}(t)$ of (11.1.33), that is, for $T > 0$ we take for granted $[0, T] \subset J(\mathbf{y}_0)$ (temporal domain of definition of the solution of an IVP is explained in § 11.1.34).

11.3.1 Definition

(11.3.1) Discrete evolution operators

Recall the Euler the methods for autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$:

$$\begin{aligned} &\text{explicit Euler:} && \mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k) , \\ &\text{implicit Euler:} && \mathbf{y}_{k+1}: \quad \mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_{k+1}) . \end{aligned}$$

Both formulas, for sufficiently small h ($\to$ Rem. 11.2.14), provide a mapping

$$(\mathbf{y}_k, h_k) \mapsto \Psi(h, \mathbf{y}_k) := \mathbf{y}_{k+1} . \tag{11.3.2}$$

If $\mathbf{y}_0$ is the initial value, then $\mathbf{y}_1 := \Psi(h, \mathbf{y}_0)$ can be regarded as an approximation of $\mathbf{y}(h)$, the value returned by the evolution operator ($\to$ Def. 11.1.39) for $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ applied to $\mathbf{y}_0$ over the period h. $\mathbf{y}(t_k)$:

$$\mathbf{y}_1 = \Psi(h, \mathbf{y}_0) \quad \longleftrightarrow \quad \mathbf{y}(h) = \boldsymbol{\Phi}^h \mathbf{y}_0 \quad \succ \quad \boxed{\Psi(h, \mathbf{y}) \approx \boldsymbol{\Phi}^h \mathbf{y}} , \tag{11.3.3}$$

In a sense the polygonal approximation methods as based on approximations for the evolution operator associated with the ODE.

This is what every single step method does: it tries to approximate the evolution operator $\boldsymbol{\Phi}$ for an ODE by a mapping of the type (11.3.2).

➙ mapping $\boldsymbol{\Psi}$ from (11.3.2) is called discrete evolution.

✎ Notation: for discrete evolutions we often write $\boldsymbol{\Psi}^h\mathbf{y} := \boldsymbol{\Psi}(h,\mathbf{y})$

Remark 11.3.4 (Discretization)

The adjective "discrete" used above designates (components of) methods that attempt to approximate the solution of an IVP by a sequence of finitely many states. "Discretization" is the process of converting an ODE into a discrete model. This parlance is adopted for all procedures that reduce a "continuous model" involving ordinary or partial differential equations to a form with a finite number of unknowns.

Above we identified the discrete evolutions underlying the polygonal approximation methods. Vice versa, a mapping $\boldsymbol{\Psi}$ as given in (11.3.2) defines a single step method.

Definition 11.3.5. Single step method (for autonomous ODE) → [?, Def. 11.2]

Given a discrete evolution $\boldsymbol{\Psi} : \Omega \subset \mathbb{R}\times D \mapsto \mathbb{R}^d$, an initial state $\mathbf{y}_0$, and a temporal mesh $\mathcal{M} := \{0 =: t_0 < t_1 < \cdots < t_N := T\}$ the recursion

$$\mathbf{y}_{k+1} := \boldsymbol{\Psi}(t_{k+1}-t_k,\mathbf{y}_k)\,,\quad k=0,\ldots,N-1\,, \tag{11.3.6}$$

defines a single step method (SSM) for the autonomous IVP $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0)=\mathbf{y}_0$ on the interval $[0,T]$.

☞ In a sense, a single step method defined through its associated discrete evolution does not approximate a concrete initial value problem, but tries to approximate an ODE in the form of its evolution operator.

In MATLAB syntax a discrete evolutions can be incarnated by a function of the following form:

$\boldsymbol{\Psi}^h\mathbf{y}$ ⟷ `function y1 = discevl(h,y0)` .
(`function y1 = discevl(@(y) rhs(y),h,y0)`)

The concept of single step method according to Def. 11.3.5 can be generalized to non-autonomous ODEs, which leads to recursions of the form:

$$\mathbf{y}_{k+1} := \boldsymbol{\Psi}(t_k,t_{k+1},\mathbf{y}_k)\,,\quad k=0,\ldots,N-1\,,$$

for a discrete evolution operator $\boldsymbol{\Psi}$ defined on $I\times I\times D$.

(11.3.7) Consistent single step methods

All meaningful single step methods turn out to be modifications of the explicit Euler method (11.2.7).

Consistent discrete evolution

The discrete evolution Ψ defining a single step method according to Def. 11.3.5 and (11.3.6) for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ invariably is of the form

$$\Psi^h \mathbf{y} = \mathbf{y} + h\psi(h,\mathbf{y}) \quad \text{with} \quad \begin{array}{l} \psi : I \times D \to \mathbb{R}^d \text{ continuous,} \\ \psi(0,\mathbf{y}) = \mathbf{f}(\mathbf{y}) \ . \end{array} \tag{11.3.9}$$

Definition 11.3.10. Consistent single step methods

A single step method according to Def. 11.3.5 based on a discrete evolution of the form (11.3.9) is called consistent with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

Example 11.3.11 (Consistency of implicit midpoint method)

The discrete evolution Ψ and, hence, the function $\psi = \psi(h,\mathbf{y})$ for the implicit midpoint method are defined only implicitly, of course. Thus, consistency cannot immediately be seen from a formula for ψ.

We examine consistency of the implicit midpoint method defined by

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(\frac{1}{2}(t_k + t_{k+1}), \tfrac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})) \ , \quad k = 0,\ldots,N-1 \ . \tag{11.2.18}$$

Assume that

- the right hand side function $\mathbf{f}$ is smooth, at least $\mathbf{f} \in C^1(D)$,
- and $|h|$ is sufficiently small to guarantee the existence of a solution $\mathbf{y}_{k+1}$ of (11.2.18), see Rem. 11.2.14.

The idea is to verify (11.3.9) by formal Taylor expansion of $\mathbf{y}_{k+1}$ in h. To that end we plug (11.2.18) into itself and rely on Taylor expansion of $\mathbf{f}$:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(\tfrac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})) \overset{(11.2.18)}{=} \mathbf{y}_k + h\underbrace{\mathbf{f}(\mathbf{y}_k + \tfrac{1}{2}h\mathbf{f}(\tfrac{1}{2}(\mathbf{y}_k + \mathbf{y}_{k+1})))}_{=\psi(h,\mathbf{y}_k)} \ .$$

Since, by the implicit function theorem, $\mathbf{y}_{k+1}$ continuously depends on h and $\mathbf{y}_k$, $\psi(h,\mathbf{y}_k)$ has the desired properties, in particular $\psi(0,\mathbf{y}) = \mathbf{f}(\mathbf{y})$ is clear.

Remark 11.3.12 (Notation for single step methods)

Many authors specify a single step method by writing down the first step for a general stepsize h

$$\mathbf{y}_1 = \text{expression in } \mathbf{y}_0,\ h \text{ and } \mathbf{f} \ .$$

Actually, this fixes the underlying discrete evolution. Also this course will sometimes adopt this practice.

(11.3.13) Output of single step methods

Here we resume and continue the discussion of Rem. 11.2.10 for general single step methods according to Def. 11.3.5. Assuming unique solvability of the systems of equations faced in each step of an implicit method, every single step method based on a mesh $\mathcal{M} = \{0 = t_0 < t_1 < \cdots < t_N := T\}$ produces a finite sequence $(\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_N)$ of states, where the first agrees with the initial state $\mathbf{y}_0$.

We expect that the states provide a pointwise approximation of the solution trajectory $t \to \mathbf{y}(t)$:

$$\mathbf{y}_k \approx \mathbf{y}(t_k) , \quad k = 1, \ldots, N .$$

Thus task (I) from § 11.2.1, computing an approximation for $\mathbf{y}(T)$, is again easy: output $\mathbf{y}_N$ as an approximation of $\mathbf{y}(T)$.

Task (II) from § 11.2.1, computing the solution trajectory, requires interpolation of the data points $(t_k, \mathbf{y}_k)$ using some of the techniques presented in Chapter 5. The natural option is $\mathcal{M}$-piecewise polynomial interpolation, generalizing the polygonal approximation (11.2.11) used in Section 11.2.

Note that from the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ the derivatives $\dot{\mathbf{y}}_h(t_k) = \mathbf{f}(\mathbf{y}_k)$ are available without any further approximation. This facilitates cubic Hermite interpolation ($\to$ Def. 5.4.1), which yields

$$\mathbf{y}_h \in C^1([0,T]): \quad \mathbf{y}_{h\,|[x_{k-1},x_k]} \in \mathcal{P}_3 , \quad \mathbf{y}_h(t_k) = \mathbf{y}_k , \quad \frac{d\mathbf{y}_h}{dt}(t_k) = \mathbf{f}(\mathbf{y}_k) .$$

Summing up, an approximate trajectory $t \mapsto \mathbf{y}_h(t)$ is built in two stages:

(i) Compute sequence $(\mathbf{y}_k)_k$ by running the single step method.

(ii) Post-process the obtained sequence, usually by applying interpolation, to get $\mathbf{y}_h$.

11.3.2 Convergence of single step methods

Supplementary reading. See [**?**, Sect. 11.5] and [**?**, Sect. 11.3] for related presentations.

(11.3.14) Discretization error of single step methods

Errors in numerical integration are called discretization errors, *cf.* Rem. 11.3.4.

Depending on the objective of numerical integration as stated in § 11.2.1 different notions of discretization error appropriate

(I) If only the solution at final time is sought, the discretization error is

$$\epsilon_N := \|\mathbf{y}(T) - \mathbf{y}_N\| ,$$

where $\|\cdot\|$ is some vector norm on $\mathbb{R}^d$.

(II) If we want to approximate the solution trajectory for (11.1.33) the discretization error is the function

$$t \mapsto \mathbf{e}(t) \quad , \quad \mathbf{e}(t) := \mathbf{y}(t) - \mathbf{y}_h(t) \, ,$$

where $t \mapsto \mathbf{y}_h(t)$ is the approximate trajectory obtained by post-processing, see § 11.3.13. In this case accuracy of the method is gauged by looking at norms of the function $\mathbf{e}$, see § 5.2.65 for examples.

(III) Between (I) and (II) is the pointwise discretization error, which is the sequence (grid function)

$$\mathbf{e} : \mathcal{M} \to D \quad , \quad \mathbf{e}_k := \mathbf{y}(t_k) - \mathbf{y}_k \, , \quad k = 0, \ldots, N \, . \tag{11.3.15}$$

In this case we may consider the maximum error in the mesh points

$$\|(\mathbf{e})\|_\infty := \max_{k \in \{1,\ldots,N\}} \|\mathbf{e}_k\| \, ,$$

where $\|\cdot\|$ is a suitable vector norm on $\mathbb{R}^d$, usually the Euclidean vector norm.

(11.3.16) Asymptotic convergence of single step methods

Once the discrete evolution $\boldsymbol{\Psi}$ associated with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ is specified, the single step method according to Def. 11.3.5 is fixed. The only way to control the accuracy of the solution $\mathbf{y}_N$ or $t \mapsto \mathbf{y}_h(t)$ is through the selection of the mesh $\mathcal{M} = \{0 = t_0 < t_1 < \cdots < t_N = T\}$.

Hence we study convergence of single step methods for families of meshes $\{\mathcal{M}_\ell\}$ and track the decay of (a norm) of the discretization error ($\rightarrow$ § 11.3.14) as a function of the number $N := \sharp\mathcal{M}$ of mesh points. In other words, we examine h-convergence. We already did this in the case of piecewise polynomial interpolation in Section 6.5.1 and composite numerical quadrature in Section 7.4.

When investigating asymptotic convergence of single step methods we often resort to families of equidistant meshes of $[0, T]$:

$$\mathcal{M}_N := \{t_k := \frac{k}{N} T : k = 0 \ldots, N\} \, . \tag{11.3.17}$$

We also call this the use of uniform timesteps of size $h := \frac{T}{N}$.

Example 11.3.18 (Speed of convergence of Euler methods)

The setting for this experiment is a follows:

- We consider the following IVP for the logistic ODE, see Ex. 11.1.5

$$\dot{y} = \lambda y(1 - y) \quad , \quad y(0) = 0.01 \, .$$

- We apply explicit and implicit Euler methods (11.2.7)/(11.2.13) with uniform timestep $h = 1/N$, $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$.
- Monitored: Error at final time $E(h) := |y(1) - y_N|$

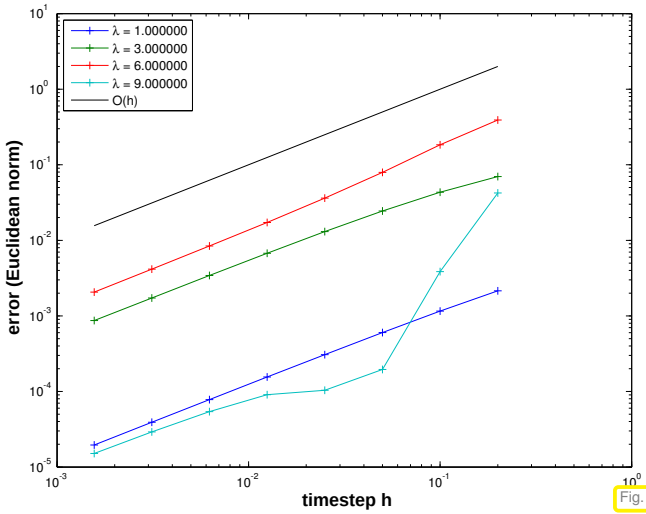


Fig. 405

explicit Euler method

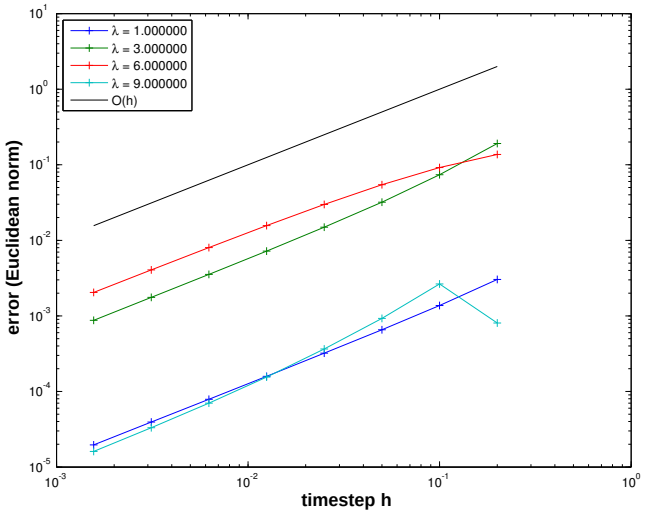


Fig. 406

implicit Euler method

◁ $O(N^{-1}) = O(h)$ algebraic convergence in both cases for $h \to 0$

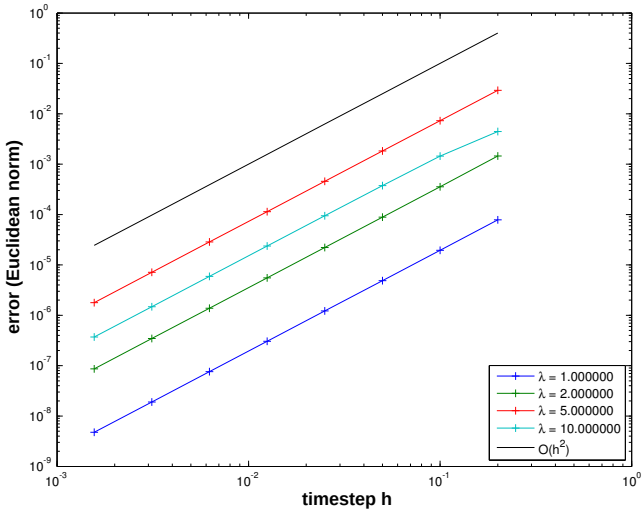


Fig. 407

However, polygonal approximation methods can do better:

◁ We study the convergence of the implicit midpoint method (11.2.18) in the above setting.

We observe algebraic convergence $O(h^2)$ for $h \to 0$.

Parlance: based on the observed rate of algebraic convergence, the two Euler methods are said to "converge with first order", whereas the implicit midpoint method is called "second-order convergent".

The observations made for polygonal timestepping methods reflect a general pattern:

Algebraic convergence of single step methods

Consider numerical integration of an initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \, , \tag{11.1.20}$$

with sufficiently smooth right hand side function $\mathbf{f} : I \times D \to \mathbb{R}^d$.

Then customary single step methods (→ Def. 11.3.5) will enjoy *algebraic convergence in the meshwidth*, more precisely, see [**?**, Thm. 11.25],

there is a $p \in \mathbb{N}$ such that the sequence $(\mathbf{y}_k)_k$ generated by the single step method for $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ on a mesh $\mathcal{M} := \{t_0 < t_1 < \cdots < t_N = T\}$ satisfies

$$\max_k \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq Ch^p \quad \text{for} \quad h := \max_{k=1,\ldots,N} |t_k - t_{k-1}| \to 0 \, , \tag{11.3.20}$$

with $C > 0$ independent of $\mathcal{M}$

Definition 11.3.21. Order of a single step method

The minimal integer $p \in \mathbb{N}$ for which (11.3.20) holds for a single step method when applied to an ODE with (sufficiently) smooth right hand side, is called the order of the method.

As in the case of quadrature rules ($\rightarrow$ Def. 7.3.1) their order is the principal intrinsic indicator for the "quality" of a single step method.

(11.3.22) Convergence analysis for the explicit Euler method [?, Ch. 74]

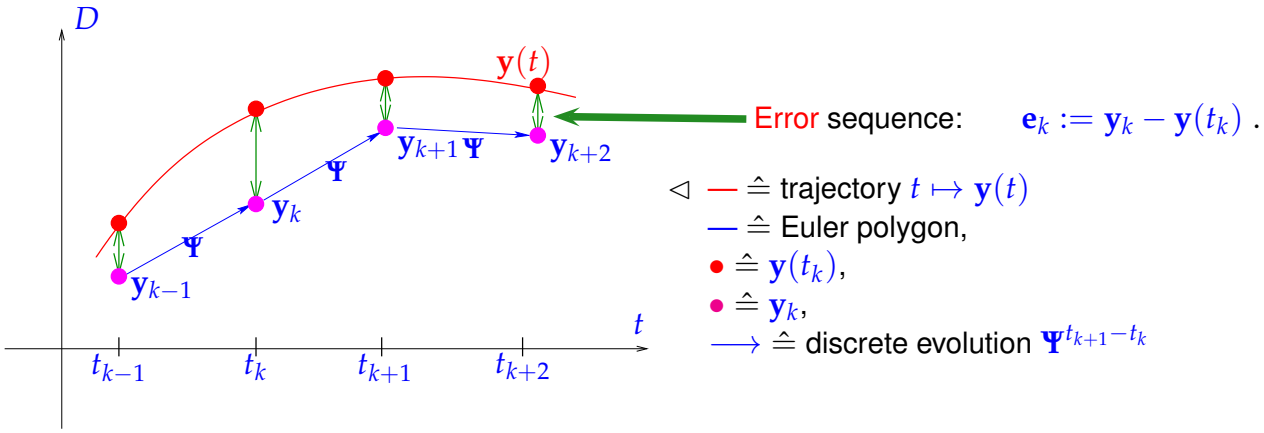
We consider the explicit Euler method (11.2.7) on a mesh $\mathcal{M} := \{0 = t_0 < t_1 < \cdots < t_N = T\}$ for a generic autonomous IVP (11.1.20) with sufficiently smooth and (*globally*) Lipschitz continuous $\mathbf{f}$, that is,

$$\exists L > 0: \quad \|\mathbf{f}(\mathbf{y}) - \mathbf{f}(\mathbf{z})\| \leq L\|\mathbf{y} - \mathbf{z}\| \quad \forall \mathbf{y}, \mathbf{z} \in D\,, \tag{11.3.23}$$

and exact solution $t \mapsto \mathbf{y}(t)$. Throughout we assume that solutions of $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ are defined on $[0, T]$ for all initial states $\mathbf{y}_0 \in D$.

Recall: recursion for explicit Euler method

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \mathbf{f}(\mathbf{y}_k)\,, \quad k = 1, \ldots, N-1\,. \tag{11.2.7}$$



① Abstract **splitting of error**:

Here and in what follows we rely on the abstract concepts of the evolution operator $\boldsymbol{\Phi}$ associated with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ ($\rightarrow$ Def. 11.1.39) and discrete evolution operator $\boldsymbol{\Psi}$ defining the explicit Euler single step method, see Def. 11.3.5:

$$\text{(11.2.7)} \quad \Rightarrow \quad \boldsymbol{\Psi}^h \mathbf{y} = \mathbf{y} + h\mathbf{f}(\mathbf{y})\,. \tag{11.3.24}$$

We argue that in this context the abstraction pays off, because it helps elucidate a general technique for the convergence analysis of single step methods.

Fundamental error splitting

$$\begin{aligned}\mathbf{e}_{k+1} &= \boldsymbol{\Psi}^{h_k}\mathbf{y}_k - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k)\\ &= \underbrace{\boldsymbol{\Psi}^{h_k}\mathbf{y}_k - \boldsymbol{\Psi}^{h_k}\mathbf{y}(t_k)}_{\text{propagated error}}\\ &\quad + \underbrace{\boldsymbol{\Psi}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k)}_{\text{one-step error}} .\end{aligned} \tag{11.3.25}$$

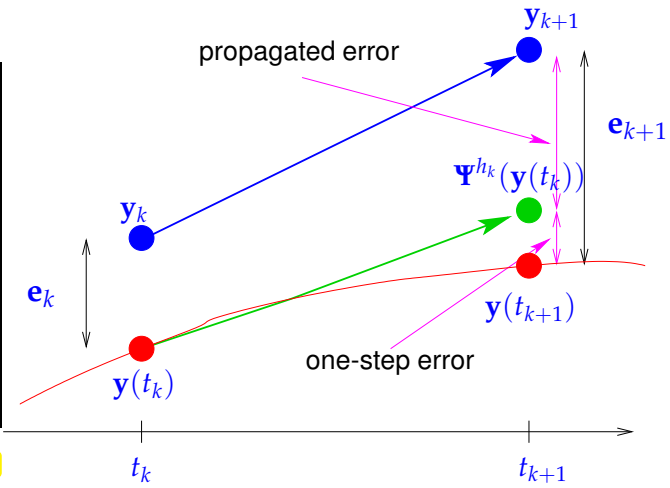


Fig. 408

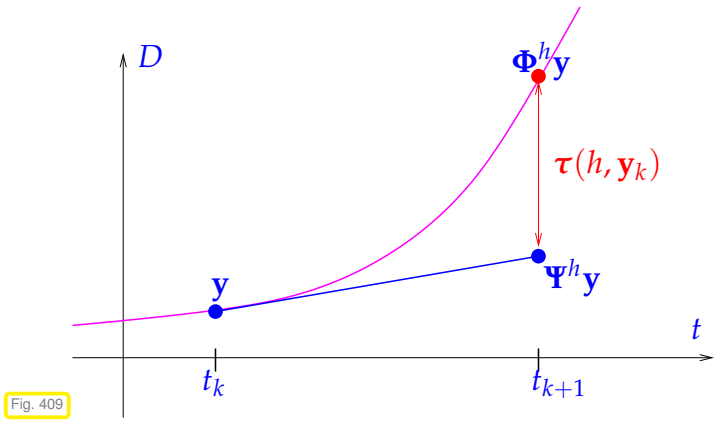


Fig. 409

A generic one-step error expressed through continuous and discrete evolutions:

$$\boldsymbol{\tau}(h,\mathbf{y}) := \boldsymbol{\Psi}^h\mathbf{y} - \boldsymbol{\Phi}^h\mathbf{y} . \tag{11.3.26}$$

◁ geometric visualisation of one-step error for explicit Euler method (11.2.7), *cf.* Fig. 401, $h := t_{k+1} - t_k$

—: solution trajectory through $(t_k, \mathbf{y})$

② **Estimate for one-step error $\boldsymbol{\tau}(h_k, \mathbf{y}(t_k))$:**

Geometric considerations: distance of a smooth curve and its tangent shrinks as the square of the distance to the intersection point (curve locally looks like a parabola in the $\xi - \eta$ coordinate system, see Fig. 411).

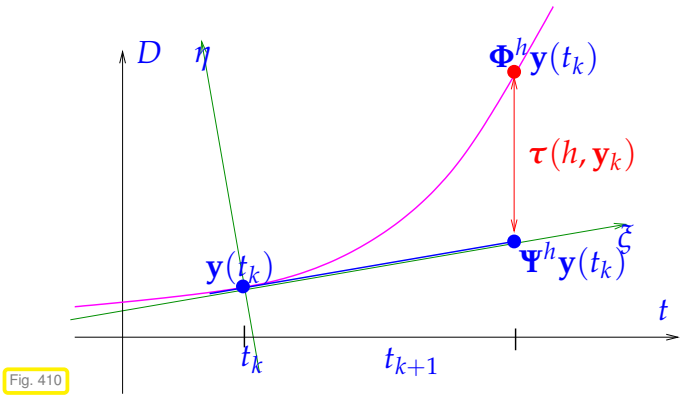


Fig. 410

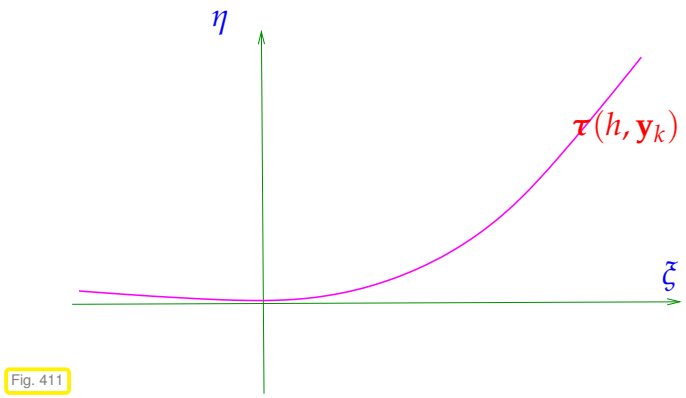


Fig. 411

The geometric considerations can be made rigorous by analysis: recall Taylor's formula for the function $\mathbf{y} \in C^{K+1}$ [**?**, Satz 5.5.1]:

$$\mathbf{y}(t+h) - \mathbf{y}(t) = \sum_{j=0}^{K} \mathbf{y}^{(j)}(t)\frac{h^j}{j!} + \underbrace{\int_t^{t+h} \mathbf{y}^{(K+1)}(\tau)\frac{(t+h-\tau)^K}{K!}\,\mathrm{d}\tau}_{= \dfrac{\mathbf{y}^{(K+1)}(\xi)}{K!}h^{K+1}} , \tag{11.3.27}$$

for some $\xi \in [t, t+h]$. We conclude that, if $\mathbf{y} \in C^2([0,T])$, which is ensured for smooth $\mathbf{f}$, see Lemma 11.1.4, then

$$\mathbf{y}(t_{k+1}) - \mathbf{y}(t_k) = \dot{\mathbf{y}}(t_k)h_k + \tfrac{1}{2}\ddot{\mathbf{z}}(\xi_k)h_k^2 = \mathbf{f}(\mathbf{y}(t_k))h_k + \tfrac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 ,$$

for some $t_k \le \xi_k \le t_{k+1}$.This leads to an expression for the one-step error from (11.3.26)

$$\begin{aligned}\boldsymbol{\tau}(h_k,\mathbf{y}(t_k)) &= \boldsymbol{\Psi}^{h_k}\mathbf{y}(t_k) - \mathbf{y}(t_{k+1}) \\ &\overset{(11.3.24)}{=} \mathbf{y}(t_k) + h_k\mathbf{f}(\mathbf{y}(t_k)) - \mathbf{y}(t_k) - \mathbf{f}(\mathbf{y}(t_k))h_k + \tfrac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 \\ &= \tfrac{1}{2}\ddot{\mathbf{y}}(\xi_k)h_k^2 \,.\end{aligned} \tag{11.3.28}$$

Sloppily speaking, we observe $\boxed{\boldsymbol{\tau}(h_k,\mathbf{y}(t_k)) = O(h_k^2)}$ uniformly for $h_k \to 0$.

③ **Estimate for the propagated error** from (11.3.25)

$$\begin{aligned}\left\|\boldsymbol{\Psi}^{h_k}\mathbf{y}_k - \boldsymbol{\Psi}^{h_k}\mathbf{y}(t_k)\right\| &= \|\mathbf{y}_k + h_k\mathbf{f}(\mathbf{y}_k) - \mathbf{y}(t_k) - h_k\mathbf{f}(\mathbf{y}(t_k))\| \\ &\overset{(11.3.23)}{\le} (1+Lh_k)\|\mathbf{y}_k - \mathbf{y}(t_k)\| \,.\end{aligned} \tag{11.3.29}$$

③ Obtain *recursion* for error norms $\epsilon_k := \|\mathbf{e}_k\|$ by $\triangle$-inequality:

$$\boxed{\epsilon_{k+1} \le (1+h_kL)\epsilon_k + \rho_k \,, \quad \rho_k := \tfrac{1}{2}h_k^2 \max_{t_k\le\tau\le t_{k+1}} \|\ddot{\mathbf{y}}(\tau)\|} \,. \tag{11.3.30}$$

Taking into account $\epsilon_0 = 0$, this leads to

$$\epsilon_k \le \sum_{l=1}^{k}\prod_{j=1}^{l-1}(1+Lh_j)\,\rho_l \,, \quad k = 1,\ldots,N \,. \tag{11.3.31}$$

Use the elementary estimate $(1+Lh_j) \le \exp(Lh_j)$ (by convexity of exponential function):

$$\text{(11.3.31)} \quad \Rightarrow \quad \epsilon_k \le \sum_{l=1}^{k}\prod_{j=1}^{l-1}\exp(Lh_j)\cdot\rho_l = \sum_{l=1}^{k}\exp(L\textstyle\sum_{j=1}^{l-1}h_j)\rho_l \,.$$

Note: $\sum_{j=1}^{l-1} h_j \le T$ for final time T and conclude

$$\epsilon_k \le \exp(LT)\sum_{l=1}^{k}\rho_l \le \exp(LT)\max_k\frac{\rho_k}{h_k}\sum_{l=1}^{k}h_l \le T\exp(LT)\max_{l=1,\ldots,k}h_l\cdot\max_{t_0\le\tau\le t_k}\|\ddot{\mathbf{y}}(\tau)\| \,.$$

$$\blacktriangleright \quad \|\mathbf{y}_k - \mathbf{y}(t_k)\| \le T\exp(LT)\max_{l=1,\ldots,k}h_l\cdot\max_{t_0\le\tau\le t_k}\|\ddot{\mathbf{y}}(\tau)\| \,. \tag{11.3.32}$$

We can summarize the insight gleaned through this theoretical analysis as follows:

Total error arises from accumulation of propagated one-step errors!

First conclusions from (11.3.32):

- error bound $= O(h)$, $h := \max_l h_l$ (➤ 1st-order algebraic convergence)
- Error bound grows *exponentially* with the length T of the integration interval.

(11.3.33) One-step error and order of a single step method

In the analysis of the global discretization error of the explicit Euler method in § 11.3.22 a one-step error of size $O(h_k^2)$ led to a total error of $O(h)$ through the effect of error accumulation over $N \approx h^{-1}$ steps. This relationship remains valid for almost all single step methods:

Consider an IVP (11.1.20) with solution $t \mapsto \mathbf{y}(t)$ and a single step method defined by the discrete evolution $\boldsymbol{\Psi}$ ($\rightarrow$ Def. 11.3.5). If the *one-step error along the solution trajectory* satisfies ($\boldsymbol{\Phi}$ is the evolution map associated with the ODE, see Def. 11.1.39)

$$\left\| \boldsymbol{\Psi}^h \mathbf{y}(t) - \boldsymbol{\Phi}^h \mathbf{y}(t) \right\| \leq Ch^{p+1} \quad \forall h \text{ sufficiently small}, \ t \in [0,T] \ ,$$

for some $p \in \mathbb{N}$ and $C > 0$, then, usually,

$$\max_k \|\mathbf{y}_k - \mathbf{y}(t_k)\| \leq \overline{C} h_{\mathcal{M}}^p \ ,$$

with $\overline{C} > 0$ independent of the temporal mesh $\mathcal{M}$.

A rigorous statement as a theorem would involve some particular assumptions on $\boldsymbol{\Psi}$, which we do not want to give here. These assumptions are satisfied, for instance, for all the methods presented in the sequel.

11.4 Explicit Runge-Kutta Methods

Supplementary reading. [**?**, Sect. 11.6], [**?**, Ch. 76], [**?**, Sect. 11.8]

So far we only know first and second order methods from 11.2: the explicit and implicit Euler method (11.2.7) and (11.2.13), respectively, are of first order, the implicit midpoint rule of second order. We observed this in Ex. 11.3.18 and it can be proved rigorously for all three methods adapting the arguments of § 11.3.22.

