

# NUMERISCHE METHODEN

Vasile Gradinaru

Skript zur Vorlesung 401-1662-10L  
für BSc Physik an der ETH Zürich  
2011-20

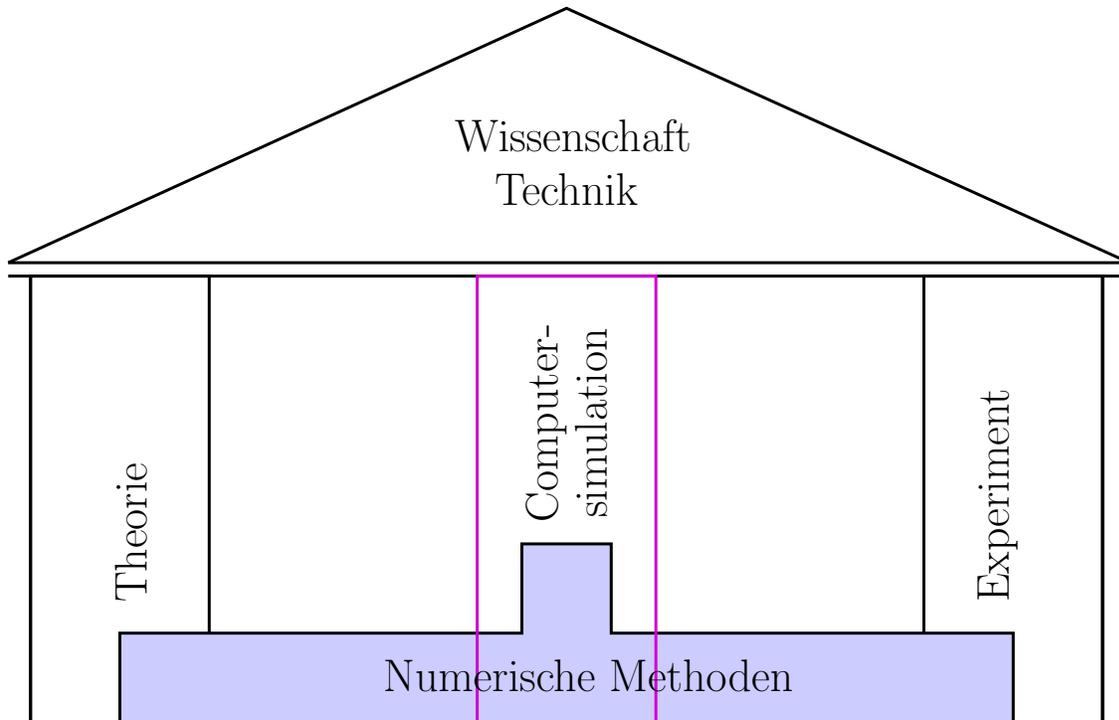
©Seminar für Angewandte Mathematik, ETH Zürich<sup>1</sup>

Generiert am July 27, 2020

---

<sup>1</sup>Alle Rechte vorbehalten; keine Vervielfältigung jeglicher Art oder Umfang ist ohne die Zustimmung des Autors erlaubt.

## Ziele dieser Vorlesung



Die numerischen Methoden sind allgegenwärtig in der modernen Wissenschaft und Technik, vor allem als Komponenten der Computersimulationen, aber auch als Hilfswerkzeuge bei Experimenten oder bei der Entwicklung neuer Theorien. Der Experimentator sollte wissen, wie die Ergebnisse, die seine Geräte liefern, zustandekommen, wie zuverlässig sie sind, warum manche Artefakte entstehen und wie diese zu vermeiden sind. Beim Aufstellen neuer Theorien braucht man oft Berechnungen, die nicht analytisch durchführbar sind. Oder der Rechner ist einfach schneller in manchen Fällen. . . Man baut oft ein neues mathematisches Modell für ein physikalisches Problem und man testet es: ob die Ergebnisse etwas mit der Realität zu tun haben? Damit dies möglich ist, muss man numerische Methoden wählen, die geeignet zur Lösung des Problems sind. Wir werden sehen, dass dies nicht so einfach ist, sobald das mathematische Modell nicht trivial ist. Wir können numerischen Approximationen zur Lösung mathematischer Probleme nur trauen, wenn diese numerische Lösungen sehr sorgfältig berechnet worden sind. Das genaue Wissen, wo die Grenzen eines numerischen Verfahrens liegen, und was typischerweise passiert, wenn diese Grenzen nicht eingehalten werden, ist sogar wichtiger als das Verfahren selber, das man einfach in einem Buch oder im Internet finden kann: wir wollen sicher die Eigenschaften eines Modells nicht mit diejenigen eines Verfahren verwechseln! Das Eineignen dieses Wissens geht nur über Probieren und Experimentieren, wenn die Zeit für die fundierte mathematische Analyse verschiedener Verfahren fehlt. Deswegen ist der Inhalt dieser Vorlesung Implementierungs- und Experimentierungsorientiert: die Studenten müssen selber die Methoden in verschiedene Umgebungen ausprobieren, um Erfahrungen zu sammeln.

Lernziele dieser Vorlesung sind:

- Übersicht über die wichtigsten Algorithmen zur Lösung der grundlegenden numerischen Probleme in der Physik und ihren Anwendungen;
- Übersicht über Softwarequellen (engl. *software repositorys*) zur Problemlösung;
- Fertigkeit konkrete Probleme mit diesen Werkzeugen numerisch zu lösen;
- Fähigkeit numerische Resultate zu interpretieren.

Der Schwerpunkt liegt auf dem Erwerb von Fertigkeiten in der Anwendung von numerischen Verfahren: Algorithmen (Prinzipien, Ziele und Grenzen) und Implementationen (Effizienz und Stabilität) werden mittels relevanten numerischen Experimenten (Design und Interpretation) vorgestellt. Wir werden Theorien und Beweise nur dann vorbringen, wenn es essenziell für das Verständnis der Algorithmen erscheinen wird.

## Materialien

Viele Ideen und Materialien, die in diesem Skript Platz gefunden haben, sind aus den Diskussionen und Erfahrungen am Seminar für Angewandte Mathematik an der ETH Zürich entstanden. Meinen Kollegen bin ich dafür sowie für deren Vertrauen, mich diese neue und aufregende Vorlesung entwickeln zu lassen, sehr dankbar. Der Autor ist von den vorhandenen Materialien der Slides *Numerical Methods for CSE* und *Numerische Mathematik*, die vor allem vom Prof. Ralf Hiptmair entwickelt worden sind, ausgegangen. Ich danke auch meinen Assistierenden und den zahlreichen Studenten der Studiengänge Physik und Rechnergestützte Wissenschaften (RW/CSE), die mir Kommentare und Verbesserungsvorschläge zur Vorlesung und zum Skript gegeben haben.

Folgende Bücher wurden für die Vorbereitung dieses Skriptes verwendet und können teilweise als Begleitlektüre zur Vorlesung in Betracht gezogen werden.

- \* M. HANKE-BOURGEAIS, *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*, Mathematische Leitfäden, B.G. Teubner, Stuttgart, 2009, [NEBIS 009860519](#)
- \* P. DEUFLHARD AND A. HOHMANN, *Numerische Mathematik. Eine algorithmisch orientierte Einführung*, DeGruyter, Berlin, 2008, [NEBIS 009855563](#)
- \* J. STÖR AND R. BULIRSCH, *Einführung in die Numerische Mathematik*, Springer Verlag, 2005, [NEBIS 009840490](#)
- \* W. DAHMEN AND A. REUSKEN, *Numerik für Ingenieure und Naturwissenschaftler*, Springer 2008, [NEBIS 00985199](#)
- \* QUARTERONI, SACCO AND SALERI, *Numerische Mathematik 1 + 2*, Springer 2007, [NEBIS 009842244](#)

- \* L.N. TREFETHEN AND D. BAU, *Numerical Linear Algebra*, SIAM 1997, [NEBIS 009819168](#)
- \* G. STRANG, *Wissenschaftliches Rechnen*, Springer 2010, [NEBIS 009860694](#)
- \* P. DEVRIES AND J. HASBUN, *A First Course in Computational Physics*, Jones and Bartlett Publishers; 2 edition, 2010, [NEBIS 006032942](#)
- \* W.L. DUNN, J.K. SHULTIS, *Exploring Monte Carlo Methods*, Amsterdam, Elsevier, 2012, [NEBIS 009882256](#)
- \* T. MÜLLER-GRONBACH, E. NOVAK, K. RITTER, *Monte Carlo-Algorithmen*, Berlin, Springer, 2012, [NEBIS 009889710](#)
- \* H.P. LANGTANGEN, *A primer on scientific programming with Python*, Springer, 2012, [NEBIS 009893026](#)
- \* H.P. LANGTANGEN, *Python scripting for computational science*, Springer, 2009, [NEBIS 009859422](#)
- \* [Python Scientific Lecture Notes](#)

Erwartet werden solide Kenntnisse in Analysis (Approximation und Vektoranalysis) und linearer Algebra (Gauss-Elimination, Matrixzerlegungen, sowie Algorithmen, Vektor- und Matrizenrechnung: Matrixmultiplikation, Determinante, LU-Zerlegung nicht-singulärer Matrizen).

## Motivation und Überblick

# Contents

<b>1</b>	<b>Numerische Quadratur</b>	<b>6</b>
1.1	Grundbegriffe . . . . .	6
1.2	Äquidistante Stützstellen . . . . .	8
1.3	Nicht äquidistante Stützstellen . . . . .	14
1.4	Adaptive Quadratur . . . . .	21
1.5	Quadratur in $\mathbb{R}^d$ und dünne Gitter . . . . .	23
1.6	Monte-Carlo Quadratur . . . . .	28
1.7	Methoden zur Reduktion der Varianz . . . . .	45
<b>2</b>	<b>Gewöhnliche Differentialgleichungen</b>	<b>56</b>
2.1	Anfangswertprobleme . . . . .	56
2.1.1	Definitionen . . . . .	57
2.1.2	Evolutionsoperator . . . . .	59
2.1.3	Lineare Anfangswertprobleme . . . . .	62
2.1.4	Existenz, Eindeutigkeit, Stabilität . . . . .	65
2.1.5	Erhaltungsgrößen . . . . .	66
2.2	Polygonzugverfahren . . . . .	70
2.2.1	Euler-Verfahren und implizite Mittelpunktsregel . . . . .	71
2.2.2	Strukturerhaltung . . . . .	73
2.3	Störmer-Verlet-Verfahren . . . . .	78
2.4	Splitting-Verfahren . . . . .	89
2.4.1	Splitting-Verfahren für gestörte Systeme . . . . .	104
2.5	Runge-Kutta-Verfahren . . . . .	108
2.6	Schrittweitensteuerung . . . . .	124
2.7	Partitionierte Runge-Kutta Verfahren . . . . .	156
2.8	Magnus-Integratoren für lineare Gleichungen . . . . .	157
<b>3</b>	<b>Nullstellensuche</b>	<b>162</b>
3.1	Iterative Verfahren . . . . .	162
3.2	Abbruchkriterien . . . . .	168
3.3	Fixpunktiteration . . . . .	169
3.4	Intervallhalbierungsverfahren . . . . .	176
3.5	Newton-Verfahren in 1 Dimension . . . . .	177
3.6	Sekantenverfahren . . . . .	179
3.7	Newton-Verfahren in $n$ Dimensionen . . . . .	181

3.8	Gedämpftes Newton-Verfahren . . . . .	186
3.9	Das Quasi-Newton Verfahren . . . . .	190
<b>4</b>	<b>Integratoren für steife Differentialgleichungen</b>	<b>195</b>
4.1	Stabilitätsanalyse der RK-Verfahren . . . . .	195
4.2	Linear-implizite RK-Verfahren . . . . .	209
4.3	Exponentielle Integratoren . . . . .	212
<b>5</b>	<b>Intermezzo über (numerische) lineare Algebra</b>	<b>216</b>
5.1.1	Grundlagen . . . . .	216
5.1.2	Wichtige Zerlegungen . . . . .	219
<b>6</b>	<b>Ausgleichsrechnung</b>	<b>239</b>
6.1	Lineare Ausgleichsrechnung . . . . .	240
6.1.1	Normalgleichungen . . . . .	242
6.1.2	Lösung mittels orthogonaler Transformationen . . . . .	245
6.1.3	Totale Ausgleichsrechnung . . . . .	249
6.2	Nichtlineare Ausgleichsrechnung . . . . .	250
6.2.1	Newton-Verfahren im $\mathbb{R}^n$ . . . . .	251
6.2.2	Gauss-Newton Verfahren . . . . .	251
6.2.3	Levenberg-Marquardt-Methode . . . . .	259
<b>7</b>	<b>Eigenwerte</b>	<b>261</b>
7.1	Motivation und Theorie . . . . .	261
7.2	“Direktes” Lösen . . . . .	265
7.3	Potenzmethoden . . . . .	271
7.3.1	Direkte Potenzmethode . . . . .	271
7.3.2	Inverse Iteration . . . . .	275
7.3.3	Vorkonditionierte inverse Iteration . . . . .	278
7.4	Krylov-Verfahren . . . . .	282
<b>8</b>	<b>Polynominterpolation</b>	<b>296</b>
8.1	Interpolation und Polynome . . . . .	296
8.2	Newton Basis und Dividierte Differenzen . . . . .	300
8.3	Lagrange- und baryzentrische Interpolationsformeln . . . . .	303
8.4	Chebyshev-Interpolation . . . . .	308
8.4.1	Fehlerbetrachtung . . . . .	311
8.4.2	Interpolation und Auswertung . . . . .	313
<b>9</b>	<b>Trigonometrische Interpolation</b>	<b>315</b>
9.1	Motivation . . . . .	315
9.2	Diskrete Fouriertransformation . . . . .	320
9.3	FFT - Schnelle Fouriertransformation . . . . .	325
9.4	Trigonometrische Interpolation und rechen-technische Aspekte . . . . .	330
9.5	Trigonometrische Interpolation und Fehlerabschätzungen . . . . .	334
9.6	DFT und Chebyshev-Interpolation . . . . .	340

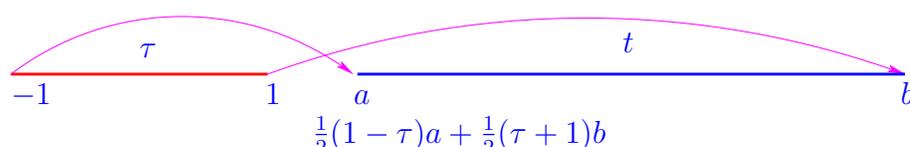
# Chapter 1

## Numerische Quadratur

Viele Integrale sind mit analytischen Mitteln nicht oder nur schwer berechenbar. In solchen Fällen bedient man sich der in diesem Kapitel erläuterten Verfahren, um die Integrale näherungsweise zu bestimmen.

### 1.1 Grundbegriffe

In diesem Kapitel arbeiten wir oft auf dem Intervall  $[-1, 1]$ . Soll eine Funktion auf einem anderen Intervall  $[a, b]$  integriert werden, so muss dieses vorher durch eine affine Transformation auf  $[-1, 1]$  transformiert werden.

$$\int_a^b f(t) dt = \frac{1}{2}(b-a) \int_{-1}^1 \widehat{f}(\tau) d\tau, \quad \widehat{f}(\tau) := f\left(\frac{1}{2}(1-\tau)a + \frac{1}{2}(\tau+1)b\right),$$


#### *Idee:*

Wir approximieren die Funktion  $f$  durch eine Funktion  $f_n \in \text{span}\{b_0, b_1, \dots, b_{n-1}\}$ , wobei die  $b_j$  eine Basis eines gewissen Unterraums des Raums der Funktionen auf  $[a, b]$  bilden, z.B. des Raums der Polynome vom Grade kleiner als  $n$ . Daraus resultiert:

$$\int_a^b f(x) dx \approx \int_a^b f_n(x) dx = \int_a^b \left( \sum_{i=0}^{n-1} \alpha_i b_i(x) \right) dx = \sum_{i=0}^{n-1} \alpha_i \int_a^b b_i(x) dx.$$

Die approximierenden Funktionen sind meistens Polynome. Bei spezifischen Problemen kann jedoch die Wahl anderer Funktionen (z.B. rationaler Funktionen oder Schwingungen verschiedener Frequenzen) von Vorteil sein.

**Definition 1.1.1.** (Quadratur) Das Integral wird durch eine gewichtete Summe von von der Funktion  $f$  an verschiedenen Stellen  $c_i^n$  angenommenen Werte approximiert:

$$\int_a^b f(x)dx \approx Q_n(f; a, b) := \sum_{i=1}^n \omega_i^n f(c_i^n). \quad (1.1.1)$$

Hierbei sind die  $\omega_i^n$  die *Gewichte* und die  $c_i^n \in [a, b]$  die *Knoten* der Quadraturformel. Für das Referenzintervall  $[-1, 1]$  können die Gewichte und Knoten für eine Quadraturformel bestimmt und tabelliert werden. Dann gilt:

$$\int_a^b f(t) dt \approx \frac{1}{2}(b-a) \sum_{j=1}^n \hat{w}_j f(\hat{c}_j) = \sum_{j=1}^n w_j f(c_j) \quad \text{mit} \quad \begin{aligned} c_j &= \frac{1}{2}(1 - \hat{c}_j)a + \frac{1}{2}(1 + \hat{c}_j)b, \\ w_j &= \frac{1}{2}(b-a)\hat{w}_j. \end{aligned}$$

**Definition 1.1.2.** (Fehler)

Der Fehler einer Quadraturformel  $Q_n(f)$  ist

$$E(n) := \left| \int_a^b f(x)dx - Q_n(f; a, b) \right|.$$

Wir reden von

**algebraischer Konvergenz** wenn  $E(n) = O(\frac{1}{n^p})$  mit  $p > 0$ ,

und von

**exponentieller Konvergenz** wenn  $E(n) = O(q^n)$  mit  $0 \leq q < 1$ .

**Definition 1.1.3.** Eine Quadraturformel besitzt Ordnung  $n+1$  wenn sie Polynome vom Grade  $n$  exakt integriert.

Zu beachten: ein Polynom vom effektiven Grade  $n$  wird durch genau  $n+1$  Koeffizienten dargestellt.

**Definition 1.1.4.** Eine Quadraturformel auf  $[-1; 1]$  mit Knoten  $c_1, \dots, c_n$  und Gewichte  $\omega_1, \dots, \omega_n$  heisst symmetrisch, falls:

$$\omega_i = \omega_{n+1-i} \quad \text{und} \quad c_i = -c_{n+1-i}.$$

**Bemerkung 1.1.5.** Die Mittelpunkts-, Trapez- und Simpson-Regel aus Abschnitt 1.2 sind symmetrisch.

Es gilt:

**Theorem 1.1.6.** Die Ordnung einer symmetrischen Quadraturformel ist gerade.

*Proof.* Wir beweisen diesen Satz mittels Induktion.

**Induktionsannahme:** die Quadraturformel sei exakt für  $p \in \mathcal{P}_{2m-2}$ .

**Induktionsschritt:** Sei  $f \in \mathcal{P}_{2m-1}$ , dann gilt  $f(x) = ax^{2m-1} + g(x)$  mit  $g(x) \in \mathcal{P}_{2m-2}$ , woraus folgt:

$$\int_{-1}^1 f(x) dx = a \int_{-1}^1 x^{2m-1} dx + \int_{-1}^1 g(x) dx = \sum_{j=1}^n \omega_j g(c_j)$$

$$Q_n(f) = \sum_{j=1}^n \omega_j f(c_j) = a \sum_{j=1}^n \omega_j c_j^{2m-1} + \sum_{j=1}^n \omega_j g(c_j).$$

Es bleibt noch zu zeigen, dass  $\sum_{j=1}^n \omega_j c_j^{2m-1} = 0$  für eine symmetrische Quadraturformel. In der Tat:

$$\sum_{j=1}^n \omega_j c_j^{2m-1} = \sum_{j=1}^n \omega_{n+1-j} \underbrace{(-c_{n+1-j})}_k^{2m-1} = - \sum_{k=1}^n \omega_k c_k^{2m-1}.$$

□

## 1.2 Äquidistante Stützstellen

Hier betrachten wir die wichtigsten Quadraturformeln mit äquidistanten Knoten.

**Mittelpunkt-Regel**[0.5ex] Die Mittelpunktsregel ist nichts anderes, als die aus der Analysis bekannte Näherung des bestimmten Integrals:

$$Q^M(f; a, b) = (b - a) f\left(\frac{a + b}{2}\right). \quad (1.2.2)$$

Die Mittelpunktsregel besitzt Ordnung 2, denn sie ist exakt für lineare Funktionen.

**Trapez-Regel**[0.5ex] Die nächste Formel der Ordnung 2 ist die aus der Schule bekannte Trapez-Regel:

$$Q^T(f; a, b) = \frac{b - a}{2} (f(a) + f(b)). \quad (1.2.3)$$

Für den Fehler der Trapez-Regel gilt:

$$E(n) = \left| -\frac{1}{12} (b - a)^3 f^{(2)}(\xi) \right|, \text{ mit einem } \xi \in [a, b].$$

**Simpson-Regel**[0.5ex] Die Simpson-Regel ist eine Quadraturformel der Ordnung 4 und bedarf dreier Stützstellen:

$$Q^S(f; a, b) = \frac{b - a}{6} \left( f(a) + 4f\left(\frac{a + b}{2}\right) + f(b) \right). \quad (1.2.4)$$

Während die Trapez-Regel auf linearer Interpolation auf  $[a, b]$  basiert, approximieren wir bei der Simpson-Regel die Funktion mit einer Parabel; somit ist die Ordnung

der Simpson-Regel mindestens 3, aber da die Quadraturformel symmetrisch ist, ist die Ordnung 4, nach Theorem 1.1.6<sup>1</sup>. Für den Fehler der Simpson-Regel gilt:

$$E(n) = \left| -\frac{1}{90} \left( \frac{b-a}{2} \right)^5 f^{(4)}(\xi) \right|, \text{ mit einem } \xi \in [a, b].$$

Es lohnt sich anzumerken, dass ab Ordnung 8 negative Gewichte auftreten, was die Quadraturformeln aufgrund von Auslöschung numerisch instabil werden lässt. Es handelt sich um dasselbe Phänomen wie bei der Interpolation mittels Polynomen höheren Grades auf äquidistanten Punkten.

**Bemerkung 1.2.1.** Schranken für den Quadraturfehler bekommt man direkt aus den entsprechenden Resultaten für die Interpolation mittels Lagrange-Polynomen vom Grad  $n - 1$ :

$$f \in C^n([a, b]) \implies \left| \int_a^b f(t) dt - Q_n(f) \right| \leq \frac{1}{n!} (b-a)^{n+1} \|f^{(n)}\|_{L^\infty([a, b])}.$$

### Summierte Quadraturformeln

Eine äusserst einfache Möglichkeit, die Genauigkeit der Integration zu verbessern, beruht auf dem “zerlege und herrsche” (“divide and conquer”)-Prinzip. Man unterteilt das Intervall in Subintervalle und wendet auf jedes Subintervall die Quadraturformel an. Die Ergebnisse werden summiert.<sup>2</sup> Formell lautet diese Aussage:

$$\int_a^b f(x) dx = \sum_{i=0}^{N-1} \int_{x_i}^{x_{i+1}} f(x) dx = \sum_{i=0}^{N-1} Q_n(f; x_i, x_{i+1}),$$

wobei hier das Intervall  $[a, b]$  in  $N$  Subintervalle unterteilt wurde. Konkret gilt für  $h = \frac{b-a}{N}$  und  $x_0 = a$ ,  $x_i = x_0 + ih$ ,  $x_N = b$ , folgendes:

**Summierte Mittelpunkt-Regel:**

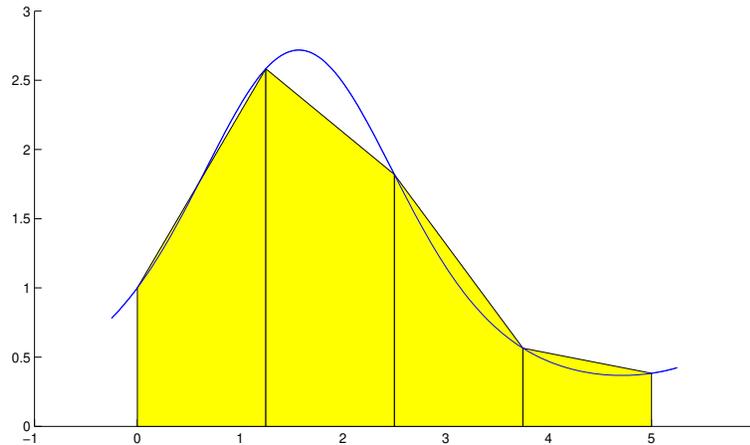
$$I(f; a, b) \approx \sum_{i=0}^{N-1} h f\left(\frac{x_i + x_{i+1}}{2}\right),$$

**Summierte Trapez-Regel:**

$$I(f; a, b) \approx \sum_{i=0}^{N-1} \frac{h}{2} (f(x_i) + f(x_{i+1})) = \frac{h}{2} \left( f(a) + 2 \sum_{i=1}^{N-1} f(x_i) + f(b) \right).$$

<sup>1</sup>Aus dem selben Grund hat die Mittelpunkt-Regel auch Ordnung 2 und nicht 1

<sup>2</sup>Dies ist analog zu den Riemannschen Summen in der Analysis



**Abb. 1.2.2.** Summierte Trapezregel: die Funktion wird durch eine stückweise lineare, aber insgesamt stetige Funktion approximiert.

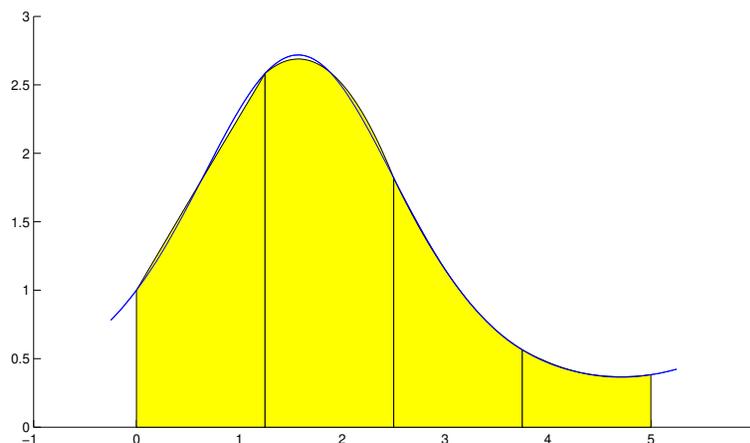
Für den Fehler gilt:

$$E(n) \leq \frac{h^2}{12}(b-a) \max_{x \in [a,b]} |f''(x)| .$$

### Summierte Simpson-Regel

$$\begin{aligned} I(f; a, b) &\approx \sum_{i=0}^{N-1} \frac{h}{6} \left( f(x_i) + 4f\left(\frac{x_i + x_{i+1}}{2}\right) + f(x_{i+1}) \right) \\ &= \frac{h}{6} \left( f(a) + 2 \sum_{i=1}^{N-1} f(x_i) + 4 \sum_{i=1}^N f\left(\frac{x_{i-1} + x_i}{2}\right) + f(b) \right) \\ &= \frac{\tilde{h}}{3} \left( f(\tilde{x}_0) + 2 \sum_{i=1}^{N-1} f(\tilde{x}_{2i}) + 4 \sum_{i=1}^N f(\tilde{x}_{2i-1}) + f(\tilde{x}_{2N}) \right) , \end{aligned}$$

wobei  $\tilde{h} = \frac{h}{2}$  und  $\tilde{x}_{2i} = x_i$  und  $\tilde{x}_{2i-1} = \frac{x_{i-1} + x_i}{2}$ .



**Abb. 1.2.3.** Simpson-Regel: die approximierende Funktion ist stückweise quadratisch und global stetig.

**Theorem 1.2.4.** (Konvergenz der zusammengesetzten Quadraturformeln)

Für eine zusammengesetzte Quadraturformel  $Q$ , die auf einer lokalen Quadraturformel der Ordnung  $p \in \mathbb{N}$  basiert, gibt es ein  $C > 0$ , so dass

$$\left| \int_I f(t) dt - Q(f) \right| \leq Ch^p \|f^{(p)}\|_{L^\infty(I)} \quad \text{für alle } f \in C^p(I).$$

**Beispiel 1.2.5.** (Konvergenz der zusammengesetzten Quadraturformeln)

Wir untersuchen nun experimentell zusammengesetzte Quadraturformeln basierend auf:

- Trapez-Regel: lokale Ordnung 2 (exakt für lineare Funktionen),
- Simpson-Regel: lokale Ordnung 4

auf einem äquidistanten Gitter  $\mathcal{M} := \{jh\}_{j=0}^n$ ,  $h = \frac{1}{n}$ ,  $n \in \mathbb{N}$ .

**Code 1.2.6:** Zusammengesetzte Trapez-Regel

```

1  from numpy import linspace, sum
3  def trapezoidal(f, a, b, N):
4      r"""
5          Trapezoidal quadrature of function f from a to b with N subintervals.
6
7          f:      Function f(x)
8          a, b:   Bounds of the integration interval
9          N:      Number of subintervals
10         """
11     x, h = linspace(a, b, N+1, retstep=True)
12     # quadrature weights:
13     # boundary nodes: w=1/2
14     # internal nodes: w=1
15     I = h/2.0 * (f(x[0]) + 2.0*sum(f(x[1:-1])) + f(x[-1]))
16     """#note: this is a compact way of writing. One could also use a for-cycle:
17         I = 0.0
18         for k in range(N):
19             I = I + h/2.0 * (f(x[k+1]) + f(x[k]))
20         #the two strategies are equivalent"""
21     return I
22
23 if __name__ == "__main__":
24     from numpy import arange, size, zeros, log, polyfit
25     from scipy import integrate
26
27     # Define a function and an interval:
28     f = lambda x: 1.0 / (1.0 + (5*x)**2)
29     left = 0.0

```

```

31     right = 1.0
33     # Exact integration with scipy.integrate.quad:
34     exact, e = integrate.quad(f, left, right)
35
36     # Trapezoidal rule for different number of quadrature points
37     N = arange(2, 101)
38     res = zeros(size(N))
39     for i, n in enumerate(N):
40         res[i] = trapezoidal(f, left, right, n)
41
42     err = abs(res - exact)
43
44     # Linear fit to determine convergence order
45     p = polyfit(log(N[-50:]), log(err[-50:]), 1)
46     print("Convergence order: %f" % -p[0])

```

Code 1.2.7: zusammengesetzte Simpson-Regel

```

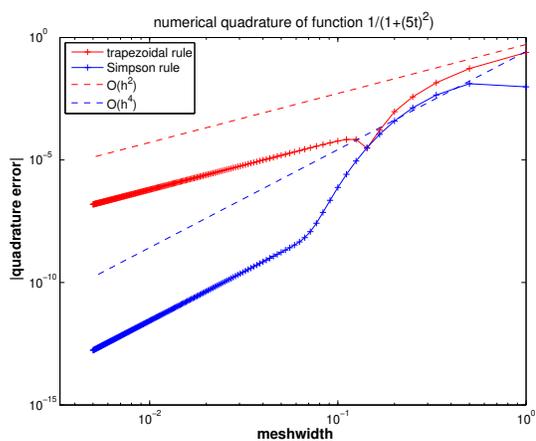
1  from numpy import linspace, sum
2
3  def simpson(f, a, b, N):
4      r"""
5          Simpson quadrature of function f from a to b with N subintervals.
6
7          f:      Function f(x)
8          a, b:   Bounds of the integration interval
9          N:      Number of subintervals
10         """
11     x, h = linspace(a, b, 2*N+1, retstep=True)
12     # quadrature weights:
13     # boundary nodes:  w=1/3
14     # internal nodes:  w=2/3
15     # midpoint nodes:  w=4/3
16     I = h/3.0 * sum(f(x[:-2:2]) + 4.0*f(x[1:-1:2]) + f(x[2::2]))
17     """equivalent, less compact way of writing:
18         I = 0.0
19         for k in range(0,2*N,2):
20             I = I + (2.0*h)/6.0 * (f(x[k]) + 4.0 * f(x[k+1]) + f(x[k+2]))"""
21     return I
22
23  if __name__ == "__main__":
24     from numpy import arange, size, zeros, log, polyfit
25     from scipy import integrate
26
27     # Define a function and an interval:
28     f = lambda x: 1.0 / (1.0 + (5*x)**2)
29     left = 0.0
30     right = 1.0
31
32     # Exact integration with scipy.integrate.quad:
33     exact, e = integrate.quad(f, left, right)

```

```

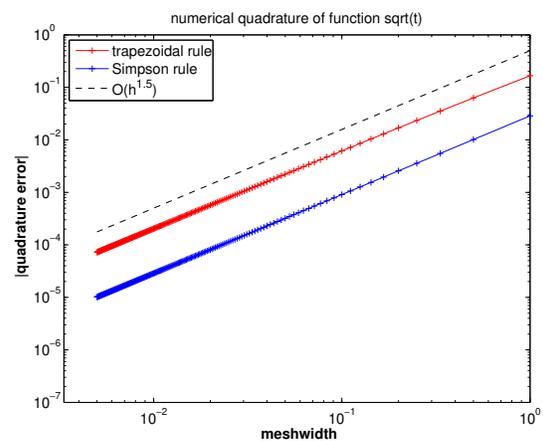
35 # Simpson rule for different number of quadrature points
36 N = arange(2, 101)
37 res = zeros(size(N))
38 for i, n in enumerate(N):
39     res[i] = simpson(f, left, right, n)
40
41     err = abs(res - exact)
42
43 # Linear fit to determine convergence order
44 p = polyfit(log(N[-50:]), log(err[-50:]), 1)
45 print("Convergence order: %f" % -p[0])

```



Quadratur-Fehler für  $f_1(t) := \frac{1}{1+(5t)^2}$

Ordnung 2 für die Trapez-Regel und  
Ordnung 4 für Simpson-Regel



Quadratur-Fehler für  $f_2(t) := \sqrt{t}$

Unglattheit von  $f$  beschränkt die  
Konvergenzordnung

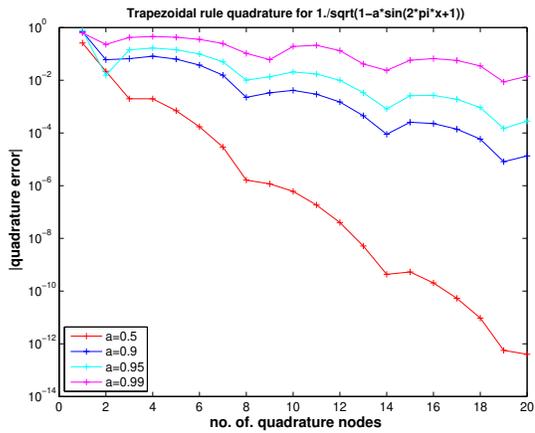
**Beispiel 1.2.8.** (Konvergenz der Trapez-Regel auf äquidistanten Punkten)

Manchmal gibt es gute Überraschungen: die Konvergenz einer zusammengesetzten Quadraturformel kann mitunter viel besser sein als erwartet von der Ordnung der lokalen Quadraturformel.

Die Trapez-Regel auf äquidistanten Punkten hat Ordnung 2, aber wenn wir sie auf den 1-periodischen glatten (analytischen) Integranden anwenden

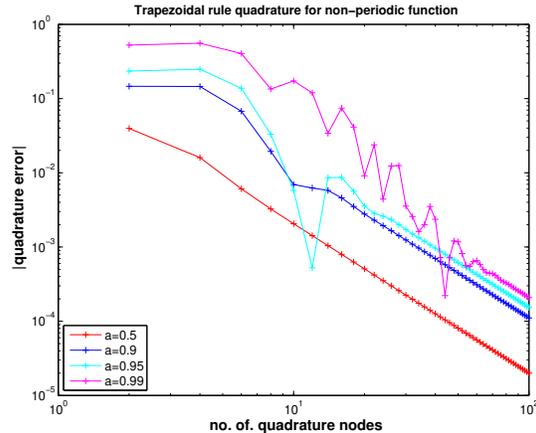
$$f(t) = \frac{1}{\sqrt{1 - a \sin(2\pi t - 1)}}, \quad 0 < a < 1$$

erhalten wir folgende erstaunliche Ergebnisse (als “exakten Wert des Integrals” verwenden wir  $T_{500}$ ):



Quadratur-Fehler für  $T_n(f)$  auf  $[0, 1]$

Exponentielle Konvergenz!



Quadratur-Fehler für  $T_n(f)$  auf  $[0, \frac{1}{2}]$

nur algebraische Konvergenz...

Die Erklärung basiert auf der Tatsache, dass  $T_m$  exakt für trigonometrische Polynome vom Grad  $< 2m$  sind:

$$f(t) = e^{2\pi ikt} \begin{cases} \int_0^1 f(t) dt = \begin{cases} 0, & \text{wenn } k \neq 0, \\ 1, & \text{wenn } k = 0. \end{cases} \\ T_m(f) = \frac{1}{m} \sum_{l=0}^{m-1} e^{\frac{2\pi i}{m}lk} \stackrel{(9.2.13)}{=} \begin{cases} 0, & \text{wenn } k \notin m\mathbb{Z}, \\ 1, & \text{wenn } k \in m\mathbb{Z}. \end{cases} \end{cases}$$

Allerdings braucht man einige Theoreme aus der Funktionentheorie, um die exponentielle Konvergenz für analytische Funktionen in diesem Experiment zu beweisen. In den numerischen Ergebnissen sieht man den Einfluss des Parameters  $a$ : die exponentielle Konvergenz verschwindet wenn  $a$  nah an 1 liegt ( $a = 1$  macht die Funktion  $f$  unstetig).

### 1.3 Nicht äquidistante Stützstellen

Wir können die Genauigkeit erhöhen, ohne das Intervall zu unterteilen. Dafür müssen wir aber eine flexiblere Wahl der Quadraturpunkte erlauben.

#### Gauss Quadratur

Wir wählen  $n$  Gewichte und  $n$  Punkte, so dass für den Fehler gilt:  $E(n) = 0$  für alle Polynome vom Grade  $\leq 2n - 1$ .

**Beispiel 1.3.1.** (Quadraturformel der Ordnung 4 mit 2 Knoten)

Die Bedingung, dass die Quadraturformel Ordnung 4 hat, ist (das erste falsche Ergebnis ist für  $\int_a^b x^4 dx$ ):

$$Q_n(p) = \int_a^b p(t) dt \quad \forall p \in \mathcal{P}_3 \iff Q_n(t^q) = \frac{1}{q+1} (b^{q+1} - a^{q+1}), \quad q = 0, 1, 2, 3.$$

Wir haben also 4 Gleichungen für die Gewichte  $\omega_j$  und Knoten  $\xi_j$ , mit  $j = 1, 2$ . Für  $a = -1, b = 1$  lauten diese:

$$\begin{aligned} \int_{-1}^1 1 dt = 2 = 1\omega_1 + 1\omega_2, & \quad \int_{-1}^1 t dt = 0 = \xi_1\omega_1 + \xi_2\omega_2 \\ \int_{-1}^1 t^2 dt = \frac{2}{3} = \xi_1^2\omega_1 + \xi_2^2\omega_2, & \quad \int_{-1}^1 t^3 dt = 0 = \xi_1^3\omega_1 + \xi_2^3\omega_2. \end{aligned} \quad (1.3.5)$$

MAPLE löst das bequem für uns:

```
> 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);
```

oder symbolic python:

```
1 from sympy import *
3 x = Symbol("x")
3 n = Symbol("n", positive=True)
5 # First we generate two streams of symbols corresponding to knots and weights
5 xigen = numbered_symbols(prefix="xi", start=1)
7 omegagen = numbered_symbols(prefix="omega", start=1)
7 # Now we specify how many parameters we need. In this case, two knots and two
7 weights. Therefore we set N = 2
9 N = 2 # Choose <= 3
11 xis = [ next(xigen) for i in range(N) ]
11 wis = [ next(omegagen) for i in range(N) ]
13 # Using sympy one can perform simple symbolic integrations, which deliver the
13 exact result.
15 # In order to do this we use the function "integrate"
15 # For every n in the interval [0, 2*N-1] we set the condition that the result of
15 the numerical integration
17 #be equal to the analytical value
17 eqns = [ integrate(x**n, (x, -1, 1)) - (sum([wi*xi**n for xi,wi in
17 zip(xis,wis)]) for n in range(2*N))
19 pprint(eqns)
19 # Therefore we get a system of equations which we need to solve:
21 #we set eqns = 0 and find the corresponding knots and weights
21 sols = solve(eqns, numerical=True)
21 pprint(sols)
```

und ergibt:

$$\text{Gewichte und Knoten: } \left\{ \omega_2 = 1, \omega_1 = 1, \xi_1 = 1/\sqrt{3}, \xi_2 = -1/\sqrt{3} \right\}$$

$$\text{Gauss-Quadraturformel: } \int_{-1}^1 f(x) dx \approx f\left(\frac{1}{\sqrt{3}}\right) + f\left(-\frac{1}{\sqrt{3}}\right). \quad (1.3.6)$$

**Bemerkung 1.3.2.** Die Stützstellen für die Gauss'sche Quadraturformel werden am effizientesten durch das Lösen des Eigenwertproblems zu folgender Matrix bestimmt:

$$\begin{pmatrix} 0 & b_1 & \cdots & 0 \\ b_1 & 0 & b_2 & \vdots \\ \vdots & \ddots & \ddots & b_n \\ 0 & \cdots & b_n & 0 \end{pmatrix}$$

mit  $b_j = \frac{j}{\sqrt{4j^2-1}}$ ,  $j = 1, 2, \dots, n$ .

Code 1.3.3: Algorithm von Golub und Welsch

```

1  from numpy import arange, diag, sqrt
2  from numpy.linalg import eigh
3
4  def gaussquad(n):
5      r"""
6          Compute nodes and weights for Gauss-Legendre quadrature.
7
8          n: Number of node-weight pairs
9          """
10     i = arange(1,n) #i = array([1,...,n-1])
11     b = i / sqrt(4*i**2 - 1)
12     # now we generate the matrix; it is symmetric
13     J = diag(b, -1) + diag(b, 1)
14     # in order to find the eigenvalues we can use eigh since J is symmetric
15     x, ev = eigh(J)
16     # finally, we apply the formula for the weights
17     w = 2 * ev[0,:]**2
18     return x, w

```

Die Gauss'sche Quadraturformel lässt sich allerdings nicht immer anwenden, da bei bestimmten Problemen die Funktionswerte lediglich an bestimmten Punkten gegeben sind und die zugrunde liegende Funktion nicht bekannt ist. Daher benutzt man in bestimmten Fällen andere Quadraturformeln.

### Clenshaw-Curtis Formeln

Die Quadraturformeln von Clenshaw-Curtis, bzw. Fejer, basieren auf der Idee der Approximation des Integranden mittels Chebyshev-Polynomen. Für die Integration beliebiger Funktionen wird die Variablensubstitution  $x = \cos(\theta)$  verwendet. Hieraus folgt:

$$\int_{-1}^1 f(x) dx = \int_0^\pi f(\cos(\theta)) \sin(\theta) d\theta.$$

Wir definieren  $F(\theta) := f(\cos(\theta))$ . Da  $F(\theta)$   $2\pi$ -periodisch und gerade ist, lässt es

sich in eine Kosinus-Reihe entwickeln:  $F(\theta) = \sum_{k=0}^{\infty} a_k \cos(k\theta)$ , woraus folgt:

$$\int_0^{\pi} F(\theta) \sin(\theta) d\theta = \int_0^{\pi} \sum_{k=0}^{\infty} a_k \cos(k\theta) \sin(\theta) d\theta = \sum_{k=0}^{\infty} a_k \int_0^{\pi} \cos(k\theta) \sin(\theta) d\theta = \sum_{k \text{ gerade}} \frac{2a_k}{1-k^2}. \quad (1.3.7)$$

Die Koeffizienten  $a_k$  lassen sich mittels FFT <sup>3</sup> oder DCT <sup>4</sup> berechnen.

Für die Stützstellen der Clenshaw-Curtis Quadraturformel gilt der folgende Satz, den wir hier ohne Beweis angeben:

**Theorem 1.3.4.** Die Stützstellen der Clenshaw-Curtis Quadraturformel sind die Extreme der Chebyshev-Polynome und die Nullstellen der Chebyshev-Polynome 2-ter Art.

Da hinter dieser Quadraturformel die Fourier-Transformation (für die Theorie) mit ihrem schnellen Algorithmus (FFT) steht, ist diese Formel sehr effizient (kurze Rechenzeit und sehr gute Genauigkeit) für glatte Funktionen  $f$ .

#### Code 1.3.5: Clenshaw-Curtis: direkte Implementierung

```

def cc2(func,a,b,N):
    """
        Clenshaw-Curtis quadrature rule
        by FFT with the function values
    """
    # first: change of variable with adaptation of the interval
    bma = 0.5*(b-a)
    x = np.cos(np.pi * np.linspace(0,N,N+1)/N) # Chebyshev-abszissa
    x *= bma
    x += 0.5*(a+b)
    fx = func(x) # F(theta) = f(cos(theta)) = f(x)
    vx = np.hstack((fx,fx[-2:0:-1])) # extend by reflection to 2π-periodic
    # note: hstack generates a new array consistig of two former arrays put
    together.
    # the index [-2:0:-1] allows to take all elements of an array, except the
    first one, in reverse order
    # now we perform the FFT to find the coefficients.
    g = np.real(np.fft.fft(vx))/(2.0*N) # we have to divide by the length, which
    is 2N after reflexion
    A = np.zeros(N+1) #A is an array with the coeffs of cosines
    A[0] = g[0]; A[N] = g[N]
    A[1:N] = g[1:N] + np.flipud(g[N+1:]) # put together terms to get coeffs of
    cosines
    w = 0.*x
    w[:,2] = 2./(1.-np.r_[N+1:2]**2) #directly apply the C.Curtis quadrature
    formula
    # now we can return the final value of the integral; the dot product is a
    clever way of writing; otherwise we could have opted for a for-cycle
    return np.dot(w,A)*bma

```

<sup>3</sup>fast Fourier transform

<sup>4</sup>discrete cosine transform

Code 1.3.6: Clenshaw-Curtis: Gewichte und Knoten

```

def cc1(func, a, b, N):
    """
        Clenshaw-Curtis quadrature rule
        by constructing the points and the weights
    """
    bma = b-a
    c = np.zeros((2,2*(N-1) ))
    c[0,0] = 2.0
    c[1,1] = 1.0
    c[1,-1] = 1.0
    for i in np.arange(2,N,2):
        val = 2.0/(1-i**2) #compare with C.Curtis formula
        c[0,i] = val
        c[0,2*(N-1)-i] = val

    f = np.real(np.fft.ifft(c))
    w = f[0,:N]; w[0] *= 0.5; w[-1] *= 0.5 # weights
    x = 0.5*((b+a)+(N-1)*bma*f[1,:N]) # points
    return np.dot(w,func(x))*bma

```

**Beispiel 1.3.7.** Wir untersuchen experimentell mit den folgenden Codes den Fehler bei (nicht-zusammengesetzter) Quadratur.

Code 1.3.8: important polynomial quadrature rules

```

from gaussquad import gaussquad
from numpy import *

def numquad(f,a,b,N,mode='equidistant'):
    """Numerical quadrature on [a,b] by polynomial quadrature formula
        f -> function to be integrated (handle)
        a,b -> integration interval [a,b] (endpoints included)
        N -> Maximal degree of polynomial
        mode (equidistant, Chebychev = Clenshaw-Curtis, Gauss) selects
        quadrature rule
    """
    # use a dictionary as "switch" statement:
    quadrule = {'gauss':quad_gauss, 'equidistant':quad_equidistant,
'chebychev':quad_chebychev}
    nvals = list(range(1,N+1)); res = []
    try:
        for n in nvals:
            res.append(quadrule[mode.lower()](f,a,b,n))
    except KeyError:
        print(mode, " is an invalid quadrature type!")
    else:
        return (nvals,res)

def quad_gauss(f,a,b,deg):
    # get Gauss points for [-1,1]
    [gx,w] = gaussquad(deg);

```

```

# transform to [a,b]
26 x = 0.5*(b-a)*gx+0.5*(a+b)
    y = f(x)
28 return 0.5*(b-a)*dot(w,y) #usual compact notation using dot product
    """"#equivalent notation:
30     for k in range()
        """"
32 def quad_equidistant(f,a,b,deg):
    p = arange(deg+1.0,0.0,-1.0)
34 w = ((b**p) - (a**p))/p #a more "refined" way of writing: power(a,p) etc.
    x = linspace(a,b,deg+1) #for a polynomial of degree n we need n+1
interpolation points
36 y = f(x)
    # 'Quick and dirty' implementation through polynomial interpolation
38 # we know the exact value of the integral for each xn, which we stored in w.
    # now with polyfit we find the coefficient of xn (for each n up to deg),
which we multiply with the corresponding w
40 poly = polyfit(x,y,deg)
    # finally, we sum everything together and obtain an approximation for the
value of the integral
42 return dot(w,poly)

44 def quad_chebychev(f,a,b,deg):
    p = arange(deg+1.0,0.0,-1.0)
46 w = (b**p - a**p)/p
    x = 0.5*(b-a)*cos((arange(0,deg+1)+0.5)/(deg+1)*pi)+0.5*(a+b);
48 y = f(x)
    # 'Quick and dirty' implementation through polynomial interpolation

50 poly = polyfit(x,y,deg)
52 return dot(w,poly)

```

Code 1.3.9: tracking errors on quadrature rules

```

1 from numquad import numquad
import matplotlib.pyplot as plt
3 from numpy import *

5 def numquaderrors():
    """"Numerical quadrature on [0,1]""""
7     N = 20;

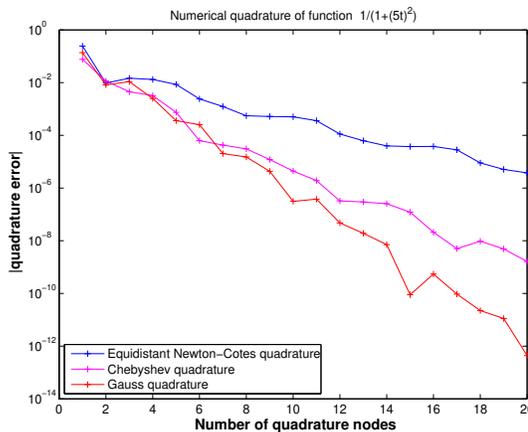
9     plt.figure()
    exact = arctan(5)/5;
11     f = lambda x: 1./(1+power(5.0*x,2))
    nvals,eqdres = numquad(f,0,1,N,'equidistant')
13     nvals, chbres = numquad(f,0,1,N,'chebychev')
    nvals,gaures = numquad(f,0,1,N,'gauss')
15     plt.semilogy(nvals,abs(eqdres-exact),'b+-',label='Equidistant Newton-Cotes
quadrature')
    plt.semilogy(nvals,abs(chbres-exact),'m+-',label='Clenshaw-Curtis
quadrature')
17     plt.semilogy(nvals,abs(gaures-exact),'r+-',label='Gauss quadrature')
    plt.title('Numerical quadrature of function 1/(1+(5t)^2)')

```

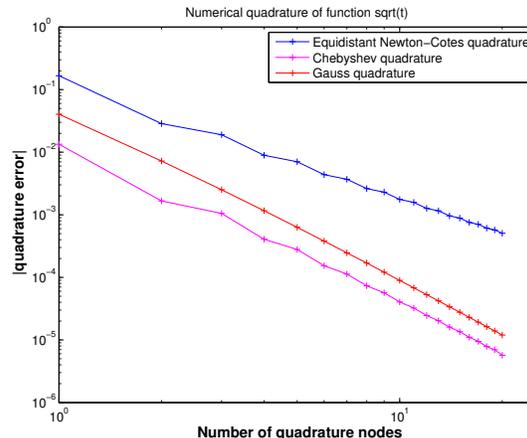
```

19 plt.xlabel('f Number of quadrature nodes')
20 plt.ylabel('f |quadrature error|')
21 plt.legend(loc="lower left")
22 plt.show()
23
24 plt.figure()
25 exact = array(2./3.);
26 f = lambda x: sqrt(x)
27 nvals,eqdres = numquad(f,0,1,N,'equidistant')
28 nvals, chbres = numquad(f,0,1,N,'chebychev')
29 nvals,gaures = numquad(f,0,1,N,'gauss')
30 plt.loglog(nvals,abs(eqdres-exact),'b+-',label='Equidistant Newton-Cotes
quadrature')
31 plt.loglog(nvals,abs(chbres-exact),'m+-',label='Clenshaw-Curtis quadrature')
32 plt.loglog(nvals,abs(gaures-exact),'r+-',label='Gauss quadrature')
33 plt.axis([1, 25, 0.000001, 1]);
34 plt.title('Numerical quadrature of function sqrt(t)')
35 plt.xlabel('f Number of quadrature nodes')
36 plt.ylabel('f |quadrature error|')
37 plt.legend(loc="lower left")
38 plt.show()
39
40 if __name__ == "__main__":
41     numquadrerrors()

```



Fehler für  $f_1(t) := \frac{1}{1+(5t)^2}$  auf  $[0, 1]$



Fehler für  $f_2(t) := \sqrt{t}$  auf  $[0, 1]$

Für Integrale der Form

$$\int_a^b f(x) \omega(x) dx$$

spielen verschiedene orthogonale Polynome eine wesentliche Rolle:

Quadratur	Intervall	Gewichtsfunktion	Polynom	Bezeichnung
Gauss	$(-1, 1)$	1	Legendre	$P_k$
Chebyshev	$(-1, 1)$	$\frac{1}{\sqrt{1-x^2}}$	Chebyshev	$T_k$
Jacobi	$(-1, 1)$	$(1-x)^\alpha(1+x)^\beta; \alpha, \beta > -1$	Jacobi	$P_k^{(\alpha, \beta)}$
Hermite	$\mathbb{R}$	$e^{-x^2}$	Hermite	$H_k$
Laguerre	$(0, \infty)$	$x^\alpha e^{-x}$	Laguerre	$L_k$

Abb. 1.3.10. Gewichtsfunktionen für Quadraturformeln

## 1.4 Adaptive Quadratur

Wie können wir Gitterpunkte definieren, so dass wir einen kleinen Fehler erhalten und gleichzeitig nicht allzu viele Quadraturknoten verwenden?

**Beispiel 1.4.1.** Wir betrachten die zusammengesetzte Quadraturformel auf dem Gitter  $\mathcal{M} := \{a = x_0 < x_1 < \dots < x_m = b\}$ .

Für den lokalen Fehler gilt (für  $f \in C^2([a, b])$ ):

$$\left| \int_{x_k}^{x_{k+1}} f(t) dt - \frac{f(x_k) + f(x_{k+1})}{2} (x_{k+1} - x_k) \right| \leq (x_{k+1} - x_k)^3 \|f''\|_{L^\infty([x_k, x_{k+1}])}$$

Deswegen ist es sinnvoll, das Gitter nur dort zu verfeinern, wo  $|f''|$  gross ist.

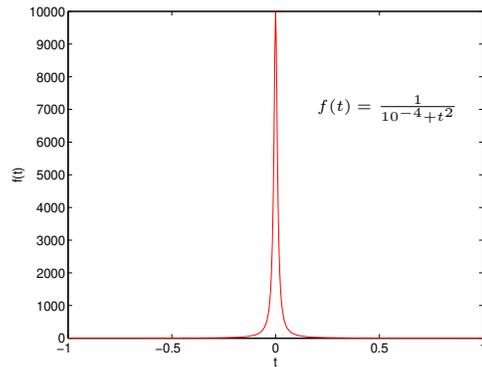


Figure 1.1: Beispiel einer Funktion, dessen Integrals mit kürzeren Intervallen bei  $x = 0$  und mit längeren Intervallen anderswo ausgewertet werden sollte.

Das Ziel der adaptiven Quadratur ist es, eine solche Unterteilung des Integrationsgebietes zu erzielen, dass die Fehler auf den einzelnen Subintervallen gleichermassen zum Gesamtfehler beitragen. Dazu müssen wir den lokalen Fehler schätzen. Dann unterteilen wir nur diejenigen Subintervalle weiter, die einen grossen Fehler aufweisen. Nun: wie können den lokalen Fehler schätzen? Wir wenden zwei Quadraturformeln an, eine von denen kleinere Fehler als die andere liefert. Wir können zum Beispiel die Trapez-Regel und die Simpson-Regel verwenden.

Wenn wir die Trapez-Regel auf das Intervall  $[x_j, x_{j+1}]$  der Länge  $h$  anwenden, haben wir

$$\int_{x_j}^{x_{j+1}} f(t) dt = Q_1(f, x_j, x_{j+1}) + \mathcal{O}(h^3).$$

Für die Simpson-Regel ergibt sich der genauere Wert:

$$\int_{x_j}^{x_{j+1}} f(t) dt = Q_2(f, x_j, x_{j+1}) + \mathcal{O}(h^5).$$

Für kleines  $h$  können wir den exakten Wert des Integrals durch  $Q_2(f, x_j, x_{j+1})$  ersetzen, um eine grobe Schätzung des lokalen Fehlers bei der Verwendung der Trapezregel zu erhalten:

$$\left| \int_{x_j}^{x_{j+1}} f(t) dt - Q_1(f, x_j, x_{j+1}) \right| \approx |Q_2(f, x_j, x_{j+1}) - Q_1(f, x_j, x_{j+1})|.$$

Ein grosser Wert hier ist ein Hinweis darauf, dass die Funktion  $f$  stark variiert, so dass es sich eventuell lohnt, das Intervall  $[x_j, x_{j+1}]$  weiter zu unterteilen. Der Preis für diese Information ist nur die Auswertung von  $f$  im Mittelpunkt  $(x_j + x_{j+1})/2$ , die bei einer weiteren Unterteilung direkt wiederverwertet werden kann. In der Praxis verwenden wir natürlich die genauere Quadraturformel für das Endergebnis; die ungenauere Formel dient hier nur als Mittel für die Fehlerschätzung. Nach diesem Prinzip sind die professionellen Codes für die Quadratur aufgebaut; der nächste Code zeigt, wie das konkret funktioniert.

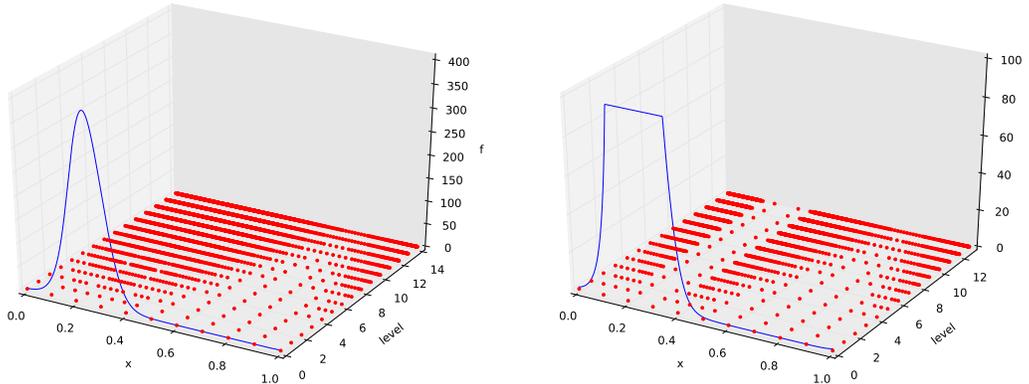
Code 1.4.2:  $h$ -adaptive Quadratur

```

2 from numpy import *
3 from scipy import integrate
4
5 def adaptquad(f,M,rtol,abstol):
6     """
7     adaptive quadrature using trapezoid and simpson rules
8     Arguments:
9     f        handle to function f
10    M        initial mesh
11    rtol     relative tolerance for termination
12    abstol  absolute tolerance for termination, necessary in case the exact
13    integral value = 0, which renders a relative tolerance meaningless.
14    """
15    h = diff(M)                # compute lengths of mesh intervals
16    mp = 0.5*( M[:-1]+M[1:] ) # compute midpoint positions
17    fx = f(M); fm = f(mp)     # evaluate function at positions and
18    midpoints                  # midpoints
19    trp_loc = h*( fx[:-1]+2*fm+fx[1:] )/4 # local trapezoid rule
20    simp_loc = h*( fx[:-1]+4*fm+fx[1:] )/6 # local simpson rule
21    I = sum(simp_loc)          # use simpson rule value as
22    intermediate approximation for integral value
23    est_loc = abs(simp_loc - trp_loc)     # difference of values obtained from
24    local composite trapezoidal rule and local simpson rule is used as an estimate
25    for the local quadrature error.
26    err_tot = sum(est_loc)           # estimate for global error (sum
27    moduli of local error contributions)
28    # if estimated total error not below relative or absolute threshold, refine
29    mesh
30    if err_tot > rtol*abs(I) and err_tot > abstol:
31        refcells = nonzero( est_loc > 0.9*sum(est_loc)/size(est_loc) )[0]
32        I = adaptquad(f,sort(append(M,mp[refcells])),rtol,abstol) # add
33        midpoints of intervalls with large error contributions, recurse.
34    return I
35
36 if __name__ == '__main__':
37     f = lambda x: exp(6*sin(2*pi*x))
38     #f = lambda x: 1.0/(1e-4+x*x)
39     M = arange(11.)/10 # 0, 0.1, ... 0.9, 1
40     rtol = 1e-6; abstol = 1e-10
41     I = adaptquad(f,M,rtol,abstol)
42     exact,e = integrate.quad(f,M[0],M[-1])
43     print('adaptquad:',I, "exact:",exact)
44     print('error:',abs(I-exact))

```

Für die Funktionen  $f(x) = \exp(6 * \sin(2 * \pi * x))$  und  $\min(f(x), 100)$  ergibt diese Art der Adaptivität die folgende Gitter:



**Bemerkung 1.4.3.** (Adaptive Quadratur in Python)

`scipy.integrate.quad` realisiert eine adaptive Quadratur und ist nur ein Wrapper zur Fortran-Bibliothek QUADPACK, siehe `scipy.integrate.explain_quad()` für Details; `scipy.integrate.quadrature` steht für adaptive Gauss'Quadratur.

## 1.5 Quadratur in $\mathbb{R}^d$ und dünne Gitter

Bisher haben wir lediglich eindimensionale Quadraturprobleme betrachtet. Insbesondere in physikalischen Anwendungen sind mehrdimensionale Integrale allgegenwärtig, auch über zwei und drei Dimensionen hinaus. In diesem Abschnitt werden wir uns mit der Effizienz von solchen Quadraturformeln beschäftigen. Insbesondere wird unser Ziel sein, ein Verfahren zu finden, das auch für grosse Dimensionen noch vernünftige Ergebnisse liefert.

### Naive Herangehensweise: Iteration von eindimensionalen Integralen

Betrachte zuerst das zweidimensionale Integral  $I$ :

$$I = \int_a^b \int_c^d f(x, y) dx dy = \int_a^b F(y) dy.$$

Für  $F$  gilt:

$$F(y) = \int_c^d f(x, y) dx \approx \sum_{j_1=1}^{n_1} \omega_{j_1}^1 f(c_{j_1}^1, y),$$

wobei hier die Gewichte und Stützstellen einer geeigneten eindimensionalen Quadraturformel entsprechen. Eben diese verwenden wir für die zweite Integration:

$$I = \sum_{j_2=1}^{n_1} \sum_{j_2=1}^{n_2} \omega_{j_1}^1 \omega_{j_2}^2 f(c_{j_1}^1, c_{j_2}^2).$$

Wir erhalten die sogenannte Tensor-Product-Quadratur: gegeben die 1-dimensionale Quadraturformeln:

$$(w_{j_k}^k, c_{j_k}^k)_{1 \leq j_k \leq n_k}, \quad k = 1, \dots, d,$$

ist das  $d$ -dimensionale Integral approximiert durch

$$I \approx \sum_{j_1=1}^{n_1} \dots \sum_{j_d=1}^{n_d} w_{j_1}^1 \dots w_{j_d}^d f(c_{j_1}^1, \dots, c_{j_d}^d).$$

Wir bemerken, dass die Funktion in  $d$  Dimensionen an  $n^d$  Stützstellen ausgewertet werden muss. Dies ist insbesondere problematisch, da  $f$  eventuell nur als teure Funktionsroutine verfügbar ist. Es sei ebenfalls angemerkt, dass für die Konvergenzrate gilt:  $K = O(N^{-r/d})$ , wobei  $r$  die Konvergenzrate der eindimensionalen Quadraturformel ist. Deswegen wird die Konvergenz schlechter mit wachsender Anzahl an Dimensionen.

### Quadratur auf dünne Gitter

Betrachten wir zunächst eine eindimensionale Quadraturformel:

$$I = \int_a^b f(x) dx \approx Q_n^1(f).$$

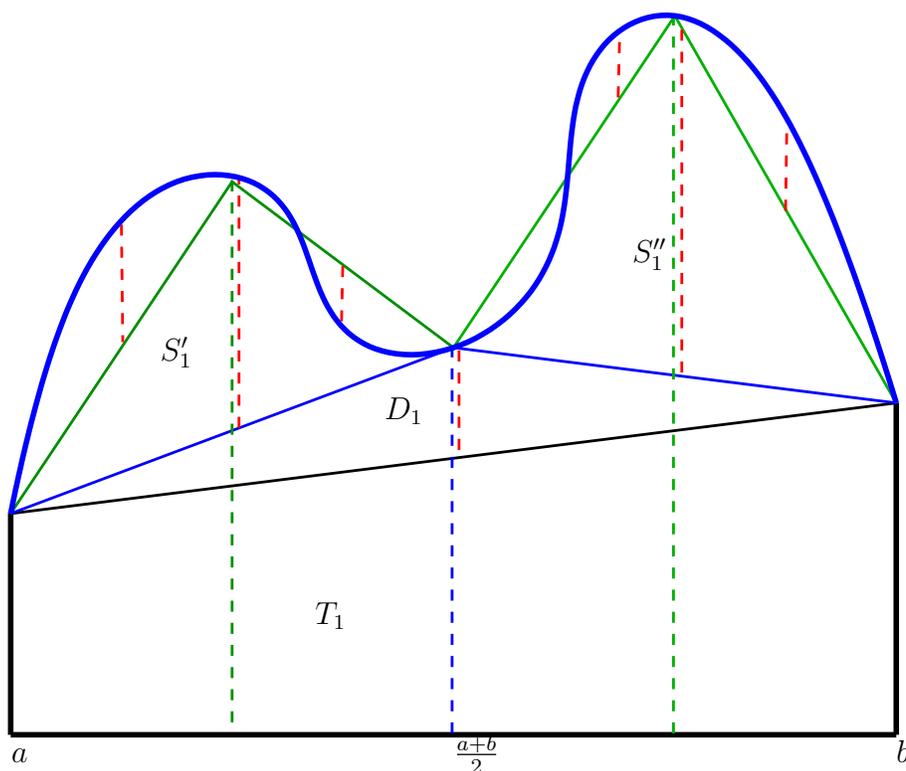
Durch das Hinzunehmen von Punkten verbessert sich das Resultat zunächst stark, dann jedoch immer weniger, d.h.  $Q_{n+1} - Q_n$  wird mit zunehmendem  $n$  abnehmen. Hier ein Beispiel mit der Trapezregel:

$$T_1(f, a, b) = \frac{f(a) + f(b)}{2} (b - a).$$

Wir schreiben den Fehler als:

$$S_1(f, a, b) = \int_a^b f(x) dx - T_1(f, a, b).$$

Graphisch lässt sich  $S_1$  anhand von Theorem 1.5.1 veranschaulichen.



**Abb. 1.5.1.** Hierarchische Struktur bei der Trapez-Regel

Hier entspricht  $S_1$  der Fläche, die vom Trapez  $T_1$  nicht ausgefüllt wird. Offensichtlich ist diese erste Näherung sehr schlecht, da wir nur die Punkte  $f(a)$  und  $f(b)$  verwendet haben. Wir erwarten, dass unter Hinzunahme der Information am Mittelpunkt  $f((a+b)/2)$  die Approximation besser wird und schreiben:

$$T_2(f, a, b) = \frac{f(a) + f((a+b)/2)}{2} + \frac{f((a+b)/2) + f(b)}{2}.$$

Bemerkung: in der Formel für  $T_2$  verwendet man nicht das bereits berechnete Wert  $T_1$ , man verwirft also bereits gesammelte Information. Das muss nicht sein. Betrachten wir wieder Theorem 1.5.1. Die Differenz zwischen  $T_1$  und  $T_2$  ist gerade das Dreieck  $D_1$ :

$$T_2(f, a, b) = T_1(f, a, b) + D_1(f, a, b),$$

wobei  $D_1$ , wie der Graph zeigt, ergibt sich als:

$$D_1(f, a, b) = \left( f\left(\frac{a+b}{2}\right) - \frac{f(a) + f(b)}{2} \right) \frac{b-a}{2} = g_1(f, a, b) \frac{b-a}{2}.$$

Die Quantität  $g_1(f, a, b) = f\left(\frac{a+b}{2}\right) - \frac{f(a)+f(b)}{2}$  heisst *hierarchischer Überschuss* der Funktion  $f$  auf dem Intervall  $[a, b]$ . Der Fehler ist nun die Summe der Flächen  $S_1'$  und  $S_1''$ :

$$S_1'\left(f, a, \frac{a+b}{2}\right) + S_1''\left(f, \frac{a+b}{2}, b\right),$$

und er hat sich durch die Hinzunahme der Auswertung im Mittelpunkt  $\frac{a+b}{2}$  genau um die Fläche vom Dreieck  $D_1$  reduziert.

Wir erwarten kleinere Terme  $S'_1, S_2''$  und somit auch hierarchische Überschüsse, wenn diese Prozedur iteriert wird (wie in 1.5.1 zu sehen ist).

Wir betrachten nun das Intervall  $[0, 1]$ , welches in  $2^l$  Teilintervalle  $(\frac{k}{2^l}, \frac{k+1}{2^l})$ , mit  $k = 0, 1, \dots, 2^l - 1$  zerfällt. Wir nennen  $l$  das *Level* und bezeichnen mit  $Q_l^1$  eine klassische Quadraturformel auf jedem Teilintervall. Typische Beispiele hierfür sind die Trapez- und die Mittelpunktsregel; die Clenshaw-Curtis- und die Gaussformeln haben nicht äquidistante Stützstellenverteilungen, können aber auch verwendet werden. Die Prozedur der hierarchischen Überflüsse lässt sich als Teleskopsumme formulieren:

$$\begin{aligned} Q_l^1 f &= Q_0^1 f + (Q_1^1 f - Q_0^1 f) + (Q_2^1 f - Q_1^1 f) + \dots + (Q_l^1 f - Q_{l-1}^1 f) \\ &= Q_0^1 f + \Delta_1^1 f + \Delta_2^1 f + \dots + \Delta_l^1 f. \end{aligned}$$

In einer Dimension vereinfacht diese Umformulierung keinerlei das Problem; in  $d$  Dimensionen gibt es jedoch einen fundamentalen Unterschied. Das Tensorprodukt der Level  $l = (l_1, \dots, l_d)$  ist:

$$\begin{aligned} Q_\ell^d &= Q_{\ell_1}^1 \otimes \dots \otimes Q_{\ell_d}^1 \\ &= \sum_{j_1=1}^{N_1} \dots \sum_{j_d=1}^{N_d} w_{j_1}^1 \dots w_{j_d}^d f(c_{j_1}^1, \dots, c_{j_d}^d) \\ &= \sum_{j=1}^d \sum_{1 \leq k_j \leq \ell_j} (\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1) f. \end{aligned}$$

Im Falle eines isotropischen Gitters  $\ell = (\ell, \dots, \ell)$  notieren wir

$$Q_\ell^d = \sum_{j=1}^d \sum_{1 \leq k_j \leq \ell} (\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1) f = \sum_{|\mathbf{k}|_\infty \leq \ell} (\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1) f.$$

Wenn  $f$  glatt ist, sind die Details  $(\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1) f$  so klein, dass sie vernachlässigt sein können. Alle Details, die ungefähr gleich gross sind, lassen wir auf ein Mal weg.

Die klassische Dünngitter-Quadraturformel (Smolyak) ist dann definiert als:

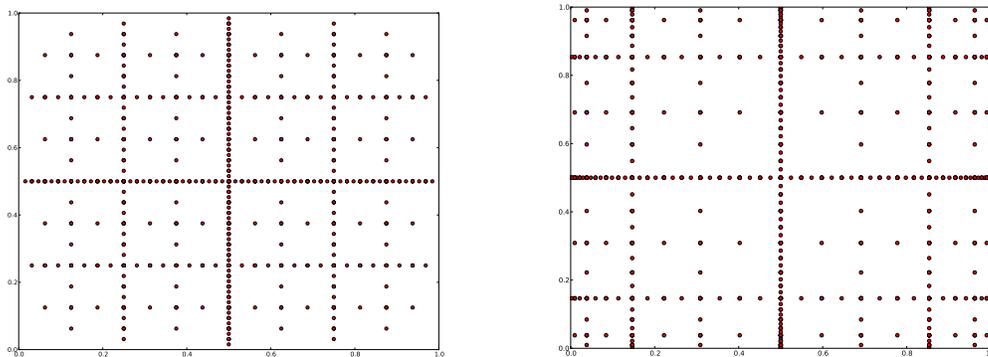
$$S_\ell^d f := \sum_{|\mathbf{k}|_1 \leq \ell + d - 1} (\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1) f.$$

Man kann dazu die Kombinationsformel beweisen:

$$S_\ell^d f = \sum_{\ell \leq |\mathbf{k}|_1 \leq \ell + d - 1} (-1)^{\ell + d - |\mathbf{k}|_1 - 1} \binom{d-1}{|\mathbf{k}|_1 - \ell} (Q_{k_1}^1 \otimes \dots \otimes Q_{k_d}^1) f,$$

welche in praktischen Implementierungen verwendet wird.

Das dünne Gitter entsteht dann aus der Vereinigung von anisotropischen vollen Gittern, die in der Kombinationsformel auftreten. Es sind  $N = O(2^\ell \ell^{d-1})$  Punkte, was deutlich weniger als die  $O(2^{d\ell})$  Punkten eines vollen Gitters sind. Der Fehler dieser klassischen Dünngitter-Quadratur ist  $O(N^{-r} \log^{(d-1)(r+1)}(N))$  für eine Funktion  $f$  mit einer gewissen Glattheit  $r$ .



**Abb. 1.5.2.** Dünne Gitter, die auf der Mittelpunktsregel (links) und offener Clenshaw-Curtis-Regel (rechts) basiert sind.

Die Clenshaw-Curtis-Regel auf ein volles Gitter in  $d = 3$  Dimensionen auf dem Level  $n = 6$  verwendet 274625 Funktionsauswertungen und braucht zirka 6.6 Sekunden auf meinem Laptop (im Jahre 2011). Dieselbe Quadraturformel aber auf dem dünnen Gitter auf dem Level  $n = 6$  braucht nur 3120 Funktionsauswertungen und 0.1 Sekunden. Der Fehler bei Verwendung des vollen Gitters ist  $1.5 \cdot 10^{-5}$  während in der Dünngitter-Quadratur ist er  $1.5 \cdot 10^{-6}$ , für den Integrand  $f(\mathbf{x}) = (1 + \frac{1}{d})^d \sqrt{x_1 \cdots x_d}$ .

Dasselbe Beispiel in  $d = 4$  Dimensionen führt bei der Verwendung eines vollen Gitters zirka 17.8 Millionen Funktionsauswertungen aus, hat einen Fehler von  $3.3 \cdot 10^{-5}$  und braucht 426 Sekunden, während im Falle eines dünnen Gitters braucht man 9065 Funktionsauswertungen in 0.3 Sekunden mit dem Fehler  $8 \cdot 10^{-6}$ .

Die Kombinationsformel lässt sich einfach auf parallele Rechnerarchitekturen implementieren. Für  $d = 10$  Dimensionen auf dem Level  $n = 6$  braucht dieser parallele Code nur 33 Sekunden und hat einen Fehler von  $3 \cdot 10^{-5}$ . Die Kombinationsformel ist in diesem Falle eine Summe mit 3003 Termen.

Mehr über die dünnen Gittern findet man in den Büchern:

M. HOLTZ, *Sparse grid quadrature in high dimensions with applications in finance and insurance*, Berlin, Springer 2011, [NEBIS 009878866](#)

J. GARCKE, *Sparse grids and applications*, Heidelberg, Springer 2013, [NEBIS 009895238](#)

## 1.6 Monte-Carlo Quadratur

### Grundrezept.

Manchmal ist es nicht klar, welche Quadratur man anwenden sollte; es kann auch sein, dass eine Quadratur eine zu lange Laufzeit bräuchte. In solchen Fällen verwendet man die Montecarlo-Quadratur. Oft wird sie benutzt, um eine erste Näherung des Wertes eines Integrals zu bekommen. Diese Technik wird insbesondere für hochdimensionale Probleme oder bei sehr unglatten Integranden verwendet.

Wir nehmen  $N$  Zahlen ( $N$  ist normalerweise eine grosse Zahl)  $t_i \in [0, 1]$  zufällig aus der uniformen Verteilung auf  $[0, 1]$ , (wir werden das später genauer sehen; hier können uns vorstellen, dass, bei der Ziehung der Zahlen  $t_i$ , keine Gebiete aus  $[0, 1]$  bevorzugt werden) und wir mitteln die Funktionswerte in diesem Intervall:

$$I = \int_0^1 z(t) dt \approx \frac{1}{N} \sum_{i=1}^N z(t_i).$$

Die Knoten  $t_i$  werden mithilfe eines Zufallszahlengenerators erzeugt. Wenn wir auf dem Intervall  $[a, b]$  statt  $[0, 1]$  arbeiten können wir folgendermassen vorgehen:

$$I = \int_a^b z(s) ds = |b-a| \int_0^1 z(a + (b-a)t) dt = |b-a| \langle f \rangle \approx |b-a| \frac{1}{N} \sum_{i=1}^N z(s_i) =: I_N, \quad (1.6.8)$$

wobei  $s_i = a + (b-a) \cdot t_i$ .

Der vorherige Wert ist ein Mittelwert. Da wir nun mit Zufallszahlen arbeiten, ist in jedem Experiment ein *verschiedener* Mittelwert zu erwarten. Was hier Konvergenz bedeutet, müssen wir noch klarstellen, siehe dazu Theorem 1.7.8.

**Bemerkung 1.6.1.** (Eigenschaften der Monte-Carlo-Methode)

- Die Konvergenz vom Monte-Carlo-Verfahren ist sehr langsam, nur  $\mathcal{O}(N^{-\frac{1}{2}})$  (siehe 1.6.25 und 1.7.8); sie hängt aber nicht (direkt) von der Dimension oder von der Glattheit des Integrandes ab!
- Diese Methode ist rein probabilistisch: das Ergebnis kann völlig falsch sein, siehe dazu Figur 1.6.18.

Das Monte-Carlo-Verfahren wird in der Wahrscheinlichkeitsrechnung und Statistik mathematisch begründet. Wir versuchen hier eine intuitive Herangehensweise, die aber so weit wie möglich auf der mathematischen Sprache basiert.

Wir notieren

$$\tilde{\sigma}_N = \sqrt{\frac{\frac{1}{N} \sum_{i=1}^N z(t_i)^2 - \left(\frac{1}{N} \sum_{i=1}^N z(t_i)\right)^2}{N-1}} = \sigma_N / \sqrt{N}. \quad (1.6.9)$$

Dann kann man rigoros beweisen (Central Limit Theorem): wenn wir dieses Experiment sehr oft wiederholen, dann enthält das Intervall

$$[I_N - \tilde{\sigma}_N, I_N + \tilde{\sigma}_N] \quad (1.6.10)$$

den wahren Wert  $\int_{[0,1]^d} z(t)dt$  in circa 68.3% der Fälle, falls die  $N$  Zahlen  $t_i$  aus der uniformen Verteilung auf  $[0, 1]^d$  gezogen worden sind (und  $N$  selber gross ist).

Jede Monte-Carlo-Methode (nicht nur für Quadratur) braucht:

- ein Gebiet für das “Experiment”: hier  $[0, 1]^d$ ;
- Zahlen, die mittels eines (guten) Zufallszahlengenerators erzeugt worden sind: hier sind sie die  $t_i$ ;
- deterministische Berechnungen: hier  $I_N$  aus (1.6.8) und  $\tilde{\sigma}_N$  aus (1.6.10);
- eine Darstellung des Ergebnisses: hier  $P(I \in [I_N - \tilde{\sigma}_N, I_N + \tilde{\sigma}_N]) = 0.683$ .

**Bemerkung 1.6.2.** Die Dimension befindet sich hier nur in der Anzahl  $N$  der generierten Zufallszahlen. Die Länge des Intervalls nimmt wie  $1/\sqrt{N}$  ab. Eine Verbesserung der Methode kann nur durch eine Verkleinerung der Varianz  $\sigma_N^2$  erzielt werden. Das wird uns bald zu Techniken zur Reduktion der Varianz führen.

### Die Theorie dahinten

Genauer verstanden werden alle diese Ideen erst nach einem Besuch von Vorlesungen in Wahrscheinlichkeit und Statistik. Hier sind nur einige Begriffe, die wir brauchen. Für uns ist eine Zufallsvariable eine Abbildung  $X : \mathbb{R} \rightarrow \mathbb{R}$ . Im mehrdimensionalen Fall ist  $X : \mathbb{R}^d \rightarrow \mathbb{R}$  und die unteren Definitionen lassen sich einfach verallgemeinern. Achtung: die Zufälligkeit liegt im Argument der Funktion  $X$ .

**Beispiel 1.6.3.** Der Wert der Funktion  $X$  sei die Anzahl der Punkte, die beim Werfen eines Würfels erscheint.  $X$  kann also nur Werte in  $\{1, 2, 3, 4, 5, 6\}$  annehmen. Die Wahrscheinlichkeit, dass  $X$  einen bestimmten Wert  $j$ , sagen wir  $j = 5$ , annimmt, ist  $1/6$ . Wir schreiben:

$$P(X = j) = \frac{1}{6}.$$

**Definition 1.6.4.** Die Verteilungsfunktion  $F$  für eine Zufallsvariable  $X$  ist definiert durch

$$F(x) = P(X \leq x).$$

**Beispiel 1.6.5.** Für das Würfelexperiment ist  $P(X \leq 2) = 2/6$ , also  $F(2) = 2/6$ .

**Definition 1.6.6.** Die Zufallsvariable  $X$  heisst (absolut) stetig, falls es eine Funktion  $f$  gibt, so dass

$$F(x) = \int_{-\infty}^x f(t)dt;$$

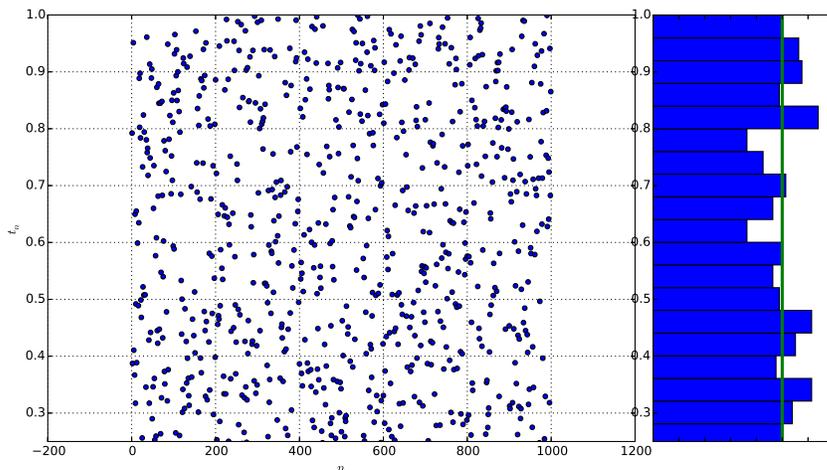
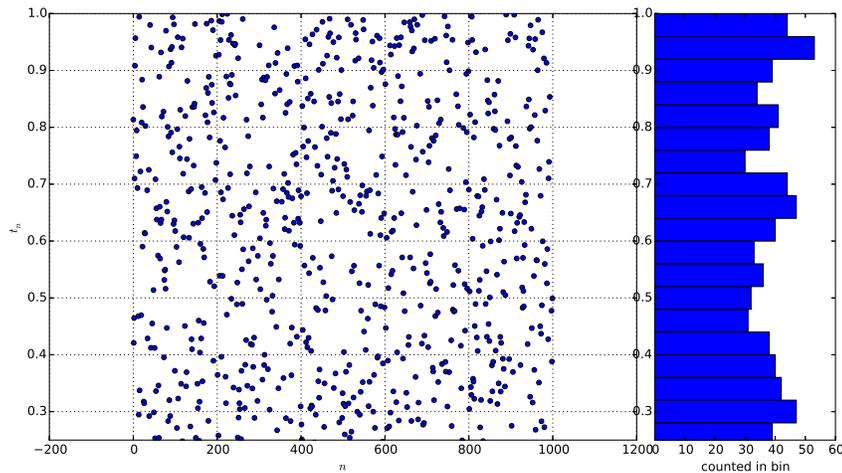
$f$  heisst die Dichte von  $X$ .

Die Dichte von  $X$  zeigt bevorzugte Gebiete für die Werte von  $X$  an:  $f(t)dt = P[X \in [t, t + dt]]$ .