Thus, barring the impact of roundoff, the low-order polygonal approximation methods are guaranteed to achieve any prescribed accuracy provided that the mesh is fine enough. Why should we need any other timestepping schemes?

Remark 11.4.1 (Rationale for high-order single step methods ***cf.*** **[?, Sect. 11.5.3])**

We argue that the use of higher-order timestepping methods is highly *advisable for the sake of efficiency*. The reasoning is very similar to that of Rem. 7.3.48, when we considered numerical quadrature. The reader is advised to study that remark again.

As we saw in § 11.3.16 error bounds for single step methods for the solution of IVPs will inevitably feature unknown constants "$C > 0$". Thus they do **not** give useful information about the discretization error for

a concrete IVP and mesh. Hence, it is too ambitious to ask how many timesteps are needed so that $\|\mathbf{y}(T) - \mathbf{y}_N\|$ stays below a prescribed bound, *cf.* the discussion in the context of numerical quadrature.

However, an easier question can be answered by *asymptotic estimates* l ike (11.3.20):

What extra computational effort buys a prescribed *reduction of the error*?

(also recall the considerations in Section 8.3.3!)

The usual concept of "computational effort" for single step methods ($\rightarrow$ Def. 11.3.5) is as follows

Computational effort $\sim$ total number of $\mathbf{f}$-evaluations for approximately solving the IVP,
$\sim$ number of timesteps, if evaluation of discete evolution $\boldsymbol{\Psi}^h$ ($\rightarrow$ Def. 11.3.5) requires fixed number of $\mathbf{f}$-evaluations,
$\sim h^{-1}$, in the case of uniform timestep size $h > 0$ (equidistant mesh (11.3.17)).

Now, let us consider a single step method of order $p \in \mathbb{N}$, employed with a uniform timestep h_{old}. We focus on the maximal discretization error in the mesh points, see § 11.3.14. As in (7.3.49) we assume that the asymptotic error bounds are *sharp*:

$$\operatorname{err}(h) \approx Ch^p \quad \text{for small meshwidth } h > 0 \,,$$

with a "generic constant" $C > 0$ independent of the mesh.

Goal: $\dfrac{\operatorname{err}(h_{\text{new}})}{\operatorname{err}(h_{\text{old}})} \overset{!}{=} \dfrac{1}{\rho}$ for reduction factor $\rho > 1$.

(11.3.20) $\Rightarrow$ $\dfrac{h_{\text{new}}^p}{h_{\text{old}}^p} \overset{!}{=} \dfrac{1}{\rho}$ $\Leftrightarrow$ $\boxed{h_{\text{new}} = \rho^{-1/p} h_{\text{old}}}$.

For single step method of order $p \in \mathbb{N}$

increase effort by factor $\rho^{1/p}$ ⟹ reduce error by factor $\rho > 1$

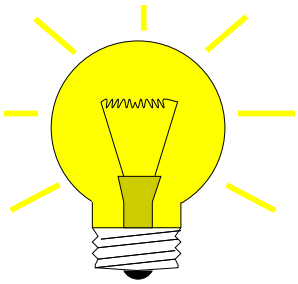
☞ the larger the order p, the less effort for a prescribed reduction of the error!

We remark that another (minor) rationale for using higher-order methods [**?**, Sect. 11.5.3]: curb impact of roundoff errors ($\rightarrow$ Section 1.5.3) accumulating during timestepping.

(11.4.2) Bootstrap construction of explicit single step methods

Now we will build a class of methods that are explicit and achieve orders $p > 2$. The starting point is a simple *integral equation* satisfied by any solution $t \mapsto \mathbf{y}(t)$ of an initial value problems for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$:

$$\text{IVP:} \quad \begin{aligned} \dot{\mathbf{y}}(t) &= \mathbf{f}(t, \mathbf{y}(t)) \,, \\ \mathbf{y}(t_0) &= \mathbf{y}_0 \end{aligned} \quad \Rightarrow \quad \mathbf{y}(t_1) = \mathbf{y}_0 + \int_{t_0}^{t_1} \mathbf{f}(\tau, \mathbf{y}(\tau)) \, \mathrm{d}\tau$$



Idea: approximate the integral by means of s-point quadrature formula ($\rightarrow$ Section 7.1, defined on the reference interval $[0,1]$) with nodes $c_1,\ldots,c_s$, weights $b_1,\ldots,b_s$.

$$\mathbf{y}(t_1) \approx \mathbf{y}_1 = \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{f}(t_0 + c_i h, \boxed{\mathbf{y}(t_0 + c_i h)}) , \quad h := t_1 - t_0 . \tag{11.4.3}$$

Obtain these values by bootstrapping

"Bootstrapping" = use the same idea in a simpler version to get $\mathbf{y}(t_0 + c_i h)$, noting that these values can be replaced by other approximations obtained by methods already constructed (this approach will be elucidated in the next example).

What error can we afford in the approximation of $\mathbf{y}(t_0 + c_i h)$ (under the assumption that $\mathbf{f}$ is Lipschitz continuous)? We take the cue from the considerations in § 11.3.22.

Goal: aim for one-step error bound $\mathbf{y}(t_1) - \mathbf{y}_1 = O(h^{p+1})$

Note that there is a factor h in front of the quadrature sum in (11.4.3). Thus, our goal can already be achieved, if only

$$\mathbf{y}(t_0 + c_i h) \text{ is approximated up to an error } O(h^p),$$

again, because in (11.4.3) a factor of size h multiplies $\mathbf{f}(t_0 + c_i, \mathbf{y}(t_0 + c_i h))$.

This is accomplished by a less accurate discrete evolution than the one we are about to build. Thus, we can construct discrete evolutions of higher and higher order, in turns, starting with the explicit Euler method. All these methods will be explicit, that is, $\mathbf{y}_1$ can be computed directly from point values of $\mathbf{f}$.

Example 11.4.4 (Simple Runge-Kutta methods by quadrature & boostrapping)

Now we apply the boostrapping idea outlined above. We write $\mathbf{k}_\ell \in \mathbb{R}^d$ for the approximations of $\mathbf{y}(t_0 + c_i h)$.

- Quadrature formula = trapezoidal rule (7.2.5):

$$Q(f) = \tfrac{1}{2}(f(0) + f(1)) \quad \leftrightarrow \quad s = 2\text{: } c_1 = 0, c_2 = 1 , \quad b_1 = b_2 = \frac{1}{2} , \tag{11.4.5}$$

  and $\mathbf{y}(t_1)$ approximated by explicit Euler step (11.2.7)

$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0) , \quad \mathbf{k}_2 = \mathbf{f}(t_0 + h, \mathbf{y}_0 + h\mathbf{k}_1) , \quad \mathbf{y}_1 = \mathbf{y}_0 + \tfrac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2) . \tag{11.4.6}$$

  (11.4.6) = explicit trapezoidal method (for numerical integration of ODEs).

- Quadrature formula $\rightarrow$ simplest Gauss quadrature formula = midpoint rule ($\rightarrow$ Ex. 7.2.3) & $\mathbf{y}(\frac{1}{2}(t_1 + t_0))$ approximated by explicit Euler step (11.2.7)

$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0) , \quad \mathbf{k}_2 = \mathbf{f}(t_0 + \tfrac{h}{2}, \mathbf{y}_0 + \tfrac{h}{2}\mathbf{k}_1) , \quad \mathbf{y}_1 = \mathbf{y}_0 + h\mathbf{k}_2 . \tag{11.4.7}$$

  (11.4.7) = explicit midpoint method (for numerical integration of ODEs) [**?**, Alg. 11.18].

Example 11.4.8 (Convergence of simple Runge-Kutta methods)

We perform an empiric study of the order of the explicit single step methods constructed in Ex. 11.4.4.

- IVP: $\dot{y} = 10y(1-y)$ (logistic ODE (11.1.6)), $y(0) = 0.01$, $T = 1$,
- Explicit single step methods, uniform timestep h.

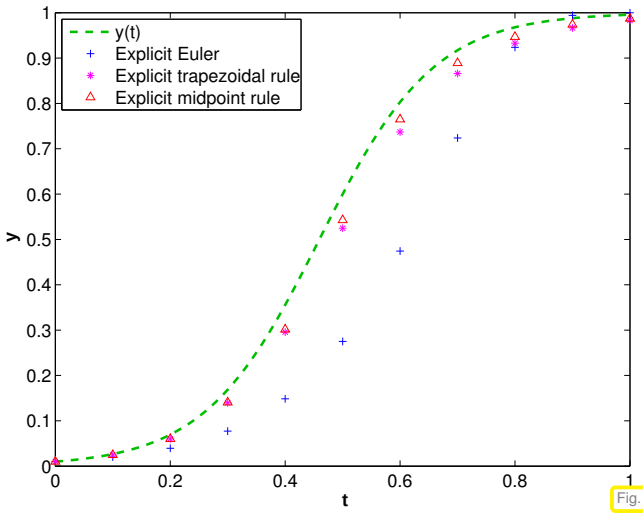


Fig. 412

$y_h(j/10)$, $j = 1, \ldots, 10$ for explicit RK-methods

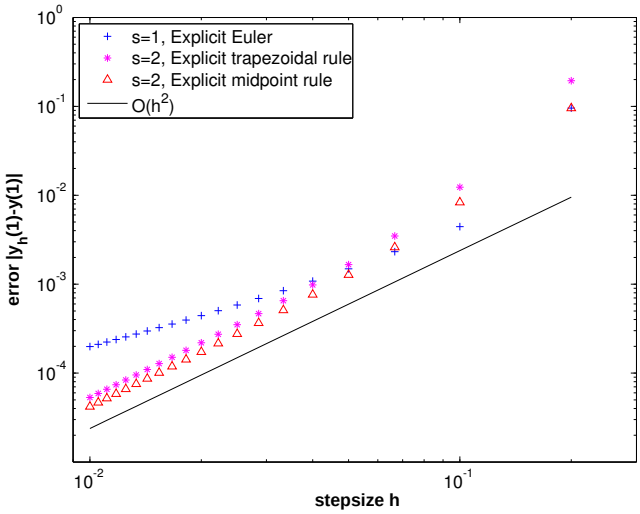


Fig. 413

Errors at final time $y_h(1) - y(1)$

Observation: obvious *algebraic convergence* in meshwidth h with integer rates/orders:

explicit trapezoidal rule (11.4.6) $\rightarrow$ order 2
explicit midpoint rule (11.4.7) $\rightarrow$ order 2

This is what one expects from the considerations in Ex. 11.4.4.

The formulas that we have obtained follow a general pattern:

Definition 11.4.9. Explicit Runge-Kutta method

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{i-1} a_{ij}$, $i, j = 1, \ldots, s$, $s \in \mathbb{N}$, an s-stage explicit Runge-Kutta single step method (RK-SSM) for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, $\mathbf{f} : \Omega \to \mathbb{R}^d$, is defined by ($\mathbf{y}_0 \in D$)

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j) \;, \quad i = 1, \ldots, s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h \sum_{i=1}^{s} b_i \mathbf{k}_i \;.$$

The vectors $\mathbf{k}_i \in \mathbb{R}^d$, $i = 1, \ldots, s$, are called increments, $h > 0$ is the size of the timestep.

Recall Rem. 11.3.12 to understand how the discrete evolution for an explicit Runge-Kutta method is specified in this definition by giving the formulas for the first step. This is a convention widely adopted in the literature about numerical methods for ODEs. Of course, the increments $\mathbf{k}_i$ have to be computed anew in each timestep.

The implementation of an s-stage explicit Runge-Kutta single step method according to Def. 11.4.9 is straightforward: The increments $\mathbf{k}_i \in \mathbb{R}^d$ are computed successively, starting from $\mathbf{k}_1 = \mathbf{f}(t_0 + c_1 h, \mathbf{y}_0)$.

➤ Only s $\mathbf{f}$-evaluations and AXPY operations ($\rightarrow$ Section 1.3.2) are required.

Butcher scheme notation for explicit RK-SSM

Shorthand notation for (explicit) Runge-Kutta methods [**?**, (11.75)]

Butcher scheme ▷

(Note: $\mathfrak{A}$ is strictly lower triangular $s \times s$-matrix)

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & 0 & \cdots & 0 \\ c_2 & a_{21} & \ddots & \vdots \\ \vdots & \vdots & & \ddots \\ c_s & a_{s1} & \cdots & a_{s,s-1} \quad 0 \\ \hline & b_1 & \cdots & b_{s-1} \quad b_s \end{array} . \tag{11.4.11}$$

Note that in Def. 11.4.9 the coefficients b_i can be regarded as weights of a quadrature formula on $[0,1]$: apply explicit Runge-Kutta single step method to "ODE" $\dot{y} = f(t)$. The quadrature rule with these weights and nodes c_j will have order ≥ 1, if the weights add up to 1!

Corollary 11.4.12. Consistent Runge-Kutta single step methods

A Runge-Kutta single step method according to Def. 11.4.9 is consistent ($\rightarrow$ Def. 11.3.10) with the ODE $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, if and only if

$$\sum_{i=1}^{s} b_i = 1 .$$

Example 11.4.13 (Butcher schemes for some explicit RK-SSM [?, Sect. 11.6.1])

The following explicit Runge-Kutta single step methods are often mentioned in literature.

- Explicit Euler method (11.2.7): $\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$ ➢ order = 1
- explicit trapezoidal rule (11.4.6): $\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$ ➢ order = 2
- explicit midpoint rule (11.4.7): $\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$ ➢ order = 2

- Classical 4th-order RK-SSM:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array} \quad \succ \quad \text{order = 4}$$

- Kutta's 3/8-rule:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array} \quad \succ \quad \text{order = 4}$$

Hosts of (explicit) Runge-Kutta methods can be found in the literature, see for example the Wikipedia page. They are stated in the form of Butcher schemes (11.4.11) most of the time.

Remark 11.4.14 (Construction of higher order Runge-Kutta single step methods)

Runge-Kutta single step methods of order $p > 2$ are not found by bootstrapping as in Ex. 11.4.4, because the resulting methods would have quite a lot of stages compared to their order.

Rather one derives order conditions yielding large non-linear systems of equations for the coefficients a_{ij} and b_i in Def. 11.4.9, see [**?**, Sect .4.2.3] and [**?**, Ch. III]. This approach is similar to the construction of a Gauss quadrature rule in Ex. 7.3.13. Unfortunately, the systems of equations are very difficult to solve and no universal recipe is available. Nevertheless, through massive use of symbolic computation, Runge-Kutta methods of order up to 19 have been constructed in this way.

Remark 11.4.15 ("Butcher barriers" for explicit RK-SSM)

The following table gives lower bounds for the number of stages needed to achieve order p for an explicit Runge-Kutta method.

order p	1	2	3	4	5	6	7	8	≥ 9
minimal no. s of stages	1	2	3	4	6	7	9	11	$\geq p+3$

No general formula is has been discovered. What is known is that for explicit Runge-Kutta single step methods according to Def. 11.4.9

$$\text{order } p \quad \leq \quad \text{number } s \text{ of stages of RK-SSM}$$

(11.4.16) EIGEN-compatible adaptive explicit embedded Runge-Kutta integrator

An implementation of an explicit embedded Runge-Kutta single-step method with adaptive stepsize control for solving an autonomous IVP is provided by the utility class **ode45**. The terms "embedded" and "adaptive" will be explained in Section 11.5.

The class is templated with two type parameters:

(i) **StateType**: type for vectors in state space V, e.g. a fixed size vector type of EIGEN: `Eigen::Matrix<double,N,1>`, where `N` is an integer constant § 11.2.1.

(ii) **RhsType**: a functor type, see Section 0.2.3, for the right hand side function $\mathbf{f}$; must match **StateType**, default type provided.

C++11 code 11.4.17: Runge-Kutta-Fehlberg 4(5) numerical integrator class

```
2  template <class StateType,
3            class RhsType = std::function<StateType(const StateType &)>>
4  class ode45 {
5  public:
6    // Idle constructor
7    ode45(const RhsType & rhs) : f(rhs) {}
8    // Main timestepping routine
9    template<class NormFunc = decltype(_norm<StateType>)>
10   std::vector< std::pair<StateType, double>>
11   solve(const StateType & y0,double T,const NormFunc & norm =
       _norm<StateType>);
12   // Print statistics and options of this class instance.
13   void print();
14   struct Options {
15     bool save_init = true;    // Set true if you want to save the
         initial data
16     bool fixed_stepsize = false;  // TODO: Set true if you want a fixed
         step size
17     unsigned int max_iterations    = 5000;
18     double min_dt = -1.; // Set the minimum step size (-1 for none)
19     double max_dt = -1.; // Set the maximum step size (-1 for none)
20     double initial_dt = -1.; // Set an initial step size
21     double start_time = 0; // Set a starting time
22     double rtol = 1e-6; // Relative tolerance for the error.
23     double atol = 1e-8; // Absolute tolerance for the error.
24     bool do_statistics = false; // Set to true before solving to save
         statistics
25     bool  verbose = false;  // Print more output.
26   } options;
27   // Data structure for usage statistics.
28   // Available after a call of solve(), if do_statistics is set to
       true.
29   struct Statistics {
30     unsigned int cycles = 0; // Number of loops (sum of all accepted
         and rejected steps)
31     unsigned int steps = 0;  // Number of actual time steps performed
         (accepted step)
32     unsigned int rejected_steps = 0; // Number of rejected steps per
         step
33     unsigned int funcalls = 0;     // Function calls
34   } statistics;
35 };
```

The functor for the right hand side $\mathbf{f} : D \subset V \to V$ of the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ is specified as an argument of the constructor.

The single-step numerical integrator is invoked by the templated method

```
template<class NormFunc = decltype(_norm<StateType>)>
  std::vector< std::pair<StateType, double>>
  solve(const StateType & y0,double T,const NormFunc & norm =
     _norm<StateType>);
```

The following arguments have to be supplied:

1. `y0`: the initial value $\mathbf{y}_0$
2. `T`: the final time T, initial time $t_0 = 0$ is assumed, because the class can deal with autonomous ODEs only, recall § 11.1.21.
3. `norm`: a functor returning a suitable norm for a state vector. Defaults to EIGEN's maximum vector norm.

The method returns a vector of 2-tuples $(\mathbf{y}_k, t_k)$ (note the order!), $k = 0, \ldots, N$, of temporal mesh points t_k, $t_0 = 0$, $t_N = T$, see § 11.2.2, and approximate states $\mathbf{y}_k \approx \mathbf{y}(t_k)$, where $t \mapsto \mathbf{y}(t)$ stands for the exact solution of the initial value problem.

The next self-explanatory code snippet uses the numerical integrator class **ode45** for solving a scalar autonomous ODE.

C++11 code 11.4.18: Invocation of adaptive embedded Runge-Kutta-Fehlberg integrator

```
int main(void)
{
  // Types to be used for a scalar ODE with state space ℝ
  using StateType = double;
  using RhsType = std::function<StateType(StateType)>;
  // Logistic differential equation (11.1.6)
  RhsType f = [](StateType y) { return 5*y*(1-y); };
  StateType y0 = 0.2;   // Initial value
  // Exact solution of IVP, see (11.1.7)
  auto y = [y0](double t) { return y0/(y0+(1-y0)*exp(-5*t)); };
  // State space ℝ, simple modulus supplies norm
  auto normFunc = [](StateType x){ return fabs(x); };

  // Invoke explicit Runge-Kutta method with stepsize control
  ode45<StateType,RhsType> integrator(f);
  std::vector<std::pair<StateType, double>> states =
     integrator.solve(y0,1.0,normFunc);
  // Output information accumulation during numerical integration
  integrator.options.do_statistics = true; integrator.print();

  for (auto state : states)
    std::cout << "t = " << state.second << ", y = " << state.first
          << ", |err| = " << fabs(state.first - y(state.second)) <<
             std::endl;
}
```

Remark 11.4.19 (Explicit ODE integrator in MATLAB)

MATLAB provides a built-in numerical integrator based on explicit RK-SSM, see [**?**] and [**?**, Sect. 7.2]. Its calling syntax is

```
[t,y] = ode45(odefun,tspan,y0);
```

`odefun`	:	Handle to a function of type `@(t,y)` $\leftrightarrow$ r.h.s. $\mathbf{f}(t,\mathbf{y})$
`tspan`	:	vector $[t_0, T[^\top$, initial and final time for numerical integration
`y0`	:	(vector) passing initial state $\mathbf{y}_0 \in \mathbb{R}^d$

Return values:

`t`	:	temporal mesh $\{t_0 < t_1 < t_2 < \cdots < t_{N-1} = t_N = T\}$
`y`	:	sequence $(\mathbf{y}_k)_{k=0}^N$ (column vectors)

MATLAB-code 11.4.20: Code excerpts from MATLAB's integrator `ode45`

```
function varargout = ode45(ode,tspan,y0,options,varargin)
% Processing of input parameters omitted
% :
% Initialize method parameters, c.f.  Butcher scheme (11.4.11)
pow = 1/5;
A = [1/5, 3/10, 4/5, 8/9, 1, 1];
B = [
    1/5          3/40     44/45    19372/6561      9017/3168        35/384
    0            9/40     -56/15   -25360/2187     -355/33          0
    0            0        32/9     64448/6561      46732/5247
       500/1113
    0            0        0        -212/729        49/176           125/192
    0            0        0        0               -5103/18656
       -2187/6784
    0            0        0        0               0                11/84
    0            0        0        0               0                0
    ];
E = [71/57600; 0; -71/16695; 71/1920; -17253/339200; 22/525; -1/40];
% : (choice of stepsize and main loop omitted)
% ADVANCING ONE STEP.
hA = h * A;
hB = h * B;
f(:,2) = feval(odeFcn,t+hA(1),y+f*hB(:,1),odeArgs{:});
f(:,3) = feval(odeFcn,t+hA(2),y+f*hB(:,2),odeArgs{:});
f(:,4) = feval(odeFcn,t+hA(3),y+f*hB(:,3),odeArgs{:});
f(:,5) = feval(odeFcn,t+hA(4),y+f*hB(:,4),odeArgs{:});
f(:,6) = feval(odeFcn,t+hA(5),y+f*hB(:,5),odeArgs{:});

tnew = t + hA(6);
if done, tnew = tfinal; end   % Hit end point exactly.
h = tnew - t;       % Purify h.
ynew = y + f*hB(:,6);
% : (stepsize control, see Sect. 11.5 dropped
```

Example 11.4.21 (Numerical integration of logistic ODE in MATLAB)

This example demonstrates the use of `ode45` for a scalar ODE ($d = 1$)

MATLAB-CODE: usage of ode45

```
fn = @(t,y) 5*y*(1-y);
[t,y] = ode45(fn,[0 1.5],y0);
plot(t,y,'r-');
```

MATLAB-integrator: `ode45()`:

Handle passing r.h.s. function $\mathbf{f} = \mathbf{f}(t,y)$,
initial and final time as row vector,
initial state $\mathbf{y}_0$, as column vector,

11.5 Adaptive Stepsize Control

Supplementary reading. [**?**, Sect. 11.7], [**?**, Sect. 11.8.2]

Example 11.5.1 (Oregonator reaction)

Chemical reaction kinetics is a field where ODE based models are very common. This example presents a famous reaction with extremely abrupt dynamics. Refer to [**?**, Ch. 62] for more information about the ODE-based modelling of kinetics of chemical reactions.

This is a apecial case of an "oscillating" Zhabotinski-Belousov reaction [**?**]:

$$\begin{aligned} \mathrm{BrO}_3^- + \mathrm{Br}^- &\mapsto \mathrm{HBrO}_2 \\ \mathrm{HBrO}_2 + \mathrm{Br}^- &\mapsto \mathrm{Org} \\ \mathrm{BrO}_3^- + \mathrm{HBrO}_2 &\mapsto 2\,\mathrm{HBrO}_2 + \mathrm{Ce(IV)} \\ 2\,\mathrm{HBrO}_2 &\mapsto \mathrm{Org} \\ \mathrm{Ce(IV)} &\mapsto \mathrm{Br}^- \end{aligned} \tag{11.5.2}$$

$$\begin{aligned} y_1 := c(\mathrm{BrO}_3^-): \quad \dot{y}_1 &= -k_1 y_1 y_2 - k_3 y_1 y_3 \,, \\ y_2 := c(\mathrm{Br}^-): \quad \dot{y}_2 &= -k_1 y_1 y_2 - k_2 y_2 y_3 + k_5 y_5 \,, \\ y_3 := c(\mathrm{HBrO}_2): \quad \dot{y}_3 &= k_1 y_1 y_2 - k_2 y_2 y_3 + k_3 y_1 y_3 - 2k_4 y_3^2 \,, \\ y_4 := c(\mathrm{Org}): \quad \dot{y}_4 &= k_2 y_2 y_3 + k_4 y_3^2 \,, \\ y_5 := c(\mathrm{Ce(IV)}): \quad \dot{y}_5 &= k_3 y_1 y_3 - k_5 y_5 \,, \end{aligned} \tag{11.5.3}$$

with (non-dimensionalized) reaction constants:

$$k_1 = 1.34\,, \quad k_2 = 1.6 \cdot 10^9\,, \quad k_3 = 8.0 \cdot 10^3\,, \quad k_4 = 4.0 \cdot 10^7\,, \quad k_5 = 1.0\,.$$

periodic chemical reaction ➽ Video 1, Video 2

MATLAB simulation with initial state $y_1(0) = 0.06$, $y_2(0) = 0.33 \cdot 10^{-6}$, $y_3(0) = 0.501 \cdot 10^{-10}$, $y_4(0) = 0.03$, $y_5(0) = 0.24 \cdot 10^{-7}$:

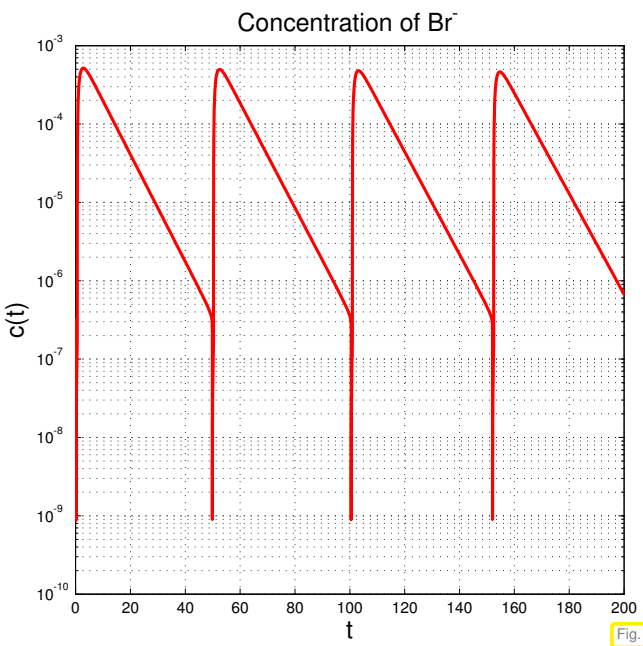


Fig. 414

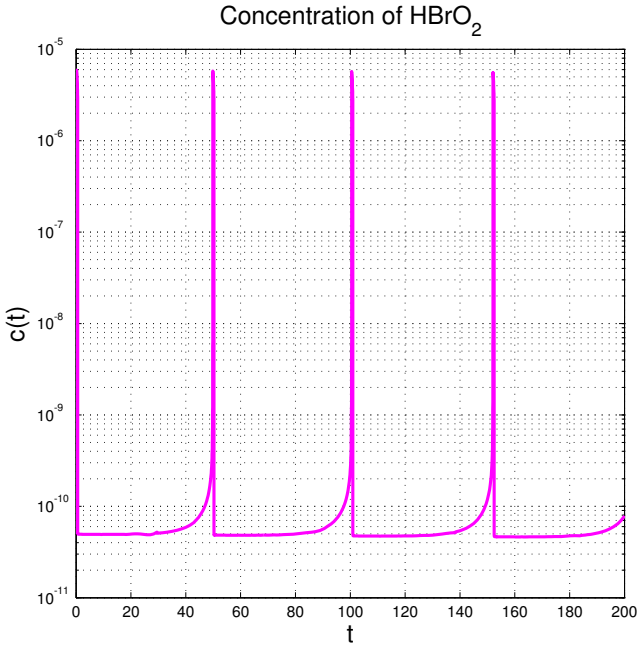


Fig. 415

We observe a strongly non-uniform behavior of the solution in time.

This is very common with evolutions arising from practical models (circuit models, chemical reaction models, mechanical systems)

Example 11.5.4 (Blow-up)

We return to the "explosion ODE" of Ex. 11.1.35 and consider the scalar autonomous IVP:

$$\dot{y} = y^2 \,, \quad y(0) = y_0 > 0 \,.$$

$$\blacktriangleright \quad y(t) = \frac{y_0}{1 - y_0 t} \,, \quad t < 1/y_0 \,.$$

As we have seen a solution exists only for finite time and then suffers a Blow-up, that is, $\lim_{t \to 1/y_0} y(t) = \infty$: $J(y_0) =]-\infty, 1/y_0]$!

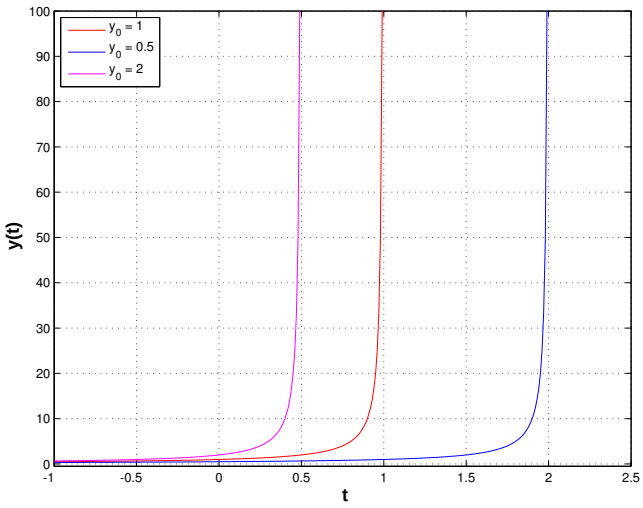


Fig. 416

How to choose temporal mesh $\{t_0 < t_1 < \cdots < t_{N-1} < t_N\}$ for single step method in case $J(y_0)$ is not known, even worse, if it is not clear a priori that a blow up will happen?

Just imagine: what will result from equidistant explicit Euler integration (11.2.7) applied to the above IVP?

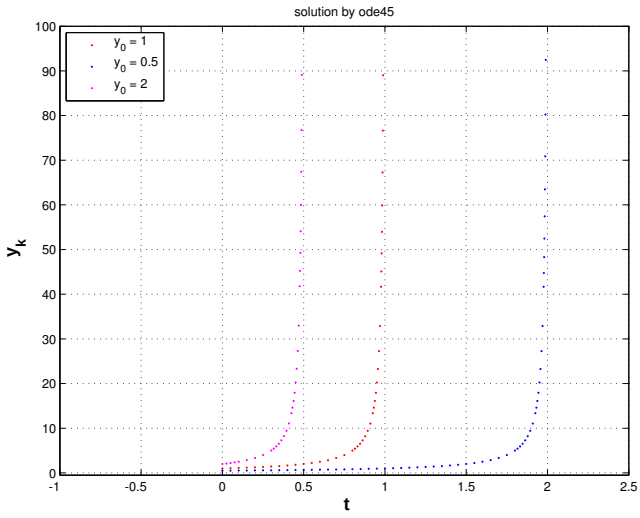


Fig. 417

Simulation with MATLAB's `ode45`:

MATLAB-code 11.5.5:

```
fun = @(t,y) y.^2;
t1,y1] = ode45(fun,[0 2],1);
[t2,y2] = ode45(fun,[0
   2],0.5);
[t3,y3] = ode45(fun,[0 2],2);
```

MATLAB warning messages:

```
  Warning: Failure at t=9.999694e-01.  Unable to meet integration
  tolerances without reducing the step size below the smallest
  value allowed (1.776357e-15) at time t.
> In ode45 at 371
  In simpleblowup at 22
Warning: Failure at t=1.999970e+00.  Unable to meet integration
  tolerances without reducing the step size below the smallest
  value allowed (3.552714e-15) at time t.
> In ode45 at 371
  In simpleblowup at 23
Warning: Failure at t=4.999660e-01.  Unable to meet integration
  tolerances without reducing the step size below the smallest
  value allowed (8.881784e-16) at time t.
> In ode45 at 371
  In simpleblowup at 24
```

We observe: `ode45` manages to reduce stepsize more and more as it approaches the singularity of the solution! How can it accomplish this feat!

Key challenge (discussed for autonomous ODEs below):

How to choose a *good temporal mesh* $\{0 = t_0 < t_1 < \cdots < t_{N-1} < t_N\}$
for a given single step method applied to a concerete IVP?

What does "good" mean ?

Be efficient! Be accurate!

Stepsize adaptation for single step methods

Objective: N as small as possible & $\max_{k=1,\ldots,N} \|\mathbf{y}(t_k) - \mathbf{y}_k\| < \mathtt{TOL}$ or $\|\mathbf{y}(T) - \mathbf{y}_N\| < \mathtt{TOL}$, $\mathtt{TOL}$ = tolerance

Policy: Try to curb/balance one-step error by

- ✦ adjusting *current* stepsize h_k,
- ✦ predicting suitable *next* timestep h_{k+1}

} local-in-time stepsize control

Tool: Local-in-time one-step error estimator (*a posteriori*, based on $\mathbf{y}_k, h_{k-1}$)

Why local-in-time timestep control (based on estimating only the one-step error)**?**

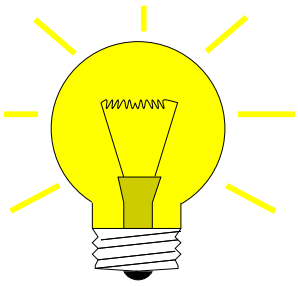
Consideration: If a small time-local error in a single timestep leads to large error $\|\mathbf{y}_k - \mathbf{y}(t_k)\|$ at later times, then local-in-time timestep control is powerless about it and will not even notice!!

Nevertheless, local-in-time timestep control is used almost exclusively,

- ☞ because we do not want to discard past timesteps, which could amount to tremendous waste of computational resources,
- ☞ because it is inexpensive and it works for many practical problems,
- ☞ because there is no reliable method that can deliver guaranteed accuracy for general IVP.

(11.5.7) Local-in-time error estimation

We "recycle" heuristics already employed for adaptive quadrature, see Section 7.5, § 7.5.10. There we tried to get an idea of the local quadrature error by comparing two approximations of different order. Now we pursue a similar idea over a single timestep.



Idea: *Estimation of one-step error*

Compare results for two discrete evolutions $\boldsymbol{\Psi}^h$, $\widetilde{\boldsymbol{\Psi}}^h$ of different order over *current timestep* h:

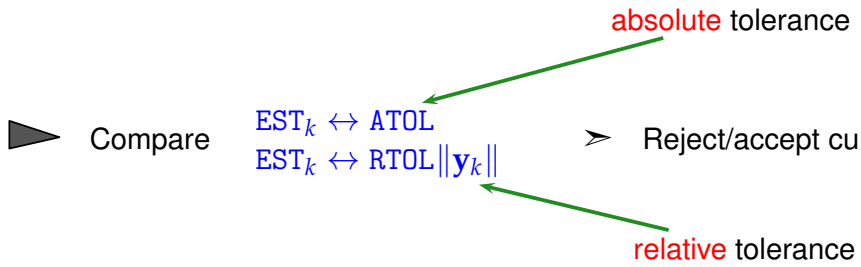
If $\mathrm{Order}(\widetilde{\boldsymbol{\Psi}}) > \mathrm{Order}(\boldsymbol{\Psi})$, then we expect

$$\underbrace{\boldsymbol{\Phi}^h\mathbf{y}(t_k) - \boldsymbol{\Psi}^h\mathbf{y}(t_k)}_{\text{one-step error}} \approx \mathrm{EST}_k := \widetilde{\boldsymbol{\Psi}}^h\mathbf{y}(t_k) - \boldsymbol{\Psi}^h\mathbf{y}(t_k)\,. \tag{11.5.8}$$

Heuristics for concrete h

(11.5.9) Temporal mesh refinement

We take for granted a local error estimate EST_k.



(11.5.10)

For a similar use of absolute and relative tolerances see Section 8.1.2: termination criteria for iterations, in particular (8.1.25).

☞ Simple algorithm:

$\mathrm{EST}_k < \max\{\mathrm{ATOL}, \|\mathbf{y}_k\|\mathrm{RTOL}\}$: Carry out next timestep (stepsize h)
Use larger stepsize (e.g., αh with some $\alpha > 1$) for following step $(*)$

$\mathrm{EST}_k > \max\{\mathrm{ATOL}, \|\mathbf{y}_k\|\mathrm{RTOL}\}$: *Repeat* current step with smaller stepsize $< h$, e.g., $\frac{1}{2}h$

Rationale for $(*)$: if the current stepsize guarantees sufficiently small one-step error, then it might be possible to obtain a still acceptable one-step error with a larger timestep, which would enhance efficiency (fewer timesteps for total numerical integration). This should be tried, since timestep control will usually provide a safeguard against undue loss of accuracy.

C++11 code 11.5.11: Simple local stepsize control for single step methods

```
// Auxiliary function:  default norm for an EIGEN vector type
template <class State>
double _norm(const State &y) { return y.norm(); }

template <class DiscEvolOp, class State, class NormFunc =
   decltype(_norm<State>)>
std::vector<std::pair<double, State> >
odeintadapt(DiscEvolOp &Psilow, DiscEvolOp &Psihigh,
            State& y0,double T, double h0,
            double reltol, double abstol, double hmin,
            NormFunc &norm = _norm<State>) {
  double t = 0; // initial time
  State y = y0; // initial state
  double h = h0; // timestep to start with
  std::vector<std::pair<double,State>> states; // for output
  states.push_back({t, y});

  while ((states.back().first < T) && (h >= hmin))  { //
    State yh = Psihigh(h, y0); // high order discrete evolution Ψ̃^h
    State yH = Psilow(h, y0); // low order discrete evolution Ψ^h
    double est = norm(yH-yh); // local error estimate EST_k

    if (est < max(reltol*norm(y0), abstol)) { // step accepted
      y0 = yh; // use high order approximation
      t = t+min(T-t,h); // next time t_k
      states.push_back({t,y0}); //
```

```
27        h = 1.1*h;  // try with increased stepsize
28      }
29      else // step rejected
30        h = h/2; // try with half the stepsize
31    }
32    // Numerical integration has ground to a halt !
33    if (h < hmin)  {
34      cerr << "Warning: Failure at t=" << states.back().first
35        << ". Unable to meet integration tolerances without reducing
             the step "
36        << "size below the smallest value allowed ("<< hmin <<") at
             time t." << endl;
37    }
38    return states;
39  }
```

Comments on Code 11.5.11:

- Input arguments:
  - `Psilow`, `Psihigh`: functors passing discrete evolution operators for autonomous ODE of different order, type `@(y,h)`, expecting a state (usually a column vector) as first argument, and a stepsize as second,
  - `T`: final time $T > 0$,
  - `y0`: initial state $\mathbf{y}_0$,
  - `h0`: stepsize h_0 for the first timestep
  - `reltol`, `abstol`: relative and absolute tolerances, see (11.5.10),
  - `hmin`: minimal stepsize, timestepping terminates when stepsize control $h_k < h_{\min}$, which is relevant for detecting blow-ups or collapse of the solution.
- line 18: check whether final time is reached or timestepping has ground to a halt ($h_k < h_{\min}$).
- line 19, 20: advance state by low and high order integrator.
- line 21: compute norm of estimated error, see (11.5.8).
- line 23: make comparison (11.5.10) to decide whether to accept or reject local step.
- line 26, 27: step accepted, update state and current time and suggest 1.1 times the current stepsize for next step.
- line 30 step rejected, try again with half the stepsize.
- Return value is a vector of pairs consisting of
  - times `t` $\leftrightarrow$ temporal mesh $t_0 < t_1 < t_2 < \ldots < t_N < T$, where $t_N < T$ indicated premature termination (collapse, blow-up),
  - states `y` $\leftrightarrow$ sequence $(\mathbf{y}_k)_{k=0}^{N}$.

Remark 11.5.12 (Estimation of "wrong" error?)

We face the same conundrum as in the case of adaptive numerical quadrature, see Rem. 7.5.17:

! By the heuristic considerations, see (11.5.8) it seems that EST_k measures the one-step error for the low-order method Ψ and that we should use $\mathbf{y}_{k+1} = \Psi^{h_k}\mathbf{y}_k$, if the timestep is accepted.

However, it would be foolish not to use the better value $\mathbf{y}_{k+1} = \widetilde{\Psi}^{h_k}\mathbf{y}_k$, since it is available for free. This is what is done in every implementation of adaptive methods, also in Code 11.5.11, and this choice can be justified by control theoretic arguments [**?**, Sect. 5.2].

Example 11.5.13 (Simple adaptive stepsize control)

We test adaptive timestepping routine from Code 11.5.11 for a scalar IVP and compare the estimated local error and true local error.

- IVP for ODE $\dot{y} = \cos(\alpha y)^2$, $\alpha > 0$, solution $y(t) = \arctan(\alpha(t-c))/\alpha$ for $y(0) \in]-\pi/2, \pi/2[$
- Simple adaptive timestepping based on explicit Euler (11.2.7) and explicit trapezoidal rule (11.4.6)

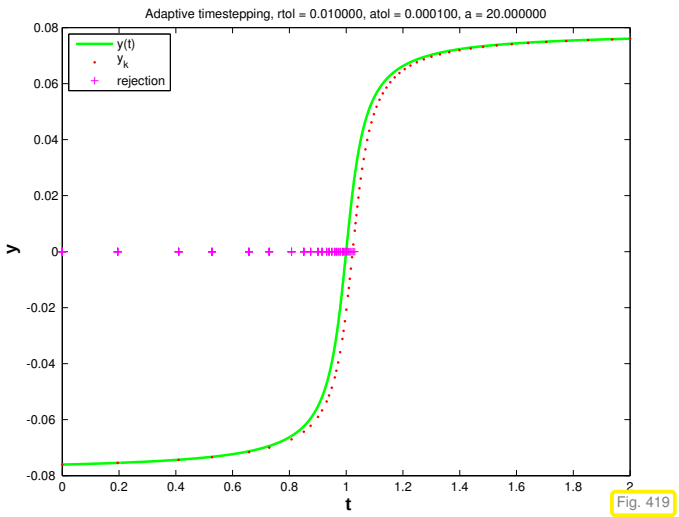


Fig. 418

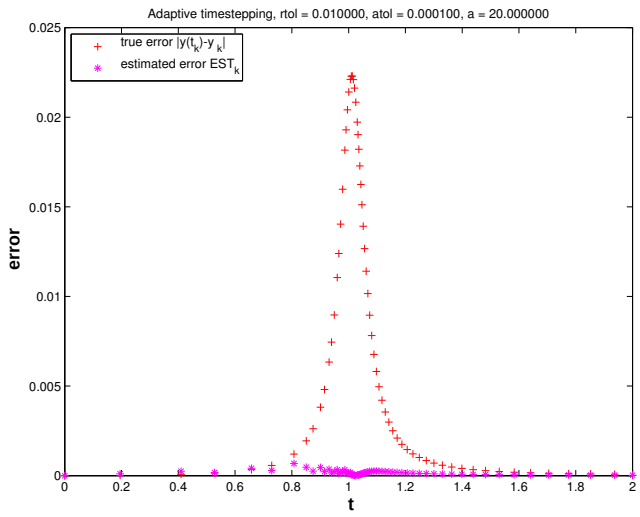


Fig. 419

Statistics: `66 timesteps, 131 rejected timesteps`

Observations:

- ☞ Adaptive timestepping well resolves local features of solution $y(t)$ at $t = 1$
- ☞ Estimated error (an estimate for the one-step error) and true error are **not** related! To understand this recall Rem. 11.5.12.

Example 11.5.14 (Gain through adaptivity → Ex. 11.5.13)

In this experiment we want to explore whether adaptive timestepping is worth while, as regards reduction of computational effort without sacrificing accuracy.

We retain the simple adaptive timestepping from previous experiment Ex. 11.5.13 and also study the same IVP.

New: initial state $y(0) = 0$!

Now we examine the dependence of the maximal discretization error in mesh points on the computational effort. The latter is proportional to the number of timesteps.

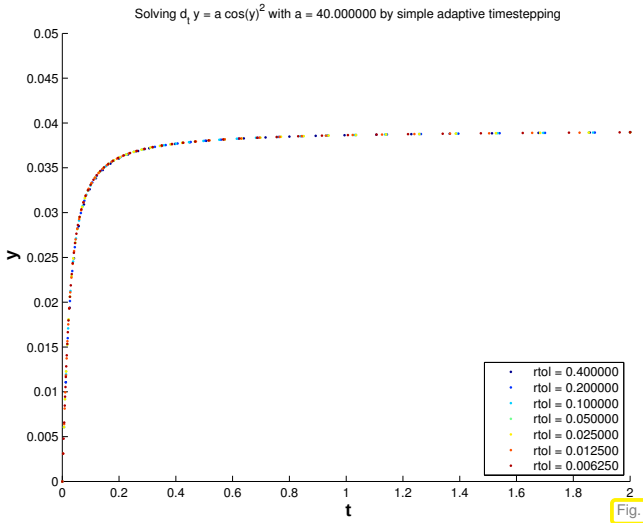


Fig. 420

Solutions $(\mathbf{y}_k)_k$ for different values of `rtol`

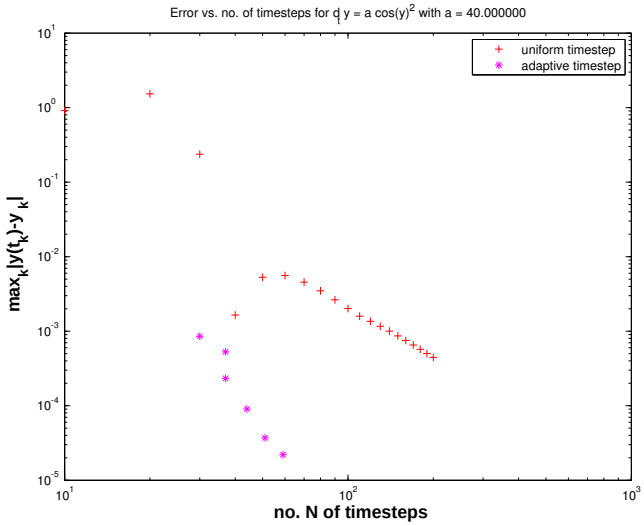


Fig. 421

Error vs. computational effort

Observations:

☞ Adaptive timestepping achieves much better accuracy for a fixed computational effort.

Example 11.5.15 ("Failure" of adaptive timestepping → Ex. 11.5.14)

Same ODE and simple adaptive timestepping as in previous experiment Ex. 11.5.14.

$$\dot{y} = \cos^2(\alpha y) \quad \Rightarrow \quad y(t) = \arctan(\alpha(t-c))/\alpha \text{ ,} y(0) \in]-\frac{\pi}{2\alpha}, -\frac{\pi}{2\alpha}[\text{ ,}$$

for $\alpha = 40$.

Now: initial state $y(0) = -0.0386 \approx \frac{\pi}{2\alpha}$ as in Ex. 11.5.13

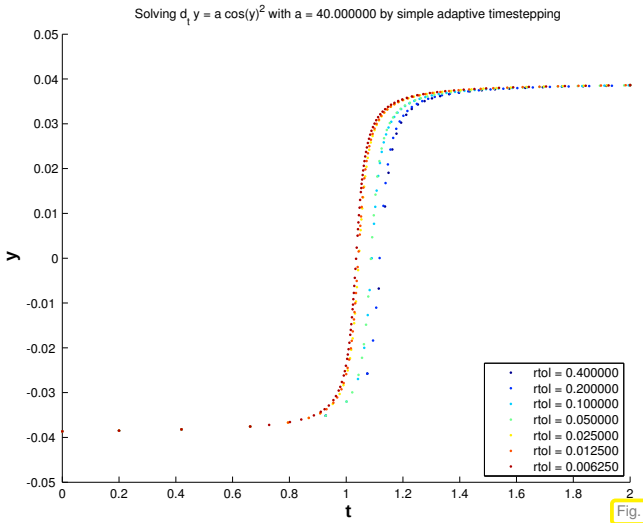


Fig. 422

Solutions $(\mathbf{y}_k)_k$ for different values of `rtol`

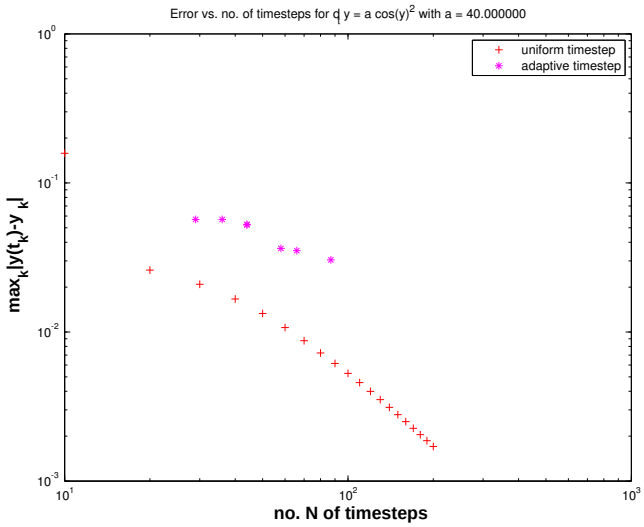


Fig. 423

Error vs. computational effort

Observations:

☞ Adaptive timestepping leads to larger errors at the same computational cost as uniform timestepping **!**

Explanation: the position of the steep step of the solution has a sensitive dependence on an initial value, if $y(0) \approx \frac{\pi}{2alpha}$:

$$y(t) = \frac{1}{\alpha}\arctan(\alpha(t + \tan(y_0/\alpha))) \ , \quad \text{step at} \quad \approx -\tan(y_0/\alpha) \ .$$

Hence, small local errors in the initial timesteps will lead to large errors at around time $t \approx 1$. The stepsize control is mistaken in condoning these small one-step errors in the first few steps and, therefore, incurs huge errors later.

However, the perspective of backward error analysis (→ § 1.5.84) rehabilitates adaptive stepsize control in this case: it gives us a numerical solution that is very close to the exact solution of the ODE with slightly perturbed initial state y_0.

Remark 11.5.16 (Refined local stepsize control → [?, Sect. 11.7])

The above algorithm (Code 11.5.11) is simple, but the rule for increasing/shrinking of timestep "squanders" the information contained in EST_k : TOL:

More ambitious goal ! When $\mathrm{EST}_k > \mathrm{TOL}$: stepsize adjustment better $h_k = ?$
When $\mathrm{EST}_k < \mathrm{TOL}$: stepsize prediction good $h_{k+1} = ?$

Assumption: At our disposal are two discrete evolutions:

- $\boldsymbol{\Psi}$ with order$(\boldsymbol{\Psi}) = p$ (➙ "low order" single step method)
- $\widetilde{\boldsymbol{\Psi}}$ with order$(\widetilde{\boldsymbol{\Psi}}) > p$ (➙ "higher order" single step method)

These are the same building blocks as for the simple adaptive strategy employed in Code 11.5.11 (passed as arguments `Psilow`, `Psihigh` there).

Asymptotic expressions for one-step error for $h \to 0$:

$$\begin{aligned} \boldsymbol{\Psi}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k) &= ch^{p+1} + O(h_k^{p+2}) \ , \\ \widetilde{\boldsymbol{\Psi}}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k) &= O(h_k^{p+2}) \ , \end{aligned} \tag{11.5.17}$$

with some (unknown) $c > 0$.

Why h^{p+1}? Remember estimate (11.3.28) from the error analysis of the explicit Euler method: we also found $O(h_k^2)$ there for the one-step error of a single step method of order 1.

Heuristics: the timestep h_k is small ➥ "higher order terms" $O(h_k^{p+2})$ can be ignored.

$$\blacktriangleright \quad \begin{aligned} \boldsymbol{\Psi}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k) &\doteq ch_k^{p+1} + O(h_k^{p+2}) \ , \\ \widetilde{\boldsymbol{\Psi}}^{h_k}\mathbf{y}(t_k) - \boldsymbol{\Phi}^{h_k}\mathbf{y}(t_k) &\doteq O(h_k^{p+2}) \ . \end{aligned} \quad \Rightarrow \quad \boxed{\mathrm{EST}_k \doteq ch_k^{p+1}} \ . \tag{11.5.18}$$

✎ notation: $\doteq$ equality up to higher order terms in h_k

$$\mathrm{EST}_k \doteq ch_k^{p+1} \quad \Rightarrow \quad c \doteq \frac{\mathrm{EST}_k}{h_k^{p+1}} \ . \tag{11.5.19}$$

Available in algorithm, see (11.5.8)

For the sake of *accuracy* (stipulates "$\mathrm{EST}_k < \mathrm{TOL}$") & *efficiency* (favors "$>$") we aim for

$$\mathrm{EST}_k \overset{!}{=} \mathrm{TOL} := \max\{\mathrm{ATOL}, \|\mathbf{y}_k\|\mathrm{RTOL}\} . \tag{11.5.20}$$

What timestep h_* can actually achieve (11.5.20), if we "believe" in (11.5.18) (and, therefore, in (11.5.19))?

$$\text{(11.5.19) \& (11.5.20)} \quad \Rightarrow \quad \mathrm{TOL} = \frac{\mathrm{EST}_k}{h_k^{p+1}} h_*^{p+1} .$$

➤ "'Optimal timestep": (stepsize prediction)

$$\boxed{h_* = h \sqrt[p+1]{\frac{\mathrm{TOL}}{\mathrm{EST}_k}}} . \tag{11.5.21}$$

adjusted stepsize (**R**) & suggested stepsize (**A**)

(**R**eject): In case $\mathrm{EST}_k > \mathrm{TOL}$ ➢ *repeat step* with stepsize h_*.

(**A**ccept): If $\mathrm{EST}_k \leq \mathrm{TOL}$ ➢ use h_* as stepsize for *next step*.

C++11 code 11.5.22: Refined local stepsize control for single step methods

```
// Auxiliary function: default norm for an EIGEN vector type
template <class State>
double _norm(const State &y) { return y.norm(); }

template <class DiscEvolOp, class State, class NormFunc =
   decltype(_norm<State>)>
std::vector<std::pair<double, State> >
odeintssctrl(DiscEvolOp &Psilow, unsigned int p, DiscEvolOp &Psihigh,
             State& y0,double T, double h0,
             double reltol, double abstol, double hmin,
             NormFunc &norm = _norm<State>) {
  double t = 0; // initial time
  State y = y0; // initial state
  double h = h0; // timestep to start with
  std::vector<std::pair<double,State>> states; // for output
  states.push_back({t, y});

  // Main timestepping loop
  while ((states.back().first < T) && (h >= hmin)) { //
    State yh = Psihigh(h, y0); // high order discrete evolution Ψ̃^h
    State yH = Psilow(h, y0); // low order discrete evolution Ψ^h
    double est = norm(yH-yh); // ↔ EST_k
    double tol = max(reltol*norm(y0), abstol); // effective tolerance

    // Optimal stepsize according to (11.5.21)
    h = h*max(0.5,min(2.,pow(tol/est,1./(p+1)))); //
    if (est < tol)  // step accepted
      states.push_back({t = t+min(T-t,h),y0 = yh}); // store next
         approximate state
```

```
29    }
30    if (h < hmin) {
31      cerr << "Warning: Failure at t="
32        << states.back().first
33        << ". Unable to meet integration tolerances without reducing
              the step"
34        << " size below the smallest value allowed ("<< hmin <<") at
              time t." << endl;
35    }
36    return states;
37  }
```

Comments on Code 11.5.22 (see comments on Code 11.5.11 for more explanations):

- Input arguments as for Code 11.5.11, except for `p` $\hat{=}$ order of lower order discrete evolution.
- line 26: compute presumably better local stepsize according to (11.5.21),
- line 27: decide whether to repeat the step or advance,
- line 27: extend output arrays if current step has not been rejected.

Remark 11.5.23 (Stepsize control in MATLAB)

The name of MATLAB's standard integrator `ode45` already indicates the orders of the pair of single step methods used for adaptive stepsize control:

Specifying tolerances for MATLAB's integrators is done as follows:

```
options = odeset('abstol',atol,'reltol',rtol,'stats','on');
[t,y] = ode45(@(t,x) f(t,x),tspan,y0,options);
```

(`f` = function handle, `tspan` $\hat{=} [t_0, T]$, `y0` $\hat{=} \mathbf{y}_0$, `t` $\hat{=} t_k$, `y` $\hat{=} \mathbf{y}_k$)

The possibility to pass tolerances to numerical integrators based on adaptive timestepping may tempt one into believing that they allow to control the accuracy of the solutions. However, as is clear from Rem. 11.5.16, these tolerances are solely applied to local error estimates and, inherently, have nothing to do with global discretization errors, see Ex. 11.5.13.

No global error control through local-in-time adaptive timestepping

The absolute/relative tolerances imposed for local-in-time adaptive timestepping do *not* allow to predict accuracy of solution!

Remark 11.5.25 (Embedded Runge-Kutta methods)

For higher order RK-SSM with a considerable number of stages computing different sets of increments ($\rightarrow$ Def. 11.4.9) for two methods of different order just for the sake of local-in-time stepsize control would mean incommensurate effort.

Embedding idea: Use two RK-SSMs based on the same increments, that is, built with the same coefficients a_{ij}, but different weights b_i, see Def. 11.4.9 for the formulas, and *different orders* p and $p+1$.

Butcher scheme for embedded explicit Runge-Kutta methods $\triangleright$

(Lower order scheme has weights $\widehat{b}_i$.)

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \\ \hline & \widehat{\mathbf{b}}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \\ \hline & \widehat{b}_1 & \cdots & \widehat{b}_s \end{array} .$$

Example 11.5.26 (Commonly used embedded explicit Runge-Kutta methods)

The following two embedded RK-SSM, presented in the form of their extended Butcher schemes, provided single step methods of orders 4 & 5.

$$\begin{array}{c|ccccc} 0 & & & & & \\ \frac{1}{3} & \frac{1}{3} & & & & \\ \frac{1}{3} & \frac{1}{6} & \frac{1}{6} & & & \\ \frac{1}{2} & \frac{1}{8} & 0 & \frac{3}{8} & & \\ 1 & \frac{1}{2} & 0 & -\frac{3}{2} & 2 & \\ \hline y_1 & \frac{1}{6} & 0 & 0 & \frac{2}{3} & \frac{1}{6} \\ \hline \widehat{y}_1 & \frac{1}{10} & 0 & \frac{3}{10} & \frac{2}{5} & \frac{1}{5} \end{array}$$

Merson's embedded RK-SSM

$$\begin{array}{c|ccccc} 0 & & & & & \\ \frac{1}{2} & \frac{1}{2} & & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & & \\ 1 & 0 & 0 & 1 & & \\ \frac{3}{4} & \frac{5}{32} & \frac{7}{32} & \frac{13}{32} & -\frac{1}{32} & \\ \hline y_1 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} & \\ \hline \widehat{y}_1 & -\frac{1}{2} & \frac{7}{3} & \frac{7}{3} & \frac{13}{6} & -\frac{16}{3} \end{array}$$

Fehlberg's embedded RK-SSM

Example 11.5.27 (Adaptive timestepping for mechanical problem)

We test the effect of adaptive stepsize control in MATLAB for the equations of motion describing the planar movement of a point mass in a conservative force field $\mathbf{x} \in \mathbb{R}^2 \mapsto F(\mathbf{x}) \in \mathbb{R}^2$: Let $t \mapsto \mathbf{y}(t) \in \mathbb{R}^2$ be the trajectory of point mass (in the plane).

From Newton's law: $$\ddot{\mathbf{y}} = F(\mathbf{y}) := -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2} . \tag{11.5.28}$$

acceleration — force

As in Rem. 11.1.23 we can convert the second-order ODE (11.5.28) into an equivalent 1st-order ODE by introducing the velocity $\mathbf{v} := \dot{\mathbf{y}}$ as an extra solution component:

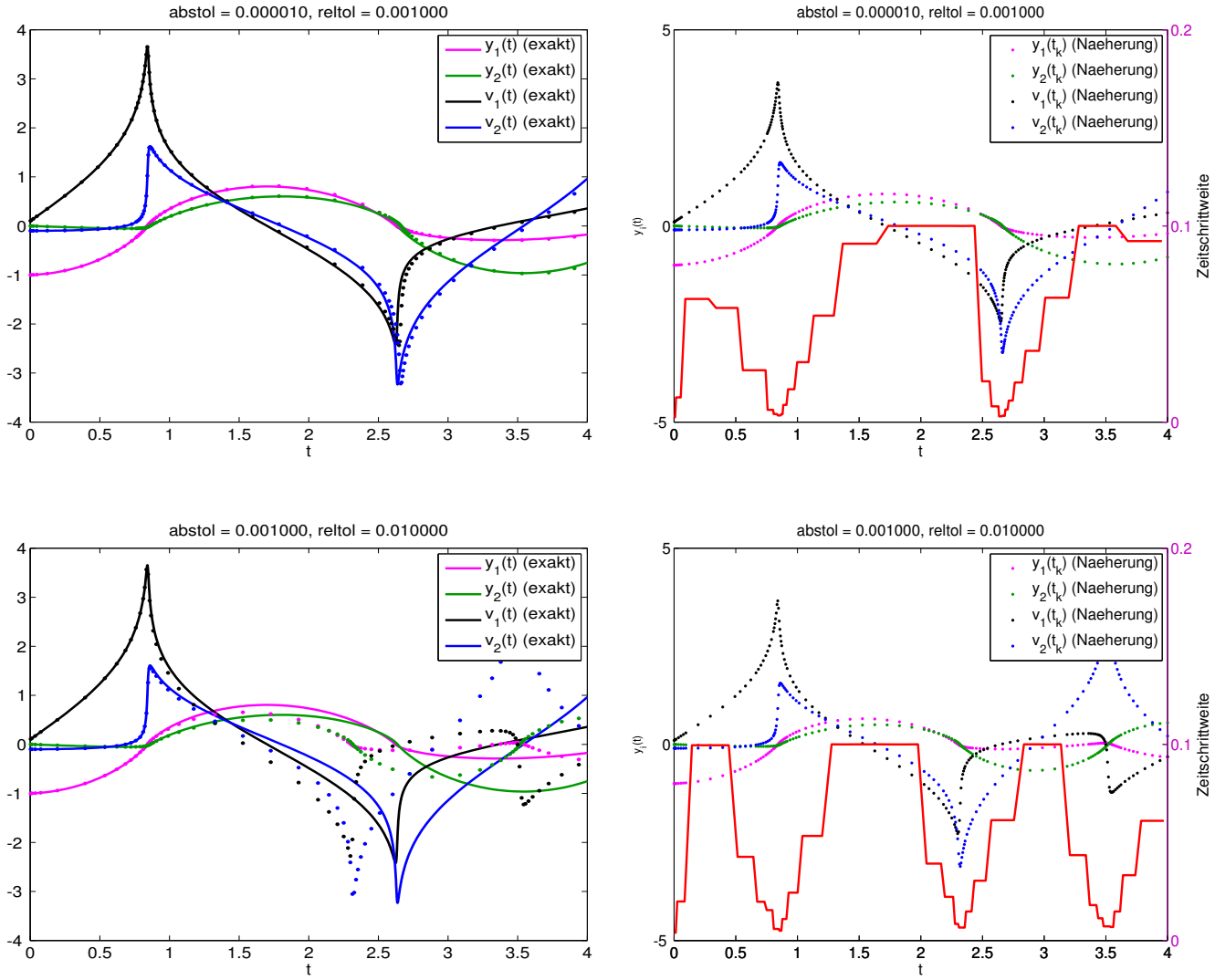
$$(11.5.28) \quad \Rightarrow \quad \begin{bmatrix} \dot{\mathbf{y}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2} \end{bmatrix} . \tag{11.5.29}$$

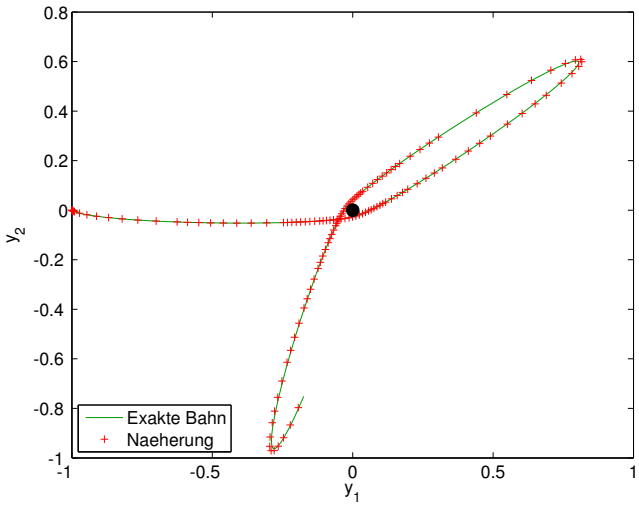
The following initial values used in the experiment:

$$\mathbf{y}(0) := \begin{bmatrix} -1 \\ 0 \end{bmatrix} \quad , \quad \mathbf{v}(0) := \begin{bmatrix} 0.1 \\ -0.1 \end{bmatrix}$$

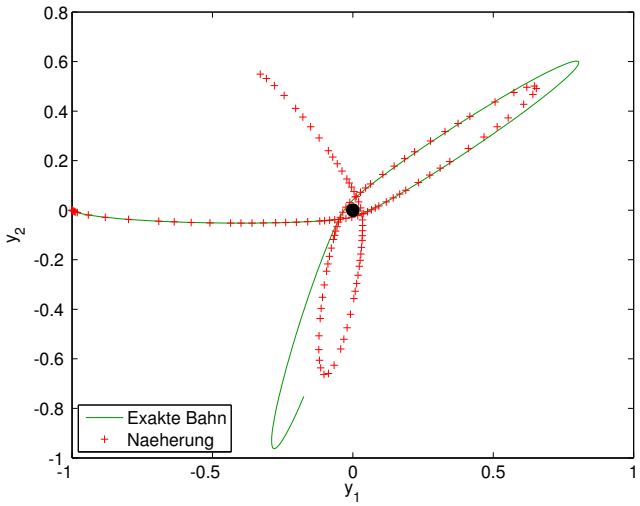
Adaptive numerical integration in MATLAB

```
ode45(@(t,x) f(t,x),[0 4],[-1;0;0.1;-0.1,],options):
❶ options = odeset('reltol',0.001,'abstol',1e-5);
❷ options = odeset('reltol',0.01,'abstol',1e-3);
```





reltol=0.001, abstol=1e-5



reltol=0.01, abstol=1e-3

Observations:

- ☞ Fast changes in solution components captured by adaptive approach through very small timesteps.
- ☞ Completely wrong solution, if tolerance reduced slightly.