**Beispiel 1.6.7.**  $X$  ist *uniform verteilt* in  $[a, b]$  und man schreibt  $X \sim \mathcal{U}[0, 1]$ , wenn die Dichte von  $X$

$$f(x) = \begin{cases} \frac{1}{b-a}, & \text{wenn } x \in [a, b], \\ 0, & \text{sonst} \end{cases}$$

ist; die entsprechende Verteilungsfunktion ist dann  $F(x) = \int_a^x \frac{1}{b-a} dx = \frac{x-a}{b-a}$ .

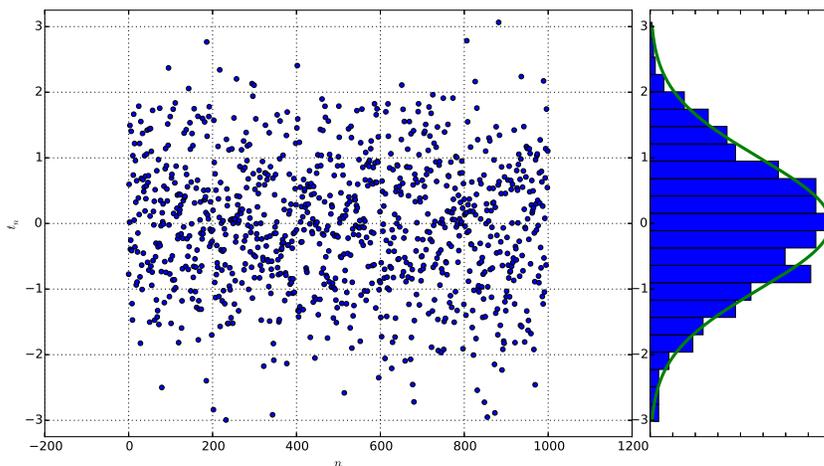
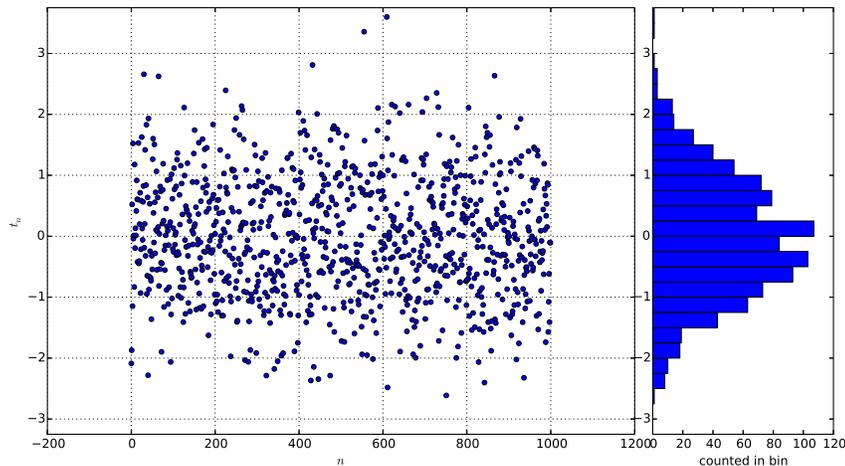


**Abb. 1.6.8.** Aus der uniformen Verteilung gezogene  $N = 1000$  Zufallszahlen; oben: Anzahl im Körbchen, unten: eine andere Ziehung und normiert.

**Beispiel 1.6.9.**  $X$  ist *normalverteilt* mit Erwartungswert  $\mu$  und Varianz  $\sigma^2$  und man schreibt  $X \sim \mathcal{N}(\mu, \sigma^2)$ , wenn die Dichte von  $X$

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

ist. Im Falle wo  $\mu = 0$  und  $\sigma^2 = 1$  redet man von Standardnormalverteilung. Bei einer normalverteilten Zufallsvariablen haben die Zahlen rund um den Erwartungswert  $\mu$  grössere Wahrscheinlichkeit zu erscheinen.



**Abb. 1.6.10.** Aus der normalen Verteilung gezogene  $N = 1000$  Zufallszahlen; oben: Anzahl im Körbchen, unten: eine andere Ziehung und normiert.

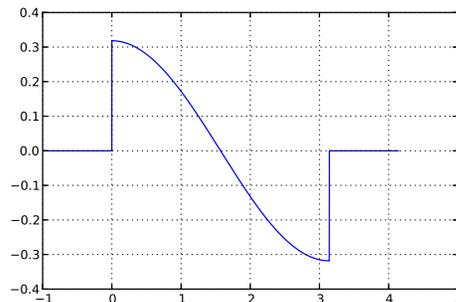
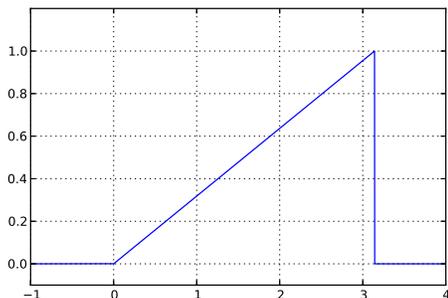
**Definition 1.6.11.** Für eine Zufallsvariable  $X$  mit Dichte  $f$  definieren wir den Erwartungswert von  $X$  als

$$\mathbb{E}(X) = \int_{\mathbb{R}} xf(x)dx$$

und den Erwartungswert von einer Funktion von  $X$   $z(X)$  als

$$\mathbb{E}(z(X)) = \int_{\mathbb{R}} z(x)f(x)dx .$$

**Beispiel 1.6.12.** Wenn  $X \sim \mathcal{U}(0, \pi)$ , dann  $\mathbb{E}(X) = \int_{\mathbb{R}} xf(x)dx = \int_0^{\pi} x \frac{1}{\pi} dx = \frac{\pi}{2}$ .  
 $\mathbb{E}(\cos X) = \int_0^{\pi} \cos x \frac{1}{\pi} dx = 0$ .



**Bemerkung 1.6.13.** Es gilt:

$$\mathbb{E}(az(X)) = a\mathbb{E}(z(X)) \text{ und } \mathbb{E}(z(X) + f(X)) = \mathbb{E}(z(X)) + \mathbb{E}(f(X)) ,$$

d.h.  $\mathbb{E}$  ist ein linearer Operator. Wenn keine zweideutige Bedeutung möglich ist, lassen wir die Klammern weg und schreiben einfach  $\mathbb{E}z(X)$ .

**Definition 1.6.14.** Wir nennen die Varianz der Zufallsvariable  $X$  die erwartete quadratische Abweichung vom Erwartungswert:

$$\mathbb{V}(X) = \mathbb{E}[(X - \mathbb{E}X)^2] = \mathbb{E}(X^2) - (\mathbb{E}X)^2 .$$

**Bemerkung 1.6.15.** Die Varianz ist kein linearer Operator, denn

$$\mathbb{V}(az(X)) = a^2\mathbb{V}(z(X)) ,$$

aber wenn  $X_1$  und  $X_2$  zwei unabhängige Zufallsvariablen sind, dann gilt

$$\mathbb{V}(z(X_1) + z(X_2)) = \mathbb{V}(z(X_1)) + \mathbb{V}(z(X_2)) .$$

Das Monte-Carlo-Verfahren beruht auf der einfachen Bemerkung, dass

$$\int_{[0,1]^d} z(x)dx = \mathbb{E}z(X) \text{ mit } X \sim \mathcal{U}([0, 1]^d) . \tag{1.6.11}$$

Um das Integral  $\int_{[0,1]^d} z(x)dx$  zu approximieren reicht es also aus, eine Schätzung des Erwartungswertes  $\mathbb{E}z(X)$  zu erhalten. Die Monte-Carlo-Methode ist die Kunst, einen Erwartungswert durch den Mittelwert der Funktionswerte simulierter Zufallsvariablen zu approximieren. Es werden stochastische Experimente durchgeführt, die Werte  $x_i$  einer Zufallsvariablen  $X$ , die uniform verteilt auf  $[0, 1]^d$  ist, produzieren; man spricht von “Realisierungen” von  $X$ . Ein Schätzer ergibt dann auf Grund dieser Werte eine Schätzung von  $\mathbb{E}z(X)$ . Die Genauigkeit der Approximation wird von der Varianz  $\mathbb{V}(X)$  beeinflusst. Wenn wir  $N$  Werte verwenden wollen, dann führen wir eben  $N$  voneinander unabhängige stochastische Experimente durch, um die  $N$  Werte  $x_i$  zu produzieren. Wir haben es also eigentlich mit  $N$  unabhängigen Zufallsvariablen  $X_1, \dots, X_N$  zu tun. Diese Zufallsvariablen sind alle auf  $[0, 1]^d$  uniform verteilt. Ein guter Schätzer für den Erwartungswert von  $X$  ist die Zufallsvariable

$$m_N(z(X)) := \frac{1}{N} \sum_{i=1}^N z(X_i) \quad (1.6.12)$$

und eine Schätzung auf Grund der Realisierung  $\mathbf{x} = (x_1, \dots, x_N)$  ist

$$m_N(z(\mathbf{x})) := \frac{1}{N} \sum_{i=1}^N z(x_i) .$$

**Bemerkung 1.6.16.** Es ist einfach zu sehen, dass

$$\mathbb{E}m_N(z(X)) = N \frac{1}{N} \mathbb{E}z(X) = \int_{[0,1]^d} z(x)dx .$$

Wie gut ist die Approximation von  $I = \mathbb{E}z(X)$  durch  $m_N(\mathbf{x})$ ? Das lässt sich nicht so leicht sagen. Ein Vertrauensintervall wird uns zusätzliche Information geben, die uns einen Einblick auf die mögliche (doch nicht tatsächliche) Genauigkeit dieser Approximation ermöglicht. Intuitiv erhält man eine bessere Approximation bei kleinerer Varianz:

$$\mathbb{V}m_N(z(X)) = \mathbb{V}\left(\frac{1}{N} \sum_{i=1}^N z(X_i)\right) = \frac{1}{N^2} N \mathbb{V}(z(X)) = \frac{1}{N} \mathbb{V}(z(X)) \longrightarrow 0 .$$

Die zwei Gesetze der grossen Zahlen beschreiben verschiedene *Konvergenzarten*, die das Monte-Carlo-Verfahren begründen. Der zentrale Grenzwertsatz (Central Limit Theorem) besagt, dass für grosses  $N$ , die Zufallsvariable

$$\frac{m_N(z(X)) - \mathbb{E}z(X)}{\sqrt{\frac{1}{N} \mathbb{V}(z(X))}}$$

sich fast wie eine normalverteilte  $\mathcal{N}(0, 1)$  Zufallsvariable verhält. Mathematisch sagt man, dass die Verteilungsfunktion dieser Zufallsvariable punktweise gegen die

Verteilungsfunktion der  $\mathcal{N}(0, 1)$  Zufallsvariable konvergiert. Daraus folgt mit der Notation  $\sigma(z(X)) = \sqrt{\mathbb{V}(z(X))}$ :

$$P\left(|m_N(z(X)) - \mathbb{E}z(X)| \leq \frac{\lambda\sigma(z(X))}{\sqrt{N}}\right) = \frac{1}{\sqrt{2\pi}} \int_{-\lambda}^{\lambda} e^{-x^2/2} dx + \mathcal{O}\left(\frac{1}{\sqrt{N}}\right).$$

Wir sehen hier bereits die langsame Konvergenz des Monte-Carlo-Verfahrens. Einerseits haben wir eine langsame Konvergenz  $\mathcal{O}(\frac{1}{\sqrt{N}})$  für die Verteilungsfunktionen (punktweise). Der Wert des Integrals  $p(\lambda) = \frac{1}{\sqrt{2\pi}} \int_{-\lambda}^{\lambda} e^{-x^2/2} dx$  hängt nur von der Länge des Integrationsintervalls  $2\lambda$  ab und stellt eine  $\mathcal{O}(\frac{1}{\sqrt{N}})$ -gute Approximation der Wahrscheinlichkeit, dass der Fehler  $|m_N(z(X)) - \mathbb{E}z(X)|$  kleiner als  $\frac{\lambda\sigma(z(X))}{\sqrt{N}}$  ist, dar. Eine Wahrscheinlichkeit  $p(\lambda) = 0.5$  erhalten wir für  $\lambda = 0.6745$ ,  $p(\lambda) = 0.9$  braucht  $\lambda = 1.645$  und  $p(\lambda) = 0.95$  braucht  $\lambda = 1.9605$ . Die Wahl  $\lambda = 1$  ergibt die am Anfang als ‘‘magische Zahl’’ angegebene Wahrscheinlichkeit  $p(1) = 0.683$ . Diese Zahlen sehen anderes aus, wenn  $N$  nicht so gross ist, denn dann musste man mit der Student  $t$ -Verteilung statt mit der normalen Verteilung arbeiten. Hier nehmen wir an, dass  $N$  gross genug ist ( $N \geq 30$ ). Andererseits, die Länge des Vertrauensintervalls nimmt nur wie  $1/\sqrt{N}$  ab. Ein kürzeres Intervall entspricht einem erwarteten kleineren Fehler, es lässt sich aber nur mit erheblich mehr Zufahlszahlen realisieren.

Den Wert  $\sigma(z(X)) = \sqrt{\mathbb{V}(z(X))}$  haben wir hier als bekannt angenommen. Um für eine Realisierung  $\mathbf{x}$  auch ein Vertrauensintervall auf Grund voriger Aussage zu bauen, ohne  $\sigma(z(X))$  exakt zu kennen, brauchen wir zunächst einen Schätzer für  $\mathbb{V}(z(X)) = \mathbb{E}(z(X)^2) - (\mathbb{E}z(X))^2$ . Der Schätzer

$$d_2^*(z(X)) := \frac{1}{N-1} \sum_{j=1}^N (z(X_j) - m_N(z(X)))^2 = \frac{N}{N-1} (m_N(z(X)^2) - (m_N(z(X)))^2)$$

hat die gewünschte Eigenschaft  $\mathbb{E}d_2^*(z(X)) = \mathbb{V}(z(X))$ . Eine Schätzung von  $\sqrt{\mathbb{V}(z(X))}$  ist also

$$\sqrt{\frac{N}{N-1} (m_N(z(\mathbf{x})^2) - (m_N(z(\mathbf{x})))^2)},$$

was die Schätzung (1.6.9)

$$\sqrt{\frac{1}{N-1} (m_N(z(\mathbf{x})^2) - (m_N(z(\mathbf{x})))^2)}$$

für  $\sqrt{\mathbb{V}(z(X))}/\sqrt{N}$  liefert. Somit erhalten wir

**Theorem 1.6.17.** Das Intervall

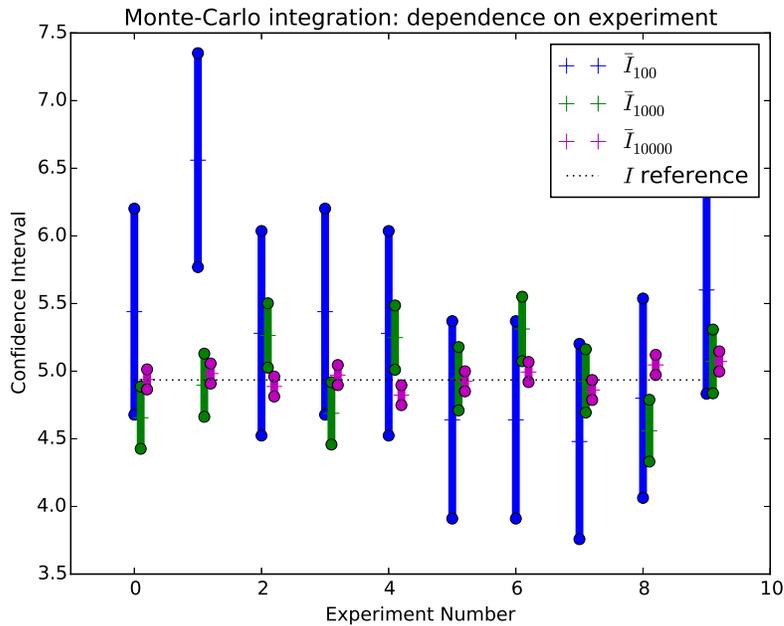
$$[m_n(z(\mathbf{X})) - \lambda d_2^*(z(X))/\sqrt{N}, m_n(z(\mathbf{X})) + \lambda d_2^*(z(X))/\sqrt{N}]$$

enthält den wahren Wert  $I = \mathbb{E}z(X)$  mit der Wahrscheinlichkeit  $p(\lambda) = \frac{1}{\sqrt{2\pi}} \int_{-\lambda}^{\lambda} e^{-x^2/2} dx$  bis auf einen  $\mathcal{O}(\frac{1}{\sqrt{N}})$ -Fehler.

Für das berechnete Vertrauensintervall

$$[m_n(z(\mathbf{x})) - \lambda \tilde{\sigma}_N, m_n(z(\mathbf{x})) + \lambda \tilde{\sigma}_N]$$

kann man in einem Experiment nichts sagen. Das Theorem sagt aber, wenn wir dieses experiment  $M$ -mal (mit  $M$  gross) wiederholen, dann enthält der Vertrauensintervall den Wahren Wert  $I$  in circa  $100p(\lambda)$  Prozent der  $M$  Fälle.



**Abb. 1.6.18.** MC-Approximation und 10 Konfidenzintervalle aus Code [1.6.24](#) für verschiedene Anzahl  $N$  der Realisierungen.

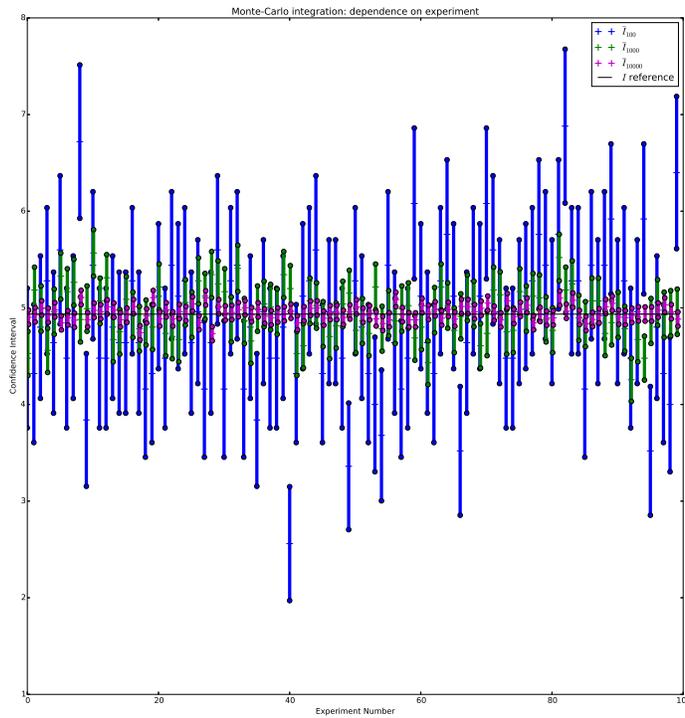
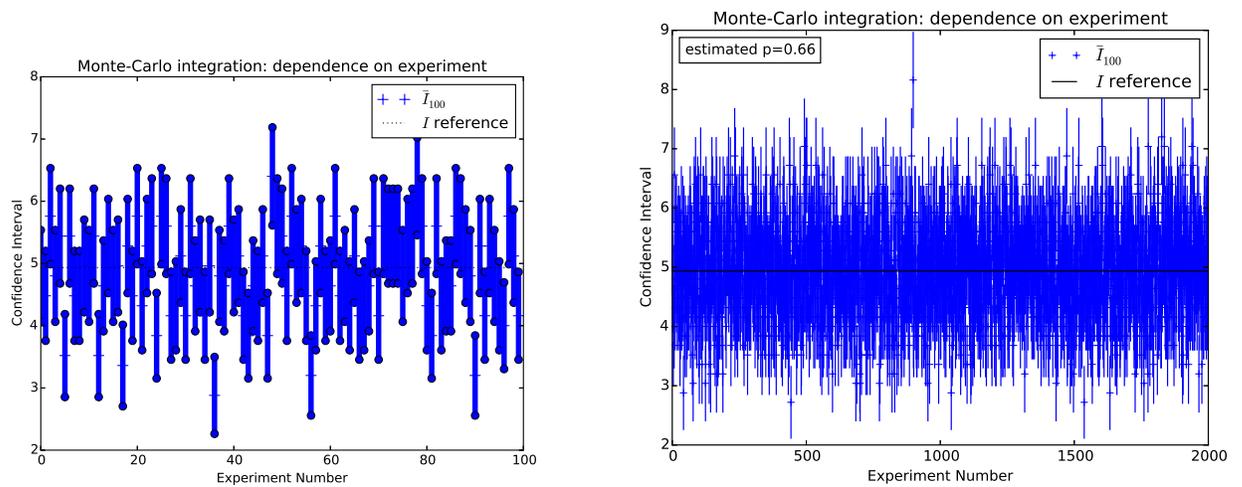
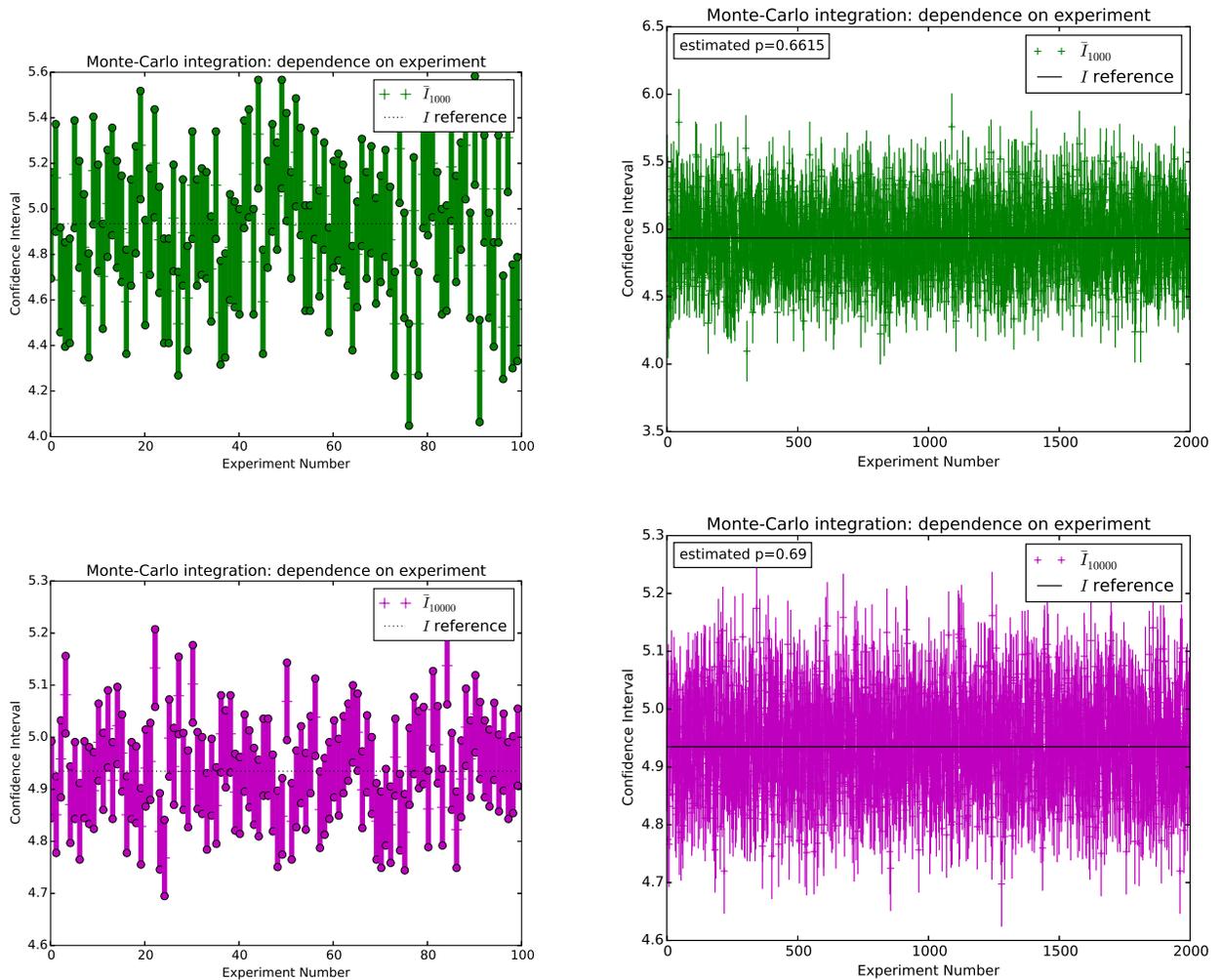


Abb. 1.6.19. MC-Approximation und 100 Konfidenzintervalle aus Code 1.6.24 für verschiedene Anzahl  $N$  der Realisierungen.





**Abb. 1.6.20.** MC-Approximationen und  $M = 100$  (links) bzw.  $M = 2000$  (rechts) Konfidenzintervalle aus Code 1.6.24 für verschiedene Anzahl  $N$  der Realisierungen; bitte beachten Sie die Proportion  $p$  der Intervallen, die den wahren Wert des Integrals enthalten; diese sollte (für grosses  $M$ ) nah am 0.683 sein.

**Bemerkung 1.6.21.** Ein kurzes Vertrauensintervall entspricht einer niedrigen Wahrscheinlichkeit; grosse Wahrscheinlichkeit, dass das Vertrauensintervall den wahren Wert enthält, ist nur für eine grössere Intervalllänge möglich. Im “Grundrezept” werden  $\lambda = 1$  und entsprechend  $p(1) = 0.683$  verwendet.

Gegeben eine akzeptable Wahrscheinlichkeit, bekommt man ein kleineres Vertrauensintervall entweder wenn man ein viel grösseres  $N$  wählt (aber nur  $1/\sqrt{N}$  kleiner) oder wenn man die Berechnungen so organisiert, dass die Varianz  $\mathbb{V}(z(X))$  kleiner wird. Dies werden wir nachher untersuchen, im Kapitel über die Reduktion der Varianz.

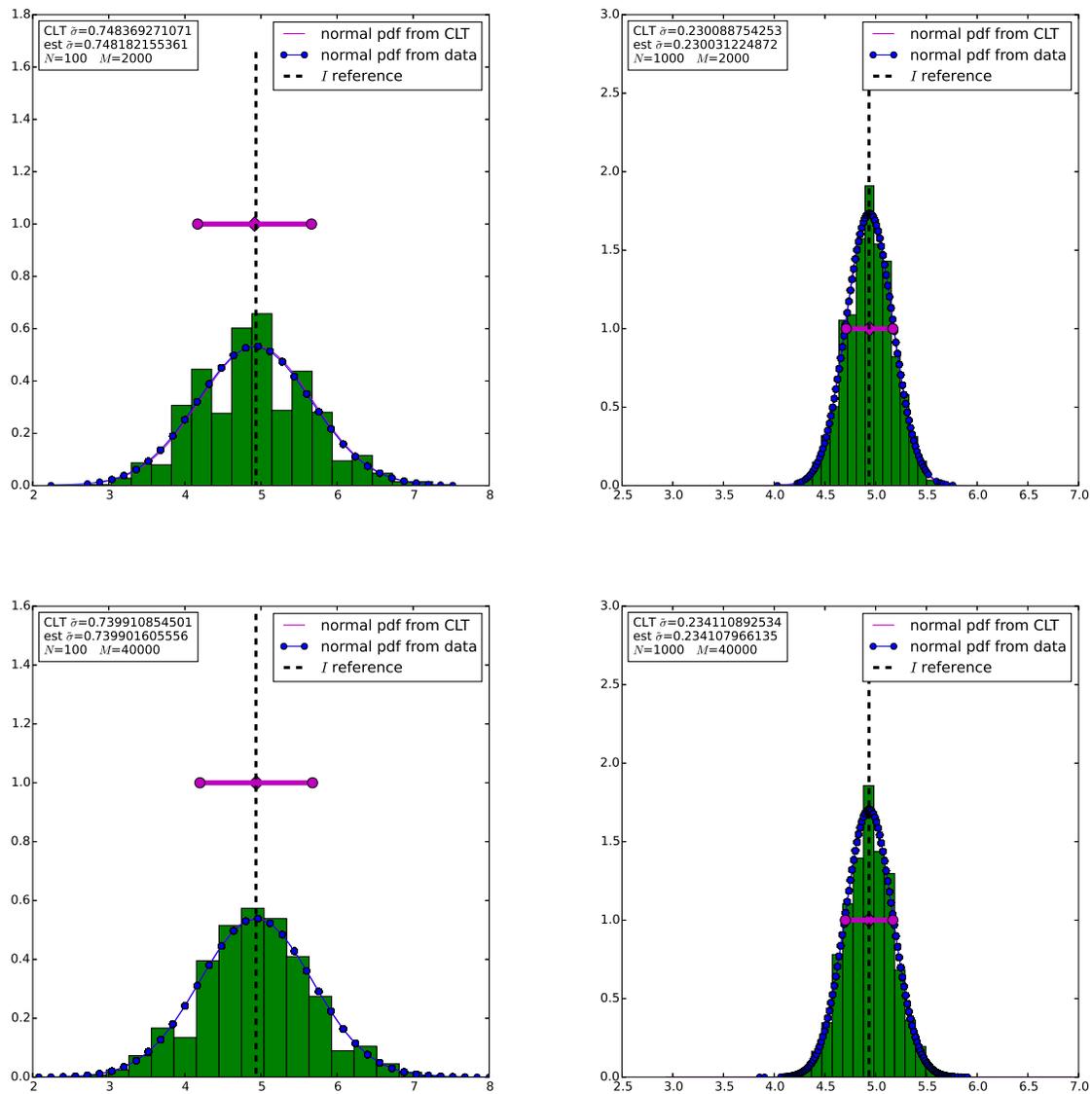
**Bemerkung 1.6.22.** Um das stochastische Experiment korrekt durchführen zu können, brauchen wir gute Zufallszahlengeneratoren. Die generierten Zahlen sollten einerseits statistische Tests für die entsprechende Verteilung erfüllen und gleichzeitig reproduzierbar sein.

Das ist auf den ersten Blick erstaunlich: reproduzierbare Zufallszahlen? Für eine numerische Simulation brauchen wir viele verschiedene komplexe Tools: physikalische Modelle, mathematische Vereinfachungen, Approximationen und numerische Algorithmen, Implementierungen in verschiedenen Codes, etc. Wenn man eine Diskrepanz mit dem physikalischen Experiment oder mit Erwartungen hat, die aus anderen Theorien stammen, und somit einen Fehler vermutet, der eventuell in den Berechnungen versteckt ist, muss man im Stande sein, die Berechnungen zu wiederholen, um eben den Fehler zu suchen. Ernsthaftige wissenschaftliche Arbeit muss immer reproduzierbar sein (sonst grenzt es an Schamanismus)<sup>5</sup>. Deshalb werden Pseudo-Zufallszahlen verwendet, die von deterministischen Algorithmen erzeugt werden. Es gibt einige gute und viele schlechte Zufallszahlengeneratoren. Man sollte immer diejenigen verwenden, die sich bereits als gut geeignet für ähnliche Probleme gezeigt haben. Für wichtige Rechnungen sollte man die erzielte Ergebnisse unter Verwendung verschiedener Arten von Zufallsgeneratoren überprüfen. Für die uniforme Verteilung sind die Generatoren Mersene-Twister (aktuell in python), Marsaglia (C-MWC), WELL zu verwenden; SPRNG (Masagni) eignet sich besonders für paralleles Rechnen. Für die normale Verteilung verwendet man die von Marsaglia verbesserte Box-Muller-Methode oder die Ziggurat-Methode von Marsaglia. Die Methoden, die auf die Berechnung der Inverse einer Verteilungsfunktion basiert sind, sind zu vermeiden, denn sie sind meistens schlecht-konditionierte Nullstellen-Probleme.

Der folgende Code implementiert das Monte-Carlo-Verfahren für die Berechnung des Volumens der Einheitskugel in  $d$  Dimensionen. Es werden  $M$  Experimente, d.h. Monte-Carlo-Approximationen für  $I$ , für gewisse festgelegte Anzahl Realisierungen  $X_1, \dots, X_N$  aufgeführt. Die Ergebnisse dieser  $M$  Experimente werden gemittelt und deren Varianz geschätzt. Somit sind die berechnete und gemittelte  $\bar{I}_N$  auch eine Art gemittelter Vertrauensintervalle; sie enthalten die Information von  $M$ -Experimenten mit je  $N$  Zufallszahlen. Für  $M = 1$  haben wir ein Vertrauensintervall im oberen Sinne.

---

<sup>5</sup> Die Ergebnisse, die mit den teuren Supercomputers erzielt werden, können allerdings aus rein physikalischen Gründen oft nicht reproduzierbar sein. Umdenken und Reformulierungen der Reproduzierbarkeitsanforderungen sind zu erwarten.



**Abb. 1.6.23.** MC-Approximationen zusammen mit entsprechenden Dichten der Normalverteilung aus dem zentralen Grenzwertsatz für  $N = 100$  (links) bzw.  $M = 1000$  (rechts) Realisierungen und verschiedene  $M$  aus Code 1.6.24.

**Code 1.6.24:** Monte-Carlo für das Volumen der Einheitskugel

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import math
4
5 def mc_experiments(func,a,b,N,M):
6
7     """
8     Purpose: perform M experiments of Monte-Carlo method
9     """

```

```

9
10 Input: func ... function to integrate
11         a ... lower interval bound
12         b ... upper interval bound
13         N ... number of samples
14         M ... number of experiments
15
16 Output: Imean ... mean of Monte-Carlo approx. to integral over
17         experiments
18         Ivar2 ... (empirical) variance of Monte-Carlo approx.
19         to integral over experiments
20
21 Notes: None
22
23 """
24
25 # perform experiments
26 I = np.zeros(M) #this is an array of length M
27 for iexp in np.arange(M):
28     [I[iexp],ivar] = mcquad(func,a,b,N)
29     #we calculate an approximation with N calls, M times and store all the
30     values in I
31
32 # compute mean of experiments
33 Imean = sum(I)/M
34
35 # compute square (empirical) variance of experiments
36 Ivar = 0.
37 if M>1:
38     Ivar = sum(I**2)/(M-1.) - (M/(M-1.))*(Imean**2)
39 else:
40     Ivar = ivar # this is what mcquad returns, here sigmatilde**2
41
42 # return
43 return Imean,Ivar
44
45 def mcquad(func,a,b,N):
46
47     """
48     Purpose: compute definite integral of func enclosed between a & b
49             with the Monte-Carlo method
50
51     Input: func ... function to integrate
52           a ... lower interval bound
53           b ... upper interval bound
54           N ... number of samples
55
56     Output: I ... Monte-Carlo approximation to integral
57           var2 ... (sample) variance/sqrt(N)
58
59     Notes: None
60
61     """
62
63     # dimension of the domain

```

```

63  if ( len(a) == len(b) ):
        dim = len(a)
65  else:
        return
67
        # generate N samples in [a,b]
69  x = np.random.rand(dim,N) #rand generates a matrix of given diensiuons with
random numbers
        #dim is the dimension of the space on which f takes its values, in the simplest
        case dim=1
71  #but we often use Monte Carlo for higher-dimensional integrals
        for idim in np.arange(dim):
73      #we are working on the interval [a,b] but we generate numbers in [0,1], so
we have to change interval
            x[idim,:] = a[idim] + (b[idim] - a[idim])*x[idim,:]
75
            # volume of domain [a,b]
77  vol = 1.
        for idim in np.arange(dim):
79      vol = vol*(b[idim] - a[idim]) #compare with the simple 2-d case of a
rectangular surface

81  # evaluate function at sample points
        fx = func(x)
83
        # evaluate integral (i.e. compute the expectation value)
85  I = np.sum(fx)/N

87  # compute sigmatilde**2 here not further used
        if ( N > 1 ):
89      #we use the estimator for sigma given in the script
            var2 = (np.sum(fx**2)/N - I**2)/(N - 1)
91  else:
            var2 = 0.
93
        return I*vol, var2*vol**2
95
if __name__ == "__main__":
97
        # characteristic function of unit sphere in d-dimensions
99  def f(x):
        [dim,n] = np.shape(x)
101  r2 = 0.
        for idim in np.arange(dim):
103      r2 = r2 + x[idim,:]**2 #with this recursion we compute  $\langle x, x \rangle^2$ 
            fx = 1.*(r2 <= 1) #if  $\langle x, x \rangle^2 < 1$  then x is in the unit sphere, so f(x)=1.
Otherwise f(x)=0
105  #this allows us to calculate the volume of the unit sphere via integration of
f(x)
            return fx
107
        # parameters
109  dim = 4 # dimension
        a = - np.ones(dim) # lower bound
111  b = + np.ones(dim) # upper bound
        Iref = np.pi**(dim/2.)/math.gamma(dim/2.+1) # reference volume of

```

```

113                                     # unit sphere in d-dimensions
M = 1                                # number of experiments
115 N = 10**np.arange(7) # number of samples (list!)

117 # convergence study
Imean_1 = [] # initialize list for experiments means of the integral
119 Ivar_1 = [] # initialize list for experiments variance of the integral
error_1 = [] # initialize list for error
121 for Nk in N:
    [Imean_k,Ivar_k] = mc_experiments(f,a,b,Nk,M) # here M=1
123     Imean_1.append(Imean_k)
    Ivar_1.append(Ivar_k) # M=1 so this is  $\sigma^2$  for each Nk
125     error_1.append(np.abs(Imean_k - Iref)) #the corresponding error

127 M = 100
# convergence study
129 Imean = [] # initialize list for experiments means of the integral
Ivar = [] # initialize list for experiments variance of the integral
131 error = [] # initialize list for error
for Nk in N:
133     [Imean_k,Ivar_k] = mc_experiments(f,a,b,Nk,M)
    Imean.append(Imean_k)
135     Ivar.append(Ivar_k) # here M>1 hence N-averaged  $\sigma^2$ 
    error.append(np.abs(Imean_k - Iref)) #the corresponding error

137 #----- plots -----
139 def plot_1():
    plt.figure(1)
141     plt.clf()
    plt.loglog(N,error_1,'b-+',label='Single') #we plot using a logarithmic scala
143     plt.axis([N[0],N[-1],1e-5,1e2])
    plt.title('Monte-Carlo integration')
145     plt.xlabel('$N$')
    plt.ylabel('$Error$')
147     plt.legend()

149 def plot_2():
    plt.figure(1)
151     plt.loglog(N,error,'r-+',label='Average over M='+str(M)+' experiments')
    plt.legend()

153 def plot_3():
    plt.figure(1)
155     plt.loglog([1.e0,1.e6],[1.e-2,1.e-5],'k--',label='$\propto 1/\sqrt{N}$')
    plt.legend()

159 def plot_4():
    plt.figure(2)
161     plt.clf()
    plt.semilogx([N[0],N[-1]],[Iref,Iref], 'k:' ,label=r"$I$ reference")
163     plt.semilogx(N,Imean , 'r-+',label=r"$\bar{I}_N$")
    plt.semilogx(N,Imean - np.sqrt(Ivar) , 'r^--' \
165         ,label=r"$\bar{I}_M - \bar{\sigma}_M$")
    plt.semilogx(N,Imean + np.sqrt(Ivar) , 'rv--' \

```

```

167         ,label=r"$\bar{I}_M + \bar{\sigma}_M$")
168     plt.title('Monte-Carlo integration')
169     plt.xlabel('$N$')
170     plt.ylabel('$I_N$')
171     plt.legend()
172
173 def plot_5():
174     plt.figure(2)
175     plt.clf()
176     plt.semilogx([N[0],N[-1]], [Iref,Iref], 'k:', label=r"$I$ reference")
177     plt.semilogx(N, Imean, 'r+', label=r"$\bar{I}_N$")
178     plt.semilogx(N, Imean - np.sqrt(Ivar), 'r^--' \
179                 ,label=r"$\bar{I}_N - \bar{\sigma}_N$")
180     plt.semilogx(N, Imean + np.sqrt(Ivar), 'rv--' \
181                 ,label=r"$\bar{I}_N + \bar{\sigma}_N$")
182     plt.title('Monte-Carlo integration: dependence on $N$')
183     plt.xlabel('$N$')
184     plt.ylabel('$I_N$')
185     plt.legend()
186
187     plt.figure(3)
188     plt.clf()
189     plt.loglog(N, 2*np.sqrt(Ivar), 'rv--' \
190              ,label=r"$2\bar{\sigma}_N$", linewidth=2)
191     plt.loglog(N, 2/np.sqrt(N), 'k:', label=r"$2/\sqrt{N}$", linewidth=2)
192     plt.title(r'Monte-Carlo integration: dependence on $N$')
193     plt.xlabel(r'$N$')
194     plt.ylabel(r'Interval length')
195     plt.legend()
196
197 def runfew(Nk, Mk):
198     Imean_Mk = [] # initialize list for experiments means of the integral
199     Ivar_Mk = [] # initialize list for experiments variance of the integral
200     error_Mk = [] # initialize list for error
201     for m in Mk:
202         [Imean_m, Ivar_m] = mc_experiments(f, a, b, Nk, 1)
203         Imean_Mk.append(Imean_m)
204         Ivar_Mk.append(Ivar_m)
205         error_Mk.append(np.abs(Imean_m - Iref))
206     return Imean_Mk, Ivar_Mk, error_Mk
207
208 def run_and_plot():
209     plt.figure(5)
210     plt.clf()
211     Mk = np.arange(10)
212     # a few runs
213     Nk = 100
214     Imean_Mk, Ivar_Mk, error_Mk = runfew(Nk, Mk)
215
216     Ileft_Mk = Imean_Mk - np.sqrt(Ivar_Mk)
217     Iright_Mk = Imean_Mk + np.sqrt(Ivar_Mk)
218     for k in Mk:
219         z = [k, k], [Ileft_Mk[k], Iright_Mk[k]]
220         plt.plot(z[0], z[1], 'bo-', markersize=10, linewidth=5)

```

```

221     plt.plot(Mk, Imean_Mk
label=r"$\bar{I}_{\text{"+str(Nk)+"}}$"
223     #
225     Nk = 1000
Imean_Mk, Ivar_Mk, error_Mk = runfew(Nk,Mk)
227     Ileft_Mk = Imean_Mk - np.sqrt(Ivar_Mk)
Iright_Mk = Imean_Mk + np.sqrt(Ivar_Mk)
229     for k in Mk:
z = [k+0.1,k+0.1], [Ileft_Mk[k],Iright_Mk[k]]
231     plt.plot(z[0], z[1], 'go-', markersize=10, linewidth=5)
plt.plot(Mk+0.1,Imean_Mk
233     label=r"$\bar{I}_{\text{"+str(Nk)+"}}$"
235     Nk = 10000
Imean_Mk, Ivar_Mk, error_Mk = runfew(Nk,Mk)
237     Ileft_Mk = Imean_Mk - np.sqrt(Ivar_Mk)
Iright_Mk = Imean_Mk + np.sqrt(Ivar_Mk)
239     for k in Mk:
z = [k+0.2,k+0.2], [Ileft_Mk[k],Iright_Mk[k]]
241     plt.plot(z[0], z[1], 'mo-', markersize=10, linewidth=5)
plt.plot(Mk+0.2,Imean_Mk
243     label=r"$\bar{I}_{\text{"+str(Nk)+"}}$"
245     plt.plot(Mk,Iref*(1+0.*Mk),'k:' ,label=r"$I$ reference")
plt.xlim(-1,10)
247     plt.title('Monte-Carlo integration: dependence on experiment')
plt.xlabel('Experiment Number')
249     plt.ylabel('Confidence Interval')
plt.legend()

```

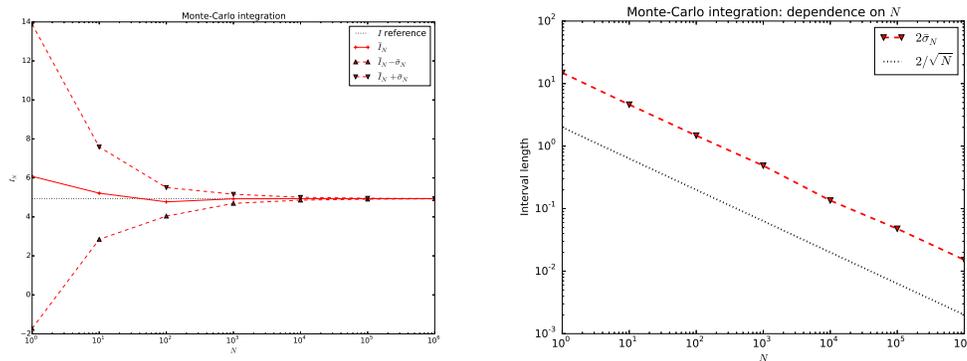
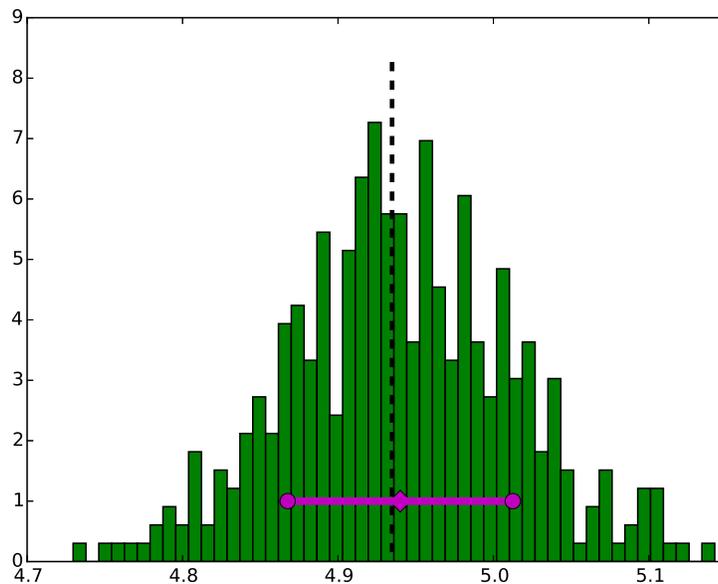


Abb. 1.6.25. MC-Approximation und Vertaruensintervall aus Code 1.6.24,  $M = 100$  Experimente.



**Abb. 1.6.26.** Histogramm der erhaltenen  $M = 400$  Approximationen aus Code 1.6.24,  $N = 10^4$  Realisierungen.

## 1.7 Methoden zur Reduktion der Varianz

Mit dem Monte-Carlo-Verfahren kann man genauere Ergebnisse bekommen, wenn man die Varianz reduziert. Das Prinzip ist folgendes: wenn wir den exakten Wert eines Integrals kennen, dann sollten wir ihn auch verwenden<sup>6</sup>. Als erstes Beispiel betrachten wir das folgende Integral:

$$I_0(x) = \frac{1}{\pi} \int_0^\pi e^{-x \cos t} dt .$$

$I_0(x)$  ist eine modifizierte Bessel-Funktion der Ordnung 0, die in der Physik vorkommt. Zuerst approximieren wir  $I_0(1)$  mittels eines einfachen Monte-Carlo-Verfahrens.

### Code 1.7.1: Plain Monte-Carlo

```

1  """
2  Computes integral
3  I0(1) = (1/pi) int(z=0..pi) exp(-cos(z)) dz by raw MC.
4  Abramowitz and Stegun give I0(1) = 1.266066
5  """
6
7  import numpy as np
8  import time

```

<sup>6</sup>und zwar bis nichts mehr zufällig bleibt, wenn möglich: “You use Monte Carlo until you understand the problem”, Mark Kac.

```

9
11 t1 = time.time()
13
13 M = 100 # number of times we run our MC inetgration
13 asval = 1.266065878 #exact value
13 ex = np.zeros(M)
15 print('A and S tables: IO(1) = ',asval)
15 print('sample      variance      MC IO val')
17 print('-----      -')
17 k = 5 # how many experiments
19 N = 10**np.arange(1,k+1) #we take 10,100,...10k sample points
19 v = []; e = [] #containers for variance and expectation value
21 for n in N:
21     for m in range(M):
23         x = np.random.rand(n) # sample
23         x = np.exp(np.cos(-np.pi*x))
25         ex[m] = sum(x)/n # quadrature
25         ev = sum(ex)/M #estimate the value of the integral
27         vex = np.dot(ex,ex)/(M-1.)
27         vex -= (M/(M-1.))*ev**2 #estimate the variance
29         v += [vex]; e += [ev] #store the values in the dedicated containers
29         print(n, vex, ev)
31
31 t2 = time.time() #measure the time needed for the operations
33 t = t2-t1
35
35 print("Serial calculation completed, time = %s s" % t)

```

Wir betrachten zwei Techniken zur Reduktion der Varianz: “Control Variates” und “Importance Sampling”.

### Control Variates

Das Integral

$$I = \int_0^1 f(t) dt$$

kann man umschreiben als

$$\begin{aligned}
 I &= \int_0^1 (f(t) - \phi(t)) dt + \int_0^1 \phi(t) dt \\
 &\approx \frac{1}{N} \sum_{i=1}^N [f(t_i) - \phi(t_i)] + I_\phi.
 \end{aligned}$$

Wir nehmen nun  $\phi$ , sodass:

- $\phi(u) \approx f(u)$  und
- $I_\phi = \int_0^1 \phi(u) du$  ist bekannt.

Für unser Beispiel schreiben wir

$$\begin{aligned} f(t) &= e^{-\cos(\pi t)} \\ &= 1 - \cos(\pi t) + \frac{1}{2}(\cos(\pi t))^2 + \dots \end{aligned}$$

und nehmen

$$\phi(t) = \text{die ersten drei Terme.}$$

Wir integrieren  $\phi(t)$  analytisch und  $g(t) = f(t) - \phi(t)$  mit dem Monte-Carlo Verfahren.

**Code 1.7.2:** Control Variates Monte-Carlo

```

1  """
2  Computes integral
3  I0(1) = (1/pi) int(z=0..pi) exp(-cos(z)) dz
4  by control variate. The splitting is
5  < exp(-cos) > = < exp(-cos) - phi > + int(0..pi) phi dz
6  where phi = 1 + cos - (1/2)*cos*cos is the control.
7  The exact integral of the control
8  int(z=0..pi) (1 + cos - (1/2)*cos*cos) dz = 1.25.
9  Abramowitz and Stegun give I0(1) = 1.266066
10 """
11
12 import numpy as np
13
14 M = 100 # number of times we run our MC inetgration
15 asval = 1.266065878 #exact value of the integral
16 ex = np.zeros(M)
17 print('A and S tables: I0(1) = ',asval)
18 print('sample      variance      MC I0 val')
19 print('-----      -')
20 k = 5 # how many experiments
21 N = 10**np.arange(1,k+1)
22 v = []; e = []
23 for n in N:
24     for m in range(M):
25         x = np.pi*np.random.rand(n) # sample
26         ctv = np.exp(-np.cos(x)) - 1. + np.cos(x) - 0.5*np.cos(x)*np.cos(x)
27         ex[m] = 1.25 + sum(ctv)/n # quadrature, 1.25 is the exact value of the
28         integral of the known part
29         ev = sum(ex)/M
30         vex = np.dot(ex,ex)/(M-1.) #dot (x,x) is a clever way for calculating sum < x_j^2
31         vex -= (M/(M-1.))*ev**2 #usual routine for estimating the variance
32         v += [vex]; e += [ev]
33     print(n, vex, ev)

```

## Importance Sampling

Bei dem “importance sampling”-Verfahren wird das Integral als

$$I = \int_{[0,1]^d} z(x) dx = \int_{[0,1]^d} \frac{z(x)}{g(x)} g(x) dx$$

geschrieben, wobei  $g(x)$  eine geeignete Dichtefunktion ist. Es werden nach  $g(x)$  verteilte Zufallszahlen verwendet um eine niedrigere Varianz zu erhalten.

Diese Technik zur Reduktion der Varianz beruht auf der Verallgemeinerung der Bemerkung 1.6.16 im folgenden Satz.

**Theorem 1.7.3.** Seien  $X_1, \dots, X_N$  Zufallsvariablen mit der Dichte  $f(x)$ . Dann

$$m_N(z(X)) := \frac{1}{N} \sum_{i=1}^N \frac{z(X_i)}{f(X_i)}$$

ist eine Zufallsvariable mit

$$\mathbb{E}m_N(z(X)) = \int_{[0,1]^d} z(x) dx .$$

In der Tat,

$$\mathbb{E}m_N(z(X)) = \frac{1}{N} \sum_{i=1}^N \mathbb{E} \frac{z(X_i)}{f(X_i)} = \frac{1}{N} N \mathbb{E} \frac{z(X)}{f(X)} = \int_{[0,1]^d} \frac{z(x)}{f(x)} f(x) dx = \int_{[0,1]^d} z(x) dx .$$

Nun stellt sich die Frage: welche Dichte  $g$  sollen wir wählen? Wir sollen mehr Punkte dort investieren, wo die Funktion grössere Werte annimmt:

$$\theta = \int_0^1 f(x) dx = \int_0^1 \frac{f(x)}{g(x)} g(x) dx = \int_0^1 \frac{f(x)}{g(x)} dG(x),$$

wobei  $g$  und  $G$  erfüllen

$$G(x) = \int_0^x g(y) dy, \quad G(1) = \int_0^1 g(y) dy = 1,$$

und  $G(x)$  is eine Verteilungsfunktion. Die Varianz ist nun

$$\sigma_{f/g}^2 = \int_0^1 (f(x)/g(x) - \theta)^2 dG(x) .$$

Normalerweise nimmt man  $g$ , sodass sie analytisch integrierbar ist und  $g/f \approx$  konstant ist. Wie wählen wir eine gute Verteilungsfunktion  $G$  aus?

Dazu können wir uns zuerst ein Beispiel anschauen.

**Beispiel 1.7.4.**  $\int_0^1 f(x) dx$  mit  $f(x) = 1/\sqrt{x(1-x)}$  ist sogar singular in  $x = 0, 1$ . Der allgemeine Trick ist hier, die Singularitäten zu isolieren:

$$g(x) = \frac{1}{4\sqrt{x}} + \frac{1}{4\sqrt{1-x}} = \frac{\sqrt{x} + \sqrt{1-x}}{4\sqrt{x(1-x)}} \implies \int_0^1 h(x) dG(x)$$

mit

$$h(x) = \frac{f(x)}{g(x)} = \frac{4}{\sqrt{x} + \sqrt{1-x}}$$

und aus  $dG(x)$  wird gezogen mit dem (nicht hier zu beweisendem) Trick (das übliche, allgemeine Verfahren werden wir danach sehen):

```

u = rand(N)
v = rand(N)
x = u*u
w = where(v>0.5)
x[w] = 1 - x[w]

```

## Code 1.7.5: Importance Sampling Monte-Carlo

```

2  from scipy import where, sqrt, arange, array
3  from numpy.random import rand
4  from time import time
5
6  from scipy.integrate import quad
7  f = lambda x: 1/sqrt(x*(1-x))
8  print('quad(f,0,1) = ', quad(f,0,1))
9
10 func = lambda x: 4./(sqrt(x)+sqrt(1-x))
11
12 def exotic(N): #the "trick" given in the script
13     u = rand(N)
14     v = rand(N)
15     x = u*u
16     w = where(v>0.5)
17     x[w] = 1 - x[w]
18     return x
19
20 def ismcquad():
21     k = 5 # how many experiments
22     N = 10**arange(1,k+1) # 10,100,...10k
23     ex = [] #container for results
24     for n in N:
25         x = exotic(n) # sample
26         x = func(x) #we use func and not f anymore: importance sampling
27         integration
28         ex += [x.sum()/n] # usual monte carlo quadrature
29     return ex
30
31 def mcquad():
32     k = 5 # how many experiments
33     N = 10**arange(1,k+1) # 10,100,...10k
34     ex = [] #container for results
35     for n in N:
36         x = rand(n) # sample
37         x = f(x)
38         ex += [x.sum()/n] # usual monte carlo quadrature
39     return ex
40
41 M = 100 # number of times we run our MC inetgration
42
43 t1 = time()
44 results = []
45 for m in range(M):
46     results += [ismcquad()]

```

```

46 t2 = time()
   t = t2-t1 #the time needed for running the simulation
48
   #first we adopt importance sampling monte-carlo method
50 print("ISMC Serial calculation completed, time = %s s" % t)
52 ex = array(results)
   ev = ex.sum(axis=0)/M
54 vex = (ex**2).sum(axis=0)/(M-1.)
   vex = vex - (M/(M-1.))*ev**2 #usual estimateors for expected for expected value
   and variance
56 print(ev[-1]) #value of the integral found with iportance sampling
   print(vex) #variance
58
   #now we uses standard monte-carlo method
60 t1 = time()
   results = []
62 for m in range(M):
       results += [mcquad()]
64
   t2 = time()
66 t = t2-t1
68
   print("MC Serial calculation completed, time = %s s" % t)
70
   ex = array(results)
   ev = ex.sum(axis=0)/M
72 vex = (ex**2).sum(axis=0)/(M-1.)
   vex = vex - (M/(M-1.))*ev**2
74 print(ev[-1])
   print(vex)

```

**Beispiel 1.7.6.** Wir wollen die zwei Techniken für die Reduktion der Varianz an  $\int_0^1 e^x dx$  zeigen. Nun

$$e^x - (1 + x) = x^2/2 + \dots$$

und  $\int_0^1 (1+x)dx = 3/2$ , so dass Variance Control einfach zu implementieren ist. Für Importance Sampling gehen wir folgendermassen vor:

1. Wir definieren  $g(x) = \frac{2}{3}(1+x)$
2. Unser Integral hat nun die Form:

$$I = \int_0^1 \frac{e^x}{\frac{2}{3}(1+x)} \frac{2}{3}(1+x) dx$$

3. Wir suchen  $y = y(x)$ , sodass  $y'(x) = \frac{2}{3}(1+x)dx$ . Es ergibt sich:

$$y(x) = \frac{2}{3}\left(x + \frac{x^2}{2}\right)$$

4. Dann können wir schreiben

$$x(y) = -1 + \sqrt{1 + 3y}$$

5. Schlussendlich wird unser Integral zu:

$$I = \int_{y(0)=0}^{y(1)=1} \frac{e^{x(y)}}{\frac{2}{3}(1+x(y))} dy$$

6. Wir wenden das Monte-Carlo Verfahren auf dieses Integral.

Diese Substitution folgt einem Satz der Wahrscheinlichkeitstheorie, der besagt, dass wenn  $x$  uniform verteilt ist, dann  $y = G^{-1}(x)$   $G$  als Wahrscheinlichkeitsverteilung besitzt.

Die beiden Techniken werden im folgenden Code implementiert.

**Code 1.7.7:** Monte-Carlo mit Reduktion der Varianz

```

1
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import math
6
7 def mc1d_experiments(func,a,b,N,M,p=None,prnd=None):
8     """
9     Purpose: perform M experiments of Monte-Carlo method
10
11     Input: func ... function to integrate
12            a    ... lower interval bound
13            b    ... upper interval bound
14            N    ... number of samples
15            M    ... number of experiments
16            p    ... probability density p(x), standard None ==> Uniform dist.
17            prnd ... random number generator with probability density p(x),
18                    standard None ==> Uniform dist.
19
20     Output: Imean ... mean of Monte-Carlo approx. to integral over
21             experiments
22            Ivar2 ... (empirical) variance of Monte-Carlo approx.
23                    to integral over experiments
24
25     Notes: None
26
27     """
28
29     # perform experiments
30     I = np.zeros(M)
31     for iexp in np.arange(M):
32         [I[iexp],ivar] = mc1dquad(func,a,b,N,p=p,prnd=prnd)
33
34     # compute mean of experiments

```

```

35 Imean = sum(I)/M
37 # compute square (empirical) variance of experiments
Ivar = sum(I**2)/(M-1.) - (M/(M-1.))*Imean**2
39
# return
41 return Imean,Ivar
43 def mc1dquad(func,a,b,N,p=None,prnd=None):
45     """
46     Purpose: compute definite integral of func enclosed between a & b
47             with the Monte-Carlo method
49     Input: func ... function to integrate
50            a ... lower interval bound
51            b ... upper interval bound
52            N ... number of samples
53            p ... probability density p(x), standard None ==> Uniform dist.
54            prnd ... random number generator with probability density p(x),
55                  standard None ==> Uniform dist.
57     Output: I ... Monte-Carlo approximation to integral
58            var2 ... (sample) variance/sqrt(N)
59
60     Notes: None
61     """
62
63     # generate N samples in [a,b]
64     if ( p == None ): # uniform distribution
65         x = np.random.rand(N)
66     else: # probability density p(x)
67         x = prnd(N) #we obtain x(y) as above
68     x[:] = a + (b - a)*x[:]
69
70     # volume of domain [a,b]
71     vol = b - a
72
73     # evaluate function at sample points
74     fx = func(x)
75     if ( p != None ):
76         px = p(x)
77
78     # evaluate integral (i.e. compute the expectation value)
79     if ( p == None ):
80         I = np.sum(fx)/N
81     else:
82         I = np.sum(fx/px)/N #division by px: corresponds to point 5) in the
83     "kochrezept" given above
84
85     # compute (sample) variance
86     if ( N > 1 ):
87         var2 = (np.sum(fx**2)/N - I**2)/(N - 1) #usual formula for the variance

```

```

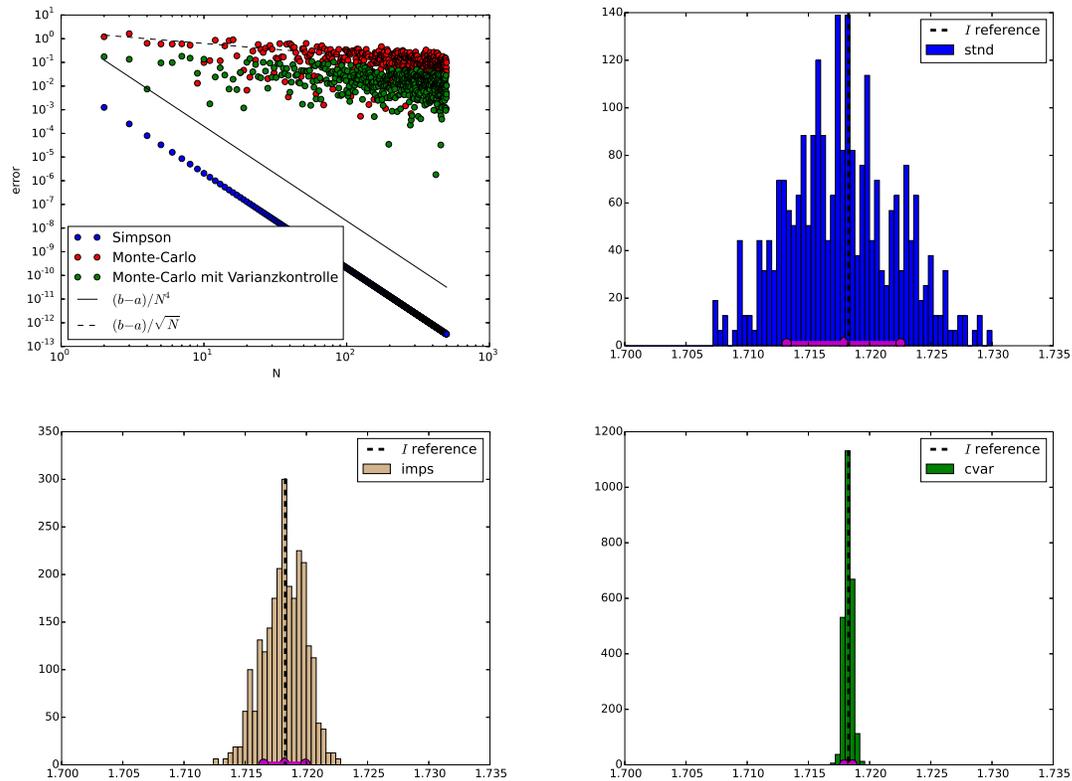
89     else:
90         var2 = 0.
91
92     # return
93     return vol*I,var2*vol**2
94
95 if __name__ == "__main__":
96
97     # exp(x)
98     def f(x):
99         fx = np.exp(x)
100        return fx
101
102     # exp(x) - phi(x)
103     def fmp(x):
104         fmp = np.exp(x) - (1. + x + x**2/2. + x**3/3.)
105         return fmp
106
107     # probability density p(x) = 2/3*(1 + x)
108     #the denominator in the integral. see its use in the function mc1dquad
109     def p(x):
110         px = 2.*(1 + x)/3.
111         return px
112
113     # random numbers with probability density p(x) = 2/3*(1 + x)
114     def prnd(N):
115         prnd = -1. + np.sqrt(1. + 3.*np.random.rand(N)) #in fact, we wrote: x = 1 +
sqrt(1+3y)
116         return prnd
117
118     # parameters
119     a = 0. # lower bound
120     b = 1. # upper bound
121     Iref = np.exp(1.) - 1. # reference
122     Iphi = 1. + 1./2. + 1./6. + 1./12. # in. phi
123     M = 100 # number of experiments
124
125     N = np.arange(10,10**4,10**2) # number of samples (list!)
126
127     # convergence study
128     Imean = [] # initialize list for experiments means of the integral
129     Imean_cv = [] # initialize list for experiments means of the integral
130     Imean_is = [] # initialize list for experiments means of the integral
131     Ivar = [] # initialize list for experiments variance of the integral
132     Ivar_cv = [] # initialize list for experiments variance of the integral
133     Ivar_is = [] # initialize list for experiments variance of the integral
134     error = [] # initialize list for error
135     error_cv = [] # initialize list for error
136     error_is = [] # initialize list for error
137     for Nk in N:
138         # standard
139         [Imean_k,Ivar_k] = mc1d_experiments(f,a,b,Nk,M)
140         Imean.append(Imean_k)
141         Ivar.append(Ivar_k)

```