In this example we face a rather sensitive dependence of the trajectories on initial states or intermediate states. Small perturbations at one instance in time can be have a massive impact on the solution at later times. Loca stepsize control is powerless about preventing this.

Summary and Learning Outcomes

After having studied this chapter you should

- know the concept of evolution operator for an ODE and its relationship with solutions of associated initial value problems.
- be able to convert higher-order and non-autonomous ODEs into the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.
- know about discrete evolutions and how they induce single step methods (SSMs).
- remember that single step method converge asymptotically algebraically for stepsize $h \to 0$ and that the rate of convergence is called the order of the SSM.
- know the general form of explicit Runge-Kutta methods and Butcher schemes.
- understand when adaptive timestep control is essential for meaningful numerical integration of initial value problems.
- be able to describe the policy to time-local adaptive timestep control for embedded Runge-Kutta methods.

Chapter 12

Single Step Methods for Stiff Initial Value Problems

Supplementary reading. [**?**, Sect. 11.9]

Explicit Runge-Kutta methods with stepsize control (→ Section 11.5) seem to be able to provide approximate solutions for any IVP with good accuracy provided that tolerances are set appropriately.

Everything settled about numerical integration**?**

Example 12.0.1 (`ode45` for stiff problem)

In this example we will witness the near failure of a high-order adaptive explicit Runge-Kutta method for a simple scalar autonomous ODE.

$$\text{IVP considered:} \quad \dot{y} = \lambda y^2(1-y)\,, \quad \lambda := 500 \quad , \quad y(0) = \tfrac{1}{100}\,. \tag{12.0.2}$$

This is a logistic ODE as introduced in Ex. 11.1.5. We try to solve it by means of an explicit adaptive embedded Runge-Kutta-Fehlberg method (→ Rem. 11.5.25) using the class **ode45** from § 11.4.16 (Preprocessor switch `MATLABCOEFF` activated).

C++11 code 12.0.3: Solving (12.0.2) **with class ode45**

```
2  // Types to be used for a scalar ODE with state space ℝ
3  using StateType = double;
4  using RhsType = std::function<StateType(StateType)>;
5  // Logistic differential equation (11.1.6)
6  double lambda = 500.0;
7  RhsType f = [lambda](StateType y) { return lambda*y*y*(1-y); };
8  StateType y0 = 0.01;   // Initial value, will create a STIFF IVP
9  // State space ℝ, simple modulus supplies norm
10 auto normFunc = [](StateType x){ return fabs(x); };
11
12 // Invoke explicit Runge-Kutta method with stepsize control
13 ode45<StateType,RhsType> integrator(f);
14 // Set rather loose tolerances
15 integrator.options.rtol = 0.1;
```

```
integrator.options.atol = 0.001;
integrator.options.min_dt = 1E-18;
std::vector<std::pair<StateType, double>> states =
    integrator.solve(y0,1.0,normFunc);
// Output information accumulation during numerical integration
integrator.options.do_statistics = true; integrator.print();
```

Statistics of the integrator run

```
- number of steps:                183
- number of rejected steps:       185
- function calls:                 1302
```

The following plots have been generated with MATLAB using its built-in adaptive explicit RK-SSM:

MATLAB-script 12.0.4: Use of MATLAB integrator `ode45` for a stiff problem

```
fun = @(t,x) 500*x^2*(1-x);
options = odeset('reltol',0.1,'abstol',0.001,'stats','on');
[t,y] = ode45(fun,[0 1],y0,options);
```

The option `stats = 'on'` makes MATLAB print statistics about the run of the integrators.

```
186 successful steps
55 failed attempts
1447 function evaluations
```

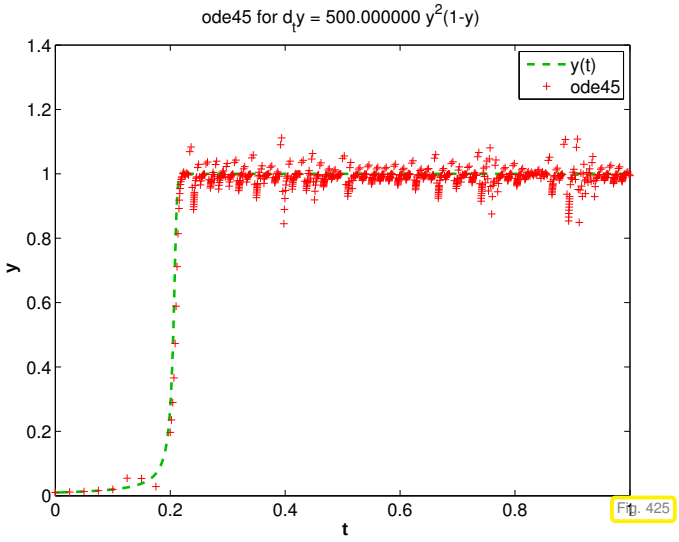


Fig. 424

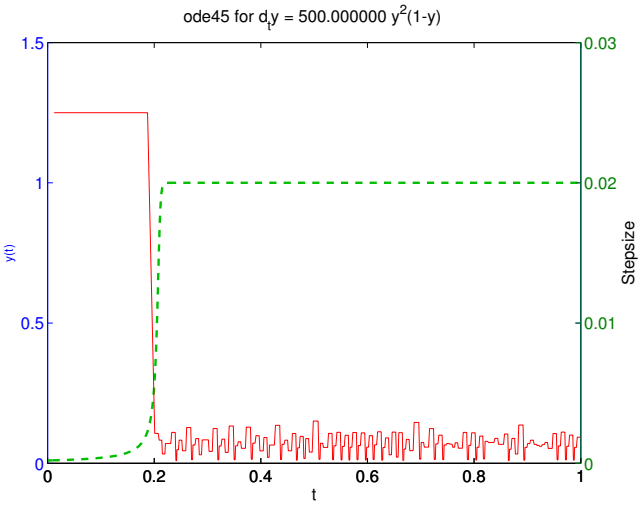


Fig. 425

Stepsize control of `ode45` running amok!

? The solution is virtually constant from $t > 0.2$ and, nevertheless, the integrator uses tiny timesteps until the end of the integration interval.

Contents

12.1 Model problem analysis

Supplementary reading. See also [**?**, Ch. 77], [**?**, Sect. 11.3.3].

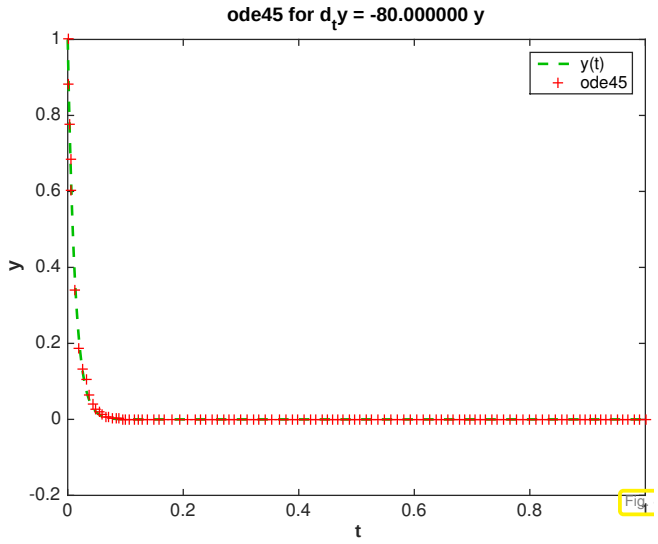
Experiment 12.1.1 (Adaptive explicity RK-SSM for simple decay ODE)

To rule out that what we observed in Ex. 12.0.1 might have been a quirk of the IVP (12.0.2) we conduct the same investigations for the simple linear, scalar, autonomous IVP

$$\dot{y} = -\lambda y \quad , \quad \lambda := 80 \quad , \quad y(0) = 1 \, . \tag{12.1.2}$$

We use the **ode45** class to solve (12.1.2) with the same parameters as in Code 12.0.3. ▷

```
1 - number of steps:                33
2 - number of rejected steps:       32
3 - function calls:                 231
```



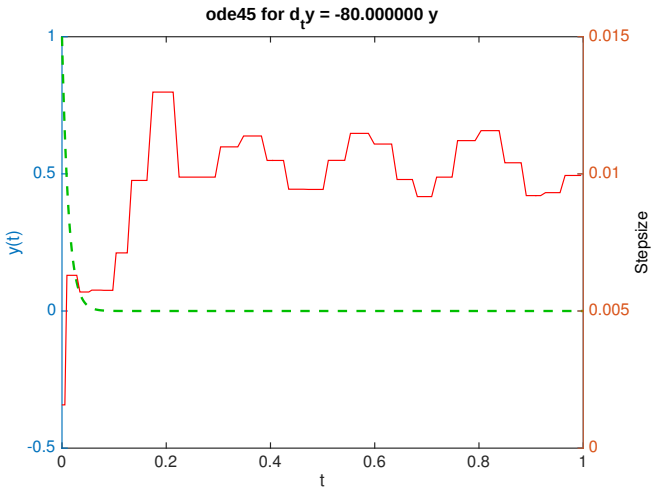


Fig.

Observation: Though $y(t) \approx 0$ for $t > 0.1$, the integrator keeps on using "unreasonably small" timesteps even then.

In this section we will discover a simple explanation for the startling behavior of **ode45** in Ex. 12.0.1.

Example 12.1.3 (Blow-up of explicit Euler method)

The simplest explicit RK-SSM is the explicit Euler method, see Section 11.2.1. We know that it should converge like $O(h)$ for meshwidth $h \to 0$. In this example we will see that this may be true only for sufficiently small h, which may be extremely small.

✦ We consider the IVP for the scalar linear decay ODE:

$$\dot{y} = f(y) := \lambda y \quad , \quad y(0) = 1 \, .$$

✦ We apply the explicit Euler method (11.2.7) with uniform timestep $h = 1/N$, $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$.

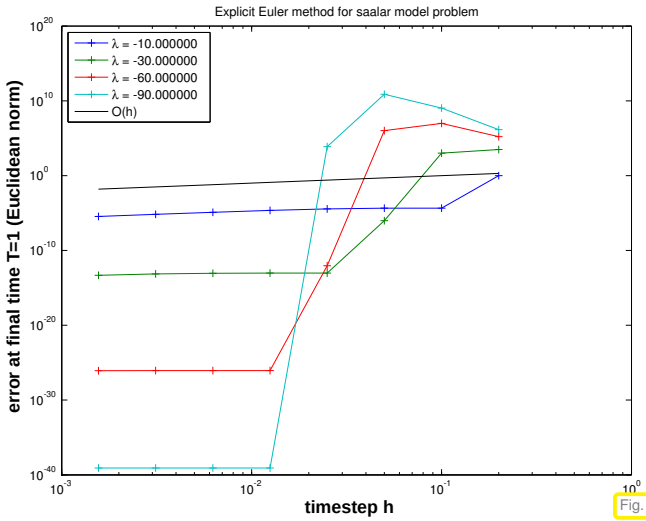


Fig. 428

λ large: blow-up of y_k for large timestep h

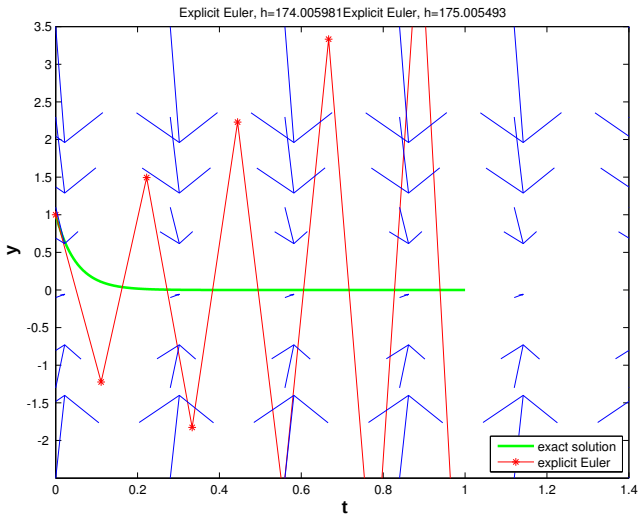


Fig. 429

$\lambda = -20$: — $\hat{=}$ $y(t)$, — $\hat{=}$ Euler polygon

Explanation: From Fig. 429 we draw the geometric conclusion that, if h is "large in comparison with λ^{-1}", then the approximations y_k way miss the stationary point $y = 0$ due to overshooting.

This leads to a sequence $(y_k)_k$ with exponentially increasing oscillations.

✦ Now we look at an IVP for the logistic ODE, see Ex. 11.1.5:

$$\dot{y} = f(y) := \lambda y(1-y) \quad , \quad y(0) = 0.01 \, .$$

✦ As before, we apply the explicit Euler method (11.2.7) with uniform timestep $h = 1/N$, $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$.

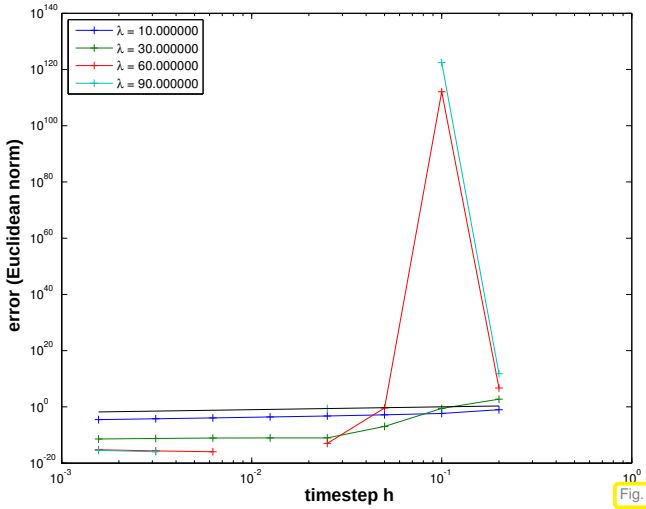


Fig. 430

λ large: blow-up of y_k for large timestep h

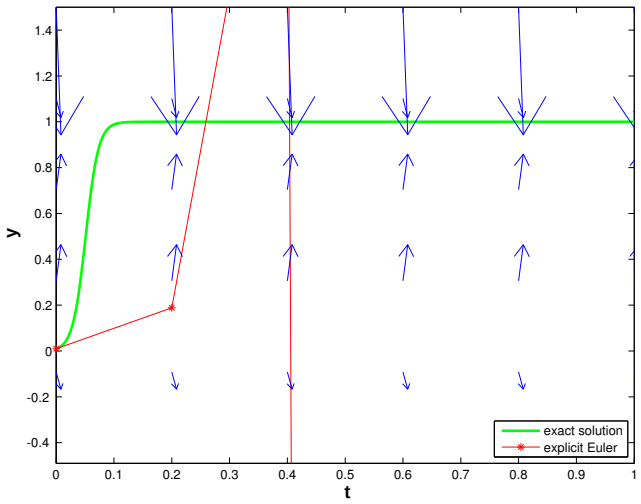


Fig. 431

$\lambda = 90$: — $\hat{=}$ $y(t)$, — $\hat{=}$ Euler polygon

For large timesteps h we also observe oscillatory blow-up of the sequence $(y_k)_k$.

Deeper analysis:

For $y \approx 1$: $f(y) \approx \lambda(1-y)$ ➢ If $y(t_0) \approx 1$, then the solution of the IVP will behave like the solution of $\dot{y} = \lambda(1-y)$, which is a linear ODE. Similary, $z(t) := 1 - y(t)$ will behave like the solution of the "decay equation" $\dot{z} = -\lambda z$. Thus, around the stationary point $y = 1$ the explicit Euler method behaves like it did for $\dot{y} = \lambda y$ in the vicinity of the stationary point $y = 0$; it grossly overshoots.

(12.1.4) Linear model problem analysis: explicit Euler method

The phenomenon observed in the two previous examples is accessible to a remarkably simple rigorous analysis: Motivated by the considerations in Ex. 12.1.3 we study the explicit Euler method (11.2.7) for the

$$\text{linear model problem:} \quad \dot{y} = \lambda y \;, \quad y(0) = y_0 \;, \quad \text{with} \quad \lambda \ll 0 \;, \tag{12.1.5}$$

which has *exponentially decaying* exact solution

$$y(t) = y_0 \exp(\lambda t) \to 0 \quad \text{for } t \to \infty \;.$$

Recall the recursion for the explicit Euler with uniform timestep $h > 0$ method for (12.1.5):

$$\text{(11.2.7) for } f(y) = \lambda y\text{:} \qquad y_{k+1} = y_k(1 + \lambda h) \;. \tag{12.1.6}$$

We easily get a closed form expression for the approximations y_k:

$$\blacktriangleright \quad y_k = y_0(1+\lambda h)^k \quad \Rightarrow \quad |y_k| \to \begin{cases} 0 & \text{, if } \lambda h > -2 \quad \text{(qualitatively correct) ,} \\ \infty & \text{, if } \lambda h < -2 \quad \text{(qualitatively wrong) .} \end{cases}$$

Observed: timestep constraint

Only if $|\lambda| h < 2$ we obtain a decaying solution by the explicit Euler method!

Could it be that the timestep control is desperately trying to enforce the qualitatively correct behavior of the numerical solution in Ex. 12.1.3? Let us examine how the simple stepsize control of Code 11.5.11 fares for model problem (12.1.5):

Example 12.1.8 (Simple adaptive timestepping for fast decay)

In this example we let a transparent adaptive timestep struggle with "overshooting":

- "Linear model problem IVP": $\dot{y} = \lambda y$, $y(0) = 1$, $\lambda = -100$
- Simple adaptive timestepping method as in Ex. 11.5.13, see Code 11.5.11. Timestep control based on the pair of 1st-order explicit Euler method and 2nd-order explicit trapezoidal method.

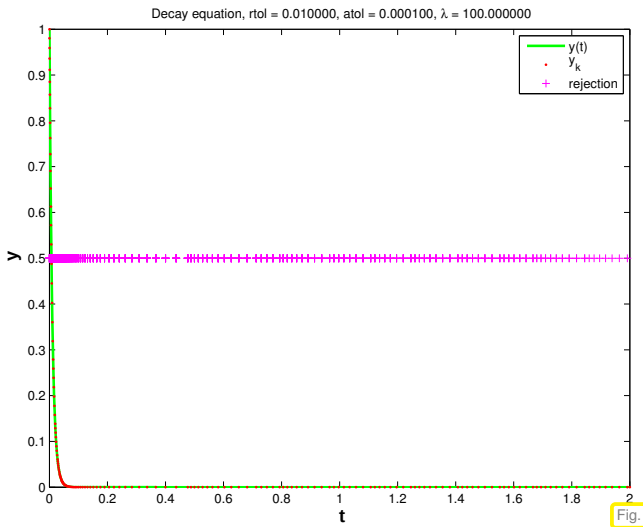


Fig. 432

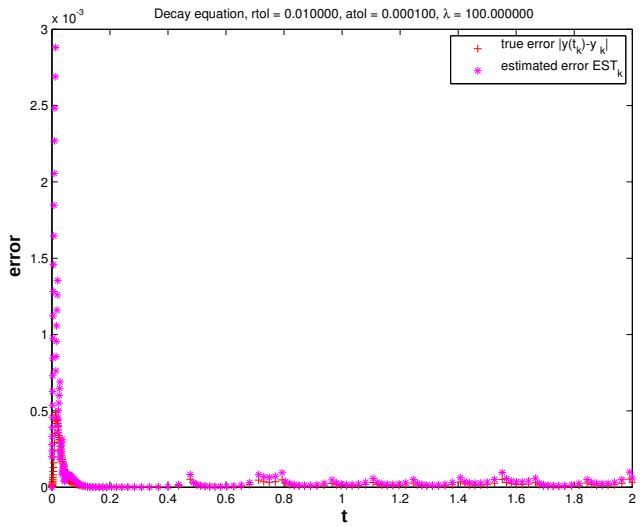


Fig. 433

Observation: in fact, stepsize control enforces small timesteps even if $\mathbf{y}(t) \approx 0$ and persistently triggers rejections of timesteps. This is necessary to prevent overshooting in the Euler method, which contributes to the estimate of the one-step error.

We see the purpose of *stepsize control thwarted*, because after only a very short time the solution is almost zero and then, in fact, large timesteps should be chosen.

Are these observations a particular "flaw" of the explicit Euler method? Let us study the behavior of another simple explicit Runge-Kutta method applied to the linear model problem.

Example 12.1.9 (Explicit trapzoidal method for decay equation → [?, Ex. 11.29])

Recall recursion for the explicit trapezoidal method derived in Ex. 11.4.4:

$$\mathbf{k}_1 = \mathbf{f}(t_0, \mathbf{y}_0)\,, \quad \mathbf{k}_2 = \mathbf{f}(t_0 + h, \mathbf{y}_0 + h\mathbf{k}_1)\,, \quad \mathbf{y}_1 = \mathbf{y}_0 + \tfrac{h}{2}(\mathbf{k}_1 + \mathbf{k}_2)\,. \tag{11.4.6}$$

Apply it to the model problem (12.1.5), that is, the scalar autonomous ODE with right hand side function $\mathbf{f}(y) = f(y) = \lambda y$, $\lambda < 0$:

$$k_1 = \lambda y_0\,, \quad k_2 = \lambda(y_0 + hk_1) \quad \Rightarrow \quad y_1 = \underbrace{(1 + \lambda h + \tfrac{1}{2}(\lambda h)^2)}_{=:S(h\lambda)} y_0\,. \tag{12.1.10}$$

the sequence of approximations generated by the explicit trapezoidal rule can be expressed in closed form as

$$y_k = S(h\lambda)^k y_0\,, \quad k = 0, \ldots, N\,. \tag{12.1.11}$$

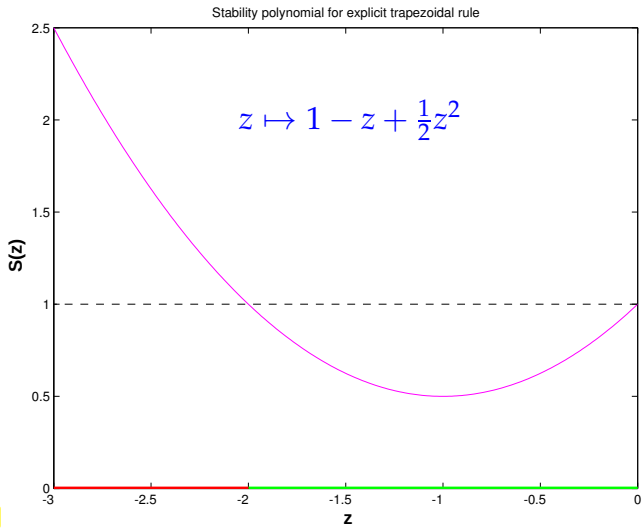


Fig. 434

Clearly, blow-up can be avoided only if $|S(h\lambda)| \leq 1$:

$$|S(h\lambda)| < 1 \quad \Leftrightarrow \quad -2 < h\lambda < 0\,.$$

Qualitatively correct decay behavior of $(y_k)_k$ only under timestep constraint

$$h \leq |2/\lambda|\,. \tag{12.1.12}$$

◁ the stability function for the explicit trapezoidal method

(12.1.13) Model problem analysis for general explicit Runge-Kutta single step methods

Apply the explicit Runge-Kutta method (→ Def. 11.4.9): encoded by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ to the autonomous scalar linear ODE (12.1.5) ($\dot{y} = \lambda y$). We write down the equations for the increments and y_1

from Def. 11.4.9 for $f(y) := \lambda y$ and then convert the resulting system of equations into matrix form:

$$\blacktriangleright \quad \begin{aligned} k_i &= \lambda(y_0 + h\sum_{j=1}^{i-1} a_{ij}k_j)\,, \\ y_1 &= y_0 + h\sum_{i=1}^{s} b_i k_i \end{aligned} \quad \Rightarrow \quad \begin{bmatrix} \mathbf{I} - z\mathfrak{A} & 0 \\ -z\mathbf{b}^\top & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{k} \\ y_1 \end{bmatrix} = y_0 \begin{bmatrix} \mathbf{1} \\ 1 \end{bmatrix}, \tag{12.1.14}$$

where $\boldsymbol{k} \in \mathbb{R}^s \triangleq$ denotes the vector $[k_1, \ldots, k_s]^\top / \lambda$ of increments, and $z := \lambda h$. Next we apply block Gaussian elimination (→ Rem. 2.3.11) to solve for y_1 and obtain

$$y_1 = S(z)y_0 \quad \text{with} \quad S(z) := 1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1}\,. \tag{12.1.15}$$

Alternatively we can express y_1 through determinants appealing to Cramer's rule,

$$y_1 = y_0 \frac{\det \begin{bmatrix} \mathbf{I} - z\mathfrak{A} & \mathbf{1} \\ -z\mathbf{b}^\top & 1 \end{bmatrix}}{\det \begin{bmatrix} \mathbf{I} - z\mathfrak{A} & 0 \\ -z\mathbf{b}^\top & 1 \end{bmatrix}} \quad \Rightarrow \quad S(z) = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T)\,, \tag{12.1.16}$$

and note that $\mathfrak{A}$ is a strictly lower triangular matrix, which means that $\det(\mathbf{I} - z\mathfrak{A}) = 1$. Thus we have proved the following theorem.

Theorem 12.1.17. Stability function of explicit Runge-Kutta methods → [?, Thm. 77.2], [?, Sect. 11.8.4]

The discrete evolution Ψ_λ^h of an explicit s-stage Runge-Kutta single step method (→ Def. 11.4.9) with Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ *(see* (11.4.11)*) for the ODE $\dot{y} = \lambda y$ amounts to a multiplication with the number*

$$\Psi_\lambda^h = S(\lambda h) \quad \Leftrightarrow \quad y_1 = S(\lambda h)y_0\,,$$

where S is the stability function

$$S(z) := 1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1} = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T)\,, \quad \mathbf{1} := [1, \ldots, 1]^\top \in \mathbb{R}^s\,. \tag{12.1.18}$$

Example 12.1.19 (Stability functions of explicit Runge-Kutta single step methods)

From Thm. 12.1.17 and their Butcher schemes we can instantly compute the stability functions of explicit RK-SSM. We do this for a few methods whose Butcher schemes were listed in Ex. 11.4.13

- Explicit Euler method (11.2.7): $\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$ ➢ $S(z) = 1 + z$.
- Explicit trapezoidal method (11.4.6): $\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$ ➢ $S(z) = 1 + z + \frac{1}{2}z^2$.

- Classical RK4 method:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array} \quad \succ \quad S(z) = 1 + z + \tfrac{1}{2}z^2 + \tfrac{1}{6}z^3 + \tfrac{1}{24}z^4 \,.$$

These examples confirm an immediate consequence of the determinant formula for the stability function $S(z)$.

Corollary 12.1.20. Polynomial stability function of explicit RK-SSM

For a consistent (→ Def. 11.3.10) s-stage explicit Runge-Kutta single step method according to Def. 11.4.9 the stability function S defined by (12.1.18) *is a non-constant polynomial of degree $\leq s$: $S \in \mathcal{P}_s$.*

Remark 12.1.21 (Stability function and exponential function)

Compare the two evolution operators:

- $\Phi \mathrel{\hat{=}}$ evolution operator (*to* Def. 11.1.39) for $\dot{y} = \lambda y$,
- $\Psi \mathrel{\hat{=}}$ discrete evolution operator (→ § 11.3.1) for an s-stage Runge-Kutta single step method.

$$\Phi^h y = e^{\lambda h} y \quad \longleftrightarrow \quad \Psi^h y = S(\lambda h) y \,.$$

In light of $\Psi \approx \Phi$, see (11.3.3), we expect that

$$S(z) \approx \exp(z) \quad \text{for small} \quad |z| \,. \tag{12.1.22}$$

A more precise statement is made by the following lemma:

Lemma 12.1.23. Stability function as approximation of $\exp$ for small arguments

Let S denote the stability function of an s-stage explicit Runge-Kutta single step method of order $q \in \mathbb{N}$. Then

$$|S(z) - \exp(z)| = O(|z|^{q+1}) \quad \textit{for} \quad |z| \to 0 \,. \tag{12.1.24}$$

This means that the lowest $q+1$ coefficients of $S(z)$ must be equal to the first coefficients of the exponential series:

$$S(z) = \sum_{j=0}^{q} \frac{1}{j!} z^j + z^{q+1} p(z) \quad \text{with some} \quad p \in \mathcal{P}_{s-q-1} \,.$$

Corollary 12.1.25. Stages limit order of explicit RK-SSM

An explicit s-stage RK-SSM has maximal order $q \leq s$.

(12.1.26) Stability induced timestep constraint

In § 12.1.13 we established that for the sequence $(y_k)_{k=0}^{\infty}$ produced by an explicit Runge-Kutta single step method applied to the linear scalar model ODE $\dot{y} = \lambda y$, $\lambda \in \mathbb{R}$, with uniform timestep $h > 0$ holds

$$y_{k+1} = S(\lambda h) y_k \quad \Rightarrow \quad y_k = S(\lambda h^k y_0 \, .$$

$$\begin{array}{rcl} (y_k)_{k=0}^{\infty} \text{ non-increasing} & \Leftrightarrow & |S(\lambda h)| \leq 1 \, , \\ (y_k)_{k=0}^{\infty} \text{ exponentially increasing} & \Leftrightarrow & |S(\lambda h)| > 1 \, . \end{array} \tag{12.1.27}$$

where $S = S(z)$ is the stability function of the RK-SSM as defined in (12.1.18).

Invariably polynomials tend to $\pm\infty$ for large (in modulus) arguments:

$$\forall S \in \mathcal{P}_s, S \neq \text{const} : \quad \lim_{|z| \to \infty} S(z) = \infty \quad \text{uniformly} \, . \tag{12.1.28}$$

So, for any $\lambda \neq 0$ there will be a threshold $h_{\max} > 0$ so that $|y_k| \to \infty$ as $|h| > h_{\max}$.

Reversing the argument we arrive at a timestep constraint, as already observed for the explicit Euler methods in § 12.1.4.

Only if one ensures that $|\lambda h|$ is sufficiently small, one can avoid exponentially increasing approximations y_k (qualitatively wrong for $\lambda < 0$) when applying an explicit RK-SSM to the model problem (12.1.5) with uniform timestep $h > 0$,

For $\lambda \ll 0$ this stability induced timestep constraint may force h to be much *smaller than required by demands on accuracy*: in this case timestepping becomes inefficient.

Remark 12.1.29 (Stepsize control detects instability)

Ex. 12.0.1, Ex. 12.1.8 send the message that local-in-time stepsize control as discussed in Section 11.5 selects timesteps that avoid blow-up, with a hefty price tag however in terms of computational cost and poor accuracy.