```

141     error.append(np.abs(Imean_k - Iref))
        # controlled variates
143     [Imean_k,Ivar_k] = mcl_d_experiments(fmphi,a,b,Nk,M)
        Imean_k = Imean_k + Iphi
145     Imean_cv.append(Imean_k)
        Ivar_cv.append(Ivar_k)
147     error_cv.append(np.abs(Imean_k - Iref))
        # controlled variates
149     [Imean_k,Ivar_k] = mcl_d_experiments(f,a,b,Nk,M,p=p,prnd=prnd)
        Imean_is.append(Imean_k)
151     Ivar_is.append(Ivar_k)
        error_is.append(np.abs(Imean_k - Iref))
153
def plot_1():
155     plt.figure(1)
        plt.clf()
157     plt.loglog(N,error    , 'b--+',label='Single'    )
        plt.loglog(N,error_cv,'r--+',label='Single CV')
159     plt.loglog(N,error_is,'g--+',label='Single IS')
        plt.axis([N[0],N[-1],1e-8,1e2])
161     plt.title('Monte-Carlo integration')
        plt.xlabel('$N$')
163     plt.ylabel('$Error$')
        plt.legend()
165
def plot_2():
167     plt.figure(1)
        plt.loglog(N,error    , 'b---+',label='Average over M experiments'    )
169     plt.loglog(N,error_cv,'r---+',label='Average over M experiments CV')
        plt.loglog(N,error_is,'g---+',label='Average over M experiments IS')
171     plt.legend()
173
def plot_3():
        plt.figure(1)
175     plt.loglog([1.e0,1.e6],[1.e-5,1.e-8], 'k--',label='$\propto 1/\sqrt{N}$')
        plt.legend()
177
def plot_4():
179     plt.figure(2)
        plt.clf()
181     plt.semilogx([N[0],N[-1]],[Iref,Iref], 'k:' ,label=r"$I$ reference")
        plt.semilogx(N,Imean    , 'b--+',label=r"$\bar{I}_N$")
183     plt.semilogx(N,Imean - np.sqrt(Ivar) , 'b^--')
        plt.semilogx(N,Imean + np.sqrt(Ivar) , 'bv--')
185     plt.semilogx(N,Imean_cv    , 'r--+',label=r"$\bar{I}_N$ CV")
        plt.semilogx(N,Imean_cv - np.sqrt(Ivar_cv), 'r^--')
187     plt.semilogx(N,Imean_cv + np.sqrt(Ivar_cv), 'rv--')
        plt.semilogx(N,Imean_is    , 'g--+',label=r"$\bar{I}_N$ IS")
189     plt.semilogx(N,Imean_is - np.sqrt(Ivar_is), 'g^--')
        plt.semilogx(N,Imean_is + np.sqrt(Ivar_is), 'gv--')
191     plt.title('Monte-Carlo integration')
        plt.xlabel('$N$')
193     plt.ylabel('$I_N$')
        plt.legend()

```



**Abb. 1.7.8.** Typische Ergebnisse bei der Quadratur mittels verschiedener Methoden; die Histogrammen sind für  $M = 400$  und  $N = 10^4$  Realisierungen im Beispiel 1.7.6.

Mehr über die Monte-Carlo-Methode findet man in den folgenden Büchern:

W.L. DUNN, J.K. SHULTIS, *Exploring Monte Carlo Methods*, Amsterdam, Elsevier, 2012, [NEBIS 009882256](#)

T. MÜLLER-GRONBACH, E. NOVAK, K. RITTER, *Monte Carlo-Algorithmen*, Berlin, Springer, 2012, [NEBIS 009889710](#)

# Chapter 2

## Gewöhnliche Differentialgleichungen

### 2.1 Anfangswertprobleme

In diesem Kapitel wollen wir Methoden entwickeln, um gewöhnliche Differentialgleichungen numerisch zu lösen. Dabei möchten wir uns speziell mit einer Klasse von Lösungsmethoden, den sogenannten *Einschrittverfahren* (genaue Definition hierzu später), beschäftigen. In vielen, vor allem anwendungsorientierten Bereichen, tauchen Differentialgleichungen auf, die bestimmte Vorgänge beschreiben (z.B. die Entwicklung eines physikalischen Systems oder das Wachstum einer Bevölkerung). Häufig sind aber die Differentialgleichungen schon für scheinbar einfache Probleme schwer oder gar nicht elementar lösbar. Betrachten wir hierzu den einfachen Fall des mathematischen Pendels:

**Beispiel 2.1.1.** (Mathematisches Pendel)

Die Differentialgleichung für den Auslenkungswinkel eines mathematischen Pendels lautet:

$$\ddot{\varphi}(t) = -\frac{g}{l} \sin \varphi(t), \quad (2.1.1)$$

wobei  $g$  die Schwerebeschleunigung und  $l$  die Länge des Pendels ist. Wir sehen, dass diese Differentialgleichung nichtlinear ist, und es existiert keine elementare Darstellung der Lösung. Wie es dem Leser wahrscheinlich aus der Vorlesung “Physik 1” schon bekannt ist, können wir für kleine Winkel  $\varphi$  die Differentialgleichung linearisieren:

$$\sin \varphi(t) \approx \varphi(t) \implies \ddot{\varphi}(t) = -\frac{g}{l} \varphi(t)$$

und dann eine exakte Lösung des linearisierten Problems hinschreiben. Für grosse Winkel  $\varphi$  ist diese Approximation aber nicht mehr gut.

Zuerst wollen wir uns einige Definitionen anschauen.

### 2.1.1 Definitionen

**Definition 2.1.2.** (Gewöhnliche Differentialgleichung)

Eine gewöhnliche Differentialgleichung erster Ordnung hat die allgemeine Form

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad (2.1.2)$$

wobei die Unbekannte  $\mathbf{y} : I \rightarrow D$  eine Funktion von der Zeit  $t$  ist,  $\dot{\mathbf{y}} = \frac{d\mathbf{y}}{dt}$ ,  $\mathbf{f} : I \times D \rightarrow \mathbb{R}^d$ , und  $D \subset \mathbb{R}^d$  den Zustands- oder Phasenraum mit  $\dim(D) = d \geq 1$  bezeichnet.

**Hinweis:**

Wir werden im Folgenden häufig ODE (engl: ordinary differential equation) für Gewöhnliche Differentialgleichung schreiben. Der Begriff “gewöhnlich” deutet darauf hin, dass nur Ableitungen nach einer Variablen (nämlich  $t$ ) vorkommen.

**Bemerkung 2.1.3.** In Vektorschreibweise hat eine ODE für  $d > 1$  also generell folgende Form:

$$\begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \\ \vdots \\ \dot{y}_d \end{bmatrix} = \begin{bmatrix} f_1(t, \mathbf{y}) \\ f_2(t, \mathbf{y}) \\ \vdots \\ f_d(t, \mathbf{y}) \end{bmatrix} \quad \text{mit} \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix}. \quad (2.1.3)$$

Man spricht in diesem Falle auch von einem *System von Differentialgleichungen*.

Die Lösung einer Differentialgleichung auf  $J \subset I$  ist also eine Funktion  $\mathbf{y}(t)$ , die die Gleichung

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}(t)) \quad \text{für alle } t \in J \quad (2.1.4)$$

erfüllt.

**Definition 2.1.4.** (Autonome ODE)

Eine gewöhnliche Differentialgleichung heisst autonom, wenn  $\mathbf{f}$  unabhängig von  $t$  ist, also  $\mathbf{f} : D \rightarrow \mathbb{R}^d$ . Das ist zum Beispiel der Fall bei  $\dot{y}(t) = -\lambda y(t)$ ,  $\lambda = \text{const}$

**Definition 2.1.5.** (ODE  $n$ -ter Ordnung)

Eine gewöhnliche Differentialgleichung  $n$ -ter Ordnung hat die allgemeine Form:

$$\mathbf{y}^{(n)} = \mathbf{f}(t, \mathbf{y}, \mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n-1)}) \quad \text{mit} \quad \mathbf{y} \in \mathbb{R}^d \quad \text{und} \quad \mathbf{y}^{(i)} = \frac{d^i \mathbf{y}}{dt^i}.$$

, wobei  $\mathbf{f}$  eine Funktion mit  $\mathbf{f} : I \times D \rightarrow \mathbb{R}^d$  ist. ( $D \subset \mathbb{R}^{dn}$ ).

**Definition 2.1.6.** (Anfangswertproblem  $n$ -ter Ordnung)

Ein Anfangswertproblem ist eine gewöhnliche Differentialgleichung  $n$ -ter Ordnung mit einem Anfangswert  $\mathbf{y}_0$ :

$$\begin{aligned} \mathbf{y}^{(n)} &= \mathbf{f}(t, \mathbf{y}, \mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n-1)}) \\ \mathbf{y}(t_0) &= \mathbf{y}_0, \mathbf{y}^{(1)}(t_0) = \mathbf{y}_1, \dots, \mathbf{y}^{(n-1)}(t_0) = \mathbf{y}_{n-1}. \end{aligned}$$

## Umformungen von Differentialgleichungen

Die meisten Verfahren, die wir kennenlernen werden, sind für Differentialgleichungen erster Ordnung ausgelegt. Was sollten wir allerdings tun, wenn wir eine Differentialgleichung höherer Ordnung lösen möchten? Mit kleineren Umformungen können wir leicht allgemeine ODE's in die gewünschte Form bringen. Hierzu einige Methoden und Beispiele.

### A) Umwandlung in Differentialgleichung erster Ordnung

Angenommen wir haben eine Differentialgleichung  $n$ -ter Ordnung gegeben:

$$\mathbf{y}^{(n)} = \mathbf{f}(t, \mathbf{y}, \mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n-1)}) \quad \text{mit } \mathbf{y} \in \mathbb{R}^m.$$

Wir führen eine neue Variable  $\mathbf{z} \in \mathbb{R}^{n \cdot m}$  ein, die wir wie folgt definieren:

$$\mathbf{z} = \begin{bmatrix} \mathbf{z}_1 \\ \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ \mathbf{y}^{(1)} \\ \vdots \\ \mathbf{y}^{(n-1)} \end{bmatrix}.$$

Dann gilt:

$$\dot{\mathbf{z}}(t) = \mathbf{g}(t, \mathbf{z}) \quad \text{mit } \mathbf{g}(t, \mathbf{z}) = \begin{bmatrix} \mathbf{z}_2 \\ \vdots \\ \mathbf{z}_n \\ \mathbf{f}(t, \mathbf{z}_1, \mathbf{z}_2, \dots, \mathbf{z}_n) \end{bmatrix}.$$

Dabei ist  $\mathbf{g}$  eine Funktion mit  $\mathbf{g} : I \times D \rightarrow \mathbb{R}^{n \cdot m}$ . Wir haben also eine ODE  $n$ -ter Ordnung in ein System von Differentialgleichungen erster Ordnung mit  $D \subset \mathbb{R}^{n \cdot m}$  überführt.

### Beispiel 2.1.7. (Newtonsche Gleichung)

Angenommen wir haben eine Bewegungsgleichung der Form  $m\ddot{u} = F(t, u)$ . Wir setzen  $\ddot{u} = f_2(t, u)$ , wobei  $f_2(t, u) = \frac{1}{m}F(t, u)$ . Nun können wir eine neue Variable  $\mathbf{y}$  einführen:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} u \\ \dot{u} \end{bmatrix} \quad \Longrightarrow \quad \dot{\mathbf{y}} = \begin{bmatrix} \dot{y}_1 \\ \dot{y}_2 \end{bmatrix} = \begin{bmatrix} y_2 \\ f_2(t, y_1) \end{bmatrix} = \mathbf{f}(t, \mathbf{y}).$$

Damit haben wir das Problem auf ein System von Differentialgleichungen erster Ordnung reduziert.

### B) Autonomisierung von Differentialgleichungen

Angenommen wir haben eine ODE erster Ordnung:

$$\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}) \quad \text{mit } \mathbf{y} \in \mathbb{R}^n.$$

Wir führen eine neue Variable  $\mathbf{z} \in \mathbb{R}^{n+1}$  ein und schreiben:

$$\mathbf{z} = \begin{bmatrix} \mathbf{y} \\ t \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{z}} \\ z_{n+1} \end{bmatrix} \quad \text{mit} \quad \tilde{\mathbf{z}} = \mathbf{y}. \quad (2.1.5)$$

Dann gilt:

$$\dot{\mathbf{z}} = \mathbf{g}(\mathbf{z}) \quad \text{mit} \quad \mathbf{g}(\mathbf{z}) = \begin{bmatrix} \mathbf{f}(z_{n+1}, \tilde{\mathbf{z}}) \\ 1 \end{bmatrix}.$$

Wir haben die Zeitabhängigkeit in die Differentialgleichung integriert. Dadurch erhalten wir ein autonomes System erster Ordnung.

**Bemerkung 2.1.8.** Diese Umformungen rechtfertigen es, im folgenden nur noch Differentialgleichungen erster Ordnung zu betrachten, da wir alle anderen Formen in diese überführen können.

## 2.1.2 Evolutionsoperator

Zur detaillierten mathematischen Beschreibung führen wir einige Definitionen ein.

**Definition 2.1.9.** (Evolutionsoperator)

Die zweiparametrische Familie  $\Phi^{s,t}$  von Abbildungen  $\Phi^{s,t}: D \mapsto D$  heisst **Evolutionsoperator** zur Differentialgleichung  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ , wenn

$$\Phi^{s,t} \mathbf{y}_0 =: \mathbf{y}(t) \text{ ist Lösung des AWP's } \begin{cases} \dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}), \\ \mathbf{y}(s) = \mathbf{y}_0 \end{cases} \quad \text{für alle } \mathbf{y}_0 \in D.$$

Wir wollen eine numerische Approximation  $\Psi$  des Evolutionsoperators  $\Phi$  finden. Dieser Operator  $\Psi$  heisst *diskreter Evolutionsoperator*; diskret heisst er deshalb, weil wir die Lösung nur mit endlich vielen Punkten berechnen; dazu verwenden wir eine Schrittweite  $\Delta t$  bei der numerischen Berechnung.

**Satz 2.1.10.** (Rechenregeln)

1.  $\Phi^{t,t} = Id$ ,
2.  $\Phi^{s,r} = \Phi^{t,r} \circ \Phi^{s,t}$ ,
3. Für autonome ODE's gilt zusätzlich  $\Phi^{s,t} = \Phi^{0,t-s} := \Phi^{t-s}$ .

**Beispiel 2.1.11.** (Autonome skalare Differentialgleichungen)

Hier wählen wir die Dimension des Zustandsraumes  $d = 1$ .

- 1)  $f(t, y) = -\lambda y$ ,  $\lambda \in \mathbb{R}$ , mit Lösung des AWP  $y(t) = y_0 e^{-\lambda t}$ ,  $t \in \mathbb{R}$ , die für alle Zeiten  $t \in \mathbb{R}$  existiert, für jedes  $y_0$  (globale Lösung). Der zugehörige Evolutionsoperator ist

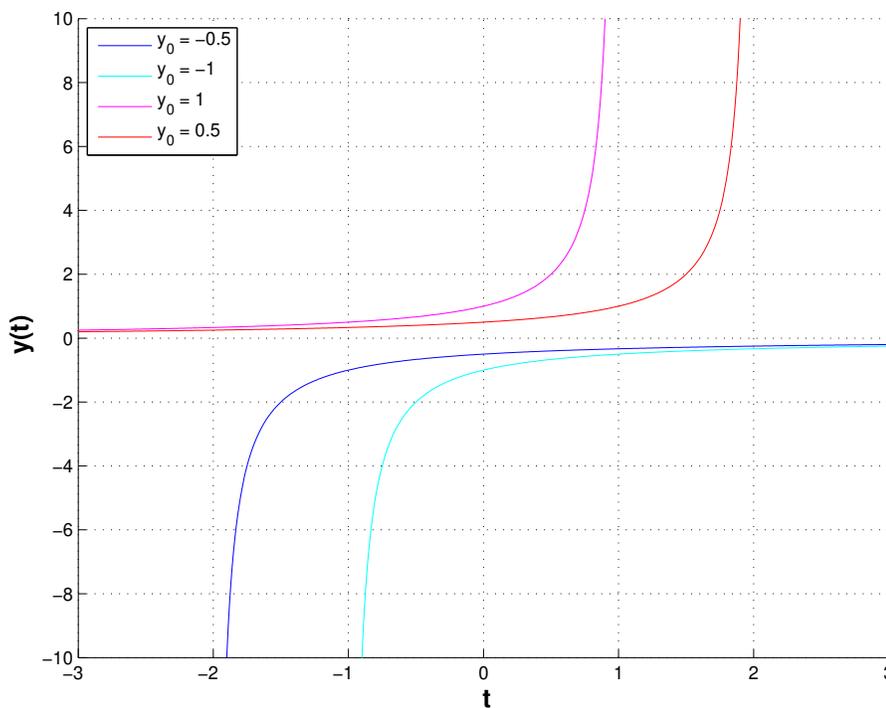
$$\Phi^t : \mathbb{R} \mapsto \mathbb{R}, \quad \Phi^t(y_0) = e^{-\lambda t} y_0.$$

2)  $f(t, y) = \lambda y^2$ ,  $\lambda \in \mathbb{R}$ . Die Lösung ist

$$y(t) = \begin{cases} \frac{1}{y_0^{-1} - \lambda t}, & \text{falls } y_0 \neq 0, \\ 0, & \text{falls } y_0 = 0. \end{cases}$$

Wenn  $\lambda$  und  $y_0$  strikt positiv sind, dann existiert eine Lösung für  $t \in ] - \infty, 1/(\lambda y_0)[$ .

Einige Lösungskurven für  $\lambda = 1$  sind in Abbildung 2.1.12 abgebildet.



**Abb. 2.1.12.** Lösungskurven für die Explosionsgleichung, wenn  $\lambda = 1$

3)  $f(t, y) = -\frac{1}{\sqrt{y}}$ ,  $D = \mathbb{R}^+$ ; für den Anfangswert  $y(0) = 1$  gibt es die Lösung

$$y(t) = (1 - 3t/2)^{2/3}, \quad t \in ] - \infty, 2/3[.$$

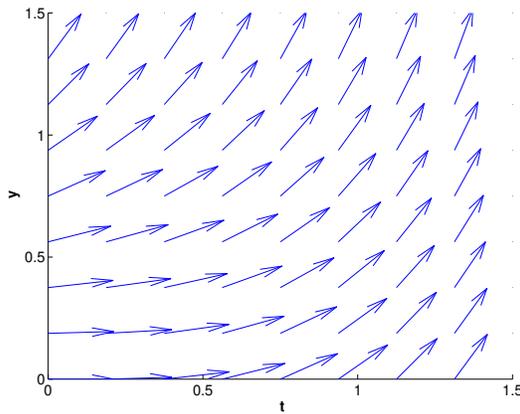
Die Lösung läuft bis zum Rand  $y = 0$  des erweiterten Zustandsraumes.

4)  $f(t, y) = \sin(1/y) - 2$ , mit  $D = \mathbb{R}^+$  und Anfangswert  $y(0) = 1$ . Die Lösung  $y(t)$  erfüllt  $\dot{y} \leq -1$  also  $y(t) \leq 1 - t$  und wir sehen einen Kollaps für  $t^* < 1$ : für die exakte Lösung haben wir  $\lim_{t \rightarrow t^*} y(t) = 0$ , während die rechte Seite an  $y = 0$  nicht definiert ist.

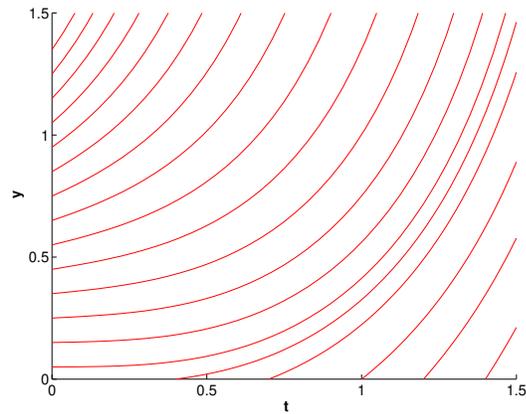
**Beispiel 2.1.13.** (Richtungsfeld und Lösungskurven)

Die Riccati-Differentialgleichung lautet:

$$\dot{y} = y^2 + t^2; \text{ hier: } d = 1, D = \mathbb{R}^+ \text{ und } t \in \mathbb{R}^+.$$



Richtungsfeld



Lösungskurven

Die Lösungskurven sind tangential zum Richtungsfeld an jedem Punkt des erweiterten Zustandsraumes.

Eine alternative Interpretation des *Richtungsfeldes als Geschwindigkeitsfeld* einer Flüssigkeit ist: die Lösungskurven sind die Trajektorien von Teilchen, die sich in der Flüssigkeit bewegen.

**Definition 2.1.14.** Der Punkt  $y^* \in D$  mit der Eigenschaft  $f(t, y^*) = 0$  für alle  $t$  heisst *Fixpunkt* oder stationärer Punkt der ODE  $\dot{y} = f(t, y)$ .

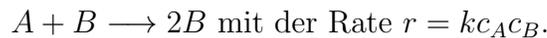
**Beispiel 2.1.15.** (Ressourcenbegrenztes Wachstum)

Die autonome logistische Differentialgleichung ( $d = 1, D = \mathbb{R}^+, t \in \mathbb{R}$ ) ist

$$\dot{y} = (\alpha - \beta y) y. \tag{2.1.6}$$

Hierbei ist  $y$  die Populationsdichte, in SI-Einheiten  $[y] = \frac{1}{m^2}$ . Die Populationsdynamik wird durch die Wachstumsrate  $\alpha - \beta y$  mit Wachstumskoeffizienten  $\alpha, \beta > 0$ ,  $[\alpha] = \frac{1}{s}$ ,  $[\beta] = \frac{m^2}{s}$  bestimmt.

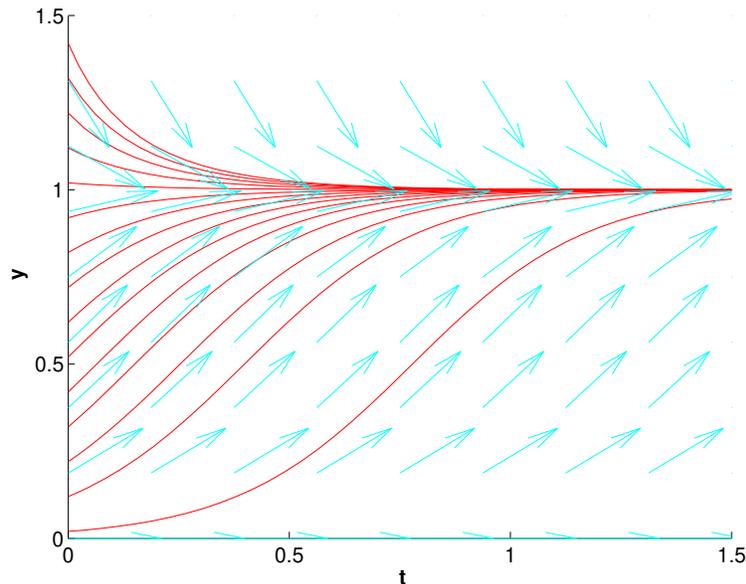
Die logistische Differentialgleichung tritt auch in autokatalytischen Reaktionen auf:



Für die Dichten  $c_A$  und  $c_B$  gilt  $\dot{c}_A = -r$  und  $\dot{c}_B = -r + 2r = r$ , also  $c_A + c_B = c_A(0) + c_B(0) = D$  ist konstant und wir erhalten die entkoppelten Gleichungen

$$\begin{aligned} \dot{c}_A &= -k(D - c_A)c_A \\ \dot{c}_B &= k(D - c_B)c_B, \end{aligned}$$

die in Gleichung (2.1.6) mit  $\beta = k, \alpha = kD$  für den Stoff  $B$ , und mit  $\beta = -k, \alpha = -kD$  für den Stoff  $A$  verwendet werden.



**Abb. 2.1.16.** Richtungsfeld und Lösungskurven der Gleichung (2.1.6) für  $\alpha, \beta = 5$

Wir sehen in Abbildung 2.1.16:

- $y = \alpha/\beta$  ist ein attraktiver Fixpunkt (der Punkt gegen den alle Lösungskurven konvergieren);
- $y = 0$  ist repulsiver Fixpunkt.

Eine Separation der Variablen liefert die exakte Lösung der logistischen Differentialgleichung mit  $y(0) = y_0 > 0$ :

$$y(t) = \frac{\alpha y_0}{\beta y_0 + (\alpha - \beta y_0) \exp(-\alpha t)}, \quad \text{für alle } t \in \mathbb{R}. \quad (2.1.7)$$

### 2.1.3 Lineare Anfangswertprobleme

Häufig werden Anfangswertprobleme um einen stationären Punkt linearisiert. An einem stationären Punkt  $\tilde{\mathbf{y}}$  gilt  $\mathbf{f}(t, \tilde{\mathbf{y}}) = 0$ , d.h. wenn  $\tilde{\mathbf{y}}$  der Startwert ist, ist die Lösung  $\mathbf{y}(t)$  konstant. Falls  $\mathbf{f} \in C^2$  können wir  $\mathbf{f}$  mit dem Satz von Taylor um den stationären Punkt  $\tilde{\mathbf{y}}$  entwickeln:

$$\mathbf{f}(\mathbf{y}) = 0 + D_y \mathbf{f}(t, \tilde{\mathbf{y}}) \cdot (\mathbf{y} - \tilde{\mathbf{y}}) + O(\|\mathbf{y} - \tilde{\mathbf{y}}\|^2).$$

Damit erhalten wir ein vereinfachtes Modell, das linearisierte Anfangswertproblem:

$$\dot{\mathbf{y}}(t) = \mathbf{f}(t, \mathbf{y}) = D_y \mathbf{f}(t, \tilde{\mathbf{y}}) \cdot (\mathbf{y} - \tilde{\mathbf{y}}).$$

Sei nun die folgende lineare gewöhnliche Differentialgleichung gegeben:

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} + \mathbf{g}(t) \quad (2.1.8)$$

mit  $D = \mathbb{R}^d$ ,  $\Omega = I \times \mathbb{R}^d$  und als “Quellterm” die stetige Funktion  $\mathbf{g} : I \rightarrow \mathbb{R}^d$ . Wir nehmen hier an, dass  $A$  diagonalisierbar ist, d.h. es gibt  $\mathbf{S} \in \mathbb{R}^{d,d}$  regulär, so dass  $\mathbf{S}^{-1} \mathbf{A} \mathbf{S} = \text{diag}(\lambda_1, \dots, \lambda_d)$ ,  $\lambda_i \in \mathbb{C}$ .

Im Falle wo  $\mathbf{g} \equiv 0$  haben wir es mit einer autonomen homogenen linearen Differentialgleichung zu tun und

$$\hat{\mathbf{y}} := \mathbf{S}^{-1} \mathbf{y} \quad \text{löst} \quad \begin{array}{l} \hat{y}_1 = \lambda_1 \hat{y}_1, \\ \vdots \\ \hat{y}_d = \lambda_d \hat{y}_d \end{array} \quad \text{also } \hat{y}_i(t) = (\mathbf{S}^{-1} \mathbf{y}_0)_i e^{\lambda_i t}, \quad t \in \mathbb{R}.$$

Somit ist in diesem Fall die Lösung von (2.1.8):

$$\mathbf{y}(t) = \mathbf{S} \begin{bmatrix} e^{\lambda_1 t} & & & \\ & \ddots & & \\ & & \ddots & \\ & & & e^{\lambda_d t} \end{bmatrix} \mathbf{S}^{-1} \mathbf{y}_0.$$

Die Lösung der autonomen homogenen linearen Differentialgleichung ist genau  $e^{\mathbf{A}t} \mathbf{y}_0$ , das Ergebnis der Anwendung des Matrixexponentialoperators  $e^{\mathbf{A}t}$  auf den Anfangswert  $\mathbf{y}_0$ . Allerdings ist es eine schlechte Idee, den Matrixexponentialoperator über die Diagonalisierung zu berechnen. Für relativ kleine Matrizen  $\mathbf{A}$  ist die Padé-Approximation, die in `expm` implementiert ist, stabiler und schneller. Für grössere Matrizen ist ein Krylov-Raum-Verfahren (Arnoldi- oder Lanczos-Iteration) deutlich besser, wie wir bereits im Kapitel über die Eigenwertprobleme gesehen haben.

Wir wissen, dass eine unitäre Matrix  $\mathbf{V}_m \in \mathbb{R}^{d \times m}$  existiert, sodass gilt:

$$\mathbf{V}_m^H \mathbf{A} \mathbf{V}_m = \mathbf{H}_m \quad \text{wobei } m \ll d,$$

wobei  $\mathbf{H}_m \in \mathbb{R}^{m \times m}$  eine obere Hessenbergmatrix ist. Wir wollen nun eine Approximation  $u_m(t)$  von  $y(t)$  im Krylov-Raum  $\mathcal{K}_m(\mathbf{A}, \mathbf{y}_0) = \text{Span}\{\mathbf{y}_0, \mathbf{A} \mathbf{y}_0, \dots, \mathbf{A}^{m-1} \mathbf{y}_0\}$ . Dabei soll das Residuum  $\dot{\mathbf{u}}_m(t) - \mathbf{A} \mathbf{u}_m(t)$  orthogonal zum Approximationsraum  $\mathcal{K}_m(\mathbf{A}, \mathbf{y}_0)$  stehen:

$$\mathbf{u}_m(0) = \mathbf{y}_0 \quad \text{und} \quad \langle \mathbf{w}, \dot{\mathbf{u}}_m(t) - \mathbf{A} \mathbf{u}_m(t) \rangle = 0 \quad \forall \mathbf{w} \in \mathcal{K}_m(\mathbf{A}, \mathbf{y}_0).$$

Wir stellen  $\mathbf{u}_m(t)$  mit den orthonormalen Basisvektoren  $\mathbf{v}_i$  (die  $v_i$  sind die Spalten der Matrix  $\mathbf{V}_m$ ) dieses Krylov-Raumes dar, wobei deren Koeffizienten  $c_k$  zeitabhängig sind:

$$\mathbf{u}_m(t) = \sum_{k=1}^m c_k(t) \mathbf{v}_k = \mathbf{V}_m \cdot \mathbf{c}(t) \quad \text{wobei} \quad \mathbf{c}(t) = \begin{bmatrix} c_1(t) \\ \vdots \\ c_m(t) \end{bmatrix}.$$

Wir können also das ursprüngliche Anfangswertproblem auf ein kleineres System von Differentialgleichungen erster Ordnung reduzieren:

$$\dot{\mathbf{c}}(t) = \mathbf{H}_m \mathbf{c}(t), \quad \mathbf{c}(0) = \|\mathbf{y}_0\| \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Nun wenden wir einfach unsere Methode für kleine Matrizen an:

$$\mathbf{c}(t) = \|\mathbf{y}_0\| e^{\mathbf{H}_m t} \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Daraus erhalten wir dann unsere Approximation der Lösung  $\mathbf{u}_m(t) \approx \mathbf{y}(t)$ :

$$\mathbf{u}_m(t) = \|\mathbf{y}_0\| \mathbf{V}_m e^{\mathbf{H}_m t} \mathbf{c}(0).$$

Im inhomogenen Fall ( $g$  beliebig) bekommen wir die Lösung durch die Technik der **Variation der Konstanten**:

Wir suchen eine Lösung der Form  $\mathbf{y}(t) = e^{\mathbf{A}t} \mathbf{z}(t)$  mit einer stetig differenzierbaren Funktion  $\mathbf{z} \in C^1(\mathbb{R}, \mathbb{R}^d)$ .

Einerseits:

$$\dot{\mathbf{y}}(t) = \mathbf{A} e^{\mathbf{A}t} \mathbf{z}(t) + e^{\mathbf{A}t} \dot{\mathbf{z}}(t),$$

andererseits:

$$\mathbf{A} \mathbf{y}(t) + \mathbf{g}(t) = \mathbf{A} e^{\mathbf{A}t} \mathbf{z}(t) + \mathbf{g}(t),$$

also muss gelten:

$$\dot{\mathbf{z}}(t) = e^{-\mathbf{A}t} \mathbf{g}(t) \implies \mathbf{z}(t) = \mathbf{z}(t_0) + \int_{t_0}^t e^{-\mathbf{A}\tau} \mathbf{g}(\tau) d\tau.$$

Somit ist die Lösung:

$$\mathbf{y}(t) = e^{\mathbf{A}(t-t_0)} \mathbf{y}_0 + \int_{t_0}^t e^{\mathbf{A}(t-\tau)} \mathbf{g}(\tau) d\tau =: \Phi^{t_0,t} \mathbf{y}_0, \quad (2.1.9)$$

d.h., die Lösung des homogenen Problems plus ihre Faltung mit der Inhomogenität  $g$ .

**Bemerkung 2.1.17.** Zeitabhängige lineare Anfangswertprobleme

$$\dot{\mathbf{y}} = \mathbf{A}(t) \mathbf{y}$$

werden am besten mit Magnus-Methoden gelöst, Abschnitt 2.8.

### 2.1.4 Existenz, Eindeutigkeit, Stabilität

Bisher haben wir uns noch nicht die Frage gestellt, ob zu einem Anfangswertproblem tatsächlich immer eine Lösung existiert. Das zu wissen ist aber notwendig um herauszufinden, ob es überhaupt Sinn ergibt, nach Lösungen zu suchen.

**Definition 2.1.18.** (Lokale Lipschitz-Stetigkeit) Eine Funktion  $f : I \times D \rightarrow \mathbb{R}^d$  heißt lokal Lipschitz-stetig falls:

$$\forall (t, y) \in I \times D \quad \exists \delta > 0, L > 0, \quad \text{sodass} \quad \|f(t, z) - f(\tau, w)\| \leq L \|z - w\| ,$$

wobei gilt:

$$\|z - y\| < \delta \quad \|w - y\| < \delta \quad \|t - \tau\| < \delta .$$

Der folgende Satz gilt:

**Proposition 2.1.19.** Es gilt:

- (a) Wenn  $f$  lokal Lipschitz-stetig ist, dann ist  $f$  global Lipschitz-stetig auf kompaktem  $K \subset I \times D$ .
- (b) Wenn  $f$  und  $\frac{\partial f}{\partial y}$  stetig auf  $I \times D$  sind, dann ist  $f$  lokal Lipschitz-stetig.

**Satz 2.1.20.** (Picard-Lindelöf)

Angenommen  $\mathbf{f}(\mathbf{y})$  ist lokal Lipschitz-stetig in  $\mathbf{y}$  in einer geeigneten Umgebung von  $\mathbf{y}_0$ , so gilt:

das Anfangswertproblem  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  hat für beliebige Anfangswerte  $(t_0, \mathbf{y}_0) \in I \times D$  eine eindeutige Lösung (für mindestens eine kurze Zeitspanne  $[t_0, t_0 + \varepsilon]$ ). Dieses Zeitintervall kann dann analog erweitert werden.

**Bemerkung 2.1.21.** Wir werden im Laufe dieser Vorlesung immer Glattheit voraussetzen, damit wir alle Berechnungen durchführen können.

Nun stellt sich die Frage: wie unterscheiden sich zwei exakte Lösungen, die zu zwei verschiedenen Anfangswerten gehören? Es gilt der folgende Satz:

**Satz 2.1.22.** (Lipschitz-stetige Abhängigkeit vom Anfangswert)

Es seien  $\mathbf{y}$  und  $\tilde{\mathbf{y}}$  Lösungen des Anfangswertproblems  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  zu Anfangswerten  $\mathbf{y}_0$  und  $\tilde{\mathbf{y}}_0$ . Sei  $f$  global Lipschitz-stetig mit der Lipschitz-Konstante  $L(t)$  stetig in  $t$ . Dann gilt:

$$\|\mathbf{y}(t) - \tilde{\mathbf{y}}(t)\| \leq \|\mathbf{y}_0 - \tilde{\mathbf{y}}_0\| e^{\int_{t_0}^t L(\tau) d\tau} , \quad \text{für alle } t \in I .$$

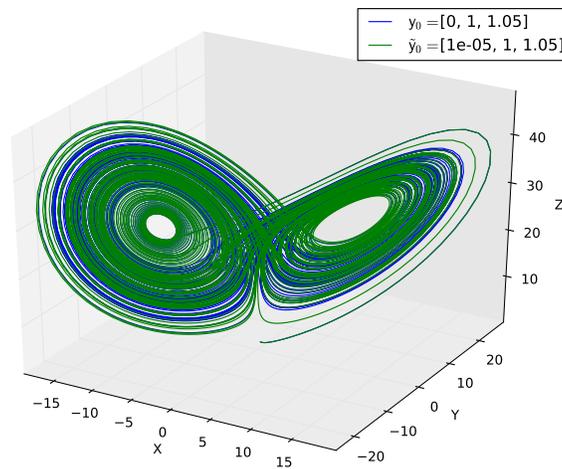
Manchmal ergeben winzige Störungen der Anfangswerte völlig verschiedene Zustände nach kurzer Zeit (wie in der Wettervorhersage).

**Beispiel 2.1.23.** (Lorenz-System)

Die autonome Differentialgleichung

$$\begin{aligned}\dot{x} &= \sigma(y - x) , \\ \dot{y} &= x(\rho - z) - y , \\ \dot{z} &= xy - \beta z ,\end{aligned}\tag{2.1.10}$$

mit  $D = \mathbb{R}^3$  und Konstanten  $\sigma, \rho, \beta \in \mathbb{R}^+$  ist das typische Beispiel eines chaotischen Systems. Kleine Mess- oder Rundungsfehler wirken sich in unvorhersehbaren Evolutionen.



**Abb. 2.1.24.** Zwei Lösungskurven des Lorenz-Systems für  $\sigma = 10, \rho = 28, \beta = 8/3$ .

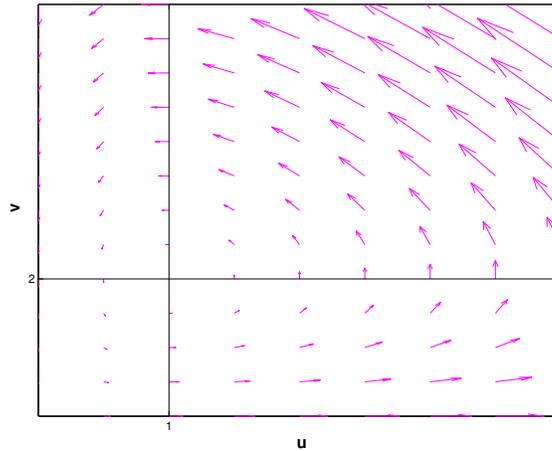
Für solche chaotische dynamische Systeme darf nicht mehr die einzelne Trajektorie das Ziel einer numerischen Simulation sein, sondern nur die Identifikation typischer Verhalten. In dem Lorenz-System in Abbildung 2.1.24 sind zwei Zonen (Attraktoren) zu sehen, wo die Trajektorien sich am meisten befinden. Essentiell ist dann die korrekte Behandlung von Erhaltungsgrößen, z.B. der Gesamtenergie eines mechanischen Systems.

## 2.1.5 Erhaltungsgrößen

Die autonome Lotka-Volterra-Differentialgleichung ( $d = 2$ ) ist

$$\begin{aligned}\dot{u} &= (\alpha - \beta v)u , \\ \dot{v} &= (\delta u - \gamma)v ,\end{aligned}\quad I = \mathbb{R}, \quad D = (\mathbb{R}^+)^2, \quad \alpha, \beta, \gamma, \delta > 0.\tag{2.1.11}$$

Dieses Problem modelliert die Evolution der Populationsdichten der Beuten  $u$  und der Räuber  $v$  in einem geschlossenen System.



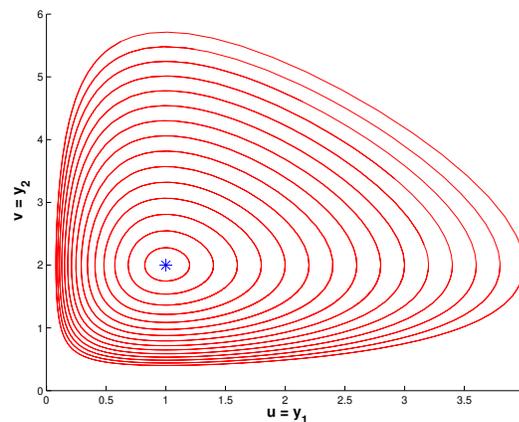
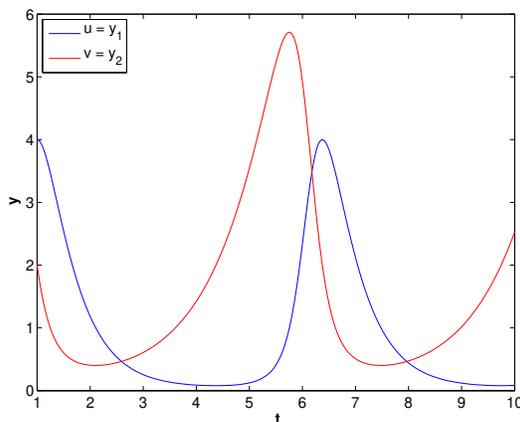
**Abb. 2.1.25.** Vektorfeld  $f$  für Lotka-Volterra-Differentialgleichung

Die Lösungskurven von (2.1.11) sind Trajektorien von Teilchen, die vom Geschwindigkeitsfeld  $f$  mitgetragen werden:

$$\dot{v} = \left(\delta - \frac{\gamma}{u}\right)uv \quad \text{und} \quad \dot{u} = \left(\frac{\alpha}{v} - \beta\right)uv$$

$$\text{also } \left(\frac{\alpha}{v} - \beta\right)\dot{v} = \left(\delta - \frac{\gamma}{u}\right)\dot{u} \implies \frac{d}{dt} \underbrace{\left(\delta u - \gamma \log u - \alpha \log v + \beta v\right)}_{I(u(t), v(t))} = 0.$$

Falls  $(u(t), v(t))$  Lösung von (2.1.11) ist, dann ist  $I(u(t), v(t)) \equiv \text{Konstant}$ . Die Lösungen von (2.1.11) sind Niveaulinien von  $I$ .



Zeitabhängige Lösung  $\mathbf{y}_0 := \begin{pmatrix} u(1) \\ v(1) \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$

Lösungskurven von (2.1.11) sind Niveaulinien von  $I$

**Definition 2.1.26.** (Erstes Integral)

Ein Funktional  $I : D \mapsto \mathbb{R}$  heißt erstes “Integral/Invariante” der ODE, wenn

$$I(\mathbf{y}(t)) \equiv \text{const.}$$

für jede Lösung  $\mathbf{y} = \mathbf{y}(t)$  der ODE.

**Vorsicht:**

Das erste Integral ist konstant für jede Lösung der ODE separat, es ist aber nicht konstant über alle Lösungen der ODE hinweg.

**Satz 2.1.27.** Eine notwendige und hinreichende Bedingung für *differenzierbares* erstes Integral der ODE ist

$$\text{grad } I(\mathbf{y}) \mathbf{f}(t, \mathbf{y}) = 0 \quad \text{für alle } (t, \mathbf{y}) \in \Omega. \quad (2.1.12)$$

*Proof.* Der Beweis folgt direkt aus der Kettenregel:

$$\text{grad}(I(\mathbf{y})) \cdot \mathbf{f}(t, \mathbf{y}) = \frac{d}{dy} I(\mathbf{y}) \cdot \dot{\mathbf{y}}(t) \quad \underbrace{=} \quad \frac{d}{dt} I(\mathbf{y}) = \frac{d}{dt} I(\mathbf{y}(t)) = 0. \quad \text{Somit sind}$$

Kettenregel

die zwei Ausdrücke äquivalent. □

**Definition 2.1.28.** (Hamiltonsche Differentialgleichung)

Sei  $H : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$  mit  $H = H(p, q)$  stetig differenzierbar, dann heisst das System gewöhnlicher Differentialgleichungen erster Ordnung

$$\begin{cases} \dot{p}(t) &= -\frac{\partial H}{\partial q}(p(t), q(t)) \\ \dot{q}(t) &= \frac{\partial H}{\partial p}(p(t), q(t)) . \end{cases} \quad (2.1.13)$$

*Autonomes Hamilton-System* mit der *Hamilton-Funktion*  $H$ .

Um diesen Begriff verständlicher zu machen, betrachten wir nochmals das Beispiel des mathematischen Pendels:

**Beispiel 2.1.29.** (Mathematisches Pendel)

Sei  $\alpha \in D$  der Auslenkungswinkel mit dem Zustandsraum  $D = [-\pi, \pi]$ , so gilt folgende Differentialgleichung für  $\alpha$

$$\ddot{\alpha}(t) = -\frac{g}{l} \sin(\alpha(t)) .$$

Wir führen eine neue Variable  $p = \dot{\alpha}$  ein und schreiben:

$$\begin{bmatrix} \dot{\alpha} \\ \dot{p} \end{bmatrix} = \begin{bmatrix} p \\ -\frac{g}{l} \sin(\alpha(t)) \end{bmatrix} . \quad (2.1.14)$$

Aus der Physik wissen wir, dass bei dieser Schwingung die Gesamtenergie  $E$  erhalten bleibt:

$$E(t) = \frac{1}{2} m l^2 p(t)^2 - m g l \cos(\alpha(t)) = \text{Konstant}.$$

$E$  ist also ein erstes Integral, da es für alle Lösungen  $(\alpha(t), p(t))$  und für alle  $t$  konstant ist.

Die zu diesem System gehörige Hamilton-Funktion ist in diesem Fall mit  $q = \alpha$  und  $p = \dot{\alpha}$

$$H(p, q) = \frac{1}{2}p^2 - \frac{g}{l} \cos(q) = E_{kin} + E_{pot} = E,$$

Die Hamilton-Funktion beschreibt die Gesamtenergie. Die partiellen Ableitungen  $-\frac{\partial H}{\partial q}, \frac{\partial H}{\partial p}$  führen auf die Gleichungen in Gleichung (2.1.14).

**Bemerkung 2.1.30.** Bewegungsgleichungen in der klassischen Mechanik führen aufgrund der Energieerhaltung auf ein autonomes Hamilton-System. Sie werden darauf in der Vorlesung "Allgemeine Mechanik" nochmals detailliert eingehen.

Der folgende Satz gilt:

**Satz 2.1.31.** (Erstes Integral  $H$ )

Die Hamilton-Funktion  $H$  ist ein erstes Integral des dazugehörigen autonomen Hamilton-Systems.

*Proof.* Wir können die Differentialgleichung wie folgt schreiben:

$$y = \begin{bmatrix} p \\ q \end{bmatrix} \implies \dot{y} = J^{-1}(\text{grad } H(y)), \quad \text{wobei } J = \begin{bmatrix} 0 & Id \\ -Id & 0 \end{bmatrix}. \quad (2.1.15)$$

Sei  $\text{grad } H(\mathbf{y}) = [\mathbf{a}, \mathbf{b}]^T$

$$(\text{grad } H(\mathbf{y})) \cdot (\mathbf{J}^{-1}(\text{grad } H(\mathbf{y}))) = \mathbf{a} \cdot \mathbf{b} - \mathbf{a} \cdot \mathbf{a} = 0. \quad (2.1.16)$$

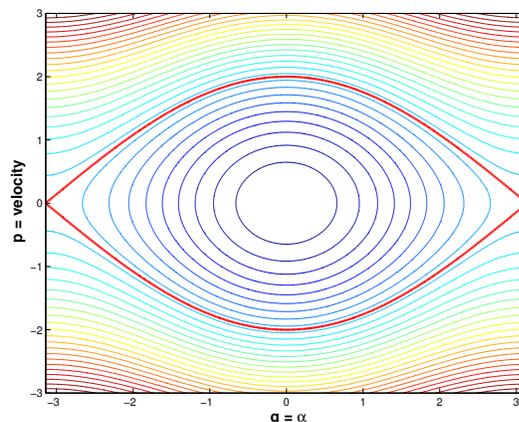
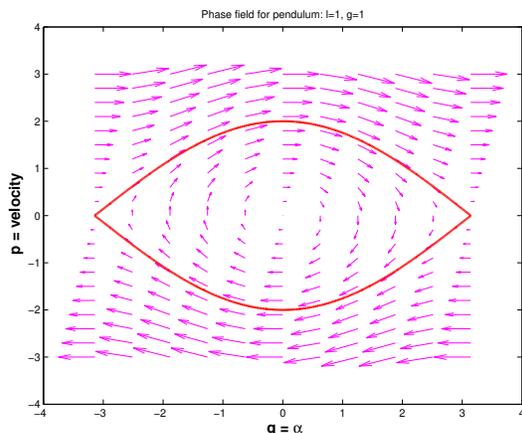
Mit Theorem 2.1.27 folgt die Aussage.  $\square$

Alternativ kann man dies auch mittels Nachrechnen zeigen:

da  $\text{grad}(H(\mathbf{y}))^T \cdot \mathbf{J}^{-1} \cdot \text{grad}(H(\mathbf{y}))$  eine Zahl ist, kann man diese transponieren ohne das Resultat zu ändern. Mit  $J^{-1} = J^T = -J$  folgt:

$$\begin{aligned} \text{grad}(H(\mathbf{y}))^T \cdot \mathbf{J}^{-1} \cdot \text{grad}(H(\mathbf{y})) &= \text{grad}(H(\mathbf{y}))^T \cdot \mathbf{J}^T \cdot \text{grad}(H(\mathbf{y})) = \\ &= \text{grad}(H(\mathbf{y}))^T \cdot (-\mathbf{J}) \cdot \text{grad}(H(\mathbf{y})). \end{aligned}$$

Daraus folgt, dass  $(H(\mathbf{y}))^T \cdot \mathbf{J}^{-1} \cdot \text{grad}(H(\mathbf{y})) = 0$  bzw. dass  $(H(\mathbf{y}))$  ein erstes Integral ist.



**Abb. 2.1.32.** Das Vektorfeld für (2.1.14) und die Isolinien der Energie entsprechen den Lösungskurven.

**Beispiel 2.1.33.** Die Newtonschen Bewegungsgleichungen eines Körpers (Ortskoordinate  $\mathbf{r} = \mathbf{r}(t)$ ) mit Masse  $m > 0$  im Kraftfeld  $\mathbf{f} \in \mathbb{R}^n$ ,  $n \in \mathbb{N}$  sind:

$$m \ddot{\mathbf{r}}(t) = \mathbf{f}(\mathbf{r}(t)). \quad (2.1.17)$$

Wir betrachten hier den Spezialfall eines radialsymmetrischen konservativen Kraftfeldes

$$\mathbf{f}(\mathbf{x}) = -\text{grad} U(\mathbf{x}), \quad \mathbf{x} \in \mathbb{R}^n, \quad U(\mathbf{x}) = G(\|\mathbf{x}\|). \quad (2.1.18)$$

Noch spezieller ist  $G(r) = -\frac{G_0}{r}$ , denn das ist das Keplerproblem. Es geht um die Bewegung im Gravitationsfeld einer Zentralmasse (z.B. Sonne – Planet), nun beschrieben als Hamiltonsches System mit Konfigurationsraum  $M := \mathbb{R}^n \setminus \{0\}$ ,  $\mathbf{q} := \mathbf{r}$ , und Hamilton-Funktion (Energie)

$$H(\mathbf{p}, \mathbf{q}) := \frac{1}{2m} \|\mathbf{p}\|^2 + G(\|\mathbf{q}\|). \quad (2.1.19)$$

Die Notation  $p = m\dot{r} = \text{Impuls}$  liefert

$$\begin{cases} \dot{q} = \frac{1}{m} p \\ \dot{p} = -G'(\|q\|) \frac{q}{\|q\|} \end{cases},$$

denn (Newton:  $m\ddot{r}(t) = f(r(t))$ ):

$$f(q) = -\text{grad} G(\|q(x)\|) = -G'(\|q\|) \text{grad}_x \|q\| = -G'(\|q\|) \frac{q}{\|q\|}.$$

## 2.2 Polygonzugverfahren

Nachdem wir uns mit der Theorie der Differentialgleichungen näher beschäftigt haben, möchten wir nun einige Methoden erarbeiten, mit deren Hilfe wir Differentialgleichungen durch Zeitdiskretisierung lösen können. Nehmen wir an, dass wir eine Differentialgleichung haben, die durch  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$  und einen Anfangswert  $y(0) = y_0$  gegeben ist.

Mögliche **Ziele** sind:

- 1)  $y(T) \approx y_h(T)$ . d.h. nur die Lösung zur Endzeit  $t = T$  ist uns wichtig.
- 2)  $y(t) \approx y_h(t)$ , d.h. wir wollen eine Approximation der Lösung zu allen Zeiten  $t$ .

Wir zerlegen das Zeitintervall  $[t_0, T]$  in  $N$  kleine Intervalle

$$[t_{k-1}, t_k], \quad t_N = T, \quad k = 1, 2, \dots, N,$$

der Längen  $h_k$  und approximieren die Lösung  $y(t_k)$  mit  $y_k = y_h(t_k)$ . Dies werden wir mittels verschiedener numerischer Verfahren tun. Hier ist  $h = \min\{h_1, \dots, h_N\}$  der kleinste Zeitschritt.

### 2.2.1 Euler-Verfahren und implizite Mittelpunktsregel

Wir approximieren die zeitlokalen Lösungskurven durch die Tangenten am aktuellen Anfangszeitpunkt:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h_n \mathbf{f}(t_n, \mathbf{y}_n), \quad n = 0, \dots, N - 1.$$

Das ist das *explizite Euler-Verfahren*, das wir auch erhalten können, indem wir die Ableitung durch einen Vorwärts-Differenzen-Quotienten approximieren:

$$\mathbf{f}(t_n, \mathbf{y}(t_n)) = \dot{\mathbf{y}}(t_n) \approx \frac{\mathbf{y}(t_n + h_n) - \mathbf{y}(t_n)}{h_n}.$$

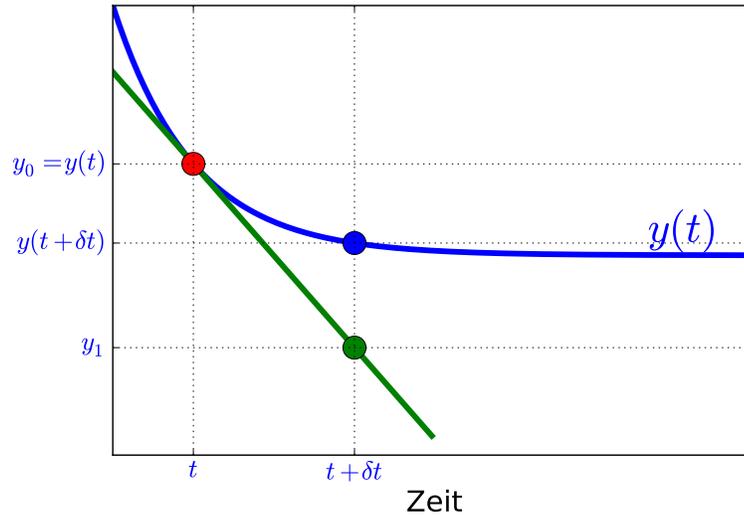


Abb. 2.2.1. Ein Schritt des expliziten Euler-Verfahrens

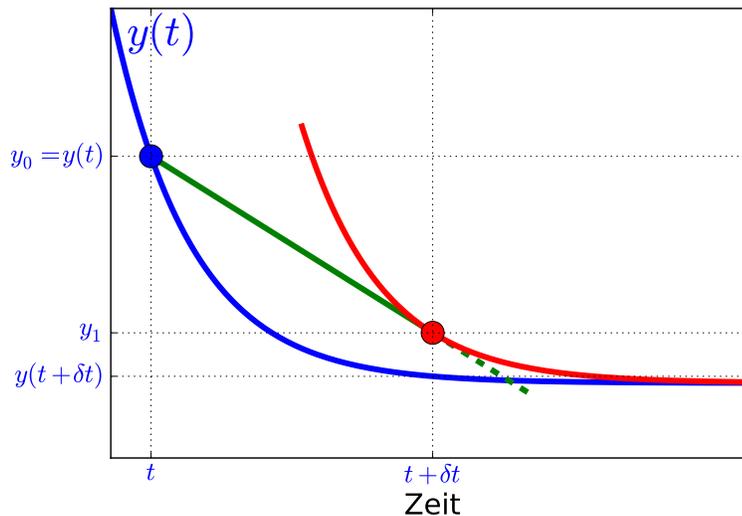
Wir können aber auch verlangen, dass die Gerade, die die exakte Lösung approximiert, tangential zur Lösung zur Zeit  $t_{n+1}$  ist (statt zur Anfangszeit  $t_n$ ):

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h_n \mathbf{f}(t_{n+1}, \mathbf{y}_{n+1}), \quad n = 0, \dots, N - 1.$$

Das ist das *implizite Eulerverfahren*, das wir auch erhalten können, indem wir die Ableitung durch einen Rückwärts-Differenzen-Quotienten approximieren:

$$\mathbf{f}(t_{n+1}, \mathbf{y}(t_{n+1})) = \dot{\mathbf{y}}(t_{n+1}) \approx \frac{\mathbf{y}(t_{n+1} - h_n) - \mathbf{y}(t_{n+1})}{-h_n}.$$

Dieses Verfahren heisst *implizit*, da die Lösung eines nicht-linearen Systems nötig ist, um den Wert von  $y_{n+1}$  zu erhalten.



**Abb. 2.2.2.** Ein Schritt des impliziten Euler-Verfahrens

Eine noch bessere Idee ist es, die Steigung am mittleren Zeitpunkt zu verwenden. Wir notieren mit  $t^* = \frac{1}{2}(t_n + t_{n+1})$  und  $\mathbf{y}^* = \frac{1}{2}(\mathbf{y}_n + \mathbf{y}_{n+1})$ , und ersetzen die Steigung in  $t^*$ , die eigentlich  $f(t^*, y(t^*))$  mit der Approximation  $f(t^*, \mathbf{y}^*)$ . Das ergibt:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h_n \mathbf{f}\left(\frac{1}{2}(t_n + t_{n+1}), \frac{1}{2}(\mathbf{y}_n + \mathbf{y}_{n+1})\right).$$

Das ist die implizite Mittelpunktsregel, die wir auch erhalten können, indem wir die Ableitung durch einen Zentralen-Differenzen-Quotienten approximieren:

$$\mathbf{f}\left(\frac{1}{2}(t_n + t_{n+1}), \frac{1}{2}(\mathbf{y}_n + \mathbf{y}_{n+1})\right) = \dot{\mathbf{y}}\left(\frac{1}{2}(t_n + t_{n+1})\right) \approx \frac{\mathbf{y}(t_n + h_n) - \mathbf{y}(t_n)}{h_n}.$$

Wir haben es wieder mit einem *impliziten* Verfahren zu tun, das jedoch wichtige Vorteile gegenüber den beiden Euler-Verfahren besitzt.

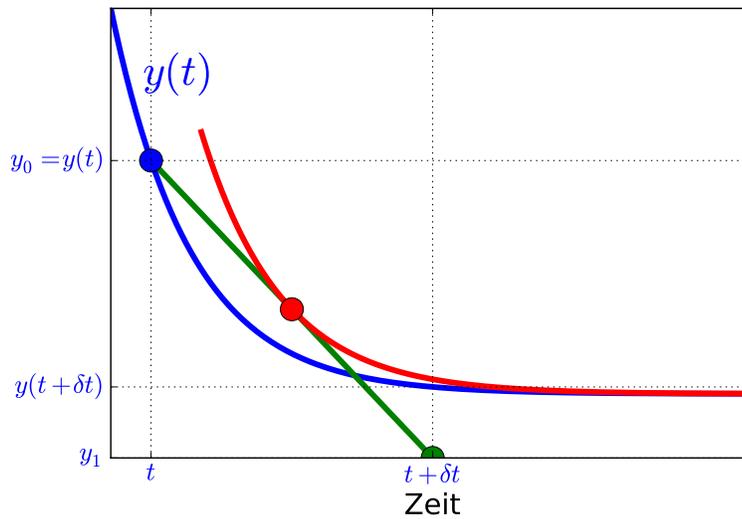


Abb. 2.2.3. Ein Schritt der impliziten Mittelpunktsregel

### 2.2.2 Strukturerhaltung

**Beispiel 2.2.4.** (Mathematisches Pendel)

Wie schon bekannt, ist die Hamiltonsche Bewegungsgleichung für das Pendel gegeben durch

$$\begin{bmatrix} \dot{\alpha} \\ \dot{p} \end{bmatrix} = \begin{bmatrix} p \\ -\frac{g}{l} \sin(\alpha) \end{bmatrix}.$$

Nun möchten wir diese Differentialgleichung mit dem expliziten und dem impliziten Eulerverfahren lösen. In Theorem 2.2.5 sind die Lösungen abgebildet, die wir mittels der beiden Verfahren erhalten .

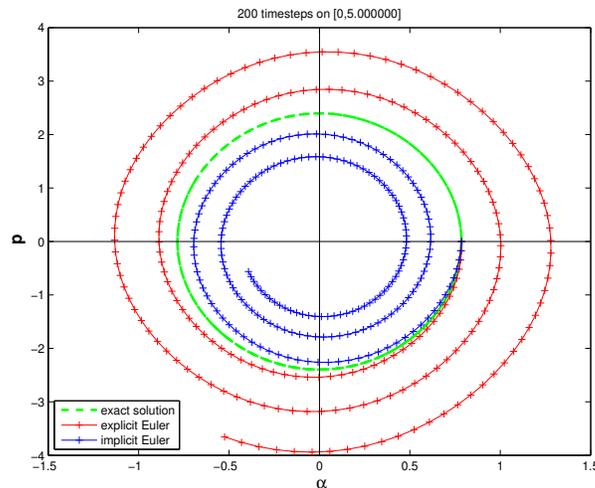
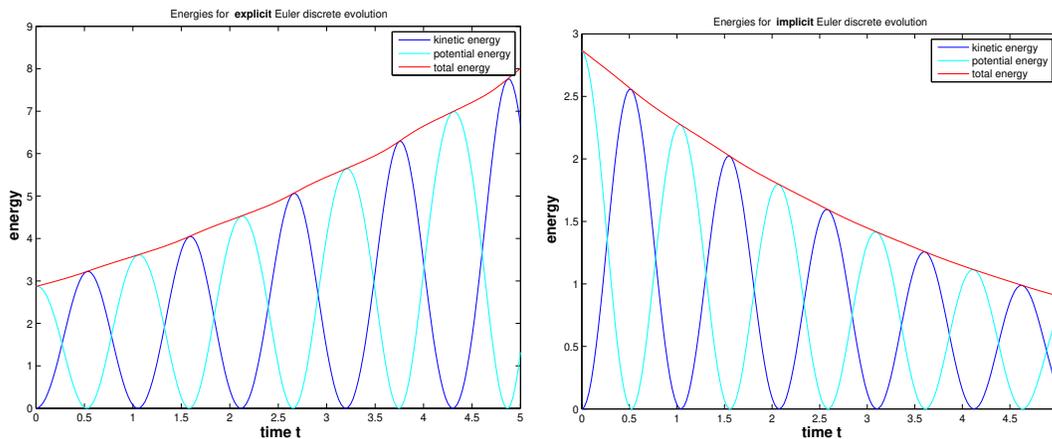


Abb. 2.2.5. Vergleich der beiden Euler-Verfahren im Phasenraum

Es fällt sofort auf, dass die beiden Ergebnisse stark von der korrekten Lösung abweichen. Betrachten wir dazu die Energie  $E$  des Systems für  $m = 1$ ,  $l = 1$ :

$$E(p, \alpha) = \frac{1}{2}p^2 - g \cos(\alpha). \quad (2.2.20)$$

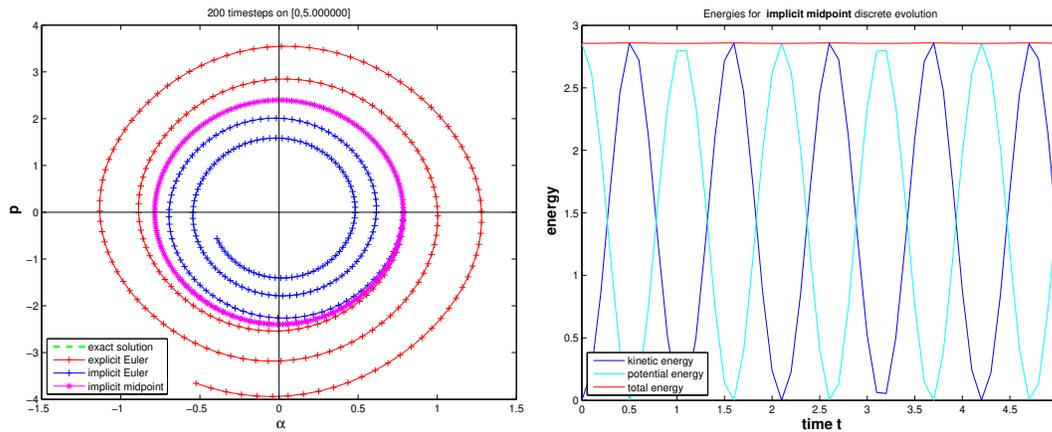
$E$  ist eine konstante Grösse in der Schwingung; wir würden also erwarten, dass die Lösung die Energieerhaltung erfüllt. Für eine harmonische Schwingung sollte der Plot im Phasenraum eine geschlossene Kurve ergeben, denn alle anderen Lösungen gewinnen oder verlieren Energie. Die Energie der Lösung, die wir durch das implizite Eulerverfahren erhalten haben, nimmt mit der Zeit aber immer weiter ab (die Trajektorie  $(\alpha, p)$  stürzt ins Zentrum  $(0, 0)$ ), während beim expliziten Eulerverfahren die Energie immer weiter zunimmt (die Trajektorie  $(\alpha, p)$  entfernt sich immer weiter vom Zentrum). In Theorem 2.2.6 a) und Theorem 2.2.6 b) ist die Energie als Funktion der Zeit aufgetragen um diesen Sachverhalt zu verdeutlichen. Beide Lösungen verletzen also die Energieerhaltung und stellen deswegen ein falsches Bild des physikalischen Systems dar.



a) expliziter Euler: Energiezunahme      b) impliziter Euler: Energieabnahme

**Abb. 2.2.6.** Energieverlauf der mittels Euler Verfahren berechneten Lösungen

Nun wollen wir die Lösung des Problems mit der impliziten Mittelpunktsregel betrachten. In Theorem 2.2.7 a) sind die drei Lösungen der drei Lösungsverfahren im Vergleich abgebildet, in Theorem 2.2.7 b) sehen wir den Verlauf der Energie für die implizite Mittelpunktsregel.



a) Vergleich e.E., i.E., i.M.

b) implizite Mittelpunktsregel:  
konstante Energie

**Abb. 2.2.7.** Strukturerhaltungseigenschaften der impliziten Mittelpunktsregel

Die implizite Mittelpunktsregel liefert tatsächlich ein besseres Ergebnis als die beiden Eulerverfahren. Selbst für relativ grosse Schrittweiten (die nicht für eine besonders gute Genauigkeit in der Lösung sorgen) bleibt die Energie erhalten; in Theorem 2.2.7 b) wurde  $h$  gross gewählt: daher kommen die "Knicke" in der Kurve.

Solche Ergebnisse können mit dem folgenden Code erhalten werden:

```

1  from numpy import linspace, cos, array, zeros, sin
3  from scipy.optimize import fsolve
   from matplotlib.pyplot import figure, grid, plot, title, xlabel, ylabel, show,
   legend
5
   #first the parameters
7  g = 9.81
   l = 1.0
9  t_init = 0.0
   t_end = 5.0
11 y_0 = array([0.5,0.0])

13 #the differential equation
   f = lambda y: array([y[1],-g/l*sin(y[0])])
15
   #formula for the energy:
17 E_pot = lambda y: l*g*(1.0 - cos(y[0]))
   E_kin = lambda y: 0.5*(y[1])**2
19 E = lambda y: E_pot(y) + E_kin(y)

21 #number of time steps, time steps and interval length
   N = 1000
23 t, h = linspace(t_init,t_end,N, retstep="True")

```

```

25 y_ExplE1 = zeros((N,2))
    y_ImplE1 = zeros((N,2))
27 y_ImplMp = zeros((N,2))

29 #set initial values
    y_ExplE1[0,:] = y_0
31 y_ImplE1[0,:] = y_0
    y_ImplMp[0,:] = y_0
33
35 #perform the integration
    for k in range(N-1):
        #explicit Euler:
37         y_ExplE1[k+1,:] = y_ExplE1[k,:] + h*f(y_ExplE1[k,:])
        #implicit Euler:
39         F = lambda x: x - y_ImplE1[k,:] - h*f(x)
            y_ImplE1[k+1,:] = fsolve(F, y_ImplE1[k,:]) #use yk as starting value for
            finding the root of F
41         #implicit middle point rule
            F = lambda x: x - y_ImplMp[k,:] - h*f(0.5*x + 0.5*y_ImplMp[k,:])
43         y_ImplMp[k+1,:] = fsolve(F, y_ImplMp[k,:]) #use yk as starting value for
            finding the root of F

45 #first we plot the two solutions in the phase space
    figure()
47 plot(y_ExplE1[:,0],y_ExplE1[:,1],"-+",label="Expl. Euler")
    plot(y_ImplE1[:,0],y_ImplE1[:,1],"-+",label="Impl. Euler")
49 plot(y_ImplMp[:,0],y_ImplMp[:,1],"-+",label="Impl. Mi.po")
    xlabel("alpha"); ylabel("p"); grid(True); legend(loc="best"); title("Movement in
    phase space")
51 show()

53 #then we plot the computed energies, which should be constant
    figure()
55 plot(t,array([E(y_ExplE1[k,:]) for k in range(N)]),label="Expl. Euler")
    plot(t,array([E(y_ImplE1[k,:]) for k in range(N)]),label="Impl. Euler")
57 plot(t,array([E(y_ImplMp[k,:]) for k in range(N)]),label="Impl. Mi.po")
    xlabel("t"); ylabel("E(t)"); grid(True); legend(loc="best"); title("Energies")
59 show()

61 figure()
    plot(t,array([E(y_ExplE1[k,:]) for k in range(N)]),label="total energy")
63 plot(t,array([E_pot(y_ExplE1[k,:]) for k in range(N)]),label="potential energy")
    plot(t,array([E_kin(y_ExplE1[k,:]) for k in range(N)]),label="kinetic energy")
65 xlabel("t"); ylabel("E(t)"); grid(True); legend(loc="best"); title("Expl Euler")
    show()

67 figure()
69 plot(t,array([E(y_ImplE1[k,:]) for k in range(N)]),label="total energy")
    plot(t,array([E_pot(y_ImplE1[k,:]) for k in range(N)]),label="potential energy")
71 plot(t,array([E_kin(y_ImplE1[k,:]) for k in range(N)]),label="kinetic energy")
    xlabel("t"); ylabel("E(t)"); grid(True); legend(loc="best"); title("Impl Euler")
73 show()

75 figure()
    plot(t,array([E(y_ImplMp[k,:]) for k in range(N)]),label="total energy")

```

```

77 plot(t,array([E_pot(y_ImplMp[k,:]) for k in range(N)]),label="potential energy")
plot(t,array([E_kin(y_ImplMp[k,:]) for k in range(N)]),label="kinetic energy")
79 xlabel("t"); ylabel("E(t)"); grid(True); legend(loc="best"); title("Impl
Midpoint")
show()

```

**Frage:** Besitzt die implizite Mittelpunktsregel die allgemeine Eigenschaft der Erhaltung von Konstanten, d.h. Ersten Integralen, der Differentialgleichung? Die Antwort wird von dem folgenden Satz geliefert:

**Satz 2.2.8.** (Erhaltungseigenschaften der impliziten Mittelpunktsregel)

Sei  $I : \mathbb{R}^d \supset D \rightarrow \mathbb{R}$  von der Form

$$I(\mathbf{y}) = \frac{1}{2} \mathbf{y}^T \mathbf{B} \mathbf{y} \quad \text{mit } \mathbf{B} \in \mathbb{R}^{d \times d}.$$

ein erstes Integral der autonomen Differentialgleichung  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  mit  $\mathbf{f}$  differenzierbar. Dann gilt:  $I(\mathbf{y}_k) = I(\mathbf{y}_0)$  für alle  $k \in \mathbb{N}$ , falls die  $\mathbf{y}_k$  aus der impliziten Mittelpunktsregel gewonnen wurden.

*Proof.* Sei  $I$  sodass  $\frac{dI}{dt} = 0$ ;  $I$  sei also erhalten.

Nach dem Theorem 2.1.27 gilt:

$$\frac{d}{dt} I(\mathbf{y}(t)) = 0 \quad \iff \quad (\text{grad } I(\mathbf{y})) \cdot \mathbf{f}(\mathbf{y}) = 0 \quad \forall \mathbf{y}.$$

Der Gradient ist aber gegeben durch

$$\text{grad } I(\mathbf{y}) = \text{grad } \frac{1}{2} \mathbf{y}^T \mathbf{B} \mathbf{y} = \frac{1}{2} ((\mathbf{y}^T \mathbf{B})^T + \mathbf{B} \mathbf{y}) = \mathbf{A} \mathbf{y} \quad \text{mit } \mathbf{A} = \frac{1}{2} (\mathbf{B} + \mathbf{B}^T) \text{ symmetrisch.}$$

Es gilt also  $\mathbf{A} \mathbf{y} \mathbf{f}(\mathbf{y}) = 0$ .

Die implizite Mittelpunktsregel  $\mathbf{y}_{k+1} = \mathbf{y}_k + h \mathbf{f}(\frac{1}{2}(\mathbf{y}_{k+1} + \mathbf{y}_k))$  ergibt dann

$$I(\mathbf{y}_{k+1}) - I(\mathbf{y}_k) = \frac{1}{2} (\mathbf{y}_{k+1} + \mathbf{y}_k)^T \mathbf{A} (\mathbf{y}_{k+1} - \mathbf{y}_k) = \mathbf{A} \mathbf{z}_k \cdot h \mathbf{f}(\mathbf{z}_k) = 0,$$

mit  $\mathbf{z}_k = \frac{1}{2}(\mathbf{y}_{k+1} + \mathbf{y}_k)$ . Also  $\mathbf{y}_{k+1} = \mathbf{y}_k$ . Damit ist der Beweis fertig.  $\square$

**Bemerkung 2.2.9.** Die Erhaltungseigenschaften können in manchen Anwendungen (Astrophysik, Quantenmechanik) wichtiger sein als die Genauigkeit der Lösung. In den Beispielen weiter unten werden wir sehen, dass auch hochgenaue Verfahren unphysikalische Lösungen nach langen Zeiten liefern können, wo bescheidene aber strukturerhaltende Methoden wie die implizite Mittelpunktsregel ausgezeichnete Dienste leisten.

## 2.3 Störmer-Verlet-Verfahren

Wir haben gesehen, dass wir durch die Einführung des Vektors  $[y(t), \dot{y}(t)]^T$  eine Differentialgleichung 2. Ordnung in eine Differentialgleichung 1. Ordnung überführen können. In manchen Fällen ergibt es allerdings Sinn, direkt mit Verfahren zu arbeiten, die sich spezifisch für Differentialgleichungen 2. Ordnung eignen.

Wir übertragen die Idee des Polygonzug-Verfahrens auf Differentialgleichungen 2. Ordnung:

$$\ddot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y}). \quad (2.3.21)$$

Seien  $\mathbf{y}_{k-1} \approx \mathbf{y}(t_{k-1})$  und  $\mathbf{y}_k \approx \mathbf{y}(t_k)$  gegeben; wir approximieren  $\mathbf{y}(t)$  auf  $[t_{k-1}, t_{k+1}]$  durch eine Parabel  $\mathbf{p}(t)$ , die durch  $(t_{k-1}, \mathbf{y}_{k-1})$ , und  $(t_k, \mathbf{y}_k)$  geht; dabei verlangen wir, dass die Differentialgleichung zur Zeit  $t_k$  erfüllt ist:  $\ddot{\mathbf{p}}(t_k) = \mathbf{f}(t_k, \mathbf{y}_k)$ . Diese Parabel ist eindeutig bestimmt und wir setzen dann

$$\mathbf{y}_{k+1} = \mathbf{p}(t_{k+1}) \approx \mathbf{y}(t_{k+1}).$$

Das ergibt das Störmer-Verlet-Verfahren für (2.3.21):

$$\mathbf{y}_{k+1} = -\frac{h_k}{h_{k-1}} \mathbf{y}_{k-1} + \left(1 + \frac{h_k}{h_{k-1}}\right) \mathbf{y}_k + \frac{1}{2} (h_k^2 + h_k h_{k-1}) \mathbf{f}(t_k, \mathbf{y}_k),$$

für  $k = 1, \dots, N-1$ .

Für uniforme Zeitschrittweite  $h$  lässt sich diese Formel vereinfachen:

$$\mathbf{y}_{k+1} = -\mathbf{y}_{k-1} + 2\mathbf{y}_k + h^2 \mathbf{f}(t_k, \mathbf{y}_k), \quad k = 1, \dots, N-1. \quad (2.3.22)$$

Wir haben es also mit einem *expliziten* Verfahren zu tun, das die Approximation zu zwei vorigen Zeiten braucht, um zu funktionieren. Dementsprechend brauchen wir auch zwei Startwerte, die aus den Anfangsbedingungen  $\mathbf{y}(0) = \mathbf{y}_0$ ,  $\dot{\mathbf{y}}(0) = \mathbf{v}_0$  zu holen sind. Dazu benutzen wir einen virtuellen Zeitpunkt  $t_{-1} := t_0 - h_0$ . Das Verfahren auf  $[t_{-1}, t_1]$  ergibt:

$$\mathbf{y}_1 = -\mathbf{y}_{-1} + 2\mathbf{y}_0 + h_0^2 \mathbf{f}(t_0, \mathbf{y}_0),$$

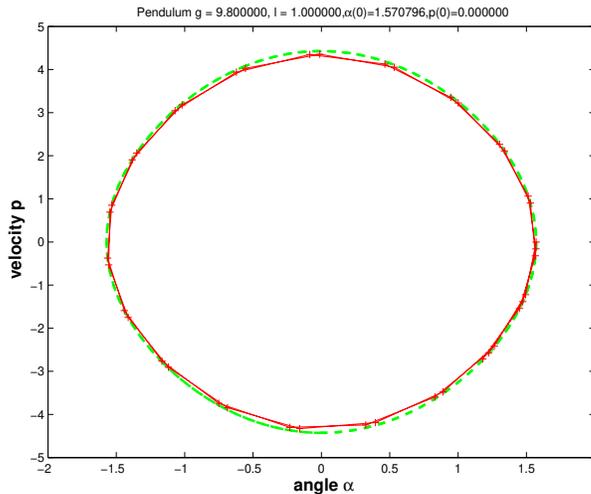
während der zentrale Differenzenquotient auf  $[t_{-1}, t_1]$  uns gibt:

$$\frac{\mathbf{y}_1 - \mathbf{y}_{-1}}{2h_0} = \mathbf{v}_0.$$

Somit ist

$$\mathbf{y}_1 = \mathbf{y}_0 + h_0 \mathbf{v}_0 + \frac{h_0^2}{2} \mathbf{f}(t_0, \mathbf{y}_0).$$

Wir beobachten eine fast perfekte Erhaltung der Energie (die Energie driftet nicht, sondern oszilliert ein bisschen). Vor allem deswegen wird dieses Verfahren oft für die numerische Simulation von konservativen mechanischen Systemen gebraucht.



Das mathematische Pendel mit  $a_0 = \pi/2$ ,  $p_0 = 0$ ,  $T = 5$  wurde mit dem Störmer-Verlet-Verfahren mit  $N = 40$  Zeitschritte und die Referenzlösung mittels ode45 mit extrem kleinen Toleranzen berechnet.

Abb. 2.3.1. Orbiten im Phasenraum

Der Code dazu:

```

1  from numpy import linspace, cos, array, zeros
2  from matplotlib.pyplot import figure, grid, plot, title, xlabel, ylabel, show,
3  legend
4
5  #first the parameters
6  g = 9.81
7  l = 1.0
8  t_init = 0.0
9  t_end = 5.0
10
11 #the differential equation
12 f = lambda y: -g/l*sin(y) #now we just need the expression for the second
13 derivative
14
15 #formula for the energy:
16 E_pot = lambda y: l*g*(1.0 - cos(y))
17 E_kin = lambda v: 0.5*(v)**2
18 E = lambda y,v: E_pot(y) + E_kin(v)
19
20 #number of time steps, time steps and interval length
21 N = 40
22 t, h = linspace(t_init,t_end,N, retstep="True")
23
24 y_StV = zeros(N) #We write y but the parameter we are referring to is alpha
25 v_StV = zeros(N)
26
27 #set initial values
28 y_StV[0] = 0.5
29 v_StV[0] = 0.0
30
31 #first step with explicit euler; otherwise we could define y_{-1}
32 y_StV[1] = y_StV[0] + h*v_StV[0]
33 #perform the integration

```

```

33 for k in range(1,N-1):
    y_StV[k+1] = -y_StV[k-1] + 2.0*y_StV[k] + (h**2)*f(y_StV[k])
    v_StV[k] = (y_StV[k+1] - y_StV[k-1])/(2.0*h)
35
37 #when plotting we discard  $y_t=tend$  because we did not calculate  $v$  here
#(One could approximate it but it is not strictly necessary for the purpose of
this demonstration)
#first we plot the two solutions in the phase space
39 figure()
plot(y_StV[:-1],v_StV[:-1],label="St\former Verlet")
41 xlabel("alpha"); ylabel("p"); grid(True); legend(loc="best"); title("Movement in
phase space")
show()
43
#then we plot the computed energies, which should be constant
45 figure()
plot(t[:-1],E_pot(y_StV[:-1]),label="St\former Verlet, Epot")
47 plot(t[:-1],E_kin(v_StV[:-1]), label="St\former Verlet, Ekin")
plot(t[:-1],E(y_StV[:-1],v_StV[:-1]),label="St\former Verlet, Etot")
49 xlabel("t"); ylabel("E(t)"); grid(True); legend(loc="best"); title("Energies")
show()

```

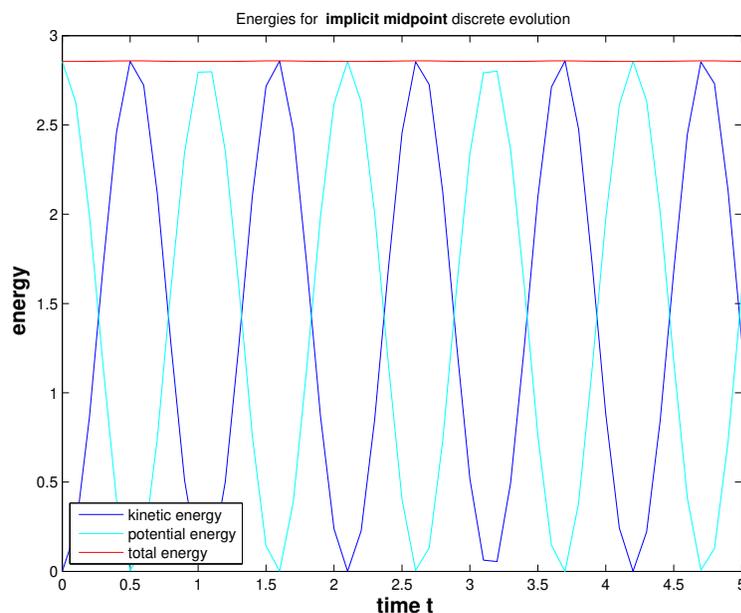


Abb. 2.3.2. Störmer-Verlet: keine Energiedrift trotz grosser Zeitschrittweite

**Bemerkung 2.3.3.** Wir können das Störmer-Verlet-Verfahren auch mittels Differenzen-Quotienten erhalten:

$$\mathbf{f}(t_k, \mathbf{y}(t_k)) = \ddot{\mathbf{y}}(t_k) \approx \frac{\frac{\mathbf{y}(t_{k+1}) - \mathbf{y}(t_k)}{h} - \frac{\mathbf{y}(t_k) - \mathbf{y}(t_{k-1})}{h}}{h} = \frac{\mathbf{y}(t_{k+1}) - 2\mathbf{y}(t_k) + \mathbf{y}(t_{k-1}))}{h^2},$$

und den zweiten Startwert  $\mathbf{y}_1$  aus der Taylor-Approximation zweiter Ordnung berechnen.

**Bemerkung 2.3.4.** (Einschrittformulierung des Störmer-Verlet-Verfahrens)

Für uniforme Zeitschrittweite können wir analog zur Umwandlung einer Differentialgleichung 2. Ordnung in eine Differentialgleichung 1. Ordnung das Zweischnittverfahren umformulieren. Die Geschwindigkeit in der Mitte von  $[t_k, t_{k+1}]$  wird approximiert durch  $\mathbf{v}_{k+\frac{1}{2}} := \frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h}$ , und  $\mathbf{v}_k$  wird aus der Gleichung für Geschwindigkeit berechnet:

$$\begin{array}{ccc}
 \ddot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) & \longleftrightarrow & \dot{\mathbf{y}} = \mathbf{v}, \\
 & & \dot{\mathbf{v}} = \mathbf{f}(\mathbf{y}) \\
 \updownarrow & & \updownarrow \\
 \mathbf{y}_{k+1} - 2\mathbf{y}_k + \mathbf{y}_{k-1} = h^2 \mathbf{f}(\mathbf{y}_k) & \longleftrightarrow & \begin{array}{l} \mathbf{v}_{k+\frac{1}{2}} = \mathbf{v}_k + \frac{h}{2} \mathbf{f}(\mathbf{y}_k), \\ \mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{v}_{k+\frac{1}{2}}, \\ \mathbf{v}_{k+1} = \mathbf{v}_{k+\frac{1}{2}} + \frac{h}{2} \mathbf{f}(\mathbf{y}_{k+1}), \end{array}
 \end{array}$$

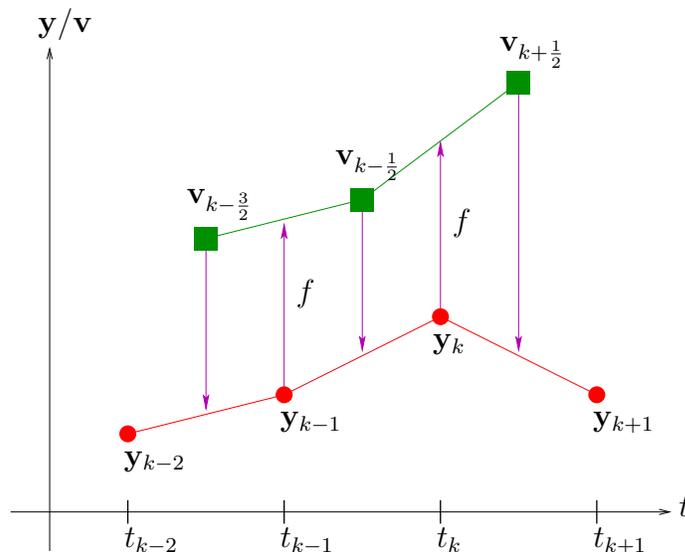
Zweischnittverfahren

Einschnittverfahren

**Bemerkung 2.3.5.** Die Einschritt-Formulierung des Störmer-Verlet-Verfahrens wird als Iteration implementiert mit dem Schritt

$$\begin{aligned}
 \mathbf{v}_{k+\frac{1}{2}} &= \mathbf{v}_{k-\frac{1}{2}} + h\mathbf{f}(\mathbf{y}_k), \\
 \mathbf{y}_{k+1} &= \mathbf{y}_k + h\mathbf{v}_{k+\frac{1}{2}},
 \end{aligned}$$

was wie eine Polygonzugmethode auf zwei versetzten Gittern (staggered grid) aussieht.



**Abb. 2.3.6.** Perspektive: Störmer-Verlet-Verfahren als Einschrittverfahren

Vor allem in Verbindung mit partiellen Differentialgleichungen ist diese Formulierung als *leap-frog method* bekannt. Diese Formulierung ist deutlich weniger anfällig für Rundungsfehler als die ursprüngliche Zwei-Schritt-Formulierung. Die Geschwindigkeiten werden allerdings nur an den Zeiten  $t_k + \frac{h}{2}$  explizit berechnet; werden diese auch an  $t_k$  benötigt, z.B. für die Berechnung der Energien, dann muss man diese noch durch  $(\mathbf{v}_{k+\frac{1}{2}} + \mathbf{v}_{k-\frac{1}{2}})/2$  berechnen. Noch geschickter ist die Formulierung als *velocity-Verlet method*:

$$\begin{aligned} \mathbf{y}_{k+1} &= \mathbf{y}_k + h\mathbf{v}_k + \frac{h^2}{2}f(t_k, \mathbf{y}_k) \\ \mathbf{v}_{k+1} &= \mathbf{v}_k + \frac{h}{2}(f(t_k, \mathbf{y}_k) + f(t_{k+1}, \mathbf{y}_{k+1})). \end{aligned} \quad (2.3.23)$$

Wir können diese Formulierung erhalten, indem wir dieselbe Idee für den Startwert verwenden: der zentrale Differenzquotient für  $\mathbf{v}_k$  ergibt  $\mathbf{y}_{k-1} = \mathbf{y}_{k+1} - 2h\mathbf{v}_k$  was zusammen mit (2.3.22) die erste Gleichung liefert. Auch der zentrale Differenzquotient für  $\mathbf{v}_k$  und (2.3.22) ergibt  $\mathbf{v}_k = \frac{\mathbf{y}_k - \mathbf{y}_{k-1}}{h} + \frac{h}{2}f(t_k, \mathbf{y}_k)$ ; wenn wir diesen Ausdruck für jeden Term in  $\mathbf{v}_{k+1} + \mathbf{v}_k$  verwenden und dann wieder den zentralen Differenzquotienten für  $\mathbf{v}_k$  betrachten, erhalten wir die zweite Gleichung.

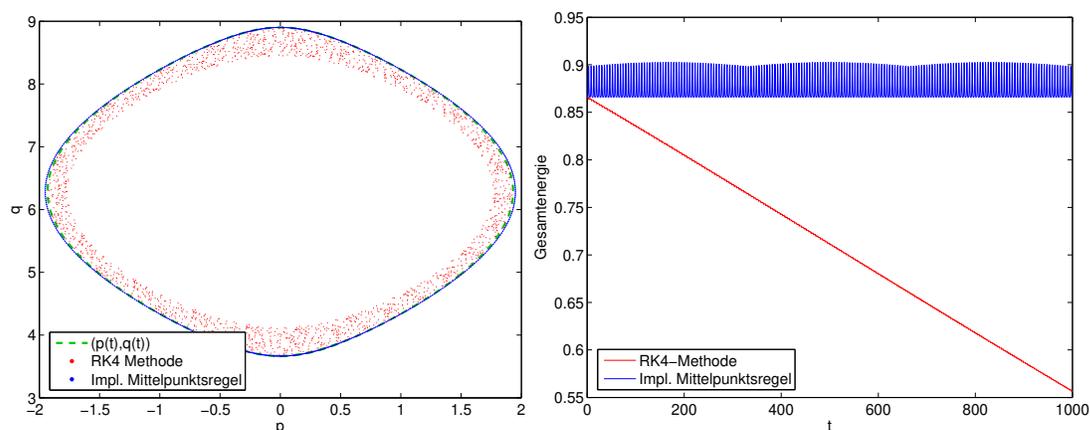
## Beispiele

Wir betrachten hier die Energieerhaltung in einigen physikalischen Beispielen, die auch mit Verfahren gelöst werden die wir später besprechen werden.

### Beispiel 2.3.7. (Mathematisches Pendel)

Wir simulieren das mathematische Pendel (2.1.14) im Zeitintervall  $[0, 1000]$ , mit Startwerten  $p(0) = 0$ ,  $q(0) = 7\pi/6$ .

Wir vergleichen das klassische Runge-Kutta-Verfahren (2.5.36) der Ordnung 4 mit der impliziten Mittelpunktsregel bei konstanter Zeitschrittweite  $h = \frac{1}{2}$ :



Trajektorien “exakter”/diskreter  
Evolutionen

Energieverhalten diskreter  
Evolutionen

**Abb. 2.3.8.** Runge Kutta versus implizite Mittelpunktsregel

Wir können die guten Strukturerhaltungseigenschaften der impliziten Mittelpunktsregel sofort erkennen.

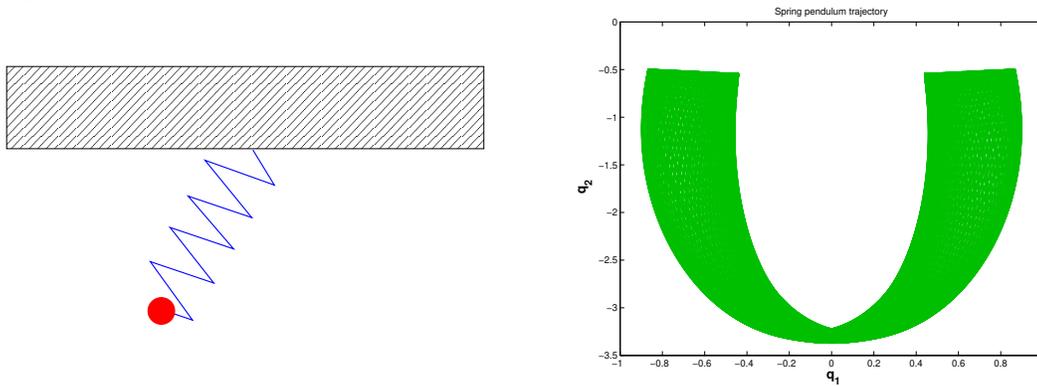
**Beispiel 2.3.9.** (Federpendel)

Die Hamilton-Funktion (Energie) ist  $H(\mathbf{p}, \mathbf{q}) = \frac{1}{2} \|\mathbf{p}\|^2 + \frac{1}{2} (\|\mathbf{q}\| - 1)^2 + q_2$ , wobei  $\mathbf{q} \hat{=}$  Position,  $\mathbf{p} \hat{=}$  Impuls, und die Differentialgleichung lautet (die Herleitung solcher Gleichungen wird in der Vorlesung "Allgemeine Mechanik" besprochen):

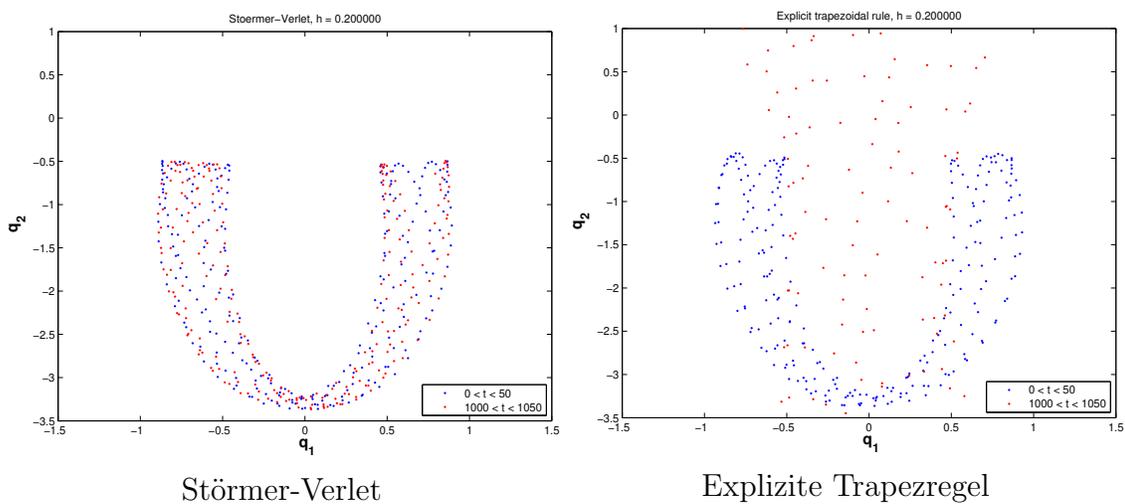
$$\dot{\mathbf{p}} = -\frac{dH}{d\mathbf{q}} = -(\|\mathbf{q}\| - 1) \frac{\mathbf{q}}{\|\mathbf{q}\|} - \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

$$\dot{\mathbf{q}} = \frac{dH}{d\mathbf{p}} \mathbf{p}.$$

Die physikalische Situation ist im Theorem 2.3.10 und die numerischen Ergebnisse für kurze und lange Zeiten und verschiedene numerische Methoden sind in der Theorem 2.3.11 zu sehen.



**Abb. 2.3.10.** Trajektorien für die Langzeit-Evolution des Federpendels



**Abb. 2.3.11.** Die diskreten Evolutionen verlassen das erlaubte Gebiet, wenn wir das System mit der expliziten Trapezregel integrieren

Der Code dazu:

```

1  from numpy import linspace, cos, array, zeros, sqrt
2  from numpy.linalg import norm
3  from matplotlib.pyplot import figure, grid, plot, title, xlabel, ylabel, show,
   legend
4  #from matplotlib import rcParams
5
6  #the differential equation for explicit Trapez-Rule (and in general every method
   for 1st order ODEs)
7  #we work with the following vector: [q1,q2,p1,p2]
8  f = lambda qp: array([qp[2],qp[3],])
9  def f(qp):
10     normq = norm(qp[:-2])
11     if(normq==0): print("Error: division by zero encountered"); normq += 0.0001
12     return array([qp[2],qp[3], -(normq-1)*qp[0]/normq,-(normq-1.0)*qp[1]/normq -
13     1.0])
14
15 #the differential equation for Störmer-Verlet
16 #now we just need the expression for the second derivative
17 fStV = lambda q: -((norm(q)-1.0)/norm(q))*q - array([0,1])
18
19 #number of time steps, (fixed) interval length
20 t_init = 0.0; t_end = 1050. #long time
21 h = 0.2
22 N = int((t_end-t_init)/h)
23
24 q_StV = zeros((N,2))
25 p_StV = zeros((N,2))
26
27 qp_extrap = zeros((N,4))
28 qp_extrap[0,:] = array([sqrt(3.)/2,-0.5,0,0])
29
30 #set initial values
31 q_StV[0,:] = array([sqrt(3.)/2,-0.5]) #initial angle = pi/6 (and l=1)
32 p_StV[0] = array([0,0]) #initially at rest
33
34 #notice: formally p=v since m=1
35
36 #St-Verlet: first step with explicit euler; otherwise we could define y_{-1}
37 q_StV[1,:] = q_StV[0,:] + h*p_StV[0,:]
38
39 #perform the integration: St-Verlet
40 for k in range(1,N-1):
41     q_StV[k+1,:] = -q_StV[k-1,:] + 2.0*q_StV[k,:] + (h**2)*fStV(q_StV[k,:])
42     p_StV[k,:] = (q_StV[k+1,:] - q_StV[k-1,:])/(2.0*h)
43
44 #perform the integration: explicit trapezoidal rule
45 for k in range(N-1):
46     qptilde = qp_extrap[k,:] + h*f(qp_extrap[k,:])
47     qp_extrap[k+1,:] = qp_extrap[k,:] + 0.5*h*(f(qp_extrap[k,:]) + f(qptilde))
48
49 #when plotting we discard y_t=tend because we did not calculate v here
50 #(one could approximate it but it is not strictly necessary for the purpose of
   this demonstration)
51 #first we plot the two solutions in the phase space

```

```
51 figure()
   plot(q_StV[:int(50/h),0],q_StV[:int(50/h),1],"o",markersize=2,label="$0<t<50$")
53 plot(q_StV[int((t_end-50)/h):-1,0],q_StV[int((t_end-50)/h):-1,1],"o",markersize=2,label="$1000<t<1050$")
   xlabel("q1"); ylabel("q2"); grid(True); legend(loc="best"); title("Trajectory,
   St\\"ormer-Verlet")
55
   show()
57
   figure()
59 plot(qp_extrap[:int(50/h),0],qp_extrap[:int(50/h),1],"o",markersize=2,label="$0<t<50$")
   plot(qp_extrap[int((t_end-50)/h):-1,0],qp_extrap[int((t_end-50)/h):-1,1],"o",markersize=2,label="$1000<t<1050$")
61 xlabel("q1"); ylabel("q2"); grid(True); legend(loc="best"); title("Trajectory,
   Explicit Trapezoidal Rule")
   show()
```

**Beispiel 2.3.12.** (Molekulardynamik)

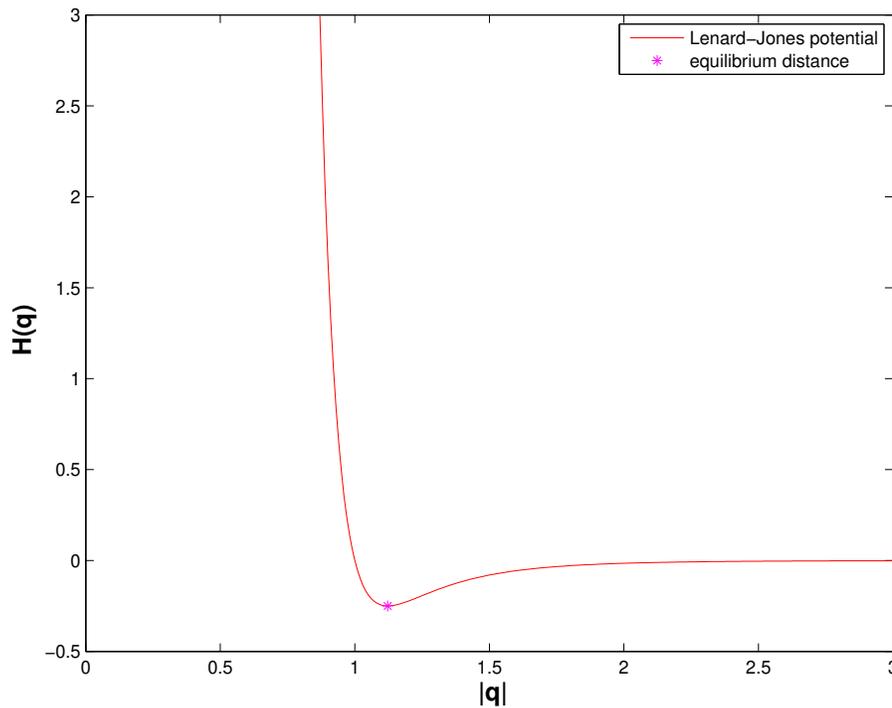
Für  $n \in \mathbb{N}$  Atome der Masse  $m = 1$  in  $d \in \mathbb{N}$  Dimensionen brauchen wir den Phasenraum  $D = \mathbb{R}^{2dn}$

(Positionen  $\mathbf{q} = [\mathbf{q}^1; \dots; \mathbf{q}^n]^T \in \mathbb{R}^{dn}$ , und Impulse  $\mathbf{p} = [\mathbf{p}^1; \dots; \mathbf{p}^n]^T \in \mathbb{R}^{dn}$ ) und wir haben die Energie (Hamilton-Funktion):

$$H(\mathbf{p}, \mathbf{q}) = \frac{1}{2} \|\mathbf{p}\|_2^2 + V(\mathbf{q}),$$

mit dem Lennard-Jones-Potential:

$$V(\mathbf{q}) = \sum_{j=1}^n \sum_{i \neq j} \mathcal{V}(\|\mathbf{q}^i - \mathbf{q}^j\|_2), \quad \mathcal{V}(\xi) = \xi^{-12} - \xi^{-6}. \quad (2.3.24)$$



**Abb. 2.3.13.** Lennard-Jones-Potential

Das Hamiltonische System ist:

$$\begin{aligned} \dot{\mathbf{p}}^j &= - \sum_{i \neq j} \mathcal{V}'(\|\mathbf{q}^j - \mathbf{q}^i\|_2) \frac{\mathbf{q}^j - \mathbf{q}^i}{\|\mathbf{q}^j - \mathbf{q}^i\|_2} \\ \dot{\mathbf{q}}^j &= \mathbf{p}^j \quad j = 1, \dots, n. \end{aligned}$$

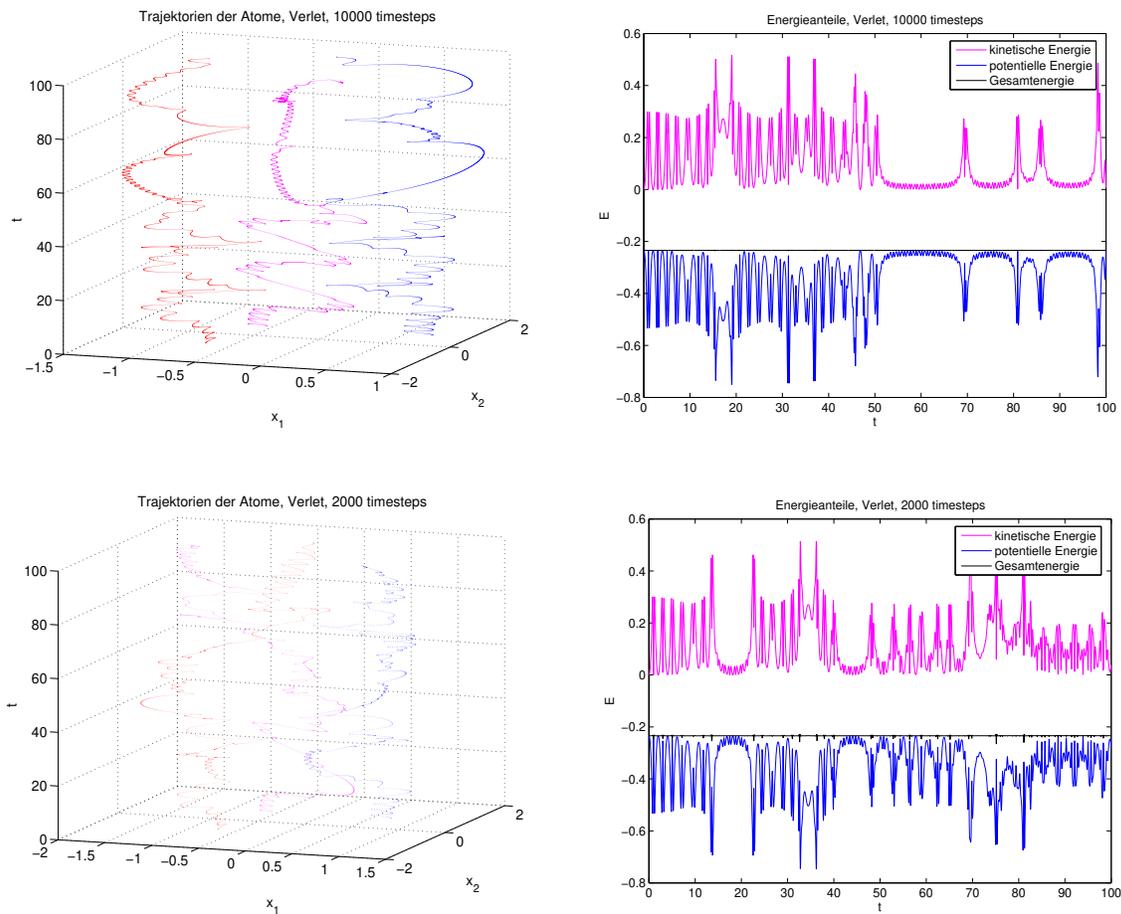
Das Störmer-Verlet-Verfahren ist hier:

$$\mathbf{q}_h\left(t + \frac{1}{2}h\right) = \mathbf{q}_h(t) + \frac{h}{2} \mathbf{p}_h(t)$$

$$\mathbf{p}_h^j(t+h) = \mathbf{p}_h^j(t) - h \sum_{i \neq j} \mathcal{V}'\left(\left\|\mathbf{q}_h^j\left(t + \frac{1}{2}h\right) - \mathbf{q}_h^i\left(t + \frac{1}{2}h\right)\right\|_2\right) \frac{\mathbf{q}_h^j\left(t + \frac{1}{2}h\right) - \mathbf{q}_h^i\left(t + \frac{1}{2}h\right)}{\left\|\mathbf{q}_h^j\left(t + \frac{1}{2}h\right) - \mathbf{q}_h^i\left(t + \frac{1}{2}h\right)\right\|_2}$$

$$\mathbf{q}_h(t+h) = \mathbf{q}_h\left(t + \frac{1}{2}h\right) + \frac{h}{2} \mathbf{p}_h(t+h).$$

Anbei die Simulationsergebnisse für  $d = 2$ ,  $n = 3$ ,  $\mathbf{q}^1(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} -1 \\ -1 \end{pmatrix}$ ,  $\mathbf{q}^2(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ ,  $\mathbf{q}^3(0) = \frac{1}{2}\sqrt{2}\begin{pmatrix} -1 \\ 1 \end{pmatrix}$ ,  $\mathbf{p}(0) = 0$  und die Endzeit  $T = 100$ .



**Abb. 2.3.14.** Störmer-Verlet für Lennard-Jones: Trajektorien und Energien

Wir beobachten sehr unterschiedliche Trajektorien der Atome, wenn wir verschiedene Zeitschritte verwenden, aber die Energie bleibt erhalten, was für eine gute qualitative Beschreibung des Systems spricht.

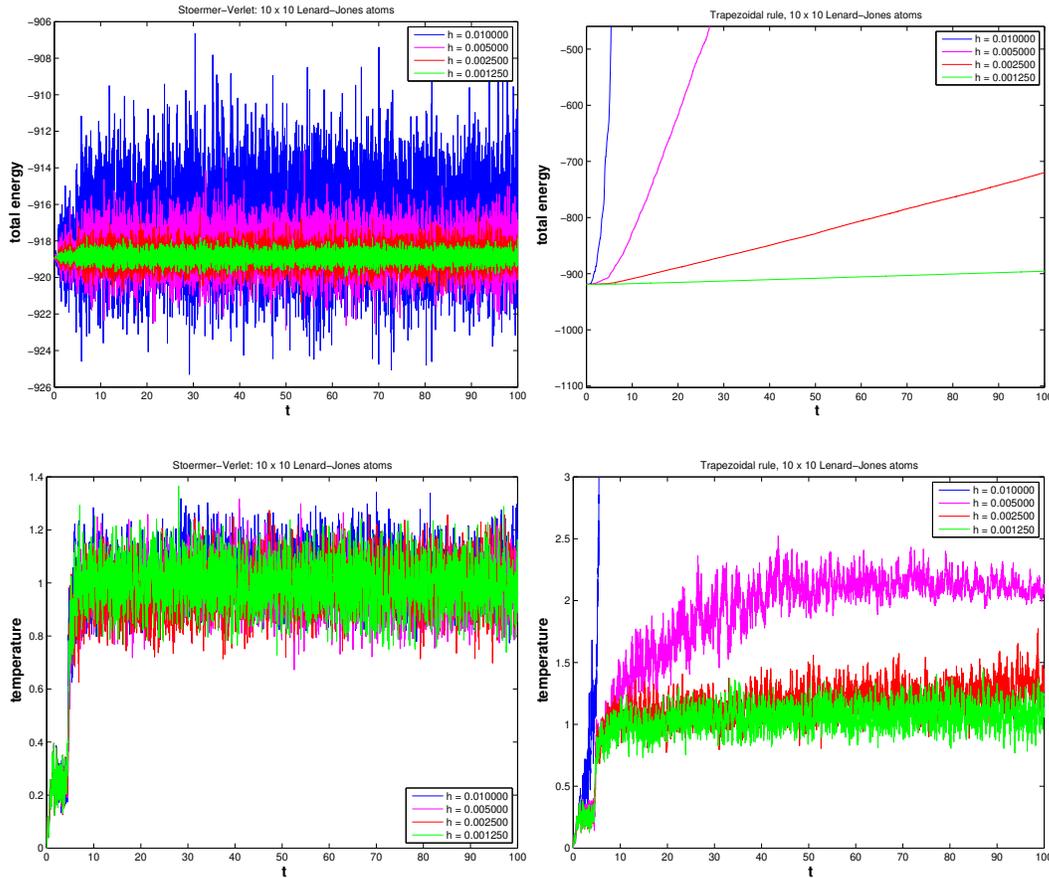


Abb. 2.3.15. Störmer-Verlet ergibt ein qualitativ korrektes Verhalten der Temperatur eines Körpers auch bei grösseren Zeitschrittweiten, was die explizite Trapezregel nicht kann.

## 2.4 Splitting-Verfahren

Manchmal kann man ein Problem zum Teil analytisch lösen; es ist in der Regel eine gute Idee, die analytischen Ergebnisse soweit wie möglich zu benutzen. Das Splitting-Verfahren beruht auf diesem Prinzip.

Wir betrachten hier autonome Anfangswertprobleme, wo wir die rechte Seite als eine Summe schreiben:

$$\dot{\mathbf{y}} = \mathbf{f}_a(\mathbf{y}) + \mathbf{f}_b(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \quad (2.4.25)$$

mit  $\mathbf{f}_a: D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$ ,  $\mathbf{f}_b: D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$  "glatt". Wir nehmen zunächst an, dass die Evolutionsoperatoren

$$\begin{aligned} \Phi_a^t &\leftrightarrow \text{für } \dot{\mathbf{y}} = \mathbf{f}_a(\mathbf{y}), \\ \Phi_b^t &\leftrightarrow \text{für } \dot{\mathbf{y}} = \mathbf{f}_b(\mathbf{y}) \end{aligned}$$

analytisch bekannt sind. Wir können dann einen ESV bauen, indem wir diese zwei Operatoren kombinieren. Zum Beispiel kann man einen Zeitschritt folgendermassen schreiben:

$$\text{Lie-Trotter-Splitting: } \Psi_1^h = \Phi_b^h \circ \Phi_a^h \text{ oder } \Psi_1^h = \Phi_a^h \circ \Phi_b^h \quad (2.4.26)$$

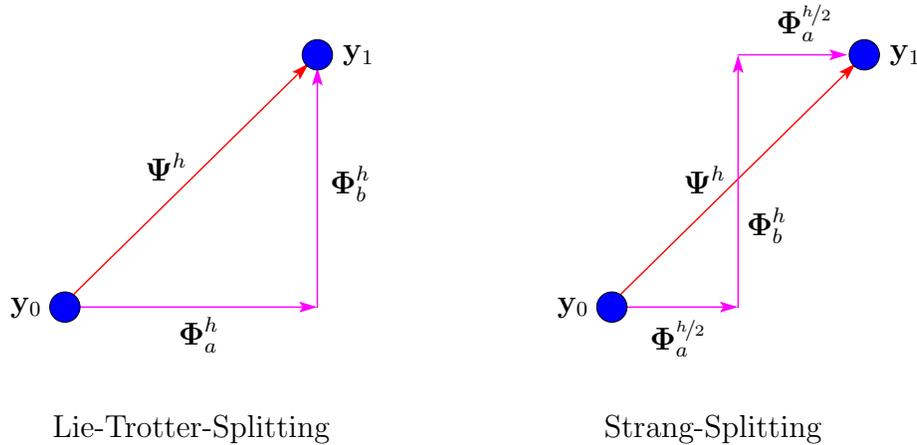
$$\text{Strang-Splitting: } \Psi_2^h = \Phi_a^{h/2} \circ \Phi_b^h \circ \Phi_a^{h/2}, \quad (2.4.27)$$

Diese zwei Verfahren besitzen global die Ordnungen  $\mathcal{O}(h)$  und  $\mathcal{O}(h^2)$ . Im allgemeinen schreibt sich ein Zeitschritt eines Splitting-Verfahrens als

$$\Psi_s^h = \prod_{i=1}^s \Phi_b^{b_i h} \Phi_a^{a_i h}, \quad (2.4.28)$$

wobei wir mindestens annehmen, dass  $\sum_{i=1}^s b_i = 1$  und  $\sum_{i=1}^s a_i = 1$ . Die Wahl der Konstanten  $a_i$  und  $b_i$  entspricht einer Zerlegung (splitting) in Unterschritte und ergibt Methoden mit verschiedenen Eigenschaften.

Wichtig: diese Verfahren eignen sich nur für autonome Systeme. Im Falle eines nicht autonomen Systems muss man die Differentialgleichung zuerst autonomisieren, siehe 2.1.5.



**Beispiel 2.4.1.** (Konvergenz einfacher Splitting-Verfahren)

Sei

$$\dot{y} = \underbrace{\lambda y(1-y)}_{=:f_a(y)} + \underbrace{\sqrt{1-y^2}}_{=:f_b(y)}, \quad y(0) = 0.$$

Die Evolutionsoperatoren der zwei Teile sind analytisch bekannt:

$$\Phi_a^t y = \frac{1}{1 + (y^{-1} - 1)e^{-\lambda t}}, \quad \text{für } t > 0, y \in ]0, 1] \text{ und}$$

$$\Phi_b^t y = \begin{cases} \sin(t + \arcsin(y)), & \text{wenn } t + \arcsin(y) < \frac{\pi}{2}, \\ 1, & \text{sonst} \end{cases} \quad \text{für } t > 0, y \in [0, 1].$$

Wir wollen sofort einen kleinen Code schreiben, um die beiden Verfahren zu vergleichen. Dazu benutzen wir  $\lambda = 1$  und  $T = 1$ .

```

1  from numpy import exp, sin, arcsin, arange, zeros, pi
   from matplotlib.pyplot import figure, loglog, show, grid, title, xlabel, ylabel,
   legend
3
   y_1_exact = 0.9397605814894437 #high precision numerical value
5
   #first we define the two evolution operators
7  def Phia(y,h):
   return 1./(1. + exp(-h)*(-1. + 1./y))
9
11 def Phib(y,h):
   if (h+arcsin(y) < pi/2.): return sin(h+arcsin(y))
   return 1.0
13
   #next we define the single step evolution operators based on Lie-Trotter and
   Strang Splitting
15 def Lt(y,h):
   return Phia(Phib(y,h),h)
17
19 def Strang(y,h):
   return Phib(Phia(Phib(y,h/2.),h),h/2.)
21
   T = 1.0 #final time: T=1
   y0 = 0.0
23   Ns = arange(10,1500,30)
   err_Lt = []
25   err_Strang = []
27
   #now we calculate the solutions and the error.
   #We do not save all the values of the function, since we are only interested in
   y(1)
29   for N in Ns:
   h = T/N
31   y_Lt = y0
   y_Strang = y0
33   for k in range(N):
   y_Lt = Lt(y_Lt,h)
35   y_Strang = Strang(y_Strang,h)
   err_Lt += [abs(y_Lt-y_1_exact)]
37   err_Strang += [abs(y_Strang-y_1_exact)]
39
   figure()
   title("Convergence of Lie-Trotter and Strang Splitting")
41   grid(True);xlabel("$h$"); ylabel("err")
   loglog(T/Ns,(T/Ns), "-", label = "$O(h)$")
43   loglog(T/Ns,(T/Ns)**2, "-", label = "$O(h^2)$")
   loglog(T/Ns,err_Lt,"x-", label = "Lie-Trotter")
45   loglog(T/Ns,err_Strang,"x-", label = "Strang")
   legend(loc="best")
47   show()

```

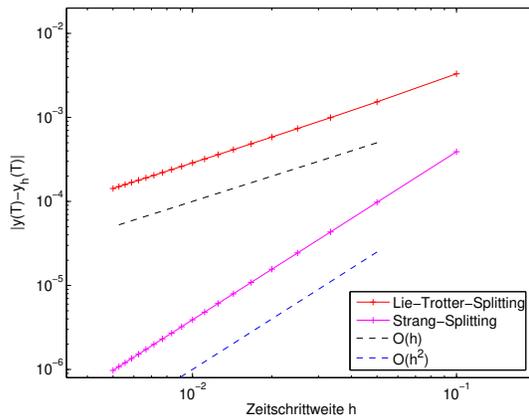


Abb. 2.4.2. Fehler zur Endzeit  $T = 1$ .

Numerisches Experiment:

$T = 1$ ,  $\lambda = 1$ , Vergleich mit einer sehr genauen numerischen Lösung.

**Bemerkung 2.4.3.** Wenn die exakten Evolutionsoperatoren der Teile nicht bekannt sind, werden wir diskrete Versionen verwenden (zum Beispiel  $\Psi_a^h$  statt  $\Phi_a^h$ ).

**Beispiel 2.4.4.** (Splitting mit diskreten Evolutionsoperatoren)

Für die Differentialgleichung im Beispiel Theorem 2.4.1, kombinieren wir verschiedene Evolutionsoperatoren:

```

1  from numpy import exp, sin, arcsin, arange, zeros, pi, sqrt
2  from matplotlib.pyplot import figure, loglog, show, grid, title, xlabel, ylabel,
3  legend
4  from scipy.optimize import fsolve
5
6  y_1_exact = 0.9397605814894437 #high precision numerical value
7
8  #first we define the two differential equations, since we need them for the
9  discrete operators
10 def fa(y):
11     return y - y**2
12
13 def fb(y):
14     return sqrt(1-y**2)
15
16 #now the analytical integrators (evolution operators)
17 def Phia(y,h):
18     return 1./(1. + exp(-h)*(-1. + 1./y))
19
20 def Phib(y,h):
21     if (h+arcsin(y) < pi/2.): return sin(h+arcsin(y))
22     return 1.0
23
24 #now the discrete integrators
25 def expl_eul(f,y,h): #explicit euler
26     return y + h*f(y)
27
28 def impl_eul(f,y,h): #implicit euler
29     F = lambda ynew: ynew - y - h*f(ynew)

```

```

    return fsolve(F, y + h*f(y)) #explicit euler for first guess
29
def expl_mid(f,y,h): #explicit middle-point rule
31     y_mid = y + 0.5*h*f(y)
    return y + h*f(y_mid)
33
T = 1.0 #final time: T=1
35 y0 = 0.0
Ns = arange(10,1500,30)
37 err_Lt_explEul = [] #Lie Trotter with expl euler
err_Strang_explEul = [] #Strang with expl euler
39 err_Strang_explEul_Phib_implEul = [] #Strang with impl eul and expl eul for fa
and Phib for fb
err_Lt_explmid = [] #Lie Trotter with explicit midpoint
41 err_Strang_explmid = [] #Strang with explicit midpoint

43 #now we calculate the solutions and the error.
#We do not save all the values of the function, since we are only interested in
y(1)
45 for N in Ns:

47     h = T/N

49     #initialize
    y_Lt_explEul = y_Strang_explEul = y_Strang_explEul_Phib_implEul =
y_Lt_explmid = y_Strang_explmid = y0

51     for k in range(N):

53         #Lie Trotter with expl euler
55         y_Lt_explEul = expl_eul(fb, y_Lt_explEul, h)
        y_Lt_explEul = expl_eul(fa, y_Lt_explEul, h)

57         #Strang with expl euler
59         y_Strang_explEul = expl_eul(fa, y_Strang_explEul, h/2.)
        y_Strang_explEul = expl_eul(fb, y_Strang_explEul, h)
61         y_Strang_explEul = expl_eul(fa, y_Strang_explEul, h/2.)

63         #Strang with impl eul and expl eul for fa and Phib for fb
        y_Strang_explEul_Phib_implEul = impl_eul(fa,
y_Strang_explEul_Phib_implEul, h/2.)
65         y_Strang_explEul_Phib_implEul = Phib(y_Strang_explEul_Phib_implEul, h)
        y_Strang_explEul_Phib_implEul = expl_eul(fa,
y_Strang_explEul_Phib_implEul, h/2.)

67         #Lie Trotter with explicit midpoint
69         y_Lt_explmid = expl_mid(fb, y_Lt_explmid, h)
        y_Lt_explmid = expl_mid(fa, y_Lt_explmid, h)

71         #Strang with explicit midpoint
73         y_Strang_explmid = expl_mid(fa, y_Strang_explmid, h/2.)
        y_Strang_explmid = expl_mid(fb, y_Strang_explmid, h)
75         y_Strang_explmid = expl_mid(fa, y_Strang_explmid, h/2.)

77     #now calculate the errors

```

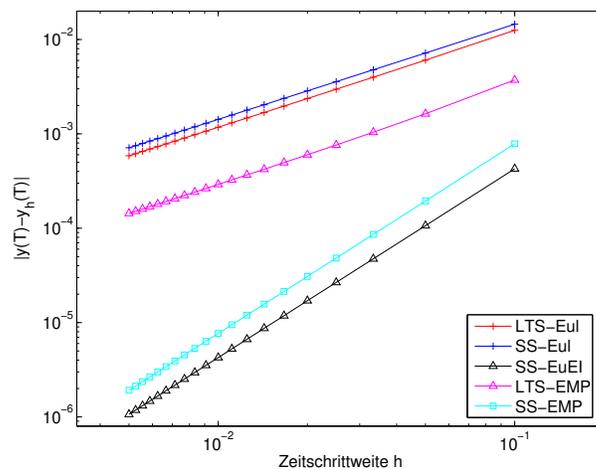
```

err_Lt_explEul += [abs(y_Lt_explEul - y_1_exact)]
err_Strang_explEul += [abs(y_Strang_explEul - y_1_exact)]
err_Strang_explEul_Phib_implEul += [abs(y_Strang_explEul_Phib_implEul -
y_1_exact)]
err_Lt_explmid += [abs(y_Lt_explmid - y_1_exact)]
err_Strang_explmid += [abs(y_Strang_explmid - y_1_exact)]

figure()
title("Convergence of Lie-Trotter and Strang Splitting with different
combinations of methods")
grid(True); xlabel("$h$"); ylabel("err")
loglog(T/Ns,(T/Ns), "-", label = "$O(h)$")
loglog(T/Ns,(T/Ns)**2, "-", label = "$O(h^2)$")
loglog(T/Ns,err_Lt_explEul,"x-", label = "Lie-Trotter, expl. Eul")
loglog(T/Ns,err_Strang_explEul,"x-", label = "Strang, Expl. Eul")
loglog(T/Ns,err_Strang_explEul_Phib_implEul,"x-", label = "Strang, Expl. Eul -
Phib - Impl Eul")
loglog(T/Ns,err_Lt_explmid,"x-", label = "Lie-Trotter, expl. mid")
loglog(T/Ns,err_Strang_explmid,"x-", label = "Strang, Expl. mid")

legend(loc="best")
show()

```



- LTS-Eul      explizites Euler als  $\Psi_{a,b}^h$ ,  $\Psi_{a,b}^h$  und Lie-Trotter-Splitting
- SS-Eul      explizites Euler als  $\Psi_{a,b}^h$ ,  $\Psi_{a,b}^h$  und Strang-Splitting
- SS-EuEI    Strang-Splitting: explizites Euler als  $\Psi_a^{h/2}$ , exaktes  $\Phi_b^h$  und implizites Euler als  $\Psi_a^{h/2}$
- LTS-EMP    explizite Mittelpunkt-Regel als  $\Psi_{a,b}^h$ ,  $\Psi_{a,b}^h$  und Lie-Trotter-Splitting
- SS-EMP    explizite Mittelpunkt-Regel als  $\Psi_{h,g}^h$ ,  $\Psi_{h,f}^h$  und Strang-Splitting

Abb. 2.4.5. Einfache Splitting-Verfahren

**Beispiel 2.4.6.** (Splitting-Verfahren für die Newtonschen Gleichungen)

Wir schreiben die Newtonschen Gleichungen als ein System Differentialgleichungen erster Ordnung um:

$$\ddot{\mathbf{r}} = a(\mathbf{r}) \iff \dot{\mathbf{y}} := \begin{bmatrix} \dot{\mathbf{r}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ a(\mathbf{r}) \end{bmatrix} =: \mathbf{F}(\mathbf{y}),$$

und wir zerlegen die rechte Seite:

$$F(\mathbf{y}) = \underbrace{\begin{bmatrix} 0 \\ a(\mathbf{r}) \end{bmatrix}}_{=: \mathbf{f}(\mathbf{y})} + \underbrace{\begin{bmatrix} \mathbf{v} \\ 0 \end{bmatrix}}_{=: \mathbf{g}(\mathbf{y})}.$$

Die exakten Evolutionsoperatoren für die zwei Teile sind:

$$\Phi_f^t \begin{bmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 \end{bmatrix} = \begin{bmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 + ta(\mathbf{r}_0) \end{bmatrix} \quad \text{und} \quad \Phi_g^t \begin{bmatrix} \mathbf{r}_0 \\ \mathbf{v}_0 \end{bmatrix} = \begin{bmatrix} \mathbf{r}_0 + t\mathbf{v}_0 \\ \mathbf{v}_0 \end{bmatrix}.$$

Das Lie-Trotter-Splitting-Verfahren (2.4.26) ergibt das symplektische Euler-Verfahren

$$\Psi^h \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix} = (\Phi_g^h \circ \Phi_f^h) \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \mathbf{r} + h(\mathbf{v} + ha(\mathbf{r})) \\ \mathbf{v} + ha(\mathbf{r}) \end{bmatrix}, \quad (2.4.29)$$

und das Strang-Splitting-Verfahren (2.4.27):

$$\Psi^h \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix} = (\Phi_g^{h/2} \circ \Phi_f^h \circ \Phi_g^{h/2}) \begin{bmatrix} \mathbf{r} \\ \mathbf{v} \end{bmatrix} = \begin{bmatrix} \mathbf{r} + h\mathbf{v} + \frac{1}{2}h^2a(\mathbf{r} + \frac{1}{2}h\mathbf{v}) \\ \mathbf{v} + ha(\mathbf{r} + \frac{1}{2}h\mathbf{v}) \end{bmatrix} \quad (2.4.30)$$

ergibt die Ein-Schritt-Formulierung des Störmer-Verlet-Verfahrens:

$$\begin{aligned} \mathbf{r}_{k+\frac{1}{2}} &= \mathbf{r}_k + \frac{1}{2}h\mathbf{v}_k, \\ (2.4.30) \quad \iff \quad \mathbf{v}_{k+1} &= \mathbf{v}_k + ha(\mathbf{r}_{k+\frac{1}{2}}), \\ \mathbf{r}_{k+1} &= \mathbf{r}_{k+\frac{1}{2}} + \frac{1}{2}h\mathbf{v}_{k+1}. \end{aligned} \quad (2.4.31)$$

**Bemerkung 2.4.7.** Diese Idee kann man sofort auf separable Hamilton-Systeme, d.h. mit der Hamilton-Funktion  $H(\mathbf{q}, \mathbf{p}) = T(\mathbf{p}) + V(\mathbf{q})$  anwenden; das Lie-Trotter-Splitting-Verfahren liefert das symplektische Euler-Verfahren

$$\begin{aligned} \mathbf{p}_1 &= \mathbf{p}_0 - hDV(\mathbf{q}_0) \\ \mathbf{q}_1 &= \mathbf{q}_0 + hDT(\mathbf{p}_1), \end{aligned}$$

während das Strang-Splitting-Verfahren zum Störmer-Verlet-Verfahren führt.

**Bemerkung 2.4.8.** Es können spezielle  $a_i$  und  $b_i$  gewählt werden, die symplektische Methoden höherer Ordnungen ergeben, siehe [S. BLANES, P.C. MOAN](#) *Journal of Computational and Applied Mathematics* 142 (2002),

## Code 2.4.9: Einige Splitting Routinen

```

# -*- coding: utf-8 -*-
2 from numpy import zeros, flipud, double, array
4
5 __all__ = ["SplittingParameters", "PrepSplittingParameters"]
6
7 class SplittingParameters(object):
8     def build(self, method):
9         """
10        :param method: A string specifying the method for time integration.
11        :return: Two arrays :math:`a` and :math:`b`.
12
13        =====
14        Method Order Authors Reference
15        =====
16        LT 1 Lie/Trotter [1]_, [3]_ page 42, equation 5.2
17        S2 2 Strang [2]_, [3]_ page 42, equation 5.3
18        SS 2 Strang [2]_, [3]_ page 42, equation 5.3
19        PRKS6 4 Blanes/Moan [4]_ page 318, table 2, 'S6'
20        BM42 4 Blanes/Moan [4]_ page 318, table 3, 'SRKNb6'
21        Y4 4 Yoshida [5]_, [3]_ page 40, equation 4.4
22        Y61 6 Yoshida [5]_, [3]_ page 144, equation 3.11
23        BM63 6 Blanes/Moan [4]_ page 318, table 3, 'SRKNa14'
24        KL6 6 Kahan/Li [6]_, [3]_ page 144, equation 3.12
25        KL8 8 Kahan/Li [6]_, [3]_ page 145, equation 3.14
26        L42 (4,2) McLachlan [7]_ page 6
27        L84 (8,4) McLachlan [7]_ page 8
28        =====
29
30        .. [1] H.F. Trotter, "On the product of semi-groups of operators",
31            Proc. Am. Math. Soc.10 (1959) 545-551.
32
33        .. [2] G. Strang, "On the construction and comparison of difference schemes",
34            SIAM J. Numer. Anal. 5 (1968) 506-517.
35
36        .. [3] E. Hairer, C. Lubich, and G. Wanner, "Geometric Numerical Integration -
37            Structure-Preserving Algorithms for Ordinary Differential Equations",
38            Springer-Verlag, New York, 2002 (Corrected second printing 2004).
39
40        .. [4] S. Blanes and P.C. Moan, "Practical Symplectic Partitioned
41            Runge-Kutta and Runge-Kutta-Nystrom Methods", J. Computational and
42            Applied Mathematics, Volume 142, Issue 2, (2002) 313-330.
43
44        .. [5] H. Yoshida, "Construction of higher order symplectic integrators",
45            Phys. Lett. A 150 (1990) 262-268.
46
47        .. [6] W. Kahan and R.-c. Li, "Composition constants for raising the orders
48            of unconventional schemes for ordinary differential equations",
49            Math. Comput. 66 (1997) 1089-1099.
50
51        .. [7] R.I. McLachlan, "Composition methods in the presence of small parameters",
52            BIT Numerical Mathematics, Volume 35, Issue 2, (1995) 258-268.
53        """
54        if method == "LT":
55            self.order = 1
56            s = 1
57            a = zeros(s)
58            b = zeros(s)
59            a[0] = 1.0
60            b[0] = 1.0
61        elif method == "S2":
62            self.order = 2
63            s = 2
64            a = zeros(s)
65            b = zeros(s)
66            a[1] = 1.0
67            b[0] = 0.5
68            b[1] = 0.5
69        elif method == "SS":
70            self.order = 2
71            s = 2
72            a = zeros(s)
73            b = zeros(s)
74            a[0] = 0.5
75            a[1] = 0.5
76            b[0] = 1.0
77        elif method == "L42":
78            self.order = 2
79            # Pattern ABA and m = s = 2
80            s = 3
81            a = zeros(s)
82            b = zeros(s)
83            a[1] = 0.5773502691896258
84            a[0] = 0.5*(1.-a[1])
85            a[2] = a[0]
86            b[0] = 0.5
87            b[1] = 0.5
88        elif method == "BM42":
89            self.order = 4

```

```

90     s = 7
91     a = zeros(s)
92     b = zeros(s)
93     a[1] = 0.245298957184271
94     a[2] = 0.604872665711080
95     a[3] = 0.5 - a[:3].sum()
96     a[4:] = flipud(a[1:4]) #Flip array in the up/down direction
97     b[0] = 0.0829844064174052
98     b[1] = 0.396309801498368
99     b[2] = -0.0390563049223486
100    b[3] = 1.0 - 2.0*b[:3].sum()
101    b[4:] = flipud(b[:3])
102    elif method == "Y4":
103        self.order = 4
104        s = 4
105        a = zeros(s)
106        b = zeros(s)
107        pp = 3.0
108        theta = 1.0/(2.0-2**(1.0/pp))
109        vi = -2**(1.0/pp)*theta
110        a[0] = 0.0
111        a[1] = theta
112        a[2] = vi
113        a[3] = theta
114        b[0] = 0.5*theta
115        b[1] = 0.5*(vi+theta)
116        b[2:] = flipud(b[:2])
117    elif method == "Y61":
118        self.order = 6
119        s = 8
120        a = zeros(s)
121        b = zeros(s)
122        a[1] = 0.78451361047755726381949763
123        a[2] = 0.23557321335935813368479318
124        a[3] = -1.17767998417887100694641568
125        a[4] = 1.0 - 2.0*a[1:4].sum()
126        a[5:] = flipud(a[1:4])
127        b[0] = 0.5*a[1]
128        b[1] = 0.5*a[1:3].sum()
129        b[2] = 0.5*a[2:4].sum()
130        b[3] = 0.5*(1-4*b[1]-a[3])
131        b[4:] = flipud(b[0:4])
132    elif method == "BM63":
133        self.order = 6
134        s = 15
135        a = zeros(s)
136        b = zeros(s)
137        a[1] = 0.09171915262446165
138        a[2] = 0.183983170005006
139        a[3] = -0.05653436583288827
140        a[4] = 0.004914688774712854
141        a[5] = 0.143761127168358
142        a[6] = 0.328567693746804
143        a[7] = 0.5 - a[:7].sum()
144        a[8:] = flipud(a[1:8])
145        b[0] = 0.0378593198406116
146        b[1] = 0.102635633102435
147        b[2] = -0.0258678882665587
148        b[3] = 0.314241403071447
149        b[4] = -0.130144459517415
150        b[5] = 0.106417700369543
151        b[6] = -0.00879424312851058
152        b[7] = 1.0 - 2.0*b[:7].sum()
153        b[8:] = flipud(b[:7])
154    elif method == "PRKS6":
155        self.order = 4
156        s = 7
157        a = zeros(s)
158        b = zeros(s)
159        a[1] = 0.209515106613362
160        a[2] = -0.143851773179818
161        a[3] = 0.5 - a[:3].sum()
162        a[4:] = flipud(a[1:4])
163        b[0] = 0.0792036964311957
164        b[1] = 0.353172906049774
165        b[2] = -0.0420650803577195
166        b[3] = 1 - 2*b[:3].sum()
167        b[4:] = flipud(b[:3])
168    elif method == "KL6":
169        self.order = 6
170        s = 10
171        a = zeros(s)
172        b = zeros(s)
173        a[1] = 0.39216144400731413927925056
174        a[2] = 0.33259913678935943859974864
175        a[3] = -0.70624617255763935980996482
176        a[4] = 0.08221359629355080023149045
177        a[5] = 1.0 - 2.0*a[1:5].sum()
178        a[6:] = flipud(a[1:5])
179        b[0] = 0.5*a[1]
180        b[1] = 0.5*a[1:3].sum()
181        b[2] = 0.5*a[2:4].sum()

```

```

182     b[3] = 0.5*a[3:5].sum()
183     b[4] = 0.5*(1-2*a[1:4]).sum()-a[4]
184     b[5:] = flipud(b[0:5])
185 elif method == "KL8":
186     self.order = 8
187     s = 18
188     a = zeros(s)
189     b = zeros(s)
190     a[0] = 0.0
191     a[1] = 0.13020248308889008087881763
192     a[2] = 0.56116298177510838456196441
193     a[3] = -0.38947496264484728640807860
194     a[4] = 0.15884190655515560089621075
195     a[5] = -0.39590389413323757733623154
196     a[6] = 0.18453964097831570709183254
197     a[7] = 0.25837438768632204729397911
198     a[8] = 0.29501172360931029887096624
199     a[9] = -0.60550853383003451169892108
200     a[10:] = flipud(a[1:9])
201     b[0:-1] = 0.5*(a[:-1]+a[1:])
202     b[-1] = 1.*b[0]
203 elif method == "L84":
204     self.order = 4
205     # Pattern ABA
206     s = 6
207     a = zeros(s)
208     b = zeros(s)
209     a[0] = 0.07534696026989288842
210     a[1] = 0.51791685468825678230
211     a[2] = -0.09326381495814967072
212     b[0] = 0.19022593937367661925
213     b[1] = 0.84652407044352625706
214     b[2] = -1.07350001963440575260
215     a[3:] = flipud(a[0:3])
216     b[3:-1] = flipud(b[0:2])
217 else:
218     raise NotImplementedError("Unknown method: " + method)
219
220 return a, b
221
222 def intsplitt(self, psi1, psi2, a, b, tspan, N, y0, getall=False, args1=(), args2=()):
223     r"""
224     Compute a single, full propagation step by operator splitting.
225
226     :param psi1: First evolution operator :math:`\Psi_a`
227     :param psi2: Second evolution operator :math:`\Psi_b`
228     :param a: Parameters for evolution with :math:`\Psi_a`
229     :param b: Parameters for evolution with :math:`\Psi_b`
230     :param tspan: Timespan :math:`t` of a single, full splitting step
231     :param N: Number of substeps to perform
232     :param y0: Current value of the solution :math:`y(t)`
233     :param getall: Return the full (internal) time evolution of the solution
234     :param args1: Additional optional arguments of :math:`\Psi_a`
235     :param args2: Additional optional arguments of :math:`\Psi_b`
236     """
237     if not type(args1) is tuple: args1 = (args1,)
238     if not type(args2) is tuple: args2 = (args2,)
239     s = a.shape[0]
240     h = (tspan[1]-tspan[0])/(1.0*N)
241     y = 1.*y0
242     if getall:
243         uu = [y0]
244         t = [tspan[0]]
245     for k in range(N): #in this for-range we perform the actual computation
246         for j in range(s):
247             y = psi1(a[j]*h, 1.*y, *args1) #apply the evolution operator corresponding to fa
248             y = psi2(b[j]*h, 1.*y, *args2) #apply the evolution operator corresponding to fb
249             if getall:
250                 uu += [y]
251                 t += [t[-1]+h]
252     if getall: return array(t), array(uu)
253     else: return y #normally we are just interested in the final value
254
255 class PrepSplittingParameters(object): #for processing methods, see Bemerkung 8.4.15
256
257     def build(self, method):
258         if method == "BCR764": # Blanes, Casas, Ros 2000, table IV
259             # kernel ABA
260             # exchanged a and b from paper in order to have consistency with previous code
261             a = zeros(4) # for Beps
262             a[1] = 1.5171479707207228
263             a[2] = 1.-2.*a[1]
264             a[3] = 1.* a[1]
265             #
266             b = zeros(4) # for Abig
267             b[0] = 0.5600879810924619
268             b[1] = 0.5-b[0]
269             b[2] = 1.*b[1]
270             b[3] = 1.*b[0]
271             # processor
272             z = zeros(6) # for Abig
273             z[0] = -0.3346222298730

```

```

274     z[1] = 1.097567990732164
275     z[2] = -1.038088746096783
276     z[3] = 0.6234776317921379
277     z[4] = -1.102753206303191
278     z[5] = -0.0141183222088869
279     #
280     y = zeros(6) # for Beps
281     y[0] = -1.621810118086801
282     y[1] = 0.0061709468110142
283     y[2] = 0.8348493592472594
284     y[3] = -0.0511253369989315
285     y[4] = 0.5633782670698199
286     y[5] = -0.5
287
288     else:
289         raise NotImplementedError("Unknown method: " + method)
290
291     forBeps = a
292     forAbig = b
293     y4Beps = y
294     z4Abig = z
295     #return a, b, y, z
296     return forBeps, forAbig, y4Beps, z4Abig
297
298 def intprepsplit(self, Beps, Abig, forBeps, forAbig, y, z, tspan, N, u0, getall=False):#, args1=(), args2=()):
299     r"""
300     """
301     #if type(args1) != type(): args1 = (args1,)
302     #if type(args2) != type(): args2 = (args2,)
303
304     s = forBeps.shape[0]
305     p = y.shape[0]
306     h = (tspan[1]-tspan[0])/(1.0*N)
307     u = 1.*u0
308     if getall:
309         uu = [u0]
310         t = [tspan[0]]
311
312     #print h
313
314     for j in range(p): # preprocessor
315         u = Abig(-z[j]*h, 1.*u)#, *args1)
316         u = Beps(-y[j]*h, 1.*u)#, *args2)
317
318     #if getall:
319     # uu += [u]
320     # t+= [t[-1]+h]
321
322     for k in range(N): # kernel
323         for j in range(s):
324             u = Beps(forBeps[j]*h, 1.*u)#, *args1)
325             u = Abig(forAbig[j]*h, 1.*u)#, *args2)
326         if getall:
327             v = 1.* u
328             for j in range(p-1,-1,-1): # postprocessor
329                 v = Abig(y[j]*h, 1.*v)#, *args1)
330                 v = Beps(z[j]*h, 1.*v)#, *args2)
331
332             uu += [v]
333             t+= [t[-1]+h]
334
335     for j in range(p-1,-1,-1): # postprocessor
336         u = Beps(y[j]*h, 1.*u)#, *args1)
337         u = Abig(z[j]*h, 1.*u)#, *args2)
338
339     #if getall:
340     # uu += [u]
341     # t+= [t[-1]+h]
342
343     if getall: return array(t), array(uu)
344     else: return u

```

**Beispiel 2.4.10.** Wir können nun experimentell die Energieerhaltung und die Konvergenzordnung beim Einsatz verschiedener Methoden für den mathematischen Pendel anschauen.

### Code 2.4.11: Energieerhaltung beim Splitting Verfahren

```

1 from numpy import array, cos, sin, pi, shape, vstack, min, max, double, zeros
2 import matplotlib.pyplot as plt
3 from ode45 import ode45
4 import scipy.optimize
5 import numpy as np
6 from time import time
7
8 # savethem = True

```

```

10 savethem = False
11 pict_kind = '.pdf'
12 plt.rcParams["text.usetex"] = r"True"
13 plt.rcParams["text.latex.preamble"] = r"\usepackage{color}"

14 def PendulumODE(y,l,g):
15     """PendulumODE return the right-hand side of the math. pendulum"""
16     dydt = array([ y[1], -g*sin(y[0])/l])
17     return dydt

18 def IntegratePendulum(phi0,tEnd=1.8,l=0.6,g=9.81,flag=False):
19     """IntegratePendulum solve the mathematical pendulum with ode45
20     flag == false --> y = complete solution phi,phi'
21     flag == true --> y = phi(tEnd)
22     (order of the 2 outputs reversed wrt the usual to use fzero)
23     """
24     if shape(phi0) == (1,): phi0 = phi0[0]
25     y0 = array([phi0, 0])
26     tspan = (0, tEnd)
27     t,y = ode45(lambda t,y: PendulumODE(y,l,g), tspan, y0)
28     if flag:
29         return y[-1][0]
30     else:
31         return (y,t)

32 def IntegratePendulumIM(phi0,N,tEnd=1.8,l=0.6,g=9.81,flag=False):
33     """IntegratePendulum solve the mathematical pendulum with Implicit Midpoint
34     flag == false --> y = complete solution phi,phi'
35     flag == true --> y = phi(tEnd)
36     """
37     if shape(phi0) == (1,): phi0 = phi0[0]
38     y0 = array([phi0, 0])
39
40     h = double(tEnd)/N #obtain h by dividing the total time difference by the number of steps
41     y = zeros((N+1,2))
42     y[0,:] = y0 #starting value
43     t = zeros(N+1)
44     t[0] = 0
45
46     for k in range(int(N)):
47         tmp = y[0]
48         F = lambda x: x+0.5*h*g/l*sin((x+y0[0])/2.0) - y0[0] - h*y0[1] #implicit equation to solve
49         y0[0] = scipy.optimize.fsolve(F,y0[0]+h*y0[1])
50         y0[1] = y0[1] - h*g/l*sin((tmp+y0[0])/2.0)
51         y[k+1,:]=y0
52         t[k+1] = (k+1)*h
53
54     if flag:
55         return y[-1][0]
56     else:
57         return (y,t)

58 def Energy(y,g,l):
59     #computes the total energy of the pendulum for a given configuration
60     E_kin = 0.5*(y[:,1]**2); #with m=1 Ekin=v^2/2
61     E_pot = -g/l*(cos(y[:,0]));
62     E_tot = E_kin + E_pot;
63     return E_tot

64 # ----- splitting parts -----
65 #we include the time by using the vector [q,p,t]
66 #as a first step, we define the two integrators

67 def phiA(dt, u,g,l):
68     t = u[2] + dt
69     q = 1.*u[0]
70     p = u[1] - dt*sin(q)*g/l
71     return array([q,p,t])

72 def phiB(dt, u):
73     t = 1.*u[2]
74     p = 1.*u[1]
75     q = u[0] + dt*p
76     return array([q,p,t])

77 # ----- physical constants -----
78 l = 0.4
79 g = 9.81
80 tEnd = 0.25*1500. # 2*1.8
81 phiT = 2.8*pi/6
82 #phiT = 5.*pi/6

83 print('ODE45 integrate ...')
84 # ODE45
85 starttime = time()
86 yODE45, tODE45 = IntegratePendulum(phiT, tEnd, l, g, False)
87 t1 = time() - starttime #measure time needed for ODE45 to run
88 print('ODE45 needs', tODE45.shape, 'timesteps in', t1, 'seconds')
89 N = tODE45.shape[0]
90 dt = double(tEnd)/N
91 # Implicit Midpoint
92 print('IMP integrate ...')

```

```

starttime = time()
102 y_IM,t_IM = IntegratePendulumIM(phiT,N,tEnd,l,g, False)
t2 = time() - starttime #measure time needed for implicit midpoint to run
104 print('IM with', N, ' equal timesteps in', t2, 'seconds')

# -----
106 from SplittingParameters import *
108 S = SplittingParameters() #use parameters given in the previous code
y0 = phiT
110 y0p = 0.
tspan = [0.,tEnd]
112 u0 = array([y0,y0p,0.])
ys = 1.*u0
114 Methods = ['LT','S2', 'PRKS6' , 'Y4', 'KL6'] #list of different methods that we are going to use
#Methods = ['PRK-S6', 'Y4']
116 plt.figure()
for method in Methods:
118     a,b = S.build(method) #obtain parameters from the values stored in the class
    print(method + ' integrate ...')
120     starttime = time()
    tt, uu = S.intsplit(phiA,phiB,a,b,tspan,N,ys, getall=True,args1=(g,l))
122     #perform the (whole) integration using the function already defined in the class in the previous code
    t2 = time() - starttime #measure the time needed for the integration to run
124     print(method+ ' with', N, ' equal timesteps in', t2, 'seconds')
    E_tot = Energy(uu,g,l) #calculate the energy at every time
126     #plot the variations of the energy
    plt.semilogy(uu[:,2],abs(E_tot-E_tot[0]),label=method)

128 # Implicit Midpoint
y = y_IM;
130 E_tot = Energy(y,g,l)
print('plot energy evolution in IM')
132 plt.semilogy(t_IM,abs(E_tot-E_tot[0]),label='IMP')#,marker=styles.next()
# ode45
y = yODE45;
136 E_tot = Energy(y,g,l) #calculate the energy of the solution found with ODE45 and plot it
plt.semilogy(tODE45,abs(E_tot-E_tot[0]),label='ode45')#,marker=styles.next()
138 plt.ylabel('Energieabweichung')
plt.xlabel('t')
140 plt.legend(loc=4)

142 filename = 'PendSplitEnergies'+str(N)+pict_kind
if savethem: plt.savefig(filename)
144 else: plt.show()

```

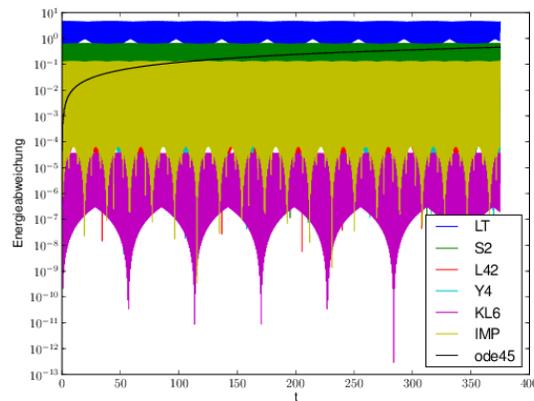


Abb. 2.4.12. Abweichung der Energie bei einige Integratoren

## Code 2.4.13: Fehler der Splitting Verfahren

```

2  from numpy import array, cos, sin, pi, shape, vstack, min, max, double, zeros
import matplotlib.pyplot as plt
4  from ode45 import ode45
import scipy.optimize
import numpy as np
6  from time import time

8  #savethem = True
savethem = False
10 pict_kind = '.pdf'

12 import itertools
gfx_params = {
14     'axes.labelsize': 30,
     'text.fontsize': 30,
16     'legend.fontsize': 20,
     'xtick.labelsize': 20,
18     'ytick.labelsize': 20,
     'axes.titlesize': 30,
20     'lines.markeredgewidth' : 2,
     'lines.markersize' : 8,
22     'lines.linewidth':2.0
}

24
linestyles = ['--', ':', '-.', '_']
26 linecolors = ['r', 'g', 'b', 'c', 'm', 'k']
markers = ['<', '>', 'D', 's', '^', 'd', 'v', 'x', 'o', '*', '+']
28 styles = itertools.cycle(markers)
colors = itertools.cycle(linecolors)
30 plt.rcParams["text.usetex"] = r"True"
plt.rcParams["text.latex.preamble"] = r"\usepackage{color}"

32
def PendulumODE(y,l,g):
34     """PendulumODE return the right-hand side of the math. pendulum"""
     dydt = array([ y[1], -g*sin(y[0])/l])
36     return dydt

38
def IntegratePendulum(phi0,tEnd=1.8,l=0.6,g=9.81,flag=False):
40     """IntegratePendulum solve the mathematical pendulum with ode45
     flag == false --> y = complete solution phi,phi'
     flag == true --> y = phi(tEnd)
42     (order of the 2 outputs reversed wrt the usual to use fzero)
     """
44     if shape(phi0) == (1,): phi0 = phi0[0]
     y0 = array([phi0, 0])
46     tspan = (0, tEnd)
     t,y = ode45(lambda t,y: PendulumODE(y,l,g), tspan, y0)
48     if flag:
         return y[-1][0]
50     else:
         return (y,t)

52
def Energy(y,g,l): #total energy of the system
54     E_kin = 0.5*(y[:,1]**2); #m=1, so Ekin=v^2/2
     E_pot = -g/l*(cos(y[:,0]));
56     E_tot = E_kin + E_pot;
     return E_tot

58
# ----- splitting parts -----
60 #first we define the two integrators
def phiA(dt, u,g,l):
62     t = u[2] + dt
     q = 1.*u[0]
64     p = u[1] - dt*sin(q)*g/l
     return array([q,p,t])

66
def phiB(dt, u):
68     t = 1.*u[2]
     p = 1.*u[1]
70     q = u[0] + dt*p
     return array([q,p,t])

72 # ----- physical constants -----
l = 0.4
74 g = 9.81
phiT = 2.8*pi/6 # initial angle
76 #phiT = 5.*pi/6
tEnd = 0.25*1500. # 2*1.8
N = 2**15 # number of time-steps at reference
80 dt = double(tEnd)/N # time-step
print('ODE45 integrate ...')
# ODE45
82 starttime = time()
yODE45 = IntegratePendulum(phiT, tEnd, l, g, True)
84 t1 = time() - starttime
print('ODE45 needs', t1, 'seconds')

86
# -----
88 from SplittingParameters import *
S = SplittingParameters()

```

```

90
91 #we first look for a reference solution, which we obtain by using a high-order (but expensive) method
92 print("Compute an accurate solution by a method of 8th order, say KL8")
93 method = 'KL8'
94 y0 = phiT
95 y0p = 0.
96 tspan = [0.,tEnd]
97 u0 = array([y0,y0p,0.])
98 a,b = S.build(method)
99 ys = 1.*u0
100
101 # integrate with a method of order 8, expensive
102
103 print(method + ' integrate ...')
104 starttime = time()
105 uuKL8 = S.intsplit(phiA,phiB,a,b,tspan,N,ys, getall=False,args1=(g,1))
106 t2 = time() - starttime
107 print(method+ ' with', N, ' equal timesteps in', t2, 'seconds')
108 # see how wrong ode45 is
109 print('error of ode45 at time', tEnd, ' is ', abs(yODE45 - uuKL8[0]))
110
111 # go for all
112 #now we perform the integration by means of the other methods, too.
113 #This way we can take a look at the accuracy of each method.
114 Methods = ['S2', 'L42', 'BM42', 'Y4', 'Y61', 'BM63', 'KL6', 'KL8']
115 Ns = 2**np.arange(8,16)
116 plt.figure()
117 for method in Methods:
118     print('-----', method, ' -----')
119     a,b = S.build(method)
120     err = []
121     for N in Ns:
122         ys = 1.*u0
123         #perform the actual computation with the function written in the class contained in the previous code
124         uu = S.intsplit(phiA,phiB,a,b,tspan,N,ys,args1=(g,1))
125         #estimate the absolute value of the error using the reference solution.
126         #getall=False->we are only interested in the error at the final time
127         err0 = abs(uu[0]-uuKL8[0]) #the error
128         err += [err0]
129     err = array(err)
130     #finally, we produce a logarithmic plot the the error as a function of the time interval length h
131     plt.loglog((tspan[1] - tspan[0])/Ns,err,label=method,marker=next(styles))
132
133 z = (tspan[1] - tspan[0])/Ns
134 #now we plot some possible convergence relations in order to be able to estimate the ones of the methods
135 #that we are studying just by taking a look at the graphs
136 plt.loglog(z,z**2,'--',label='$h^{-2}$')
137 plt.loglog(z,z**4,'-',label='$h^{-4}$')
138 plt.loglog(z,z**8,':',label='$h^{-8}$')
139
140 #plt.title('convergence of splitting schemes')
141 plt.xlabel('Zeitschrittweite h')
142 plt.ylabel(r'$|y[T]-y_h[T]|$')
143 plt.legend(loc='best')
144
145 filename = 'PendSplitErr'+str(N)+pict_kind
146 if savethem: plt.savefig(filename)
147 else: plt.show()

```

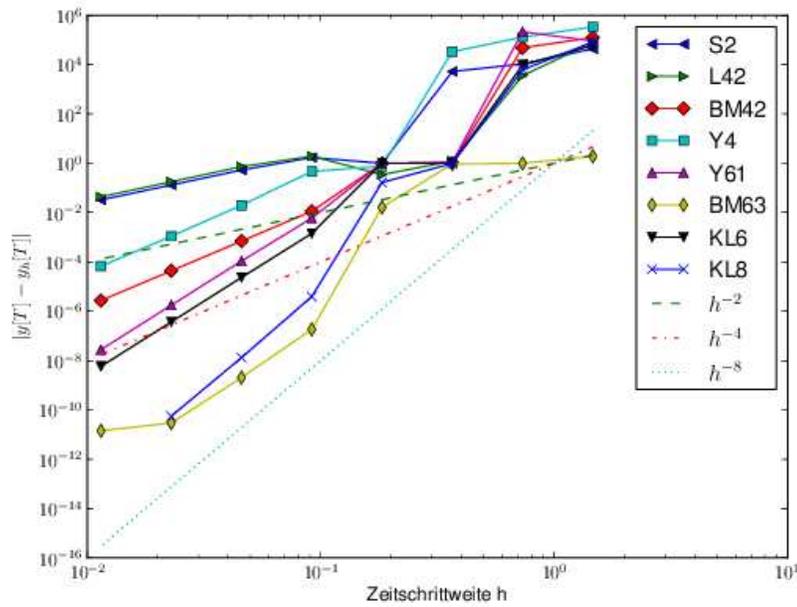


Abb. 2.4.14. Fehler einiger Integratoren

**Bemerkung 2.4.15.** Processing-Methoden sind Splitting-Verfahren der Form

$$\hat{\Psi}^h = \Pi^h \circ \Psi^h \circ (\Pi^h)^{-1}, \quad (2.4.32)$$

mit Postprocessor  $\Pi^h$  und Kernel  $\Psi^h$ , was nach  $n$  Zeit-Schritten

$$\left(\hat{\Psi}^h\right)^n = \Pi^h \circ (\Psi^h)^n \circ (\Pi^h)^{-1}$$

ergibt. Diese Idee ist vorteilhaft, wenn  $\hat{\Psi}^h$  genauer als  $\Psi^h$  ist,  $\Pi^h$  günstig ist, und wir nicht so viele Ausgaben vor Endzeit brauchen.

**Beispiel 2.4.16.** Das Störmer-Verlet-Verfahren ist eine Processing-Methode mit dem symplektischen Euler als Kernel:

$$\Psi_2^h = \Phi_a^{h/2} \circ \Phi_b^h \circ \Phi_a^{h/2} = \Phi_a^{h/2} \circ \Phi_b^h \circ \Phi_a^h \circ \Phi_a^{-h/2} = \Phi_a^{h/2} \circ \Psi_1^h \circ \Phi_a^{-h/2}.$$

mit  $\circ \Psi_1^h = \circ \Phi_b^h \circ \Phi_a^h$ .

### 2.4.1 Splitting-Verfahren für gestörte Systeme

Oft liegen die gewöhnlichen Differentialgleichungen, die bei der Lösung naturwissenschaftlicher Probleme auftreten, sehr nah an exakt lösbaren Problemen: wir reden hier von gestörten Problemen der Art

$$\dot{\mathbf{y}} = \mathbf{f}_a(\mathbf{y}) + \varepsilon \mathbf{f}_b(\mathbf{y}),$$

wobei die Lösung von  $\dot{\mathbf{y}} = \mathbf{f}_a(\mathbf{y})$  entweder analytisch oder durch Anwendung einfacher numerischer Verfahren zur Verfügung steht, während  $\varepsilon$  ein kleiner fester Parameter ist. Zum Beispiel wird die Bewegung eines Satelliten im Gravitationsfeld eines abgeplatteten (Engl.: oblate) Ellipsoids durch das gestörte Kepler-Problem modelliert, das ein Hamiltonsches System mit der Hamilton-Funktion

$$H(\mathbf{q}, \mathbf{p}) = \frac{1}{2}(p_1^2 + p_2^2) - \frac{1}{\|\mathbf{q}\|} - \frac{\varepsilon}{2\|\mathbf{q}\|^3} \left(1 - \frac{3q_1^2}{\|\mathbf{q}\|^2}\right). \quad (2.4.33)$$

ist. Ein einfacheres Beispiel, wo die Teilprobleme exakt lösbar sind, ist der gestörte Harmonische Oszillator

$$H(\mathbf{q}, \mathbf{p}) = \frac{1}{2}(\|\mathbf{p}\|^2 + \|\mathbf{q}\|^2) + \frac{\varepsilon}{2}\|\mathbf{q}\|^2. \quad (2.4.34)$$

Für solche gestörte Systeme kann man das Splitting-Verfahren optimieren, so dass der Fehler von der Grösse

$$\mathcal{O}(\varepsilon h^{r_1} + \varepsilon^2 h^{r_2} + \dots + \varepsilon^m h^{r_m}), \text{ mit } r_i \geq r_{i+1}$$

ist, was auch unter  $(r_1, r_2, \dots, r_m)$  im Namen von Verfahren ersichtlich gemacht wird. Details befinden sich unter [R. McLACHLAN BIT 35\(2\) \(1995\)](#); die Methoden L42 und L84 im Code 2.4.9 sind (4, 2) und (8, 4), während BCR764 eine optimierte Processing-Methode ist, die (7, 6, 4) ist, und die auch mit grossen Zeitschrittweiten verwendbar ist.

#### Code 2.4.17: Fehler beim Splitting Verfahren

```

1  import numpy as np
2  import itertools
3  gfx_params = {
4      'axes.labelsize': 30,
5      'text.fontsize': 30,
6      'legend.fontsize': 20,
7      'xtick.labelsize': 20,
8      'ytick.labelsize': 20,
9      'axes.titlesize': 30,
10     'lines.markeredgewidth' : 2,
11     'lines.markersize' : 8,
12     'lines.linewidth':2.0
13 }
14
15 #parameters for distinguishing the plots. Not relevant for the actual numerical
16 #part.
17 linestyle = ['--', ':', '-.', '_']
18 linecolors = ['r', 'g', 'b', 'c', 'm', 'k']
19 markers = ['<', '>', 'D', 's', '^', 'd', 'v', 'x', 'o', '*', '+']
20
21 styles = itertools.cycle(markers)
22 colors = itertools.cycle(linecolors)
23
24 # General parameters

```

```

Tend = 62*np.pi
25 tspan = [0.,Tend]
eps = 0.05
27 y0 = 1.; y0p = 1.

29 from SplittingParameters import *
S = SplittingParameters()
31 P = PrepSplittingParameters()

33 #first we define the integrators
#note that the time is directly insterted in the vector we are working with
35 #as usual we have two integrators, phi1 and phi2
#note the presence of eps in phi2, indicating that phi2 is the perturbation
37
39 def psi1(dt, u):
    t = u[2] + dt
    st = np.sin(dt)
    41 ct = np.cos(dt)
    q = u[0]*ct + u[1]*st
    43 p = -u[0]*st + u[1]*ct
    return np.array([q,p,t])
45
47 def psi2(dt,u):
    q = 1.*u[0]
    p = 1.*u[1]
    49 #here and in the following we are multiplying with 1. so as to make sure
    that we are not
    #dealing with integers
    51 return np.array([q, -dt*eps*q+p,u[2]])

53 #first we want to use a high-order method as reference
print("Compute an accurate solution by a method of 8th order, say KL8")
55 method = 'KL8'
N = 5*(10**4) #number of discrete steps
57 u0 = np.array([y0,y0p,tspan[0]]) #initial values
a,b = S.build(method) #extract parameters
59 ys = 1.*u0
uuKL8 = S.intsplit(psi1,psi2,a,b,tspan,N,ys)
61
63 Methods = ['SS', 'L42', 'BM42', 'L84']
PrepMethods = ['BCR764']

65 Ns = 2**np.arange(4,16)

67 import matplotlib.pyplot as plt
fig = plt.figure()
69 ax = plt.subplot(111)

71 #now we perform the actual computations with the function defined in the codes
above
#(i.e. in the code containing the class with all the parameters for the
different methods)
73
75 for method in Methods:
    print('----- ', method, ' -----')
```

```

a,b = S.build(method)
err = []
77 for N in Ns:
81     ys = 1.*u0
83     uu = S.intsplit(psi1,psi2,a,b,tspan,N,ys)
85     err0 = abs(uu[0]-uuKL8[0]) #the error
87     err += [err0]
89     err = np.array(err)
91     plt.loglog((tspan[1] - tspan[0])/Ns,err,label=method,marker=next(styles))

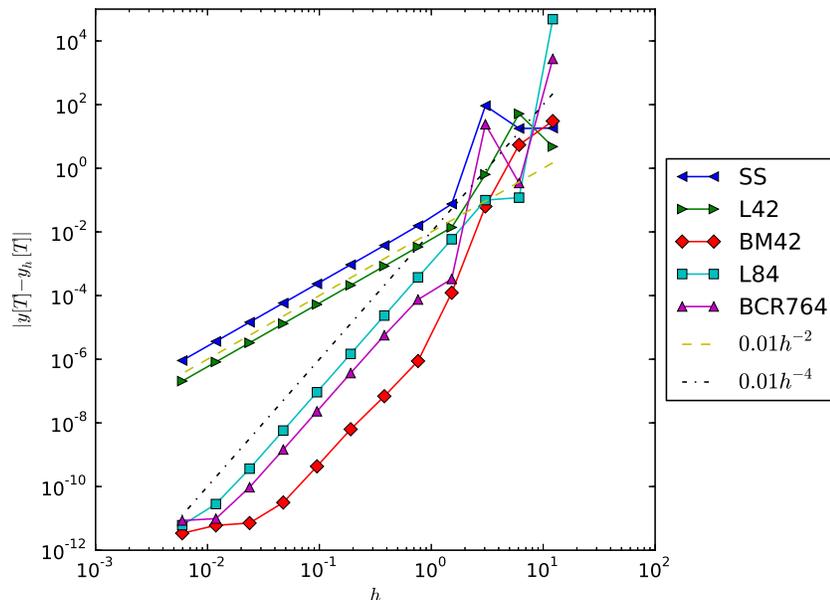
for method in PrepMethods:
93     print('----- ', method, ' -----')
95     aa,bb,y,z = P.build(method)
97     err = []
99     for N in Ns:
101         ys = 1.*u0
103         uu = P.intprepsplit(psi2,psi1,aa,bb,y,z,tspan,N,ys)
105         #we "switch" psi1 and psi2 just because of the way intprepsplit is
107         written
109         err0 = abs(uu[0]-uuKL8[0]) #the error, as above
111         err += [err0]
113         err = np.array(err)
115         plt.loglog((tspan[1] - tspan[0])/Ns,err,label=method,marker=next(styles))

zz = (tspan[1] - tspan[0])/Ns
117 plt.loglog(zz,0.01*zz**2,'--',label='$0.01 h^{-2}$')
119 plt.loglog(zz,0.01*zz**4,'-.',label='$0.01 h^{-4}$')

121 plt.xlabel('$h$')
123 plt.ylabel(r'$|y[T]-y_h[T]|$')

125 plt.legend(loc='best')
127 plt.savefig('splitBenchPerHarm.eps')
129 plt.savefig('splitBenchPerHarm.pdf')
131 plt.show()

```



**Abb. 2.4.18.** Fehler einiger Integratoren für den gestörten harmonischen Oszillator bei  $\varepsilon = 0.05$

## 2.5 Runge-Kutta-Verfahren

Wir betrachten das Anfangswertproblem  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ ,  $\mathbf{y}(t_0) = \mathbf{y}_0$ . Durch Integration von  $t_0$  bis  $t_1$  erhalten wir

$$\mathbf{y}(t_1) = \mathbf{y}_0 + \int_{t_0}^{t_1} \mathbf{f}(\tau, \mathbf{y}(\tau)) d\tau.$$

Nun können wir eine Quadraturformel (auf  $[0, 1]$ ) mit  $s$  Knoten  $c_1, \dots, c_s$  anwenden:

$$\mathbf{y}(t_1) \approx \mathbf{y}_1 (= \mathbf{y}_h(t_1)) = \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{f}(t_0 + c_i h, \mathbf{y}(t_0 + c_i h)), \quad h := t_1 - t_0$$

Die Werte  $\mathbf{y}(t_0 + c_i h)$  sind noch unbekannt. Unser Ziel ist  $|\mathbf{y}(t_1) - \mathbf{y}_1| = O(h^{p+1})$ , und wir haben bereits einen Faktor  $h$  vor der Summe. Deswegen reicht es, wenn wir  $\mathbf{y}(t_0 + c_i h)$  für  $i = 1, 2, \dots, s$  durch eine Methode approximieren, die nur  $O(h^p)$  gut ist.

### Beispiel 2.5.1.

1)  $QF =$  Trapezregel auf  $[0, 1]$

$$Q(f) = \frac{1}{2} (b - a)(f(a) + f(b)) \approx \int_a^b f(t) dt$$

$$s = 2, \quad c_1 = 0, \quad c_2 = 1, \quad b_1 = b_2 = \frac{1}{2}.$$

Somit:

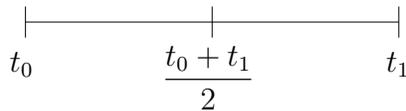
$$\mathbf{y}(t_1) \approx \mathbf{y}_0 + h \left( \frac{1}{2} f(t_0, \mathbf{y}_0) + \frac{1}{2} f(t_1, \mathbf{y}(t_1)) \right).$$

Wir approximieren  $y(t_1)$  aus dem expliziten Euler und organisieren die Berechnung obiger Formel folgendermassen:

$$\begin{cases} k_1 := f(t_0, y_0) \\ k_2 := f(t_0 + h, y_0 + h k_1) \\ y_1 := y_0 + \frac{h}{2} (k_1 + k_2) \text{ explizite Trapezregel} \end{cases}.$$

2)  $QF =$  Mittelpunktsregel und  $y(\frac{1}{2}(t_0+t_1))$  aus dem expliziten Euler-Verfahren:

$$\begin{cases} k_1 := f(t_0, y_0) \\ k_2 := f\left(t_0 + \frac{h}{2}, y_0 + \frac{h}{2} k_1\right) \text{ explizite Mittelpunktsregel} \\ y_1 := y_0 + h k_2 \end{cases}$$



**Definition 2.5.2.** (Runge-Kutta-Verfahren)

Für  $b_i, a_{ij} \in \mathbb{R}$ ,  $c_i := \sum_{j=1}^s a_{ij}$ ,  $i, j = 1, \dots, s$ ,  $s \in \mathbb{N}$  definiert

$$\mathbf{k}_i := \mathbf{f}\left(t_0 + c_i h, \mathbf{y}_0 + h \sum_{j=1}^s a_{ij} \mathbf{k}_j\right), \quad i = 1, \dots, s,$$

$$\Psi^{t_0, t_0+h} \mathbf{y}_0 := \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i$$

ein  $s$ -stufiges Runge-Kutta-Einschrittverfahren (RK-ESV) für AWP mit Inkrementen  $\mathbf{k}_i \in \mathbb{R}^d$ .

Man bemerke, dass die  $\mathbf{k}_i$  selbst in ihrer eigenen Definition vorkommen. Intuitiv kann man das folgendermassen verstehen: in der Quadratur braucht man die Werte  $\mathbf{y}(t_0 + c_i h)$ ; diese werden allerdings schon mittels einer Quadratur berechnet. Wir erhalten also ein Gleichungssystem für die Berechnung der Inkremente. In manchen Fällen (explizite Verfahren) lässt sich die Berechnung der  $\mathbf{k}_i$  allerdings stark vereinfachen.

**Bemerkung 2.5.3.** Falls  $a_{ij} = 0$  für  $i \leq j$ , haben wir es mit einem expliziten Runge-Kutta-Verfahren zu tun. Wir schreiben ein RK-ESV mit Hilfe des Butcher-Schemas:

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \dots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \dots & a_{ss} \\ \hline & b_1 & \dots & b_s \quad . \end{array}$$

$\mathbf{A}$  echte untere Dreiecksmatrix  $\implies$  explizites Runge-Kutta-Verfahren

$\mathbf{A}$  untere Dreiecksmatrix  $\implies$  diagonal-implizites Runge-Kutta-Verfahren (DIRK)

**Beispiel 2.5.4.** (Butcher-Schema für einige explizite RK-ESV)

- Explizites Euler:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \longrightarrow \mathcal{O}(h)$$

- Explizite Trapezregel:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \longrightarrow \mathcal{O}(h^2)$$

- Explizite Mittelpunktsregel:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array} \longrightarrow \mathcal{O}(h^2)$$

- Klassisches RK4:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array} \longrightarrow \mathcal{O}(h^4)$$

- Kutta's 3/8-Rule:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\ \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\ 1 & 1 & -1 & 1 & 0 \\ \hline & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8} \end{array} \longrightarrow \mathcal{O}(h^4)$$

Um vertraut mit der Schreibweise zu werden ist es eine gute Übung, die Verfahren aus ihrem Butcher-Schema abzulesen und sie explizit aufzuschreiben. In manchen Fällen (z.B. explizites Eulerverfahren) wird man auf die Formulierung schon bekannter Verfahren kommen.

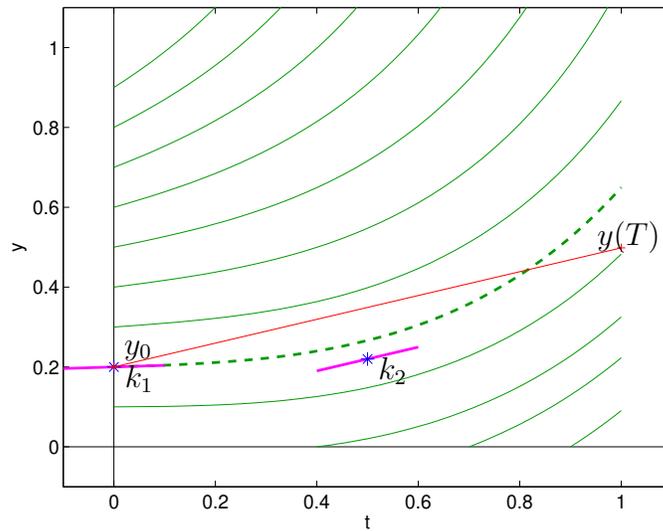
**Beispiel 2.5.5.** (Explizites RK4 für die Riccati-Gleichung) Wir betrachten die folgende Differentialgleichung:

$$\dot{y} = t^2 + y^2, \quad y(0) = 0.2 \tag{2.5.35}$$

Eine geometrische Interpretation der expliziten RK-Methoden wird unten mittels Polygonlinien gezeigt; am Ende jeder Strecke wird eine neue Steigung  $k$  berechnet. Wir wollen verschiedene Methoden mit Graphen veranschaulichen.

Explizite Mittelpunktsregel:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}$$

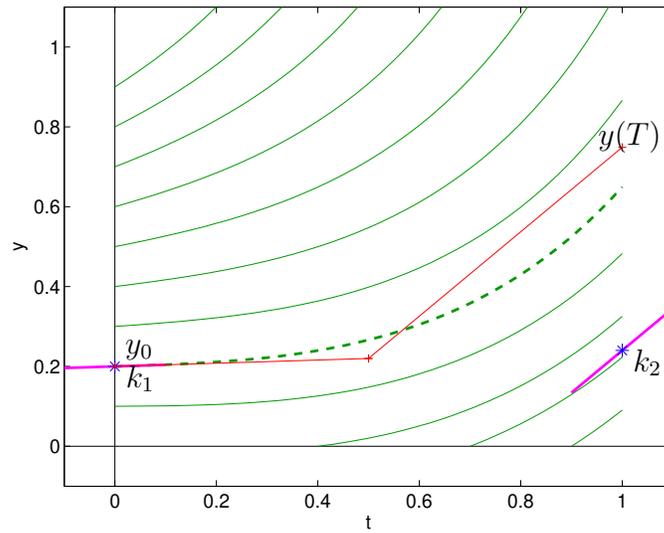


**Grün:** Lösung; **Magenta:** lokale Steigungen  $k_i$ ; **\***: Punkte, wo  $f$  ausgewertet wird;  
**Rot:** Polygonlinie.

**Abb. 2.5.6.** Explizite Mittelpunktsregel als RK

Explizite Trapezregel:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

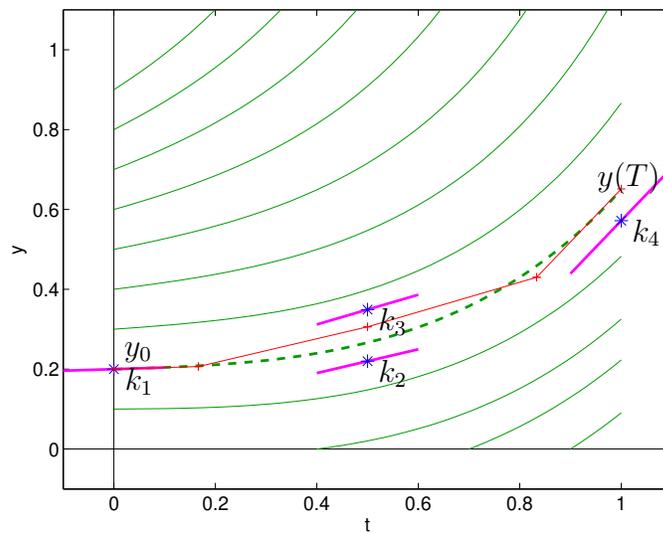


**Grün:** Lösung; **Magenta:** lokale Steigungen  $k_i$ ; **\***: Punkte, wo  $f$  ausgewertet wird;  
**Rot:** Polygonlinie.

**Abb. 2.5.7.** Explizite Trapezregel als RK

Klassische Runge-Kutta-Methode (RK4):

$$\begin{array}{c|cccc}
 0 & 0 & 0 & 0 & 0 \\
 \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\
 \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\
 1 & 0 & 0 & 1 & 0 \\
 \hline
 & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6}
 \end{array} \tag{2.5.36}$$

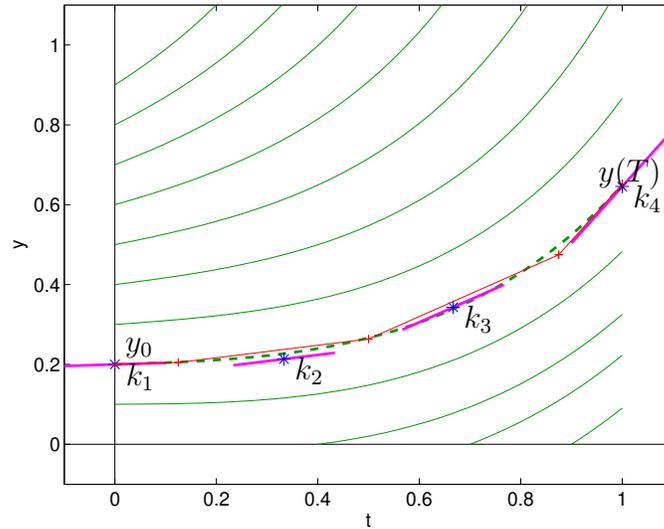


Grün: Lösung; Magenta: lokale Steigungen  $k_i$ ; \*: Punkte, wo  $f$  ausgewertet wird; Rot: Polygonlinie.

Abb. 2.5.8. Klassische Runge-Kutta-Methode (RK4)

Kutta 3/8-Regel:

$$\begin{array}{c|cccc}
 0 & 0 & 0 & 0 & 0 \\
 \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 \\
 \frac{2}{3} & -\frac{1}{3} & 1 & 0 & 0 \\
 1 & 1 & -1 & 1 & 0 \\
 \hline
 & \frac{1}{8} & \frac{3}{8} & \frac{3}{8} & \frac{1}{8}
 \end{array} \tag{2.5.37}$$



Grün: Lösung; Magenta: lokale Steigungen  $k_i$ ; \*: Punkte, wo  $f$  ausgewertet wird; Rot: Polygonlinie.

Abb. 2.5.9. Kutta 3/8-Regel

Beispiel 2.5.10. (Konvergenz einfacher RK-ESV)

- $\dot{y} = 10y(1 - y)$  (logistische ODE (2.1.6)),  $y(0) = 0.01, T = 1$

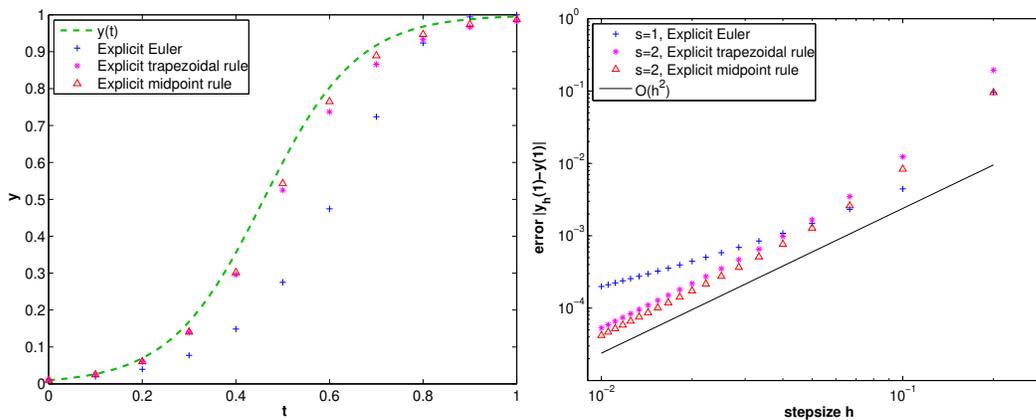


Abb. 2.5.11. Links:  $y_h(j/10), j = 1, \dots, 10$  für einige explizite RK-Methoden  
Rechts: Fehler zur Endzeit  $y_h(1) - y(1)$

Wir können diese Graphen mittels des folgenden Codes erhalten:

Code 2.5.12: Explizites Runge-Kutta Verfahren

```

from numpy import linspace, array, shape, zeros, exp, squeeze, arange
2 from matplotlib.pyplot import figure, loglog, show, grid, title, xlabel, ylabel,
legend
    
```

```

4 #the logistic ODE.
  f = lambda y: 10*y*(1-y)
6
  #the exact solution:
8  y_analytic = lambda t: exp(10*t)/(99.+exp(10*t))
10 t_init = 0.0; t_end = 1.0
   y_0 = 0.01
12
  #we want to adopt the compact way of writing of the Butcher Scheme.
14 #BE CAREFUL: THE FOLLOWING CODE IS ONLY VALID FOR EXPLICIT METHODS
   #later we will see a generalization for implicit methods
16
  def RK_expl_step(B,t,y,h,rhs,dim):
18     """B: Butcher Schema
        t: time
20     y: y(t-h)
        h:time step
22     rhs: ODE
        dim: dimension of y. (we could calculate it directly in the code)
24     """

26     #first we want to know the number of increments s
     s = B.shape[0] - 1
28     #then we extract the three parts of the butcher scheme
     b = B[s][1:]; c = B[:,0][:-1]; A = B[:-1,1:]
30     #now we prepare a "container" for the increments
     K = zeros((dim,s))
32
     #now we calculate the increments.
34     #since we are using an explicit method, we just compute them in the order
     they appear in
     for i in range(s):
36         K[:,i] = rhs(y + h*dot(A[i,:].T,K.T))
     #in python, dot is a matrix product.
38     #make sure you understand how we've just translated into code the definition
     of the  $k_j$ 
     #note: if the ODE is not autonomous we have to write
40     #K[:,i] = rhs(t + c[i]*h, y + h*dot(A[i,:].T,K.T))
     """ #the code for the implicit case would be:
42     K = K.ravel() #flatten the array for use in fsolve()
     def F(x):
44         z = x.reshape((dim,s)); temp = zeros((dim,s))
         for i in range(s):
46             temp[i] = rhs(t + c[i]*h, y + h*dot(A[i,:].T,z.T))
         return temp.ravel()
48     K = fsolve(lambda x: x - F(x), zeros((dim,s)).ravel() )
     #finally we reshape K:
50     K.shape = ((dim,s))
     """

52     #finally, we calculate the single-step solution
     tnew = t + h
54     ynew = y + h*squeeze(dot(K,b))

```

```

56     return tnew, ynew

58 #the butcher schemes
B_explEul = array([[0.,1.],[0.,1.]])
60 B_explMid = array([[0.,0.,0.],[0.5,0.5,0.],[0.,0.,1.]])
B_explTrap = array([[0.,0.,0.],[1.,1.,0.],[0.,0.5,0.5]])

62
#first we want to plot the solutions
64 N = 10 #first we consider a small N to see the differences between the methods
t, h = linspace(t_init, t_end, N, retstep="True")
66 y_explEul = zeros(N); y_explEul[0] = y_0
y_explMid = zeros(N); y_explMid[0] = y_0
68 y_explTrap = zeros(N); y_explTrap[0] = y_0

70 for k in range(N-1):
    y_explEul[k+1] = RK_expl_step(B_explEul, t[k], y_explEul[k], h, rhs=f,
    dim=1)[1]
72     #note: ...[0] would be the time, which we already know (we wanted to write a
    more general function above)

74     y_explMid[k+1] = RK_expl_step(B_explMid, t[k], y_explMid[k], h, rhs=f,
    dim=1)[1]

76     y_explTrap[k+1] = RK_expl_step(B_explTrap, t[k], y_explTrap[k], h, rhs=f,
    dim=1)[1]

78 figure()
title("Comparison of simple explicit RK methods - logistic ODE")
80 plot(linspace(t_init,t_end,1000),y_analytic(linspace(t_init,t_end,1000)),"--",
label = "exact solution")
plot(t, y_explEul,"o", label = "expl. Euler")
82 plot(t, y_explMid,"v", label = "expl. Midpoint")
plot(t, y_explTrap,"^", label = "expl. Trapez.")
84 legend(loc = "best")
grid(True)
86 xlabel("$t$"); ylabel("$y(t)$")
show()

88
#now we want to take a look at the convergence of these methods
90 #to this end, as we did in the previous chapters, we consider the error on
y(t)
#we do this for different values of N
92 #since we are just interesting about the value of y at t = t_end we do not store
all the values in an array

94 y_end_exact = y_analytic(t_end)

96 Ns = arange(10,1500,50)
err_explEul = []
98 err_explMid = []
err_explTrap = []

100 for N in Ns:
102     h = (t_end-t_init)/N

```

```

104 #initialize
106 y_explEul = y_explMid = y_explTrap = y_0

108 for k in range(N):

110     #expl euler
112     y_explEul = RK_expl_step(B_explEul, t_init + k*h, y_explEul, h, rhs=f,
114     dim=1)[1]

116     #expl midpoint
118     y_explMid = RK_expl_step(B_explMid, t_init + k*h, y_explMid, h, rhs=f,
120     dim=1)[1]

122     #expl trapez.
124     y_explTrap = RK_expl_step(B_explTrap, t_init + k*h, y_explTrap, h,
126     rhs=f, dim=1)[1]

128     #now calculate the errors
130     err_explEul += [abs(y_explEul - y_end_exact)]
132     err_explMid += [abs(y_explMid - y_end_exact)]
134     err_explTrap += [abs(y_explTrap - y_end_exact)]

136 figure()
138 title("Convergence of Lie-Trotter and Strang Splitting with different
140 combinations of methods")
142 grid(True); xlabel("$h$"); ylabel("err")
144 loglog((t_end-t_init)/Ns,err_explEul,"x", label = "expl. Euler")
146 loglog((t_end-t_init)/Ns,err_explMid,"x", label = "expl. Midpoint")
148 loglog((t_end-t_init)/Ns,err_explTrap,"x", label = "expl. Trapez.")
150 legend(loc="best")
152 show()

```

Wir beobachten algebraische Konvergenz; sowohl die explizite Trapezregel als auch die explizite Mittelpunktsregel sind Verfahren zweiter Ordnung.

**Bemerkung 2.5.13.** Es gibt folgende Schranken (“Butcher barriers”) für die expliziten RK-ESV:

Ordnung $p$	1	2	3	4	5	6	7	8	$\geq 9$
Minimale Anzahl $s$ an Stufen	1	2	3	4	6	7	9	11	$\geq p + 3$

Alternativ zur Kombination von Integration und Quadratur, kann man die Lösung zwischen  $t_0$  und  $t_0 + h$  durch einen Polynom vom Grade  $s$  approximieren; dieser wird dann so bestimmt, dass die Differentialgleichung an den Kollokationspunkten  $t_0 + c_i h$  erfüllt wird. Diese Methoden sind die sogenannten **Kollokationsverfahren**, die sich als RK-ESV mit

$$a_{ij} = \int_0^{c_i} \ell_j(\tau) d\tau, \quad b_i = \int_0^1 \ell_i(\tau) d\tau$$

schreiben lassen, wobei  $\ell_i$  die Lagrange-Polynomen zur Stützstellen  $c_1, \dots, c_s$  sind. Eine sehr wichtige Familie solcher Methoden sind die Gauss-Kollokationsmethoden der Ordnungen 4 und 6:

$$\begin{array}{c|cc}
 \frac{1}{2} - \frac{\sqrt{3}}{6} & \frac{1}{4} & \frac{1}{4} - \frac{\sqrt{3}}{6} \\
 \frac{1}{2} + \frac{\sqrt{3}}{6} & \frac{1}{4} + \frac{\sqrt{3}}{6} & \frac{1}{4} \\
 \hline
 & \frac{1}{2} & \frac{1}{2}
 \end{array} \tag{2.5.38}$$

$$\begin{array}{c|ccc}
 \frac{1}{2} - \frac{\sqrt{15}}{10} & \frac{5}{36} & \frac{2}{9} - \frac{\sqrt{15}}{15} & \frac{5}{36} - \frac{\sqrt{15}}{30} \\
 \frac{1}{2} & \frac{5}{36} + \frac{\sqrt{15}}{24} & \frac{2}{9} & \frac{5}{36} - \frac{\sqrt{15}}{24} \\
 \frac{1}{2} + \frac{\sqrt{15}}{10} & \frac{5}{36} + \frac{\sqrt{15}}{30} & \frac{2}{9} + \frac{\sqrt{15}}{15} & \frac{5}{36} \\
 \hline
 & \frac{5}{18} & \frac{4}{9} & \frac{5}{18}
 \end{array} \tag{2.5.39}$$

Explizite Verfahren		Implizite Verfahren	
Explizites Euler	$p = 1$	Implizites Eulerverfahren	$p = 1$
Explizite Trapezregel	$p = 2$	Implizite Mittelpunktsregel	$p = 2$
Explizite Mittelpunktsregel	$p = 2$	Gauss-Kollokationsverfahren	$p = 2s$
Klassisches Runge-Kutta-Verfahren	$p = 4$		
Kutta 3/8-Regel	$p = 4$		

Ordnungsschranken:

- Für explizite Runge-Kutta-Verfahren  $p \leq s$
- Für allgemeine Runge-Kutta-Verfahren  $p \leq 2s$

Gauss-Kollokationsverfahren realisieren maximale Ordnung.

Hier ist eine Implementierung einiger Runge-Kutta Verfahren im Objekt-orientierten Styl:

Code 2.5.14: Runge-Kutta Verfahren: Objekt-orientierten Styl

```

1  # -*- coding: utf-8 -*-
2  import numpy as np
3  from scipy.optimize import fsolve
4
5  class RungeKutta(object):
6      """Basisklasse fuer alle Runge-Kutta Verfahren."""
7
8      def __init__(self, A, b):
9          """Runge-Kutta Loeser mit dem Butcher Tableau (A, b).
10
11          Wir implementieren nur konsistente Runge-Kutta Verfahren,

```

```

    d.h. solche mit
13
        c[i] = sum_j A[i,j]
15
    Input:
17
        A: Butcher Matrix A.
        b: Butcher Vektor b.
19
    """
    self.A = A.copy()
21
    self.b = b.copy()
    self.c = np.sum(A, axis=1)
23
    self.n_stages = A.shape[0]
25
    # Die method `__call__` erlaubt uns das object wie eine Funktion zu
    # benutzen:
27
    # rk = RungeKutta(A, b)
    # t, y = rk(f, y0, t_end, n_steps)
29
    def __call__(self, f, y0, t_end, n_steps):
31
        """ Berechnet die Approximation der Loesung.