Objection: simple linear scalar IVP (12.1.5) may be an oddity rather than a model problem: the weakness of explicit Runge-Kutta methods discussed above may be just a peculiar response to an unusual situation. Let us extend our investigations to systems of linear ODEs, $d > 1$.

(12.1.30) Systems of linear ordinary differential equations

A generic linear ordinary differential equation on state space $\mathbb{R}^d$ has the form

$$\dot{\mathbf{y}} = \mathbf{M}\mathbf{y} \quad \text{with a matrix} \quad \mathbf{M} \in \mathbb{R}^{d,d} \, . \tag{12.1.31}$$

As explained in [**?**, Sect. 8.1], (12.1.31) can be solved by diagonalization: If we can find a *regular* matrix $\mathbf{V} \in \mathbb{C}^{d,d}$ such that

$$\mathbf{MV} = \mathbf{VD} \quad \text{with diagonal matrix} \quad \mathbf{D} = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_d \end{bmatrix} \in \mathbb{C}^{d,d} , \tag{12.1.32}$$

then the 1-parameter family of global solutions of (12.1.31) is given by

$$\mathbf{y}(t) = \mathbf{V} \begin{bmatrix} \exp(\lambda_1 t) & & 0 \\ & \ddots & \\ 0 & & \exp(\lambda_d t) \end{bmatrix} \mathbf{V}^{-1}\mathbf{y}_0 , \quad \mathbf{y}_0 \in \mathbb{R}^d . \tag{12.1.33}$$

The columns of $\mathbf{V}$ are a basis of eigenvectors of $\mathbf{M}$, the $\lambda_j \in \mathbb{C}$, $j = 1, \ldots, d$ are the associated eigenvalues of $\mathbf{M}$, see Def. 9.1.1.

The idea behind diagonalization is the transformation of (12.1.31) into d *decoupled* scalar linear ODEs:

$$\dot{\mathbf{y}} = \mathbf{My} \xrightarrow{\mathbf{z}(t):=\mathbf{V}^{-1}\mathbf{y}(t)} \dot{\mathbf{z}} = \mathbf{Dz} \leftrightarrow \begin{aligned} \dot{z}_1 &= \lambda_1 z_1 \\ &\vdots \\ \dot{z}_d &= \lambda_d z_d \end{aligned} \quad \text{, since} \quad \mathbf{M} = \mathbf{VDV}^{-1} .$$

The formula (12.1.33) can be generalized to

$$\mathbf{y}(t) = \exp(\mathbf{M}t)\mathbf{y}_0 \quad \text{with matrix exponential} \quad \exp(\mathbf{B}) := \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{B}^k , \quad \mathbf{B} \in \mathbb{C}^{d,d} . \tag{12.1.34}$$

Example 12.1.35 (Transient simulation of RLC-circuit)

Consider circuit from Ex. 11.1.13 ▷

Transient nodal analysis leads to the second-order linear ODE

$$\ddot{u} + \alpha\dot{u} + \beta u = g(t) ,$$

with coefficients $\alpha := (RC)^{-1}$, $\beta = (LC)^{-1}$, $g(t) = \alpha\dot{U}_s(t)$.

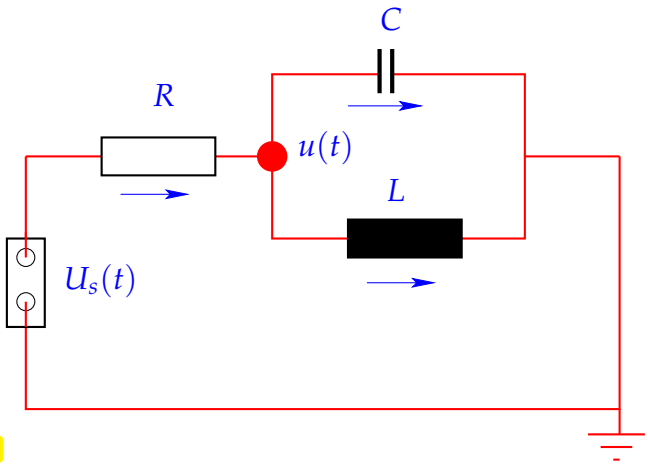


Fig. 435

We transform it to a linear 1st-order ODE as in Rem. 11.1.23 by introducing $v := \dot{u}$ as additional solution component:

$$\underbrace{\begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix}}_{=:\dot{\mathbf{y}}} = \underbrace{\begin{bmatrix} 0 & 1 \\ -\beta & -\alpha \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} - \begin{bmatrix} 0 \\ g(t) \end{bmatrix}}_{=:\mathbf{f}(t,\mathbf{y})} .$$

We integrate IVPs for this ODE by means of MATLAB's adaptive integrator `ode45`.

MATLAB-code 12.1.36: simulation of linear RLC circuit using `ode45`

```
function stiffcircuit(R,L,C,Us,tspan,filename)
% Transient simulation of simple linear circuit of
   Ex. refex:stiffcircuit
% R,L,C: paramters for circuits elements (compatible units required)
% Us:  exciting time-dependent voltage U_s = U_s(t), function handle
% zero initial values

% Coefficient for 2nd-order ODE ü + αu̇ + β = g(t)
alpha = 1/(R*C); beta = 1/(C*L);
% Conversion to 1st-order ODE  y = My + (0; g(t)).  Set up right hand side
   function.
M = [0 , 1; -beta , -alpha]; rhs = @(t,y) (M*y - [ 0 ;
   alpha*Us(t)]);
% Set tolerances for MATLAB integrator, see Rem. 11.5.23
options = odeset('reltol',0.1,'abstol',0.001,'stats','on');
y0 = [0;0]; [t,y] = ode45(rhs,tspan,y0,options);
```

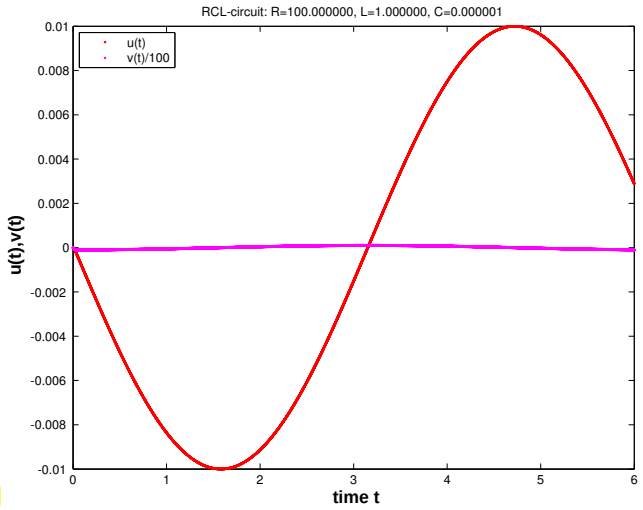


Fig. 436

$R = 100\Omega$, $L = 1\mathrm{H}$, $C = 1\mu\mathrm{F}$, $U_s(t) = 1\mathrm{V}\sin(t)$, $u(0) = v(0) = 0$ ("switch on")

`ode45` statistics:
```
17897 successful steps
1090 failed attempts
113923 function evaluations
```

Inefficient: way more timesteps than required for resolving smooth solution, *cf.* remark in the end of § 12.1.26.

Maybe the time-dependent right hand side due to the time-harmonic excitation severly affects `ode45`? Let us try a constant exciting voltage:

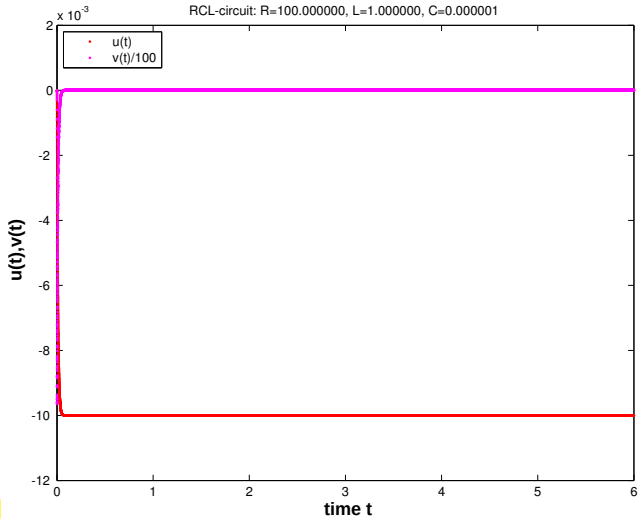


Fig. 437

$R = 100\Omega$, $L = 1\mathrm{H}$, $C = 1\mu\mathrm{F}$, $U_s(t) = 1\mathrm{V}$, $u(0) = v(0) = 0$ ("switch on")

`ode45` statistics:
```
17901 successful steps
1210 failed attempts
114667 function evaluations
```

Tiny timesteps despite virtually constant solution!

We make the same observation as in Ex. 12.0.1, Ex. 12.1.8: the local-in-time stepsize control of `ode45` (→ Section 11.5) enforces extremely small timesteps though the solution almost constant except at $t = 0$.

To understand the structure of the solutions for this transient circuit example, let us apply the diagonalization technique from § 12.1.30 to the linear ODE

$$\dot{\mathbf{y}} = \underbrace{\begin{bmatrix} 0 & 1 \\ -\beta & -\alpha \end{bmatrix}}_{=:\mathbf{M}} \mathbf{y} \quad , \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{R}^2 \, . \tag{12.1.37}$$

Above we face the situation $\beta \gg \frac{1}{4}\alpha^2 \gg 1$.

We can obtain the general solution of $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{R}^{2,2}$, by diagonalization of $\mathbf{M}$ (if possible):

$$\mathbf{M}\mathbf{V} = \mathbf{M}(\mathbf{v}_1, \mathbf{v}_2) = (\mathbf{v}_1, \mathbf{v}_2) \begin{bmatrix} \lambda_1 & \\ & \lambda_2 \end{bmatrix} . \tag{12.1.38}$$

where $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^2 \setminus \{0\}$ are the the eigenvectors of $\mathbf{M}$, λ_1, λ_2 are the eigenvalues of $\mathbf{M}$, see Def. 9.1.1. For the latter we find

$$\lambda_{1/2} = \tfrac{1}{2}(\alpha \pm D) \, , \quad D := \begin{cases} \sqrt{\alpha^2 - 4\beta} & \text{, if } \alpha^2 \geq 4\beta \, , \\ \imath\sqrt{4\beta - \alpha^2} & \text{, if } \alpha^2 < 4\beta \, . \end{cases}$$

Note that the eigenvalue have non-vanishing imaginary part in the setting of the experiment.

Then we transform $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ into *decoupled* scalar linear ODEs:

$$\dot{\mathbf{y}} = \mathbf{M}\mathbf{y} \quad \Leftrightarrow \quad \mathbf{V}^{-1}\dot{\mathbf{y}} = \mathbf{V}^{-1}\mathbf{M}\mathbf{V}(\mathbf{V}^{-1}\mathbf{y}) \quad \overset{\mathbf{z}(t):=\mathbf{V}^{-1}\mathbf{y}(t)}{\Leftrightarrow} \quad \dot{\mathbf{z}} = \begin{bmatrix} \lambda_1 & \\ & \lambda_2 \end{bmatrix} \mathbf{z} \, . \tag{12.1.39}$$

This yields the general solution of the ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, see also [**?**, Sect. 5.6]:

$$\mathbf{y}(t) = A\mathbf{v}_1 \exp(\lambda_1 t) + B\mathbf{v}_2 \exp(\lambda_2 t) \, , \quad A, B \in \mathbb{R} \, . \tag{12.1.40}$$

Note: $t \mapsto \exp(\lambda_i t)$ is general solution of the ODE $\dot{z}_i = \lambda_i z_i$.

(12.1.41) "Diagonalization" of explicit Euler method

Recall discrete evolution of explicit Euler method (11.2.7) for ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{R}^{d,d}$:

$$\boldsymbol{\Psi}^h\mathbf{y} = \mathbf{y} + h\mathbf{M}\mathbf{y} \quad \leftrightarrow \quad \mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{M}\mathbf{y}_k \, .$$

As in § 12.1.30 we assume that $\mathbf{M}$ can be diagonalized, that is (12.1.32) holds: $\mathbf{V}^{-1}\mathbf{M}\mathbf{V} = \mathbf{D}$ with a diagonal matrix $\mathbf{D} \in \mathbb{C}^{d,d}$ containing the eigenvalues of $\mathbf{M}$ on its diagonal. Next, apply the decoupling by diagonalization idea to the recursion of the explicit Euler method.

$$\mathbf{V}^{-1}\mathbf{y}_{k+1} = \mathbf{V}^{-1}\mathbf{y}_k + h\mathbf{V}^{-1}\mathbf{M}\mathbf{V}(\mathbf{V}^{-1}\mathbf{y}_k) \quad \overset{\mathbf{z}_k:=\mathbf{V}^{-1}\mathbf{y}_k}{\Leftrightarrow} \quad \underbrace{(\mathbf{z}_{k+1})_i = (\mathbf{z}_k)_i + h\lambda_i(\mathbf{z}_k)_i}_{\hat{=} \text{ explicit Euler step for } \dot{z}_i = \lambda_i z_i} \, . \tag{12.1.42}$$

Crucial insight:

The explicit Euler method generates uniformly bounded solution sequences $(\mathbf{y}_k)_{k=0}^{\infty}$ for $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ with diagonalizable matrix $\mathbf{M} \in \mathbb{R}^{d,d}$ with eigenvalues $\lambda_1, \ldots, \lambda_d$, if and only if it generates uniformly bounded sequences for **all** the scalar ODEs $\dot{z} = \lambda_i z$, $i = 1, \ldots, d$.

So far we conducted the model problem analysis under the premises $\lambda < 0$.

However, in Ex. 12.1.35 we face $\lambda_{1/2} = -\frac{1}{2}\left(\alpha \pm i\sqrt{4\beta - \alpha^2}\right)$ (complex eigenvalues!). Let us now examine how the explicit Euler method and even general explicit RK-methods respond to them.

Example 12.1.43 (Explicit Euler method for damped oscillations)

Consider linear model IVP (12.1.5) for $\lambda \in \mathbb{C}$:

$$\operatorname{Re}\lambda < 0 \quad \Rightarrow \quad \text{exponentially decaying solution} \quad y(t) = y_0 \exp(\lambda t)\,,$$

because $|\exp(\lambda t)| = \exp(\operatorname{Re}\lambda \cdot t)$.

The model problem analysis from Ex. 12.1.3, Ex. 12.1.9 can be extended verbatim to the case of $\lambda \in \mathbb{C}$. It yields the following insight for the for the explicit Euler method and $\lambda \in \mathbb{C}$:

The sequence generated by the explicit Euler method (11.2.7) for the model problem (12.1.5) satisfies

$$y_{k+1} = y_k(1 + h\lambda) \quad \blacktriangleright \quad \lim_{k\to\infty} y_k = 0 \quad \Leftrightarrow \quad |1 + h\lambda| < 1\,. \tag{12.1.6}$$

timestep constraint to get decaying (discrete) solution !

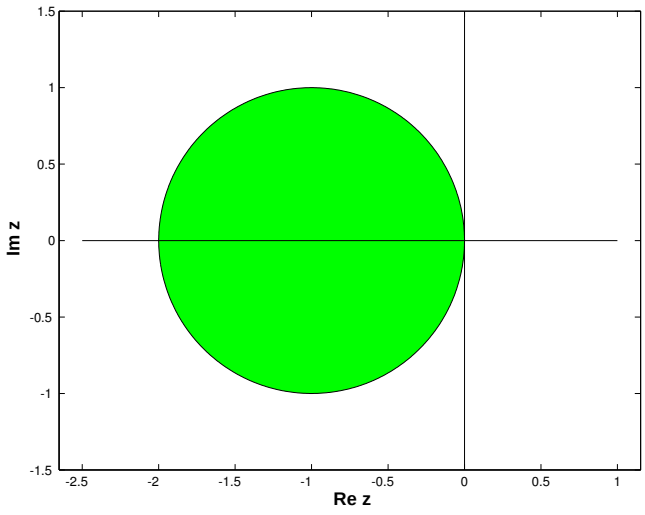


Fig. 438

$\lhd\ \{z \in \mathbb{C}:\ |1 + z| < 1\}$

The green region of the complex plane marks values for λh, for which the explicit Euler method will produce exponentially decaying solutions.

Now we can conjecture what happens in Ex. 12.1.35: the eigenvalues $\lambda_{1/2} = -\frac{1}{2}\alpha \pm i\sqrt{\beta - \frac{1}{4}\alpha^2}$ of $\mathbf{M}$ have a very large (in modulus) negative real part. Since `ode45` can be expected to behave as if it integrates $\dot{z} = \lambda_2 z$, it faces a severe timestep constraint, if exponential blow-up is to be avoided, see Ex. 12.1.3. Thus stepsize control must resort to tiny timesteps.

(12.1.44) Extended model problem analysis for explicit Runge-Kutta single step methods

We apply an explicit s-stage RK-SSM ($\rightarrow$ crefdef:rk) described by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ to the autonomous linear ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{C}^{d,d}$, and obtain (for the first step with timestep size $h > 0$)

$$\mathbf{k}_\ell = \mathbf{M}(\mathbf{y}_0 + h\sum_{j=1}^{s-1} a_{\ell j}\mathbf{k}_j)\,,\quad \ell = 1,\ldots,s \quad,\quad \mathbf{y}_1 = \mathbf{y}_0 + h\sum_{\ell=1}^{s} b_i\mathbf{k}_\ell\,. \tag{12.1.45}$$

Now assume that $\mathbf{M}$ can be diagonalized, that is (12.1.32) holds: $\mathbf{V}^{-1}\mathbf{M}\mathbf{V} = \mathbf{D}$ with a diagonal matrix $\mathbf{D} \in \mathbb{C}^{d,d}$ containing the eigenvalues $\lambda_i \in \mathbb{C}$ of $\mathbf{M}$ on its diagonal. Then apply the substitutions

$$\widehat{\mathbf{k}}_\ell := \mathbf{V}^{-1}\mathbf{k}_\ell \,, \quad \ell = 1,\ldots,s \quad , \quad \widehat{\mathbf{y}}_k := \mathbf{V}^{-1}\mathbf{y}_k \,, \quad k = 0,1 \,,$$

to (12.1.45), which yield

$$\widehat{\mathbf{k}}_\ell = \mathbf{D}(\widehat{\mathbf{y}}_0 + h\sum_{j=1}^{s-1} a_{\ell j}\widehat{\mathbf{k}}_j) \,, \quad \ell = 1,\ldots,s \quad , \quad \widehat{\mathbf{y}}_1 = \widehat{\mathbf{y}}_0 + h\sum_{\ell=1}^{s} b_i\widehat{\mathbf{k}}_\ell \,. \tag{12.1.46}$$

$$\Updownarrow$$

$$\left(\widehat{\mathbf{k}}_\ell\right)_i = \lambda_i((\mathbf{y}_0)_i + h\sum_{j=1}^{s-1} a_{\ell j}\left(\widehat{\mathbf{k}}_j\right)_i) \,, \quad (\widehat{\mathbf{y}}_1)_i = (\widehat{\mathbf{y}}_0)_i + h\sum_{\ell=1}^{s} b_i\left(\widehat{\mathbf{k}}_\ell\right)_i \,, \quad i = 1,\ldots,d \,. \tag{12.1.47}$$

We infer that, if $(\mathbf{y}_k)_k$ is the sequence produced by an explicit RK-SSM applied to $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, then

$$\mathbf{y}_k = \mathbf{V}\begin{bmatrix} y_k^{[1]} & & 0 \\ & \ddots & \\ 0 & & y_k^{[d]} \end{bmatrix}\mathbf{V}^{-1} \,,$$

where $\left(y_k^{[i]}\right)_k$ is the sequence generated by the same RK-SSM with the same sequence of timesteps for the IVP $\dot{y} = \lambda_i y$, $y(0) = \left(\mathbf{V}^{-1}\mathbf{y}_0\right)_i$.

The RK-SSM generates uniformly bounded solution sequences $(\mathbf{y}_k)_{k=0}^{\infty}$ for $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ with diagonalizable matrix $\mathbf{M} \in \mathbb{R}^{d,d}$ with eigenvalues $\lambda_1,\ldots,\lambda_d$, if and only if it generates uniformly bounded sequences for **all** the scalar ODEs $\dot{z} = \lambda_i z$, $i = 1,\ldots,d$.

Hence, understanding the behavior of RK-SSM for autonomous scalar linear ODEs $\dot{y} = \lambda y$ with $\lambda \in \mathbb{C}$ is enough to predict their behavior for general autonomous linear systems of ODEs.

From the considerations of § 12.1.26 we deduce the following fundamental result.

Theorem 12.1.48. (Absolute) stability of explicit RK-SSM for linear systems of ODEs

The sequence $(\mathbf{y}_k)_k$ of approximations generated by an explicit RK-SSM (→ Def. 11.4.9) with stability function S (defined in (12.1.18)*) applied to the linear autonomous ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{C}^{d,d}$, with uniform timestep $h > 0$ decays exponentially for* every *initial state $\mathbf{y}_0 \in \mathbb{C}^d$, if and only if $|S(\lambda_i h)| < 1$ for all eigenvalues λ_i of $\mathbf{M}$.*

Please note that

$$\operatorname{Re}\lambda_i < 0 \quad \forall i \in \{1,\ldots,d\} \quad \Longrightarrow \quad \|\mathbf{y}(t)\| \to 0 \quad \text{for} \quad t \to \infty \,,$$

for any solution of $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$. This is obvious from the representation formula (12.1.33).

(12.1.49) Region of (absolute) stability of explicit RK-SSM

We consider an explicit Runge-Kutta single step method with stability function S for the model linear scalar IVP $\dot{y} = \lambda y$, $y(0) = y_0$, $\lambda \in \mathbb{C}$. From Thm. 12.1.17 we learn that for uniform stepsize $h > 0$ we have $y_k = S(\lambda h)^k y_0$ and conclude that

$$y_k \to 0 \quad \text{for} \quad k \to \infty \quad \Leftrightarrow \quad |S(\lambda h)| < 1 \; . \tag{12.1.50}$$

Hence, the modulus $|S(\lambda h)|$ tells us for which combinations of λ and stepsize h we achieve exponential decay $y_k \to \infty$ for $k \to \infty$, which is the desirable behavior of the approximations for $\operatorname{Re}\lambda < 0$.

Definition 12.1.51. Region of (absolute) stability

Let the discrete evolution Ψ for a single step method applied to the scalar linear ODE $\dot{y} = \lambda y$, $\lambda \in \mathbb{C}$, be of the form

$$\Psi^h y = S(z) y \; , \quad y \in \mathbb{C}, h > 0 \quad \text{with} \quad z := h\lambda \tag{12.1.52}$$

and a function $S : \mathbb{C} \to \mathbb{C}$. Then the region of (absolute) stability of the single step method is given by

$$\mathcal{S}_\Psi := \{z \in \mathbb{C} : |S(z)| < 1\} \subset \mathbb{C} \; .$$

Of course, by Thm. 12.1.17, in the case of explicit RK-SSM the function S will coincide with their stability function from (12.1.18).

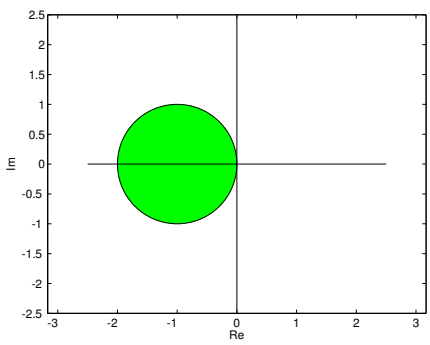
We can easily combine the statement of Thm. 12.1.48 with the concept of a region of stability and conclude that an explicit RK-SSM will generate expoentially decaying solutions for the linear ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{C}^{d,d}$, for every initial state $\mathbf{y}_0 \in \mathbb{C}^d$, if and only if $\lambda_i h \in \mathcal{S}_\Psi$ for all eigenvalues λ_i of $\mathbf{M}$.

Adopting the arguments of § 12.1.26 we conclude from Cor. 12.1.20 that

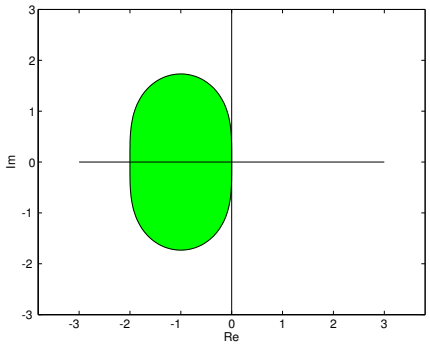
- the regions of (absolute) stability of explicit RK-SSM are bounded,
- a timestep constraint depending on the eigenvalues of $\mathbf{M}$ is necessary to have a guaranteed exponential decay RK-solutions for $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$.

Example 12.1.53 (Regions of stability of some explicit RK-SSM)

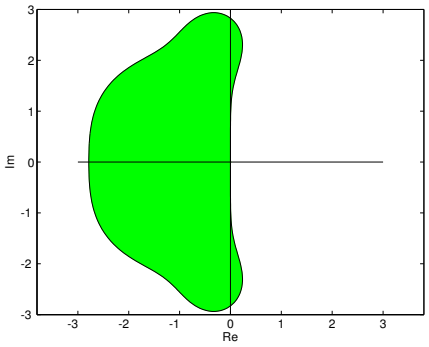
The green domains $\subset \mathbb{C}$ depict the bounded regions of stability for some RK-SSM from Ex. 11.4.13.



$\mathcal{S}_\Psi$: explicit Euler (11.2.7)



$\mathcal{S}_\Psi$: explicit trapezoidal method



$\mathcal{S}_\Psi$: classical RK4 method

In general we have for a consistent RK-SSM ($\rightarrow$ Def. 11.3.10) that their stability functions staidfy $S(z) = 1 + z + O(z^2)$ for $z \to 0$. Therefore, $\mathcal{S}_\Psi \neq \emptyset$ and the imaginary axis will be tangent to $\mathcal{S}_\Psi$ in $z = 0$.

12.2 Stiff Initial Value Problems

Supplementary reading. [**?**, Sect. 11.10]

This section will reveal that the behavior observed in Ex. 12.0.1 and Ex. 12.1.3 is typical for a large class of problems and that the model problem (12.1.5) really represents a "generic case". This justifies the attention paid to linear model problem analysis in Section 12.1.

Example 12.2.1 (Kinetics of chemical reactions $\rightarrow$ [?, Ch. 62])

In Ex. 11.5.1 we already saw an ODE model for the dynamics of a chemical reaction. Now we study an abstract reaction.

$$\text{reaction:} \quad \underbrace{\mathrm{A} + \mathrm{B} \underset{k_1}{\overset{k_2}{\leftrightarrows}} \mathrm{C}}_{\text{fast reaction}} \quad , \quad \underbrace{\mathrm{A} + \mathrm{C} \underset{k_3}{\overset{k_4}{\leftrightarrows}} \mathrm{D}}_{\text{slow reaction}} \tag{12.2.2}$$

Vastly different reaction constants: $k_1, k_2 \gg k_3, k_4$

If $c_A(0) > c_B(0)$ ➢ 2nd reaction determines overall long-term reaction dynamics

Mathematical model: non-linear ODE involving concentrations $\mathbf{y}(t) = (c_A(t), c_B(t), c_C(t), c_D(t))^T$

$$\dot{\mathbf{y}} := \frac{d}{dt}\begin{bmatrix} c_A \\ c_B \\ c_C \\ c_D \end{bmatrix} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} -k_1 c_A c_B + k_2 c_C - k_3 c_A c_C + k_4 c_D \\ -k_1 c_A c_B + k_2 c_C \\ k_1 c_A c_B - k_2 c_C - k_3 c_A c_C + k_4 c_D \\ k_3 c_A c_C - k_4 c_D \end{bmatrix}. \tag{12.2.3}$$

MATLAB computation: $t_0 = 0$, $T = 1$, $k_1 = 10^4$, $k_2 = 10^3$, $k_3 = 10$, $k_4 = 1$

MATLAB-script 12.2.4: Simulation of "stiff" chemical reaction

```
function chemstiff
% Simulation of kinetics of coupled chemical reactions with vastly
    different reaction
% rates, see (12.2.3) for the ODE model.
% reaction rates k1,k2,k3,k4, k1,k2 >> k3,k4.
k1 = 1E4; k2 = 1E3; k3 = 10; k4 = 1;
% definition of right hand side function for ODE solver
fun = @(t,y) ([-k1*y(1)*y(2) + k2*y(3) - k3*y(1)*y(3) +  k4*y(4);
               -k1*y(1)*y(2) + k2*y(3);
               k1*y(1)*y(2) - k2*y(3) - k3*y(1)*y(3) + k4*y(4);
               k3*y(1)*y(3) - k4*y(4)]);
tspan = [0 1]; % Integration time interval
L = tspan(2)-tspan(1); % Duration of simulation
y0 = [1;1;10;0]; % Initial value y0
% compute "exact" solution, using ode113 with tight error tolerances
options = odeset('reltol',10*eps,'abstol',eps,'stats','on');
% get the 'exact' solution using ode113
[tex,yex] = ode113(fun,[0 1],y0,options);
```

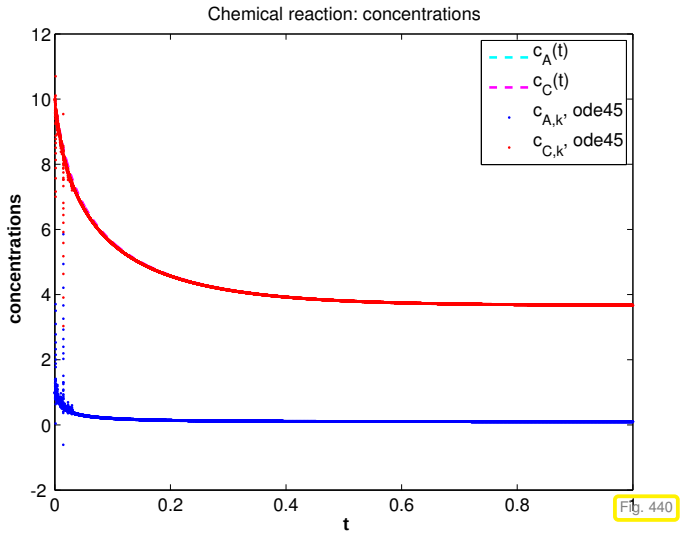


Fig. 439

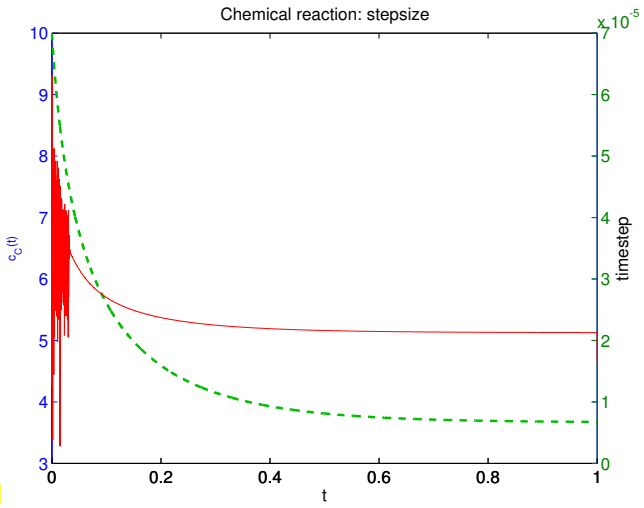


Fig. 440

Observations: After a fast initial transient phase, the solution shows only slow dynamics. Nevertheless, the explicit adaptive integrator `ode113` insists on using a tiny timestep. It behaves very much like `ode45` in Ex. 12.0.1.

Example 12.2.5 (Strongly attractive limit cycle)

We consider the non-linear Autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with

$$\mathbf{f}(\mathbf{y}) := \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2)\,\mathbf{y}\,, \tag{12.2.6}$$

on the state space $D = \mathbb{R}^2 \setminus \{0\}$

For $\lambda = 0$, the initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \binom{\cos\varphi}{\sin\varphi}$, $\varphi \in \mathbb{R}$ has the solution

$$\mathbf{y}(t) = \begin{bmatrix} \cos(t-\varphi) \\ \sin(t-\varphi) \end{bmatrix}, \quad t \in \mathbb{R}\,. \tag{12.2.7}$$

For this solution we have $\|\mathbf{y}(t)\|_2 = 1$ for all times.

(12.2.7) provides a solution even for $\lambda \neq 0$, if $\|\mathbf{y}(0)\|_2 = 1$, because in this case the term $\lambda(1 - \|\mathbf{y}\|^2)\,\mathbf{y}$ will never become non-zero on the solution trajectory.

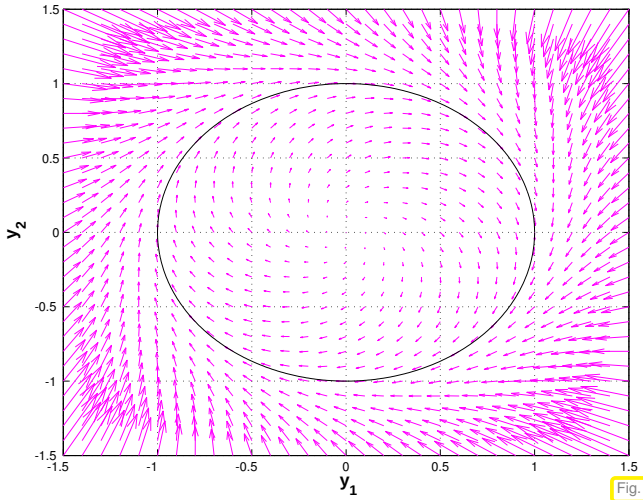


Fig. 441

vectorfield $\mathbf{f}$ ($\lambda = 1$)

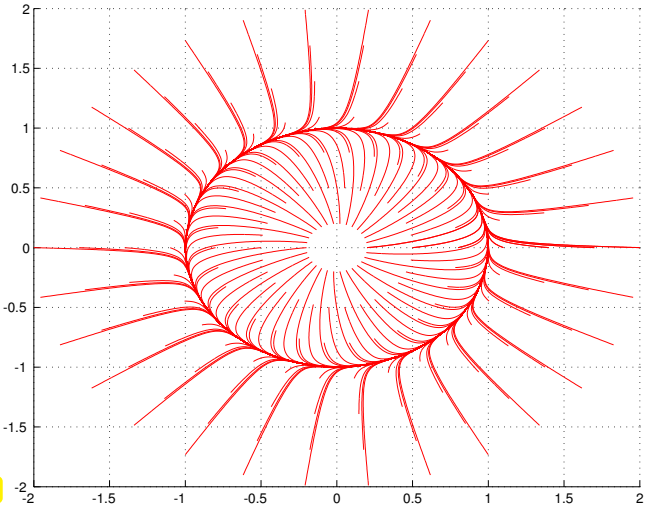


Fig. 442

solution trajectories ($\lambda = 10$)

MATLAB-script 12.2.8: Application of `ode45` for limit cycle problem

```
% MATLAB script for solving limit cycle ODE (12.2.6)
% define right hand side vectorfield
fun = @(t,y) ([-y(2);y(1)] +
   lambda*(1-y(1)\symbol{94}2-y(2)\symbol{94}2)*y);
% standard invocation of MATLAB integrator, see Ex. 11.4.21
tspan = [0,2*pi]; y0 = [1,0];
opts = odeset('stats','on','reltol',1E-4,'abstol',1E-4);
[t45,y45] = ode45(fun,tspan,y0,opts);
```

We study the response of `ode45` to different choice of λ with initial state $\mathbf{y}_0 = \begin{bmatrix}1\\0\end{bmatrix}$. According to the above considerations this initial state should completely "hide the impact of λ from our view".

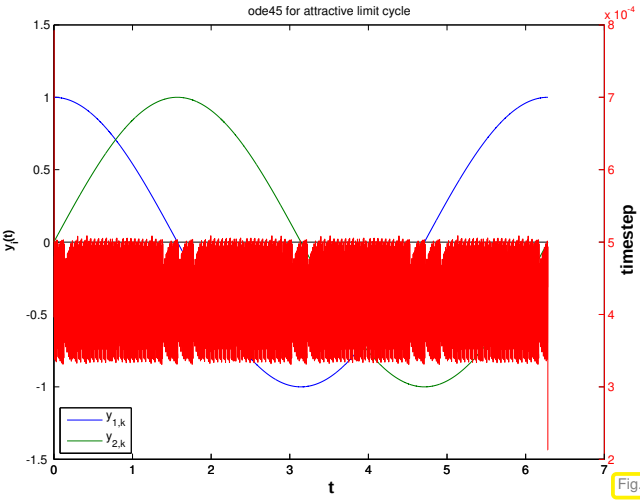


Fig. 443

many (3794) steps ($\lambda = 1000$)

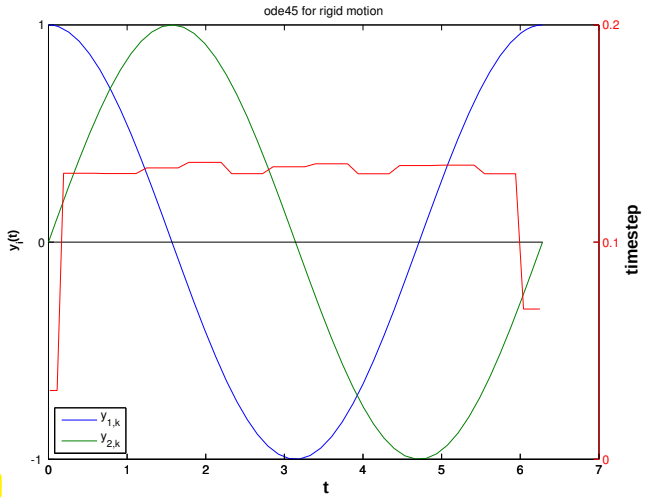


Fig. 444

accurate solution with few steps ($\lambda = 0$)

Confusing observation: we have $\|\mathbf{y}_0\| = 1$, which implies $\|\mathbf{y}(t)\| = 1 \quad \forall t$!

Thus, the term of the right hand side, which is multiplied by λ will always vanish on the exact solution trajectory, which stays on the unit circle.

Nevertheless, `ode45` is forced to use tiny timesteps by the *mere presence* of this term!

We want to find criteria that allow to predict the massive problems haunting explicit single step methods in the case of the *non-linear* IVP of Ex. 12.0.1, Ex. 12.2.1, and Ex. 12.2.5. Recall that for *linear* IVPs of the form $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{y}(0) = \mathbf{y}_0$, the model problem analysis of Section 12.1 tells us that, knowledge of the region of stability of the timestepping scheme, the eigenvalues of the matrix $\mathbf{M} \in \mathbb{C}^{d,d}$ provide full information about timestep constraint we are going to face. Refer to Thm. 12.1.48 and § 12.1.49.

Issue: extension of stability analysis to non-linear ODEs **?**

We start with a "phenomenological notion", just a keyword to refer to the kind of difficulties presented by the IVPs of Ex. 12.0.1, Ex. 12.2.1, Ex. 12.1.8, and Ex. 12.2.5.

Notion 12.2.9. Stiff IVP

An initial value problem is called stiff, if stability imposes much tighter timestep constraints on *explicit single step methods* than the accuracy requirements.

(12.2.10) Linearization of ODEs

We consider a general autonomous ODE: $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subset \mathbb{R}^d \to \mathbb{R}^d$

As usual, we assume $\mathbf{f}$ to be C^2-smooth and that it enjoys local Lipschitz continuity ($\to$ Def. 11.1.28) on D so that unique solvability of IVPs is guaranteed by Thm. 11.1.32.

We *fix* a state $\mathbf{y}^* \in D$, D the state space, write $t \mapsto \mathbf{y}(t)$ for the solution with $\mathbf{y}(0) = \mathbf{y}^*$. We set $\mathbf{z}(t) = \mathbf{y}(t) - \mathbf{y}^*$, which satisfies

$$\mathbf{z}(0) = 0 \quad , \quad \dot{\mathbf{z}} = \mathbf{f}(\mathbf{y}^* + \mathbf{z}) = \mathbf{f}(\mathbf{y}^*) + \mathrm{D}\,\mathbf{f}(\mathbf{y}^*)\mathbf{z} + R(\mathbf{y}^*, \mathbf{z}) \,, \quad \text{with} \quad \|R(\mathbf{y}^*, \mathbf{z})\| = O(\|\mathbf{z}\|^2) \,.$$

This is obtained by Taylor expansion of $\mathbf{f}$ at $\mathbf{y}^*$, see [**?**, Satz 7.5.2]. Hence, in a neighborhood of a state $\mathbf{y}^*$ on a solution trajectory $t \mapsto \mathbf{y}(t)$, the deviation $\mathbf{z}(t) = \mathbf{y}(t) - \mathbf{y}^*$ satisfies

$$\dot{\mathbf{z}} \approx \mathbf{f}(\mathbf{y}^*) + \mathrm{D}\,\mathbf{f}(\mathbf{y}^*)\mathbf{z} \,. \tag{12.2.11}$$

The short-time evolution of $\mathbf{y}$ with $\mathbf{y}(0) = \mathbf{y}^*$ is approximately governed by the affine-linear ODE

$$\boxed{\dot{\mathbf{y}} = \mathbf{M}(\mathbf{y} - \mathbf{y}^*) + \mathbf{b} \,, \quad \mathbf{M} := \mathrm{D}\,\mathbf{f}(\mathbf{y}^*) \in \mathbb{R}^{d,d} \,, \quad \mathbf{b} := \mathbf{f}(\mathbf{y}^*) \in \mathbb{R}^d} \,. \tag{12.2.12}$$

(12.2.13) Linearization of explicit Runge-Kutta single step methods

We consider one step a general s-stage RK-SSM according to Def. 11.4.9 for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, with smooth right hand side function $\mathbf{f} : D \subset \mathbb{R}^d \to \mathbb{R}^d$:

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0 + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j) \,, \quad i = 1, \ldots, s \quad , \quad \mathbf{y}_1 = \mathbf{y}_0 + h\sum_{i=1}^{s} b_i\mathbf{k}_i \,.$$

We perform linearization at $\mathbf{y}^* := \mathbf{y}_0$ and ignore all terms at least quadratic in the timestep size h:

$$\mathbf{k}_i \approx \mathbf{f}(\mathbf{y}^*) + \mathrm{D}\,\mathbf{f}(\mathbf{y}^*) h \sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j\,, \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 = \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i\,.$$

The defining equations for the same RK-SSM applied to

$$\dot{\mathbf{z}} = \mathbf{M}(\mathbf{z}) + \mathbf{b}\,, \quad \mathbf{M} := \mathrm{D}\,\mathbf{f}(\mathbf{y}^*) \in \mathbb{R}^{d,d}\,, \quad \mathbf{b} := \mathbf{f}(\mathbf{y}^*)\,,$$

which agrees with (12.2.12) after substitution $\mathbf{z}(t) - \mathbf{y}(t) - \mathbf{y}^*$, are

$$\mathbf{k}_i \approx \mathbf{b} + \mathbf{M} h \sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j\,, \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 = \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i\,.$$

We find that for small timesteps

> the discrete evolution of the RK-SSM for $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ in the state $\mathbf{y}^*$ is close to the discrete evolution of the same RK-SSM applied to the linearization (12.2.12) of the ODE in $\mathbf{y}^*$.

By straightforward manipulations of the defining equations of an explicit RK-SSM we find that, if

- $(\mathbf{y}_k)_k$ is the sequence of states generated by the RK-SSM applied to the affine-linear ODE $\dot{\mathbf{y}} = \mathbf{M}(\mathbf{y} - \mathbf{y}_0) + \mathbf{b}$, $\mathbf{M} \in \mathbb{C}^{d,d}$ regular,
- $(\mathbf{w}_k)_k$ is the sequence of states generated by the same RK-SSM applied to the linear ODE $\dot{\mathbf{w}} = \mathbf{M}\mathbf{w}$ and $\mathbf{w}_0 := \mathbf{M}^{-1}\mathbf{b}$, then

$$\mathbf{w}_k = \mathbf{y}_k - \mathbf{y}_0 + \mathbf{M}^{-1}\mathbf{b}\,.$$

➢ The analysis of the behavior of an RK-SSM for an affine-linear ODE can be reduces to understanding its behavior for a linear ODE with the same matrix.

Combined with the insights from § 12.1.44 this means that

> for small timestep the behavior of an explicit RK-SSM applied to $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ close to the state $\mathbf{y}^*$ is determined by the eigenvalues of the Jacobian $\mathrm{D}\,\mathbf{f}(\mathbf{y}^*)$.

In particular, if $\mathrm{D}\,\mathbf{f}(\mathbf{y}^*)$ has at least one eigenvalue whose modulus is large, then an exponential drift-off of the approximate states $\mathbf{y}_k$ away from $\mathbf{y}^*$ can only be avoided for sufficiently small timestep, again a timestep constraint.

How to distinguish stiff initial value problems

An initial value problem for an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ will probably be stiff, if, for substantial periods of time,

$$\min\{\mathrm{Re}\,\lambda : \lambda \in \sigma(\mathrm{D}\,\mathbf{f}(\mathbf{y}(t)))\} \ll 0\,, \tag{12.2.15}$$
$$\max\{0, \mathrm{Re}\,\lambda : \lambda \in \sigma(\mathrm{D}\,\mathbf{f}(\mathbf{y}(t)))\} \approx 0\,, \tag{12.2.16}$$

where $t \mapsto \mathbf{y}(t)$ is the solution trajectory and $\sigma(\mathbf{M})$ is the spectrum of the matrix $\mathbf{M}$, see Def. 9.1.1.

The condition (12.2.16) has to be read as "the real parts of all eigenvalues are below a bound with small modulus". If this is not the case, then the exact solution will experience blow-up. It will change drastically over very short periods of time and small timesteps will be required anyway in order to resolve this.

Example 12.2.17 (Predicting stiffness of non-linear IVPs)

❶ We consider the IVP from Ex. 12.0.1:

$$\text{IVP considered:} \quad \dot{y} = f(y) := \lambda y^2(1-y)\,, \quad \lambda := 500 \quad , \quad y(0) = \tfrac{1}{100}\,.$$

We find

$$f'(y) = \lambda(2y - 3y^2) \quad \Rightarrow \quad f'(1) = -\lambda\,.$$

Hence, in case $\lambda \gg 1$ as in Fig. 425, we face a stiff problem close to the stationary state $y = 1$. The observations made in Fig. 425 exactly match this prediction.

❷ The solution of the IVP from Ex. 12.2.5

$$\dot{\mathbf{y}} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2)\,\mathbf{y}\,, \quad \|\mathbf{y}_0\|_2 = 1\,. \tag{12.2.6}$$

satisfies $\|\mathbf{y}(t)\|_2 = 1$ for all times. Using the product rule (8.4.10) of multi-dimensional differential calculus, we find

$$\mathrm{D}\,\mathbf{f}(\mathbf{y}) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} + \lambda\left(-2\mathbf{y}\mathbf{y}^\top + (1 - \|\mathbf{y}\|_2^2\mathbf{I})\right)\,.$$

$$\blacktriangleright \quad \sigma(\mathrm{D}\,\mathbf{f}(\mathbf{y})) = \left\{-\lambda - \sqrt{\lambda^2 - 1}, -\lambda + \sqrt{\lambda^2 - 1}\right\} \quad \text{, if} \quad \|\mathbf{y}\|_2 = 1\,.$$

Thus, for $\lambda \gg 1$, $\mathrm{D}\,\mathbf{f}(\mathbf{y}(t))$ will always have an eigenvalue with large negative real part, whereas the other eigenvalue is close to zero: the IVP is stiff.

Remark 12.2.18 (Characteristics of stiff IVPs)

Often one can already tell from the expected behavior of the solution of an IVP, which is often clear from the modeling context, that one has to brace for stiffness.

Typical features of stiff IVPs:
- ✦ Presence of fast transients in the solution, see Ex. 12.1.3, Ex. 12.1.35,
- ✦ Occurrence of strongly attractive fixed points/limit cycles, see Ex. 12.2.5

12.3 Implicit Runge-Kutta Single Step Methods

Explicit Runge-Kutta single step method cannot escape tight timestep constraints for stiff IVPs that may render them inefficient, see § 12.1.49. In this section we are going to augment the class of Runge-Kutta methods by timestepping schemes that can cope well with stiff IVPs.

Supplementary reading. [**?**, Sect. 11.6.2], [**?**, Sect. 11.8.3]

12.3.1 The implicit Euler method for stiff IVPs

Example 12.3.1 (Euler methods for stiff decay IVP)

We revisit the setting of Ex. 12.1.3 and again consider Euler methods for the decay IVP

$$\dot{y} = \lambda y \quad , \quad y(0) = 1 \, , \quad \lambda < 0 \, .$$

We apply both the explicit Euler method (11.2.7) and the implicit Euler method (11.2.13) with uniform timesteps $h = 1/N$, $N \in \{5, 10, 20, 40, 80, 160, 320, 640\}$ and monitor the error at final time $T = 1$ for different values of λ.

Explicit Euler method (11.2.7)

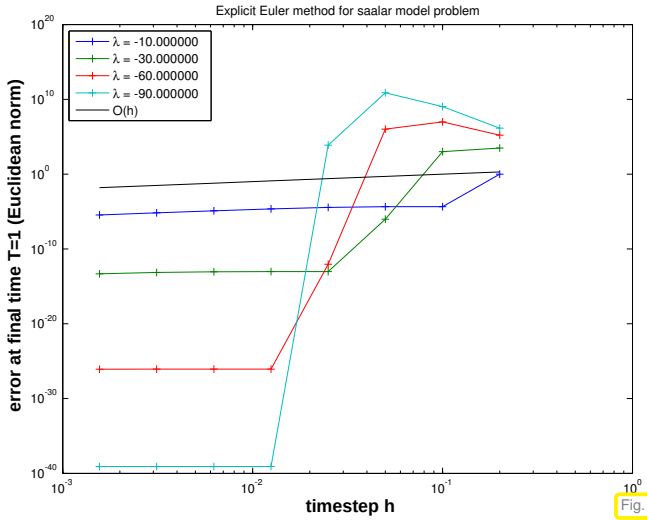


Fig. 445

λ large: blow-up of y_k for large timestep h

Implicit Euler method (11.2.13)

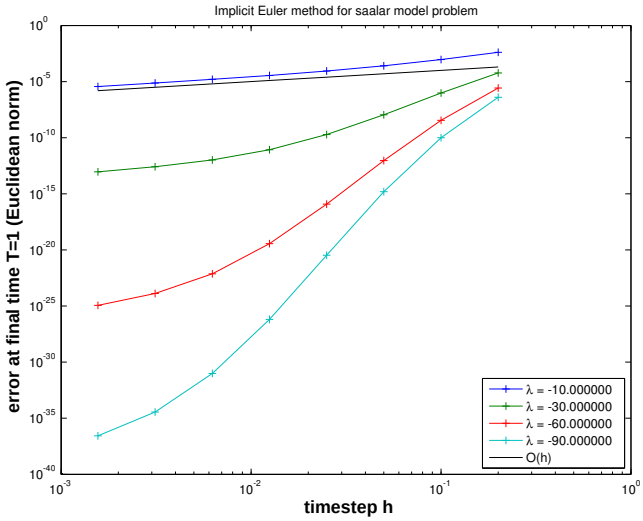


Fig. 446

λ large: stable for *all* timesteps $h > 0$ **!**

We observe onset of convergence of the implicit Euler method already for large timesteps h.

(12.3.2) Linear model problem analysis: implicit Euler method

We follow the considerations of § 12.1.4 and consider the *implicit* Euler method (11.2.13) for the

$$\text{linear model problem:} \quad \dot{y} = \lambda y \, , \quad y(0) = y_0 \, , \quad \text{with} \quad \operatorname{Re}\lambda \ll 0 \, , \tag{12.1.5}$$

with *exponentially decaying* (maybe osscillatory for $\operatorname{Im}\lambda \neq 0$) exact solution

$$y(t) = y_0 \exp(\lambda t) \to 0 \quad \text{for } t \to \infty \, .$$

The recursion of the implicit Euler method for (12.1.5) is defined by

$$\text{(11.2.13) for } f(y) = \lambda y \quad \Rightarrow \quad y_{k+1} = y_k + \lambda h y_{k+1} \; k \in \mathbb{N}_0 \, . \tag{12.3.3}$$

$$\blacktriangleright \quad \text{generated sequence} \quad y_k := \left(\frac{1}{1-\lambda h}\right)^k y_0 \, . \tag{12.3.4}$$

$$\Rightarrow \quad \boxed{\operatorname{Re}\lambda < 0 \quad \Rightarrow \quad \lim_{k\to\infty} y_k = 0 \quad \forall h > 0 \, !} \tag{12.3.5}$$

No timestep constraint: qualitatively correct behavior of $(y_k)_k$ for $\operatorname{Re}\lambda < 0$ and **any** $h > 0$!

As in § 12.1.41 this analysis can be extended to linear systems of ODEs $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{C}^{d,d}$, by means of diagonalization.

As in § 12.1.30 and § 12.1.41 we assume that $\mathbf{M}$ can be diagonalized, that is (12.1.32) holds: $\mathbf{V}^{-1}\mathbf{M}\mathbf{V} = \mathbf{D}$ with a diagonal matrix $\mathbf{D} \in \mathbb{C}^{d,d}$ containing the eigenvalues of $\mathbf{M}$ on its diagonal. Next, apply the decoupling by diagonalization idea to the recursion of the implicit Euler method.

$$\mathbf{V}^{-1}\mathbf{y}_{k+1} = \mathbf{V}^{-1}\mathbf{y}_k + h\underbrace{\mathbf{V}^{-1}\mathbf{M}\mathbf{V}}_{=\mathbf{D}}(\mathbf{V}^{-1}\mathbf{y}_{k+1}) \overset{\mathbf{z}_k := \mathbf{V}^{-1}\mathbf{y}_k}{\Leftrightarrow} \underbrace{(\mathbf{z}_{k+1})_i = \frac{1}{1-\lambda_i h}(\mathbf{z}_k)_i}_{\hat{=}\text{ implicit Euler step for } \dot{z}_i = \lambda_i z_i} . \tag{12.3.6}$$

Crucial insight:

For any timestep, the implicit Euler method generates exponentially decaying solution sequences $(\mathbf{y}_k)_{k=0}^{\infty}$ for $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ with diagonalizable matrix $\mathbf{M} \in \mathbb{R}^{d,d}$ with eigenvalues $\lambda_1, \ldots, \lambda_d$, if $\operatorname{Re}\lambda_i < 0$ for all $i = 1, \ldots, d$.

Thus we expect that the implicit Euler method will not face stability induced timestep constraints for stiff problems (→ Notion 12.2.9).

12.3.2 Collocation single step methods

Unfortunately the implicit Euler method is of first order only, see Ex. 11.3.18. This section presents an algorithm for designing higher order single step methods generalizing the implicit Euler method.

Setting: We consider the general ordinary differential equation $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, $\mathbf{f} : I \times D \to \mathbb{R}^d$ locally Lipschitz continuous, which guarantees the local existence of unique solutions of initial value problems, see Thm. 11.1.32.

We define the single step method through specifying the first step $\mathbf{y}_0 = \mathbf{y}(t_0) \to \mathbf{y}_1 \approx \mathbf{y}(t_1)$, where $\mathbf{y}_0 \in D$ is the initial step at initial time $t_0 \in I$. We assume that the exact solution trajectory $t \mapsto \mathbf{y}(t)$ exists on $[t_0, t_1]$. Use as a timestepping scheme on a temporal mesh (→ § 11.2.2) in the sense of Def. 11.3.5 is straightforward.

(12.3.7) Collocation principle

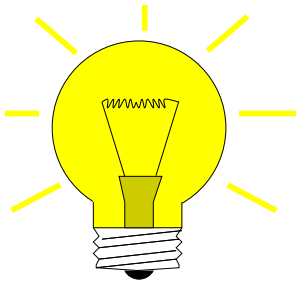
Abstract collocation idea

Collocation is a paradigm for the **discretization** (→ Rem. 11.3.4) of *differential equations*:

(I) Write the discrete solution u_h, a function, as linear combination of $N \in \mathbb{N}$ sufficiently smooth (basis) functions ➢ N unknown coefficients.

(II) Demand that u_h satisfies the differential equation at N points/times ➢ N equations.

We apply this policy to the differential equation $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ on $[t_0, t_1]$:



Idea: ❶ Approximate $t \mapsto \mathbf{y}(t)$, $t \in [t_0, t_1]$, by a function $t \mapsto \mathbf{y}_h(t) \in V$, V an $d \cdot (s+1)$-dimensional trial space V of functions $[t_0, t_1] \mapsto \mathbb{R}^d \rightarrow$ Item (I).

❷ Fix $\mathbf{y}_h \in V$ by imposing collocation conditions

$$\mathbf{y}_h(t_0) = \mathbf{y}_0 \quad , \quad \dot{\mathbf{y}}_h(\tau_j) = \mathbf{f}(\tau_j, \mathbf{y}_h(\tau_j)) \, , \quad j = 1, \ldots, s \, , \tag{12.3.9}$$

for collocation points $t_0 \leq \tau_1 < \ldots < \tau_s \leq t_1 \rightarrow$ Item (II).

❸ Choose $\mathbf{y}_1 := \mathbf{y}_h(t_1)$.

Our choice (the "standard option"): (Componentwise) polynomial trial space $V = (\mathcal{P}_s)^d$

Recalling $\dim \mathcal{P}_s = s+1$ from Thm. 5.2.2 we see that our choice makes the number $N := d(s+1)$ of collocation conditions match the dimension of the trial space V.

Now we want to derive a concrete representation for the polynomial $\mathbf{y}_h$. We draw on concepts introduced in Section 5.2.2. We define the collocation points as

$$\tau_j := t_0 + c_j h \, , \quad j = 1, \ldots, s \, , \quad \text{for} \quad 0 \leq c_1 < c_2 < \ldots < c_s \leq 1 \, , \quad h := t_1 - t_0 \, .$$

Let $\{L_j\}_{j=1}^s \subset \mathcal{P}_{s-1}$ denote the set of Lagrange polynomials of degree $s-1$ associated with the node set $\{c_j\}_{j=1}^s$, see (5.2.11). They satisfy $L_j(c_i) = \delta_{ij}$, $i, j = 1, \ldots, s$ and form a basis of $\mathcal{P}_{s-1}$.

In each of its d components, the derivative $\dot{\mathbf{y}}_h$ is a polynomial of degree $s-1$: $\dot{\mathbf{y}} \in (\mathcal{P}_{s-1})^d$. Hence, it has the following representation, compare (5.2.13).

$$\dot{\mathbf{y}}_h(t_0 + \tau h) = \sum_{j=1}^s \dot{\mathbf{y}}_h(t_0 + c_j h) L_j(\tau) \, . \tag{12.3.10}$$

As $\tau_j = t_0 + c_j h$, the comcollocation conditions make it possible to replace $\dot{\mathbf{y}}_h(c_j h)$ with an expression in the right hand side function $\mathbf{f}$:

(12.3.9) ◤ $\dot{\mathbf{y}}_h(t_0 + \tau h) = \sum_{j=1}^s \mathbf{k}_j L_j(\tau)$ with "coefficients" $\mathbf{k}_j := f(t_0 + c_j h, \mathbf{y}_h(t_0 + c_j h))$.

Next we integrate and use $\mathbf{y}_h(t_0) = \mathbf{y}_0$

◤ $\mathbf{y}_h(t_0 + \tau h) = \mathbf{y}_0 + h \sum_{j=1}^s \mathbf{k}_j \int_0^\tau L_j(\zeta) \, \mathrm{d}\zeta$.

This yields the following formulas for the computation of $\mathbf{y}_1$, which characterize the s-stage collocation single step method induced by the (normalized) collocation points $c_j \in [0,1]$, $j = 1, \ldots, s$.

$$\begin{aligned} \mathbf{k}_i &= f(t_0 + c_i h, \mathbf{y}_0 + h \sum_{j=1}^s a_{ij} \mathbf{k}_j) \, , \\ \mathbf{y}_1 := \mathbf{y}_h(t_1) &= \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i \, . \end{aligned} \quad \text{where} \quad \begin{aligned} a_{ij} &:= \int_0^{c_i} L_j(\tau) \, \mathrm{d}\tau \, , \\ b_i &:= \int_0^1 L_i(\tau) \, \mathrm{d}\tau \, . \end{aligned} \tag{12.3.11}$$

Note that, since arbitrary $\mathbf{y}_0 \in D$, $t_0, t_1 \in I$ were admitted, this defines a discrete evolution $\boldsymbol{\Psi} : I \times I \times D \to \mathbb{R}^d$ by $\boldsymbol{\Psi}^{t_0,t_1}\mathbf{y}_0 := \mathbf{y}_h(t_1)$.

Remark 12.3.12 (Implicit nature of collocation single step methods)

Note that (12.3.11) represents a generically non-linear system of $s \cdot d$ equations for the $s \cdot d$ components of the vectors $\mathbf{k}_i$, $i = 1, \ldots, s$. Usually, it will not be possible to obtain $\mathbf{k}_i$ by a fixed number of evaluations of $\mathbf{f}$. For this reason the single step methods defined by (12.3.11) are called implicit.

With similar arguments as in Rem. 11.2.14 one can prove that for sufficiently small $|t_1 - t_0|$ a unique solution for $\mathbf{k}_1, \ldots, \mathbf{k}_s$ can be found.

(12.3.13) Collocation single step methods and quadrature

Clearly, in the case $d = 1$, $f(t, \mathbf{y}) = f(t)$, $y_0 = 0$ the computation of y_1 boils down to the evaluation of a quadrature formula on $[t_0, t_1]$, because from (12.3.11) we get

$$y_1 = h \sum_{i=1}^{s} b_i f(t_0 + c_i h) , \quad b_i := \int_0^1 L_i(\tau) \, \mathrm{d}\tau , \tag{12.3.14}$$

which is a polynomial quadrature formula (7.2.2) on $[0, 1]$ with nodes c_j transformed to $[t_0, t_1]$ according to (7.1.5).

Experiment 12.3.15 (Empiric Convergence of collocation single step methods)

We consider the scalar logistic ODE (11.1.6) with parameter $\lambda = 10$ ($\to$ only mildly stiff), initial state $y_0 = 0.01$, $T = 1$.

Numerical integration by timestepping with uniform timestep h based on collocation single step method (12.3.11).

❶ Equidistant collocation points, $c_j = \frac{j}{s+1}$, $j = 1, \cdots, s$.

We observe algebraic convergence with the empiric rates

$s = 1$: $p = 1.96$
$s = 2$: $p = 2.03$
$s = 3$: $p = 4.00$
$s = 4$: $p = 4.04$

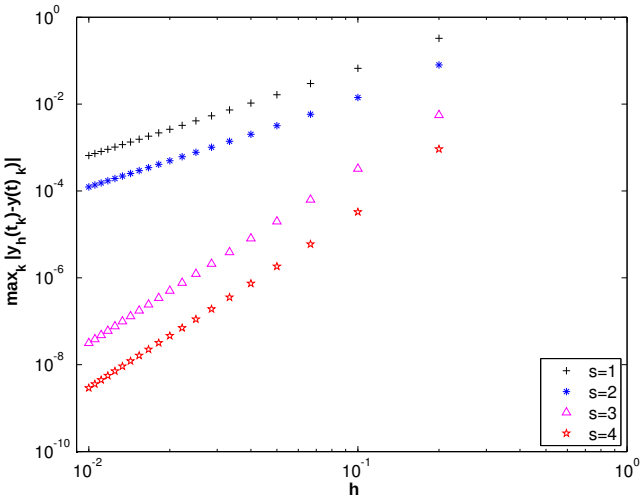


Fig. 447

In this case we conclude the following (empiric) order ($\to$ Def. 11.3.21) of the collocation single step

method:

$$\text{(empiric) order} \quad = \begin{cases} s & \text{for even } s\,, \\ s+1 & \text{for odd } s\,. \end{cases}$$

❶ Gauss points in $[0,1]$ as normalized collocation points c_j , $j = 1,\ldots,s$.

We observe algebraic convergence with the empiric rates

$s = 1$: $p = 1.96$
$s = 2$: $p = 4.01$
$s = 3$: $p = 6.00$
$s = 4$: $p = 8.02$

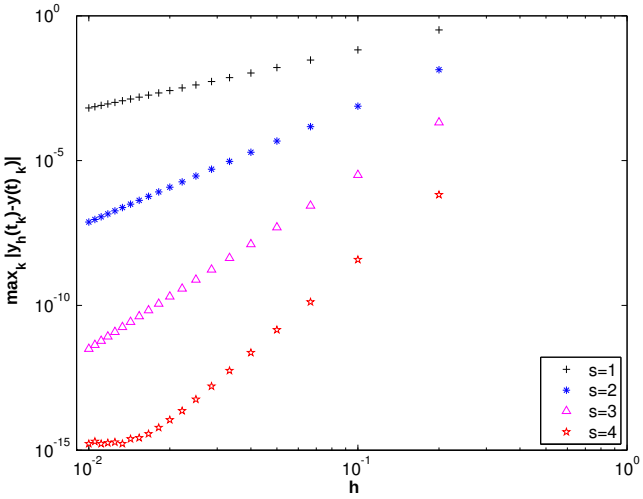


Fig. 448

Obviously, for the (empiric) order ($\to$ Def. 11.3.21) of the Gauss collocation single step method holds

$$\text{(empiric) order} \quad = 2s\,.$$

Note that the 1-stage Gauss collocation single step method is the implicit midpoint method from Section 11.2.3.