        Input:
33
            f : Rechte Seite der ODE.
35
            y0 : Anfangsbedingung.
            t_end : Endzeit.
37
            n_steps : Anzahl Zeitschritte.
        """
39
        y = np.empty((n_steps+1, y0.size))
        t, dt = np.linspace(0, t_end, n_steps+1, retstep=True)
41
        y[0,:] = y0
43
        for i in range(n_steps):
            y[i+1,:] = self.step(f, y[i,:], t[i], dt)
45
        return t, y
47
    # NOTE Wir wollen drei Faelle unterscheiden: explizites RK, semi-implizites
    # RK und voll-implizites RK. Die drei Faelle muessen unterschieden werden,
    # da man bei explizitem RK nicht unnoetig ein Gleichungssystem loesen will.
    # Diese Funktionalitaet wird also in drei seperaten Unterklassen
    # implementiert.
    # def step(self, t, y0, dt):
    #     # ...
    #     return y1
53
55
57
class ExplicitRungeKutta(RungeKutta):
    """Implementiert explizite Runge-Kutta Verfahren."""
59
    # NOTE Den Konstruktor brauchen wir nicht nochmals zu implementieren. Der
    # default ist der Konstruktor von RungeKutta (also der Basisklasse).
    # Dies gilt auch fuer alle anderen Funktionen.
    # def __init__(self, A, b, c):
    #     self.A = A.copy()
    #     # ...
63
65

```

```

# NOTE auch __call__ gibt es schon, siehe oben.
67 # def __call__(self, f, y0, t_end, n_steps):
#     # ...
69 #     return t, y

71 def step(self, rhs, y0, t, dt):
    """ Fuehrt einen expliziten RK Schritt aus.
73
    Input:
75     rhs : Rechte Seite der ODE.
77     y0  : Approximative Loesung zur Zeit `t`.
79     t   : Aktuelle Simulationszeit.
81     dt  : Zeitschritt, \Delta t.
83
    """
85     # Diese Funktion berechnet einen Schritt  $y^{\{n\}} \rightarrow y^{\{n+1\}}$ .
87     # Aber Namen wie (yn, ynp1) sind etwas kryptisch, deswegen (y0, y1), denn
89     # eigentlich koennen wir das Problem als eine ODE mit Anfangsbedingung
91     # `y0 = y0`, `t0 = t`, `t_end = t + dt` betrachten. Es besteht also keine
93     # Verwirrungsgefahr :)
95     #
97     # Eventuell ist (y_current, y_next) eine Alternative, aber viel laenger
99     # auch verglichen mit `t`, `dt`.
101
103     #print("ExRK step t = {:.3e} ...".format(t))#, end="")
105
107     A, b, c = self.A, self.b, self.c
109     s = self.n_stages
111     dydt = np.empty((y0.size, s)) # werden auf Papier `k` genannt.
113
115     for i in range(s):
117         y_star = y0 + dt*np.sum(A[i,:]*dydt[:,i], axis=-1)
119         dydt[:,i] = rhs(t + dt*c[i], y_star)

```

```

A[i,i]*dydt_i)
    #    return y_i - y_star
121
    #initial_guess = y0
123    #dydt[:,i] = rhs(t + c[i]*dt, fsolve(F, initial_guess))

    def F(x):
125        x_star = rhs(t + c[i]*dt, y0 + dt*(np.sum(A[i,:i]*dydt[:,i],
axis=-1) + A[i,i]*x))
127        return x - x_star

        initial_guess = y0#dydt[:,i]
129        dydt[:,i] = fsolve(F,initial_guess)
131    y1 = y0 + dt*np.sum(b*dydt, axis=-1)
    return y1
133

class ImplicitRungeKutta(RungeKutta):
135    """Voll implizites Runge-Kutta Verfahren."""

    def step2(self, rhs, y0, t, dt):
137        """Berechnet einen Schritt des impliziten RK Verfahrens.

139        Vergleiche `ExplicitRungeKutta.step`.
        """
141        A, b, c = self.A, self.b, self.c
143        s = self.n_stages

    def F(dydt):
145        dydt_star = np.empty_like(dydt)
147        for i in range(s):
            y_star = y0 + dt*np.sum(A[i,:]*dydt[:,i], axis=-1)
149            dydt_star[:,i] = rhs(t + c[i]*dt, y_star)

151        return dydt - dydt_star

    initial_guess = np.empty((y0.size, s))
153    for i in range(s):
155        initial_guess[:,i] = rhs(t, y0)

157    dydt = nicer_fsolve(F, initial_guess)

159    y1 = y0 + dt*np.sum(b*dydt, axis=-1)
    return y1
161

    def step(self, rhs, y0, t, dt):
163        """Berechnet einen Schritt des diagonal impliziten RK Verfahrens.

165        Vergleiche `ExplicitRungeKutta.step`.
        """
167        A, b, c = self.A, self.b, self.c
        s = self.n_stages

169        #print("hello")
171        # Mein Vorschlag:

```

```

173     def F(x):
174         x_star=np.empty_like(x)
175         for i in range(s):
176             x_star[:,i] = rhs(t + c[i]*dt, y0 + dt*np.sum(A[i,:]*x[:,:],
axis=-1))# + A[i,i]*x))
177         return x - x_star

178     initial_guess = np.empty((y0.size, s))
179     for i in range(s):
180         initial_guess[:,i] = rhs(t, y0)
181     dydt = nicer_fsolve(F,initial_guess)

182     y1 = y0 + dt*np.sum(b*dydt, axis=-1)
183     return y1

184 def nicer_fsolve(F, initial_guess):
185     """Wrapper fuer `scipy.fsolve`.
186
187     Dies ist nuetzlich, wenn man fsolve fuer 2D oder 3D Arrays
188     brauchen will.
189     """
190     shape = initial_guess.shape
191
192     initial_guess = initial_guess.reshape((-1,))
193     result = fsolve(lambda x: F(x.reshape(shape)).reshape((-1)), initial_guess)
194
195     return result.reshape(shape)

196 # Das Ordnung 5 Schema aus `ode45`, aka Fehlberg.
197 def exrk_o5():
201     A = [[ 0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
202          [ 0.25,  0.0,  0.0,  0.0,  0.0,  0.0],
203          [ 3.0/32.0,  9.0/32.0,  0.0,  0.0,  0.0,  0.0],
204          [ 1932.0/2197.0,  -7200.0/2197.0,  7296.0/2197.0,  0.0,  0.0,  0.0],
205          [ 439.0/216.0,  -8.0,  3680.0/513.0,  -845.0/4104,  0.0,  0.0],
206          [-8.0/27.0,  2.0,  -3544.0/2565.0,  1859.0/4104.0,  -11.0/40.0,  0.0]]
207     b = [16.0/135.0,  0.0,  6656.0/12825.0,  28561.0/56430.0,  -9.0/50.0,
208          2.0/55.0]

209     return ExplicitRungeKutta(np.array(A), np.array(b))

210 # Runge-Kutta Butcher scheme for implicit midpoint rule
211 def dirk_im():
212     A = np.array([[0.5]])
213     b = np.array([1.0])
214
215     return SemiImplicitRungeKutta(A, b)

216 # Diagonal implizites RK schema mit 3 Stufen und Ordnung 4.
217 def dirk_o4():
218     alpha = 2.0*np.cos(np.pi/18)/np.sqrt(3)
219     alpha2 = alpha*alpha
220
221     A = np.array([[ (1.0+alpha)/2.0,  0.0,  0.0,],
222
```

```

225         [-alpha/2.0, (1.0+alpha)/2.0, 0.0],
           [1.0+alpha, -(1.0+2.0*alpha), (1.0+alpha)/2.0]])
227     b = np.array([1.0/(6.0*alpha2), 1.0-1.0/(3.0*alpha2), 1.0/(6.0*alpha2)])
229
230     return SemiImplicitRungeKutta(A, b)
231
232 # Runge-Kutta Gauss Collocation of Order 4
233 def firk_o4():
234     A = np.array([[0.25, 0.25 - np.sqrt(3)/6.0],
235                  [0.25 + np.sqrt(3)/6.0, 0.25]])
236
237     b = np.array([0.5, 0.5])
238
239     return ImplicitRungeKutta(A, b)
240
241 # Runge-Kutta Gauss Collocation of Order 6
242 def firk_o6():
243     A = np.array([[ 5.0/36.0, 2.0/9-np.sqrt(15)/15.0,
244                    5.0/36.0-np.sqrt(15)/30.0,
245                    [ 5.0/36.0+np.sqrt(15)/24.0, 2.0/9,
246                    5.0/36.0-np.sqrt(15)/24.0],
247                    [ 5.0/36.0+np.sqrt(15)/30.0, 2.0/9+np.sqrt(15)/15.0,
248                    5.0/36.0 ]])
249
250     b = np.array([5.0/18.0, 4.0/9.0, 5.0/18.0])
251
252     return ImplicitRungeKutta(A, b)
253
254 if __name__ == "__main__":
255     #solver = firk_o4()
256     #solver = exrk_o5()
257     f = lambda t, y: -y
258     y0 = np.array([1.0, 2.0])
259
260     t_end, n_steps = 1.0, 100
261
262     #solver = dirk_o4()
263     #t, y = solver(f, y0, t_end, n_steps)
264     #print(y[-1,:] - y0*np.exp(-t_end))
265
266     solvers = [dirk_o4, firk_o4, exrk_o5]
267
268     for solvername in solvers:
269         solver = solvername()
270         print(solvername.__name__)
271         t, y = solver(f, y0, t_end, n_steps)
272         print(y[-1,:] - y0*np.exp(-t_end))

```

## 2.6 Schrittweitensteuerung

Wir möchten die adaptive Wahl des Leitgitters  $\{t_0, t_1, t_2, \dots, t_N\}$  durch lokale Fehlerabschätzung optimieren:  $t_{k-1}, t_k, y_k \implies$  Wahl von  $t_{k+1}$ .

**Idee:**

Nehmen wir  $\Psi^{t,t+h}$  und  $\tilde{\Psi}^{t,t+h}$  diskrete Evolutionen (verschiedene numerische Verfahren verschiedener Ordnung!)

Ordnung  $\tilde{\Psi} >$  Ordnung von  $\Psi = p$

$$\Phi^{t,t+h}y(t_k) - \Psi^{t,t+h}y(t_k) \approx \tilde{\Psi}^{t,t+h}y(t_k) - \Psi^{t,t+h}y(t_k) =: \text{EST}_k$$

Sei eine Toleranz TOL gegeben. Wenn  $\text{EST}_k < \text{TOL}$ , dann wählt man  $2h$  oder akzeptiert  $h$  für den nächsten Schritt. Wenn  $\text{EST}_k > \text{TOL}$ , dann nimmt man  $\frac{h}{2}$  im nächsten Schritt.

**Bemerkung 2.6.1.** Für ein Verfahren der Ordnung  $p$  brauchen wir lokal Ordnung  $p + 1$  (da die Fehlerakkumulation in  $N$  Schritte der Länge  $h$  einen  $h$  kostet!):

$$N \cdot h^{p+1} = N \cdot h \cdot h^p.$$

Da  $\tilde{\Psi}^{t,t+h}y(t_k) - \Phi^{t,t+h}y(t_k) = O(h^{p+2})$ , haben wir  $\text{EST}_k \approx ch^{p+1} \stackrel{!}{=} \text{TOL}$  für  $h \ll 1$ . Das optimale  $h^*$  wählen wir so, dass  $c(h^*)^{p+1} = \text{TOL}$ , aber zuerst schätzen wir  $c$  ab durch

$$c = \frac{\text{EST}_k}{h^{p+1}},$$

und dann  $h^* := h^{p+1} \sqrt{\frac{\text{TOL}}{\text{EST}_k}}$ .

**Code 2.6.2:** Einfache Zeitschrittsteuerung

```

def odeintadapt_ext(Psilow, Psihigh, T, y0, fy0=None, h0=None, hmin=None,
reltol=1e-2, abstol=1e-4):
2   r"""Adaptive Integrator for stiff ODEs and Systems
   based on two suitable methods of different order.
4
   Input:
6   Psilow:  Integrator of low order
   Psihigh: Integrator of high order
8   T:      Endtime: R^1
   y0:     Initial value: R^(nx1)
10  fy0:    The value f(y0): R^(nx1) used to estimate initial timestep size
   h0:     Initial timestep (optional)
12  hmin:   Minimal timestep (optional)
   reltol: Relative tolerance (optional)
14  abstol: Absolute Tolerance (optional)
16
   Output:
18  t:      Timesteps: R^K with K the number of steps performed
   y:      Solution at the timesteps: R^(nxK)
   rej:    Time of rejected timesteps: R^G with G the number of rejections
20  ee:     Estimated error: R^K

```

```

22     """
23     # Heuristic choice of initial timestep size
24     if h0 is None:
25         h0 = T / (100.0*(norm(fy0) + 0.1))
26     if hmin is None:
27         hmin = h0 / 10000.0
28
29     # Initial values
30     yi = atleast_2d(y0).copy()
31     ti = 0.0
32     hi = h0
33     n = yi.shape[0]
34
35     # Storage
36     t = [ti]
37     y = [yi]
38     rej = []
39     ee = [0.0]
40
41     while ti < T and hi > hmin:
42         yh = Psilow(hi, yi)
43         yH = Psihigh(hi, yi)
44         est = norm(yh - yH)
45         if est < min(reltol*norm(yi), abstol):
46             # Timestep is too long and ends beyond T
47             if T - ti < hi:
48                 hi = T - ti
49                 yi = Psihigh(hi, yi)
50             else:
51                 yi = yH
52
53             ti = ti + hi
54             hi = 1.1*hi
55
56             y.append(yi)
57             t.append(ti)
58             ee.append(est)
59         else:
60             hi = 0.5*hi
61
62             rej.append(ti)
63
64     return array(t).reshape(-1), array(y).T.reshape(n,-1), array(rej), array(ee)

```

Wir wollen uns nun den Code [2.6.2](#) im Detail anschauen:

- Argumente:
  - `Psilow`, `Psihigh`: diskrete Evolutionsoperatoren verschiedener Ordnungen, die den Anfangswert und den Zeitschritt als Argumente erwarten
  - `T`: Endzeit  $T > 0$
  - `y0`: Anfangswert  $y_0$

- **h0**: Zeitschritt  $h_0$  am Anfang
- **hmin**: minimaler Zeitschritt. Die Zeit-Integration endet wenn  $h_k < h_{\min}$ , was eine Erkennung von Singularitäten erlaubt.
- **reltol**, **abstol**: relative und absolute Toleranzen
- Zeile 40: überprüfe ob die Zeit-Integration beendet werden soll, entweder weil die Endzeit oder eine Singularität erreicht worden ist (d.h.  $h < h_{\min}$ ).
- Zeilen 41-42: je ein Schritt beider diskreter Evolutionen .
- Zeile 43: Schätzung des Fehlers.
- Zeile 44: entscheide, ob der aktuelle Zeitschritt annehmbar ist.
- Zeilen 47-57: der aktuelle Zeitschritt wird akzeptiert, die Lösung und die Zeit werden gespeichert und ein grösserer Zeitschritt wird für den nächsten Schritt vorgeschlagen.
- Zeilen 48-51: der aktuelle Zeitschritt ist zu gross: es wird ein neuer Versuch mit der halben Zeitschrittgrösse  $\frac{h}{2}$  durchgeführt.
- Output:
  - **t**: Zeitgitter  $t_0 < t_1 < t_2 < \dots < t_N < T$ , wobei  $t_N < T$  einen vorzeitigen Abbruch anzeigt
  - **y**: Vektor  $(\mathbf{y}_k)_{k=0}^N$

**Beispiel 2.6.3.** (Einfache Wahl der adaptiven Zeitschrittweite) Wir betrachten hier

$$\dot{y} = \cos(\alpha y)^2, \text{ mit } \alpha > 0.$$

Die exakte Lösung ist  $y(t) = \arctan(\alpha(t - c))/\alpha$  für  $y(0) \in ] -\pi/2, \pi/2[$ . Wir wollen das explizite Eulerverfahren und die explizite Trapezregel als diskrete Evolutionsoperatoren wie im vorigen Algorithmus anwenden.

**Code 2.6.4:** Function for Ex. Theorem 2.6.3

```

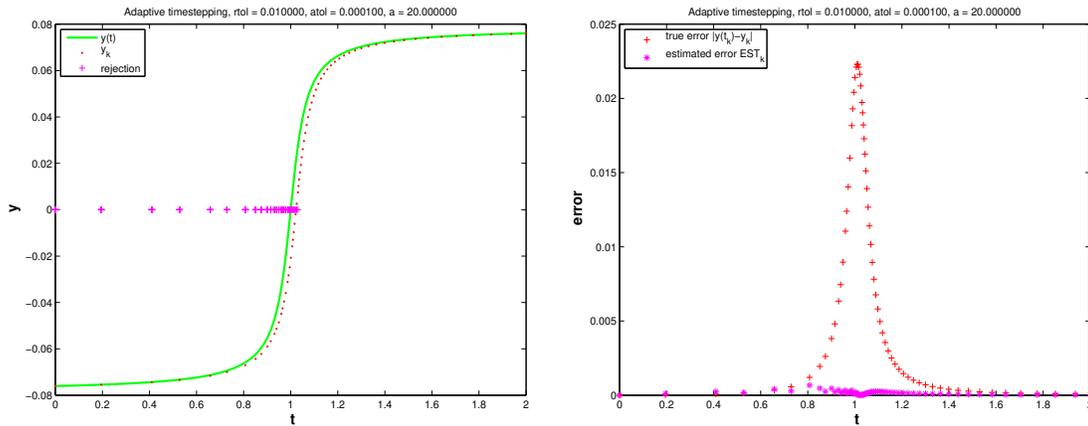
1 def odeintadaptdriver(T,a,reltol=1e-2,abstol=1e-4):
2     """Simple adaptive timestepping strategy
3     based on explicit Euler and explicit trapezoidal
4     rule
5     """
6
7     # autonomous ODE and its general solution
8     f = lambda y: (cos(a*y)**2)
9     sol = lambda t: arctan(a*(t-1))/a
10    # Initial state

```

```

11     y0 = sol(0)
13
13     # Discrete evolution operators
13     # Explicit Euler
15     Psilow = lambda h,y: y + h*f(y)
15     # Explicit trapzoidal rule
17     Psihigh = lambda h,y: y + 0.5*h*( f(y) + f(y+h*f(y)) )
19
19     # Heuristic choice of initial timestep and
19     h0 = T/(100.0*(norm(f(y0))+0.1)); hmin = h0/10000.0;
21     # Main adaptive timestepping loop
21     t,y,rej,ee = odeintadapt_ext(Psilow,Psihigh,T,y0,h0,reltol,abstol,hmin)
23     y = squeeze(y)
23     # Plotting the exact the approximate solutions and rejected timesteps
25     fig = plt.figure()
25     tp = r_[0:T:T/1000.0]
27     plt.plot(tp,sol(tp),'g-',linewidth=2, label=r'$y(t)$')
27     plt.plot(t,y,'r.',label=r'$y_k$')
29     plt.plot(rej,zeros(len(rej)), 'm+',label='rejection')
29     plt.title('Adaptive timestepping, rtol = %1.2f, atol = %1.4f, a = %d' %
(reltol,abstol,a))
31     plt.xlabel(r'${\bf t}$',fontsize=14)
31     plt.ylabel(r'${\bf y}$',fontsize=14)
33     plt.legend(loc='upper left')
33     plt.show()
35 # plt.savefig('../PICTURES/odeintadaptsol.eps')
37
37     print('%d timesteps, %d rejected timesteps' % (size(t)-1,len(rej)))
39
39     # Plotting estimated and true errors
39     fig = plt.figure()
41     plt.plot(t,abs(sol(t) - y),'r+',label=r'true error $|y(t_k)-y_k|$')
41     plt.plot(t,ee,'m*',label='estimated error EST_k')
43     plt.xlabel(r'${\bf t}$',fontsize=14)
43     plt.ylabel(r'${\bf error}$',fontsize=14)
45     plt.legend(loc='upper left')
45     plt.title('Adaptive timestepping, rtol = %1.2f, atol = %1.4f, a = %d' %
(reltol,abstol,a))
47     plt.show()
47 # plt.savefig('../PICTURES/odeintadapterr.eps')
49
49 if __name__ == '__main__':
51     odeintadaptdriver(T=2,a=20,reltol=1e-2,abstol=1e-4)

```



Statistik: 66 timesteps,

131 rejected timesteps

Abb. 2.6.5. Einfache Zeitschrittsteuerung

Wir beobachten, dass diese einfache Zeitschrittsteuerung in der Lage ist, das lokale Verhalten der Lösung zu berücksichtigen, obwohl der geschätzte Fehler und der wahre Fehler nicht notwendigerweise nah aneinander liegen.

**Beispiel 2.6.6.** (Mit Adaptivität zum Erfolg)

Wir wiederholen das obige Experiment mit  $y(0) = 0$  und beobachten die Abhängigkeit des Fehlers von der Anzahl der Zeitschritten.

Code 2.6.7: function for Ex. Theorem 2.6.6

```

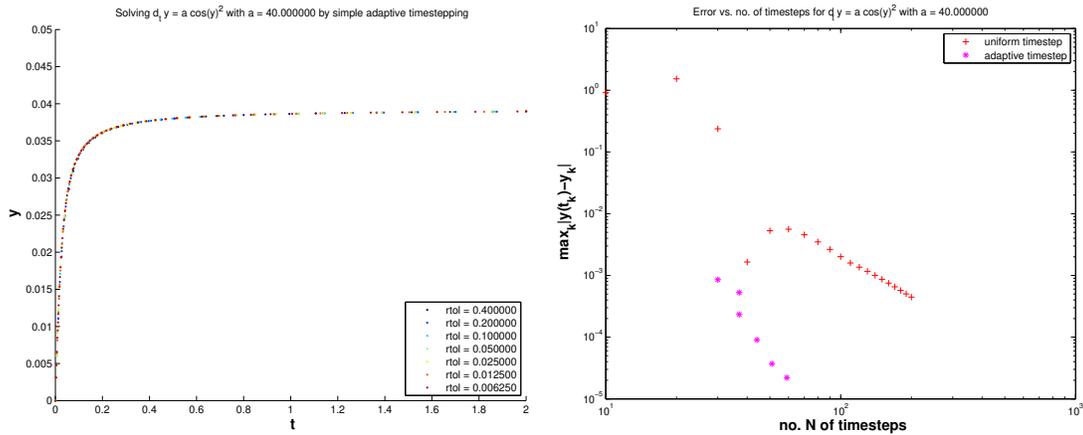
1 from numpy import *
2 import matplotlib.pyplot as plt
3 from odeintadapt import odeintadapt_ext
4
5 def adaptgain(T=2.0, a=40.0, reltol=1e-1, abstol=1e-3):
6     """Experimental study of gain through simple adaptive timestepping
7     strategy of Code~\ref{mc:odeintadapt} based on explicit Euler
8     and explicit trapezoidal rule
9     """
10
11     # autonomous ODE  $\dot{y} = \cos(ay)$  and its general solution
12     f = lambda y: (cos(a*y)**2.0)
13     sol = lambda t: arctan(a*(t))/a # the constant c is 0: arctan(a*(t-c))/a
14     # Initial state  $y_0$ 
15     y0 = sol(0)
16
17     # Discrete evolution operators
18     # Explicit Euler, lower order
19     Psilow = lambda h,y: y + h*f(y)
20     # Explicit trapezoidal rule, higher order
21     Psihigh = lambda h,y: y + 0.5*h*(f(y)+f(y+h*f(y)))
22
23     # Loop over uniform timesteps of varying length and

```

```

#integrate ODE by explicit trapezoidal rule
25 nvals = r_[10:210:10]
err_unif = []
27 for N in nvals:
    h = T/N; t = 0; y = y0; err = 0
    29 for k in arange(1,N+1):
        y = Psihigh(h,y); t = t+h
        31 err = max(err,abs(sol(t) - y));
        err_unif.append(err)
33 # Run adaptive timestepping with various tolerances, which is the only way
# to make it use a different total number of timesteps.
35 # Plot the solution sequences for different values of the relative tolerance.
plt.figure()
37 # axis([0 2 0 0.05]); hold on; col = colormap;
ntimes = [] #container for number of timesteps
39 err_ad = [] #container for the maximal error w.r.t. the analytical solution
tols = [] #container for the tolerances
41 rejs = [] #container for the rejected trials
l = 1
43 for rtol in reltol*2.0**r_[2:-5:-1]:
    # Crude choice of initial timestep and  $h_{\min}$ 
45 h0 = T/10.0; hmin = h0/10000.0
    # Main adaptive timestepping loop, see Code 2.6.2
47 t,y,rej,ee = odeintadapt_ext(Psilow,Psihigh,T,y0,h0,rtol,0.01*rtol,hmin)
ntimes.append(len(t)-1)
49 err_ad.append(max(abs(sol(t) - y)))
tols.append(rtol)
51 rejs.append(len(rej))
print('rtol = %1.3f: %d timesteps, %d rejected timesteps' %
(rtol,len(t)-1,len(rej)))
53 plt.plot(t,y,'.',label='rtol = %f'%rtol#,'color',col(10*(l-1)+1,:));
l = l+1
55 plt.xlabel(r'${\bf t}$', fontsize=14)
plt.ylabel(r'${\bf y}$', fontsize=14)
57 plt.legend(loc='lower right')
plt.title(r'Solving  $d_t y = a \cdot \cos(y)^2$  with  $a = ' + str(a) + '$ by
simple adaptive timestepping')
59 # plt.savefig('../PICTURES/adaptgainsol.eps')

61 # Plotting the errors vs. the number of timesteps
plt.figure()
63 plt.loglog(nvals, err_unif,'r+',label='uniform timestep')
plt.loglog(ntimes, err_ad,'m*',label='adaptive timestep')
65 plt.xlabel('no. N of timesteps',fontsize=14)
plt.ylabel(r'$\max_k |y(t_k)-y_k|$',fontsize=14)
67 plt.title(r'Error vs. no. of timesteps for  $d_t y = a \cdot \cos(y)^2$  with
$a = '+str(a)+'$')
plt.legend(loc='upper right')
69 plt.show()
# plt.savefig('../PICTURES/adaptgain.eps')
71
73 if __name__ == '__main__':
    adaptgain()$ 
```



Lösungen  $(y_k)_k$  für verschiedene  $rtol$

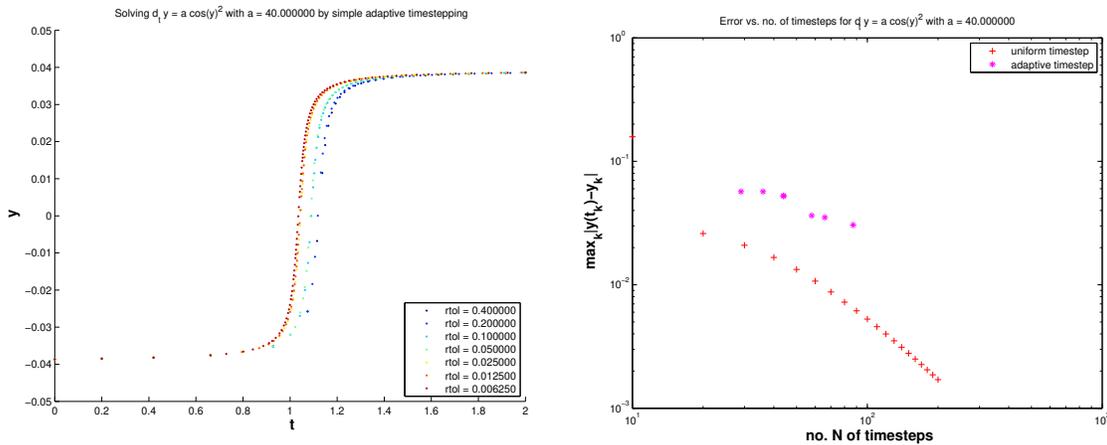
Fehler gegen Anzahl der Zeitschritte

Abb. 2.6.8. Mit Adaptivität zum Erfolg

Wir beobachten, dass diese einfache Zeitschrittsteuerung bessere Genauigkeit mit dem selben numerischen Aufwand liefert.

**Beispiel 2.6.9.** (Mit Adaptivität zum Misserfolg)

Wir wiederholen das obige Experiment mit  $y(0) = -0.0386$ .



Lösungen  $(y_k)_k$  für verschiedene  $rtol$

Fehler gegen Anzahl der Zeitschritte

Abb. 2.6.10. Mit Adaptivität zum Misserfolg

Wir beobachten, dass diese einfache Zeitschrittsteuerung diesmal schlechtere Genauigkeit zum selben numerischen Aufwand ergibt. Die Erklärung ist, dass kleine lokale Anfangsfehler grosse Fehler zur Zeit  $t \approx 1$  produzieren.

**Bemerkung 2.6.11.** Eine Verbesserung des vorigen Algorithmus ist:

Code 2.6.12: bessere adaptive Wahl des Zeitschrittes

```

1 from numpy import *
2 from numpy.linalg import norm
3 import matplotlib.pyplot as plt
4
5 def odeintssctrl(Psilow,p,Psihigh,T,y0,h0,reltol,abstol,hmin):
6     t = [0]; y = [y0]; h = h0 # set initial parameters for the integration
7     while t[-1] < T and h > hmin: # we don't want to exceed T and a too small h
8         yh = Psihigh(h,y0) # higher order integration
9         yH = Psilow(h,y0) # lower order integration
10        est = norm(yH-yh) # estimate for the error. norm returns the norm of a
vector
11        tol = max(reltol*norm(y[-1]),abstol) # we consider both a relative and
an absolute tolerance
12        h = h*max(0.5, min(2, (tol/est)**(1.0/(p+1)))) # if the estimated error
is smaller than the tolerance
#we will take a larger h in the next step (but not larger than
13        2h)
14        if est < tol: #
            y0 = yh; y.append(y0); t.append(t[-1]+min(T-t[-1],h)) #in this case
the estimated error is small so we can accept the (higher order) solution,
otherwise we perform the while-cycle again
15
16        return (array(t),array(y)) # convert to numpy array
17
18 if __name__ == '__main__':
19     """this example demonstrates that a slight reduction in the
20     tolerance can have a large effect on the quality of the solution
21     """
22     # use two different tolerances
23     reltol1=1e-2; abstol1=1e-4
24     reltol2=1e-3; abstol2=1e-5
25     T=2; a=20
26
27     # autonomous ODE  $\dot{y} = \cos(ay)$  and its general solution
28     f = lambda y: (cos(a*y)**2)
29     sol = lambda t: arctan(a*(t-1))/a
30     # Initial state  $y_0$ 
31     y0 = sol(0)
32
33     # Discrete evolution operators
34     Psilow = lambda h,y: y + h*f(y) # Explicit Euler
35     o = 1 # global order of lower-order evolution
36     Psihigh = lambda h,y: y + 0.5*h*( f(y) + f(y+h*f(y)) ) # Explicit trapezoidal
rule
37
38     # Heuristic choice of initial timestep and  $h_{\min}$ 
39     h0 = T/(100.0*(norm(f(y0))+0.1)); hmin = h0/10000.0;
40     # Main adaptive timestepping loop, see Code 2.6.2
41     t1,y1 = odeintssctrl(Psilow,o,Psihigh,T,y0,h0,reltol1,abstol1,hmin)
42     t2,y2 = odeintssctrl(Psilow,o,Psihigh,T,y0,h0,reltol2,abstol2,hmin)
43
44     # Plotting the exact the approximate solutions and rejected timesteps
45     fig = plt.figure()

```

```

48 tp = r_[0:T:T/1000.0]
plt.plot(tp,sol(tp),'g-',linewidth=2, label=r'$y(t)$')
plt.plot(t1,y1,'r.',label='reltol: %1.3f, abstol: %1.5f'%(reltol1, abstol1))
50 plt.plot(t2,y2,'b.',label='reltol: %1.3f, abstol: %1.5f'%(reltol2, abstol2))
plt.title('Adaptive timestepping, a = %d' % a)
52 plt.xlabel(r'$\bf t$',fontsize=14)
plt.ylabel(r'$\bf y$',fontsize=14)
54 plt.legend(loc='upper left')
plt.show()
56 # plt.savefig('../PICTURES/odeintssctrl.eps')
```

Das ist das Prinzip, das hinter der beliebigen Integrationsroutinen in den Bibliotheken steht. Bekannte Methoden sind: Runge-Kutta-Fehlberg 4(5) aus 1969 und Dormand-Prince 5(4) aus 1980. Beide Methoden verwenden Runge-Kutta-Methoden der Ordnungen 4 und 5. Fehlberg verwendet die Methode der Ordnung 5 um den Fehler abzuschätzen, berechnet die Lösung selber mit der Methode der Ordnung 4. Die Methode von Dormand und Prince macht das umgekehrt: die Lösung wird mit der Methode der Ordnung 5; die Butcher Tabellen wurden dafür optimiert. Moderne Codes verwenden die Methode von Dormand und Prince, die bessere Ergebnisse für fast den selben Aufwand liefert. Speziell bei dieser ist, dass die  $b$ 's, die wir in der Berechnung der Lösung der Ordnung 5, identisch mit der letzte Zeile der Matrix  $A$  sind, d.h. diese letzte Berechnung braucht keine zusätzlichen Auswertungen der rechten Seite. Weitere nennenswerte adaptiver Methoden sind: Cash-Karp 5(4), Bogacki-Shampine 5(4), Tsisorous 5(4). Cash-Karp 5(4) versucht die Knoten  $t_0 + hc_i$  so nah wie möglich an äquidistante Knoten zu stellen, ohne dass die Ordnung 5 verloren geht; dadurch wird versucht, früh in der berechnung zu erkennen, ob die Differentialgleichung steif ist. Bogacki-Shampine 5(4) verwendet deutlich mehr Auswertungen der rechte Seite aber optimiert die Butcher-Tabellen so dass die Konstanten in den Fehlerabschätzungen so klein wie möglich sind. Wenn die Auswertung der rechte Seite der Differentialgleichung günstig ist, dann ist Bogacki-Shampine 5(4) besser als Dormand-Prince 5(4). Tsisorous 5(4) wurde um 2011 entwickelt und scheint in den Tests der Dormand-Prince 5(4) überlegen.

**Beispiel 2.6.13.** • Dormand-Prince 5(4) Methode

0	0	0	0	0	0	0	0	
$\frac{1}{5}$	$\frac{1}{5}$	0	0	0	0	0	0	
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	0	0	0	0	0	
$\frac{4}{5}$	0	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$	0	0	0	
$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$	0	0	0	
1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$	0	0	
1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0	
	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0	$\longrightarrow \mathcal{O}(h^5)$
	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$	$\longrightarrow \mathcal{O}(h^4)$

• Runge-Kutta-Fehlberg 4(5) Methode

0	0	0	0	0	0	0	
$\frac{1}{4}$	0	0	0	0	0	0	
$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$	0	0	0	0	
$\frac{12}{13}$	$-\frac{7200}{2197}$	$\frac{7296}{2197}$	$-\frac{212}{729}$	0	0	0	
1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$-\frac{845}{4104}$	0	0	
$\frac{1}{2}$	$-\frac{8}{27}$	2	$-\frac{3544}{2565}$	$\frac{1859}{4104}$	$-\frac{11}{40}$	0	
	$\frac{16}{135}$	0	$\frac{6656}{12825}$	$\frac{28561}{56430}$	$-\frac{9}{50}$	$\frac{2}{55}$	→ $\mathcal{O}(h^5)$
	$\frac{25}{216}$	0	$\frac{1408}{2565}$	$\frac{2197}{4104}$	$-\frac{1}{5}$	0	→ $\mathcal{O}(h^4)$

• Cash-Karp 5(4) Methode

0	0	0	0	0	0	0	
$\frac{1}{5}$	$\frac{1}{5}$	0	0	0	0	0	
$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	0	0	0	0	
$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$	0	0	0	
1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$	0	0	
$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	0	
	$\frac{37}{378}$	0	$\frac{250}{621}$	$\frac{125}{594}$	0	$\frac{512}{1771}$	→ $\mathcal{O}(h^5)$
	$\frac{2825}{27648}$	0	$\frac{18575}{48384}$	$\frac{13525}{55296}$	$\frac{277}{14336}$	$\frac{1}{4}$	→ $\mathcal{O}(h^4)$

**Bemerkung 2.6.14.** Die `scipy`-Bibliothek hat Wrappers `scipy.integrate.ode` auf standard Fortran-Routinen. Die alte Art darauf zuzugreifen ist:  
`r = scipy.integrate.ode(f).set_integrator('dopri5')`. Die neuere und bessere Art ist via `scipy.integrate.solve_ivp`:

```
sol = solve_ivp(odefun, tspan, y0);
```

aufzurufen ist. Die Syntax unserer eigenen `ode45`, die `ode45` auf Runge-Kutta-Fehlberg basiert, ist:

```
t, y = ode45(odefun, tspan, y0);
```

- `odefun`: Handle to a function of type  $(t, y) \leftrightarrow \text{r.h.s. } f(t, y)$
- `tspan`: vector  $(t_0, T)$ , initial and final time for numerical integration
- `y0`: (vector) passing initial state  $y_0 \in \mathbb{R}^d$

Return values:

- `t`: temporal mesh  $\{t_0 < t_1 < t_2 < \dots < t_{N-1} = t_N = T\}$
- `y`: vector  $(y_k)_{k=0}^N$

**Beispiel 2.6.15.** (Numerische Lösung logistischer Differentialgleichung)

Code 2.6.16: Verwendung von `ode45` und `solve_ivp`

```

1 from ode45 import ode45
2 # Interval and initial conditions:
3 tspan = (0,1.5)
4 y0 = [0.5]
5
6 f = lambda t, y: 5.*y*(1.-y)
7
8 t, y = ode45(f, tspan, y0, rtol = 1.e-5, atol = 1.e-5)
9 print('ode45:', t[-1], y[-1], t.shape)
10
11 from scipy.integrate import solve_ivp
12 sol = solve_ivp(f,tspan,y0, rtol = 1.e-5, atol = 1.e-5)
13 print('solve_ivp free:',sol.t[-1], sol.y[:, -1], sol.t.shape)
14
15 import matplotlib.pyplot as plt
16 plt.figure()
17 plt.plot(t,y, label = 'ode45')
18 plt.plot(sol.t, sol.y[0], label = 'sol_ivp')
19 plt.legend()
20 plt.show()

```

**Beispiel 2.6.17.** (Numerische Lösung des Lotka-Volterra'Systems)

Code 2.6.18: Verwendung einiger Standard Löser

```

1 from numpy import *
2
3 import scipy.integrate
4 from numpy import size, zeros
5
6 def odewrapper(f,y0,t0,t,integrator='dopri5',rtol=1.e-5, atol=1.e-5):
7     """integrates the function f with initial values y0, t0
8     returns the value of the integral at times specified in t
9     'dopri5' is the scipy.integrate analog of matlab's ode45
10    """
11
12    # initialize ode class:
13    # constructor takes function to integrate
14    # second function sets initial values (for y and for t)
15    # third function sets integrator (default: 'dopri5' is equivalent to ode45
16    in MATLAB)
17    r = scipy.integrate.ode(f).set_initial_value(y0,t0)
18    r.set_integrator(integrator, atol=atol, rtol=rtol)
19    # do the integration, read values at points in array t.
20    i=1; n=size(t)
21    y = zeros([n,size(y0)])
22    y[0]=y0
23    while r.successful() and i < n:
24        r.integrate(t[i])
25        y[i] = r.y

```

```

24         i += 1
25     return y
26
27 class pred_prey():
28     """calculates rhs of differential equation for a predator-prey system
29
30      $y(t) = [u(t), v(t)]^T \Rightarrow y'(t) = f(y)$  defined by
31      $y_1' = (a_1 - b_1*y_2)*y_1$ 
32      $y_2' = -(a_2 - b_2*y_1)*y_2$ 
33
34     flag = 'f'   => out = f(y)           # default
35     flag = 'df'  => out = df(y)x
36     flag = 'd2f' => out = d2f(y)(x,x)
37
38     defaults values:
39     x = [1;1]   a1 = 3   a2 = 2   b1 = 0.1   b2 = 0.1
40
41     """
42
43     def __init__(self, x=array([1,1]), flag='f', a1=3, a2=2, b1=0.1, b2=0.1):
44         self.x      = x
45         self.flag   = flag
46         self.a1     = a1
47         self.a2     = a2
48         self.b1     = b1
49         self.b2     = b2
50         if flag not in ['f','df','d2f']:
51             print('invalid flag!')
52             self.flag = 'f'
53
54     def __call__(self, t, y, flag=None, x=None):
55         """this function is called when an instance of this class is called as a
56         function:
57
58         f = pred_prey(a2=1)
59         f(4,1) # __call__(f,4,1)
60
61         The system does not depend on the time t explicitly, but the ode solver
62         uses this syntax.
63         ( x is the vector to be multiplied with Df and D2f )
64         """
65
66         # use default values if nothing special specified
67         if flag == None: flag = self.flag
68         #if x is None: x = self.x
69         if not(x is None):
70             pass
71         else:
72             x = self.x
73
74         if flag == 'f':
75             out = array([(self.a1 - self.b1*y[1])*y[0], -(self.a2 -
76 self.b2*y[0])*y[1]])
77         elif flag == 'df':

```

```

        out = [[self.a1-self.b1*y[1], -self.b1*y[0]], [self.b2*y[1],
-self.a2+self.b2*y[0]]]
76         out = dot(out,x)
        elif flag == 'd2f':
78             H1 = [[0, -self.b1], [-self.b1,0]]
             H2 = [[0, self.b2], [self.b2,0]]
80             out = hstack([dot(dot(H1,x),x) , dot(dot(H2,x),x)])
            return out
82
84 if __name__ == '__main__':
    from ode45 import ode45
86     import matplotlib.pyplot as plt
    # Interval and initial conditions:
88     tspan = (0,10)
    y0 = array([100,5])
90
    # instance of pred_prey
92     f = pred_prey()
94
    # integrate via RK-Fehlberg 4(5) of our didactical code
    # standard rtol = 1.e-5, atol = 1.e-5
96     t,y = ode45(f,tspan,y0) # f(t,y)
    print('ode45:',t[-1], y[-1], t.shape)
98     y = y.transpose() # need to transpose for easy plot
100
    # integration using scipy.integrate.ode class (see odewrapper.py)
    # integrate.ode does not return the internal stepping information
102     # we'll use the time stamps given by ode45 in the stepper of the wrapper
    # Dormant-Prince 5(4) is used internally
104     ydp = odewrapper(f,y0,tspan[0],t, rtol = 1.e-5, atol = 1.e-5)
    print('ode on ode45 output times:',t[-1],ydp[-1], t.shape)
106     ydp = ydp.transpose() # need to transpose for easy plot
108
    # integration using scipy.integrate.solve_ivp class (see odewrapper.py)
    # it uses Dormant-Prince 5(4) internally
110     # this is the recommended way of work
    # let us use rtol and atol as standard from ode45
112     from scipy.integrate import solve_ivp
    sol = solve_ivp(f,tspan,y0, rtol = 1.e-5, atol = 1.e-5)
114     print('solve_ivp free:',sol.t[-1], sol.y[:,-1], sol.t.shape)
116
    # if we wish the values of this solution at the time stamps of ode45:
118     sol = solve_ivp(f,tspan,y0, t_eval = t, rtol = 1.e-5, atol = 1.e-5)
    print('solve_ivp on ode45 output times:',sol.t[-1], sol.y[:,-1], sol.t.shape)
120
    # plot
122     plt.figure()
    plt.plot(t,y[0],label = 'rkf 1')
124     plt.plot(t,y[1],label = 'rkf 2')
126
    plt.plot(t,ydp[0],label = 'dp 1')
    plt.plot(t,ydp[1],label = 'dp 2')

```

```

128     plt.plot(sol.t, sol.y[0],label = 'ivp 1')
130     plt.plot(sol.t, sol.y[1],label = 'ivp 2')
132     plt.legend()
134     plt.show()

134     plt.figure()
136     plt.plot(y[0],y[1], label = 'rkf')
138     plt.plot(ydp[0],ydp[1], label = 'dp')
138     plt.plot(sol.y[0],sol.y[1], label = 'ivp')
138     plt.legend()
138     plt.show()

```

**Beispiel 2.6.19.** Naive aber klare Implementierung verschiedener RK-Methoden und einfache Adaptive Methoden

#### Code 2.6.20: Verscheidene RKs und Adaptivität

```

from numpy import double, finfo, array, zeros, dot, size, inf, all, max, min, abs
2 from numpy import linspace, triu, sqrt, allclose, sign
from numpy.linalg import norm
4 from scipy.optimize import fsolve

6 """
8 This file contains different methods to numerically calculate the solutions of
9 ordinary
10 differential equations of the form
11
12     y'(t) = f(t,y)
13     y(t0) = y0
14
15 using Runge-Kutta's Butcher-tableaus.
16 An overview:
17
18     1) A List of useful Butcher-tableaus
19
20     2) Functions for the explicit and implicit RK steps
21
22     3) An integrator to solve the ode with a specific Butcher-tableau
23
24     4) A general adaptive RK solver that takes two Butcher-tableaus of different
25     order
26         as input.
27
28     5) A faster adaptive RK solver that takes a Butcher-tableau with an
29     additional b vector
30         as input.
31
32     6) Two examples of the above methods:
33
34         a) ode45 (Dormand-Prince method, explicit)
35         b) gauss46 (implicit solver using Gauss-Legendre tableaus)

```

```

34  --
    Gian Gentinetta, 2019
36  This file was created as part of my exercise classes in 'Numerical methods for
    physicists'
    at ETH Zurich taught by Dr. Vasile Gradinaru.
38  """"
40  # A List of useful Butcher-Tableaus
42  # Runge-Kutta-Fehlberg's 4(5) Butcher-tableau
    RKF5 = array([[0      , 0      , 0      , 0      , 0      , 0      , 0      , 0
44      ],
                [1/4    , 1/4    , 0      , 0      , 0      , 0      , 0      , 0
46      ],
                [3/8    , 3/32   , 9/32   , 0      , 0      , 0      , 0      , 0
48      ],
                [12/13, 1932/2197, -7200/2197, 7296/2197 , 0      , 0      , 0      , 0
                    ],
                [1      , 439/216 , -8      , 3680/513 , -845/4104 , 0      , 0      , 0
50      ],
                [1/2    , -8/27  , 2      , -3544/2565, 1859/4104 , -11/40, 0      , 0
52      ],
                [0      , 16/135 , 0      , 6656/12825, 28561/56430, -9/50 , 0      , 0
                    ]
                    ])
    RKF4 = array([[0      , 0      , 0      , 0      , 0      , 0      , 0      , 0
54      ],
                [1/4    , 1/4    , 0      , 0      , 0      , 0      , 0      , 0
56      ],
                [3/8    , 3/32   , 9/32   , 0      , 0      , 0      , 0      , 0
58      ],
                [12/13, 1932/2197, -7200/2197, 7296/2197 , 0      , 0      , 0      , 0
                    ],
                [1      , 439/216 , -8      , 3680/513 , -845/4104 , 0      , 0      , 0
60      ],
                [1/2    , -8/27  , 2      , -3544/2565, 1859/4104 , -11/40, 0      , 0
62      ],
                [0      , 25/216 , 0      , 1408/2565 , 2197/4104 , -1/5  , 0      , 0
                    ]
                    ])
    RKF_b_low = array([25/216, 0, 1408/2565, 2197/4104, -1/5, 0])
64  # Dormand-Prince
66  RKDP = array([[0      , 0      , 0      , 0      , 0      , 0      , 0      , 0
                    ],
                [0      , 0],
                [1/5    , 1/5    , 0      , 0      , 0      , 0      , 0      , 0
64      ],
                [0      , 0],
                [3/10, 3/40    , 9/40    , 0      , 0      , 0      , 0      , 0
66      ],
                [0      , 0],
                [4/5    , 44/45   , -56/15  , 32/9    , 0      , 0      , 0      , 0
                    ],
                [0      , 0],
                [8/9    , 19372/6561, -25360/2187, 64448/6561, -212/729, 0      , 0      , 0
                    ]
                    ])

```

```

0    , 0],
68      [1 , 9017/3168 , -355/33 , 46732/5247, 49/176 , -5103/18656,
0    , 0],
      [1 , 35/384 , 0 , 500/1113 , 125/192 , -2187/6784 ,
11/84, 0 ],
70      [0 , 35/384 , 0 , 500/1113 , 125/192 , -2187/6784 ,
11/84, 0 ]])

72 RKDP_b_low = array([5179/57600, 0, 7571/16695, 393/640, -92097/339200, 187/2100,
1/40])

74 # Classic Runge-Kutta (RK4)
RK4 = array([[0 , 0 , 0 , 0 , 0 ],
76          [1/2, 1/2 , 0 , 0 , 0 ],
          [1/2, 0 , 1/2, 0 , 0 ],
78          [1 , 0 , 0 , 1 , 0 ],
          [0 , 1/6 , 2/6, 2/6, 1/6]])

80 # Kutta-3/8 rule
82 Kutta38 = array([[0 , 0 , 0 , 0 , 0 ],
          [1/3, 1/3 , 0 , 0 , 0 ],
84          [2/3, -1/3, 1 , 0 , 0 ],
          [1 , 1 , -1 , 1 , 0 ],
86          [0 , 1/8 , 3/8, 3/8, 1/8]])

88 # Gauss-Legendre methods (implicit)
Gauss4 = array([[1/2 - sqrt(3)/6, 1/4 , 1/4 - sqrt(3)/6],
90          [1/2 + sqrt(3)/6, 1/4 + sqrt(3)/6, 1/4 , 1/4 ],
          [0 , 1/2 , 1/2 , 1/2 ]])

92 Gauss4_b_low = array([1/2 + 1/2*sqrt(3), 1/2 - 1/2*sqrt(3)])

94 Gauss6 = array([
96          [1/2 - sqrt(15)/10, 5/36 , 2/9 - sqrt(15)/15, 5/36 -
sqrt(15)/30],
          [1/2 , 5/36 + sqrt(15)/24, 2/9 , 5/36 -
98          sqrt(15)/24],
          [1/2 + sqrt(15)/10, 5/36 + sqrt(15)/30, 2/9 + sqrt(15)/15, 5/36
],
          [0 , 5/18 , 4/9 , 5/18
]])

100 #
-----
#
102 # Runge-Kutta solvers
def nicer_fsolve(F, initial_guess):
104     """Wrapper for `scipy.optimize.fsolve`.

106     This wrapper is used to solve non-linear equations of multidimensional arrays.
    """
108     shape = initial_guess.shape

110     initial_guess = initial_guess.reshape((-1,))

```

```

112     result = fsolve(lambda x: F(x.reshape(shape)).reshape((-1)), initial_guess)
113
114     return result.reshape(shape)
115
116 def nicer_norm(y):
117     """
118     Takes the infinity norm for an array y, where y does not have to be 1D or 2D.
119     """
120     return norm(y.reshape(-1),inf)
121
122 def exp_RKstep(B,f,y0,t0,h,ret_k=False):
123     """
124     Calculates y(t0 + h) using a given an explicit Butcher-tableau B and its
125     corresponding
126     Runge-Kutta method.
127
128     -----
129     Input:
130
131     B: Butcher-tableau. For nodes c, weights b and the matrix A this should
132     look like
133
134         B = c | A
135             -----
136             0 | b^T
137
138     , where  $A_{ij} = 0$  for  $i \leq j$  (explicit method).
139
140     f: Right-hand-side of the differential equation  $y'(t) = f(t,y)$ 
141
142     y0: Starting vector at time  $t = t_0$ 
143
144     t0: Initial time
145
146     h: Step size
147
148     ret_k: Set to True to return k
149
150     -----
151     Returns:
152
153     y: The result  $y(t_0 + h)$ 
154     (k: Function calls)
155     """
156     y0 = array(y0)
157
158     shape = y0.shape # Reshaping needs to be done in case y0 is 2D or larger
159     y0 = y0.reshape(-1) # as np.dot would misbehave otherwise.
160
161     c = B[:-1,0] # Nodes
162     b = B[-1,1:] # Weights
163     A = B[:-1,1:] # Matrix A

```

```

162     s = len(c)          # Number of Nodes
163     d = len(y0)       # Dimension of y
164
165     k = zeros((s,d))
166     for i in range(s):
167         k[i,:] = f(t0 + h*c[i], (y0 +
168             h*dot(A[i,:],k)).reshape(shape)).reshape(-1)
169         # Here we shape back so that rhs receives the original shape.
170
171     y = y0 + h*dot(b,k)
172
173     # Return k if specified in the argument.
174     if ret_k:
175         return y.reshape(shape), k
176
177     return y.reshape(shape)
178 def imp_RKstep(B,f,y0,t0,h,ret_k=False):
179     """
180     Calculates y(t0 + h) using a given an implicit Butcher-tableau B and its
181     corresponding
182     Runge-Kutta method.
183
184     -----
185     Input:
186
187     B: Butcher-tableau. For nodes c, weights b and the matrix A this should
188     look like
189
190         B = c | A
191             -----
192             0 | b^T
193
194     f: Right-hand-side of the differential equation y'(t) = f(t,y)
195
196     y0: Starting vector at time t = t0
197
198     t0: Initial time
199
200     h: Step size
201
202     -----
203     Returns:
204
205     y: The result y(t0 + h)
206     """
207     y0 = array(y0)
208
209     shape = y0.shape
210     y0 = y0.reshape(-1)

```

```

210     c = B[:-1,0]      # Nodes
211     b = B[-1,1:]     # Weights
212     A = B[:-1,1:]    # Matrix A

214     s = len(c)       # Number of Nodes
215     d = len(y0)      # Dimension of y

216
217     def F(x):
218         k_x = zeros((s,d))
219         for i in range(s):
220             k_x[i,:] = f(t0 + h*c[i], (y0 +
221 h*dot(A[i,:],x)).reshape(shape)).reshape(-1)
222         return k_x

224     guess = zeros((s,d))
225     k = nicer_fsolve(lambda x: F(x) - x,guess) # Solving the non-linear system
of equations

226     y = y0 + h*dot(b,k)

228     # Return k if specified in the argument.
229     if ret_k:
230         return y.reshape(shape), k

232     return y.reshape(shape)

234 def rk(B,f,tspan,y0,N):
235     """
236     Calculates the solution of an ordinary differential
237     equation with right-hand-side f
238
239      $y'(t) = f(t,y)$ 
240      $y(t_0) = y_0$ 
241
242     using a the Runge-Kutta method encoded in the Butcher-tableau B
-----
244     Input:
245
246     B: Butcher-tableau of the two Runge-Kutta method used to calculate the
247     solution. For nodes c, weights b and the matrix A this should look like
248
249     B = c | A
250         -----
251         0 | b^T
252
253     , where  $A_{ij} = 0$  for  $i \leq j$  (explicit method).
254
255     f: Right-hand-side of the differential equation  $y'(t) = f(t,y)$ 
256
257     y0: Initial value  $y(t_0) = y_0$ 
258
259     tspan: Tuple (t_0,t_end) of start and end times

```

```

262
-----
264     Returns: t, y
266
266     t: The times at which y(t) is calculated
266     y: The calculated solutions y(t)
268
268     """
270
270     y0 = array(y0,ndmin=1)
272
272     if allclose(triu(B[:-1,1:]),zeros(B[:-1,1:].shape)): # Test if the method is
explicit
274         method = exp_RKstep
274     else:
276         method = imp_RKstep
278
278     # Initialization
278     t0,tend=tspan
280     t, h = linspace(t0,tend,N+1,retstep=True)
280     y = zeros((N+1,) + y0.shape)
282     y[0,...] = y0
284
284     # Integrating the solution
284     for i in range(N):
286         y[i+1,...] = method(B,f,y[i,...],t[i],h)
288
288     return t,y
290 #
-----
290 #
290 # Adaptive Runge-Kutta solvers
292
292 def rk_adapt(B_high,B_low,f,tspan,y0,reltol=1e-6,abstol=1e-6,initialstep=None,
294             maxstep=None):
294     """
296     Adaptive Runge-Kutta method. Calculates the solution of an ordinary
differential
296     equation with right-hand-side f:
298
298          $y'(t) = f(t,y)$ 
300          $y(t_0) = y_0$ 
302
-----
304     Input:
306
306     B_high, B_low: Butcher-tableaus of the two Runge-Kutta methods used to
calculate the
306     solution. For nodes c, weights b and the matrix A this should
look like

```

```

308         B = c | A
310             -----
311             0 | b^T
312
313         , where Aij = 0 for i ≤ j (explicit method).
314
315         Examples are Bhigh = RKF5, Blow = RKF4
316
317         f: Right-hand-side of the differential equation y'(t) = f(t,y)
318
319         y0: Initial value y(t0) = y0
320
321         tspan: Tuple (t0,tend) of start and end times
322
-----
324     Optional Input:
325
326     reltol: Relative tolerance
327
328     abstol: Absolute tolerance
329
330     initialstep: Set to a float if a specific initial step size is wished
331
332     maxstep: Set to a float if the adaptive method should not be larger than
333     some maxstep
334
-----
335     Returns: t, y
336
337     t: The times at which y(t) is calculated
338     y: The calculated solutions y(t)
339
340     """
341     # Check if methods are implicit or explicit:
342
343     if allclose(triu(B_high[:-1,1:]),zeros(B_high[:-1,1:].shape)):
344         method_high = exp_RKstep
345     else:
346         method_high = imp_RKstep
347         print('implicit')
348
349     if allclose(triu(B_low[:-1,1:]),zeros(B_low[:-1,1:].shape)):
350         method_low = exp_RKstep
351     else:
352         method_low = imp_RKstep
353
354     # Ensure tspan and y0 are vectors:
355
356     tspan = array(tspan, ndmin=1)
357     y0 = array(y0,ndmin = 1)

```

```

358
360 # Setting the initial step, if not specified in the arguments.
361 if initialstep == None:
362     initialstep = double(tspan[-1] - tspan[0]) / 100
364
365 # Setting the maximal step, if not specified in the arguments.
366 if maxstep == None:
367     maxstep = double(tspan[-1] - tspan[0]) / 10
368
369 # Initialization of the adaptive solver
370
371 t0 = tspan[0] # start time
372 tend = tspan[-1] # end time
374
375 dir = sign(tend) # Determine if integrating forwards or backwards in time
376
377 h = initialstep # step size
378
379 epsilon = finfo(double).eps # The smallest possible double in Python
380
381 hmin = epsilon * double(tend - t0) # The smallest step size used in this
algorithm
382
383 t = [t0] # This will be the output list for the times
384 y = [y0] # This will be the output list for y(t).
386
387 p = 0.2 # This is the exponent used to increase/decrease the stepsize
388
389 # This is the main loop. It will stop if t = tend has been reached,
# or if the step size cannot be reduced any further.
390
391 while dir*t0 < dir*tend and dir*h >= dir*hmin:
392
393     # If t0 + h surpasses the endpoint, we want to hit tend instead
394     if t0 + h > dir*tend:
395         h = tend - dir*t0
396
397     # Estimating y(t0 + h) with our two methods
398     y_low = method_low(B_low,f,y0,t0,h)
399     y_high = method_high(B_high,f,y0,t0,h)
400
401     # Comparing the two results
402     err = nicer_norm(y_high - y_low)
404
405     # If the error is within the tolerance, we set our new values for t0 and
y0
406     # and append them to our return lists.
407     tol = max([reltol * max([nicer_norm(y0),1.]), abstol])
408
409     if err <= tol:
410         t0 = t0 + h
411         t.append(t0)
412         y0 = y_high

```

```

410         y.append(y0)
412         # We are dividing by the error, so we have to ensure that it is not zero:
413         if err == 0:
414             err = tol * (0.4 ** (1. / p)) # This doubles the step size for the
next step
416         # Now we update the step size for the next iteration.
417         if dir == 1:
418             h = min([maxstep,min(0.8 * h * (tol/err) ** p)])
419         else:
420             h = max([maxstep,max(0.8 * h * (tol/err) ** p)])
422         return array(t), array(y)
424     #
-----
425     #
426     # Adaptive Runge-Kutta Method, where only b differs in the two Butcher-Tableaux
427     def
fast_rk_adapt(B_high,b_low,f,tspan,y0,reltol=1e-6,abstol=1e-6,initialstep=None,
maxstep=None):
428         """
Adaptive Runge-Kutta method. Calculates the solution of an ordinary
differential
430         equation with right-hand-side f:
431
432         y'(t) = f(t,y)
433         y(t0) = y0
434
-----
435
436         Input:
437
438         B_high:      Butcher-tableau of the Runge-Kutta method used to calculate
the
solution. For nodes c, weights b and the matrix A this should
look like
440
B = c | A
441         -----
442             0 | b^T
443
, where Aij = 0 for i ≤ j (explicit method).
444
445         b_low: Additional b vector to provide a second (less accurate) approximation
446
447         f: Right-hand-side of the differential equation y'(t) = f(t,y)
448
449         y0: Initial value y(t0) = y0
450
451         tspan: Tuple (t_0,t_end) of start and end times
452
453
454

```

```
456 Optional Input:
458
460 reltol: Relative tolerance
462
464 abstol: Absolute tolerance
466
468 initialstep: Set to a float if a specific initial step size is wished
470
472 maxstep: Set to a float if the adaptive method should not be larger than
474 some maxstep
476
478 Returns: t, y
480
482 t: The times at which y(t) is calculated
484 y: The calculated solutions y(t)
486
488 """
490 # Check if methods are implicit or explicit:
492
494 if allclose(triu(B_high[:-1,1:]),zeros(B_high[:-1,1:].shape)):
496     method_high = exp_RKstep
498 else:
500     method_high = imp_RKstep
502     print('implicit')
504
506 # Ensure tspan and y0 are vectors:
508
510 tspan = array(tspan, ndmin=1)
512 y0 = array(y0, ndmin = 1)
514
516 # Setting the initial step, if not specified in the arguments.
518 if initialstep == None:
520     initialstep = double(tspan[-1] - tspan[0]) / 100
522
524 # Setting the maximal step, if not specified in the arguments.
526 if maxstep == None:
528     maxstep = double(tspan[-1] - tspan[0]) / 10
530
532 # Initialization of the adaptive solver
534
536 t0 = tspan[0] # start time
538 tend = tspan[-1] # end time
540
542 dir = sign(tend) # Determine if integrating forwards or backwards in time
544
546 h = initialstep # step size
548
550 epsilon = finfo(double).eps # The smallest possible double in Python
552
```

```

    hmin = epsilon * double(tend - t0) # The smallest step size used in this
algorithm
508
    t = [t0] # This will be the output list for the times
510
    y = [y0] # This will be the output list for y(t).

512
    p = 0.2 # This is the exponent used to increase/decrease the stepsize

514
    # This is the main loop. It will stop if t = tend has been reached,
    # or if the step size cannot be reduced any further.

516
    while dir*t0 < dir*tend and dir*h >= dir*hmin:
518
        # If t0 + h surpasses the endpoint, we want to hit tend instead
520
        if t0 + h > dir*tend:
            h = tend - dir*t0
522
        # Estimating y(t0 + h) with our two methods
524
        y_high, k = method_high(B_high,f,y0,t0,h,ret_k=True)
        # Repeat last part of the RK-step for the second b vector
526
        y_low = y0 + h*dot(b_low,k).reshape(y0.shape)

528
        # Comparing the two results
        err = nicer_norm(y_high - y_low)
530
        # If the error is within the tolerance, we set our new values for t0 and
y0
532
        # and append them to our return lists.
        tol = max([reltol * max([nicer_norm(y0),1.]), abstol])
534
        if err <= tol:
536
            t0 = t0 + h
            t.append(t0)
538
            y0 = y_high
            y.append(y0)
540
        # We are dividing by the error, so we have to ensure that it is not zero:
542
        if err == 0:
            err = tol * (0.4 ** (1. / p)) # This doubles the step size for the
next step
544
        # Now we update the step size for the next iteration.
546
        if dir == 1:
            h = min([maxstep,min(0.8 * h * (tol/err) ** p)])
548
        else:
            h = max([maxstep,max(0.8 * h * (tol/err) ** p)])
550
        return array(t), array(y)
552
#
-----
#
554 # Examples:

```

```
556 def ode45(f,tspan,y0,reltol=1e-6,abstol=1e-6,initialstep=None,maxstep=None):
558     """
559     Calculates the solution of an ordinary differential equation with
560     right-hand-side f
561
562      $y'(t) = f(t,y)$ 
563      $y(t_0) = y_0$ 
564
565     using Runge-Kutta Dormant-Prince 5(4) or Fehlberg's 4(5) method.
566
567     -----
568     Input:
569
570     f: Right-hand-side of the differential equation  $y'(t) = f(t,y)$ 
571
572     y0: Initial value  $y(t_0) = y_0$ 
573
574     tspan: Tuple (t_0,t_end) of start and end times
575
576     -----
577     Optional Input:
578
579     reltol: Relative tolerance
580
581     abstol: Absolute tolerance
582
583     initialstep: Set to a float if a specific initial step size is wished
584
585     maxstep: Set to a float if the adaptive method should not be larger than
586     some maxstep
587
588     -----
589     Returns: t, y
590
591     t: The times at which  $y(t)$  is calculated
592     y: The calculated solutions  $y(t)$ 
593
594     """
595
596     return fast_rk_adapt(RKDP,RKDP_b_low,f,tspan,y0,reltol=reltol,abstol=abstol,
597                        initialstep=initialstep, maxstep=maxstep)
598
599 def gauss46(f,tspan,y0,reltol=1e-6,abstol=1e-6,initialstep=None,maxstep=None):
600     """
601     Calculates the solution of an ordinary differential equation with
602     right-hand-side f
603
604      $y'(t) = f(t,y)$ 
605      $y(t_0) = y_0$ 
```

```

604     using an adaptive Runge-Kutta method with the Gauss-Legendre
Butcher-tableaus of order
606     6 and 4. As an implicit method this should be used to find solutions of stiff
equations.
608
-----
610     Input:
612
614     f: Right-hand-side of the differential equation  $y'(t) = f(t,y)$ 
616     y0: Initial value  $y(t_0) = y_0$ 
618     tspan: Tuple (t_0,t_end) of start and end times
620
-----
622     Optional Input:
624
626     reltol: Relative tolerance
628     abstol: Absolute tolerance
630     initialstep: Set to a float if a specific initial step size is wished
632     maxstep: Set to a float if the adaptive method should not be larger than
some maxstep
634
-----
636     Returns: t, y
638
640     t: The times at which  $y(t)$  is calculated
642     y: The calculated solutions  $y(t)$ 
644
646     """
648     """
650     return rk_adapt(Gauss6,Gauss4,f,tspan,y0,reltol=reltol,abstol=abstol,
initialstep=initialstep, maxstep=maxstep)
652
654     """
656     Changelog:
658     29.03.2020: Added reshapes to support more cases of y0 shapes. Added
nicer_norm for similar purposes.
660     31.03.2020: Added faster variant for the adaptive method, added butcher
tableaus for Dormand-Prince.
662     """

```

**Beispiel 2.6.21.** (Blow-up)

Sei das Anfangswertproblem

$$\dot{y} = y^2, \quad y(0) = y_0 > 0.$$

mit der Lösung  $y(t) = \frac{y_0}{1 - y_0 t}$ ,  $t < 1/y_0$

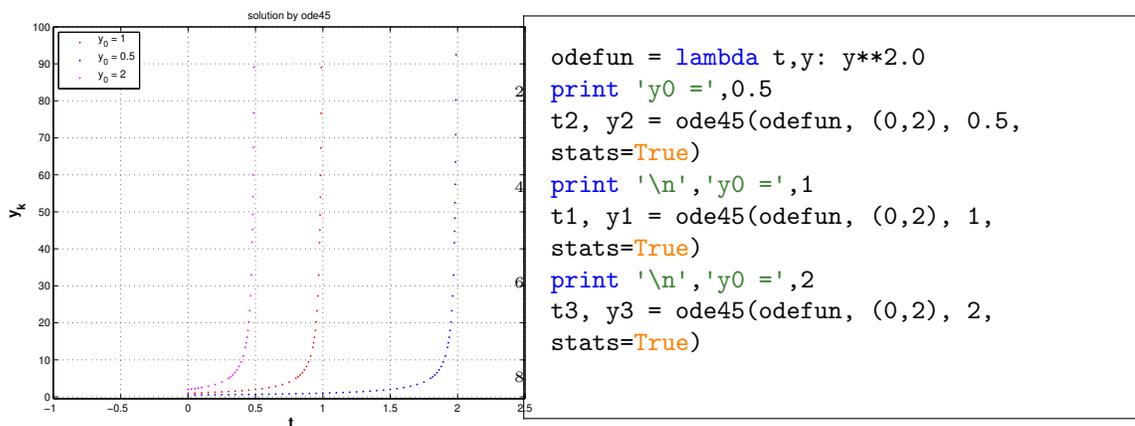
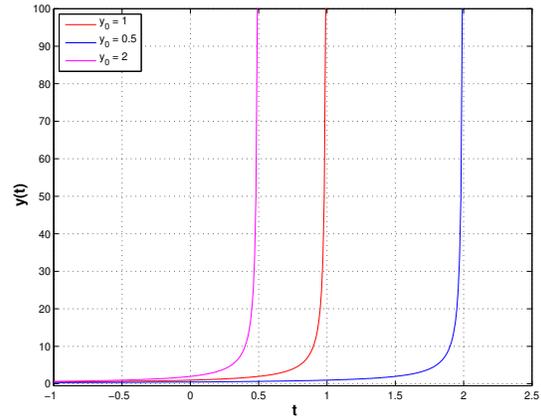


Abb. 2.6.22. Blow-up

Warnungen:

```

y0 = 0.5
error: OdePkg:InvalidArgument
Solving has not been successful.
The iterative integration loop exited at time t = 2.000000 before endpoint at tend = 2.000000 was reached.
This may happen if the stepsize grows smaller than defined in vminstepsize.
Try to reduce the value of "initialstep" and/or "maxstep" with the command "odeset".

Number of successful steps: 151
Number of failed attempts: 74
Number of function calls: 1344

y0 = 1
error: OdePkg:InvalidArgument
Solving has not been successful.
The iterative integration loop exited at time t = 1.000000 before endpoint at tend = 2.000000 was reached.
This may happen if the stepsize grows smaller than defined in vminstepsize.
Try to reduce the value of "initialstep" and/or "maxstep" with the command "odeset".

Number of successful steps: 146
Number of failed attempts: 74
Number of function calls: 1314

y0 = 2
error: OdePkg:InvalidArgument
Solving has not been successful.
The iterative integration loop exited at time t = 0.500000 before endpoint at tend = 2.000000 was reached.
This may happen if the stepsize grows smaller than defined in vminstepsize.
Try to reduce the value of "initialstep" and/or "maxstep" with the command "odeset".