(12.3.16) Order of collocation single step method

What we have observed in Exp. 12.3.15 reflects a fundamental result on collocation single step methods as defined in (12.3.11).

Theorem 12.3.17. Order of collocation single step method [?, Satz .6.40]

Provided that $\mathbf{f} \in C^p(I \times D)$, the order ($\to$ Def. 11.3.21) of an s-stage collocation single step method according to (12.3.11) *agrees with the order ($\to$ Def. 7.3.1) of the quadrature formula on $[0,1]$ with nodes c_j and weights b_j, $j = 1,\ldots,s$.*

➢ By Thm. 7.3.22 the s-stage Gauss collocation single step method whose nodes c_j are chosen as the s Gauss points on $[0,1]$ is of order $2s$.

12.3.3 General implicit RK-SSMs

The notations in (12.3.11) have deliberately been chosen to allude to Def. 11.4.9. In that definition it takes only letting the sum in the formula for the increments run up to s to capture (12.3.11).

Definition 12.3.18. General Runge-Kutta single step method (*cf.* Def. 11.4.9)

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{s} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, an s-stage Runge-Kutta single step method (RK-SSM) for the IVP (11.1.20) is defined by

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{s} a_{ij}\mathbf{k}_j) \;, \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i \;.$$

As before, the $\mathbf{k}_i \in \mathbb{R}^d$ are called increments.

Note: computation of increments $\mathbf{k}_i$ may now require the solution of *(non-linear) systems of equations* of size $s \cdot d$ ($\rightarrow$ "implicit" method, *cf.* Rem. 12.3.12)

General Butcher scheme notation for RK-SSM

Shorthand notation for Runge-Kutta methods

Butcher scheme $\triangleright$ $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$. (12.3.20)

Note: now $\mathfrak{A}$ can be a general $s \times s$-matrix.

Summary: terminology for Runge-Kutta single step methods:

$\mathfrak{A}$ strict lower triangular matrix	➤	explicit Runge-Kutta method, Def. 11.4.9
$\mathfrak{A}$ lower triangular matrix	➤	diagonally-implicit Runge-Kutta method (DIRK)

Many of the techniques and much of the theory discussed for explicit RK-SSMs carry over to general (implicit) Runge-Kutta single step methods:

- Sufficient condition for consistence from Cor. 11.4.12
- Algebraic convergence for meshwidth $h \to 0$ and the related concept of order ($\rightarrow$ Def. 11.3.21)
- Embedded methods and algorithms for adaptive stepsize control from Section 11.5

Remark 12.3.21 (Stage form equations for increments)

In Def. 12.3.18 instead of the increments we can consider the stages

$$\mathbf{g}_i := h\sum_{j=1}^{s} a_{ij}\mathbf{k}_j \;, \quad i = 1,\ldots,s \;, \quad \Leftrightarrow \quad \mathbf{k}_i = \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + \mathbf{g}_i) \;. \tag{12.3.22}$$

This leads to the equivalent defining equations in "stage form" for an implicit RK-SSM

$$\mathbf{g}_i = h\sum_{j=1}^{s} a_{ij}\mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + \mathbf{g}_j) \quad , \quad \mathbf{y}_1 = \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{f}(t_0 + c_j h, \mathbf{y}_0 + \mathbf{g}_i) \;. \tag{12.3.23}$$

In terms of implementation there is no difference.

Remark 12.3.24 (Solving the increment equations for implicit RK-SSMs)

We reformulate the increment equations in stage form (12.3.23) as a non-linear system of equations in standard form $F(\mathbf{x}) = 0$. Unknowns are the total $s \cdot d$ components of the stage vectors $\mathbf{g}_i$, $i = 1, \ldots, s$ as defined in (12.3.22).

$$\begin{aligned} \mathfrak{g} &= [\mathbf{g}_1, \ldots, \mathbf{g}_s]^\top , \\ \mathbf{g}_i &:= h \sum_{j=1}^{s} a_{ij} \mathbf{f}(t_0 + c_j h, \mathbf{y}_0 + \mathbf{g}_j) \end{aligned} \quad \blacktriangleright \quad F(\mathfrak{g}) = \mathfrak{g} - h(\mathfrak{A} \otimes \mathbf{I}) \begin{pmatrix} \mathbf{f}(t_0 + c_1 h, \mathbf{y}_0 + \mathbf{g}_1) \\ \vdots \\ \mathbf{f}(t_0 + c_s h, \mathbf{y}_0 + \mathbf{g}_s) \end{pmatrix} \stackrel{!}{=} \mathbf{0} ,$$

where $\mathbf{I}$ is the $d \times d$ identity matrix and $\otimes$ designates the Kronecker product introduced in Def. 1.4.17.

We compute an approximate solution of $F(\mathfrak{g}) = \mathbf{0}$ iteratively by means of the simplified Newton method presented in Rem. 8.4.39. This is a Newton method with "frozen Jacobian". As $\mathfrak{g} \to \mathbf{0}$ for $h \to 0$, we choose zero as initial guess:

$$\mathfrak{g}^{(k+1)} = \mathfrak{g}^{(k)} - \mathrm{D}\, F(\mathbf{0})^{-1} F(\mathfrak{g}^{(k)}) \quad k = 0, 1, 2, \ldots \quad , \quad \mathfrak{g}^{(0)} = \mathbf{0} . \tag{12.3.25}$$

with the Jacobian

$$\mathrm{D}\, F(\mathbf{0}) = \begin{bmatrix} \mathbf{I} - h a_{11} \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) & \cdots & -h a_{1s} \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) \\ \vdots & \ddots & \vdots \\ -h a_{s1} \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) & \cdots & \mathbf{I} - h a_{ss} \frac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0, \mathbf{y}_0) \end{bmatrix} \in \mathbb{R}^{sd,sd} . \tag{12.3.26}$$

Obviously, $\mathrm{D}\, F(\mathbf{0}) \to \mathbf{I}$ for $h \to 0$. Thus, $\mathrm{D}\, F(\mathbf{0})$ will be regular for sufficiently small h.

In each step of the simplified Newton method we have to solve a linear system of equations with coefficient matrix $\mathrm{D}\, F(\mathbf{0})$. If $s \cdot d$ is large, an efficient implementation has to reuse the LU-decomposition of $\mathrm{D}\, F(\mathbf{0})$, see Code 8.4.40 and Rem. 2.5.10.

12.3.4 Model problem analysis for implicit RK-SSMs

Model problem analysis for general Runge-Kutta single step methods ($\to$ Def. 12.3.18) runs parallel to that for explicit RK-methods as elaborated in Section 12.1, § 12.1.13. Familiarity with the techniques and results of this section is assumed. The reader is asked to recall the concept of stability function from Thm. 12.1.17, the diagonalization technique from § 12.1.44, and the definition of region of (absolute) stability from Def. 12.1.51.

Theorem 12.3.27. Stability function of Runge-Kutta methods, *cf.* Thm. 12.1.17

[Stability function of general Runge-Kutta methods]
The discrete evolution Ψ_λ^h of an s-stage Runge-Kutta single step method ($\rightarrow$ Def. 12.3.18) with Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ *(see* (12.3.20)*) for the ODE $\dot{y} = \lambda y$ is given by a multiplication with*

$$S(z) := \underbrace{1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1}}_{\text{stability function}} = \frac{\det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T)}{\det(\mathbf{I} - z\mathfrak{A})}\,, \quad z := \lambda h\,, \quad \mathbf{1} = [1,\ldots,1]^T \in \mathbb{R}^s\,.$$

Example 12.3.28 (Regions of stability for simple implicit RK-SSM)

We determine the Butcher schemes (12.3.20) for simple implicit RK-SSM and apply the formula from Thm. 12.3.27 to compute their stability functions.

- Implicit Euler method: $\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$ ➢ $S(z) = \dfrac{1}{1-z}\,.$
- Implicit midpoint method: $\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array}$ ➢ $S(z) = \dfrac{1+\frac{1}{2}z}{1-\frac{1}{2}z}\,.$

Their regions of stability $\mathcal{S}_\Psi$ as defined in Def. 12.1.51 can easily found from the respective stability functions:

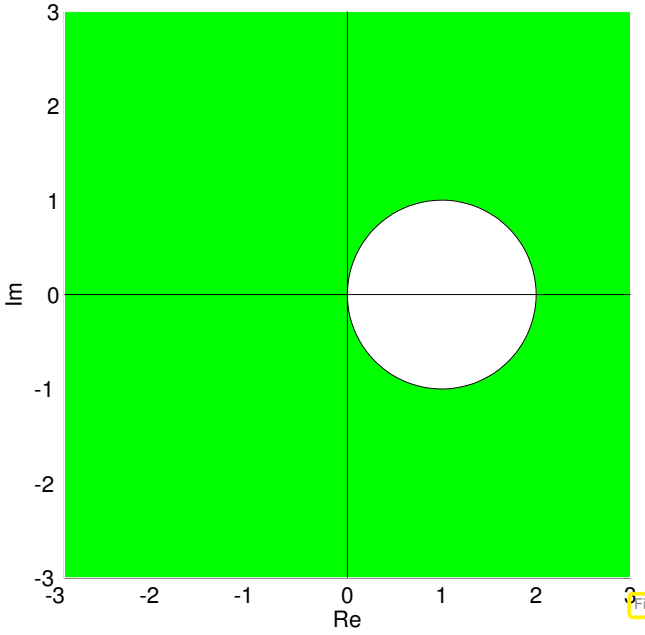


Fig. 449

$\mathcal{S}_\Psi$: implicit Euler method (11.2.13)

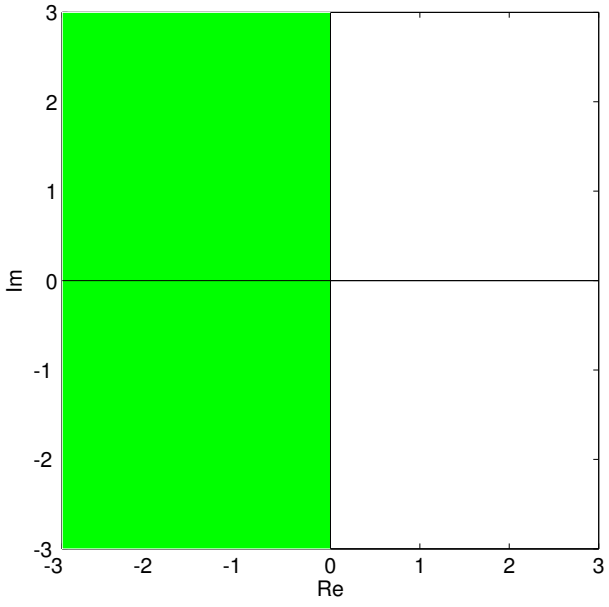


Fig. 450

$\mathcal{S}_\Psi$: implicit midpoint method (11.2.18)

We see that in both cases $|S(z)| < 1$, if $\operatorname{Re} z < 0$.

From the determinant formula for the stability function $S(z)$ we can conclude a generalization of Cor. 12.1.20.

Corollary 12.3.29. Rational stability function of explicit RK-SSM

For a consistent (→ Def. 11.3.10) s-stage general Runge-Kutta single step method according to Def. 12.3.18 the stability function S is a non-constant rational function of the form $S(z) = \frac{P(z)}{Q(z)}$ with polynomials $P \in \mathcal{P}_s$, $Q \in \mathcal{P}_s$.

Of course, a rational function $z \mapsto S(z)$ can satisfy $\lim_{|z|\to\infty} |S(z)| < 1$ as we habe seen in Ex. 12.3.28. As a consequence, the region of stability for implicit RK-SSM need not be bounded.

(12.3.30) A-stability

A general RK-SSM with stability function S applied to the scalar linear IVP $\dot{y} = \lambda y$, $y(0) = y_0 \in \mathbb{C}$, $\lambda \in \mathbb{C}$, with uniform timestep $h > 0$ will yield the sequence $(y_k)_{k=0}^{\infty}$ defined by

$$y_k = S(z)^k y_0 \quad , \quad z = \lambda h \, . \tag{12.3.31}$$

Hence, the next property of a RK-SSM guarantees that the sequence of approximations decays exponentially whenever the exact solution of the model problem IVP (12.1.5) does so.

Definition 12.3.32. A-stability of a Runge-Kutta single step method

A Runge-Kutta single step method with stability function S is A-stable, if

$$\mathbb{C}^- := \{z \in \mathbb{C} : \operatorname{Re} z < 0\} \subset \mathcal{S}_\Psi \, . \quad (\mathcal{S}_\Psi \hat{=} \text{ region of stability Def. 12.1.51})$$

From Ex. 12.3.28 we conclude that both the implicit Euler method and the implicit midpoint method are A-stable.

A-stable Runge-Kutta single step methods will not be affected by stability induced timestep constraints when applied to stiff IVP (→ Notion 12.2.9).

(12.3.33) "Ideal" region of stability

In order to reproduce the qualitative behavior of the exact solution, a single step method when applied to the scalar linear IVP $\dot{y} = \lambda y$, $y(0) = y_0 \in \mathbb{C}$, $\lambda \in \mathbb{C}$, with uniform timestep $h > 0$,

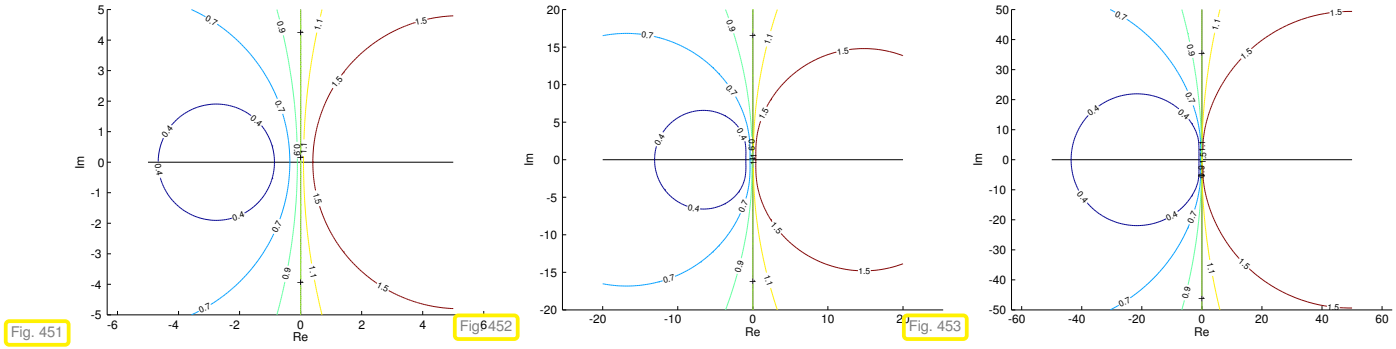
- should yield an *exponentially decaying* sequence $(y_k)_{k=0}^{\infty}$, whenever $\operatorname{Re}\lambda < 0$,
- should produce an *exponentially increasing* sequence sequence $(y_k)_{k=0}^{\infty}$, whenever $\operatorname{Re}\lambda > 0$.

Thus, in light of (12.3.31), we agree that the stability if

$$\text{"ideal" region of stability is} \quad \boxed{\mathcal{S}_\Psi = \mathbb{C}^-} \, . \tag{12.3.34}$$

Are there RK-SSMs that can boast of an ideal region of stability?

Regions of stability of Gauss collocation single step methods, see Exp. 12.3.15:



Implicit midpoint method $s = 2$ (order 4) $s = 4$ (order 8)

Level lines for $|S(z)|$ for Gauss collocation methods

Theorem 12.3.35. Region of stability of Gauss collocation single step methods [?, Satz 6.44]

s-stage Gauss collocation single step methods defined by (12.3.11) *with the nodes c_s given by the s Gauss points on $[0,1]$, feature the "ideal" stability domain:*

$$\mathcal{S}_\Psi = \mathbb{C}^- \,. \tag{12.3.34}$$

In particular, all Gauss collocation single step methods are A-stable.

Experiment 12.3.36 (Implicit RK-SSMs for stiff IVP)

We consider the stiff IVP

$$\dot{y} = -\lambda y + \beta \sin(2\pi t) \,, \quad \lambda = 10^6, \beta = 10^6 \,, \quad y(0) = 1 \,,$$

whose solution essentially is the smooth function $t \mapsto \sin(2\pi t)$. Applying the criteria (12.2.15) and (12.2.16) we immediately see that this IVP is extremely stiff.

We solve it with different implicit RK-SSM on $[0,1]$ with large uniform timestep $h = \frac{1}{20}$.

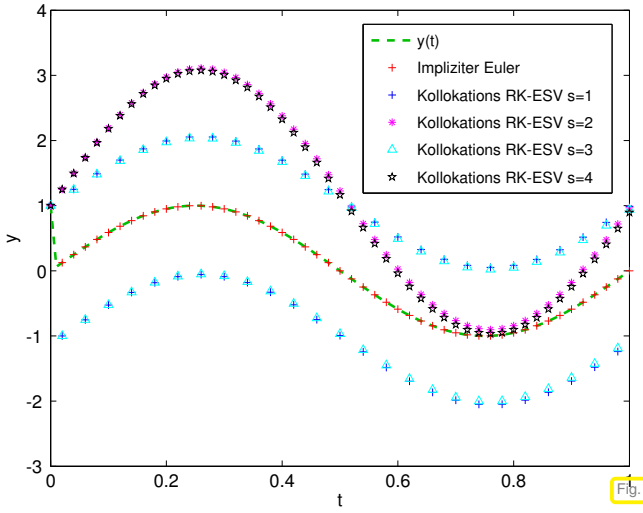


Fig. 454

Solutions by RK-SSMs

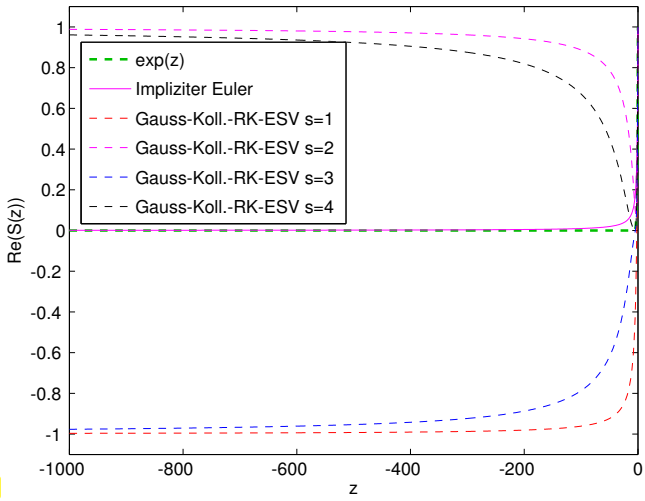


Fig. 455

Stability functions on $\mathbb{R}^-$

We observe that Gauss collocation RK-SSMs incur a huge discretization error, whereas the simple implicit Euler method provides a perfect approximation!

Explanation: The stability functions for Gauss collocation RK-SSMs satisfy

$$\lim_{|z|\to\infty} |S(z)| = 1 .$$

Hence, when they are applied to $\dot{y} = \lambda y$ with extremely large (in modulus) $\lambda < 0$, they will produce sequences that decay only very slowly or even oscillate, which misses the very rapid decay of the exact solution. The stability function for the implicity Euler method is $S(z) = (1-z)^{-1}$ and satisfies $\lim_{|z|\to\infty} S(z) = 0$, which will mean a fast exponential decay of the y_k.

(12.3.37) L-stability

In light of what we learned in the previous experiment we can now state what we expect from the stability function of a Runge-Kutta method that is suitable for stiff IVP (→ Notion 12.2.9):

Definition 12.3.38. L-stable Runge-Kutta method → [?, Ch. 77]

A Runge-Kutta method (→ Def. 12.3.18) is L-stable/asymptotically stable, if its stability function (→ Thm. 12.3.27) satisfies

$$(i) \qquad \operatorname{Re} z < 0 \;\Rightarrow\; |S(z)| < 1 , \tag{12.3.39}$$

$$(ii) \qquad \lim_{\operatorname{Re} z\to-\infty} S(z) = 0 . \tag{12.3.40}$$

Remember: L-stable :⇔ A-stable & "$S(-\infty) = 0$''

Remark 12.3.41 (Necessary condition for L-stability of Runge-Kutta methods)

Consider a Runge-Kutta single step method (→ Def. 12.3.18) described by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathcal{A} \\ \hline & \mathbf{b}^T \end{array}$.

Assume that $\mathfrak{A} \in \mathbb{R}^{s,s}$ is regular, which can be fulfilled only for an implicit RK-SSM.

For a rational function $S(z) = \frac{P(z)}{Q(z)}$ the limit for $|z| \to \infty$ exists and can easily be expressed by the leading coefficients of the polynomials P and Q:

$$\text{Thm. 12.3.27} \quad \Rightarrow \quad S(-\infty) = 1 - \mathbf{b}^T \mathfrak{A}^{-1} \mathbf{1} \,. \tag{12.3.42}$$

$$\blacktriangleright \quad \text{If } \mathbf{b}^T = (\mathfrak{A})^T_{:,j} \text{ (row of } \mathfrak{A}) \quad \Rightarrow \quad S(-\infty) = 0 \,. \tag{12.3.43}$$

Butcher scheme (12.3.20) for L-stable RK-methods, see Def. 12.3.38

$$\triangleright \quad \begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_{s-1} & a_{s-1,1} & \cdots & a_{s-1,s} \\ 1 & b_1 & \cdots & b_s \\ \hline & b_1 & \cdots & b_s \end{array} .$$

A closer look at the coefficient formulas of (12.3.11) reveals that the algebraic condition (12.3.43) will automatically satisfied for a collocation single step method with $c_s = 1$!

Example 12.3.44 (L-stable implicit Runge-Kutta methods)

There is a family of s-point quadrature formulas on $[0,1]$ with a node located in 1 and (maximal) order $2s-1$: Gauss-Radau formulas. They induce the L-stable Gauss-Radau collocation single step methods of order $2s-1$ according to Thm. 12.3.17.

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array}$$

Implicit Euler method

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}$$

Radau RK-SSM, order 3

$$\begin{array}{c|ccc} \frac{4-\sqrt{6}}{10} & \frac{88-7\sqrt{6}}{360} & \frac{296-169\sqrt{6}}{1800} & \frac{-2+3\sqrt{6}}{225} \\ \frac{4+\sqrt{6}}{10} & \frac{296+169\sqrt{6}}{1800} & \frac{88+7\sqrt{6}}{360} & \frac{-2-3\sqrt{6}}{225} \\ 1 & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \\ \hline & \frac{16-\sqrt{6}}{36} & \frac{16+\sqrt{6}}{36} & \frac{1}{9} \end{array}$$

Radau RK-SSM, order 5

The stability functions of s-stage Gauss-Radau collocation SSMs are rational functions of the form

$$S(z) = \frac{P(z)}{Q(z)} \,, \quad P \in \mathcal{P}_{s-1}, Q \in \mathcal{P}_s \,.$$

Beware that also "$S(\infty) = 0$", which means that Gauss-Radau methods when applied to problems with fast exponential blow-up may produce a spurious decaying solution.

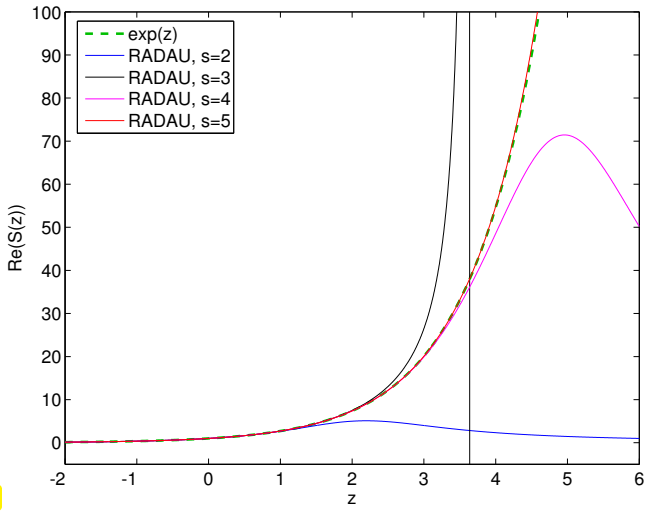


Fig. 456

Level lines of stability functions of s-stage Gauss-Radau collocation SSMs:

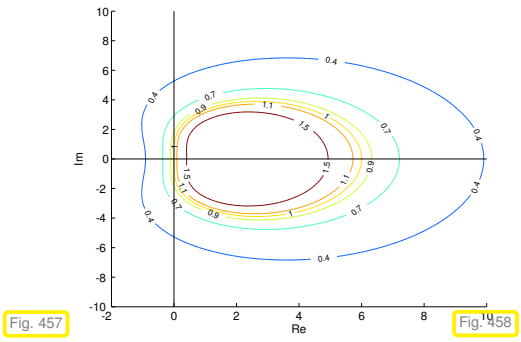


Fig. 457

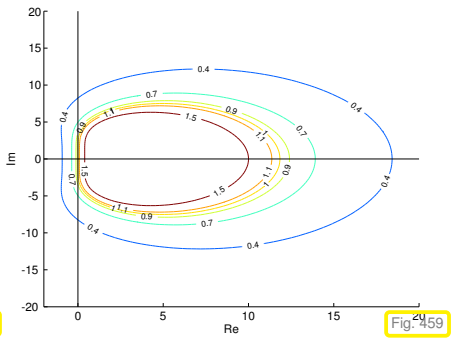


Fig. 458

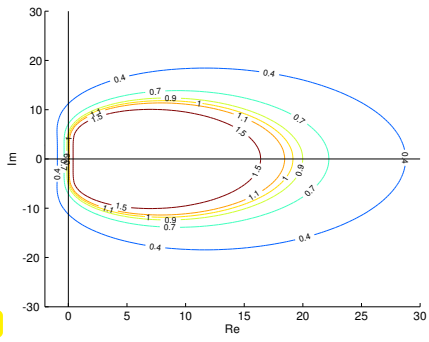


Fig. 459

$s = 2$ $s = 3$ $s = 4$

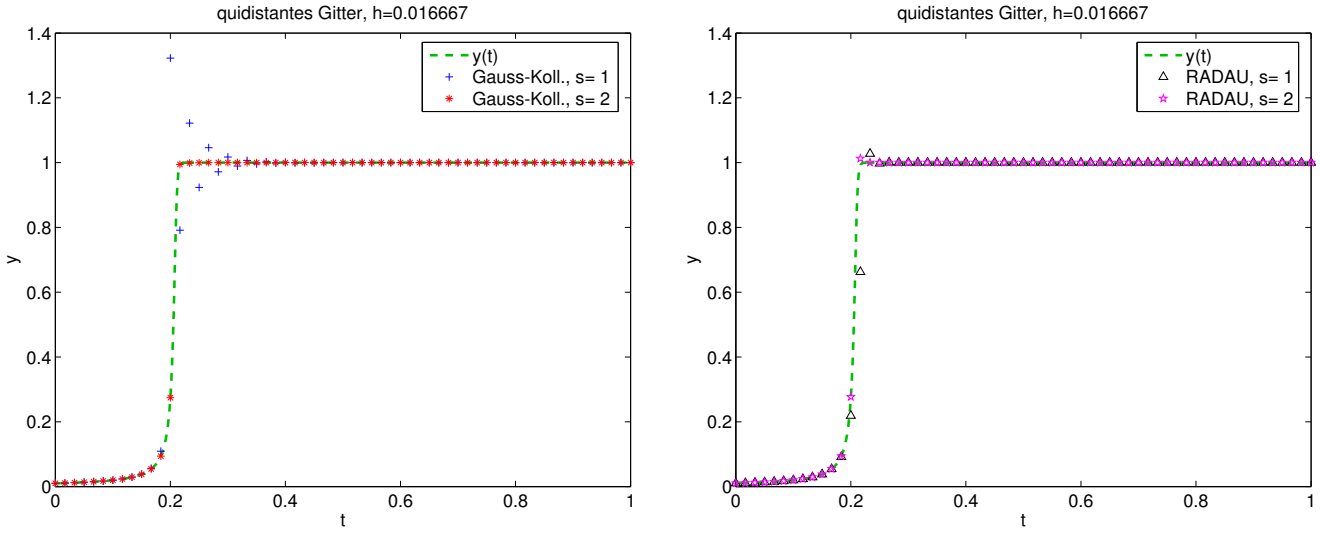
Further information about Radau-Runge-Kutta single step methods can be found in [**?**, Ch. 79].

Experiment 12.3.45 (Gauss-Radau collocation SSM for stiff IVP)

We revisit the stiff IVP from Ex. 12.0.1

$$\dot{y}(t) = \lambda y^2(1-y) \, , \quad \lambda = 500 \quad , \quad y(0) = \tfrac{1}{100} \, .$$

We compare the sequences generated by 1-stage and 2-stage Gauss collocation and Gauss-Radau collocation SSMs, respectively (uniform timestep).



The 2nd-order Gauss collocation SSM (implicit midpoint method) suffers from spurious oscillations we homing in on the stable stationary state $y = 1$. The explanation from Exp. 12.3.36 also applies to this example.

The fourth-order Gauss method is already so accurate that potential overshoots when approaching $y = 1$ are damped fast enough.

12.4 Semi-implicit Runge-Kutta Methods

Supplementary reading. [**?**, Ch. 80]

The equations fixing the increments $\mathbf{k}_i \in \mathbb{R}^d$, $i = 1, \ldots, s$, for an s-stage implicit RK-method constitute a (Non-)linear system of equations with $s \cdot d$ unknowns.

Expensive iterations needed to find $\mathbf{k}_i$ **?**

Remember that we compute approximate solutions anyway, and the increments *are weighted with the stepsize* $h \ll 1$, see Def. 12.3.18. So there is no point in determining them with high accuracy!

Idea: Use only a fixed *small* number of Newton steps to solve for the $\mathbf{k}_i$, $i = 1, \ldots, s$.

Extreme case: use only a single Newton step! Let's try.

Example 12.4.1 (Linearization of increment equations)

- We consider an Initial value problem for logistic ODE, see Ex. 11.1.5

$$\dot{y} = \lambda y(1-y) \quad , \quad y(0) = 0.1 \quad , \quad \lambda = 5 \, .$$

- We use the implicit Euler method (11.2.13) with uniform timestep $h = 1/n$, $n \in \{5, 8, 11, 17, 25, 38, 57, 85, 128, 192, 288, , 432, 649, 973, 1460, 2189, 3284, 4926, 7389\}$.

  & approximate computation of y_{k+1} by **1 Newton step** with initial guess y_k

  = semi-implicit Euler method

- Measured error $\mathrm{err} = \max\limits_{j=1,\ldots,n} |y_j - y(t_j)|$

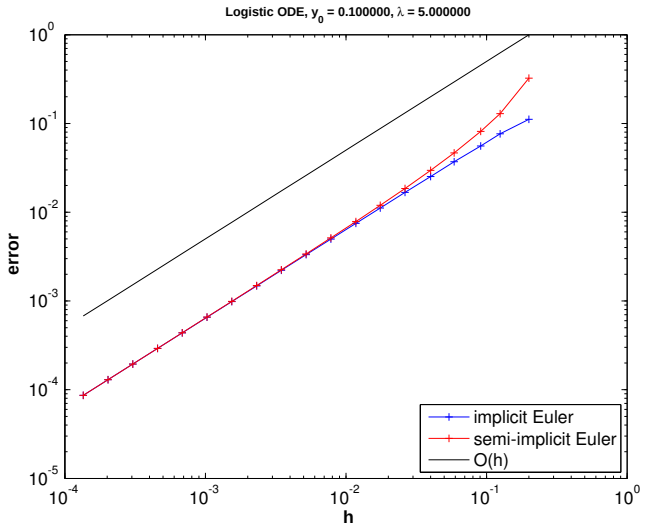


Fig. 460

From (11.2.13) with timestep $h > 0$

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}(\mathbf{y}_{k+1}) \quad \Leftrightarrow \quad F(\mathbf{y}_{k+1}) := \mathbf{y}_{k+1} - h\mathbf{f}(\mathbf{y}_{k+1}) - \mathbf{y}_k = 0 \, .$$

One Newton step (8.4.1) applied to $F(\mathbf{y}) = 0$ with initial guess $\mathbf{y}_k$ yields

$$\mathbf{y}_{k+1} = \mathbf{y}_k - \mathrm{D}\,\mathbf{f}(\mathbf{y}_k)^{-1} F(\mathbf{y}_k) = \mathbf{y}_k + (\mathbf{I} - hD\mathbf{f}(\mathbf{y}_k))^{-1} h\mathbf{f}(\mathbf{y}_k) \, .$$

Note: for linear ODE with $\mathbf{f}(\mathbf{y}) = \mathbf{A}\mathbf{y}$, $\mathbf{A} \in \mathbb{R}^{d,d}$, we recover the original implicit Euler method!

Observation: Approximate evaluation of defining equation for y_{k+1} preserves 1st order convergence.

- Now, implicit midpoint method (11.2.18), uniform timestep $h = 1/n$ as above

  & approximate computation of y_{k+1} by **1 Newton step**, initial guess y_k

- Fehlermass $\mathrm{err} = \max_{j=1,\ldots,n} |y_j - y(t_j)|$

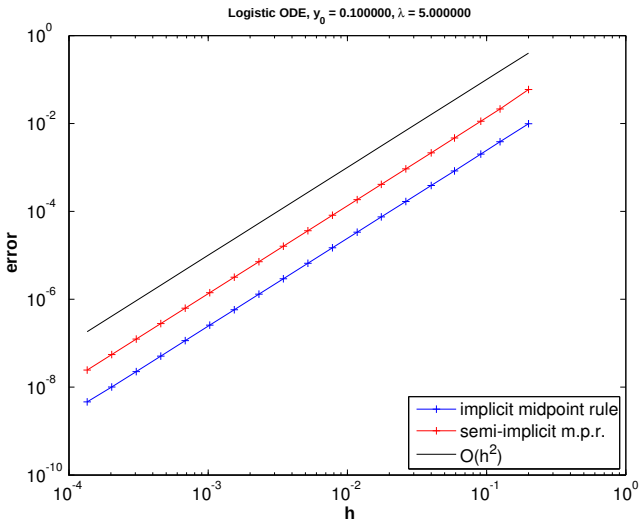
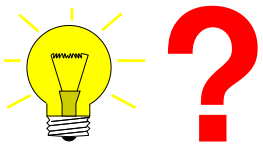


Fig. 461

We still observe second-order convergence!

Try: Use linearized increment equations for implicit RK-SSM



$$\mathbf{k}_i := \mathbf{f}(\mathbf{y}_0 + h\sum_{j=1}^{s} a_{ij}\mathbf{k}_j)\ , \quad i = 1,\ldots,s$$

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0) + h\,\mathrm{D}\,\mathbf{f}(\mathbf{y}_0)\left(\sum_{j=1}^{s} a_{ij}\mathbf{k}_j\right)\ , \quad i = 1,\ldots,s\ . \tag{12.4.2}$$

The good news is that all results about stability derived from model problem analysis (→ Section 12.1) remain valid despite linearization of the increment equations:

Linearization does nothing for linear ODEs ➢ stability function (→ Thm. 12.3.27) not affected!

The bad news is that the preservation of the order observed in Ex. 12.4.1 will no longer hold in the general case.

Example 12.4.3 (Convergence of naive semi-implicit Radau method)

- We consider an IVP for the logistic ODE from Ex. 11.1.5:

$$\dot{y} = \lambda y(1-y) \quad , \quad y(0) = 0.1 \quad , \quad \lambda = 5\,.$$

- 2-stage Radau RK-SSM, Butcher scheme

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} , \qquad (12.4.4)$$

order = 3, see Ex. 12.3.44.

- Increments from linearized equations (12.4.2)
- We monitor the error through $\mathrm{err} = \max_{j=1,\ldots,n} |y_j - y(t_j)|$

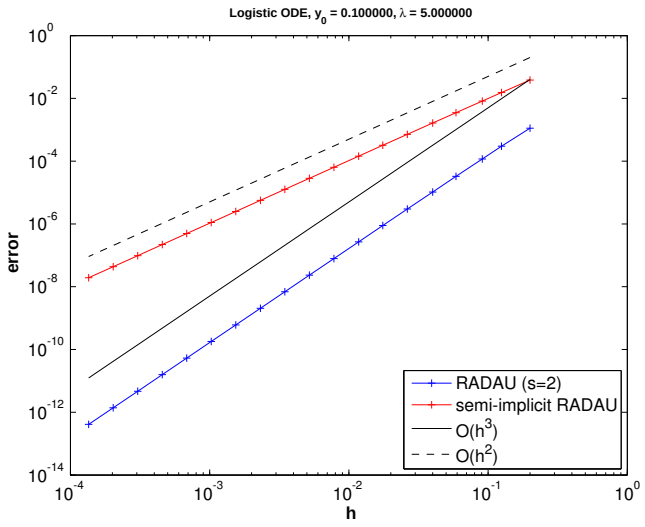


Fig. 462

Loss of order due to linearization !

(12.4.5) Rosenbrock-Wanner methods

We have just seen that the simple linearization according to (12.4.2) will degrade the order of implicit RK-SSMs and leads to a substantial loss of accuracy. This is not an option.

Yet, the idea behind (12.4.2) has been refined. One does not start from a known RK-SSM, but introduces general coefficients for structurally linear increment equations.

Class of s-stage semi-implicit (linearly implicit) Runge-Kutta methods (Rosenbrock-Wanner (ROW) methods):

$$\begin{aligned} (\mathbf{I} - ha_{ii}\mathbf{J})\mathbf{k}_i &= \mathbf{f}(\mathbf{y}_0 + h\sum_{j=1}^{i-1}(a_{ij}+d_{ij})\mathbf{k}_j) - h\mathbf{J}\sum_{j=1}^{i-1} d_{ij}\mathbf{k}_j \,, \quad \mathbf{J} = \mathrm{D}\,\mathbf{f}(\mathbf{y}_0) \,, \\ \mathbf{y}_1 &:= \mathbf{y}_0 + \sum_{j=1}^{s} b_j \mathbf{k}_j \,. \end{aligned} \qquad (12.4.6)$$

Then the coefficients a_{ij}, d_{ij}, and b_i are determined from order conditions by solving large non-linear systems of equations.

In each step s linear systems with coefficient matrices $\mathbf{I} - ha_{ii}\mathbf{J}$ have to be solved. For methods used in practice one often demands that $a_{ii} = \gamma$ for all $i = 1,\ldots,s$. As a consequence, we have to solve s linear systems with the *same* coefficient matrix $\mathbf{I} - h\gamma\mathbf{J} \in \mathbb{R}^{d,d}$, which permits us to reuse LU-factorizations, see Rem. 2.5.10.

Remark 12.4.7 (Adaptive integrator for stiff problems in MATLAB)

A ROW method is the basis for the standard integrator that MATLAB offers for stiff problems:

Handle of type `@(t,y) J(t,y)` to Jacobian $D\mathbf{f} : I \times D \mapsto \mathbb{R}^{d,d}$

```
opts = odeset('abstol',atol,'reltol',rtol,'Jacobian',J)
[t,y] = ode23s(odefun,tspan,y0,opts);
```

Stepsize control according to policy of Section 11.5:

$\Psi \widehat{=}$ RK-method of order 2 ⟷ ode23s ⟷ $\widetilde{\Psi} \widehat{=}$ RK-method of order 3

integrator for **s**tiff IVP

12.5 Splitting methods

(12.5.1) Splitting idea: composition of partial evolutions

Many relevant ordinary differential equations feature a right hand side function that is the sum to two (or more) terms. Consider an autonomous IVP with a right hand side function that can be split in an additive fashion:

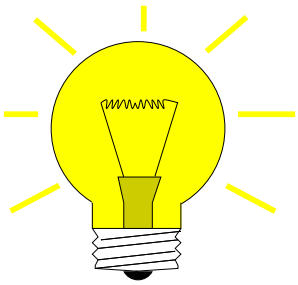
$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) + \mathbf{g}(\mathbf{y}) \quad , \quad \mathbf{y}(0) = \mathbf{y}_0 \, , \tag{12.5.2}$$

with $\mathbf{f} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$, $\mathbf{g} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$ "sufficiently smooth", locally Lipschitz continuous (→ Def. 11.1.28).

Let us introduce the evolution operators (→ Def. 11.1.39) for both summands:

$$\text{(Continuous) evolution maps:} \quad \begin{aligned} \boldsymbol{\Phi}_f^t &\leftrightarrow \text{ ODE } \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \, , \\ \boldsymbol{\Phi}_g^t &\leftrightarrow \text{ ODE } \dot{\mathbf{y}} = \mathbf{g}(\mathbf{y}) \, . \end{aligned}$$

Temporarily we assume that both $\boldsymbol{\Phi}_f^t$, $\boldsymbol{\Phi}_g^t$ are available in the form of analytic formulas or highly accurate approximations.



Idea: Build single step methods (→ Def. 11.3.5) based on the following discrete evolutions

$$\text{Lie-Trotter splitting:} \quad \boldsymbol{\Psi}^h = \boldsymbol{\Phi}_g^h \circ \boldsymbol{\Phi}_f^h \, , \tag{12.5.3}$$

$$\text{Strang splitting:} \quad \boldsymbol{\Psi}^h = \boldsymbol{\Phi}_f^{h/2} \circ \boldsymbol{\Phi}_g^h \circ \boldsymbol{\Phi}_f^{h/2} \, . \tag{12.5.4}$$

These splittings are easily remembered in graphical form:

(12.5.3) ↔

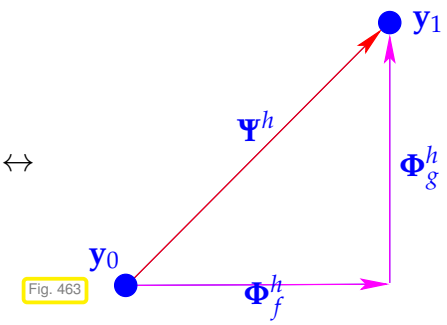


Fig. 463

(12.5.4) ↔

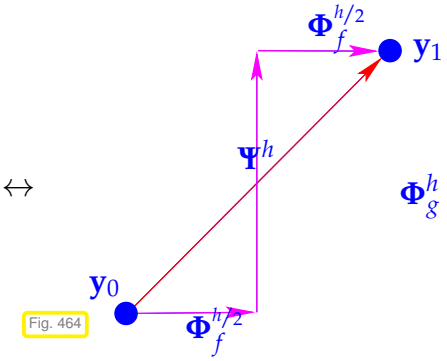


Fig. 464

Note that over many timesteps the Strang splitting approach is not more expensive than Lie-Trotter splitting, because the actual implementation of (12.5.4) should be done as follows:

$$\begin{aligned} \mathbf{y}_{1/2} &:= \boldsymbol{\Phi}_f^{h/2}\,, & \mathbf{y}_1 &:= \boldsymbol{\Phi}_g^{h}\mathbf{y}_{1/2}\,,\\ \mathbf{y}_{3/2} &:= \boldsymbol{\Phi}_f^{h}\mathbf{y}_{1}\,, & \mathbf{y}_2 &:= \boldsymbol{\Phi}_g^{h}\mathbf{y}_{3/2}\,,\\ \mathbf{y}_{5/2} &:= \boldsymbol{\Phi}_f^{h}\mathbf{y}_{2}\,, & \mathbf{y}_3 &:= \boldsymbol{\Phi}_g^{h}\mathbf{y}_{5/2}\,,\\ &\vdots & &\vdots\,, \end{aligned}$$

because $\boldsymbol{\Phi}_f^{h/2}\circ\boldsymbol{\Phi}_f^{h/2} = \boldsymbol{\Phi}_f^{h}$. This means that a Strang splitting SSM differs from a Lie-Trotter splitting SSM in the first and the last step only.

Example 12.5.5 (Convergence of simple splitting methods)

We consider the following IVP whose right hand side function is the sum of two functions for which the ODEs can be solved analytically:

$$\dot{y} = \underbrace{\lambda y(1-y)}_{=:f(y)} + \underbrace{\sqrt{1-y^2}}_{=:g(y)} \quad , \quad y(0) = 0\,.$$

- $\boldsymbol{\Phi}_f^t y = \dfrac{1}{1+(y^{-1}-1)e^{-\lambda t}}\,, t>0, y\in]0,1]$ (logistic ODE (11.1.6))
- $\boldsymbol{\Phi}_g^t y = \begin{cases} \sin(t+\arcsin(y)) & \text{, if } t+\arcsin(y) < \frac{\pi}{2}\,,\\ 1 & \text{, else,}\end{cases} \quad t>0, y\in[0,1]\,.$

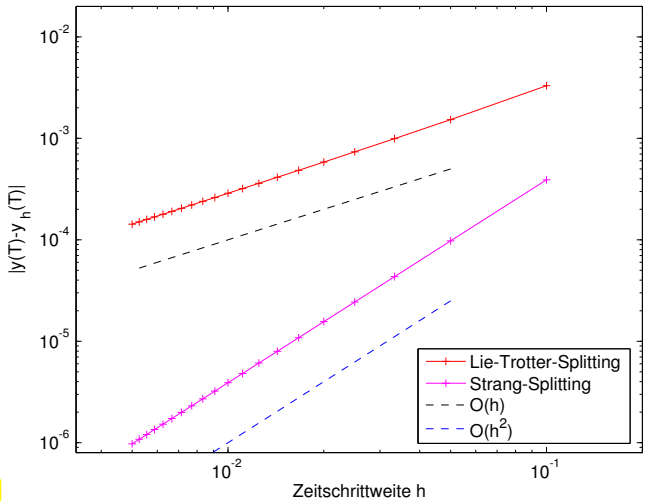


Fig. 465

Numerical experiment:

For $T = 1$, $\lambda = 1$, we compare the two splitting methods for uniform timesteps with a very accurate reference solution obtained by

```
f=@(t,x) λ*x*(1-x)+sqrt(1-x^2);
options=odeset('reltol',1.0e-10,...
               'abstol',1.0e-12);
[t,yex]=ode45(f,[0,1],y0,options);
```

◁ Error at final time $T = 1$

We observe algebraic convergence of the two splitting methods, order 1 for (12.5.3), oder 2 for (12.5.4).

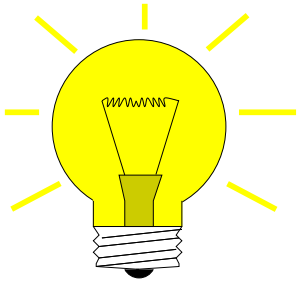
The observation made in Ex. 12.5.5 reflects a general truth:

Theorem 12.5.6. Order of simple splitting methods

Die single step methods defined by (12.5.3) *or* (12.5.4) *are of order (→ Def. 11.3.21)* 1 *and* 2, *respetively.*

(12.5.7) Inexact splitting methods

Of course, the assumption that $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ and $\dot{\mathbf{y}} = \mathbf{g}(\mathbf{y})$ can be solved exactly will hardly ever be met. However, it should be clear that a "sufficiently accurate" approximation of the evolution maps Φ_g^h and Φ_f^h is all we need



Idea: In (12.5.3)/(12.5.4) replace

$$\text{exact evolutions} \longrightarrow \text{discrete evolutions}$$
$$\Phi_g^h,\ \Phi_f^h \longrightarrow \Psi_g^h,\ \Psi_f^h \quad .$$

Example 12.5.8 (Convergence of inexact simple splitting methods)

Again we consider the IVP of Ex. 12.5.5 and inexact splitting methods based on different single step methods for the two ODE corresponding to the summands.

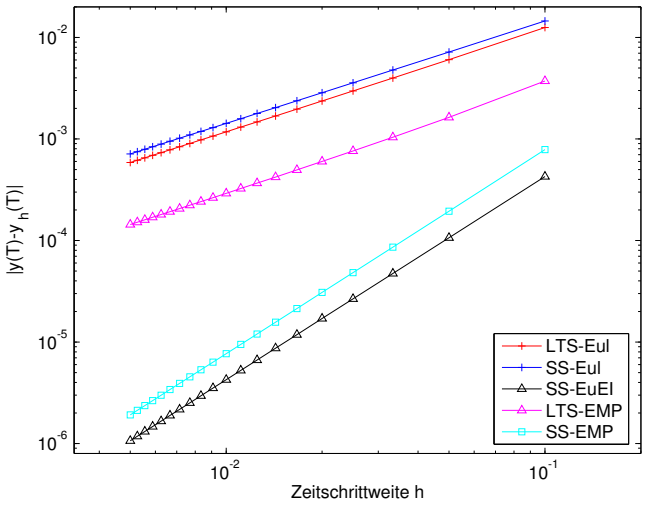


Fig. 466

LTS-Eul	explicit Euler method (11.2.7) $\rightarrow \Psi_{h,g}^h$, $\Psi_{h,f}^h$ + Lie-Trotter splitting (12.5.3)
SS-Eul	explicit Euler method (11.2.7) $\rightarrow \Psi_{h,g}^h$, $\Psi_{h,f}^h$ + Strang splitting (12.5.4)
SS-EuEI	Strang splitting (12.5.4): Explizites Euler method (11.2.7) $\circ$ exact evolution $\Phi_g^h \circ$ implicit Euler method (11.2.13)
LTS-EMP	explicit midpoint method (11.2.18) $\rightarrow \Psi_{h,g}^h$, $\Psi_{h,f}^h$ + Lie-Trotter splitting (12.5.3)
SS-EMP	explicit midpoint method (11.4.7) $\rightarrow \Psi_{h,g}^h$, $\Psi_{h,f}^h$ + Strang splitting (12.5.4)

☞ The order of splitting methods may be (but need not be) limited by the order of the SSMs used for Φ_f^h, Φ_g^h.

(12.5.9) Application of splitting methods

In the following situation the use splitting methods seems advisable:

"Splittable" ODEs

$\dot{\mathbf{y}} = f(\mathbf{y}) + g(\mathbf{y})$ "'difficult" (e.g., stiff $\rightarrow$ Section 12.2)	:	$\dot{\mathbf{y}} = f(\mathbf{y}) \rightarrow$ stiff, but with an analytic solution $\dot{\mathbf{y}} = g(\mathbf{y})$ "'easy", amenable to explicit integration.

Experiment 12.5.11 (Splitting off stiff components)

Recall Ex. 12.0.1 and the IVP studied there:

AWP $\dot{y} = \lambda y(1-y) + \alpha \sin(y)$, $\lambda = 100$, $\alpha = 1$, $y(0) = 10^{-4}$.

small perturbation

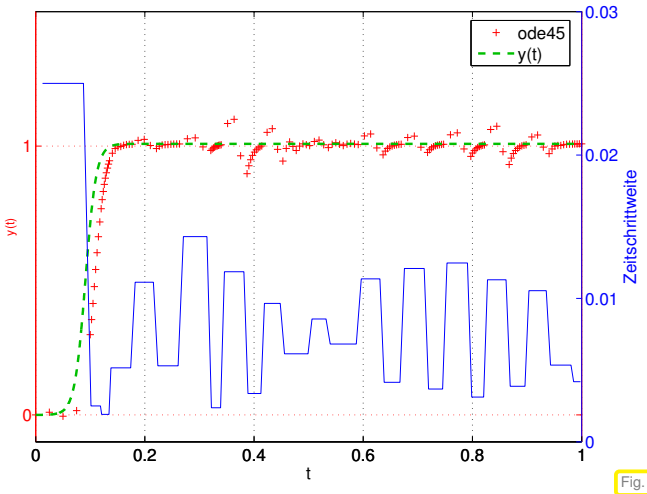


Fig. 467

Solution from `ode45`, see Ex. 12.0.1

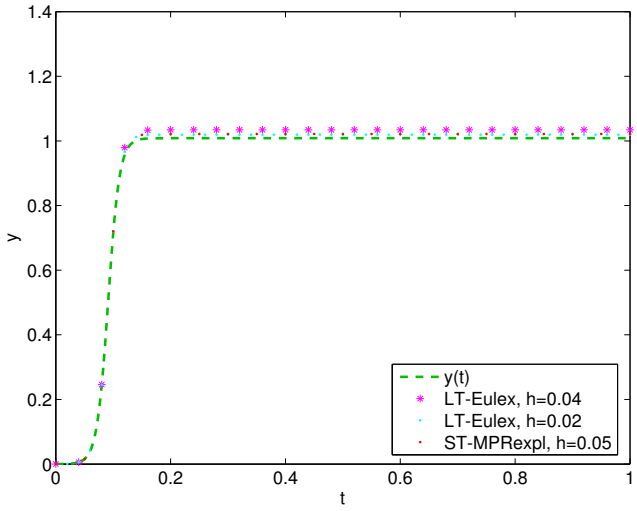


Fig. 468

inexacte splitting method: solution (y_k)

Total number of timesteps

ode45:	152
LT-Eulex, $h = 0.04$:	25
LT-Eulex, $h = 0.02$:	50
ST-MPRexpl, $h = 0.05$:	20

Details of the methods:

LT-Eulex: $\dot{y} = \lambda y(1-y) \to$ exact evolution, $\dot{y} = \alpha \sin y \to$ expl. Euler (11.2.7) **&** Lie-Trotter splitting (12.5.3)

ST-MPRexpl: $\dot{y} = \lambda y(1-y) \to$ exacte evolution, $\dot{y} = \alpha \sin y \to$ expl. midpoint rule (11.4.7) **&** Strang splitting (12.5.4)

We observe that this splitting scheme can cope well with the stiffness of the problem, because the stiff term on the right hand side is integrated exactly.

Example 12.5.12 (Splitting linear and local terms)

In the numerical treatment of partial differential equation one commonly encounters ODEs of the form

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := -\mathbf{A}\mathbf{y} + \begin{bmatrix} g(y_1) \\ \vdots \\ g(y_d) \end{bmatrix}, \quad \mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{d,d} \text{ positive definite } (\to \text{Def. 1.1.8}), \tag{12.5.13}$$

with state space $D = \mathbb{R}^d$, where $\lambda_{\min}(\mathbf{A}) \approx 1$, $\lambda_{\max}(\mathbf{A}) \approx d^2$, and the derivative of $g : \mathbb{R} \to \mathbb{R}$ is bounded. Then IVPs for (12.5.13) will be stiff, since the Jacobian

$$\mathrm{D}\,\mathbf{f}(\mathbf{y}) = -\mathbf{A} + \begin{bmatrix} g'(y_1) & & \\ & \ddots & \\ & & g'(y_d) \end{bmatrix} \in \mathbb{R}^{d,d}$$

will have eigenvalues "close to zero" and others that are large (in modulus) and negative. Hence, $D\mathbf{f}(\mathbf{y})$ will satisfy the criteria (12.2.15) and (12.2.16) for any state $\mathbf{y} \in \mathbb{R}^d$.

The natural splitting is

$$\mathbf{f}(\mathbf{y}) = \mathbf{g}(\mathbf{y}) + \mathbf{q}(\mathbf{y}) \quad \text{with} \quad \mathbf{g}(\mathbf{y}) := -\mathbf{A}\mathbf{y} \quad , \quad \mathbf{q}(\mathbf{y}) := \begin{bmatrix} g(y_1) \\ \vdots \\ g(y_d) \end{bmatrix} .$$

- For the linear ODE $\dot{\mathbf{y}} = \mathbf{g}(\mathbf{y})$ we have to use and L-stable ($\rightarrow$ Def. 12.3.38) single step method, for instance a second-order *implicit* Runge-Kutta method. Its increments can be obtained by solving a *linear system of equations*, whose coefficient matrix will be the same for every step, if uniform timesteps are used.
- The ODE $\dot{\mathbf{y}} = \mathbf{q}(\mathbf{y})$ boils down to decoupled scalar ODEs $\dot{y}_j = g(y_j)$, $j = 1, \ldots, d$. For them we can use an inexpensive *explicit RK-SSM* like the explicit trapezoidal method (11.4.6). According to our assumptions on g these ODEs are not haunted by stiffness.

Summary and Learning Outcomes

Bibliography

[1] W. Dahmen and A. Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer, Heidelberg, 2008.

[2] M.H. Gutknecht. Lineare Algebra. Lecture notes, SAM, ETH Zürich, 2009. http://www.sam.math.ethz.ch/~mhg/unt/LA/HS07/.

[3] W. Hackbusch. *Iterative Lösung großer linearer Gleichungssysteme*. B.G. Teubner–Verlag, Stuttgart, 1991.

[4] Wolfgang Hackbusch. *Iterative solution of large sparse systems of equations*, volume 95 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 1994. Translated and revised from the 1991 German original.

[5] M. Hanke-Bourgeois. *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*. Mathematische Leitfäden. B.G. Teubner, Stuttgart, 2002.

[6] A.R. Laliena and F.-J. Sayas. Theoretical aspects of the application of convolution quadrature to scattering of acoustic waves. *Numer. Math.*, 112(4):637–678, 2009.

[7] K. Nipp and D. Stoffer. *Lineare Algebra*. vdf Hochschulverlag, Zürich, 5 edition, 2002.

[8] A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*, volume 37 of *Texts in Applied Mathematics*. Springer, New York, 2000.

[9] Yousef Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, second edition, 2003.

[10] M. Struwe. Analysis für Informatiker. Lecture notes, ETH Zürich, 2009. https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145.

Index

List of Symbols

$(\mathbf{A})_{i,j} \hat{=}$ reference to entry a_{ij} of matrix $\mathbf{A}$, 49
$(\mathbf{A})_{k:l,r:s} \hat{=}$ reference to submatrix of $\mathbf{A}$ spanning rows $k,\ldots,l$ and columns $r,\ldots,s$, 49
$(\mathbf{x})_i \hat{=}$ i-th component of vector $\mathbf{x}$, 48
$(x_k) *_n (y_k) \hat{=}$ discrete periodic convolution, 301
$C(I) \hat{=}$ space of continuous functions $I \to \mathbb{R}$, 387
$C^1([a,b]) \hat{=}$ space of continuously differentiable functions $[a,b] \mapsto \mathbb{R}$, 394
$J(t_0, \mathbf{y}_0) \hat{=}$ maximal domain of definition of a solution of an IVP, 706
$O \hat{=}$ zero matrix, 50
$O(\cdot) \hat{=}$ Landau symbol, 86
$V^\perp \hat{=}$ orthogonal complement of a subspace, 223
$\mathcal{E} \hat{=}$ expected value of a random variable, 353
$\mathcal{P}_n^T \hat{=}$ space of trigonometric polynomials of degree n, 418
$\mathcal{R}_k(m,n) \hat{=}$ set of rank-k matrices, 269
$\mathrm{D}\,\Phi \hat{=}$ Jacobian of $\Phi : D \mapsto \mathbb{R}^n$ at $\mathbf{x} \in D$, 557
$\mathrm{D}_\mathbf{y}\,\mathbf{f} \hat{=}$ Derivative of $\mathbf{f}$ w.r.t.. $\mathbf{y}$ (Jacobian), 706
$\mathrm{EPS} \hat{=}$ machine precision, 103
$\mathbf{Eig}_\mathbf{A}\lambda \hat{=}$ eigenspace of $\mathbf{A}$ for eigenvalue λ, 613
$\mathcal{R}\mathbf{A} \hat{=}$ range/column space of matrix $\mathbf{A}$, 259
$\mathcal{N}(\mathbf{A}) \hat{=}$ kernel/nullspace of a matrix, 129, 218
$\mathcal{N}\mathbf{A} \hat{=}$ nullspace of matrix $\mathbf{A}$, 259
$\mathcal{K}_l(\mathbf{A},\mathbf{z}) \hat{=}$ Krylov subspace, 679
$\|\mathbf{Ax}-\mathbf{b}\|_2 \to \min \hat{=}$ minimize $\|\mathbf{Ax}-\mathbf{b}\|_2$, 225
$\|\mathbf{A}\|_F^2$, 269
$\|\mathbf{x}\|_A \hat{=}$ energy norm induced by s.p.d. matrix $\mathbf{A}$, 671
$\|\cdot\| \hat{=}$ Euclidean norm of a vector $\in \mathbb{K}^n$, 94
$\|\cdot\| \hat{=}$ norm on vector space, 121
$\|f\|_{L^\infty(I)}$, 387
$\|f\|_{L^1(I)}$, 387
$\|f\|_{L^2(I)}^2$, 387
$\mathcal{P}_k$, 365
$\mathbf{\Psi}^h\mathbf{y} \hat{=}$ discretei evolution for autonomous ODE, 715
$\mathcal{R}(\mathbf{A}) \hat{=}$ image/range space of a matrix, 129, 218
$(\cdot,\cdot)_V \hat{=}$ inner product on vector space V, 466
$\mathcal{S}_{d,\mathcal{M}}$, 403
$\mathbf{A}^\dagger \hat{=}$ Moore-Penrose pseudoinverse of $\mathbf{A}$, 227
$\mathbf{A}^\top \hat{=}$ transposed matrix, 49
$\mathbf{I} \hat{=}$ identity matrix, 50
$\mathbf{h} * \mathbf{x} \hat{=}$ discrete convolution of two vectors, 299
$\mathbf{x} *_n \mathbf{y} \hat{=}$ discrete periodic convolution of vectors, 301
$\bar{z} \hat{=}$ complex conjugation, 49
$\mathbb{C}^- := \{z \in \mathbb{C}\colon \mathrm{Re}\, z < 0\}$, 776
$\mathbb{K} \hat{=}$ generic field of numbers, either $\mathbb{R}$ or $\mathbb{C}$, 48
$\mathbb{K}_*^{n,n} \hat{=}$ set of invertible $n \times n$ matrices, 129
$\mathbb{M} \hat{=}$ set of machine numbers, 97
$\delta_{ij} \hat{=}$ Kronecker symbol, 49, 367
$\delta_{ij} \hat{=}$ Kronecker symbol, 294
$\imath \hat{=}$ imaginary unit, "$\imath := \sqrt{-1}$", 127
$\kappa(\mathbf{A}) \hat{=}$ spectral condition number, 677
$\lambda_\mathcal{T} \hat{=}$ Lebesgue constant for Lagrange interpolation on node set $\mathcal{T}$, 388
$\lambda_{\max} \hat{=}$ largest eigenvalue (in modulus), 677
$\lambda_{\min} \hat{=}$ smallest eigenvalue (in modulus), 677
$\mathbf{1} = [1,\ldots,]^\top$, 752
$\mathrm{Ncut}(\mathcal{X}) \hat{=}$ normalized cut of subset of weighted graph, 630
$\mathrm{argmin} \hat{=}$ (global) minimizer of a functional, 672
$\mathrm{cond}(\mathbf{A})$, 131
$\mathrm{cut}(\mathcal{X}) \hat{=}$ cut of subset of weighted graph, 630
$\mathrm{dist}_{\|\cdot\|}(x,V) \hat{=}$ distance of an element of a normed vector spcace from set V, 438
$\mathrm{env}(\mathbf{A})$, 202
$\mathrm{lsq}(\mathbf{A},\mathbf{b}) \hat{=}$ set of least squares solutions of $\mathbf{ax} = \mathbf{b}$, 219
nnz, 175
$\mathrm{rank}(\mathbf{A}) \hat{=}$ rank of matrix $\mathbf{A}$, 129
$\mathrm{sgn} \hat{=}$ sign function, 399
$\mathrm{vec}(\mathbf{A}) \hat{=}$ vectorization of a matrix, 63
$\mathrm{weight}(\mathcal{X}) \hat{=}$ connectivity of subset of weighted graph, 630
$\overline{} \hat{=}$ complex conjugation, 466
$\overline{m}(\mathbf{A})$, 202
$\rho(\mathbf{A}) \hat{=}$ spectral radius of $\mathbf{A} \in \mathbb{K}^{n,n}$, 613
$\rho_\mathbf{A}(\mathbf{u}) \hat{=}$ Rayleigh quotient, 627
$\mathbf{f} \hat{=}$ right hand side of an ODE, 704
$\sharp \hat{=}$ cardinality of a finite set, 53

$\sigma(\mathbf{A})$ ≙ spectrum of matrix $\mathbf{A}$, 613
$\sigma(\mathbf{M})$ $hat=$ spectrum of matrix $\mathbf{M}$, 675
$\widetilde{\star}$, 102
$\mathbb{S}^1$ ≙ unit circle in the complex plane, 418
$\underline{m}(\mathbf{A})$, 202
$\widehat{f}_j$ ≙ j-th Fourier coefficient of periodic function f, 332
$f^{(k)}$ ≙ k-th derivative of function $f : I \subset \mathbb{R} \to \mathbb{K}$, 435
$f^{(k)}$ ≙ k derivative of f, 120
$m(\mathbf{A})$, 202
$y[t_i, \ldots, t_{i+k}]$ ≙ divided difference, 383
$\| \mathbf{x} \|_1$, 121
$\| \mathbf{x} \|_2$, 121
$\| \mathbf{x} \|_\infty$, 121
$\dot{}$ ≙ Derivative w.r.t. time t, 698

TOL tolerance, 735

Examples and Remarks