Number of successful steps: 144
Number of failed attempts: 72
Number of function calls: 1290

```

Wir beobachten: `ode45` reduziert den Zeitschritt, wenn sich die Lösung der Singularität nähert und warnt den Benutzer. Was würde das explizite Euler-Verfahren auf einem festen Zeitgitter ohne Kenntnis der Singularität liefern?

**Beispiel 2.6.23.** (Schrittweitensteuerung für ein mechanisches Problem) Wir betrachten die Bewegung einer Punktmasse in einem konservativen Feld; das Ziel ist es, ihre Trajektorie  $t \mapsto \mathbf{y}(t) \in \mathbb{R}^2$  zu berechnen.

$$\text{Newton :} \quad \dot{\mathbf{y}} = F(\mathbf{y}) := -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2}. \quad (2.6.40)$$

Das äquivalente System von Differentialgleichungen erster Ordnung (mit  $\mathbf{v} := \dot{\mathbf{y}}$ ) lautet:

$$\begin{bmatrix} \dot{\mathbf{y}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ -\frac{2\mathbf{y}}{\|\mathbf{y}\|_2^2} \end{bmatrix}. \quad (2.6.41)$$

Im Experiment verwenden wir als Anfangswert

$$\mathbf{y}(0) := \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad \mathbf{v}(0) := \begin{bmatrix} 0.1 \\ -0.1 \end{bmatrix}.$$

Hier ein Teil des Simulationscodes:

```

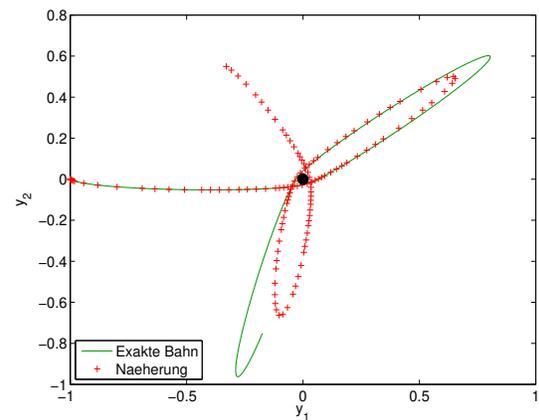
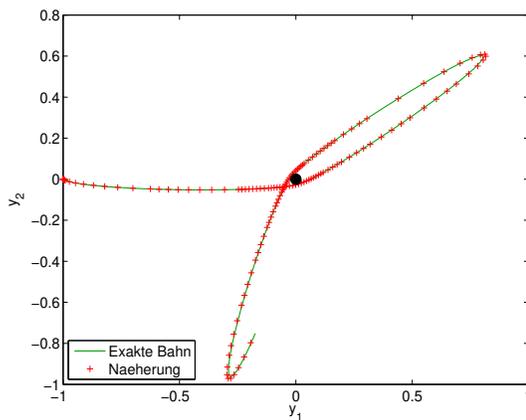
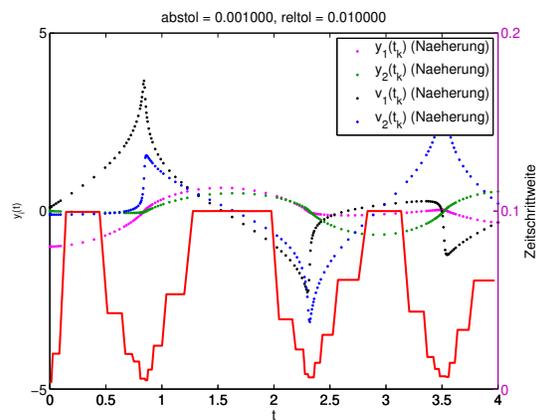
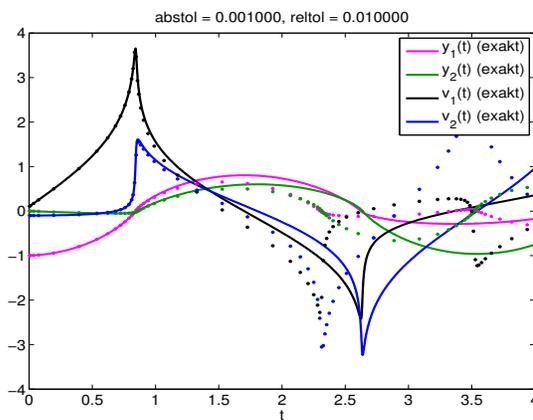
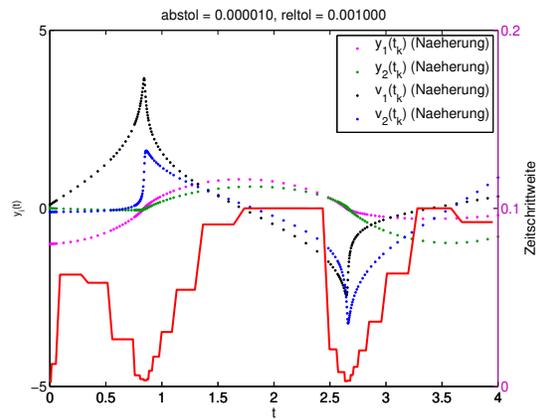
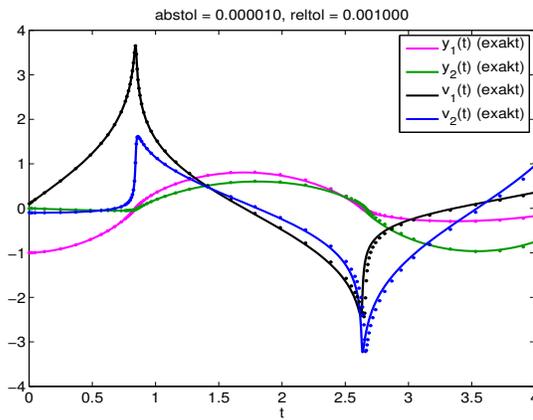
1  from numpy import array, linspace, sqrt, hstack, dot
2  from matplotlib.pyplot import figure, plot, show, xlabel, ylabel, grid, legend,
3  ylim
4
5  f_newton = lambda t,z: hstack((z[2:], -2*z[:2]/dot(z[:2],z[:2])) )
6  y0 = array([-1.0,0.0,0.1,-0.1])
7
8  #we consider as "exact" solution one solution obtained with extremely low
9  tolerances
10 tex, yex = ode45(f_newton, (0,4), y0,reltol=1.0e-6, abstol=1.0e-12,
11 initialstep=0.01,stats=True)
12
13 t1,y1 = ode45(f_newton, (0,4), y0,reltol=1.0e-3, abstol=1.0e-6,
14 initialstep=0.01,stats=True)
15 figure()
16 plot(tex,yex[:,0], "-",color="b",markersize=5,label="$y_1$ exact")
17 plot(tex,yex[:,1], "-",color="g",markersize=5,label="$y_2$ exact")
18 plot(tex,yex[:,2], "-",color="r",markersize=5,label="$v_1$ exact")
19 plot(tex,yex[:,3], "-",color="y",markersize=5,label="$v_2$ exact")
20 plot(t1,y1[:,0], "o",color="b",markersize=5,label="$y_1$ approx")
21 plot(t1,y1[:,1], "o",color="g",markersize=5,label="$y_2$ approx")
22 plot(t1,y1[:,2], "o",color="r",markersize=5,label="$v_1$ approx")
23 plot(t1,y1[:,3], "o",color="y",markersize=5,label="$v_2$ approx")
24 grid(True); legend(loc="best"); title("reltol=0.001, abstol=0.000001")
25
26 figure()
27
28 t2,y2 = ode45(f_newton, (0,4), y0,reltol=1.0e-2, abstol=1.0e-3,
29 initialstep=0.01,stats=True)
30 figure()
31 plot(tex,yex[:,0], "-",color="b",markersize=5,label="$y_1$ exact")
32 plot(tex,yex[:,1], "-",color="g",markersize=5,label="$y_2$ exact")

```

```

29 plot(tex,yex[:,2],"-",color="r",markersize=5,label="$v_1$ exact")
plot(tex,yex[:,3],"-",color="y",markersize=5,label="$v_2$ exact")
31 plot(t2,y2[:,0],"o",color="b",markersize=5,label="$y_1$ approx")
plot(t2,y2[:,1],"o",color="g",markersize=5,label="$y_2$ approx")
33 plot(t2,y2[:,2],"o",color="r",markersize=5,label="$v_1$ approx")
plot(t2,y2[:,3],"o",color="y",markersize=5,label="$v_2$ approx")
35 grid(True); legend(loc="best"); title("reltol=0.01, abstol=0.001")
show()

```



**Abb. 2.6.24.** Exakte und numerische Lösungen mittels `ode45` bei unterschiedlicher Wahl der Toleranzen.

Wir beobachten: schnelle Änderungen in den Komponenten der Lösung werden durch kleine Zeitschritte aufgelöst, aber die numerische Lösung wird sehr schlecht, wenn wir die Toleranz nur sehr wenig erhöhen. Daraus lernen wir ein wichtiges Prinzip: die Schätzung des lokalen Fehlers kann nicht die Genauigkeit der globalen Lösung voraussagen!

## 2.7 Partitionierte Runge-Kutta Verfahren

Wir betrachten hier partitionierte Systeme der Form

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}, \mathbf{z}), \quad \dot{\mathbf{z}} = \mathbf{g}(\mathbf{y}, \mathbf{z}).$$

Wir nehmen zwei verschiedene RK-ESV und wir behandeln  $\mathbf{y}$  mittels  $(a_{ij}, b_i)$  und  $\mathbf{z}$  mittels  $(\hat{a}_{ij}, \hat{b}_i)$ ; dieser Algorithmus heisst *partitioniertes Runge-Kutta-Verfahren* (PRK):

$$\begin{aligned} \mathbf{k}_i &= \mathbf{f} \left( \mathbf{y}_0 + h \sum_{j=1}^s a_{ij} \mathbf{k}_j, \mathbf{z}_0 + h \sum_{j=1}^s \hat{a}_{ij} \boldsymbol{\ell}_j \right), \\ \boldsymbol{\ell}_i &= \mathbf{g} \left( \mathbf{y}_0 + h \sum_{j=1}^s a_{ij} \mathbf{k}_j, \mathbf{z}_0 + h \sum_{j=1}^s \hat{a}_{ij} \boldsymbol{\ell}_j \right), \\ \mathbf{y}_1 &= \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i, \\ \mathbf{z}_1 &= \mathbf{z}_0 + h \sum_{i=1}^s \hat{b}_i \boldsymbol{\ell}_i. \end{aligned}$$

Das symplektische Euler-Verfahren (siehe Gleichung (2.4.29)) ist ein PRK mit Euler  $b_1 = 1, a_{11} = 1$ , kombiniert mit dem expliziten Euler  $\hat{b}_1 = 1, \hat{a}_{11} = 0$ . Auch das Störmer-Verlet-Verfahren lässt sich als PRK schreiben:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \begin{array}{c|cc} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$$

Eine Verallgemeinerung vom Störmer-Verlet ist das 3-stufige Lobatto IIIA-IIIIB Paar (Ordnung  $h^4$ ):

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{5}{24} & \frac{1}{3} & -\frac{1}{24} \\ 1 & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array} \qquad \begin{array}{c|ccc} 0 & \frac{1}{6} & -\frac{1}{6} & 0 \\ \frac{1}{2} & \frac{1}{6} & \frac{1}{3} & 0 \\ 1 & \frac{1}{6} & \frac{5}{6} & 0 \\ \hline & \frac{1}{6} & \frac{2}{3} & \frac{1}{6} \end{array}$$

Differenzialgleichungen zweiter Ordnung:

$$\ddot{\mathbf{y}} = \mathbf{g}(t, \mathbf{y}, \dot{\mathbf{y}}) :$$

lassen sich als System erster Ordnung mit  $\mathbf{z} = \dot{\mathbf{y}}$  schreiben:

$$\dot{\mathbf{y}} = \mathbf{z} \quad \dot{\mathbf{z}} = \mathbf{g}(t, \mathbf{y}, \mathbf{z})$$

und ein PRK dafür heisst Runge-Kutta-Nyström-Verfahren (RKN):

$$\begin{aligned} \ell_i &= g(t_0 + c_i h, \mathbf{y}_0 + c_i h \dot{\mathbf{y}}_0 + h^2 \sum_{j=1}^s \bar{a}_{ij} \ell_j, \dot{\mathbf{y}}_0 + h \sum_{j=1}^s \hat{a}_{ij} \ell_j) \\ \mathbf{y}_1 &= \mathbf{y}_0 + h \dot{\mathbf{y}}_0 + h^2 \sum_{i=1}^s \bar{b}_i \ell_i \\ \dot{\mathbf{y}}_1 &= \dot{\mathbf{y}}_0 + h \sum_{i=1}^s \hat{b}_i \ell_i, \end{aligned}$$

mit  $\bar{a}_{ij} = \sum_{k=1}^s a_{ik} \hat{a}_{kj}$  und  $\bar{b}_i = \sum_{k=1}^s b_k \hat{a}_{ki}$ . Wenn die rechte Seite  $\mathbf{g}$  nicht explizit von  $\dot{\mathbf{y}}$  abhängt, dann braucht man  $\hat{a}_{ij}$  nicht. Ein RKN hat Ordnung  $p$ , wenn die beiden lokalen Schritte  $\mathbf{y}_1 - \mathbf{y}(t_0 + h)$  und  $\dot{\mathbf{y}}_1 - \dot{\mathbf{y}}(t_0 + h)$  der Ordnung  $p + 1$  sind.

**Bemerkung 2.7.1.** Die partitionierten Runge-Kutta Verfahren kann man auch als Splitting-Verfahren  $\dot{\mathbf{u}} = \mathbf{f}_a \mathbf{u} + \mathbf{f}_b \mathbf{u}$  verstehen:

$$\mathbf{u} = \begin{bmatrix} \mathbf{y} \\ \mathbf{z} \end{bmatrix}, \quad \mathbf{f}_a = \begin{bmatrix} \mathbf{f}(\mathbf{u}) \\ 0 \end{bmatrix}, \quad \mathbf{f}_b = \begin{bmatrix} 0 \\ \mathbf{g}(\mathbf{u}) \end{bmatrix}.$$

Da Runge-Kutta-Nystrom-Verfahren für spezielle Art der rechten Seiten angewendet werden, sind diese Splitting-Verfahren, wo die lokalen Evolutionsoperatoren qualitativ unterschiedlich sind. Im Lichte dieser Bemerkung stellen wir fest: die Methode PRK-S6 im Code 2.4.9 ist eine symplektische PRK-Methode der Ordnung 4; BM42 und BM63 sind symplektische RKN-Methoden der Ordnungen 4 und 6. Alle drei Verfahren wurden so optimiert, dass sie die spezielle Eigenschaft haben, dass man relativ grosse Zeitschrittweiten verwenden kann.

## 2.8 Magnus-Integratoren für lineare Gleichungen

Wir betrachten hier lineare zeitabhängige Differentialgleichungen der Form

$$\dot{\mathbf{y}}(t) = \mathbf{A}(t)\mathbf{y}(t)$$

mit dem Startwert  $\mathbf{y}(t_0) = \mathbf{y}_0$ . Die Magnus-Entwicklung stammt aus der Quantenmechanik kommt und ergibt approximative Lösungen, die bestimmte physikalische

Eigenschaften der exakten Lösung besitzen. Wenn gewisse Voraussetzungen erfüllt sind, kann man annehmen, dass sich die Lösung schreiben lässt als

$$\mathbf{y}(t) = e^{\mathbf{\Omega}(t,t_0)} \mathbf{y}_0$$

mit  $\mathbf{\Omega} = \sum_{k=1}^{\infty} \mathbf{\Omega}_k$ . Wir notieren den Kommutator von  $\mathbf{A}$  und  $\mathbf{B}$  mit  $[\mathbf{A}, \mathbf{B}] = \mathbf{AB} - \mathbf{BA}$  und schreiben die ersten Terme als (Baker-Campbell-Hausdorff Formel):

$$\begin{aligned} \mathbf{\Omega}_1 &= \int_{t_0}^t \mathbf{A}(\tau_1) d\tau_1, \\ \mathbf{\Omega}_2 &= \frac{1}{2} \int_{t_0}^t \int_{t_0}^{\tau_1} [\mathbf{A}(\tau_1), \mathbf{A}(\tau_2)] d\tau_2 d\tau_1, \\ \mathbf{\Omega}_3 &= \frac{1}{12} \int_{t_0}^t \int_{t_0}^{\tau_1} \int_{t_0}^{\tau_2} [[\mathbf{A}(\tau_1), \mathbf{A}(\tau_2)], \mathbf{A}(\tau_3)] + [\mathbf{A}(\tau_1), [\mathbf{A}(\tau_2), \mathbf{A}(\tau_3)]] d\tau_3 d\tau_2 d\tau_1. \end{aligned}$$

Aus dieser Magnus-Entwicklung entstehen numerische Verfahren, indem man die Taylor-Entwicklung von  $\mathbf{A}(t)$  rund um  $t_0 + \frac{h}{2}$  und Quadraturformeln anwendet. Da die Lösung symmetrisch in der Zeit ist und Polynome niedrigen Grades exakt durch die Quadratur integrierbar sind, lassen sich einige Magnus-Methoden einfach herleiten.

**Beispiel 2.8.1** (Magnus-Verfahren 2. Ordnung). Die Mittelpunktsregel auf  $[0, 1]$  ergibt den Schritt von  $t_n$  zu  $t_{n+1}$  als:

$$\mathbf{y}_{n+1} = e^{\mathbf{\Omega}^{[2]}} \mathbf{y}_n \text{ mit } \mathbf{\Omega}^{[2]} = h\mathbf{A}(t_n + \frac{h}{2}),$$

mit dem lokalen Fehler  $\mathcal{O}(h^{2+1})$  und dem globalen Fehler  $\mathcal{O}(h^2)$ .

**Beispiel 2.8.2** (Magnus-Verfahren 4. Ordnung). Die Gauss-Quadratur mit 2 Punkten auf  $[0, 1]$  ergibt den Schritt von  $t_n$  zu  $t_{n+1}$  als:

$$\mathbf{y}_{n+1} = e^{\mathbf{\Omega}^{[4]}} \mathbf{y}_n \text{ mit } \mathbf{\Omega}^{[4]} = \frac{h}{2}(\mathbf{A}_1 + \mathbf{A}_2) - h^2 \frac{\sqrt{3}}{12} [\mathbf{A}_1, \mathbf{A}_2]$$

wobei

$$\begin{aligned} \mathbf{A}_1 &= \mathbf{A} \left( t_n + \left( \frac{1}{2} - \frac{\sqrt{3}}{6} \right) h \right), \\ \mathbf{A}_2 &= \mathbf{A} \left( t_n + \left( \frac{1}{2} + \frac{\sqrt{3}}{6} \right) h \right). \end{aligned}$$

Der lokale Fehler ist  $\mathcal{O}(h^{4+1})$  und der globale Fehler ist  $\mathcal{O}(h^4)$ .

**Beispiel 2.8.3** (Magnus-Verfahren 4. Ordnung - Simpson-Regel). Die Simpson-Regel mit 3 Punkten auf  $[0, 1]$  ergibt den Schritt von  $t_n$  zu  $t_{n+1}$  als:

$$\mathbf{y}_{n+1} = e^{\mathbf{\Omega}^{[4]}} \mathbf{y}_n \text{ mit } \mathbf{\Omega}^{[4]} = \frac{h}{6}(\mathbf{A}_1 + 4\mathbf{A}_2 + \mathbf{A}_3) - h^2 \frac{1}{72} [\mathbf{A}_1 + 4\mathbf{A}_2 + \mathbf{A}_3, \mathbf{A}_3 - \mathbf{A}_1]$$

wobei

$$\begin{aligned}\mathbf{A}_1 &= \mathbf{A}(t_n), \\ \mathbf{A}_2 &= \mathbf{A}\left(t_n + \frac{h}{2}\right), \\ \mathbf{A}_3 &= \mathbf{A}(t_n + h).\end{aligned}$$

Der lokale Fehler ist  $\mathcal{O}(h^{4+1})$  und der globale Fehler ist  $\mathcal{O}(h^4)$ . Eine andere Methode derselben Ordnung ergibt sich mit

$$\Omega^{[4]} = \frac{h}{6}(\mathbf{A}_1 + 4\mathbf{A}_2 + \mathbf{A}_3) - h^2 \frac{1}{12}[\mathbf{A}_1, \mathbf{A}_3]$$

Wie bei der Simpson-Quadratur kann man mehrere Zeitschritte kombinieren, so dass der Rechenaufwand kleiner wird.

**Beispiel 2.8.4** (Magnus-Verfahren 6. Ordnung). Die Gauss-Legendre-Quadratur auf  $[0, 1]$  braucht für den Schritt von  $t_n$  zu  $t_{n+1}$

$$\begin{aligned}\mathbf{A}_1 &= \mathbf{A}\left(t_n + \left(\frac{1}{2} - \frac{\sqrt{15}}{10}\right)h\right), \\ \mathbf{A}_2 &= \mathbf{A}\left(t_n + \frac{h}{2}\right), \\ \mathbf{A}_3 &= \mathbf{A}\left(t_n + \left(\frac{1}{2} + \frac{\sqrt{15}}{10}\right)h\right).\end{aligned}$$

Mit der Notation

$$\alpha_1 = h\mathbf{A}_2, \quad \alpha_2 = \frac{\sqrt{13}}{3}h(\mathbf{A}_3 - \mathbf{A}_1), \quad \alpha_3 = \frac{10}{3}h(\mathbf{A}_3 - 2\mathbf{A}_2 + \mathbf{A}_1)$$

ergibt sich  $\Omega^{[6]}$  von

$$\begin{aligned}C_1 &= [\alpha_1, \alpha_2], \quad C_2 = -\frac{1}{60}[\alpha_1, 2\alpha_3 + C_1] \\ \Omega^{[6]} &= \alpha_1 + \frac{1}{12}\alpha_3 + \frac{1}{240}[-20\alpha_1 - \alpha_3 + C_1, \alpha_2 + C_2].\end{aligned}$$

**Bemerkung 2.8.5.** Die Magnus-Verfahren lassen sich manchmal auch ohne Kommutatoren hinschreiben. Zum Beispiel lässt sich die Methode der 4-ten Ordnung, die auf der Gauss-Quadratur beruht, als splitting Verfahren in zwei Arten umschreiben:

$$\mathbf{y}_{n+1} = e^{\frac{\sqrt{3}}{12}h(\mathbf{A}_2 - \mathbf{A}_1)} e^{\frac{h}{2}(\mathbf{A}_1 + \mathbf{A}_2)} e^{-\frac{\sqrt{3}}{12}h(\mathbf{A}_2 - \mathbf{A}_1)} \mathbf{y}_n$$

oder sogar nur mit zwei Exponentialfunktionen:

$$\mathbf{y}_{n+1} = e^{\alpha_1 h \mathbf{A}_1 + \alpha_2 h \mathbf{A}_2} e^{\alpha_2 h \mathbf{A}_1 + \alpha_1 h \mathbf{A}_2} \mathbf{y}_n$$

wobei  $\alpha_1 = \frac{3-2\sqrt{3}}{12}$  und  $\alpha_2 = \frac{3+2\sqrt{3}}{12}$ .

Für Details, siehe [S. BLANES, P.C. MOAN](#), *Fourth- and sixth-order commutator-free Magnus integrators for linear and non-linear dynamical systems*, Applied Numerical Mathematics 56 (2006).

**Beispiel 2.8.6.** Die Mathieu-Gleichung

$$\ddot{y} + (\omega^2 + \varepsilon \cos t)y = 0$$

$$y(0) = 1, \dot{y}(0) = 0$$

lässt sich mit dem folgenden Code lösen, wobei  $f(t) = \omega^2 + \varepsilon \cos t$  ist.

**Code 2.8.7:** Magnus Verfahren 4ter Ordnung ohne Kommutatoren

```

1  """
2  Example 1 from S. Blanes, P.C. Moan / Applied Numerical Mathematics 56 (2006)
3  1519-1537 commutator free Magnus integrator of order 4
4  """
5
6  from numpy import zeros, cos, array, pi, dot, arange
7  from scipy.linalg import expm
8  import matplotlib.pyplot as plt
9
10 # Magnus constants
11 c1 = 0.5*(1. - 0.5773502691896258)
12 c2 = 0.5*(1. + 0.5773502691896258)
13 a1 = 0.5*(0.5 - 0.5773502691896258)
14 a2 = 0.5*(0.5 + 0.5773502691896258)
15
16 # Commutator free Magnus expansion of order 4
17 def Magnus_CF4(tspan, u0, A, N):
18     u = 1.*u0
19     h = (tspan[1] - tspan[0])/(1.*N)
20     for k in range(N):
21         t0 = tspan[0] + k*h
22         A1 = A(t0 + c1*h)
23         A2 = A(t0 + c2*h)
24         u = dot(expm(a1*h*A1 + a2*h*A2), dot(expm(a2*h*A1 + a1*h*A2), u))
25     return u
26
27 # Parameters
28 tspan = [0,20*pi]
29 eps = 0.25
30 omega = 1.
31 y0 = 1
32 y0p = 0
33
34 # Coefficient of the Mathieu equation
35 f = lambda t, omega, eps: omega**2 + eps*cos(t)
36
37 # Right-hand-side matrix of the corresponding 1st order equation
38 def A(t):
39     res = zeros((2,2))
40     res[0,1] = 1.

```

```
    res[1,0] = -f(t, omega, eps)
41     return res;

43     print('Compute the almost exact solution by brute force')
    N = 2**16
45     u0 = array([y0,y0p])
    print('N = ' + str(N))
47     u_exact = Magnus_CF4(tspan, u0, A, N)

49     print('Compute approximate solutions and their errors')
    err = []
51     Ns = 2**arange(0,13)
    for N in Ns:
53         print('N = ' + str(N))
        u = Magnus_CF4(tspan, u0, A, N)
55         err_m = abs(u[0] - u_exact[0])
        print('error = ' + str(err_m))
57         err += [err_m]

59     # Convergence plot
    plt.title('Mathieu equation solved by Magnus expansion')
61     plt.loglog(Ns, err, 'o-', label='Magnus_CF4')
    plt.ylabel('Error')
63     plt.loglog(Ns, 1./Ns**4, label='$x^{-4}$')
    plt.xlabel('Number of time steps')
65     plt.legend()
    plt.show()
```

# Chapter 3

## Nullstellensuche

Dieses Kapitel behandelt die Suche nach Lösungen nichtlinearer Gleichungssysteme mittels iterativer Methoden. Das Problem kann man wie folgt ausdrücken: Gegeben sei die Funktion

$$F: D \subset \mathbb{R}^n \rightarrow \mathbb{R}^n, \quad n \in \mathbb{N}. \quad (3.0.1)$$

Wir suchen  $x^* \in D$ , so dass  $F(x^*) = 0$ . Im allgemeinen wissen wir nicht, ob eine Lösung existiert und, wenn es so ist, ob sie eindeutig ist. Trotzdem besprechen wir allgemeine Verfahren, die aber im Einzelfall noch anzupassen sind, um leistungsfähige Algorithmen zu erhalten.

Bevor wir aber zur eigentlichen Nullstellensuche kommen, wollen wir zuerst allgemeine iterative Verfahren betrachten und insbesondere die Fixpunktiteration.

### 3.1 Iterative Verfahren

**Definition 3.1.1.** Ein iteratives Verfahren ist ein Algorithmus  $\phi_F$ , der die Folge  $x^{(0)}, x^{(1)}, x^{(2)}, \dots$  approximativer Lösungen  $x^{(j)}$  generiert.

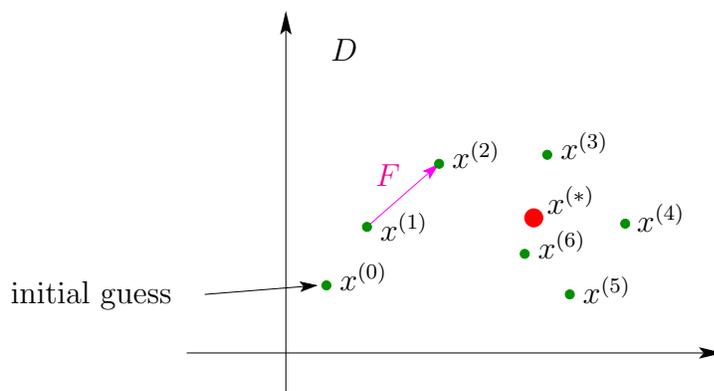


Abb. 3.1.2. Iterative Methoden

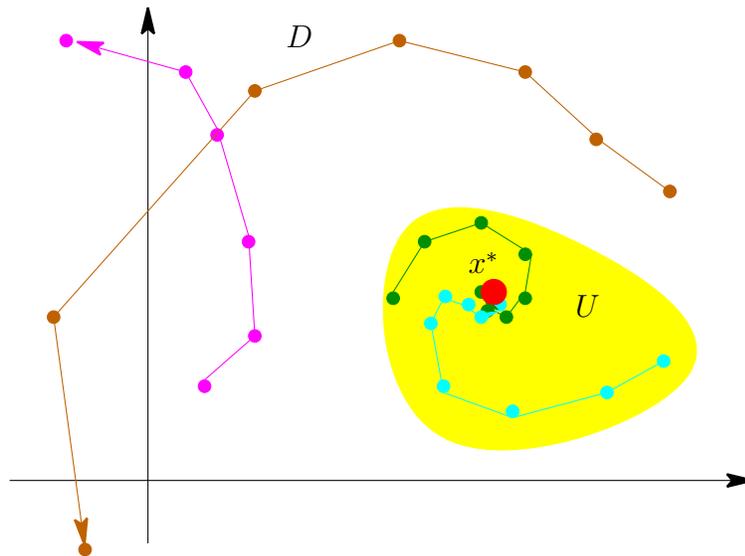


Abb. 3.1.3. Lokale Konvergenz

**Beispiel 3.1.4.** Bei gegebenem Anfangswert  $x^{(0)}$  und Funktion  $\phi$  ist ein iteratives Verfahren definiert durch  $x^{(k)} := \phi(x^{(k-1)})$ .

**Definition 3.1.5.** Ein iteratives Verfahren  $\phi_F$  zur Lösung von  $F(x^*) = 0$  konvergiert, wenn  $x^{(k)} \rightarrow x^* \in D$  und  $F(x^*) = 0$ .

**Bemerkung 3.1.6.** Normalerweise benötigt man einen Startwert, der nah am gesuchten Wert  $x^*$  liegen soll. Wie nah der Anfangswert an der Lösung sein soll hängt von dem Gleichungssystem und dem gewählten Algorithmus ab.

**Bemerkung 3.1.7.** Bei iterativen Verfahren stehen zwei Fragen im Vordergrund: wie schnell ist die Konvergenz und wann sollten wir die Iteration beenden? Die Antwort auf die erste Frage erleichtert uns die Wahl eines bestimmten Verfahrens. Eine gute Antwort auf die zweite Frage sichert die Qualität der numerischen Lösung.

**Definition 3.1.8.** (Norm) Sei  $X$  ein linearer Raum über  $\mathbb{K}$  (mit  $\mathbb{K} = \mathbb{C}$  oder  $\mathbb{R}$ ). Eine Abbildung  $\|\cdot\| : X \mapsto \mathbb{R}_0^+$  heisst *Norm* auf  $X$ , falls sie erfüllt:

- (i)  $\forall \mathbf{x} \in X: \mathbf{x} \neq 0$  dann und nur dann, wenn  $\|\mathbf{x}\| > 0$ ,
- (ii)  $\|\lambda \mathbf{x}\| = |\lambda| \|\mathbf{x}\|$  für alle  $\mathbf{x} \in X$ , und alle  $\lambda \in \mathbb{K}$ ,
- (iii)  $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$  für alle  $\mathbf{x}, \mathbf{y} \in X$  (Dreiecksungleichung).

**Beispiel 3.1.9.** Wir betrachten Vektoren  $\mathbf{x} = (x_1, x_2, \dots, x_n)^T \in \mathbb{K}^n$

Name	:	Definition	numpy.linalg
Euklidische Norm	:	$\ \mathbf{x}\ _2 := \sqrt{ x_1 ^2 + \dots +  x_n ^2}$	<code>norm(x)</code>
1-Norm	:	$\ \mathbf{x}\ _1 :=  x_1  + \dots +  x_n $	<code>norm(x, 1)</code>
$\infty$ -Norm, Max-Norm	:	$\ \mathbf{x}\ _\infty := \max\{ x_1 , \dots,  x_n \}$	<code>norm(x, inf)</code>

**Definition 3.1.10.** Zwei Normen  $\|\cdot\|_1$  und  $\|\cdot\|_2$  auf einem linearen Raum  $V$  sind äquivalent, wenn es Konstanten  $\underline{C}$  und  $\overline{C}$  gibt, so dass

$$\underline{C} \|v\|_1 \leq \|v\|_2 \leq \overline{C} \|v\|_1$$

für alle  $v \in V$ .

**Theorem 3.1.11.** Wenn  $\dim V < \infty$ , dann sind alle Normen auf  $V$  äquivalent.

**Beispiel 3.1.12.** Einfache explizite Normäquivalenzen: für alle  $\mathbf{x} \in \mathbb{K}^n$  gilt:

$$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n} \|\mathbf{x}\|_2, \quad (3.1.1)$$

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_2 \leq \sqrt{n} \|\mathbf{x}\|_\infty, \quad (3.1.2)$$

$$\|\mathbf{x}\|_\infty \leq \|\mathbf{x}\|_1 \leq n \|\mathbf{x}\|_\infty. \quad (3.1.3)$$

**Definition 3.1.13.** Die Folge  $x^{(k)}$  heisst *linear konvergent* gegen  $x^*$ , falls:

$$\text{es ein } L < 1 \text{ gibt, so dass } \|x^{(k+1)} - x^*\| \leq L \|x^{(k)} - x^*\| \text{ für alle } k \geq k_0. \quad (3.1.4)$$

Die Konstante  $L$  nennt man Konvergenzrate.

**Bemerkung 3.1.14.** Die Konvergenz eines Verfahrens ist *unabhängig* von der Wahl der Norm. Die Konvergenzrate oder sogar die Tatsache, dass die Konvergenz linear ist, sind jedoch von der Normwahl abhängig.

**Definition 3.1.15.** Ein Verfahren hat *Konvergenzordnung*  $p$ , falls:

$$\text{es ein } C > 0 \text{ gibt, so dass } \|x^{(k+1)} - x^*\| \leq C \|x^{(k)} - x^*\|^p \text{ für alle } k \in \mathbb{N} \quad (3.1.5)$$

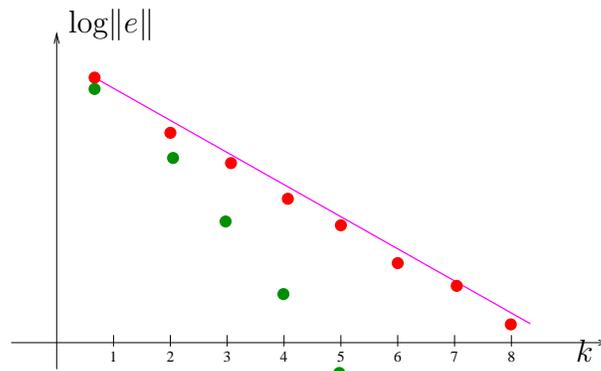
mit  $C < 1$  für  $p = 1$ .

Hier wird stillschweigend angenommen, dass der Startwert nah an der Lösung  $x^*$  liegt, z.B.  $\|x^0 - x^*\| < 1$ , damit wir eine konvergente Folge erhalten.

**Bemerkung 3.1.16.** In einem lin-log Plot der Fehler sehen wir eine Gerade im Fall linearer Konvergenz, denn

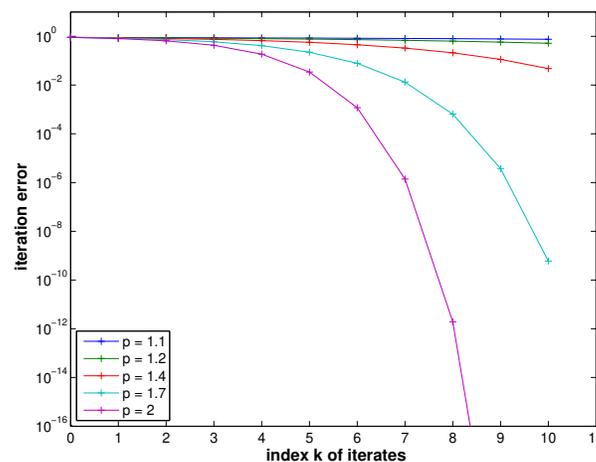
$$\|\mathbf{e}^{(k)}\| \leq L^k \|\mathbf{e}^{(0)}\|,$$

$$\log(\|\mathbf{e}^{(k)}\|) \leq k \log(L) + \log(\|\mathbf{e}^{(0)}\|),$$



**Abb. 3.1.17.** Beispiel linearer Konvergenz; jede schnellere Konvergenz ist auch mindestens linear!

Eine konvergenz höherer Ordnung ist an gekrümmten Konvergenzkurven erkennbar:



**Abb. 3.1.18.** Iterationen mit verschiedenen Konvergenzordnungen  $p$ .

**Bemerkung 3.1.19.** Die Konvergenzrate bei linearer Konvergenz lässt sich leicht abschätzen. Sei  $\epsilon_k := \|\mathbf{x}^{(k)} - \mathbf{x}^*\|$  die Norm des Fehlers im  $k$ -ten Schritt. Bei linearer Konvergenz (Definition 3.1.13) nehmen wir an (hier  $0 < L < 1$ ):

$$\epsilon_{k+1} \approx L\epsilon_k \implies \log \epsilon_{k+1} \approx \log L + \log \epsilon_k \implies \log \epsilon_{k+1} \approx k \log L + \log \epsilon_0. \quad (3.1.6)$$

Somit beschreibt  $\log L < 0$  die Steigung der Geraden im lin-log Plot des Fehlers.

**Beispiel 3.1.20.** Wir beobachten das Verhalten folgender Iteration:

$$x^{(k+1)} = x^{(k)} + \frac{\cos x^{(k)} + 1}{\sin x^{(k)}}.$$

**Code 3.1.21:** einfache Fixpunktiteration

```
from numpy import sin, cos, array
```

```

2
4
6
8
10
12
#iteration with linear convergence

def lincvg(x):
    y = [] #container for the  $x^{(j)}$ 
    for k in range(15):
        x = x + (cos(x)+1)/sin(x) #apply the iteration formula
        y += [x] #store the value in the container
    err = abs(array(y) - x) #estimation for the error
    rate = err[1:]/err[:-1] #estimation for convergence rate
    return err, rate

```

**Achtung:**

Wir verwenden  $x^{(15)}$  statt der exakten Lösung  $x^*$  in der Berechnung der Konvergenzrate; im Allgemeinen ist die exakte Lösung ja unbekannt. Wenn aber die Iteration nicht nach 15 Schritten nicht konvergiert hat, ist eine solche Schätzung des Fehlers sinnlos.

$k$	$x^{(0)} = 0.4$		$x^{(0)} = 0.6$		$x^{(0)} = 1$	
	$x^{(k)}$	$\frac{ x^{(k)} - x^{(15)} }{ x^{(k-1)} - x^{(15)} }$	$x^{(k)}$	$\frac{ x^{(k)} - x^{(15)} }{ x^{(k-1)} - x^{(15)} }$	$x^{(k)}$	$\frac{ x^{(k)} - x^{(15)} }{ x^{(k-1)} - x^{(15)} }$
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

Konvergenzrate  $L \approx 0.5$

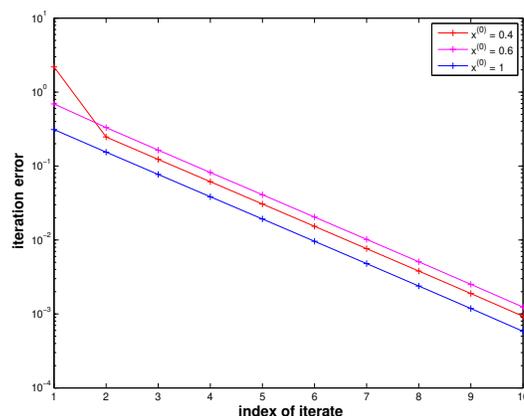


Abb. 3.1.22. Lineare Konvergenz

**Beispiel 3.1.23.** (Quadratische Konvergenz) Sei die Fixpunktiteration zur Berechnung von  $\sqrt{a}$ :

$$x^{(k+1)} = \frac{1}{2}\left(x^{(k)} + \frac{a}{x^{(k)}}\right) \implies |x^{(k+1)} - \sqrt{a}| = \frac{1}{2x^{(k)}}|x^{(k)} - \sqrt{a}|^2. \quad (3.1.7)$$

Da  $\sqrt{vw} \leq \frac{1}{2}(v+w)$ , gilt:  $x^{(k+1)} = \frac{1}{2}(v+w) \geq \sqrt{vw} = \sqrt{a}$  für  $k \geq 1$  (mit  $v = x^{(k)}$  und  $w = \frac{a}{x^{(k)}}$ ) ist die Folge durch  $\sqrt{a}$  nach unten beschränkt.

Es gilt ausserdem  $x^{(k+1)} < x^{(k)}$  für alle  $k \geq 2$ . Damit ist die Folge  $(x^{(k)})_{k \in \mathbb{N}_0}$  monoton und beschränkt, also konvergent.

Nun wollen wir aus einem numerischen Experiment die Konvergenzordnung dieser Iteration erraten. Wir notieren  $\epsilon_k := \|\mathbf{x}^{(k)} - \mathbf{x}^*\|$ . Dann:

$$\epsilon_{k+1} \approx C\epsilon_k^p \implies \log \epsilon_{k+1} \approx \log C + p \log \epsilon_k \implies \frac{\log \epsilon_{k+1} - \log \epsilon_k}{\log \epsilon_k - \log \epsilon_{k-1}} \approx p.$$

Das numerische Experiment für  $a = 2$  liefert:

$k$	$x^{(k)}$	$e^{(k)} := x^{(k)} - \sqrt{2}$	$\log \frac{ e^{(k)} }{ e^{(k-1)} } : \log \frac{ e^{(k-1)} }{ e^{(k-2)} }$
0	2.000000000000000000	0.58578643762690485	
1	1.500000000000000000	0.08578643762690485	
2	1.416666666666666652	0.00245310429357137	1.850
3	1.41421568627450966	0.00000212390141452	1.984
4	1.41421356237468987	0.00000000000159472	2.000
5	1.41421356237309492	0.00000000000000022	

Wir beobachten, dass sich die Anzahl der exakten Dezimalstellen im Ergebnis in jedem Schritt verdoppelt. Das gilt für jede konvergente Iteration mit Ordnung 2. Wenn wir den relativen Fehler im Schritt  $k$  mit  $\delta_k$  notieren, haben wir:

$$\begin{aligned} x^{(k)} &= x^*(1 + \delta_k) \implies x^{(k)} - x^* = \delta_k x^*, \\ \implies |x^* \delta_{k+1}| &= |x^{(k+1)} - x^*| \leq C|x^{(k)} - x^*|^2 = C|x^* \delta_k|^2 \\ &\implies |\delta_{k+1}| \leq C|x^*| \delta_k^2. \end{aligned} \quad (3.1.8)$$

Wenn  $C \approx 1$ , dann  $\delta_k = 10^{-\ell}$  und (3.1.8) zeigt  $\delta_{k+1} \approx 10^{-2\ell}$ .

**Code 3.1.24:** Beispiel einer stationären Iteration

```

from numpy import sqrt, array
2
def sqrtit(a,x): #iteration for computing sqrt(a)
4     exact = sqrt(a) #exact value found with numpy
     e = [x] #array with the values of the iteration
6     x_old = -1.
     while x_old != x:
8         x_old = x
         x = 0.5*(x+a/x)
10        e += [x] #store new iteration value

```

```

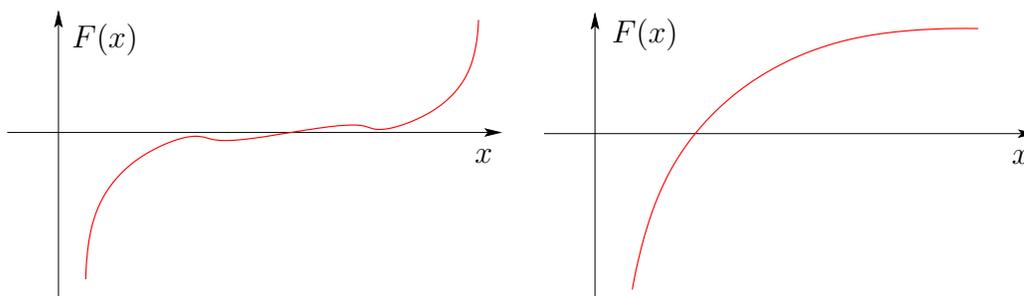
12     e = array(e)
13     e = abs(e-exact) #compute the absolute errors
14     return e
15
16 e = sqrtit(2.,1.)
17 print(e)

```

## 3.2 Abbruchkriterien

Bei iterativen Verfahren stellt sich die Frage, bei welchem  $k$  man die Iteration abbrechen soll oder kann. Hierbei ist es entscheidend, ob die gesetzten Ziele, wie zum Beispiel Genauigkeit, erreicht wurden oder vielleicht keine Verbesserung des Ergebnisses mehr eintreten kann. Einige mögliche Kriterien sind:

- A priori* Kriterium: nach einer festen Anzahl  $k_0$  an Schritten wird das Verfahren beendet. Dies ist unsicher.
- A posteriori* Kriterium: verwende die berechneten Werte. Wenn der Fehler unter einer gewissen Toleranz ( $\|x^{(k)} - x^*\| < \tau$ ) liegt, bricht man ab. Man kennt aber den gesuchten Wert  $x^*$  nicht.
- Führe die Iteration bis  $x^{(k+1)} \approx x^{(k)}$  durch. Diese Methode ist aber auch sehr ineffizient.
- Verwende das Residuum, also Abbruch wenn  $\|F(x^{(k)})\| < \tau$ . Dies birgt jedoch Probleme, denn  $\|F(x^{(k)})\|$  kann klein sein, obwohl  $\|x^{(k)} - x^*\|$  gross ist (siehe Bild).



**Abb. 3.2.1.** Im Falle einer wie im Graphen a) aussehenden Funktion kann das Residuum als Abbruchkriterium problematisch sein.

Folgende Bemerkung zeigt, wie man ein a-posteriori Abbruchkriterium im Falle einer linear-konvergenten Iteration bauen kann.

**Bemerkung 3.2.2.** Bei linearer Konvergenz und bekanntem  $L$  kann man folgende Abschätzung verwenden:

$$\|x^{(k+1)} - x^*\| \leq \frac{L}{1-L} \|x^{(k+1)} - x^{(k)}\|. \quad (3.2.1)$$

*Proof.*

$$\begin{aligned}
 \|x^{(k+1)} - x^*\| &\leq L\|x^{(k)} - x^*\| \leq L\|x^{(k+1)} - x^{(k)}\| + L\|x^{(k+1)} - x^*\| \\
 \Rightarrow (1 - L)\|x^{(k+1)} - x^*\| &\leq L\|x^{(k+1)} - x^{(k)}\| \\
 \Rightarrow \|x^{(k+1)} - x^*\| &\leq \frac{L}{1 - L} \|x^{(k+1)} - x^{(k)}\|.
 \end{aligned} \tag{3.2.2}$$

□

Somit berechnen wir die Abschätzung

$$\text{errest}[k] = \frac{L}{1 - L} \|x^{(k)} - x^{(k-1)}\|,$$

und wir brechen die Iteration ab, sobald  $\text{errest}[k] < \tau$ .

**Beispiel 3.2.3.** Im Beispiel 3.1.20 haben wir eine lineare Konvergenz gegen  $\pi$  mit einer Konvergenzrate 0.5 beobachtet. Wir überprüfen die Qualität unseres Fehlerschätzers für dieses Beispiel. Hier sind die Ergebnisse (absoluter Fehler, Wert der Fehlerschätzung und Fehler in der Abschätzung des Fehlers) für den Startwert  $x^0 = 0.4$ :

$k$	$ x^{(k)} - \pi $	$\frac{L}{1-L} x^{(k)} - x^{(k-1)} $	Fehler in der Abschätzung
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

Wir sehen, dass der a-posteriori-Fehlerschätzer in diesem Fall sehr genau ist.

### 3.3 Fixpunktiteration

Wir schreiben die Nullstellensuche  $F(x) = 0$  in ein Fixpunktproblem  $\phi(x) = x$  um. Die Funktion  $\phi$  liefert nun die Iterationsvorschrift! Ein 1-Punkt-Verfahren benötigt dabei nur den vorhergegangenen Schritt:  $x^{(k+1)} = \phi(x^{(k)})$ .

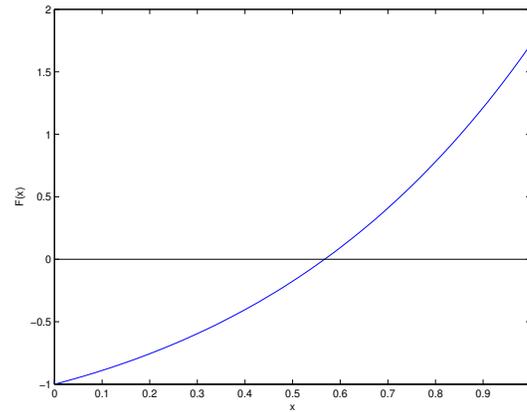
**Definition 3.3.1.** Eine Fixpunktiteration heisst konsistent mit  $F(x) = 0$ , falls

$$F(x) = 0 \Leftrightarrow \phi(x) = x. \quad (3.3.1)$$

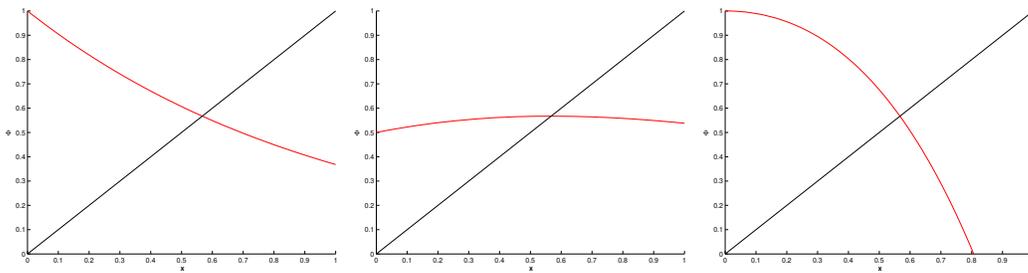
**Beispiel 3.3.2.** Sei  $F(x) = xe^x - 1$  mit  $x \in [0, 1]$ . Dann betrachten wir die folgenden iterativen Verfahren:

- $\phi_1(x) = e^{-x}$  liefert lineare Konvergenz;
- $\phi_2(x) = \frac{1+x}{1+e^x}$  gibt quadratische Konvergenz;
- $\phi_3(x) = x + 1 - xe^x$  erzeugt eine divergente Folge.

Hier sind die Ergebnisse dieser drei Iterationen:



**Abb. 3.3.3.** Funktion  $F$



**Abb. 3.3.4.** Die Funktionen  $\phi_1$ ,  $\phi_2$  und  $\phi_3$ .

$k$	$x^{(k+1)} := \phi_1(x^{(k)})$	$x^{(k+1)} := \phi_2(x^{(k)})$	$x^{(k+1)} := \phi_3(x^{(k)})$
0	0.5000000000000000	0.5000000000000000	0.5000000000000000
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

$k$	$ x_1^{(k+1)} - x^* $	$ x_2^{(k+1)} - x^* $	$ x_3^{(k+1)} - x^* $
0	0.067143290409784	0.067143290409784	0.067143290409784
1	0.039387369302849	0.000832287212566	0.108496074240152
2	0.021904078517179	0.00000125374922	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

Wir beobachten lineare Konvergenz für  $x_1^{(k)}$ , quadratische Konvergenz für  $x_2^{(k)}$ , und keine Konvergenz für  $x_3^{(k)}$ ; für alle drei Iterationen hat  $x_i^{(0)} = 0.5$  als Startwert gedient.

Wir sehen, dass man ein Nullstellenproblem in verschiedene Fixpunktiterationen umschreiben kann, die sich aber sehr unterschiedlich verhalten können. Wann und wie schnell konvergiert eine Fixpunktiteration?

**Definition 3.3.5.** Eine Funktion  $\phi$  heisst *Kontraktion*, falls

$$\text{es ein } L < 1 \text{ gibt, so dass } \|\phi(x) - \phi(y)\| \leq L\|x - y\| \text{ für alle } x, y. \quad (3.3.2)$$

**Bemerkung 3.3.6.** Wenn  $x^*$  ein Fixpunkt der Kontraktion  $\phi$  ist, dann ist

$$\|x^{(k+1)} - x^*\| = \|\phi(x^{(k)}) - \phi(x^*)\| \leq L\|x^{(k)} - x^*\|,$$

Das heisst, dass iterative Verfahren  $x^{k+1} = \phi(x^k)$  mindestens linear konvergieren.

**Satz 3.3.7.** (Banach'scher Fixpunktsatz)

Sei  $D \subset \mathbb{K}^n$  ( $\mathbb{K} = \mathbb{R}, \mathbb{C}$ ), mit  $D$  abgeschlossen, und  $\phi: D \rightarrow D$  einer Kontraktion. Dann existiert ein eindeutiger Fixpunkt  $x^*$ , also  $\phi(x^*) = x^*$ . Dieser ist der Grenzwert der Folge  $x^{(k+1)} = \phi(x^{(k)})$ .

*Proof.* Betrachte die Einpunktiteration  $x^{(k)} = \phi(x^{(k-1)})$  mit dem Startwert  $x^{(0)} \in D$ . Die Dreiecksungleichung und die Kontraktionseigenschaft von  $\phi$  ergeben:

$$\begin{aligned} \|x^{(k+N)} - x^{(k)}\| &\leq \sum_{j=k}^{k+N-1} \|x^{(j+1)} - x^{(j)}\| \leq \sum_{j=k}^{k+N-1} L^j \|x^{(1)} - x^{(0)}\| \\ &\leq \frac{L^k}{1-L} \|x^{(1)} - x^{(0)}\| \xrightarrow{k \rightarrow \infty} 0. \end{aligned}$$

Somit ist  $(x^{(k)})_{k \in \mathbb{N}_0}$  eine Cauchy-Folge in  $\mathbb{K}^n$ , und demnach konvergent. Wir bezeichnen mit  $x^*$  den Grenzwert:  $x^{(k)} \xrightarrow{k \rightarrow \infty} x^*$ . Die Stetigkeit von  $\phi$  ergibt dann  $\phi(x^*) = x^*$ .  $\square$

Wir wollen nun eine wichtige Eigenschaft der Fixpunktiteration  $x^{(k+1)} = \phi(x^{(k)})$  verstehen.

**Hilfssatz 3.3.8.** (Hinreichende Bedingung für lokale lineare Konvergenz einer Fixpunktiteration) Es sei  $U$  konvex und  $\phi: U \subset \mathbb{R}^n \mapsto \mathbb{R}^n$  stetig differenzierbar mit  $L := \sup_{x \in U} \|D\phi(x)\| < 1$  (wobei  $D\phi(x)$  die Jacobi-Matrix von  $\phi(x)$  bezeichnet). Wenn  $\phi(x^*) = x^*$  für  $x^* \in U$ , dann konvergiert die Fixpunktiteration  $x^{(k+1)} = \phi(x^{(k)})$  gegen  $x^*$  lokal mindestens linear.

*Beweis.* Die Definition der Ableitung ergibt

$$\|\phi(y) - \phi(x^*) - D\phi(x^*)(y - x^*)\| \leq \psi(\|y - x^*\|)\|y - x^*\|,$$

mit  $\psi: \mathbb{R}_0^+ \mapsto \mathbb{R}_0^+$  so dass  $\lim_{t \rightarrow 0} \psi(t) = 0$ .

Wir wählen ein  $\delta > 0$ , so dass

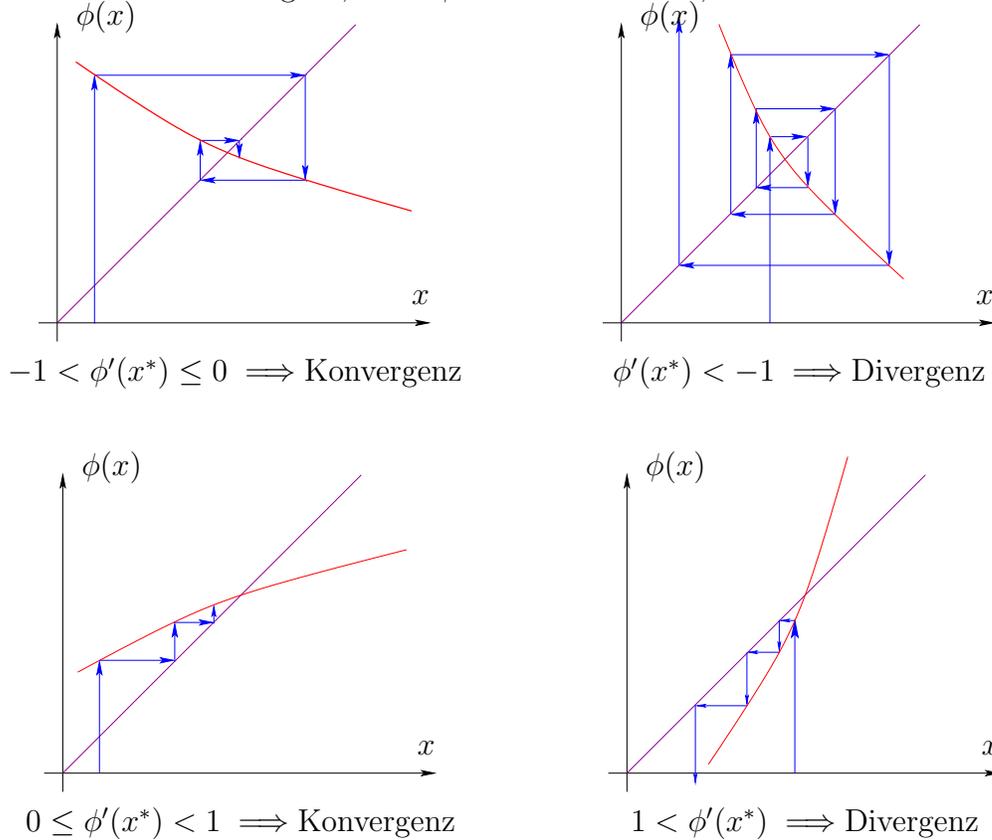
$$\psi(t) + \|D\phi(x^*)\| \leq \frac{1}{2}(1 + \|D\phi(x^*)\|) \leq L < 1 \quad \forall 0 \leq t < \delta.$$

Die Dreiecksungleichung, zusammen mit  $x^{(k+1)} = \phi(x^{(k)})$ , ergibt dann für die Fixpunktiteration:

$$\begin{aligned} \|\phi(x) - x^*\| - \|D\phi(x^*)(x - x^*)\| &\leq \psi(\|x - x^*\|)\|x - x^*\| \\ \|x^{(k+1)} - x^*\| &\leq (\psi(t) + \|D\phi(x^*)\|) \|x^{(k)} - x^*\| \leq L \|x^{(k)} - x^*\|, \end{aligned}$$

für  $\|x^{(k)} - x^*\| < \delta$ .  $\square$

**Beispiel 3.3.9.** Wir können das letzte Ergebnis mittels einer eindimensionalen Funktion veranschaulichen: wenn  $\phi$  ziemlich flach um den Fixpunkt  $x^*$  ist, dann haben wir lokale Konvergenz; wenn  $\phi$  dort aber steil ist, dann haben wir Divergenz.



**Abb. 3.3.10.** Verhalten einiger Iterationen

**Hilfssatz 3.3.11.** Sei  $\phi: \mathbb{R}^n \rightarrow \mathbb{R}^n$  mit  $\phi(x^*) = x^*$  und  $\phi$  stetig differenzierbar in  $x^*$ . Ist  $\|D\phi(x^*)\| < 1$ , dann konvergiert die Fixpunktiteration  $x^{(k+1)} = \phi(x^{(k)})$  lokal und (mindestens) linear, mit  $L = \|D\phi(x^*)\|$ .

*Beweis.* Es gilt:

$$\begin{aligned} \phi(x) - \phi(y) &= \int_0^1 D\phi(x + \tau(y-x))(y-x) \, d\tau \implies \\ \|\phi(x) - \phi(y)\| &\leq L \|y-x\|. \end{aligned}$$

Da  $\|D\phi(x^*)\| < 1$  und  $D\phi$  stetig ist, gilt  $\|D\phi(x^*)\| < 1$  in einer ganzen Umgebung von  $x^*$ . Somit ist  $L = \max_{\tau \in [0,1]} \|D\phi(x + \tau(y-x))(y-x)\|$  strikt kleiner als 1. Damit ist  $\phi$  eine Kontraktion in einer Umgebung von  $x^*$  und hat den Fixpunkt  $x^*$ .

**Beispiel 3.3.12.** (Fixpunktiteration in mehrdimensionalen Räumen)

Wir schreiben das folgende Gleichungssystem in ein Fixpunktproblem um:

$$\begin{cases} x_1 - c(\cos x_1 - \sin x_2) = 0 \\ (x_1 - x_2) - c \sin x_2 = 0 \end{cases} \implies \begin{cases} c(\cos x_1 - \sin x_2) = x_1 \\ c(\cos x_1 - 2 \sin x_2) = x_2. \end{cases}$$

Wir definieren:

$$\phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = c \begin{pmatrix} \cos x_1 - \sin x_2 \\ \cos x_1 - 2 \sin x_2 \end{pmatrix} \implies D\phi \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = -c \begin{pmatrix} \sin x_1 & \cos x_2 \\ \sin x_1 & 2 \cos x_2 \end{pmatrix}.$$

Wir wählen eine *geeignete* Norm:  $\|\cdot\| = \infty$ -norm  $\|\cdot\|_\infty$ :

$$\text{wenn } c < \frac{1}{3} \implies \|D\phi(x)\|_\infty < 1 \quad \forall \mathbf{x} \in \mathbb{R}^2,$$

und somit ergibt diese Fixpunktiteration mindestens lineare Konvergenz. Die Existenz eines Fixpunktes ist auch gesichert, da  $\phi$  in der abgeschlossenen Menge  $[-3, 3]^2$  agiert und der Banach'sche Fixpunktsatz angewendet werden kann.

**Satz 3.3.13.** (Formel von Taylor)

Sei  $U \subseteq \mathbb{R}$  ein Intervall,  $\phi: U \rightarrow \mathbb{R}$   $(m+1)$ -mal differenzierbar und  $x \in U$ . Dann gilt für jedes  $y \in U$ :

$$\phi(y) - \phi(x) = \sum_{k=1}^m \frac{1}{k!} \phi^{(k)}(x)(y-x)^k + O(|y-x|^{m+1}). \tag{3.3.3}$$

**Hilfssatz 3.3.14.** Sei  $U \subseteq \mathbb{R}$  ein Intervall und  $\phi: U \rightarrow \mathbb{R}$   $(m+1)$ -mal differenzierbar mit  $\phi(x^*) = x^* \in U$ . Sei weiterhin  $\phi^{(l)}(x^*) = 0$  für  $l = 1, 2, \dots, m$  ( $m \geq 1$ ). Dann konvergiert die Fixpunktiteration  $x^{(k+1)} = \phi(x^{(k)})$  gegen  $x^*$  lokal mit der Ordnung  $p \geq m+1$ .

*Proof.* Setzt man  $y = x^{(k)}$  und  $x = x^*$ , dann ergibt die Formel von Taylor:

$$x^{(k+1)} - x^* = \phi(x^{(k)}) - \phi(x^*) = \sum_{l=1}^m \frac{1}{l!} \underbrace{\phi^{(l)}(x^*)}_{=0} (x^{(k)} - x^*)^l + O(|x^{(k)} - x^*|^{m+1}). \quad (3.3.4)$$

□

**Beispiel 3.3.15.**

- $\phi_1(x) = e^{-x}$  ist linear konvergent, da  $\phi'(x^*) = -e^{-x^*} = -x^* \in [-1, 0]$ .
- $\phi_2(x) = \frac{F(x)}{(1+e^x)^2}$  ist quadratisch konvergent, da  $\phi_2'(x) = 0$ .

**Lemma 3.3.16.** Konvergiert die Kontraktion  $\phi$  linear mit dem Faktor  $L < 1$ , dann gilt die folgende Abschätzung:

$$\|x^* - x^{(k)}\| \leq \frac{L^{k-l}}{1-L} \|x^{(l+1)} - x^{(l)}\|. \quad (3.3.5)$$

*Proof.*

$$\begin{aligned} \|x^{k+m} - x^{(k)}\| &\leq \sum_{j=k}^{k+m-1} \|x^{(j+1)} - x^{(j)}\| \\ &\leq \sum_{j=k}^{k+m-1} L^{j-k} \|x^{k+1} - x^{(k)}\| \\ &= \frac{1-L^m}{1-L} \|x^{(k+1)} - x^{(k)}\| \\ &\leq \frac{1-L^m}{1-L} L^{k-l} \|x^{(l+1)} - x^{(l)}\|. \end{aligned} \quad (3.3.6)$$

Nimmt man jetzt auf beiden Seiten den Limes  $m \rightarrow \infty$ , so erhält man die gesuchte Abschätzung. □

**Korollar 3.3.17.** •  $l = 0$  ergibt ein a priori Abbruchkriterium:

$$\|x^* - x^{(k)}\| \leq \frac{L^k}{1-L} \|x^{(1)} - x^{(0)}\| \leq \tau \Rightarrow k = \dots;$$

Wir können also das maximale  $k$  schätzen.

- $l = k - 1$  ergibt ein a posteriori Abbruchkriterium:

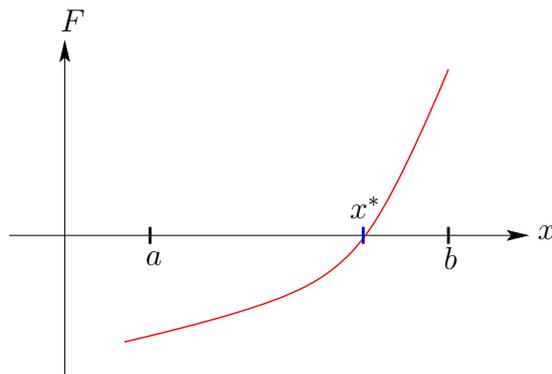
$$\|x^* - x^{(k)}\| \leq \frac{L}{1-L} \|x^{(k)} - x^{(k-1)}\| \leq \tau.$$

Diese Gleichung wird in jedem Schritt verifiziert. Wenn sie erfüllt ist wird die Iteration abgebrochen.

## 3.4 Intervallhalbierungsverfahren

**Erster Ansatz:** Mit Hilfe des Zwischenwertsatzes.

Ist  $F$  stetig auf einem Intervall  $[a, b]$  und  $F(a)F(b) < 0$  (d.h.  $F$  hat verschiedene Vorzeichen an den Endpunkten), so muss es ein  $x^* \in [a, b]$  geben mit  $F(x^*) = 0$ . Man setzt zuerst  $c = \frac{a+b}{2}$ , betrachtet  $F(a)F(c)$  und  $F(b)F(c)$  und fährt dann auf dem Intervall  $[a, c]$  oder  $[c, b]$  fort, je nachdem in welchem Intervall sich das Vorzeichen geändert hat.



Code 3.4.1: Bisektionsverfahren

```

def mybisect(f,a,b,tol=1e-12):
2
    if a>b:
4        t = a; a = b; b = t #swap a and b

        #evaluate f at two endpoints
6        fa = f(a); fb = f(b)

8
    if fa*fb > 0: raise ValueError #fa and fb must have different signs

10
    v = 1
12    if fa>0: v = -1
        x = 0.5*(a+b) #middle point between a and b
14    k = 1 #number of iterations
        while (b-a>tol) and (a<x) and (x<b):
16        if v*f(x)>0: b = x #if fa<0 v=1 then we just substitute b with x if
f(x)>0.
            #On the contrary, if fa>0 the function is decreasing (since then fb<0),
18            #so if also f(x) < 0 we substitute b with x
            else: a = x #if the above conditions are not fulfilled with choose a=x.
                #Hint: draw a graph in order to understand this idea.
                x = 0.5*(a+b)
22            k += 1 #update the number of iterations
    return x, k

24
    if __name__=='__main__':
26        f = lambda x: x**3 - 2*x**2 + 4.*x/3. - 8./27.

```

```

28     x, k = mybisect(f,0,1)
        print("Results of mybisect:")
30     print('x_bisect = ', x, ' after k=', k, ' iterations')

32     #fsolve is a profession solver from scipy and bisect is scipy's
implementation of bisection method
        #we want to compare our result with the one produced by these standard
routines
34     from scipy.optimize import fsolve, bisect
        print(20*" -")
36     print("Results of fsolve:")
        print(fsolve(f,0,full_output=True))
38     print(20*" -")
        print("Results of scipy's bisect")
40     print(bisect(f,0,1))

```

**Beachte:** Dieses Verfahren hat folgende Eigenschaften:

- Es ist ziemlich einfach, denn es braucht nur Auswertungen von  $F$ ;
- Im allgemeinen ist es ein sicheres Verfahren;
- Die Konvergenz gegen  $x^*$  ist linear, also ziemlich langsam;
- Es ist schwierig, es in  $n$  Dimensionen formulieren.

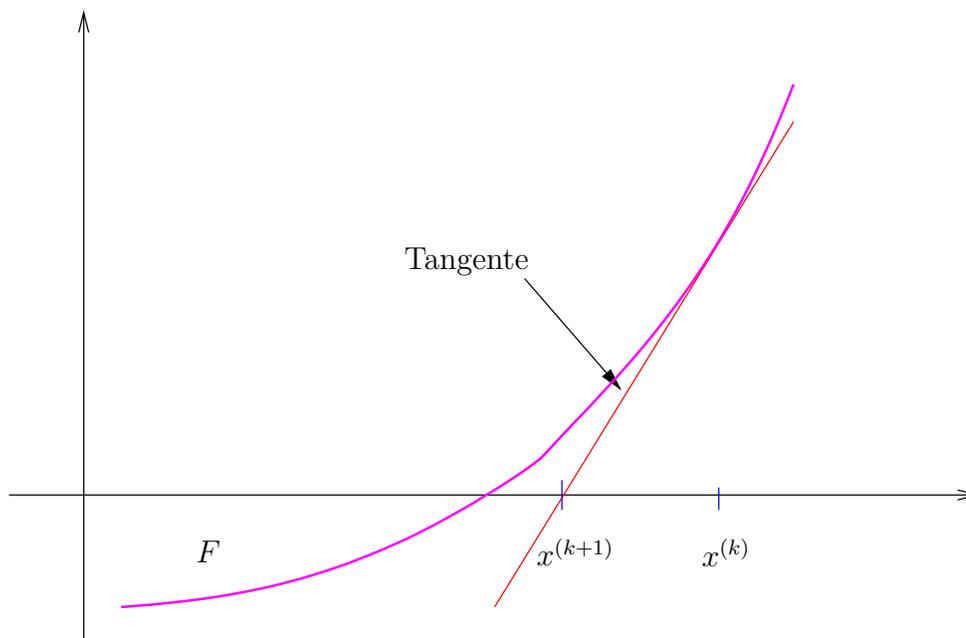
Manchmal kann der folgende Trick nützlich sein: **Idee:**

Ersetze in jedem Iterationsschritt  $F$  durch ein einfacheres  $\tilde{F}$  und löse dann  $\tilde{F}(x) = 0$ . Daraus entsteht das sogenannte *Newton-Verfahren*.

### 3.5 Newton-Verfahren in 1 Dimension

In jedem Iterationsschritt verwendet man eine lineare Funktion  $\tilde{F}$ , die mithilfe der Ableitung von  $F$  definiert wird, also  $\tilde{F} = F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)})$ . Die Nullstelle von  $\tilde{F}$  ist dann

$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})}, \quad \text{falls } F'(x^{(k)}) \neq 0. \quad (3.5.1)$$



**Abb. 3.5.1.** Ein Schritt des Newton-Verfahrens

**Bemerkung 3.5.2.** Die Theorie der Fixpunktiterationen sichert quadratische, lokale Konvergenz, falls  $F'(x^*) \neq 0$ . In der Tat, eine Newton-Iteration ist eine Fixpunktiteration mit

$$\phi(x) = x - \frac{F(x)}{F'(x)} \implies \phi'(x) = \frac{F(x)F''(x)}{(F'(x))^2} \implies \phi'(x^*) = 0,$$

falls  $F(x^*) = 0$  und  $F'(x^*) \neq 0$ .

**Beispiel 3.5.3.** (Newton-Verfahren in 1D)

Newton-Iterationen für zwei verschiedene skalare nicht-lineare Gleichungen mit gleicher Lösungen:

$$\begin{aligned} F(x) = xe^x - 1 &\implies F'(x) = e^x(1+x) \\ &\implies x^{(k+1)} = x^{(k)} - \frac{x^{(k)}e^{x^{(k)}} - 1}{e^{x^{(k)}}(1+x^{(k)})} = \frac{(x^{(k)})^2 + e^{-x^{(k)}}}{1+x^{(k)}} \end{aligned}$$

$$\begin{aligned} F(x) = x - e^{-x} &\implies F'(x) = 1 + e^{-x} \\ &\implies x^{(k+1)} = x^{(k)} - \frac{x^{(k)} - e^{-x^{(k)}}}{1 + e^{-x^{(k)}}} = \frac{1 + x^{(k)}}{1 + e^{x^{(k)}}}. \end{aligned}$$

**Beispiel 3.5.4.** (Angepasstes Newton-Verfahren)

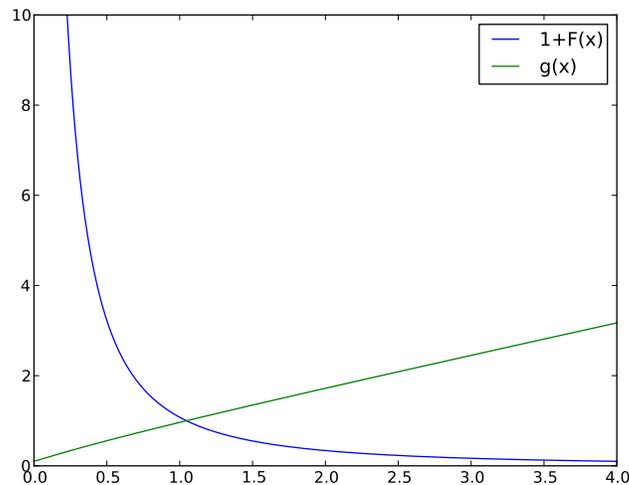
$$F(x) = \frac{1}{(x+1)^2} + \frac{1}{(x+0.1)^2} - 1, \quad x > 0.$$

**Bemerkung:**

$$F(x) + 1 \approx 2x^{-2} \text{ für } x \gg 1;$$

und somit:

$$g(x) := \frac{1}{\sqrt{F(x) + 1}} \text{ "fast linear" für } x \gg 1.$$



**Idee:**

Statt  $F(x)=0$  direkt zu lösen, behandle  $g(x)=1$  mit dem Newton-Verfahren (3.5.1). Hier sind die Ergebnisse der Iteration für den Startwert  $x^{(0)} = 0$ :

$k$	$x^{(k)}$	$F(x^{(k)})$	$x^{(k)} - x^{(k-1)}$	$x^{(k)} - x^*$
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

### 3.6 Sekantenverfahren

Wenn die Ableitung  $F'(x^{(k)})$  teuer zu berechnen ist oder gar nicht zur Verfügung steht, dann ersetzt man sie durch  $q^{(k)} := \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}}$ . Dann lautet ein Schritt des Verfahrens:

$$\tilde{F}(x) = F(x^{(k)}) + q^{(k)}(x - x^{(k)}) \Rightarrow x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{q^{(k)}}, \text{ falls } q^{(k)} \neq 0. \quad (3.6.1)$$

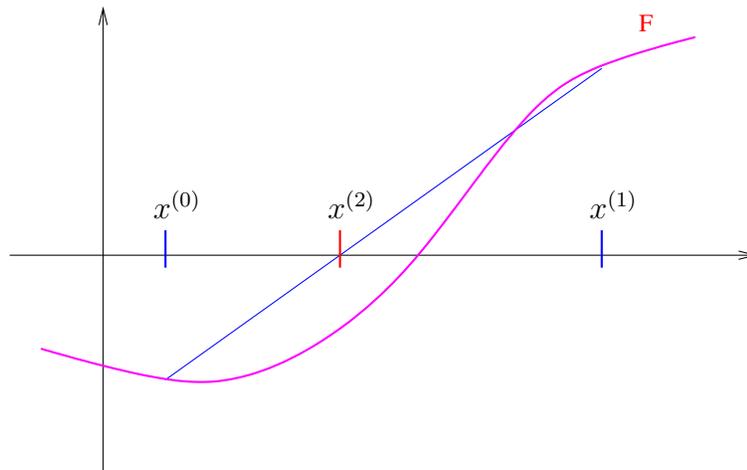


Abb. 3.6.1. Ein Schritt in Sekantenverfahren

Code 3.6.2: Sekantenverfahren

```

1 def secant(f,x0,x1,maxit=50,tol=1e-12):
2
3     fo = f(x0)
4     for k in range(maxit): #maxit: maximal number of iterations
5         fn = f(x1)
6         print('x1=',x1, 'f(x1)=', fn)
7         s = fn*(x1-x0)/(fn-fo) #estimate for derivative
8         x0 = x1; x1 -= s
9         if abs(s)<tol: #we are reasonably near to the solution and can return
the value we obtained
10            x = x1
11            return x, k #we also return the total number of iterations needed
12
13     fo = fn
14     x = NaN
15     return x, maxit

```

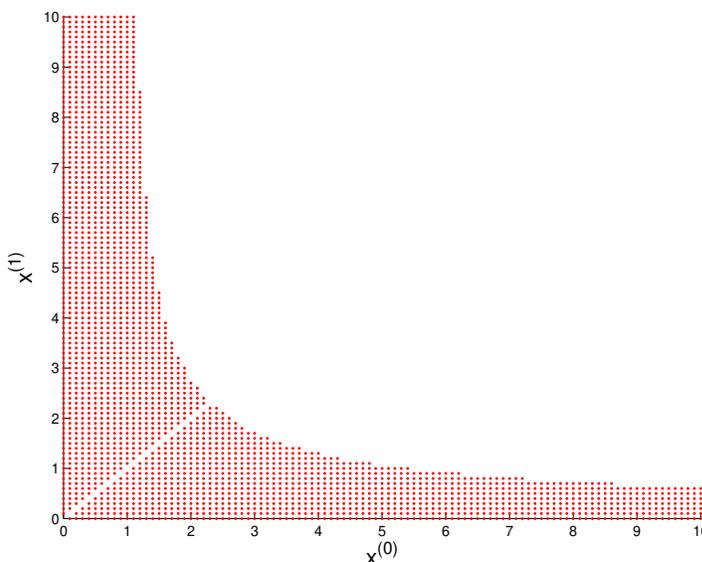
**Beispiel 3.6.3.**  $F(x) = xe^x - 1$ ,  $x^{(0)} = 0$ ,  $x^{(1)} = 5$ .

$k$	$x^{(k)}$	$F(x^{(k)})$	$e^{(k)} := x^{(k)} - x^*$	$\frac{\log  e^{(k+1)}  - \log  e^{(k)} }{\log  e^{(k)}  - \log  e^{(k-1)} }$
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	

Wir beobachten eine fraktionale Konvergenzordnung  $p$ ! Man kann  $p = \frac{1}{2}(1 + \sqrt{5}) \approx 1.62$  beweisen.

**Beispiel 3.6.4.** (Lokale Konvergenz des Sekantenverfahrens)

$$F(x) = \arctan(x)$$



**Abb. 3.6.5.**  $\hat{=}$  Sekantenverfahren konvergiert für ein Paar  $(x^{(0)}, x^{(1)})$  als Startwert.

### 3.7 Newton-Verfahren in $n$ Dimensionen

Sei  $F: D \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^n$  stetig differenzierbar. Gesucht ist  $x^* \in D$  mit  $F(x^*) = 0$ . Die Idee ist dieselbe wie in einer Dimension: man ersetzt  $F$  durch ein lineares  $\tilde{F}$ :

$$\tilde{F}^{(k)}(x) = F(x^{(k)}) + DF(x^{(k)})(x - x^{(k)}), \quad (3.7.1)$$

wobei  $DF(x^{(k)})$  die Jacobi-Matrix von  $F$  ist:  $DF(x) = \left[ \frac{\partial F_j}{\partial x_k}(x) \right]_{j,k=1,2,\dots,n}$ .

Die Nullstelle von  $\tilde{F}^{(k)}$  ist dann

$$x^{(k+1)} := x^{(k)} - DF(x^{(k)})^{-1}F(x^{(k)}), \quad (3.7.2)$$

falls  $DF(x^{(k)})$  invertierbar ist. Der Term  $DF(x^{(k)})^{-1}F(x^{(k)})$  heisst Newton-Korrektur und wird als Lösung des linearen Gleichungssystems  $DF(x^{(k)})s = F(x^{(k)})$  in jedem Schritt berechnet.

**Beachte:** Man berechnet **niemals** die Inverse einer Matrix:  $s = A^{-1}b$ , sondern löst das Gleichungssystem  $As = b$ .

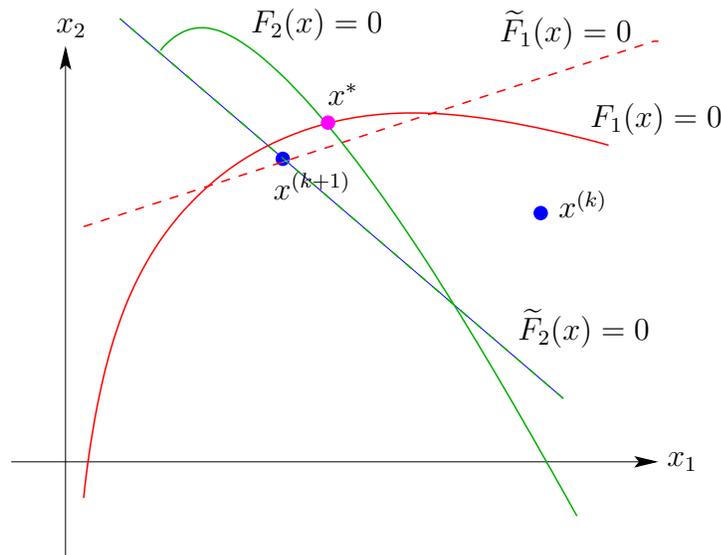


Abb. 3.7.1. Ein Schritt vom Newton-Verfahren in 2D

Code 3.7.2: Newtonverfahren

```

1 from numpy import atleast_2d
2 from numpy.linalg import solve, norm
3
4 def newton(x, F, DF, tol=1e-12, maxit=50):
5     """Newton Iteration
6     x:   Startwert der Iteration
7     F:   Funktion F(x)
8     DF:  Jacobi Matrix von F(x)
9     """
10    # 'solve' erwartet x als 2-dimensionaler numpy array
11    x = atleast_2d(x)
12    # Newton Iteration
13    for i in range(maxit):
14        s = solve(DF(x), F(x))
15        x -= s
16        if norm(s) < tol*norm(x):
17            return x

```

**Abbruchkriterium:**

Liegt  $x^{(k)}$  nah an der Lösung, so gilt

$$\|x^{(k)} - x^*\| \approx \|x^{(k)} - x^{(k+1)}\| = \|DF(x^{(k)})^{-1}F(x^{(k)})\| \leq \tau. \quad (3.7.3)$$

Da die Berechnung von  $DF(x^{(k)})^{-1}F(x^{(k)})$  aber teuer ist und wir  $DF(x^{(k)})^{-1}$  eventuell nicht mehr brauchen (im Falle eines Abbruchs), kann man auch  $DF(x^{(k)})^{-1} \approx DF(x^{(k-1)})^{-1}$  verwenden, da  $x^{(k)}$  ja nahe an der Lösung liegt. Somit erhält man folgendes Abbruchkriterium:

$$\|DF(x^{(k-1)})^{-1}F(x^{(k)})\| \leq \tau \implies \text{Abbruch.} \quad (3.7.4)$$

$DF(x^{(k-1)})^{-1}F(x^{(k)})$  heisst die vereinfachte Newton-Korrektur.

**Beispiel 3.7.3.** (Newton-Verfahren in 2D)

$$F(\mathbf{x}) = \begin{pmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \in \mathbb{R}^2 \quad \text{mit Lösung} \quad F \begin{pmatrix} 1 \\ 1 \end{pmatrix} = 0.$$

Die Jacobi-Matrix können wir analytisch berechnen:

$$DF(\mathbf{x}) = \begin{pmatrix} \partial_{x_1} F_1(x) & \partial_{x_2} F_1(x) \\ \partial_{x_1} F_2(x) & \partial_{x_2} F_2(x) \end{pmatrix} = \begin{pmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{pmatrix}.$$

Die Newton-Iteration (3.7.2) lautet nun:

1. Löse das lineare Gleichungssystem

$$\begin{pmatrix} 2x_1 & -4x_2^3 \\ 1 & -3x_2^2 \end{pmatrix} \Delta \mathbf{x}^{(k)} = \begin{pmatrix} x_1^2 - x_2^4 \\ x_1 - x_2^3 \end{pmatrix}.$$

wobei  $\mathbf{x}^{(k)} = (x_1, x_2)^T$ .

2. Setze  $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \Delta \mathbf{x}^{(k)}$ .

**Code 3.7.4:** Newton-Iteration in 2D

```

1  from numpy import array, diff, log, zeros, vstack
2  from scipy.linalg import norm, solve
3
4  F = lambda x: array([x[0]**2 - x[1]**4, x[0]-x[1]**3]) #the system we want to
5  DF = lambda x: array([[2*x[0], - 4*x[1]**3], [1, -3*x[1]**2]]) #its jacob
6  matrix
7
8  #x is the starting vector
9  x = array([0.7, 0.7])
10
11 #x0 is the exact solution
12 x0 = array([1., 1.])
13 tol = 1e-10
14
15 res = zeros(4); res[1:3] = x; res[3] = norm(x-x0)
16 #res is a vector containing:
17 #in its first entry, k (for the k-th iteration)
18 #in its second and third entry, the approximation of the solution found in the
19 #k-th step
20 #in its last entry, the norm of the difference between the k-th approximation an
21 the exact solution
22
23 print(DF(x))

```

```

21 print(F(x))
    s = solve(DF(x),F(x))
23 #important: we never calculate the inverse of a matrix, we solve a linear system
    instead
    x -= s
25 res1 = zeros(4); res1[0] = 1.; res1[1:3] = x; res1[3] = norm(x-x0)
    res = vstack((res,res1)) #[res,res1]
27 k = 2

29 #now we perform the newton method until we find a reasonably good solution
    while norm(s) > tol*norm(x):
31     s = solve(DF(x),F(x))
        x -= s
33     res1 = zeros(4); res1[0] = k; res1[1:3] = x; res1[3] = norm(x-x0)
        res = vstack((res,res1)) #[res0,res1,...,resk]
35     k += 1 #update the iteration number

37 logdiff = diff(log(res[:,3]))
    rates = logdiff[1:]/logdiff[:-1] #usual way of estimating the convergence rate
39
41 print(res)
    print(rates)

```

Die Ergebnisse sind:

$k$	$\mathbf{x}^{(k)}$	$\epsilon_k := \ \mathbf{x}^* - \mathbf{x}^{(k)}\ _2$
0	$(0.7, 0.7)^T$	4.24e-01
1	$(0.8785000000000000, 1.064285714285714)^T$	1.37e-01
2	$(1.01815943274188, 1.00914882463936)^T$	2.03e-02
3	$(1.00023355916300, 1.00015913936075)^T$	2.83e-04
4	$(1.00000000583852, 1.00000002726552)^T$	2.79e-08
5	$(0.999999999999998, 1.000000000000000)^T$	2.11e-15
6	$(1, 1)^T$	

Das Paket `sympy` kann bei der Berechnung der Ableitungen hilfreich sein:

**Code 3.7.5:** Newton-Iteration in 2D mit `sympy`

```

1 import numpy as np
  import scipy.optimize
3 from sympy import *
  import matplotlib.pyplot as plt

5
  x_, y_ = symbols('x y') # define symbols
7
  f = sin(pi*sqrt(x**2+y**2)) # define function f for sympy
9
  dfdx = diff(f, x_, 1) # its derivative wrt x
11 dfdy = diff(f, y_, 1) # its derivative wrt y

13 g = y_-x_**2 # define function g for sympy
    dgdx = diff(g, x_, 1) # its derivative wrt x

```

```
15 dgdy = diff(g, y_, 1) # its derivative wrt y
17 # make them useable with numpy:
18 f1 = lambdify((x_, y_), f, "numpy")
19 f2 = lambdify((x_, y_), g, "numpy")
21 # prepare 2D grid:
22 N=500
23 x = np.linspace(-3,3,N)
24 X, Y = np.meshgrid(x,x)
25 Z = f1(X, Y)
26 cutoff=5
27 # draw contour lines with value 0
28 CS =plt.contour(X, Y, Z, [0],linestyle='--', colors='b', label='Nullstellen von
29 $f_1$')
30 plt.gca().clabel(CS, inline=1, fontsize=10)
31 plt.contour(X, Y, f2(X,Y), [0], colors='g', label='Nullstellen von $f_2$')
32 plt.legend()
33 plt.grid(True)
34 plt.show()
35 # draw f1
36 plt.pcolormesh(X, Y, f1(X,Y), vmax=20, vmin=-20)
37 plt.colorbar()
38 plt.show()
39 # a variant of Newton
40 def newton(f, x0, df, tol=1e-5, N=1000):
41     error = np.linalg.norm(f(x0))
42     i=0
43     x=x0
44     while error > tol and i < N:
45         fprime = df(x)
46         fx = f(x)
47         x = x - np.linalg.solve(fprime, fx)
48         error = np.linalg.norm(f(x))
49         i+=1
50     return x
51
52 # make derivatoves useable in numpy
53 df1dx = lambdify((x_, y_), dfdx, 'numpy')
54 df1dy = lambdify((x_, y_), dfdy, 'numpy')
55 #
56 df2dx = lambdify((x_, y_), dgdx, 'numpy')
57 df2dy = lambdify((x_, y_), dgdy, 'numpy')
58 # build Jacobi matrix for w= [x,y]
59 dprime = lambda w: np.array([[df1dx(w[0], w[1]), df1dy(w[0], w[1])],
60                             [df2dx(w[0], w[1]), df2dy(w[0], w[1])]])
61
62 # build F for w = [x,y]
63 F = lambda w: np.array([f1(w[0], w[1]), f2(w[0], w[1])])
64 # failure
65 x = newton(F, np.array([1,1]), dprime)
66 print(x)
67 print(F(x))
```

```

# success
69 s = np.array([0.75,0.75])
   x = newton(F,s, dprime)
71 print(x)
   print(F(x))

```

**Bemerkung 3.7.6.** Im Falle vieler Dimensionen ( $n \gg 1$ ) ist die Berechnung einer Newton-Korrektur teuer, denn  $DF(x^{(k)})$  sind (volle)  $n \times n$  Matrizen.

**Bemerkung 3.7.7.** Das vereinfachte Newton-Verfahren verwendet  $DF(x^{(k-1)})$  für mehrere Schritte:

$$\Delta \bar{x}^{(k)} := DF(\mathbf{x}^{(k-1)})^{-1} F(\mathbf{x}^{(k)}).$$

Der Vorteil ist, dass in diesem Fall die LU-Zerlegung von  $DF(\mathbf{x}^{(k-1)})$  wiederverwendet werden kann, sodass die vereinfachte Newton-Korrektur  $\Delta \bar{x}^{(k)}$  in  $O(n^2)$  Operationen zur Verfügung steht. Der Preis dafür ist der Verlust der quadratischen Konvergenzordnung: das vereinfachte Newton-Verfahren konvergiert nur linear.

## 3.8 Gedämpftes Newton-Verfahren

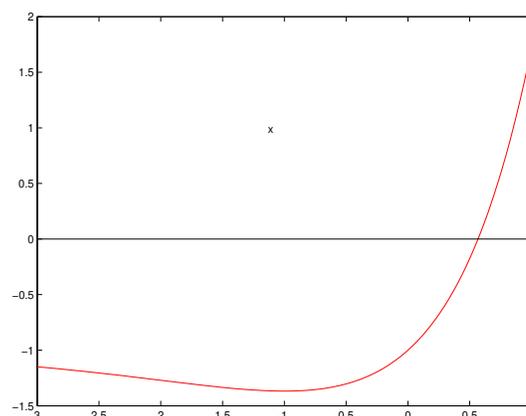
Das Newton-Verfahren stellt sich oft als gute Methode heraus; allerdings konvergiert es nicht immer. Oft hängt der Erfolg der Iteration von der Wahl des Anfangsparameter ab. Wir wollen dazu zwei Beispiele betrachten:

**Beispiel 3.8.1.** (Lokale Konvergenz vom Newton-Verfahren)

$$F(x) = xe^x - 1 \implies F'(-1) = 0$$

$$x^{(0)} < -1 \implies x^{(k)} \rightarrow -\infty$$

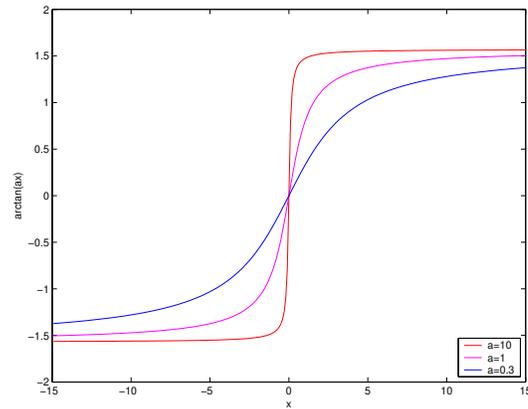
$$x^{(0)} > -1 \implies x^{(k)} \rightarrow x^*.$$



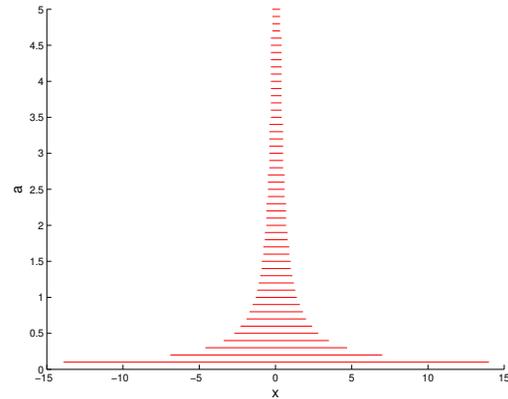
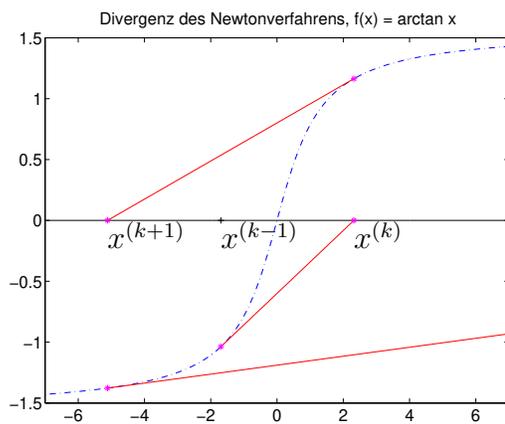
**Abb. 3.8.2.** Verhalten von  $F(x) = xe^x - 1$

**Beispiel 3.8.3.** (Konvergenzgebiet vom Newton-Verfahren)

$$F(x) = \arctan(ax), \quad a > 0, x \in \mathbb{R} \quad \text{mit der Nullstelle } x^* = 0.$$



**Abb. 3.8.4.** Verhalten von  $F(x) = \arctan(ax)$  für verschiedene Parameter  $a$ .



$$\text{Rote Zone} = \{x^{(0)} \in \mathbb{R}, x^{(k)} \rightarrow 0\}$$

**Abb. 3.8.5.** Wertebereich für den Startwert  $x^{(0)}$ , damit das Newton-Verfahren für  $F(x) = \arctan(ax)$  konvergiert

Sei  $F(x) = \arctan(x)$ . Wendet man das normale Newton-Verfahren auf  $F$  an, so konvergiert dieses nur für Startwerte im ungefähren Intervall  $[-1.39, 1.39]$ . Für Werte ausserhalb dieses Bereichs divergiert das Verfahren.

Wir beobachten, dass die Korrekturen zu lang sind. Wir dämpfen sie mit dem schrittabhängigen Faktor  $\lambda^{(k)}$ . Die Wahl von diesem Dämpfungsparameter  $\lambda^{(k)}$  ist heuristisch. Wir können zum Beispiel wie im Folgenden vorgehen:

Sei  $\Delta x^{(k)} = DF(x^{(k)})^{-1}F(x^{(k)})$  die ursprüngliche Newton-Korrektur und  $\Delta x(\lambda^{(k)}) = DF(x^{(k)})^{-1}F(x^{(k)}) - \lambda^{(k)}\Delta x^{(k)}$  eine provisorische Korrektur mit dem vereinfachten Newton-Verfahren. Wir wählen das grösste  $\lambda^{(k)} > 0$ , so dass

$$\|\Delta x(\lambda^{(k)})\|_2 \leq \left(1 - \frac{\lambda^{(k)}}{2}\right) \|\Delta x^{(k)}\|_2.$$

Dann ist  $x^{(k+1)} = x^{(k)} - \lambda^{(k)} DF(x^{(k)})^{-1}F(x^{(k)})$ .

### Code 3.8.6: Gedämpftes Newton-Verfahren

```

1 from scipy.linalg import lu_solve, lu_factor, norm
2 from numpy import array, arctan, exp
4 def dampnewton(x,F,DF,q=0.5,tol=1e-10):
6     cvg = [] #container storing, for each iteration, lambda, xn and F(xn)
7     lup = lu_factor(DF(x)) #LU decomposition
8     s = lu_solve(lup,F(x)) #lusolve is more efficient
9     xn = x-s
10    lam = 1
11    st = lu_solve(lup,F(xn))
12
13    while norm(st) > tol*norm(xn): #a posteriori termination criteria
14        while norm(st) > (1-lam*0.5)*norm(s): #natural monotonicity test
15            lam *= 0.5 # reduce damping factor
16            if lam < 1e-10:
17                cvg = -1
18                print('DAMPED NEWTON: Failure of convergence')
19                return x, cvg
20            xn = x-lam*s #simplified Newton
21            st = lu_solve(lup,F(xn))
22            cvg += [[lam, norm(xn), norm(F(xn))]]
23            x = xn
24            lup = lu_factor(DF(x))
25            s = lu_solve(lup,F(x))
26            lam = min(lam/q, 1.) #notice that we must have 1 - lambda/2 > 0
27            xn = x-lam*s #damped newton verfahren
28            st = lu_solve(lup,F(xn)) #simplified Newton, prepare st for the next
iteration
30    x = xn
31    return x, array(cvg)
32
#now we test the damped newton method in three different cases

```

```

34 if __name__=='__main__':
    print('----- 2D F -----')
36 F = lambda x: array([x[0]**2 - x[1]**4, x[0]-x[1]**3])
    DF = lambda x: array([[2*x[0], - 4*x[1]**3], [1, -3*x[1]**2]])
38 x = array([0.7, 0.7])
    x0 = array([1., 1.])
40 x, cvg = dampnewton(x,F,DF)
    print(x)
42 print(cvg)

44 print('----- arctan -----')
    F = lambda x: array([arctan(x[0])])
46 DF = lambda x: array([[1./(1.+x[0]**2)])]
    x = array([20.])
48 x, cvg = dampnewton(x,F,DF)
    print(x)
50 print(cvg)

52 print('-----x e^x - 1 -----')
    F = lambda x: array([x[0]*exp(x[0])-1.])
54 DF = lambda x: array([[exp(x[0])*(x[0]+1.)]])
    x = array([-1.5])
56 x, cvg = dampnewton(x,F,DF)
    print(x)
58 print(cvg)

```

**Beispiel 3.8.7.** (Gedämpftes Newton-Verfahren)(Siehe Beispiel 3.8.3):  $F(x) = \arctan(x)$ :

- $x^{(0)} = 20$
- $q = \frac{1}{2}$
- LMIN = 0.001

**Beobachtung:** asymptotisch quadratische Konvergenz.

$k$	$\lambda^{(k)}$	$x^{(k)}$	$F(x^{(k)})$
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

**Beispiel 3.8.8.** (Gedämpftes Newton-Verfahren funktioniert nicht immer)

- Wie im Beispiel 3.8.1:  $F(x) = xe^x - 1$ ,
- Wir nehmen als Startwert für das gedämpfte Newton-Verfahren:  $x^{(0)} = -1.5$ :

$k$	$\lambda^{(k)}$	$x^{(k)}$	$F(x^{(k)})$
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 <code>lambda &lt; LMIN</code> !			

**Beobachtung:** Die Newton-Korrektur zeigt in die “falsche Richtung”, also wir haben keine Konvergenz.

### 3.9 Das Quasi-Newton Verfahren

Was machen wir, falls  $DF(x)$  zu teuer ist um es zu berechnen oder falls es gar nicht zur Verfügung steht? Für eindimensionale Probleme haben wir das Sekantenverfahren eingeführt, wo man den Differenzenquotienten statt  $F'(x^k)$  verwendet:

$$F'(x^{(k)}) \approx \frac{F(x^{(k)}) - F(x^{(k-1)})}{x^{(k)} - x^{(k-1)}} = J_k. \quad (3.9.1)$$

Dabei wurden  $F(x^{(k-1)})$  und  $F(x^{(k)})$  bereits berechnet und somit stehen sie günstig zur Verfügung.

Was sollen wir im mehrdimensionalen Fall tun? Da wir durch einen Vektor nicht teilen können, werden wir die Gleichung (3.9.1) in

$$J_k \cdot (x^{(k)} - x^{(k-1)}) = F(x^{(k)}) - F(x^{(k-1)}) \quad (3.9.2)$$

umschreiben. Im eindimensionalen Fall ( $n = 1$ ) ist  $J_k$  eine durch (3.9.2) eindeutig definierte Zahl. Für  $n \geq 2$  gibt es aber viele  $n \times n$  Matrizen  $J_k$ , die (3.9.2) erfüllen.

Hilfreich ist es, wenn sich  $J_k$  durch eine einfache Änderung aus  $J_{k-1}$  ergibt. Die Broyden-Bedingung dazu ist:

$J_k z = J_{k-1} z$  für alle  $z$ , die orthogonal zu  $x^{(k)} - x^{(k-1)}$  sind, d.h.  $J_k$  und  $J_{k-1}$  wirken in gleicher Weise auf den Raum, der orthogonal zur vorigen Korrektur ist. Somit gilt, dass:

$$J_k := J_{k-1} + \frac{F(x^{(k)})(x^{(k)} - x^{(k-1)})^T}{\|x^{(k)} - x^{(k-1)}\|_2^2}.$$

Das ergibt die Broyden-Quasi-Newton Methode:

$$\begin{aligned} \Delta x^{(k)} &:= -J_k^{-1} F(x^{(k)}) \\ x^{(k+1)} &:= x^{(k)} + \Delta x^{(k)} \\ J_{k+1} &:= J_k + \frac{F(x^{(k+1)})(\Delta x^{(k)})^T}{\|\Delta x^{(k)}\|_2^2}. \end{aligned} \tag{3.9.3}$$

Dabei müssen wir  $J_0$  z.B. durch  $DF(x^{(0)})$  definieren.

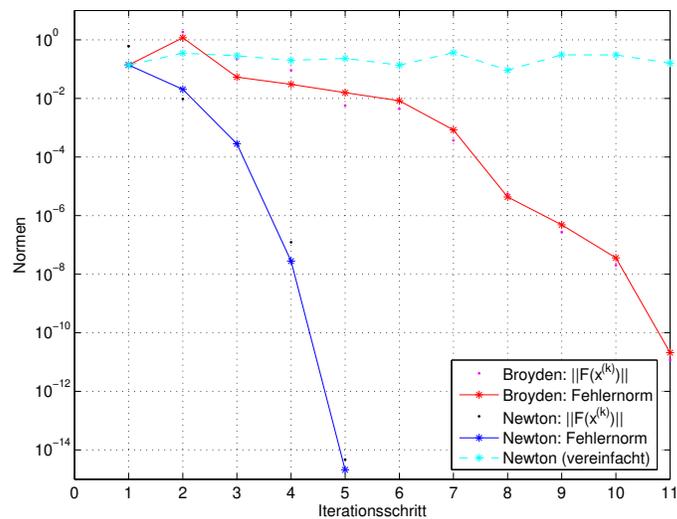
**Bemerkung 3.9.1.** Das Broyden-Update von  $J_k$  (3.9.3) gibt die bezüglich der  $\|\cdot\|_2$ -Norm minimale relative Korrektur zu  $J_k$ , sodass die Sekantenbedingung (3.9.2) erfüllt ist.

Die Implementierung des Broydens-Verfahrens beruht auf der Sherman-Morrison-Woodbury Update-Formel (die wir hier nicht beweisen wollen):

$$J_{k+1}^{-1} = \left( I - \frac{J_k^{-1} F(x^{(k+1)})(\Delta x^{(k)})^T}{\|\Delta x^{(k)}\|_2^2 + (\Delta x^{(k)})^T J_k^{-1} F(x^{(k+1)})} \right) J_k^{-1}.$$

**Beispiel 3.9.2.** (Broyden-Quasi-Newton-Verfahren: Konvergenz)

Wir betrachten das numerische Beispiel 3.7.3, also  $n = 2$ ; wir starten mit  $x^{(0)} = (0.7, 0.7)^T$  und  $J_0 = DF(x^{(0)})$ .



**Abb. 3.9.3.** Fehler in Broyden-, Newton- und vereinfachtes Newton-Verfahren.

Die Ergebnisse zeigen: diese Methode konvergiert langsamer als das Newton-Verfahren (3.7.2), aber schneller als das vereinfachte Newton-Verfahren.

#### Code 3.9.4: Broyden-Verfahren

```

1  from scipy.linalg import lu_solve, lu_factor, norm, solve
2  from numpy import dot, zeros, zeros_like
3
4  def fastbroyd(x0, F, J, tol=1e-12, maxit=20):
5
6      x = x0.copy()
7      lup = lu_factor(J) #LU decomposition of J
8
9      k = 0; s = lu_solve(lup,F(x)) #linear system can be solved faster with LU
10     deco x -= s; f = F(x); sn = dot(s,s) #step, sn is the squared norm of the
11     correction
12
13     #containers for storing s and sn:
14     dx = zeros((maxit,len(x)))
15     dxn = zeros(maxit)
16     dx[k] = s; dxn[k] = sn
17     k += 1; #k is the number of the iteration
18
19     w = lu_solve(lup,f) #f = F(starting value) (see above)
20
21     #now we perform the iteration
22
23     f = F(x)
24
25     while sn > tol and k < maxit:
26         w = lu_solve(lup,f) #f = F(starting value) (see above)
27         #now we update the correction for the k-th step
28         #using the sherman morrison woodbury formel
29         for r in range(1,k):
30             w += dx[r]*( dot(dx[r-1],w) )/dxn[r-1]
31         z = dot(s,w)
32         s = (1+z/(sn-z))*w
33         sn = dot(s,s)
34         dx[k] = s; dxn[k] = sn
35         x -= s; f = F(x); k+=1 #update x and iteration number k
36
37     return x, k #return the final value and the numbers of iterations needed

```

**Beispiel 3.9.5.** (Broyden-Verfahren für ein grosses, nicht-lineares System)

$$F(\mathbf{x}) = \begin{cases} \mathbb{R}^n & \mapsto \mathbb{R}^n \\ \mathbf{x} & \mapsto \text{diag}(\mathbf{x})\mathbf{A}\mathbf{x} - \mathbf{b} , \end{cases}$$

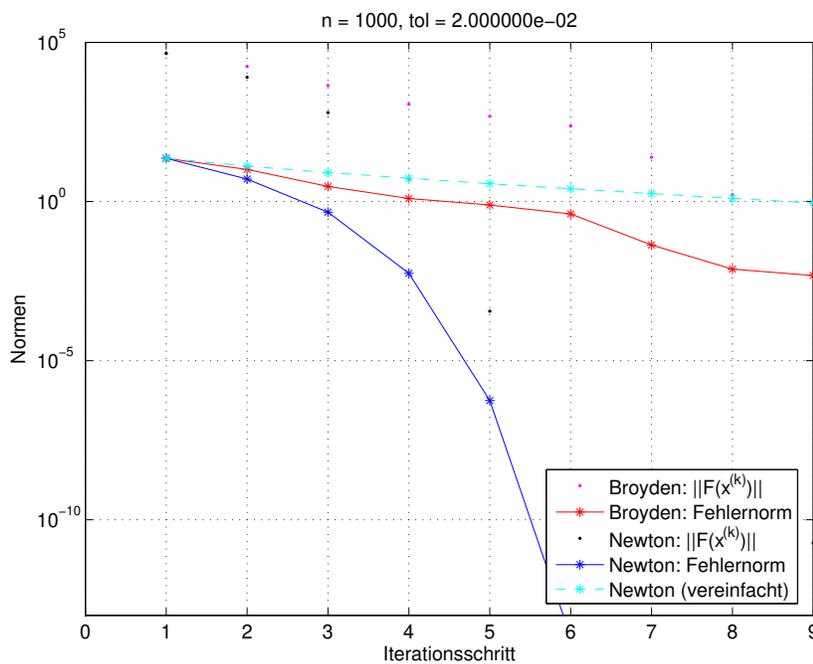
$$\mathbf{b} = (1, 2, \dots, n)^T \in \mathbb{R}^n ,$$

$$\mathbf{1} = (1, 1, \dots, 1)^T \in \mathbb{R}^n ,$$

$$\mathbf{A} = \mathbf{I} + \mathbf{a}\mathbf{a}^T \in \mathbb{R}^{n,n} ,$$

$$\mathbf{a} = \frac{1}{\sqrt{\mathbf{1} \cdot (\mathbf{b} - \mathbf{1})}}(\mathbf{b} - \mathbf{1}) .$$

Als Startwert nehmen wir den Vektor  $x^0 = [2, 2 + h, \dots, 4 - h]$ , wobei  $h = 2/n$  ist. Die Ergebnisse sind den Resultaten vom Beispiel 3.9.2 ähnlich:



**Abb. 3.9.6.** Fehler in Broyden-, Newton- und vereinfachtes Newton-Verfahren.

Schlussendlich können wir die Effizienz der beiden Verfahren vergleichen. Hier haben wir  $\text{tol} = 2n \cdot 10^{-5}$  gewählt.

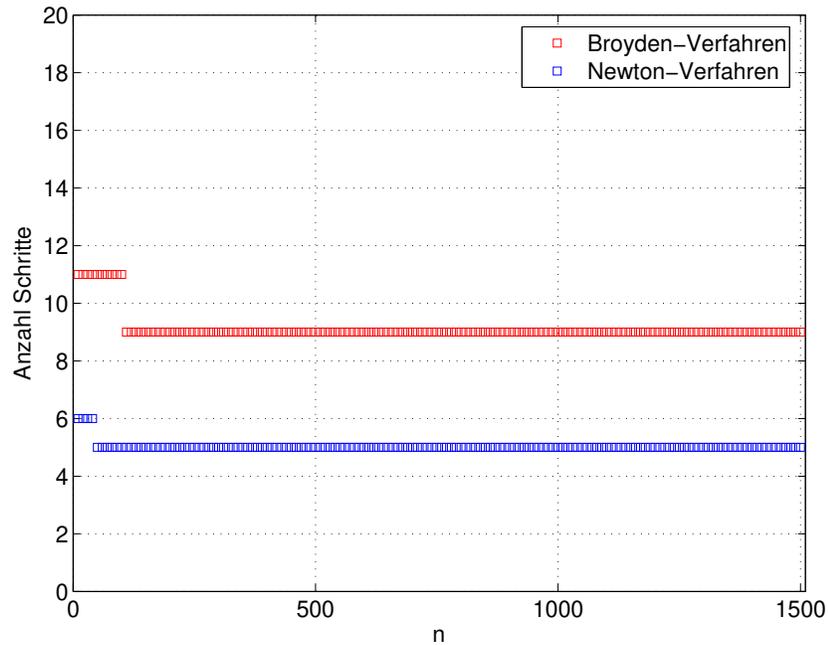


Abb. 3.9.7. Anzahl der nötigen Schritte in Broyden- und Newton-Verfahren.

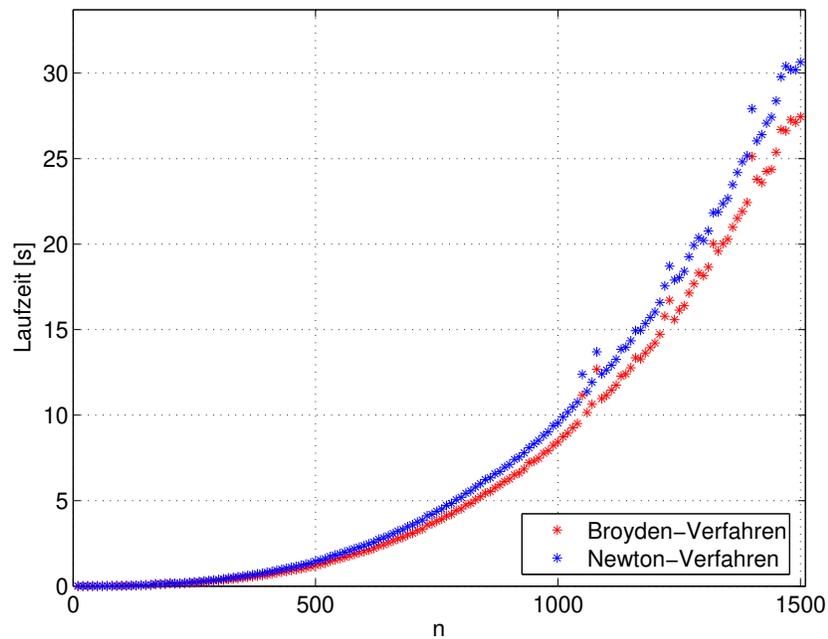


Abb. 3.9.8. Laufzeiten der Broyden- und Newton-Verfahren

Das Broyden-Verfahren lohnt sich für grosse Dimensionen  $n \gg 1$  und bescheidene Genauigkeitsanforderungen.

# Chapter 4

## Integratoren für steife Differentialgleichungen

### 4.1 Stabilitätsanalyse der RK-Verfahren

Wir haben im vorigen Kapitel gesehen, dass explizite Runge-Kutta-Verfahren einfach zu implementieren sind und, zusammen mit einer adaptiven Wahl der Zeitschrittweite, sehr genaue numerische Approximationen der Lösungen gewöhnlicher Differentialgleichungen liefern. Trotzdem sind diese nicht blind anzuwenden; hierzu betrachten wir folgendes Beispiel.

**Beispiel 4.1.1.** (ode45 für steife Differentialgleichungen)

$$\text{AWP: } \quad \dot{y} = \lambda y^2(1 - y), \quad \lambda := 500, \quad y(0) = \frac{1}{100}.$$

**Code 4.1.2:** ode45 für steife Differentialgleichungen

```
import matplotlib.pyplot as plt
2 from numpy import diff, finfo, double
3 from ode45 import ode45
4
5 def ode45stiff():
6     """tries to solve the logistic differential equation using ode45 with
7     adaptive stepsize control and draws the results:
8     resulting data points -> logode451.eps
9     timestep size         -> logode452.eps
10    both plots also contain the 'exact' solution
11    """
12
13    # define the differential equation
14    fun = lambda t,x: 500*x*(1-x)
15    # define the time span
16    tspan = (0.0,1.0)
17    # get total time span
18    L = diff(tspan)
```

```

# starting value
20 y0 = 0.01

# make the figure
22 fig = plt.figure()

# exact solution
# set options
24
26 eps = finfo(double).eps
options = {'reltol':10*eps,'abstol':eps,'stats':'on'}
28 tex,yex = ode45(fun,(0,1),y0,**options)
# plot 'exact' results
30 plt.plot(tex,yex,'--',color=[0,0.75,0],linewidth=2,label='$y(t)$')
32 plt.hold(True)

# ODE45
# set options
34
36 options = {'reltol':0.01,'abstol':0.001,'stats':'on'}
# get the solution using ode45
38 [t,y] = ode45(fun,(0,1),y0,**options)
# plot the solution
40 plt.plot(t,y,'r+',label='ode45')

# set label, legend, ....
42 plt.xlabel('t',fontSize=14)
44 plt.ylabel('y')
plt.title('ode45 for d_ty = 500 y(1-y)')
46 plt.show()
# write to file
48 # plt.savefig('../PICTURES/logode451.eps')

# now plot stepsizes
50 fig = plt.figure()
52 plt.title('ode45 for d_ty = 500 y(1-y)')
ax1 = fig.add_subplot(111)
54 ax1.plot(tex,yex,'r--',linewidth=2)
ax1.set_xlabel('t')
56 ax1.set_ylabel('y(t)')

58 ax2 = ax1.twinx()
ax2.plot(0.5*(t[:-1]+t[1:]),diff(t),'r-')
60 ax2.set_ylabel('stepsize')

62 plt.show()

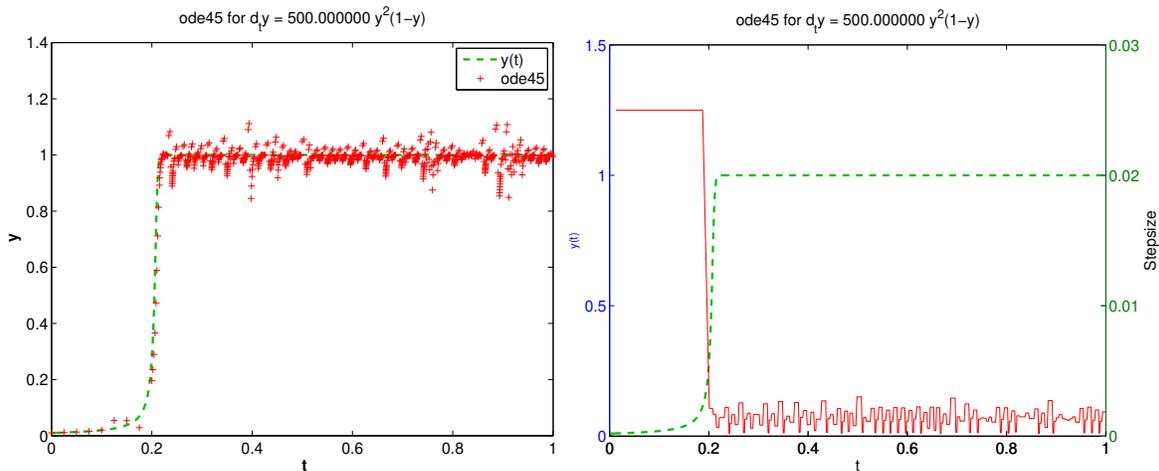
# write to file
64 #plt.savefig('../PICTURES/logode452.eps')

66
68 if __name__ == '__main__':
    print('warning: this function takes a long time to complete! (not really
worth it)\n')
    ode45stiff()

```

Die Option 'stats': 'on' lässt ode45 Statistiken über den Verlauf der Integration zurückgeben.

```
Number of successful steps:
2119
Number of failed attempts: 3
Number of function calls:
12726
```



**Abb. 4.1.3.** Die Zeitschrittkontrolle von ode45 macht die Integration unnötig teuer!

Obwohl die Lösung ab der Zeit  $t > 0.2$  fast konstant ist, wählt der Integrator hinter ode45 immer winzige Zeitschritte bis ans Ende der Integrationszeit  $T$ . Diese Situation ist höchst unbefriedigend, denn das bedeutet eine Verschwendung der Rechenzeit. Aus diesem Grund kann man für viele praktische Probleme nicht die Lösung an interessanten Zeiten mithilfe von ode45 approximieren, denn der Zeitschritt kann unter Umständen sehr klein werden, sodass das Verfahren “auf der Stelle tritt”.

Um diesem Phänomen auf die Spur zu kommen, betrachten wir zunächst das Modellproblem:

$$\dot{y} = \lambda y \tag{4.1.1}$$

mit der Konstanten  $\lambda < 0$ . Die Lösung ist

$$y(t) = e^{\lambda t} y(0),$$

und  $\lim_{t \rightarrow \infty} y(t) = 0$ . Hierbei ist die Konvergenz umso schneller je grösser  $|\lambda|$  ist. Wir werden uns gleich mit der Frage befassen, wann erfüllt eine numerische Lösung  $(\Psi_{\lambda}^{kh} y_0)_{k=0}^{\infty}$  diese Eigenschaft auch?

**Bemerkung 4.1.4.** Dieses Modellproblem ist relevant, denn durch Linearisierung um einen Fixpunkt erhält man immer eine lineare Differentialgleichung. Zum Beispiel, für die logistische Differentialgleichung

$$\dot{y} = f(y) \quad \text{mit} \quad f(y) = \lambda y(1 - y)$$

erhalten wir durch Linearisierung um 1:

$$\dot{y} = -\lambda(y - 1),$$

da  $\frac{df}{dy}(1) = -\lambda$ , und mit der Notation  $z = y - 1$  haben wir es in der Tat mit dem Modellproblem zu tun.  $\square$

Betrachten wir nun die numerische Lösung, die von dem expliziten Euler-Verfahren geliefert wird:

$$y_1 = y_0 + \lambda h y_0 = (1 + \lambda h) y_0,$$

also

$$y_k = (1 - |\lambda|h)^k y_0.$$

Diese numerische Lösung  $y_k$  konvergiert gegen 0 nur dann, wenn  $|1 - |\lambda|h| < 1$ , das heisst nur falls eine relativ kleine Zeitschrittweite

$$h < \frac{2}{|\lambda|}$$

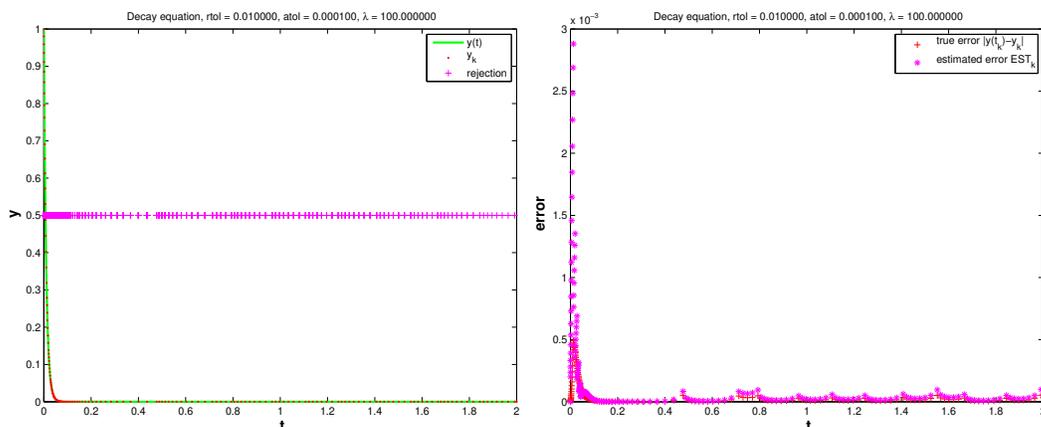
gewählt worden ist.

Wenn  $h > \frac{2}{|\lambda|}$ , explodiert sogar die numerische Lösung nach langen Zeiten.

Es stellt sich nun die Frage: was macht ein adaptives Verfahren, wenn  $|\lambda|$  gross ist? Dazu wollen wir folgendes Beispiel betrachten.

**Beispiel 4.1.5.** (Einfache Zeitschrittweitensteuerung für den schnellen radioaktiven Zerfall)

- Lineares Modell:  $\dot{y} = \lambda y$ ,  $y(0) = 1$ ,  $\lambda = -100$ .
- (Einfache Zeitschrittweitensteuerung wie in Ex. Theorem 2.6.3, see Code 2.6.2



**Abb. 4.1.6.** Zeitschrittweitensteuerung bei radioaktivem Abfall

**Beobachtung:**

Die Zeitschrittweitensteuerung zwingt kleine Zeitschritte auf, auch wenn die Lösung  $\mathbf{y}(t) \approx 0$  nicht mehr interessant ist. Grössere Zeitschrittweiten werden immer abgelehnt, damit der Euler-Schritt, welcher in der Fehlerschätzung eine wesentliche Rolle spielt, im Stabilitätsbereich bleibt, d.h. nicht zu den oben besprochenen grossen Werten führt:  $h < \frac{1}{50}$ .

Geschieht dasselbe auch mit anderen Methoden? Betrachten wir die explizite Trapez-Regel:

$$k_1 = \lambda y_0$$

$$k_2 = \lambda(y_0 + k_1 h)$$

ergibt

$$y_1 = \left(1 + \lambda h + \frac{1}{2} (\lambda h)^2\right) y_0 =: S(\lambda h) y_0.$$

Somit:

$$y_n(t) = \Psi_\lambda^t y_0 = \Psi_1^{(\lambda t)} y_0 = S(\lambda h)^n y_0.$$

Das qualitative Abklingen der numerischen Lösung passiert nur für Zeitschrittweiten  $h$ , die

$$|S(\lambda h)| < 1$$

erfüllen, was wiederum  $h < \frac{2}{|\lambda|}$  bedeutet.

**Definition 4.1.7.** (Stabilitätsgebiet eines Einschrittverfahrens)

Das Stabilitätsgebiet eines ESV für das AWP auf der Grundlage der diskreten Evolution  $\Psi_\lambda^h \mathbf{y} =: S(z) \mathbf{y}$  mit  $z := \lambda h \in \mathbb{C}$  und  $S: D_S \subset \mathbb{C} \mapsto \mathbb{C}$ , ist

$$S_\Psi := \{z \in D_S : |S(z)| < 1\} \subset \mathbb{C}.$$

**Theorem 4.1.8.** (Stabilitätsfunktion von Runge-Kutta-Verfahren)

Der diskrete Evolutionsoperator  $\Psi_\lambda^h$  zu einem  $s$ -stufigen Runge-Kutta-Einschrittverfahren mit Butcher-Schema  $\begin{array}{c|c} \mathbf{c} & \mathbf{U} \\ \hline & \mathbf{b}^T \end{array}$  (für die ODE  $\dot{\mathbf{y}} = \lambda \mathbf{y}$  mit  $\lambda \in \mathbb{C}$ ) ist ein Multiplikationsoperator der Form:

$$\Psi_\lambda^h = \underbrace{1 + z \mathbf{b}^T (\mathbf{I} - z \mathbf{U})^{-1} \mathbf{1}}_{\text{Stabilitätsfunktion } S(z)} = \frac{\det(\mathbf{I} - z \mathbf{U} + z \mathbf{1} \mathbf{b}^T)}{\det(\mathbf{I} - z \mathbf{U})}, \quad z := \lambda h, \quad \mathbf{1} = (1, \dots, 1)^T \in \mathbb{R}^s.$$

**Bemerkung 4.1.9.** Für explizite Runge-Kutta-Verfahren ist  $S(z)$  ein Polynom vom Grad  $s$ , während wir es für allgemeine Runge-Kutta-Verfahren mit einer rationalen Funktion zu tun haben.

**Bemerkung 4.1.10.** Die Stabilitätsfunktion  $S(z)$  ist eine Approximation der Exponentialfunktion, denn  $y_n = S(\lambda h)^n y_0 \approx e^{n\lambda h} y_0$ .

**Beispiel 4.1.11.** (Stabilitätsfunktionen einiger RK-ESV)

- Explizites Euler-Verfahren:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \quad \text{hat} \quad S(z) = 1 + z.$$

- Implizites Euler-Verfahren:

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \quad \text{hat} \quad S(z) = \frac{1}{1 - z}.$$

- Explizite Trapezregel:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} \quad \text{hat} \quad S(z) = 1 + z + \frac{1}{2} z^2.$$

- Implizite Mittelpunktsregel:

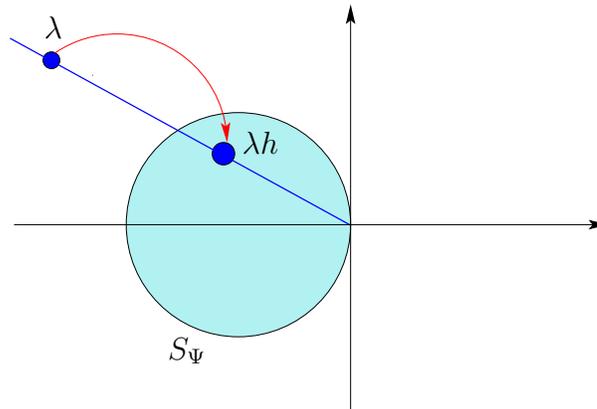
$$\begin{array}{c|c} \frac{1}{2} & \frac{1}{2} \\ \hline & 1 \end{array} \quad \text{hat} \quad S(z) = \frac{1 + \frac{1}{2} z}{1 - \frac{1}{2} z}.$$

- RK4-Verfahren:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array} \quad \text{hat} \quad S(z) = 1 + z + \frac{1}{2} z^2 + \frac{1}{6} z^3 + \frac{1}{24} z^4.$$

Die Stabilität  $y_n(t) \rightarrow 0$  ist also nur dann erfüllt, wenn  $|S(\lambda h)| < 1$ , was eventuell für eine Beschränkung der Zeitschrittweite  $h$  sorgt. Bei expliziten RK-Verfahren ist  $S$  ein Polynom und  $\lim_{|z| \rightarrow \infty} |S(z)| = \infty$ , was eine Beschränkung der Zeitschrittweite nach sich zieht:

$$h < \sup\{t > 0; t\lambda \in S_\Psi\}.$$



**Abb. 4.1.12.** Der Stabilitätsgebiet eines expliziten Verfahrens ist beschränkt, was eine Einschränkung von  $h$  zwingt.

**Bemerkung 4.1.13.** Damit ein exponentieller Zuwachs in der numerischen Lösung (was für  $\dot{y} = \lambda y$  bei  $\lambda < 0$  qualitativ falsch ist) vermieden wird, müssen wir bei einem expliziten RK-Verfahren sicherstellen, dass  $|\lambda h|$  klein ist. Somit erzwingt die Stabilität die Wahl eines kleinen Zeitschrittes auch dann, wenn die Genauigkeit es gar nicht verlangt! Das führt zu sehr ineffizienter numerischer Integration, wie in den obigen Beispielen gezeigt. Die Adaptivität entdeckt die Instabilität und reduziert die Zeitschrittweite auf Kosten der Rechenzeit.

**Beispiel 4.1.14.** Bei einem System von Differentialgleichungen kommen gleichzeitig mehrere Konstanten  $\lambda$  ins Spiel. Sei

$$\dot{y} = \underbrace{\begin{bmatrix} 0 & 1 \\ -\beta & -\alpha \end{bmatrix}}_M y \quad \text{mit } \beta \gg \frac{1}{4} \alpha^2 \gg 1$$

Die Diagonalisierung der Matrix  $M$  erlaubt es uns, die exakte Lösung zu finden:

$$\dot{y} = My; \quad MV = M[v_1 \ v_2] = [v_1 \ v_2] \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix}$$

$$y(t) = c_1 v_1 e^{\lambda_1 t} + c_2 v_2 e^{\lambda_2 t} \quad \text{mit } c_1, c_2 \in \mathbb{R} \text{ und}$$

$$\lambda_{1,2} = \frac{1}{2} \alpha \pm i \sqrt{\beta - \frac{1}{4} \alpha^2}.$$

Wenn  $\operatorname{Re} \lambda < 0$  dann  $|y(t)| = |e^{\lambda t} y(0)| = e^{t \operatorname{Re} \lambda} |y_0| \rightarrow 0$ . Somit ist es klar: bei einem  $\operatorname{Re} \lambda \ll -1$  brauchen explizite RK-Verfahren sehr kleine Schritte.

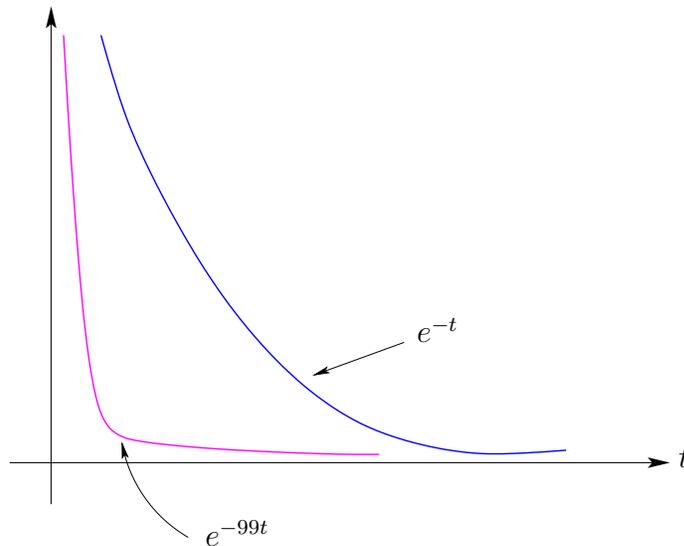
**Beispiel 4.1.15.** Wir wollen den oberen Sachverhalt numerisch untersuchen; sei

$$\frac{d}{dt} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -50 & 49 \\ 49 & -50 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

mit der exakten Lösung

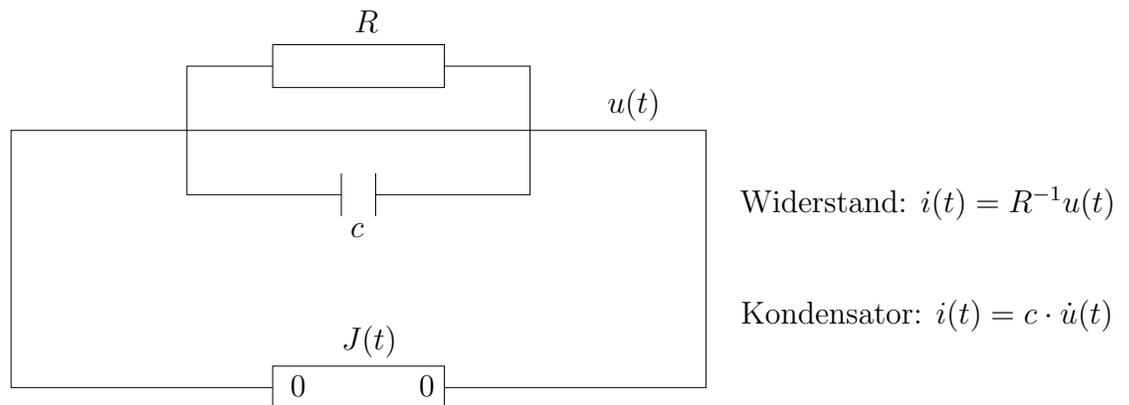
$$\begin{cases} y_1(t) = e^{-t} + e^{-99t} \\ y_2(t) = e^{-t} - e^{-99t} \end{cases}.$$

Ein explizites Verfahren verlangt  $99h \leq c$ , also  $h \leq \frac{c}{99}$ :



**Abb. 4.1.16.** Die schnell abfallende Komponente verlangt eine kleine Zeitschrittweite, obwohl die deutlich interessantere (nicht so schnell abfallende) Komponente mit einer grösseren Zeitschrittweite gut approximiert werden kann.

**Beispiel 4.1.17.** Schauen wir uns nun ein komplexeres Beispiel an:



**Abb. 4.1.18.** Steife Schaltkreisgleichungen im Zeitbereich

Die Knotenanalyse ergibt

$$c\dot{u}(t) = -\frac{1}{R}u(t) + I(t)$$

$$C = 1\text{pF}, \quad R = 1\text{k}\Omega, \quad I(t) = \sin(2\pi \text{1Hz}t) \text{ mA}$$

$$u(0) = 0 \text{ V},$$

was wir in eine dimensionslose Gleichung umschreiben:

$$\dot{u}(t) = -10^9 u(t) + 10^9 \sin(\pi t).$$

Die Lösung ist fast eine Schwingung

$$u(t) \approx \sin(2\pi t).$$

Es handelt sich um eine einfache lineare Gleichung mit Quellterm:

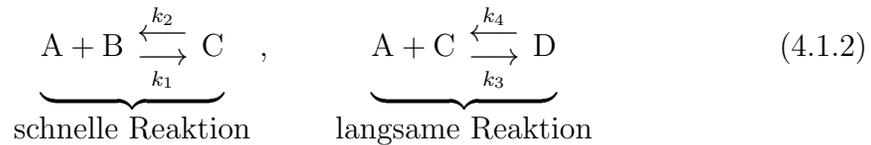
$$\dot{y} = -\lambda y + g(t), \quad \lambda \gg 1,$$

mit dem ‘Fixpunkt’  $y^*(t) = \frac{1}{\lambda} g(t)$ . Ein explizites Verfahren wird ein winziges  $h$  aufzwingen, obwohl der wesentliche Teil der Lösung es nicht braucht.

**Definition 4.1.19.** Eine gewöhnliche Differentialgleichung heisst *steif*, falls ein explizites Verfahren einen kleinen Zeitschritt braucht um die Lösung vernünftig zu approximieren.

**Bemerkung 4.1.20.** Explizite RKs sind bei steifen Problemen zu vermeiden; für steife Differentialgleichungen sollte man implizite Verfahren verwenden!

**Beispiel 4.1.21.** (Reaktionskinetik) Gegeben sei die Reaktion



wobei die zwei Teile sehr verschiedene Reaktionskonstanten aufweisen:

$$k_1, k_2 \gg k_3, k_4.$$

Falls  $c_A(0) > c_B(0)$ , dann bestimmt der zweite Teil die Langzeitdynamik der Reaktion.

Das mathematische Modell ist ein System gewöhnlicher Differentialgleichungen für die Konzentrationen  $\mathbf{y}(t) = (c_A(t), c_B(t), c_C(t), c_D(t))^T$

$$\dot{\mathbf{y}} := \frac{d}{dt} \begin{pmatrix} c_A \\ c_B \\ c_C \\ c_D \end{pmatrix} = \mathbf{f}(\mathbf{y}) := \begin{pmatrix} -k_1 c_A c_B + k_2 c_C - k_3 c_A c_C + k_4 c_D \\ -k_1 c_A c_B + k_2 c_C \\ k_1 c_A c_B - k_2 c_C - k_3 c_A c_C + k_4 c_D \\ k_3 c_A c_C - k_4 c_D \end{pmatrix}.$$

In den Simulationen nehmen wir  $t_0 = 0$ ,  $T = 1$ ,  $k_1 = 10^4$ ,  $k_2 = 10^3$ ,  $k_3 = 10$ ,  $k_4 = 1$  an.

## Code 4.1.22: Explicit Integration of Stiff Equations of Chemical reactions

```

1 from numpy import array, linspace, double, diff
2 from ode45 import ode45
3 import matplotlib.pyplot as plt
4 from odewrapper import odewrapper

6 def chemstiff():
7     """Simulation of kinetics of coupled chemical reactions with vastly
8     different reaction
9     rates, see \eqref{eq:chemstiff} for the ODE model.
10    reaction rates  $\{k_1, k_2, k_3, k_4\}$ , \Magenta $\{k_1, k_2\}$  \gg  $k_3, k_4$ .
11    """

12    #the constants involed in our differential equation
13    k1 = 1e4; k2 = 1e3; k3 = 10; k4 = 1

14    # definition of right hand side function for ODE solver
15    fun = lambda t,y: array([-k1*y[0]*y[1] + k2*y[2] - k3*y[0]*y[2] + k4*y[3],
16                            -k1*y[0]*y[1] + k2*y[2],
17                            k1*y[0]*y[1] - k2*y[2] - k3*y[0]*y[2] + k4*y[3],
18                            k3*y[0]*y[2] - k4*y[3]])

19    tspan = [0, 1] # Integration time interval
20    L = tspan[1]-tspan[0] # Duration of simulation
21    y0 = array([1,1,10,0]) # Initial value  $y_0$ 

22    # compute the ‘exact’ solution, using fortran routines (using wrapper for
23    scipy.integrate.ode class)
24    tex = linspace(0,1,201)
25    yex = odewrapper(fun,y0,0,tex)

26    # Compute solution with ode45 and moderate tolerances
27    options = {'RelTol':0.01, 'AbsTol':0.001, 'Stats':'on'}
28    t,y = ode45(fun,[0, 1],y0,**options)

29    # plot the solution
30    plt.figure()
31    plt.plot(t,y[:,0],'b.', label=r'$c_{A,k}$, low tolerance')
32    plt.plot(t,y[:,2],'r.', label=r'$c_{C,k}$, low tolerance')

33    # plot the 'exact' results
34    plt.plot(tex,yex[:,0],'c--',linewidth=3, label=r'$c_A(t)$')
35    plt.plot(tex,yex[:,2],'m--',linewidth=3, label=r'$c_C(t)$')
36    plt.axis([0,1,0,12])
37    plt.xlabel('t')
38    plt.ylabel('concentrations')

39    #plt.title('Chemical reaction: concentrations')
40    plt.legend()
41    #plt.savefig('../PICTURES/chemstiff.eps')

42    # Plot stepsizes together with ‘exact’ solution
43    fig = plt.figure()
44    #plt.title('Chemical reaction: stepsize')

```

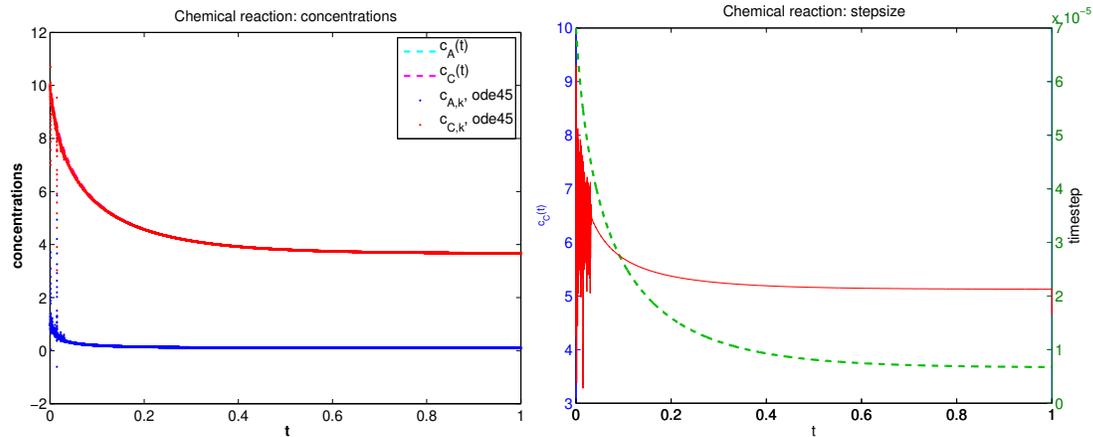
```

52 ax1 = fig.add_subplot(111)
ax1.plot(tex,yex[:,2], 'r--', linewidth=2)
ax1.set_xlabel('t')
54 ax1.set_ylabel('c_C(t)')

56 ax2 = ax1.twinx()
ax2.plot(0.5*(t[:-1]+t[1:]),diff(t), 'r-')
58 ax2.set_ylabel('timestep')

60 #plt.savefig('../PICTURES/chemstiffss.eps')
plt.show()

64 if __name__ == '__main__':
    chemstiff()
    
```



**Abb. 4.1.23.** Konzentrationen von  $A$  und  $C$  und die nötigen Zeitschrittweiten zusammen mit  $c_C$

Wie man sehen kann ist ODE45 ziemlich ineffizient und liefert ziemlich ungenaue Ergebnisse.

**Beispiel 4.1.24** (Attraktiver Grenzyzyklus). Gegeben sei die autonome Differentialgleichung

$$\mathbf{f}(\mathbf{y}) := \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2) \mathbf{y},$$

auf  $D = \mathbb{R}^2 \setminus \{0\}$ . Für  $\lambda = 10$  sehen wir die Trajektorien in der Abbildung 4.1.25.

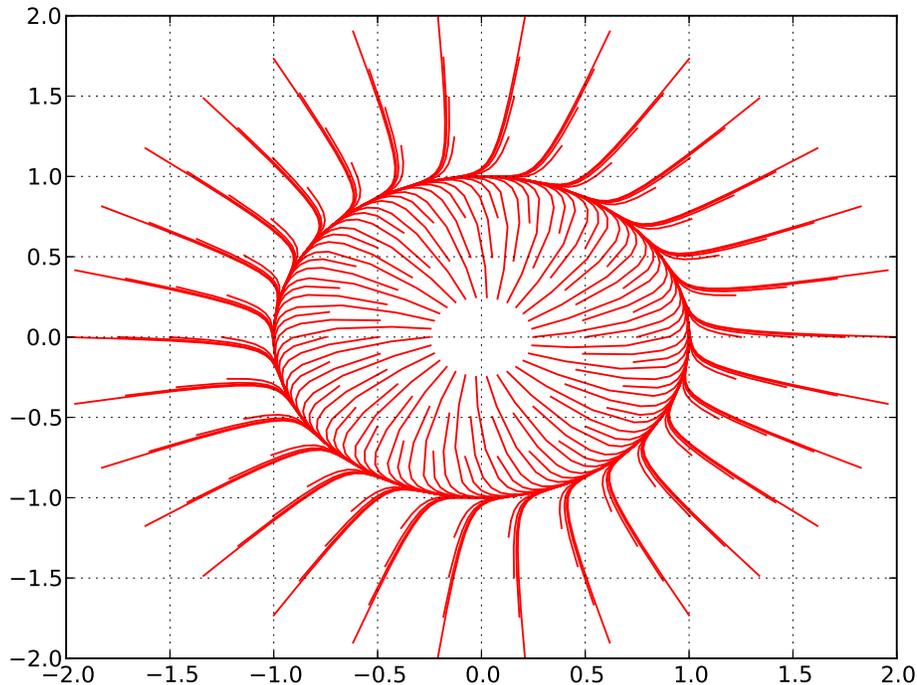


Abb. 4.1.25. Alle Trajektorien führen auf lange Zeit auf einem Kreis = attraktiver Grenzzyklus.

Code 4.1.26: Integration of IVP with limit cycle

```

1 from ode45 import ode45
2 from numpy import *
3 import matplotlib.pyplot as plt
4
5 def limitcycle(lam,y0=array((1,0))):
6     #lam ist  $\lambda$ , the parameter which we want to vary in order to see what effects
7     it produces on the solution of the ODE
8     # PYTHON script for solving limit cycle ODE
9     # define right hand side vectorfield
10    fun = lambda t,y: array([-y[1],y[0]]) + lam*(1-y[0]**2-y[1]**2)*y
11    # standard invocation our integrator
12    tspan = (0,2*pi)
13    #opts: options for ODE45
14    opts = {'stats':True,'reltol':1e-4,'abstol':1e-4}
15    t45,y45 = ode45(fun,tspan,y0,**opts)
16    return (t45,y45)
17
18 if __name__ == '__main__':
19     for lam in [0,1000]:
20         #for  $\lambda = 0$  and  $\lambda = 1000$  we solve the differential equation numerically
21         t,y = limitcycle(lam)
22         #plot the numerical solution

```

```

22     fig = plt.figure()
23     plt.title(r'\lambda = ' + str(lam)+ '$')
24     ax1 = fig.add_subplot(111)
25     lineObjects = ax1.plot(t,y)
26     ax1.legend(lineObjects, ('$y_1$', '$y_2$'),loc=4)
27     ax1.set_ylabel(r'$y_i(t)$')
28     ax1.set_xlabel('$t$')
29     #ax1.legend(loc=4)
30     ax2 = ax1.twinx()
31     if lam == 1000:
32         ax2.set_ylim([0,2*10**-3])
33         ax2.set_autoscaley_on(False)
34     ax2.plot(0.5*(t[:-1]+t[1:]),diff(t), 'r-', label='$h$')
35     ax2.set_ylabel('$h$')
36     ax2.legend(loc=3)
37     #if lam == 0 : plt.savefig('../PICTURES/rigidmotionode45.eps')
38     #else: plt.savefig('../PICTURES/limitcycleode45.eps')
39     plt.show()
40
41     # now plot some nice solution trajectories
42     lam = 10
43     plt.figure()
44     #plt.title(r'Solution Trajectories (\lambda = 10)')
45     #we use radial coordinate (radiusc = rad and angle = phi) to find suitable
initial values
46     for rad in r_[0.25:2.25:0.25]: #array([0.25, 0.5, 0.75, 1., 1.25, 1.5, 1.75,
2.])
47         for phi in r_[0:2*pi:31j][:-1]:
48             x,y = rad*cos(phi),rad*sin(phi)
49             t,y = limitcycle(lam,array([x,y]))
50             plt.plot(y[:,0],y[:,1], 'r')
51     plt.grid(True)
52     # plt.savefig('../PICTURES/limitcyclclerhs.eps')
53     plt.show()

```

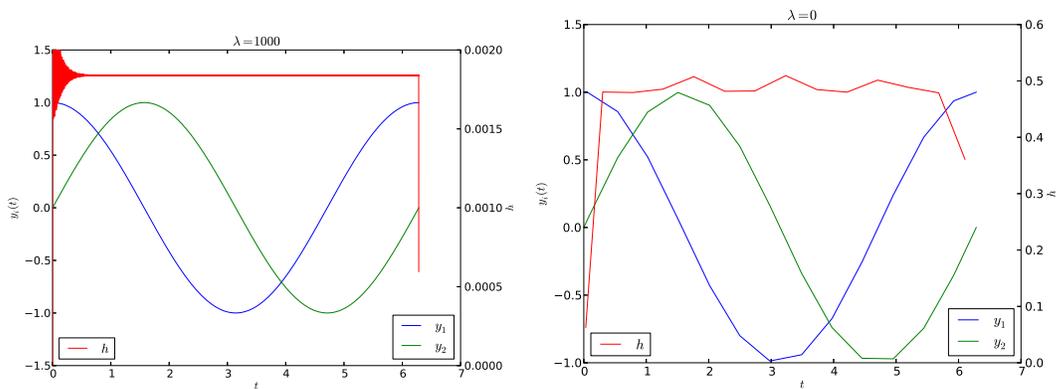


Abb. 4.1.27.  $\lambda = 1000$  braucht über 3000 Schritte im Unterschied zu  $\lambda = 0$

**Definition 4.1.28.** Ein Verfahren heisst  $A$ -stabil, falls die (ganze) linke komplexe Ebene im Stabilitätsgebiet des Verfahrens enthalten ist:

$$\mathbb{C}^- \subset S_\Psi.$$

**Bemerkung 4.1.29.** Da  $S(z) \approx e^z$ , ist auch  $S(-\infty) = 0$  wünschenswert.

**Definition 4.1.30.** Eine Methode heisst  $L$ -stabil falls sie  $A$ -stabil ist und  $S(-\infty) = 0$ .

**Bemerkung 4.1.31.** Man kann zeigen: wenn  $b^T = a_{.j}^T$  ( $j$ -te Zeile von  $A$ ) für ein RK-Verfahren mit dem Butcher-Tableau

$$\frac{c \mid a}{\mid b^T},$$

dann ist das Verfahren  $L$ -stabil.

Deswegen wählt man  $c_s = 1$  und  $c_1, \dots, c_{s-1}$  als Knoten einer Quadraturformel maximaler Ordnung  $s - 1$  (also einer Gauss-Radau-Quadratur) und man erhält dann implizite  $K$ -stufige  $L$ -stabile RK-Verfahren (kurz Radau ESV) der Ordnung  $2s-1$ :

**Beispiel 4.1.32.** ( $L$ -stabile implizite Runge-Kutta-Verfahren)

$$\frac{1 \mid 1}{\mid 1} \qquad \frac{\frac{1}{3} \mid \frac{5}{12} \quad -\frac{1}{12}}{\mid 1 \quad \frac{3}{4} \quad \frac{1}{4}} \qquad \frac{\frac{1}{3} \mid \frac{5}{12} \quad -\frac{1}{12}}{\mid \frac{3}{4} \quad \frac{1}{4}}$$

Implizites Euler-ESV

Radau-ESV, Ordnung 3

$$\frac{\frac{4 - \sqrt{6}}{10} \mid \frac{88 - 7\sqrt{6}}{360} \quad \frac{296 - 169\sqrt{6}}{1800} \quad \frac{-2 + 3\sqrt{6}}{225}}{\frac{4 + \sqrt{6}}{10} \mid \frac{296 + 169\sqrt{6}}{1800} \quad \frac{88 + 7\sqrt{6}}{360} \quad \frac{-2 - 3\sqrt{6}}{225}}{1 \mid \frac{16 - \sqrt{6}}{36} \quad \frac{16 + \sqrt{6}}{36} \quad \frac{1}{9}} \qquad \frac{\frac{16 - \sqrt{6}}{36} \mid \frac{16 + \sqrt{6}}{36} \quad \frac{1}{9}}{\mid \frac{16 - \sqrt{6}}{36} \quad \frac{16 + \sqrt{6}}{36} \quad \frac{1}{9}}$$

Radau-ESV, Ordnung 5

## 4.2 Linear-implizite RK-Verfahren

Ein implizites RK-Verfahren mit  $s$  Stufen verlangt für jeden Zeitschritt die Lösung eines nicht-linearen Systems der Dimension  $s \cdot d$  (für die  $s$  Inkremente  $k$ ).

**Beispiel 4.2.1.** Implizites Euler-Verfahren: für  $\dot{y} = f(y)$  ist

$$y_{k+1} = y_k + h f(y_{k+1}),$$

also müssen wir

$$F(y_{k+1}) = 0$$

lösen mit  $F(z) = z - h f(z) - y_k$ . Das kann sehr teuer sein; billiger ist es, nur einen einzigen Newton-Schritt für  $F(y_{k+1}) = 0$  mit dem Startwert  $y_k$  durchzuführen:

$$y_{k+1} = y_k - DF(y_k)^{-1} F(y_k),$$

also:

$$y_{k+1} = y_k + [I - h Df(y_k)]^{-1} h f(y_k),$$

was *semi-implizites Euler-Verfahren* heisst.

**Beispiel 4.2.2.** Eine ähnliche Idee ist es, die Inkrementgleichungen zu linearisieren

$$\mathbf{k}_i = \mathbf{f}(\mathbf{y}_{(1)}) + h D\mathbf{f}(\mathbf{y}_{(1)}) \left( \sum_{j=1}^s a_{ij} \mathbf{k}_j \right) \text{ für } i = 1, \dots, s,$$

was ein lineares Gleichungssystem der Dimension  $d \cdot s$  liefert.

**Beispiel 4.2.3.** (Implizite RK-ESV mit linearisierten Inkrementgleichungen)

- Anfangswertproblem für logistische Differentialgleichung:

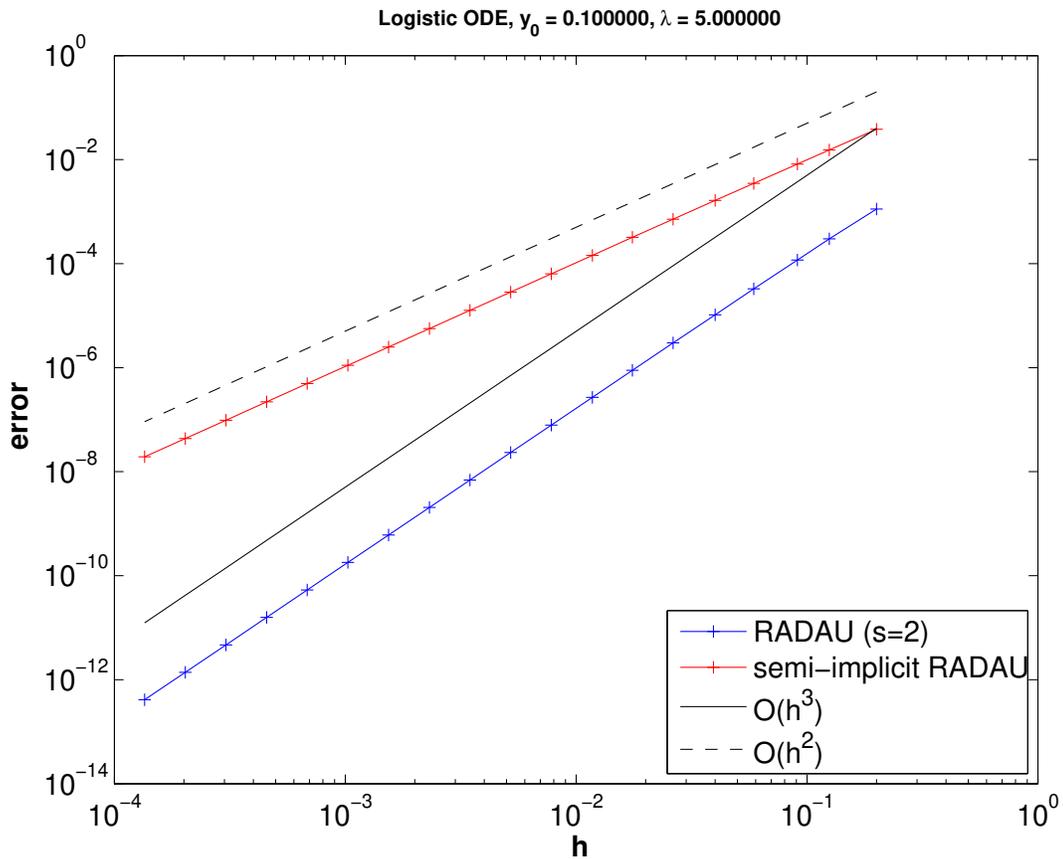
$$\dot{y} = \lambda y(1 - y), \quad y(0) = 0, 1, \quad \lambda = 5.$$

- 2-stufiges Radau-ESV, Butcher Schema:

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ \frac{1}{4} & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array},$$

Ordnung 3.

- Inkremente aus linearisierten Gleichungen.
- Fehlermass  $\text{err} = \max_{j=1, \dots, n} |y_j - y(t_j)|$



Wir beobachten, dass die Konvergenzordnung sich verschlechtert, wenn wir diese Linearisierung durchführen.

Wir können die Ordnung retten, wenn wir eine bessere Startnäherung für den (einzelnen) Newtonschritt haben. Dafür betrachten wir die diagonal-impliziten RK-Verfahren (DIRK):

$$\begin{array}{c|c}
 \mathbf{c} & \mathbf{U} \\
 \hline
 & \mathbf{b}^T
 \end{array}
 :=
 \begin{array}{c|cccc}
 c_1 & a_{11} & 0 & \cdots & 0 \\
 c_2 & a_{21} & a_{22} & 0 & 0 \\
 \vdots & \vdots & & \ddots & \vdots \\
 \vdots & \vdots & & & \vdots \\
 \vdots & \vdots & & & 0 \\
 c_s & a_{s1} & \cdots & & a_{ss} \\
 \hline
 & b_1 & \cdots & \cdots & b_s
 \end{array}
 .$$

In diesem Fall haben wir ein gestaffeltes nicht-lineares Gleichungssystem für die

Inkremente:

$$\begin{cases} \mathbf{k}_i = \mathbf{f}\left(\mathbf{y}_0 + h \sum_{j=1}^i a_{ij} \mathbf{k}_j\right) \\ \mathbf{y}_1 = \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i \end{cases} .$$

Für jeden Zeitschritt haben wir ein Nullstellenproblem für die  $i$ -te Inkrementgleichung

$$F(\mathbf{k}) := \mathbf{k} - \mathbf{f}(\mathbf{y}_0 + \mathbf{z} + h a_{ii} \mathbf{k})$$

mit  $\mathbf{z} = h \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j$ ,

mit der Ableitung

$$DF(\mathbf{k}) = \mathbf{I} - D\mathbf{f}(\mathbf{y}_0 + \mathbf{z} + h a_{ii} \mathbf{k}) h a_{ii} .$$

Ein Newton-Schritt mit Startwert  $\mathbf{k}_i^{(0)}$  ist dann

$$\mathbf{k}_i^{(1)} = \mathbf{k}_i^{(0)} - [\mathbf{I} - D\mathbf{f}(\mathbf{y}_0 + \mathbf{z} + h a_{ii} \mathbf{k}_i^{(0)}) h a_{ii}]^{-1} \cdot [\mathbf{k}_i^{(0)} - \mathbf{f}(\mathbf{y}_0 + \mathbf{z} + h a_{ii} \mathbf{k}_i^{(0)})] .$$

Nun machen wir einen allgemeinen Ansatz für den Startwert  $\mathbf{k}^{(0)}$  des Newton-Verfahrens:

$$\mathbf{k}_i^{(0)} = \sum_{j=1}^{i-1} \frac{d_{ij}}{a_{ii}} \mathbf{k}_j .$$

Dann müssen wir noch

$$(\mathbf{I} - h a_{ii} \mathbf{J}) \mathbf{k}_i = \mathbf{f}(\mathbf{y}_0 + h \sum_{j=1}^{i-1} (a_{ij} + d_{ij}) \mathbf{k}_j) - h \mathbf{J} \sum_{j=1}^{i-1} d_{ij} \mathbf{k}_j$$

lösen, wobei

$$\mathbf{J} := D\mathbf{f}\left(\mathbf{y}_0 + h \sum_{j=1}^{i-1} (a_{ij} + d_{ij}) \mathbf{k}_j\right) .$$

Wenn wir  $\mathbf{J} \approx \tilde{\mathbf{J}} := D\mathbf{f}(\mathbf{y}_0)$  approximieren, haben wir ein vereinfachtes Newton-Verfahren (mit eingefrorener Jakobi-Matrix).

Nun kann man  $a_{ij}$ ,  $d_{ij}$  (und  $b_i$ ) bestimmen, sodass die gewünschte Ordnung erreicht wird. Die Methoden, die auf diese Weise entwickelt werden, heissen *linear-implizite Runge-Kutta-Verfahren* oder *Rosenbrock-Wanner (ROW)-Methoden*, und werden in der leicht veränderten Form geschrieben ( $a_{ij} = a$  und sonst  $a_{ij}$  zweckmässig gewählt):

$$(\mathbf{I} - h a \mathbf{J}) \mathbf{k}_i = \mathbf{f}(\mathbf{y}_0 + h \sum_{j=1}^{i-1} a_{ij} \mathbf{k}_j) - h \mathbf{J} \sum_{j=1}^{i-1} d_{ij} \mathbf{k}_j \quad (4.2.3)$$

für  $i = 1, \dots, s$  und

$$\mathbf{y}_1 = \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i.$$

**Beispiel 4.2.4.** (Linear-implizite MPR)

$$\mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{k} \text{ mit } \left[ \mathbf{I} - \frac{h}{2} D_y \mathbf{f}(\mathbf{y}_i) \right] \mathbf{k} = \mathbf{f}(\mathbf{y}_i).$$

**Beispiel 4.2.5.** Für steife Differentialgleichungen werden oft die Routinen `ode15s` und `ode23s` verwendet. Ein python-wrapper zu `ode15s` erhält man mit

```
scipy.integrate.ode(f).set_integrator('vode').
```

`ode15s` ist ein impliziter adaptiver (Ordnungen 1 und 5) Integrator, der aber nicht zu der Klasse der ROW-Methoden Integratoren gehört. `ode23s` ist ein adaptives Verfahren wie `ode45`, das auf folgenden ROW-Methoden der Ordnungen 2 und 3 basiert (zur Zeit noch nicht in `scipy` implementiert).

Das ROW-Verfahren der Ordnung 2 ist:

$$\begin{cases} [\mathbf{I} - ah\mathbf{J}] \mathbf{k}_1 = \mathbf{f}(\mathbf{y}_i) \\ [\mathbf{I} - ah\mathbf{J}] \mathbf{k}_2 = \mathbf{f}\left(\mathbf{y}_i + h \frac{1}{2} \mathbf{k}_1\right) - ah\mathbf{J}\mathbf{k}_1 \\ \mathbf{y}_{i+1} = \mathbf{y}_i + h\mathbf{k}_2 \end{cases}$$

mit  $a = \frac{1}{2+\sqrt{2}}$ . In den Notationen von 4.2.3 ist  $b_1 = 0$ ,  $b_2 = 1$ ,  $d_{21} = a$ ,  $a_{21} = \frac{1}{2}$ .

Das ROW-Verfahren der Ordnung 3 ist:

$$\begin{cases} [\mathbf{I} - ah\mathbf{J}] \mathbf{k}_1 = \mathbf{f}(\mathbf{y}_i) \\ [\mathbf{I} - ah\mathbf{J}] \mathbf{k}_2 = \mathbf{f}\left(\mathbf{y}_i + h \frac{1}{2} \mathbf{k}_1\right) - ah\mathbf{J}\mathbf{k}_1 \\ [\mathbf{I} - ah\mathbf{J}] \mathbf{k}_3 = \mathbf{f}(\mathbf{y}_i + h\mathbf{k}_2) - d_{31} h\mathbf{J}\mathbf{k}_1 - d_{32} h\mathbf{J}\mathbf{k}_2 \\ \mathbf{y}_{i+1} = \mathbf{y}_i + \frac{h}{6} (\mathbf{k}_1 + 4\mathbf{k}_2 + \mathbf{k}_3) \end{cases}$$

mit  $a = \frac{1}{2+\sqrt{2}}$ ,  $d_{31} = -\frac{4+\sqrt{2}}{2+\sqrt{2}}$ ,  $d_{32} = \frac{6+\sqrt{2}}{2+\sqrt{2}}$  und  $\mathbf{J} = D_y \mathbf{f}(\mathbf{y}_i)$ .

### 4.3 Exponentielle Integratoren

Sei

$$(ODE) \quad \dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad \text{mit} \quad \mathbf{f} \in C^1.$$

Wir linearisieren die ODE:  $\dot{\mathbf{y}} = \mathbf{J}\mathbf{y}$  mit  $\mathbf{D} = D\mathbf{f}(y_0)$  und schreiben sie um:

$$(ODE) \quad \dot{\mathbf{y}} = \mathbf{J}\mathbf{y} + (\mathbf{f}(\mathbf{y}) - \mathbf{J}\mathbf{y})$$

oder

$$\dot{\mathbf{y}} = \mathbf{J}\mathbf{y} + \mathbf{g}(\mathbf{y}),$$

wobei  $\mathbf{g}(y) = \mathbf{f}(y) - \mathbf{J}y$ .

Durch Variation der Konstanten erhalten wir dann:

$$\mathbf{y}(h) = e^{\mathbf{J}h}\mathbf{y}_0 + \int_0^h e^{\mathbf{J}(h-z)}\mathbf{g}(\mathbf{y}(z)) dz. \quad (4.3.4)$$

**Bemerkung 4.3.1.**  $e^M = \sum_{n=0}^{\infty} \frac{1}{n!} M^n$  (Definition), aber es wird nicht so berechnet!  $e^M$  lässt sich mittels der Padé-Approximation `expm` oder Schur-Zerlegung

$$M = Q^T(D + U)Q$$

oder für grosse, dünnbesetzte Matrizen via Krylov-Verfahren, berechnen.

Die Standard-Routinen sind in FORTRAN- oder C-Code im Packet EXPOKIT implementiert. In `scipy` gibt es Wrappers dazu:

`expm` für Padé, `expm2` für Schur-Zerlegung, und es gibt effiziente Krylov-Verfahren, die von einem RW/CSE-Studenten an der ETH, Robert Gantner, implementiert worden sind: [Arnoldi\\_PyCpp](#). Auch `expm` wurde in [Pade\\_PyCpp](#) beschleunigt.

Für das Integral in (Gleichung (4.3.4)) ergibt jede Wahl der Quadratur ein anderes Verfahren; die einfachste Wahl ist:

$$\int_0^h e^{\mathbf{J}(h-\tau)}\mathbf{g}(\mathbf{y}(\tau))d\tau \approx \int_0^h e^{\mathbf{J}(h-\tau)}\mathbf{g}(\mathbf{y}_0)d\tau = h\varphi(h\mathbf{J})\mathbf{g}(\mathbf{y}_0)$$

mit

$$\varphi(z) = \frac{e^z - 1}{z} = \sum_{n=1}^{\infty} \frac{1}{n!} z^{n-1} = \sum_{n=0}^{\infty} \frac{z^n}{(n+1)!}.$$

In der Tat:

$$\begin{aligned} \int_0^h e^{\mathbf{J}(h-\tau)}\mathbf{g}(\mathbf{y}_0)d\tau &= \int_0^h \sum_{n=0}^{\infty} \frac{1}{n!} (\mathbf{J}(h-\tau))^n \mathbf{g}(\mathbf{y}_0)d\tau = \\ &= \sum_{n=0}^{\infty} \int_0^h \frac{1}{n!} (\mathbf{J}(h-\tau))^n \mathbf{g}(\mathbf{y}_0)d\tau = \\ &= \sum_{n=0}^{\infty} \frac{1}{n!} \mathbf{J}^n (-1) \frac{(h-\tau)^{n+1}}{n+1} \Big|_0^h \mathbf{g}(\mathbf{y}_0) = \\ &= \sum_{n=0}^{\infty} \frac{h}{(n+1)!} \mathbf{J}^n h^n \mathbf{g}(\mathbf{y}_0) = h\varphi(h\mathbf{J})\mathbf{g}(\mathbf{y}_0). \end{aligned}$$

Nun aber:  $\mathbf{g}(\mathbf{y}_0) = \mathbf{f}(\mathbf{y}_0) - \mathbf{J}\mathbf{y}_0$ . Durch Multiplikation mit  $h\varphi(h\mathbf{J})$  erhalten wir somit:

$$h\varphi(h\mathbf{J})\mathbf{g}(\mathbf{y}_0) = h\varphi(h\mathbf{J})\mathbf{f}(\mathbf{y}_0) - h\varphi(h\mathbf{J})\mathbf{J}\mathbf{y}_0.$$

Es gilt allerdings, dass:

$$h\varphi(h\mathbf{J})\mathbf{J}\mathbf{y}_0 = h \sum_{n=0}^{\infty} \frac{h^n}{(n+1)!} \mathbf{J}^{n+1}\mathbf{y}_0 = e^{\mathbf{J}h}\mathbf{y}_0 - \mathbf{y}_0,$$

also

$$\mathbf{y}(h) \approx e^{\mathbf{J}h}\mathbf{y}_0 + h\varphi(h\mathbf{J})\mathbf{g}(\mathbf{y}_0) = e^{\mathbf{J}h}\mathbf{y}_0 + h\varphi(h\mathbf{J})\mathbf{f}(\mathbf{y}_0) - e^{\mathbf{J}h}\mathbf{y}_0 + \mathbf{y}_0$$

und somit:

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h_k \varphi(h_k \mathbf{J}) \mathbf{f}(\mathbf{y}_k) \text{ für } k = 0, 1, \dots, N \text{ und } \mathbf{J} = D\mathbf{f}(\mathbf{y}_k).$$

Diese Methode heisst exponentielles Rosenbrock-Euler-Verfahren und ist  $\mathcal{O}(h^2)$ . Falls die Jacobi-Matrix konstant  $\mathbf{J} = D\mathbf{f}(\mathbf{y}_0)$  in allen Schritten beibehalten wird, was für grosse Systeme Rechenzeit spart, reden wir über das exponentielle Euler-Verfahren und wir verlieren eine Konvergenzordnung.

**Bemerkung 4.3.2.** 1) Dieses Verfahren ist exakt für

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} + \mathbf{g} \text{ mit } \mathbf{A} \in \mathbb{R}^{d \times d}, \mathbf{g} \in \mathbb{R}^d \text{ konstant.}$$

- 2) Die Stabilitätsfunktion ist  $S(z) = e^z$ , das heisst wir haben das ideale Stabilitätsgebiet  $S_{\Psi} = \mathbb{C}^-$ .
- 3) Exponentielle Verfahren brauchen die Jacobi-Matrix der rechten Seite, wie die impliziten Methoden auch, aber im Unterschied zu diesen muss hier kein nicht-lineares Nullstellenproblem gelöst werden. Das teuerste hier ist die Auswertung der Wirkung des Operators  $\varphi(h\mathbf{J})$  auf einen Vektor, was mittels des Krylov-Verfahrens günstig für grosse dünnbestetzte Matrizen ist:

$$\varphi(\mathbf{A})\mathbf{b} = \mathbf{V}_m \varphi(\mathbf{H}_m) \|\mathbf{b}\| \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

nach  $m$  Krylov-Schritten mit dem Startvektor  $\mathbf{b}$ .

**Beispiel 4.3.3.** Wir betrachten die logistische Differentialgleichung 2. Ordnung:

$$\begin{cases} \dot{y} = \lambda y^2(1 - y), & \lambda = 500, \quad h = \frac{1}{80} \\ y(0) = 0.01. \end{cases}$$

Verallgemeinerung: exponentielle RK-ESV

$$\mathbf{J} = D\mathbf{f}(\mathbf{y}_k).$$

**Semi-implizites Euler-Verfahren:**  $\mathbf{y}_{k+1} = \mathbf{y}_k + (\mathbf{Id} - h\mathbf{J})^{-1} h\mathbf{f}(\mathbf{y}_k).$

**Exponentielles Euler-Verfahren:**  $\mathbf{y}_{k+1} = \mathbf{y}_k + \varphi(h\mathbf{J}) h\mathbf{f}(\mathbf{y}_k).$

**Idee:** Ersetze  $(\mathbf{Id} - \gamma h \mathbf{J})^{-1}$  durch  $\varphi(\gamma h \mathbf{J})$  im linear-impliziten RK.

$\implies$  exponentielle RK-ESV:

$$\mathbf{k}_i := \varphi(a_{ii} h \mathbf{J}) \left[ \mathbf{f}(u_i) + h \mathbf{J} \sum_{j=1}^{i-1} d_{ij} \mathbf{k}_j \right] \text{ für } i = 1, 2, \dots$$

$$\mathbf{u}_i := \mathbf{y}_0 + h \sum_{j=1}^{i-1} (a_{ij} + d_{ij}) \mathbf{k}_j$$

$$\mathbf{y}_1 := \Psi^h \mathbf{y}_0 := \mathbf{y}_0 + h \sum_{i=1}^s b_i \mathbf{k}_i.$$

Das ist ein  $s$ -stufiges exponentielles RK-ESV für  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ .

Gewünschte Ordnung  $\implies b_i, a_{ij}, c, d_{ij}$ .

Zusatzbedingung: exakte Integration von  $\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} + \mathbf{g}$ .

# Chapter 5

## Intermezzo über (numerische) lineare Algebra

Im Folgenden wird  $\mathbf{A}$  eine  $n \times n$  Matrix mit reellen oder komplexen Einträgen bezeichnen.

### 5.1.1 Grundlagen

**Bemerkung 5.1.1.** Das Ergebnis der Multiplikation einer Matrix  $\mathbf{A}$  mit einem Vektor  $\mathbf{x}$  ist eine lineare Kombination der Spalten von  $\mathbf{A}$ :

$$\mathbf{A} \mathbf{x} = \sum_{i=1}^n x_i \mathbf{A}_{:,i}.$$

Invertierbar	Nicht Invertierbar
$A$ ist regulär	$A$ ist singulär
Spalten sind linear unabhängig	Spalten sind linear abhängig
Zeilen sind linear unabhängig	Zeilen sind linear abhängig
$\det A \neq 0$	$\det A = 0$
$Ax = 0$ hat eine Lösung $x = 0$	$Ax = 0$ hat unendlich viele Lösungen
$Ax = b$ hat eine Lösung $x = A^{-1}b$	$Ax = b$ hat keine oder unendlich viele Lösungen
$A$ hat vollen Rang	$A$ hat Rang $r < n$
$A$ hat $n$ nicht-Null-Pivoten	$A$ hat $r < n$ Pivoten
$\text{span}\{A_{:,1}, \dots, A_{:,n}\}$ hat Dimension $n$	$\text{span}\{A_{:,1}, \dots, A_{:,n}\}$ hat Dimension $r < n$
$\text{span}\{A_{1,:}, \dots, A_{n,:}\}$ hat Dimension $n$	$\text{span}\{A_{1,:}, \dots, A_{n,:}\}$ hat Dimension $r < n$
Alle Eigenwerte von $A$ sind nicht-Null	$0$ ist Eigenwert von $A$
$0 \notin \sigma(A) = \text{Spektrum von } A$	$0 \in \sigma(A)$
$A^H A$ ist symmetrisch positiv definit	$A^H A$ ist nur semidefinit
$A$ hat $n$ (positive) Singulärwerte	$A$ hat $r < n$ (positive) Singulärwerte

**Definition 5.1.2.** Die Vektoren  $\mathbf{q}_1, \dots, \mathbf{q}_n$  heissen *orthogonal*, falls

$$\mathbf{q}_i^H \cdot \mathbf{q}_j = 0 \text{ für alle } i \neq j;$$

wenn sie ausserdem normiert sind, d.h.  $\mathbf{q}_i^H \cdot \mathbf{q}_i = 1$ , dann heissen sie *orthonormal*.

**Bemerkung 5.1.3.** In der vorherigen Definition wird die Euklidische Norm verwendet:

$$\|\mathbf{q}\|_2^2 = \mathbf{q}^H \cdot \mathbf{q}.$$

**Bemerkung 5.1.4.** Wenn  $\mathbf{q}_1, \dots, \mathbf{q}_n$  orthonormal sind, dann

$$\mathbf{Q}^H \cdot \mathbf{Q} = \begin{bmatrix} \text{---} & \mathbf{q}_1^H & \text{---} \\ \text{---} & \mathbf{q}_2^H & \text{---} \\ & \vdots & \\ \text{---} & \mathbf{q}_n^H & \text{---} \end{bmatrix} \begin{bmatrix} \left| \right| & \left| \right| & & \left| \right| \\ \mathbf{q}_1 & \mathbf{q}_2 & \dots & \mathbf{q}_n \\ \left| \right| & \left| \right| & & \left| \right| \end{bmatrix} = \begin{bmatrix} 1 & & & 0 \\ & 1 & & \\ & & \ddots & \\ 0 & & & 1 \end{bmatrix} = \mathbf{I}_n.$$

Wenn  $Q$  quadratisch ist, dann ist  $Q$  eine orthogonale (bzw. unitäre) Matrix, also

$$\mathbf{Q}^{-1} = \mathbf{Q}^H.$$

**Bemerkung 5.1.5.** Orthogonale/Unitäre Transformationen erhalten die Euklidische Norm:

$$\|\mathbf{Q}\mathbf{x}\|_2^2 = (\mathbf{Q}\mathbf{x})^H (\mathbf{Q}\mathbf{x}) = \mathbf{x}^H \mathbf{Q}^H \mathbf{Q} \mathbf{x} = \mathbf{x}^H \mathbf{I}_n \mathbf{x} = \|\mathbf{x}\|_2^2.$$

**Bemerkung 5.1.6.** Permutationsmatrix: Zeilen wie  $\mathbf{I}_n$ , andere Reihenfolge. Durch Umordnen der Einträge in einem Vektor wird ihre Länge nicht verändert!

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad \mathbf{P}^H \mathbf{P} = \mathbf{I}.$$

**Bemerkung 5.1.7.** Drehungen verändern die Länge der Vektoren nicht! Die Drehmatrix mit Winkel  $\theta$  in der 1-3-Ebene ist:

$$\mathbf{P} = \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

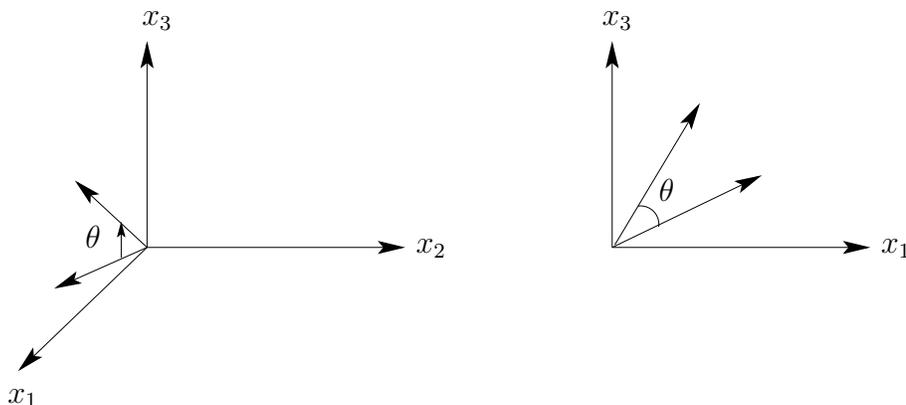


Abb. 5.1.8. Drehung mit Winkel  $\theta$  in der 1-3-Ebene.

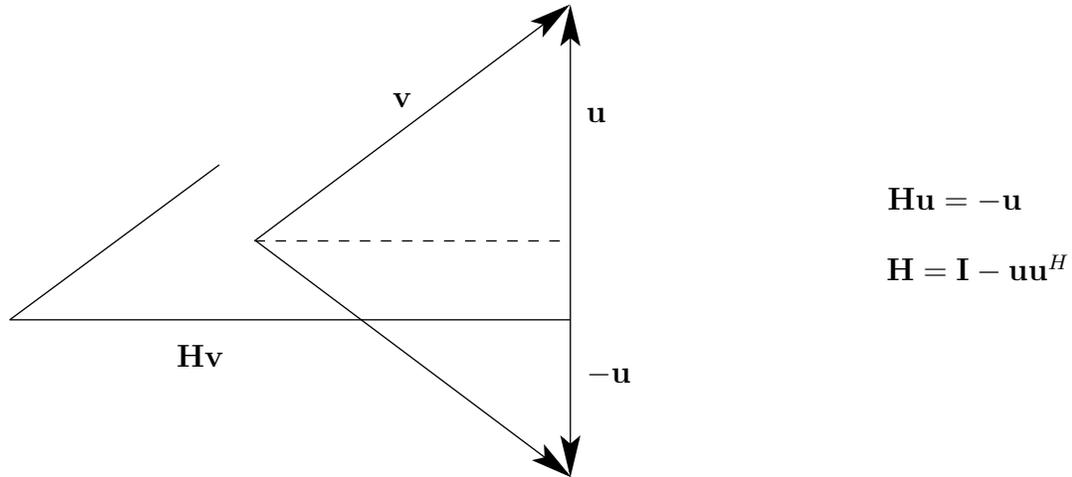


Abb. 5.1.9. Längen werden durch Spiegelungen nicht verändert!

**Bemerkung 5.1.10.** Gegeben die linear unabhängigen Vektoren  $\mathbf{a}_1, \dots, \mathbf{a}_m$ , kann man  $\mathbf{q}_1, \dots, \mathbf{q}_m$  orthonormale Vektoren finden, die denselben Raum wie  $\mathbf{a}_1, \dots, \mathbf{a}_m$  aufspannen. In der Vorlesung über die lineare Algebra haben Sie den Gram-Schmidt-Algorithmus kennengelernt:

$$\mathbf{q}_1 := \frac{1}{\|\mathbf{q}_1\|} \mathbf{a}_1$$

$$\mathbf{q}_2 := \mathbf{a}_2 - (\mathbf{q}_1^H \mathbf{a}_2) \mathbf{q}_1; \quad \mathbf{q}_2 := \frac{1}{\|\mathbf{q}_2\|_2} \mathbf{q}_2$$

etc.

Dieser Algorithmus ist aber instabil; deswegen berechnet man  $q_1, \dots, q_n$  numerisch mit dem modifizierten Gram-Schmidt-Algorithmus, wie in der entsprechenden Übungsaufgabe beschrieben.

**Bemerkung 5.1.11.** Das Gram-Schmidt-Verfahren liefert eine  $QR$ -Zerlegung der Matrix  $\mathbf{A}$ :

$$\mathbf{A} = \mathbf{Q}\mathbf{R},$$

wobei

$$\mathbf{A} = \begin{bmatrix} \left| \right. & \left| \right. & \cdots & \left| \right. \\ \mathbf{a}_1 & \mathbf{a}_2 & & \mathbf{a}_n \\ \left| \right. & \left| \right. & & \left| \right. \end{bmatrix}, r_{ii} = \|\mathbf{a}_i\|_2^2 \text{ und } r_{ik} = \mathbf{a}_i^H \mathbf{a}_k.$$



Die Gauss-Elimination ergibt die  $LU$ -Zerlegung.

**Beispiel 5.1.12.** Gauss-Elimination:

$$\begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} \iff \begin{array}{rcl} x_1 + x_2 & = & 4 \\ 2x_1 + x_2 - x_3 & = & 1 \\ 3x_1 - x_2 - x_3 & = & -3 \end{array} .$$

$$\begin{array}{ccc} \begin{bmatrix} 1 & & \\ & 1 & \\ & & 1 \end{bmatrix} & \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix} & \begin{bmatrix} 4 \\ 1 \\ -3 \end{bmatrix} \implies \\ \begin{bmatrix} 1 & & \\ \color{red}{2} & 1 & \\ & 0 & 1 \end{bmatrix} & \begin{bmatrix} \color{blue}{1} & \color{blue}{1} & 0 \\ \color{red}{0} & -1 & -1 \\ 3 & -1 & -1 \end{bmatrix} & \begin{bmatrix} 4 \\ -7 \\ -3 \end{bmatrix} \implies \\ \begin{bmatrix} 1 & & \\ 2 & 1 & \\ \color{red}{3} & 0 & 1 \end{bmatrix} & \begin{bmatrix} \color{blue}{1} & \color{blue}{1} & 0 \\ 0 & -1 & -1 \\ \color{red}{0} & -4 & -1 \end{bmatrix} & \begin{bmatrix} 4 \\ -7 \\ -15 \end{bmatrix} \implies \\ \underbrace{\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 3 & \color{red}{4} & 1 \end{bmatrix}}_{=L} & \underbrace{\begin{bmatrix} 1 & 1 & 0 \\ 0 & \color{blue}{-1} & \color{blue}{-1} \\ 0 & \color{red}{0} & 3 \end{bmatrix}}_{=U} & \begin{bmatrix} 4 \\ -7 \\ 13 \end{bmatrix} . \end{array}$$

Blau: **Pivot-Zeile**, Blau und fett: **Pivot-Elemente**, Rot: **negative Multiplikatoren**

**Bemerkung 5.1.13.** Numerische Lösung eines linearen Gleichungssystems:

$$\mathbf{Ax} = \mathbf{b} .$$

1. Zerlege  $\mathbf{A} = \mathbf{LU}$  (Kosten für einen  $n \times n$  Matrix  $\mathbf{A}$ :  $\frac{1}{3} n(n-1)(n-2)$  Operationen<sup>1</sup>).
2. Löse  $\mathbf{Lz} = \mathbf{b}$  durch Vorwärtssubstitution (Kosten:  $\frac{1}{2} n(n-1)$  Operationen).
3. Löse  $\mathbf{Vx} = \mathbf{z}$  durch Rückwärtssubstitution (Kosten:  $\frac{1}{2} n(n-1)$  Operationen).

---

<sup>1</sup>Das ist der teure Schritt. Die Gauss-Elimination kann bis  $\mathcal{O}(n^{2.373})$  mit sehr komplizierten (also unpraktischen) Algorithmen beschleunigt werden. Die Methode, die ein lineares Gleichungssystem in  $\mathcal{O}(n^2)$  exakt löst, wird die ganze Wissenschaft revolutionieren, so wie die FFT nach 1965 es tat.

**Beispiel 5.1.14.**

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 2 \\ 2 & -7 & 2 \\ 1 & 24 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & -7 & 2 \\ 1 & 2 & 2 \\ 1 & 24 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & -7 & 2 \\ 0 & 5.5 & 1 \\ 0 & 27.5 & -1 \end{bmatrix} \rightarrow$$

$$\begin{bmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 5.5 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 0 & 1.2 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 2 & -7 & 2 \\ 0 & 27.5 & -1 \\ 0 & 0 & 1.2 \end{bmatrix}, \quad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 0.2 & 1 \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

**Lemma 5.1.15.** Für jede  $n \times n$  reguläre Matrix  $A$  gibt es eine Permutationsmatrix  $P$  und eine untere Dreiecksmatrix mit 1 auf der Hauptdiagonalen  $L$  zusammen mit einer oberen Dreiecksmatrix  $U$ , sodass

$$\mathbf{PA} = \mathbf{LU}.$$

**Bemerkung 5.1.16.** Jede Vektornorm in  $\mathbb{R}^n$  oder  $\mathbb{C}^n$  induziert eine Matrixnorm

$$\|\mathbf{A}\| := \sup_{\|\mathbf{x}\| \neq 0} \frac{\|\mathbf{Ax}\|}{\|\mathbf{x}\|} = \sup_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\|.$$

**Definition 5.1.17.** Die *Konditionszahl* einer Matrix  $\mathbf{A}$  ist

$$\text{cond}(\mathbf{A}) := \|\mathbf{A}^{-1}\| \cdot \|\mathbf{A}\|.$$

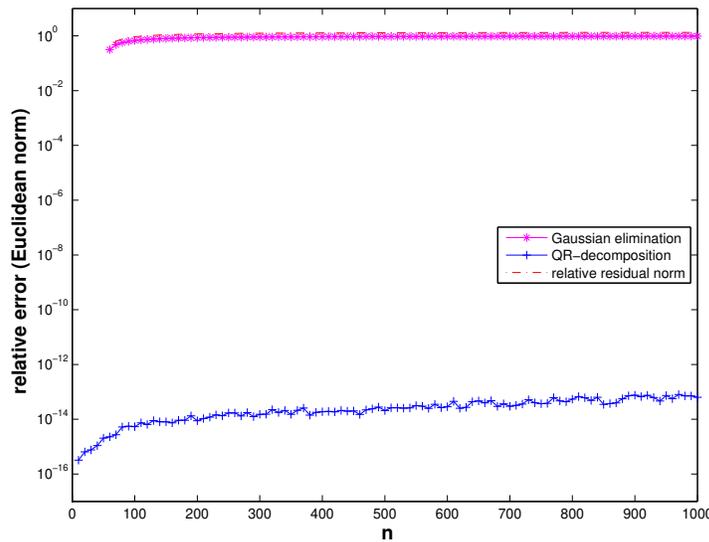
**Bemerkung 5.1.18.**  $\text{cond}(\mathbf{I}_m) = 1$ .

Wenn  $\text{cond}(\mathbf{A}) \gg 1$ , dann können sich die Rundungsfehler propagieren, sodass sehr grosse Fehler in der Gauss-Elimination auftreten<sup>2</sup>.

**Beispiel 5.1.19.** Wilkinson-Matrix

$$a_{ij} = \begin{cases} 1 & , \text{ wenn } i = j , \\ 1 & , \text{ wenn } j = n , \\ -1 & , \text{ wenn } i > j , \\ 0 & \text{ sonst.} \end{cases} \quad \mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1 & 0 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & 1 & 0 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 0 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 \end{bmatrix}$$

<sup>2</sup>Allerdings passiert dies selten in der Praxis, so wie Wilkinson selber meint: *Anyone that unlucky has already been run over by a bus!* Das ist heute noch eine offene Frage: warum zeigt sich die Gauss-Elimination in allen praktischen Fällen stabil?



**Abb. 5.1.20.** Relativer Fehler von LU- und QR-Zerlegungen

Wie man sieht, ist die euklidische Norm mit der QR-Zerlegung viel besser erhalten, während die LU-Zerlegung sich schlechter verhält. Die QR-Zerlegung werden wir nachher im Detail anschauen.

Ein ähnlicher Graph kann mittels des folgenden Codes erhalten werden:

**Code 5.1.21:** Erhaltung der Norm für QR und LU Zerlegung

```

1 from numpy import*
2 from numpy.linalg import norm
3 from scipy.linalg import lu, qr
4 from matplotlib.pyplot import figure, semilogy, show, legend, xlabel, ylabel
5
6 # first we define a function for producing the n × n wilkinson matrix
7 def wil_mat(n):
8     wilm = zeros((n,n))
9     for k in range(n):
10        for j in range(n):
11            if (j==k or j==n-1):
12                wilm[k,j] = 1.0
13            if (j<k):
14                wilm[k,j] = -1.0
15    return wilm
16
17 Ns = arange(5,1000,20)
18
19 err_lu = []
20 err_qr = []
21
22 for n in Ns:
23     matrix = wil_mat(n)
24     L, U = lu(matrix, permute_l=True) # P is a permutation matrix
25     Q, R = qr(matrix)

```

```

26 # now compute the relative errors. We divide by the norm of LU and QR
because it turns out that in the case of LU it is much bigger than the norm of
the wilkinson matrix
err_lu += [norm(dot(L,U)-matrix)/norm(dot(L,U))]
28 err_qr += [norm(dot(Q,R)-matrix)/norm(dot(Q,R))]

30 figure()
semilogy(Ns, err_lu, "-x", label = "LU")
32 semilogy(Ns, err_qr, "-*", label = "QR")
xlabel("$n$")
34 ylabel("relative error (Euclidean norm)")
legend(loc="best")
36 show()

```

Ein Ausweg ist die QR-Zerlegung von  $\mathbf{A}$ , siehe unten.

### 1a) Cholesky-Zerlegung

Für eine (Hermite)-symmetrische Matrix  $A$  können wir die LU-Zerlegung so umschreiben, dass die Pivoten oder das Diagonalelement von  $V$  isoliert sind:

$$\mathbf{A} = \mathbf{LU} = \mathbf{LDL}^H = \mathbf{R}^H \mathbf{R} \text{ mit } \mathbf{R} = \sqrt{\mathbf{D}} \mathbf{L}^H$$

und  $\sqrt{\mathbf{D}}$  ist die Diagonalmatrix mit Einträgen  $\sqrt{D_{ii}}$ ,  $i = 1, 2, \dots, n$ .

Für eine symmetrisch positiv definierte Matrix ist keine Pivot-Technik in der Gauss-Elimination nötig und der Algorithmus lässt sich mit der Hälfte der Anzahl an Operatoren (im Vergleich zur LU-Zerlegung) durchführen.

Die entsprechenden Python-Funktionen befinden sich in `scipy.linalg` und heissen `lu`, `lu_factor`, `lu_solve`, `cholesky`, `cho_factor`, `cho_solve`.

### 2) QR-Zerlegung

Durch die LU-Zerlegung können sich Matrizen ergeben, die eine grosse Konditionszahl haben (schlecht konditionierte Matrizen), sodass sich die Rundungsfehler propagieren und vergrößern.

Transformationen, die aber normerhaltend sind, lassen die Konditionszahl einer Matrix unverändert. In der Tat, sei  $\mathbf{A} : \mathbb{E}^n \rightarrow \mathbb{E}^m$ . Die Kondition der Matrix  $\mathbf{A}$  ist  $\text{cond}(\mathbf{A}) = \|\mathbf{A}^{-1}\| \cdot \|\mathbf{A}\|$ . Seien  $\mathbf{T} : \mathbb{E}^n \rightarrow \mathbb{E}^n$  und  $\mathbf{S} : \mathbb{E}^m \rightarrow \mathbb{E}^m$  Basistransformationsmatrizen, die die Norm erhalten.

Wir haben ein kommutierendes Diagramm:

$$\begin{array}{ccccc}
 \mathbf{x} \in \mathbb{E}^n & \xrightarrow{\mathbf{A}} & \mathbb{E}^m & \mathbf{y} & \text{alte Basis } \mathcal{B} \\
 \downarrow & \mathbf{T}^{-1} \downarrow \uparrow \mathbf{T} & & \mathbf{S}^{-1} \downarrow \uparrow \mathbf{S} & \downarrow \\
 \tilde{\mathbf{x}} \in \mathbb{E}^n & \xrightarrow{\mathbf{B}} & \mathbb{E}^m & \tilde{\mathbf{y}} & \text{neue Basis } \tilde{\mathcal{B}};
 \end{array}$$

$\mathbf{T}$  und  $\mathbf{S}^{-1}$  sind Norm-erhaltend, dies bedeutet  $\|\mathbf{T}\tilde{\mathbf{x}}\| = \|\tilde{\mathbf{x}}\|$  und  $\|\mathbf{S}^{-1}\mathbf{y}\| = \|\mathbf{y}\|$ . Daraus folgt:

$$\begin{aligned} \|\mathbf{B}\| &= \sup_{\|\tilde{\mathbf{x}}\|=1} \|\mathbf{B}\tilde{\mathbf{x}}\| = \sup_{\|\tilde{\mathbf{x}}\|=1} \|\mathbf{S}^{-1}\mathbf{A}\mathbf{T}\tilde{\mathbf{x}}\| = \\ &= \sup_{\|\tilde{\mathbf{x}}\|=1} \|\mathbf{A}\underbrace{\mathbf{T}\tilde{\mathbf{x}}}_{\mathbf{x}}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \|\mathbf{A}\|, \end{aligned}$$

und analog für  $\|\mathbf{A}^{-1}\|$ , wobei  $\mathbf{T}^{-1}$  und  $\mathbf{S}$  Norm-erhaltend sein sollen. Wir können also schliessen:

**Bemerkung 5.1.22.** Unitäre/orthogonale Transformationen lassen die Konditionszahl einer Matrix unverändert.

Das Gram-Schmidt-Verfahren verwendet Dreiecksmatrizen um eine orthogonale Basis zu finden. Dabei kann sich die Kondition der Matrix ändern. Nun wollen wir anders vorgehen: wie können wir eine Matrix  $\mathbf{A}$  auf obere Rechtecksform  $\mathbf{R}$  bringen, indem wir nur orthogonale/unitäre Transformation verwenden?

Es gibt zwei Möglichkeiten: Spiegelungen und Drehungen.

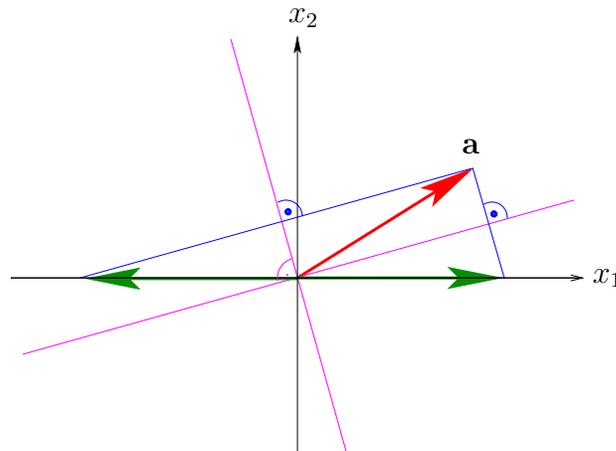
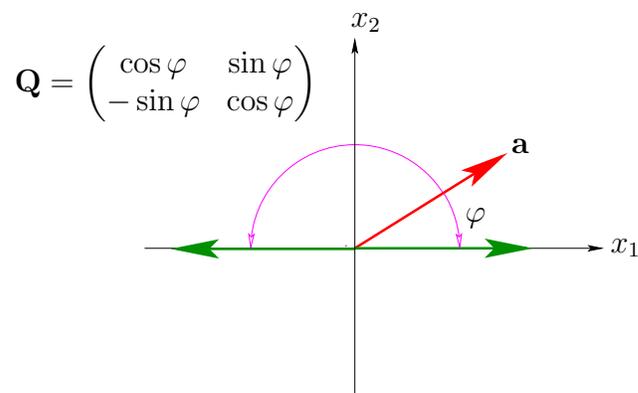


Abb. 5.1.23. Spiegelung um die Winkelhalbierende



$$\mathbf{Q} = \begin{pmatrix} \cos \varphi & \sin \varphi \\ -\sin \varphi & \cos \varphi \end{pmatrix}$$

**Abb. 5.1.24.** Drehung:  $\mathbf{a}$  wird über die  $x_1$ -Axe gedreht

Aufgabe: gegeben  $\mathbf{a} \in \mathbb{R}^n$ , finde  $\mathbf{Q} \in \mathbb{R}^{n \times n}$  orthogonal, sodass  $\mathbf{Q}\mathbf{a} = \|\mathbf{a}\|_2 \mathbf{e}_1$  (wobei  $\mathbf{e}_1 = [1 \ 0 \dots 0]^T$  der kanonische Einheitsvektor in  $\mathbb{R}^n$  ist). Eine mögliche Lösungsmethode ist die Householder Spiegelung.

Die Householder Spiegelung verwendet die Matrix  $\mathbf{Q} = \mathbf{H}(\mathbf{v}) := \mathbf{I} - 2 \frac{\mathbf{v}\mathbf{v}^H}{\mathbf{v}^H\mathbf{v}}$  mit  $\mathbf{v} = \frac{1}{2} (\mathbf{a} + \|\mathbf{a}\|_2 \mathbf{e}_1)$ ; normalerweise nimmt man:

$$\mathbf{v} = \begin{cases} \frac{1}{2} (\mathbf{a} + \|\mathbf{a}\|_2 \mathbf{e}_1), & \text{falls } a_1 > 0 \\ \frac{1}{2} (\mathbf{a} - \|\mathbf{a}\|_2 \mathbf{e}_1), & \text{falls } a_1 \leq 0. \end{cases}$$

Im Fall komplexer Vektoren,  $a_1 = |a_1| e^{i\varphi}$  mit der Phase  $\varphi \in [0, 2\pi]$  und

$$\mathbf{v} = \frac{1}{2} (\mathbf{a} + \|\mathbf{a}\|_2 \mathbf{e}_1 e^{-i\varphi}).$$

Eine Householder-Spiegelung produziert  $\|\mathbf{a}\|_2 \mathbf{e}_1$  auf einen Schlag. Für dünnbesetzte Matrizen kann es aber günstiger sein, die fehlende Null in der Matrix schrittweise zu erstellen. Dafür sind alle Givens-Rotationen geeignet:

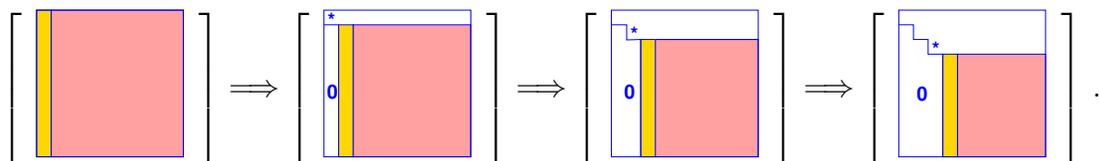
$$\mathbf{G}_{1k}(a_1 a_k) := \begin{bmatrix} \bar{\gamma} & \cdots & \bar{\sigma} & \cdots & 0 \\ \vdots & \ddots & \vdots & & \vdots \\ -\sigma & \cdots & \gamma & \cdots & 0 \\ \vdots & & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_k \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} a_1^{(1)} \\ \vdots \\ \mathbf{0} \\ \vdots \\ a_n \end{bmatrix}, \quad \begin{aligned} \gamma &= \frac{a_1}{\sqrt{|a_1|^2 + |a_k|^2}}, \\ \sigma &= \frac{a_k}{\sqrt{|a_1|^2 + |a_k|^2}}. \end{aligned} \quad (5.1.2)$$

Für jede Spalte sind so viele Givens-Rotationen nötig, wie die Anzahl der Einträge, die nicht null sind.

Ob mittels Spiegelungen oder Rotationen, brauchen wir für jede Spalte unitäre Transformationen und  $n - 1$  Schritte:

$$\mathbf{Q}_{n-1} \mathbf{Q}_{n-2} \dots \mathbf{Q}_1 \mathbf{A} = \mathbf{R}$$

und somit  $\mathbf{A} = \mathbf{Q}\mathbf{R}$  mit  $\mathbf{Q} = \mathbf{Q}_1^H \dots \mathbf{Q}_{n-1}^H$  unitär und  $\mathbf{R}$  obere Dreiecksmatrix.



**Abb. 5.1.25.** Transformation zu einer oberen Dreiecksmatrix mittels unitären Transformationen



## Code 5.1.27: Rechenzeit für QR-Zerlegungen

```

1  from numpy import r_, mat, vstack, eye, ones, zeros
2  from scipy.linalg import qr
3  import timeit

4
5  def qr_full(): #full qr decomposition using scipy
6      global A
7      Q, R = qr(A)

8
9  def qr_econ(): #economic qr decomposition using scipy
10     global A
11     #Q, R = qr(A, econ=True)
12     Q, R = qr(A, mode='economic')

13
14 def qr_ovecon(): #economic mode, overwrite = True
15     global A
16     Q, R = qr(A, mode='economic', overwrite_a=True)

17
18 def qr_r(): #mode = "r": returns only the triangular matrix R, not Q
19     global A
20     R = qr(A, mode='r')

21
22 nrEXP = 4 #number of experiments
23 sizes = 2*r_[2:7]
24 qrtimes = zeros((5,sizes.shape[0]))
25 k = 0
26 for n in sizes:
27     print('n=', n)
28     m = n*2#4*n
29     A = mat(1.*r_[1:m+1]).T*mat(1.*r_[1:n+1])
30     A += vstack((eye(n), ones((m-n,n)) ))

31
32     #we perform the computation with the three techniques
33
34     t = timeit.Timer('qr_full()', 'from __main__ import qr_full')
35     avqr = t.timeit(number=nrEXP)/nrEXP
36     #in order to measure the necessary time we let the code run nrEXP times and
37     then divide by nrEXP
38     #so as to get the average time
39     print(avqr)
40     qrtimes[0,k] = avqr

41
42     t = timeit.Timer('qr_econ()', 'from __main__ import qr_econ')
43     avqr = t.timeit(number=nrEXP)/nrEXP
44     print(avqr)
45     qrtimes[1,k] = avqr

46
47     t = timeit.Timer('qr_ovecon()', 'from __main__ import qr_ovecon')
48     avqr = t.timeit(number=nrEXP)/nrEXP
49     print(avqr)
50     qrtimes[2,k] = avqr

51
52     t = timeit.Timer('qr_r()', 'from __main__ import qr_r')

```

```

53     avqr = t.timeit(number=nrEXP)/nrEXP
54     print(avqr)
55     qrtimes[3,k] = avqr
56
57     k += 1
58
59 #now we plot the results in order to understand their efficiency (at least
60 qualitatively)
61 import matplotlib.pyplot as plt
62 plt.loglog(sizes, qrtimes[0], 's', label='qr')
63 plt.loglog(sizes, qrtimes[1], '*', label="qr(econ=True)")
64 plt.loglog(sizes, qrtimes[2], '.', label="qr(econ=True, overwrite_a=True)")
65 plt.loglog(sizes, qrtimes[3], 'o', label="qr(mode='r')")
66
67 v4 = qrtimes[1,1] * (sizes/sizes[1])**4
68 v6 = qrtimes[0,1] * (sizes/sizes[1])**6
69 plt.loglog(sizes, v4, label='$O(n^4)$')
70 plt.loglog(sizes, v6, '--', label='$O(n^6)$')
71 plt.legend(loc=2)
72 plt.xlabel('$n$')
73 plt.ylabel('time [s]')
74 plt.savefig('qrtiming.eps')
75 plt.show()

```

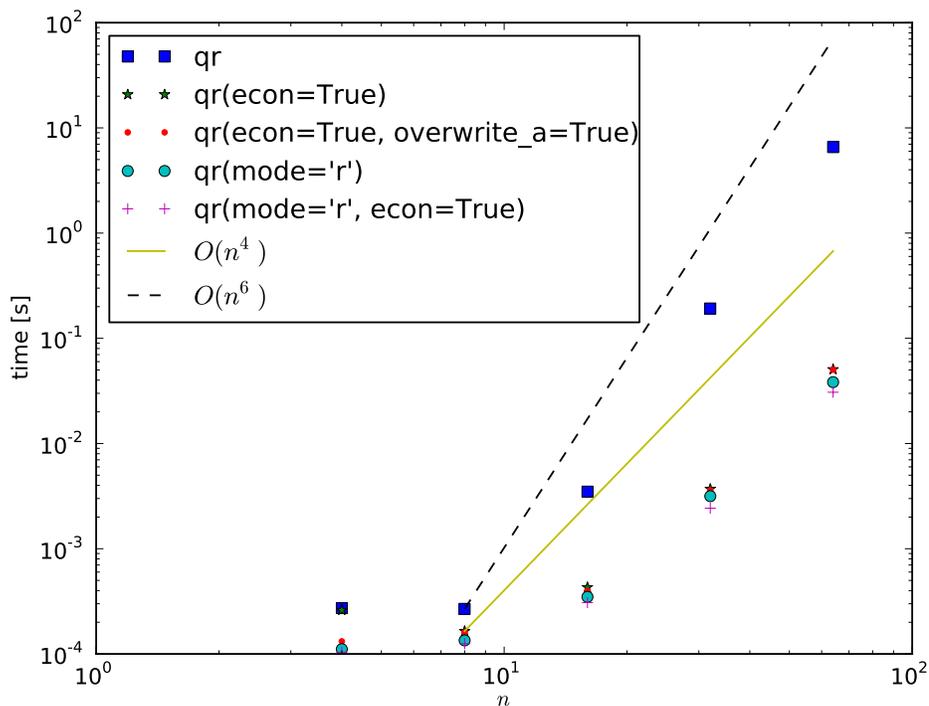
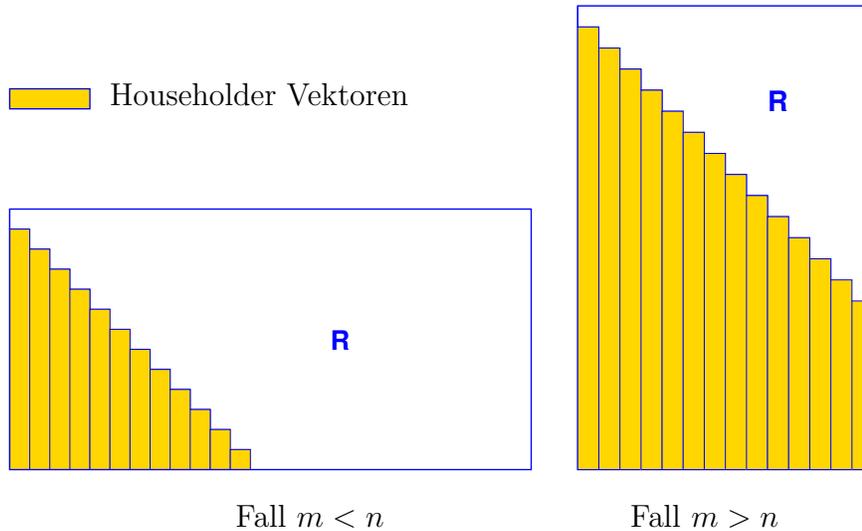


Abb. 5.1.28. Zeitaufwand verschiedener Varianten der QR-Zerlegung

**Bemerkung 5.1.29.** Wie können wir die verwendeten orthogonalen Transformationen speichern? Die Householder-Vektoren  $\mathbf{v}_j$  werden in jedem Schritt kürzer, so dass wir diese in dem unteren Teil von  $\mathbf{A}$  speichern können.



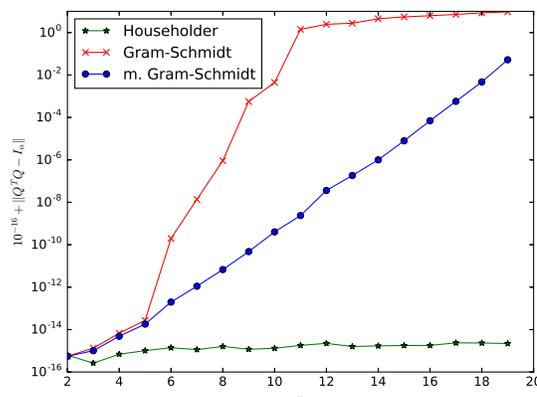
Man verwendet die folgende *Konvention* für die Givens-Drehungen ( $\mathbb{K} = \mathbb{R}$ ):

$$\mathbf{G} = \begin{bmatrix} \bar{\gamma} & \bar{\sigma} \\ -\sigma & \gamma \end{bmatrix} \implies \text{speichere } \rho := \begin{cases} 1, & \text{if } \gamma = 0, \\ \frac{1}{2} \text{sign}(\gamma)\sigma, & \text{if } l|\sigma| < |\gamma|, \\ 2 \text{sign}(\sigma)/\gamma, & \text{if } |\sigma| \geq |\gamma|. \end{cases}$$

$$\begin{cases} \rho = 1 \implies \gamma = 0, & \sigma = 1 \\ |\rho| < 1 \implies \sigma = 2\rho, & \gamma = \sqrt{1 - \sigma^2} \\ |\rho| > 1 \implies \gamma = 2/\rho, & \sigma = \sqrt{1 - \gamma^2}. \end{cases}$$

Man speichert die Rotationsmatrix  $\mathbf{G}_{ij}(ab)$  als  $(i, j, \rho)$ , was effizient ist, falls die Matrix  $\mathbf{A}$  dünnbestetzt ist.

### Numerische Stabilität



**Abb. 5.1.30.** Die Qualität verschiedener Algorithmen zur Orthogonalisierung:  $\|\mathbf{I} - \mathbf{Q}^t \mathbf{Q}\|_2$  gegen Anzahl Spalten einer Vandermonde Matrix.

Die Entwicklung mehrerer Methoden zur Lösung eines Problems hat meistens etwas mit der numerischen Stabilität der einzelnen Verfahren zu tun.

Die Abbildung 5.1.30 zeigt den Vergleich der Orthogonalität der Matrix  $\mathbf{Q}$  für die QR-Zerlegung einer Vandermonde Matrix ( $m = 20$ ):

$$\mathbf{A} = \begin{bmatrix} t_0^{n-1} & \cdots & t_0^1 & 1 \\ t_1^{n-1} & \cdots & t_1^1 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ t_{19}^{n-1} & \cdots & t_{19}^1 & 1 \end{bmatrix}, \text{ mit } t_i = \frac{i}{(m-1)}.$$

**Bemerkung 5.1.31.** Es gibt mehrere Möglichkeiten, die QR-Zerlegung einer Matrix  $A$  zu berechnen:

- 1) Der modifizierte Gram-Schmidt Algorithmus orthogonalisiert  $A$  via trianguläre Operationen (obere Dreiecksmatrizen); Vorteil: wir können früher aufhören und haben schon die ersten orthogonalen Spalten. Nachteil: die Stabilität ist nicht garantiert.
- 2) Orthogonale Transformationen (Householder-Spiegelungen für eine vollbesetzte Matrix oder Givens-Rotationen für eine dünnbestetzte Matrix) triangularisieren  $A$ ; Vorteil:  $\mathbf{Q}$  ist orthogonal trotz Rundungsfehler.

### 3) Singulärwertzerlegung

**Beispiel 5.1.32.** In der Hauptkomponentenanalyse (Englisch: principal component analysis, PCA) sind  $n$  Punkte (Messungen)  $\mathbf{a}_1, \dots, \mathbf{a}_n \in \mathbb{R}^m$  in einem  $m$ -dimensionalen Raum (welche Größen gemessen worden sind) gegeben. Die entsprechenden Vektoren stellen wir in einer Matrix dar:

$$\mathbf{A} = [\mathbf{a}_1 \quad \mathbf{a}_2 \quad \dots \quad \mathbf{a}_n] \in \mathbb{R}^{m \times n}.$$

Ziel der PCA ist, nur so viel Information aus den Messungen zu behalten wie notwendig; mathematisch bedeutet dies, dass wenn  $\mathbf{a}_1, \dots, \mathbf{a}_n$  linear abhängig sind, d.h., wenn sie in einem  $r$ -dimensionalen Raum ( $r \leq \min\{m, n\}$ ) liegen, liefert die PCA die Zahl  $r = \text{rank}(A)$  und eine orthonormale Basis von  $\text{span}\{\mathbf{a}_1, \dots, \mathbf{a}_n\} = \text{Im } A$ . Auf diese deterministische Art und Weise werden einige Messfehler eliminiert und alle Daten komprimiert.

**Bemerkung 5.1.33.** Die PCA ist eine Aufgabe der linearen Algebra; analog wie beim Lösen von linearen Gleichungssystemen ist es wichtig, dass wir stabile (das heisst nicht Rundungsfehler-anfällige) Algorithmen zur Ermittlung von  $r$  und der Basis in  $A$  verwenden.

**Theorem 5.1.34.** Sei  $\mathbf{A} \in \mathbb{C}^{m \times n}$  beliebig. Dann gibt es unitäre Matrizen  $\mathbf{V} \in \mathbb{C}^{n \times n}$  und  $\mathbf{U} \in \mathbb{C}^{m \times m}$  und die  $m \times n$  Diagonalmatrix in  $\mathbf{\Sigma} = \text{diag}(\sigma_1, \dots, \sigma_p)$  mit  $p = \min\{m, n\}$  und  $\sigma \geq \sigma_2 \geq \dots \geq \sigma_p \geq 0$ , sodass

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^H.$$

**Definition 5.1.35.** Diese Zerlegung heisst Singulärwertzerlegung (SVD = singular value decomposition) mit den Singulärwerten  $\sigma_i$ . Sie kann wie folgt veranschaulicht werden:

$$\left[ \begin{array}{c} \mathbf{A} \end{array} \right] = \left[ \begin{array}{c} \mathbf{U} \end{array} \right] \left[ \begin{array}{c} \mathbf{\Sigma} \end{array} \right] \left[ \begin{array}{c} \mathbf{V}^H \end{array} \right]$$

$$\left[ \begin{array}{c} \mathbf{A} \end{array} \right] = \left[ \begin{array}{c} \mathbf{U} \end{array} \right] \left[ \begin{array}{c} \mathbf{\Sigma} \end{array} \right] \left[ \begin{array}{c} \mathbf{V}^H \end{array} \right]$$

**Bemerkung 5.1.36.** Die Singulärwertzerlegung ermöglicht die Darstellung der (eventuell grossen) Matrix  $\mathbf{A}$  als Summe von Matrizen von Rang 1:

$$\mathbf{A} = \sum_{k=1}^p \sigma_k \mathbf{u}_k \mathbf{v}_k^H.$$

$\mathbf{u}_1, \dots, \mathbf{u}_p$  sind die ersten  $p$ -Zeilen von  $\mathbf{U}$  (die linken Singulärvektoren von  $\mathbf{A}$  genannt) und  $\mathbf{v}_1, \dots, \mathbf{v}_p$  sind die ersten  $p$ -Spalten von  $\mathbf{V}$  (auch die rechten Singulärvektoren von  $\mathbf{A}$  genannt). Somit lässt sich die Information der  $m \times n$ -Einträge aus  $\mathbf{A}$  mittels  $p + pn + pm = p(1 + n + m) \ll m \cdot n$  Einträgen darstellen. Die SVD wird somit zur Datenkompression verwendet (und war der technologische Schlüssel zum Erfolg von Google vor vielen Jahren).

**Bemerkung 5.1.37.** Die SVD ist nicht eindeutig, aber die Singulärwerte schon.

*Proof.* Angenommen,  $\mathbf{A}$  hat zwei Singulärwertzerlegungen:

$$\mathbf{A} = \mathbf{U}_1 \mathbf{\Sigma}_1 \mathbf{V}_1^H = \mathbf{U}_2 \mathbf{\Sigma}_2 \mathbf{V}_2^H.$$

Dann:

$$\mathbf{A}\mathbf{A}^H = \mathbf{U}_1 \mathbf{\Sigma}_1 \mathbf{V}_1^H \mathbf{V}_1 \mathbf{\Sigma}_1^H \mathbf{U}_1^H = \mathbf{U}_1 \mathbf{\Sigma}_1 \mathbf{\Sigma}_1^H \mathbf{U}_1^H$$

und gleichzeitig

$$\mathbf{A}\mathbf{A}^H = \mathbf{U}_2 \Sigma_2 \mathbf{V}_2^H \mathbf{V}_2 \Sigma_2^H \mathbf{U}_2^H = \mathbf{U}_2 \Sigma_2 \Sigma_2^H \mathbf{U}_2^H.$$

Somit sind die Diagonalmatrizen

$$\Sigma_1 \Sigma_1^H = \text{diag}(s_{1,1}^2, \dots, s_{1,p}^2)$$

und

$$\Sigma_2 \Sigma_2^H = \text{diag}(s_{2,1}^2, \dots, s_{2,p}^2),$$

ähnlich via orthogonale Transformationen und also gleich. □

**Bemerkung 5.1.38.**  $\sigma_1^2 \geq \dots \geq \sigma_r^2 > 0$  sind Eigenwerte von  $\mathbf{A}^H \mathbf{A}$ ,  $\mathbf{A}\mathbf{A}^H$  und

$$\begin{bmatrix} 0 & \mathbf{A}^H \\ \mathbf{A} & 0 \end{bmatrix} \text{ zu den Eigenvektoren } \mathbf{v}_1, \dots, \mathbf{v}_r; \mathbf{u}_1, \dots, \mathbf{u}_p; \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{v}_1 \end{bmatrix}, \dots, \begin{bmatrix} \mathbf{u}_p \\ \mathbf{v}_p \end{bmatrix}.$$

*Proof.*  $\mathbf{A}\mathbf{A}^H = \mathbf{U} \Sigma \mathbf{V}^H \mathbf{V} \Sigma^H \mathbf{U}^H = \mathbf{U} \Sigma \Sigma^H \mathbf{U}^H$ . Daher sind die Eigenwerte von  $\Sigma \Sigma^H$  auch Eigenwerte von  $\mathbf{A}\mathbf{A}^H$ . □

Die entsprechende Python-Funktion ist: `scipy.linalg.svd`.

`full_matrices=False`: sparsame SVD:

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} = \begin{bmatrix} \mathbf{U} \end{bmatrix} \begin{bmatrix} \Sigma \end{bmatrix} \begin{bmatrix} \mathbf{V}^H \end{bmatrix}$$

Komplexität:

$$\begin{aligned} 2mn^2 + 2n^3 + O(n^2) + O(mn) & \text{ für } \mathbf{s} = \text{svdvals}(\mathbf{A}), \\ 4m^2n + 22n^3 + O(mn) + O(n^2) & \text{ für } \mathbf{U}, \mathbf{S}, \mathbf{V} = \text{svd}(\mathbf{A}), \\ O(mn^2) + O(n^3) & \text{ für } \mathbf{U}, \mathbf{S}, \mathbf{V} = \text{svd}(\mathbf{A}, \text{full\_matrices=False}), m \gg n. \end{aligned}$$

**Theorem 5.1.39.** Wenn die Singulärwerte von  $\mathbf{A}$  erfüllen:

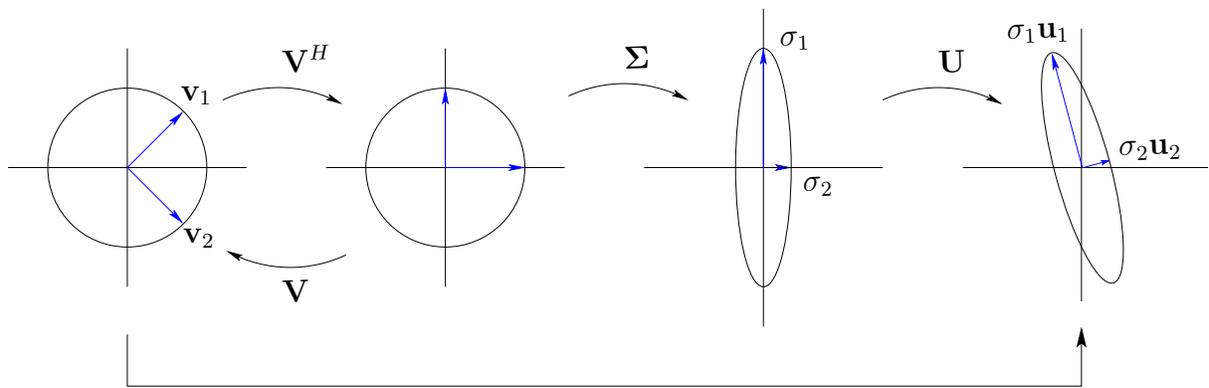
$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0$$

dann gilt:

$$\begin{aligned} \text{rank } \mathbf{A} &= r, \ker \mathbf{A} = \text{span}\{\mathbf{v}_{r+1}, \mathbf{v}_{r+2}, \dots, \mathbf{v}_n\}, \\ \text{Im } \mathbf{A} &= \text{span}\{\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_r\}. \end{aligned}$$

$$\begin{bmatrix} \mathbf{A} \end{bmatrix} = \begin{bmatrix} \mathbf{U} \end{bmatrix} \begin{bmatrix} \Sigma \end{bmatrix} \begin{bmatrix} \mathbf{V}^H \end{bmatrix} \quad (5.1.4)$$

columns = ONB of  $\text{Im}\mathbf{A}$                       rows = ONB of  $\ker\mathbf{A}$



$$\mathbf{A} = \mathbf{U} \Sigma \mathbf{V}^H$$

$$\begin{cases} \mathbf{A} \mathbf{v}_j = \sigma_j \mathbf{u}_j & \text{für } j \leq r \\ \mathbf{A} \mathbf{v}_j = 0 & \text{für } j > r \end{cases}$$

und

$$\begin{cases} \mathbf{A}^T \mathbf{u}_j = \sigma_j \mathbf{v}_j & \text{für } j \leq r \\ \mathbf{A}^T \mathbf{u}_j = 0 & \text{für } j > r \end{cases}$$

**Abb. 5.1.40.** Veranschaulichung der Singulärwertzerlegung in zwei Dimensionen.

**Bemerkung 5.1.41.** Wenn Messfehler oder Rundungsfehler auftreten, dann hat man typischerweise

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r \gg \sigma_{r+1} \approx \dots \approx \sigma_p \approx 0.$$

Der *numerische* Rang  $r$  ist hier gegeben durch den Sprung

$$\sigma_r \gg \sigma_{r+1} \approx 0$$

und ist in den Anwendungen relevant.

**Beispiel 5.1.42.** Wir betrachten zwei funktionale Abhängigkeiten, z.B. zwei an  $m = 50$  verschiedenen Orten gemessene Signale. Hier haben die zwei Signale die Form  $\sin(\pi t)$  und  $\cos(\pi t)$  für  $t = \frac{1}{50}, \frac{2}{50}, \dots, \frac{50}{50}$ . Die Messfehler simulieren wir durch  $m$  Zufallszahlen zwischen 0 und 1 (alle Zahlen zwischen 0 und 1 haben die gleiche Wahrscheinlichkeit zu erscheinen), die wir entweder mit dem Signal multiplizieren (für  $\sin(\pi t)$ ) oder ihm mit dem Dämpfungsfaktor 0.1 addieren (für  $\cos(\pi t)$ ).

Wir wiederholen das “Experiment”  $n = 10$  mal.

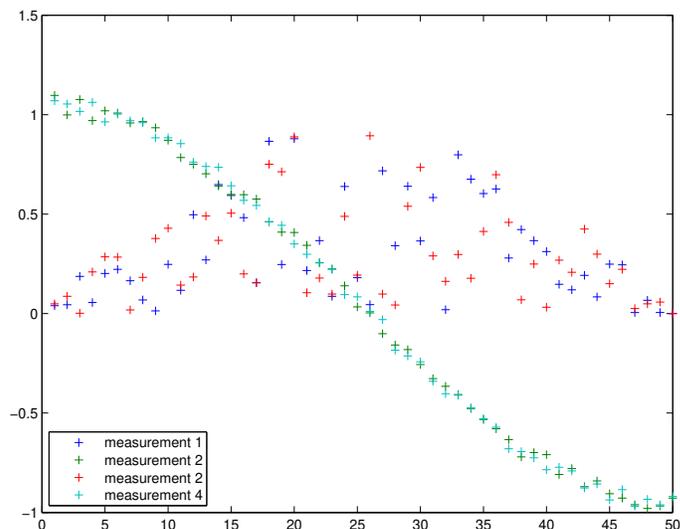
```

1 from numpy import linspace, sin, cos, pi, zeros
2 from numpy.linalg import norm
3 from numpy.random import rand
4 from matplotlib.pyplot import figure, show, plot, legend, xlabel, ylabel

6 n = 10; m = 50
7 r = linspace(1,m,m)
8 x = sin(pi*r/m)
9 y = cos(pi*r/m)
10 A = zeros((2*n,m))
11 for k in range(n):
12     A[2*k,:] = x*rand(m)
13     A[2*k+1,:] = y + 0.1*rand(m)

14
15 figure()
16 plot(r,A[0,:],"x", label="Measurement 1")
17 plot(r,A[1,:],"o", label="Measurement 2")
18 plot(r,A[2,:],"*", label="Measurement 3")
19 plot(r,A[3,:],"v", label="Measurement 4")
20 legend(loc="best")
    show()

```



**Abb. 5.1.43.** Einige rohe Daten aus dem Experiment

Nun haben wir die Ergebnisse von 10 Experimenten, d.h. 20 Vektoren der Länge 50. Mittels PCA wollen wir verfahren:

- 1) Wie viele Abhängigkeiten von  $t$  existieren? In der Sprache der linearen Algebra: wie viele dieser 20 Vektoren sind linear unabhängig?
- 2) Welche sind die wichtigen Komponenten in den Daten und wie wichtig sind sie überhaupt?

Dazu benutzen wir den folgenden Code:

```

1  from numpy.linalg import svd
2
3  U, S, Vh = svd(A) #we are not interested in U and V
4
5  figure()
6  I = argsort(1/S) #order the singular values from bigger to smaller
7  plot(arange(len(S)),S[I],"o")
8  xlabel("Number of singular value"); ylabel("Singular value");
9  title("Singular values")
10 show()
11
12 #now we want to study the principal components (stored in V)
13 figure()
14 plot(r,-Vh[0,:],"o", label="First principal component")
15 plot(r,y/norm(y),"+", label="First model vector")
16 plot(r,Vh[1,:],"o", label="Second principal component")
17 plot(r,x/norm(x),"+", label="Second model vector")
18 legend(loc="best")
19 xlabel("Number of measurement"); ylabel("Principal Component");
20 title("Principal Components")
21 show()
22
23 from numpy.linalg import norm
24
25 #now we find the contributions of the principal vectors. compare with
26 formula  $A = \sum \sigma_k u_k v_k^H$ 
27 first_vec_contrib = abs(U[:,0])
28 second_vec_contrib = abs(U[:,1])
29
30 figure()
31 plot(arange(2*n),first_vec_contrib,"+",label="First principal component")
32 plot(arange(2*n),second_vec_contrib,"o",label="Second principal component")
33 xlabel("Measurement"); ylabel("Contributions of principal components")
34 legend(loc="best")
35 show()

```

Die Singulärwertzerlegung der Matrix  $A$  ergibt zwei dominierende Singulärwerte. Die Messungen zeigen sich linear abhängig mit zwei Hauptkomponenten  $\mathbf{u}_1$  und  $\mathbf{u}_2$ .

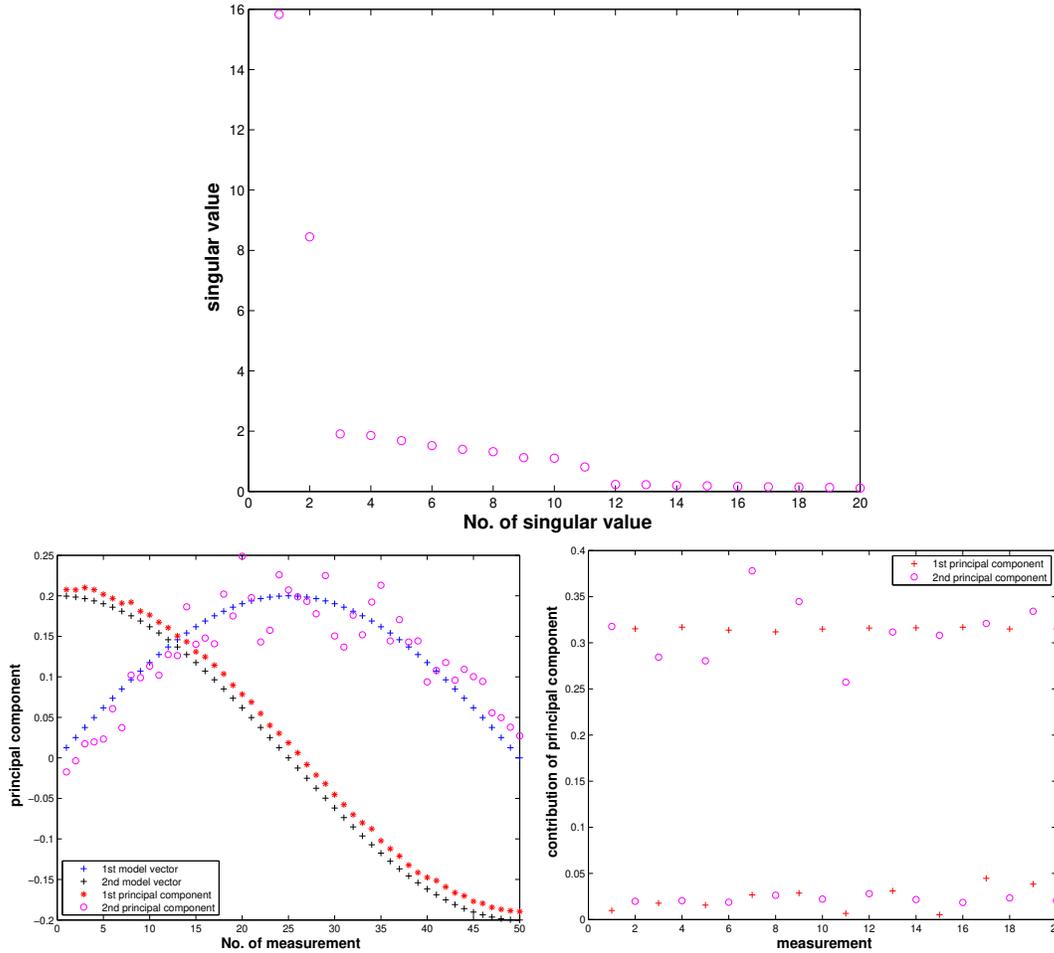


Abb. 5.1.44. Ergebnisse der PCA mittels SVD für die obigen rohen Daten

**Bemerkung 5.1.45.** Es gibt spezialisierte Module für PCA; z.B. `matplotlib.mlab.PCA` oder den Modular Toolkit für Data Processing.

**Bemerkung 5.1.46.** Gegeben  $A \in \mathbb{C}^{m \times n}$ ,  $m > n$ , finde  $x \in \mathbb{C}^n$ ,  $\|x\|_2 = 1$ , sodass  $\|Ax\|_2$  minimal ist. Die SVD hilft, denn unitäre Matrizen erhalten die 2-Norm:

$$\begin{aligned} \min_{\|x\|_2=1} \|Ax\|_2^2 &= \min_{\|x\|_2=1} \|U \Sigma V_x^H\|_2^2 = \min_{\|V_x^H\|_2=1} = \\ &= \min_{\|y\|_2=1} \|\Sigma y\|_2^2 = \min_{\|y\|_2=1} (\sigma_1^2 y_1^2 + \dots + \sigma_n^2 y_n^2) \geq \sigma_n^2, \end{aligned}$$

denn das Minimum ist genau für  $\mathbf{y} = (0, \dots, 0, 1)^T$  erreicht und somit  $x = \mathbf{v}_n$ .

Analog kann man zeigen:

$$\sigma_1 = \max_{\|x\|_2=1} \|Ax\|_2 \quad \text{und} \quad \mathbf{v}_1 = \arg \max_{\|x\|_2=1} \|Ax\|_2 .$$

Somit haben wir

**Theorem 5.1.47.** Sei  $\mathbf{A} \in \mathbb{C}^{m \times n}$ . Dann:  $\|\mathbf{A}\|_2 = \sigma_1(\mathbf{A})$ . Falls  $\mathbf{A}$  invertierbar ist, dann

$$\text{cond}_2(\mathbf{A}) = \frac{\sigma_1}{\sigma_m} .$$

**Bemerkung 5.1.48.** Die Python-Funktionen `norm(A)` und `cond(A)` verwenden `svd(A)`.

**Definition 5.1.49.** Die Frobenius-Norm der  $m \times n$ -Matrix  $\mathbf{A}$  ist

$$\|\mathbf{A}\|_F^2 := \sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2 ,$$

Es ist klar, dass die Frobenius-Norm invariant unter unitären/orthogonalen Transformationen ist. Es ist auch einfach zu sehen, dass

$$\|\mathbf{A}\|_F^2 = \sum_{j=1}^p \sigma_j^2 . \quad (5.1.5)$$

**Theorem 5.1.50.** [Bestapproximation durch Niedrigrangmatrizen]

Für die  $m \times n$ -Matrix  $\mathbf{A}$  mit Rang  $r$  gelte die Singulärwertzerlegung  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^H$ , mit  $m \geq n$  und  $\mathbf{U} = [\mathbf{u}_1 \mathbf{u}_2 \dots \mathbf{u}_m]$  und  $\mathbf{V} = [\mathbf{v}_1 \mathbf{v}_2 \dots \mathbf{v}_n]$ . Für  $k \in \{1, 2, \dots, r\}$  sei die  $m \times k$ -Matrix  $\mathbf{U}_k = [\mathbf{u}_1 \mathbf{u}_2 \dots \mathbf{u}_k]$ , die  $n \times k$ -Matrix  $\mathbf{V}_k = [\mathbf{v}_1 \mathbf{v}_2 \dots \mathbf{v}_k]$  und die  $k \times k$ -Matrix  $\Sigma_k := \text{diag}(\sigma_1, \dots, \sigma_k)$ . Für  $\|\cdot\| = \|\cdot\|_F$  und  $\|\cdot\| = \|\cdot\|_2$  gilt dann:

$$\|\mathbf{A} - \mathbf{U}_k \Sigma_k \mathbf{V}_k^H\| \leq \|\mathbf{A} - \mathbf{B}\| \quad \text{für alle } m \times n\text{-Matrizen } \mathbf{B} \text{ vom Rang } k .$$

*Proof.* Sei  $\mathbf{A}_k = \mathbf{U}_k \Sigma_k \mathbf{V}_k^H$ , das offensichtlich Rang  $k$  hat. Dann

$$\|\mathbf{A} - \mathbf{A}_k\| = \|\Sigma - \mathbf{U}^H \mathbf{U}_k \Sigma_k \mathbf{V}_k^H \mathbf{V}\| = \begin{cases} \sigma_{k+1} & , \text{ for } \|\cdot\| = \|\cdot\|_2 , \\ \sqrt{\sigma_{k+1}^2 + \dots + \sigma_r^2} & , \text{ for } \|\cdot\| = \|\cdot\|_F . \end{cases}$$

Für die Euklidische Norm  $\|\cdot\|_2$ :

Sei die  $m \times n$ -Matrix  $\mathbf{B}$  vom Rang  $k$ , also  $\text{Ker } \mathbf{B}$  hat Dimension  $n - k$  und enthält somit die Menge  $\text{Ker } \mathbf{B} \cap \text{Span}\{\mathbf{v}_1, \dots, \mathbf{v}_{k+1}\}$  mindestens einen Vektor  $\mathbf{x}$  (der Länge  $n$ ) mit  $\|\mathbf{x}\|_2 = 1$ . Dann:

$$\|\mathbf{A} - \mathbf{B}\|_2^2 \geq \|(\mathbf{A} - \mathbf{B})\mathbf{x}\|_2^2 = \|\mathbf{A}\mathbf{x}\|_2^2 = \left\| \sum_{j=1}^{k+1} \sigma_j (\mathbf{v}_j^H \mathbf{x}) \mathbf{u}_j \right\|_2^2 = \sum_{j=1}^{k+1} \sigma_j^2 (\mathbf{v}_j^H \mathbf{x})^2 \geq \sigma_{j+1}^2 ,$$

$$\text{da } \sum_{j=1}^{k+1} (\mathbf{v}_j^H \mathbf{x})^2 = 1 .$$

Für die Frobenius Norm  $\|\cdot\|_F$ : sei  $\{\mathbf{z}_1, \dots, \mathbf{z}_{n-k}\}$  eine Orthonormalbasis in  $\text{Ker } \mathbf{B}$ . Wir setzen die  $n \times n - k$  Matrix  $\mathbf{Z} = [\mathbf{z}_1 \dots \mathbf{z}_{n-k}]$  zusammen. Dann:

$$\|\mathbf{A} - \mathbf{B}\|_F^2 \geq \|(\mathbf{A} - \mathbf{B})\mathbf{Z}\|_F^2 = \|\mathbf{A}\mathbf{Z}\|_F^2 = \sum_{i=1}^{n-k} \|\mathbf{A}\mathbf{z}_i\|_2^2 = \sum_{i=1}^{n-k} \sum_{j=1}^r \sigma_j^2 (\mathbf{v}_j^H \mathbf{z}_i)^2 . \quad \square$$

□

**Bemerkung 5.1.51.** Die Information, die in einer  $m \times n$ -Matrix der Rang  $k$  steckt, kann durch  $k(m+n)$  Zahlen dargestellt werden. Die Approximation mittels Matrizen niedrigen Rangs dient zur Datenkomprimierung.

# Chapter 6

## Ausgleichsrechnung

In diesem Abschnitt beschäftigen wir uns mit dem Problem der optimalen Anpassung einer gegebenen Funktion an einen Datensatz. Um dieses Thema einzuführen wollen wir zuerst mit zwei Beispielen aus der Physik anfangen.

### Nichtlineare Ausgleichsrechnung

In einem Versuch aus dem Anfängerpraktikum für Physiker soll die Wärmeleitfähigkeit von Nickel bestimmt werden. Die Messung erfolgt mit Hilfe einer Kompensationschaltung. Hierbei wirkt ein Stück des Nickelstabes als (wärme)variabler Widerstand, der kompensiert werden soll. Das Experiment wird einmal beim Heizvorgang und einmal beim Abkühlvorgang durchgeführt. Die gemessenen Werte sind in der unteren Tabelle aufgelistet (links: Abkühlvorgang, rechts: Heizvorgang).

<i>Zeit</i> [min]	<i>Stromstärke</i> [mA]	
0	9.66	24.34
5	18.8	18.93
10	22.36	17.09
15	24.07	16.27
20	24.59	15.97
25	24.91	15.91

Temperaturabhängige Stromstärke in Nickel

Es ist bekannt, dass die Korrelation durch eine Exponentialfunktion beschrieben wird:  $I(t) = \alpha \pm \beta e^{(-\gamma t)}$ . Das Ziel ist es, mit Hilfe dieser Funktion und der gemessenen Werte die thermische Leitfähigkeit des Nickels zu bestimmen.

### Lineare Ausgleichsrechnung

Es soll die Temperatur des absoluten Nullpunktes ohne übermäßigen Aufwand bestimmt werden. Um dies zu bewerkstelligen, wird ein Gaskolben mit Helium gefüllt und bei 20°C auf den Luftdruck normiert. Danach misst man die Druckdifferenz am Gefrierpunkt des Wassers und am Siedepunkt von flüssigem Stickstoff.

Unter Verwendung des Gesetzes von Gay-Lussac soll der absolute Nullpunkt abgeschätzt werden. Die einzelnen Messwerte (kochendes Wasser, Gefrierpunkt von Wasser und kochendem Stickstoff) sind in der folgenden Tabelle aufgelistet.

Temp[C]	Druck [mmHg]
98.269	852.7
0	624.5
-194.96	172.7

Temperatur-Druck-Relation für ein ideales Gas.

Sei ein mathematisches Modell  $y = \Psi(\mathbf{a}, x)$  mit  $n$  Parametern  $\mathbf{a} = (a_1, a_2, \dots, a_n)$  gegeben. Für gegebene Werte  $x_i$  kann man  $\Psi(\mathbf{a}, x_i)$  ausrechnen und auch entsprechende physikalische Eigenschaften  $y_i$  messen. Das Ziel der Ausgleichsrechnung ist es, einen Wert für den Parameter  $a$  zu bestimmen, so dass der Fehler

$$\sum_{i=1}^m \|\Psi(\mathbf{a}, x_i) - y_i\|^2$$

minimiert wird.

Wenn die Funktion  $\Psi$  linear vom Parameter  $\mathbf{a}$  abhängt, dann haben wir mit *linearer Ausgleichsrechnung* zu tun.

## 6.1 Lineare Ausgleichsrechnung

**Beispiel 6.1.1.** (Lineare Ausgleichsrechnung)

Seien  $y_i \in \mathbb{R}$  die zu entsprechenden  $\mathbf{x}_i \in \mathbb{R}^n$  ( $i = 1, 2, \dots, m$ ) gemessenen Werte. In  $x_i$  und  $y_i$  befinden sich Messfehler, die im allgemeinsten Falle keine besonderen Bedingungen erfüllen müssen. Sei das zugrunde liegende Modell eine affine Beziehung

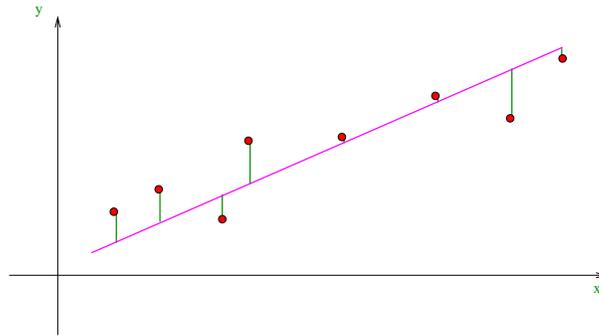
$$y = \mathbf{a}^T \mathbf{x} + c,$$

wobei  $c \in \mathbb{R}$  und  $\mathbf{a} = (a_1, a_2, \dots, a_n)$  die Modellparameter sind, die wir mit Hilfe des Experiments abschätzen möchten.

Typischerweise führt man auch mehr Messungen durch als es Parameter gibt, also  $m \geq n + 1$ . Da die Messungen fehlerbehaftet sind, können die gemessenen Werte  $y_i$  nicht in einer Ebene in  $\mathbb{R}^n$  liegen. Wir können aber eine solche Ebene bauen, so dass die Summe der Abstände von den gemessenen Werten zu dieser Ebene minimal ist. Diese Methode heisst *Methode der kleinsten Quadrate*:

$$(\mathbf{a}, c) = \operatorname{argmin}_{\mathbf{p} \in \mathbb{R}^n, q \in \mathbb{R}} \sum_{i=1}^m |y_i - \mathbf{p}^T \mathbf{x}_i - q|^2 \quad (6.1.1)$$

Die Abbildung 6.1.2 zeigt eine solche lineare Regression in der Dimension  $n = 1$  mit  $m = 8$  Messungen.



**Abb. 6.1.2.** Lineare Regression in der Dimension  $n = 1$  mit  $m = 8$  Messungen

**Beispiel 6.1.3.** (Daten-Fit)

Die affine Relation aus dem vorigen Beispiel kann zu einer Basis-Darstellung verallgemeinert werden. Nehmen wir an, wir haben  $m$  “Knoten”  $(t_i, y_i) \in I \times \mathbb{R} \subset \mathbb{R}^2$  und  $n$  “Basisfunktionen”  $b_j : I \mapsto \mathbb{K}$ . Wir suchen dann  $n$  Koeffizienten  $x_j \in \mathbb{R}$ , so dass

$$\sum_{i=1}^m |f(t_i) - y_i|^2 \rightarrow \min, \quad \text{wobei } f(t) := \sum_{j=1}^n x_j b_j(t). \quad (6.1.2)$$

Im Fall der Polynome  $b_j(t) = t^{j-1}$  spricht man von *Ausgleichspolynomen*. In `python` verwenden wir die `numpy.polyfit` Funktion.

Natürlich kann man diese Idee sowohl auf  $t \in \mathbb{R}^d$  als auch auf  $y \in \mathbb{R}^d$  anwenden.

**Beispiel 6.1.4.** Im Beispiel 6.1.1 könnten wir  $\mathbf{a}$  und  $c$  finden, indem wir das lineare Gleichungssystem

$$\begin{bmatrix} \mathbf{x}_1^T & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^T & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ c \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$

lösen, aber wir haben mehr Gleichungen als Unbekannten, falls  $m > n + 1$ . Im Beispiel 6.1.3 liefert dieselbe Idee:

$$\begin{bmatrix} b_1(t_1) & \dots & b_n(t_1) \\ \vdots & & \vdots \\ b_1(t_m) & \dots & b_n(t_m) \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix},$$

mit demselben Schwierigkeitsgrad im Fall  $m > n$ .

**Aufgabe (Lineare Ausgleichsrechnung):**

gegeben:  $\mathbf{A} \in \mathbb{K}^{m,n}$ ,  $m, n \in \mathbb{N}$ ,  $\mathbf{b} \in \mathbb{K}^m$ ,  
finde:  $\mathbf{x} \in \mathbb{K}^n$ , sodass

$$(i) \quad \|\mathbf{Ax} - \mathbf{b}\|_2 = \inf\{\|\mathbf{Ay} - \mathbf{b}\|_2 : \mathbf{y} \in \mathbb{K}^n\}, \quad (6.1.3)$$

$$(ii) \quad \|\mathbf{x}\|_2 \text{ ist minimal.}$$

**Bemerkung 6.1.5.** Lassen Sie uns die ersten zwei Beispiele umformulieren:

$$\text{Beispiel 6.1.1: } \mathbf{A} = \begin{bmatrix} \mathbf{x}_1^T & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^T & 1 \end{bmatrix} \in \mathbb{R}^{m,n+1}, \mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^m, \mathbf{x} = \begin{bmatrix} \mathbf{a} \\ c \end{bmatrix} \in \mathbb{R}^{n+1}.$$

$$\text{Beispiel 6.1.3: } \mathbf{A} = \begin{bmatrix} b_1(t_1) & \dots & b_n(t_1) \\ \vdots & & \vdots \\ b_1(t_m) & \dots & b_n(t_m) \end{bmatrix} \in \mathbb{R}^{m,n}, \mathbf{b} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^m, \mathbf{x} \in \mathbb{R}^n.$$

**Bemerkung 6.1.6.** Die Norm des Residuums  $\|\mathbf{b} - \mathbf{Ax}\|_2$  erlaubt es uns, die Qualität des Modells zu beurteilen.

**Bemerkung 6.1.7.** In python verwenden wir die `numpy.linalg.lstsq` Funktion.

**Bemerkung 6.1.8.** Wir werden (durch Konstruktion) sehen, dass das lineare Ausgleichsproblem (6.1.3) eine eindeutige Lösung besitzt. Damit kann man die Pseudoinverse (Moore-Penrose) einer Matrix definieren.

In python stehen dafür `numpy.linalg.pinv` und `scipy.linalg.pinv2` zur Verfügung.

## 6.1.1 Normalgleichungen

Sei

$$\begin{aligned} f(x) &= \|\mathbf{b} - \mathbf{Ax}\|_2^2 = (\mathbf{b} - \mathbf{Ax})^H (\mathbf{b} - \mathbf{Ax}) = \\ &= (\mathbf{b}^H - \mathbf{x}^H \mathbf{A}^H) (\mathbf{b} - \mathbf{Ax}) = \mathbf{b}^H \mathbf{b} - \mathbf{b}^H \mathbf{Ax} - \mathbf{x}^H \mathbf{A}^H \mathbf{b} + \mathbf{x}^H \mathbf{A}^H \mathbf{Ax} = \\ &= \mathbf{b}^H \mathbf{b} - \mathbf{b}^H \mathbf{Ax} - \mathbf{x}^H \mathbf{A}^H \mathbf{b} + \mathbf{x}^H \mathbf{A}^H \mathbf{Ax} = \mathbf{b}^H \mathbf{b} - 2\mathbf{b}^H \mathbf{Ax} + \mathbf{x}^H \mathbf{A}^H \mathbf{Ax}. \end{aligned}$$

Wenn  $f(x)$  minimal ist, dann gilt es, dass  $Df(x) = 0$ , also

$$2(\mathbf{A}^H \mathbf{A})\mathbf{x} - 2\mathbf{A}^H \mathbf{b} = 0,$$

und somit muss

$$\mathbf{A}^H \mathbf{Ax} = \mathbf{A}^H \mathbf{b} \tag{6.1.4}$$

erfüllt werden.

**Definition 6.1.9.** Die Gleichung (6.1.4) heisst *Normalgleichung*.

**Bemerkung 6.1.10.** Die Matrix  $\mathbf{A}^H \mathbf{A}$  ist Hermite-symmetrisch und, wenn  $\mathbf{A}$  vollen Rang hat, dann auch positiv definit; somit besitzt das System  $\mathbf{A}^H \mathbf{Ax} = \mathbf{A}^H \mathbf{b}$  eine eindeutige Lösung. Sie lässt sich mit Hilfe der LU-Zerlegung oder, noch besser, der Cholesky-Zerlegung bestimmen.

Der Nachteil der Normalgleichungen ist vor allem ihre schlechte Konditionierung und die damit verbundene numerische Instabilität. In der Tat gilt für die Kondition von  $\mathbf{A}^H \mathbf{A}$ :

$$\begin{aligned} \text{cond}(\mathbf{A}^H \mathbf{A}) &= \text{cond}(\mathbf{V} \boldsymbol{\Sigma} \mathbf{U}^H \mathbf{U} \boldsymbol{\Sigma} \mathbf{V}^H) = \text{cond}(\mathbf{V} \boldsymbol{\Sigma}^2 \mathbf{V}^H) = \\ \text{cond}(\boldsymbol{\Sigma}^2) &= \frac{\sigma_1^2}{\sigma_n^2} = \text{cond}(\mathbf{A})^2. \end{aligned}$$

Es ist deshalb nicht empfehlenswert, Normalgleichungen zur Lösung von linearen Ausgleichsproblemen zu verwenden, die durch eine schlecht konditionierte Matrix dargestellt werden. In diesem Falle läuft man Gefahr, im Verlauf der Berechnungen eine enorme Vergrößerung der Rundungsfehler auftreten zu lassen, was zu einer falschen Lösung führen würde.

Ein weiteres Problem der Normalgleichungen tritt auf, wenn die Matrix  $\mathbf{A}$  dünn besetzt ist.<sup>1</sup> In der Tat, wenn  $\mathbf{A}$  dünn besetzt ist, so ist es  $\mathbf{A}^H \mathbf{A}$  im Allgemeinen nicht. Dies kann, bei grossem  $\mathbf{A}$ , zu einem Speicherüberlauf oder zu sehr langen Rechenzeiten führen. Weiterhin wird so die Möglichkeit der Anwendung effizienter direkter Lösungsmethoden für dünnbesetzte Matrizen verschwendet.

**Bemerkung 6.1.11.** Eine Möglichkeit, die Berechnung von  $\mathbf{A}^H \mathbf{A}$  umzugehen, soll im Folgenden erläutert werden.

Das Residuum  $\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}$  wird als eine weitere Unbekannte behandelt. Man kann die Gleichung (6.1.4) folgendermassen umschreiben:

$$\mathbf{A}^H \mathbf{A}\mathbf{x} = \mathbf{A}^H \mathbf{b} \Leftrightarrow \mathbf{B} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} := \begin{bmatrix} -\mathbf{I} & \mathbf{A} \\ \mathbf{A}^H & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}. \quad (6.1.5)$$

Allgemeiner definiert man  $\mathbf{r} := \frac{1}{a}(\mathbf{A}\mathbf{x} - \mathbf{b})$ , wobei  $a > 0$  gilt; dann kann man in Gleichung (6.1.5) die Matrix  $\mathbf{B}$  durch

$$\mathbf{B}_a := \begin{bmatrix} -a\mathbf{I} & \mathbf{A} \\ \mathbf{A}^H & \mathbf{0} \end{bmatrix}$$

ersetzen. Dabei wählt man  $a$  so, dass  $\kappa(\mathbf{B}_a)$  minimal wird. Die Umformung (6.1.5) von Gleichung (6.1.4) sowie ihre Variante führen bei dünn besetztem  $\mathbf{A}$  wieder zu einer dünn besetzten Matrix  $\mathbf{B}_a$  und sind somit mit Hilfe von effizienten Verfahren für dünn besetzte Matrizen lösbar.

<sup>1</sup>Englisch: *sparse matrix*

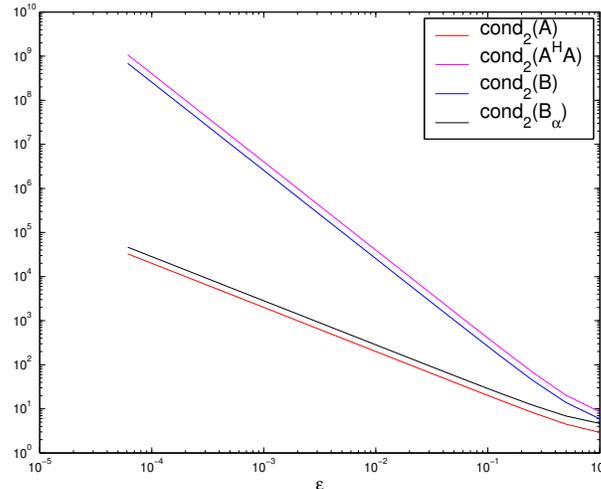


Abb. 6.1.12. Kondition von Normalgleichungen für verschiedene  $\epsilon$ .

**Beispiel 6.1.13.** Die bessere Kondition des Systems 6.1.5 lässt sich leicht an einem Beispiel zeigen. Sei

$$\mathbf{A} = \begin{bmatrix} 1 + \epsilon & 1 \\ 1 - \epsilon & 1 \\ \epsilon & \epsilon \end{bmatrix}.$$

Weiterhin wird für  $\mathbf{B}_a$   $\epsilon a = \frac{\|\mathbf{A}\|}{\sqrt{2}}$  verwendet (ohne Beweis). Ein Vergleich der Konditionszahlen für verschiedene  $\epsilon$  ist in Abbildung 6.1.12 aufgezeichnet.

Der folgende Code wurde verwendet:

Code 6.1.14: Vergleich der Kondition von verschiedenen Matrizen

```

1 from numpy import zeros, sqrt, zeros_like, linspace, shape, dot, conj, eye
2 from numpy.linalg import cond, norm
3 from matplotlib.pyplot import figure, loglog, legend, xlabel, ylabel, show
4 from matplotlib import rcParams #this line only produces bigger and clearer
   images for the notebook
5 rcParams['figure.figsize'] = (20.0, 10.0)
6
7 #first we define functions for building the two matrices
8
9 def A(eps):
10     A = zeros((3,2))
11     A[0,0] = 1. + eps; A[0,1] = 1.0
12     A[1,0] = 1. - eps; A[1,1] = 1.0
13     A[2,0] = eps; A[2,1] = eps
14     return A
15
16 def B(eps, adapt): #adapt: boolean value for choosing a = 1. or adapting a to
   minimize cond
17     a = 1.0
18     mat = A(eps)
19     if adapt==True:
20         a = eps*norm(mat)/sqrt(2.0)

```

```

21     dimB = shape(mat)[0]+shape(mat)[1]
22     B = zeros((dimB,dimB))
23     B[:shape(mat)[0],:shape(mat)[0]] = -a*eye(shape(mat)[0])
24     B[:shape(mat)[0],shape(mat)[0]:] = mat
25     B[shape(mat)[0]:,:shape(mat)[0]] = conj(mat).T #hermite transpose, .H does
not work with ndarray
    return B
27
28     epsilons = linspace(1e-5,1,100)
29
30     #containers for condition of the different matrices as a function of epsilon
31     condA = zeros_like(epsilons); condAHA = zeros_like(epsilons)
32     condB = zeros_like(epsilons); condBa = zeros_like(epsilons)
33
34     #print(B(A(0.04), adapt=True))
35
36     for k in range(100):
37         mat = A(epsilons[k])
38         condA[k] = cond(mat)
39         condAHA[k] = cond(dot(conj(mat).T,mat))
40         condB[k] = cond(B(epsilons[k], adapt = False))
41         condBa[k] = cond(B(epsilons[k], adapt = True))
42
43     #now we plot the results
44     figure()
45     loglog(epsilons,condA,label="cond($A$)")
46     loglog(epsilons,condAHA,label="cond($A^HA$)")
47     loglog(epsilons,condB,label="cond($B$)")
48     loglog(epsilons,condBa,label="cond($B_a$)")
49     xlabel("$\epsilon$"); ylabel("cond")
50     legend(loc="best")
51     show()

```

## 6.1.2 Lösung mittels orthogonaler Transformationen

Aus Beispiel 6.1.13 sieht man, dass die Normalengleichungen schlecht zur numerischen Lösung des Ausgleichsproblems geeignet sind. Auch das LU-Verfahren kann für eine Matrix mit  $m > n$  nicht sinnvoll angewandt werden. Man benötigt also ein Verfahren, welches numerisch stabil ist und gleichzeitig für überbestimmte Systeme geeignet ist.

Wir betrachten wieder

$$\mathbf{r} = \mathbf{Ax} - \mathbf{b}$$

und versuchen,  $\|\mathbf{r}\|_2^2$  zu minimieren. Es ist bekannt, dass unitäre/orthonormale Matrizen die Norm unverändert lassen. Betrachtet man die vollständige QR-Zerlegung von  $\mathbf{A}$ , so lässt sich ein Lösungsansatz herleiten, der stabiler als die Normalengleichungen ist. Wenn

$$\mathbf{A} = \mathbf{QR} = \mathbf{Q} \begin{bmatrix} \tilde{\mathbf{R}} \\ 0 \end{bmatrix}$$

mit der unitären Matrix  $\mathbf{Q}$ , dann führt dies zu einer Umformulierung von (6.1.3):

$$\begin{aligned} \|\mathbf{r}\|_2^2 &= \|\mathbf{Ax} - \mathbf{b}\|_2^2 = \|\mathbf{QRx} - \mathbf{QQ}^H\mathbf{b}\|_2^2 = \\ &\|\mathbf{Q}(\mathbf{Rx} - \mathbf{Q}^H\mathbf{b})\|_2^2 = \|\mathbf{Rx} - \tilde{\mathbf{b}}\|_2^2, \end{aligned} \tag{6.1.6}$$

wobei hier  $\tilde{\mathbf{b}} = \mathbf{Q}^H\mathbf{b}$  ist. Somit gilt f

$$\|\mathbf{Ax} - \mathbf{b}\|_2 \rightarrow \min \Leftrightarrow \left\| \begin{pmatrix} \mathbf{R} \\ \mathbf{0} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} - \begin{pmatrix} \tilde{b}_1 \\ \vdots \\ \tilde{b}_m \end{pmatrix} \right\|_2 \rightarrow \min$$

und eine explizite Lösung des Ausgleichsproblems ist

$$\mathbf{Rx} - \tilde{\mathbf{b}} = \begin{bmatrix} \tilde{\mathbf{R}} \\ \mathbf{0} \end{bmatrix} \mathbf{x} - \begin{bmatrix} \tilde{\mathbf{b}}_1 \\ \tilde{\mathbf{b}}_2 \end{bmatrix} \Rightarrow \tilde{\mathbf{R}}\mathbf{x} = \tilde{\mathbf{b}}_1 \Leftrightarrow \mathbf{x} = \tilde{\mathbf{R}}^{-1}\tilde{\mathbf{b}}_1.$$

$$\mathbf{x} = \begin{pmatrix} \tilde{\mathbf{R}} \\ \mathbf{0} \end{pmatrix}^{-1} \begin{bmatrix} \tilde{b}_1 \\ \vdots \\ \tilde{b}_n \end{bmatrix}, \quad \text{Residuum } \mathbf{r} = \mathbf{Q} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \tilde{b}_{n+1} \\ \vdots \\ \tilde{b}_m \end{bmatrix}.$$

Weiterhin folgt für die Norm des Residuums  $\|\mathbf{r}\|_2 = \|\mathbf{Q}\tilde{\mathbf{b}}_2\|_2 = \|\tilde{\mathbf{b}}_2\|_2 = \sqrt{\tilde{b}_{n+1}^2 + \dots + \tilde{b}_m^2}$ .

Da die QR-Zerlegung relativ teuer ist und die Matrix  $\mathbf{Q}$  nicht benötigt wird, muss sie nicht gespeichert werden. Führt man die QR-Zerlegung der Matrix  $\mathbf{A}$  durch (z.B. mittels Householder-Spiegelungen) und wendet die Transformationen in jedem Schritt auch auf  $\mathbf{b}$  an, erhält man die rechte Seite von Gleichung (6.1.6) ohne die Matrix  $\mathbf{Q}$  je vollständig im Speicher zu haben, was bei grossen Matrizen einen Geschwindigkeitsvorteil bringt.

Wenn die Matrix  $\mathbf{A}$  nicht vollen Rang hat (was in konkreten Fällen vorkommen kann) ist es besser, die Singulärwertzerlegung zu verwenden. Diese steckt

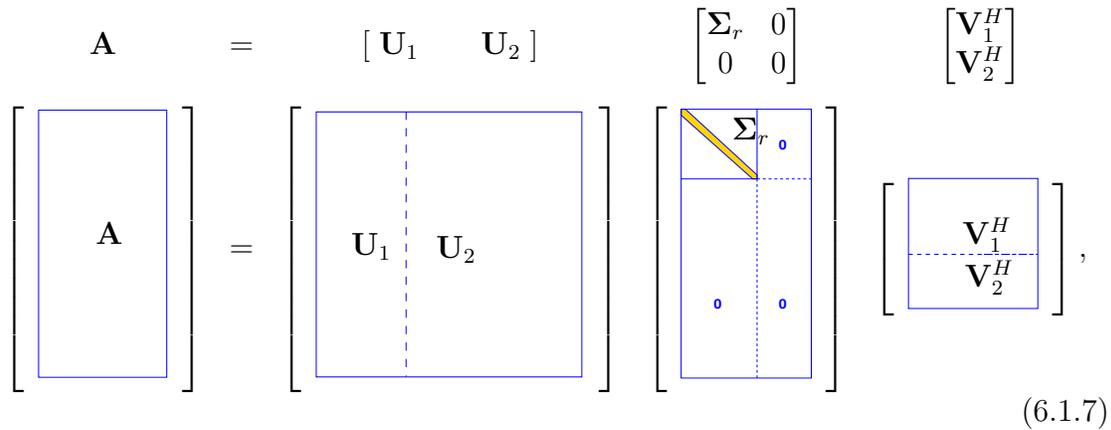
auch hinter professionellen Lösern wie `numpy.linalg.lstsq`. Die Normerhaltung unitärer/orthogonaler Transformationen liefert wie vorher:

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \|\mathbf{U}\Sigma\mathbf{V}^H\mathbf{x} - \mathbf{b}\|_2 = \|\Sigma\mathbf{V}^H\mathbf{x} - \mathbf{U}^H\mathbf{b}\|_2.$$

Wenn  $\text{rank}(A) = r < \min(m, n)$ , dann

$$\Sigma = \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix},$$

und alle Zeilenvektoren  $\mathbf{v}_j$  der Matrix  $\mathbf{V}$ , für die gilt:  $j > r$ , liegen im Kern von  $\mathbf{A}$ . Ähnliches gilt auch für die Matrix  $\mathbf{U}$ . Beide werden folgendermassen unterteilt:

$$\mathbf{A} = [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{bmatrix}$$


(6.1.7)

mit je orthonormalen Spalten in den Matrizen  $U_1, U_2, V_1$  und  $V_2$ :

$$\mathbf{U}_1 \in \mathbb{K}^{m \times r}, \mathbf{U}_2 \in \mathbb{K}^{m \times n-r}, \mathbf{V}_1 \in \mathbb{K}^{n \times r}, \mathbf{V}_2 \in \mathbb{K}^{n \times n-r}.$$

Damit gilt für das Residuum:

$$\|\mathbf{Ax} - \mathbf{b}\|_2 = \left\| [\mathbf{U}_1 \quad \mathbf{U}_2] \begin{bmatrix} \Sigma_r & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{V}_1^H \\ \mathbf{V}_2^H \end{bmatrix} \mathbf{x} - \mathbf{b} \right\|_2 = \left\| \begin{bmatrix} \Sigma_r \mathbf{V}_1^H \mathbf{x} \\ 0 \end{bmatrix} - \begin{bmatrix} \mathbf{U}_1^H \mathbf{b} \\ \mathbf{U}_2^H \mathbf{b} \end{bmatrix} \right\|_2.$$

Somit wird das Residuum genau dann minimal, wenn

$$\Sigma_r \mathbf{V}_1^H \mathbf{x} = \mathbf{U}_1^H \mathbf{b} \Leftrightarrow \mathbf{V}_1^H \mathbf{x} = \Sigma_r^+ \mathbf{U}_1^H \mathbf{b}, \quad (6.1.8)$$

wobei  $\Sigma_r^+ = \text{diag}(\frac{1}{\sigma_1}, \dots, \frac{1}{\sigma_r})$  die "Inverse" von  $\Sigma_r$  ist. Das Residuum dazu ist  $\|\mathbf{U}_2^H \mathbf{b}_2\|_2$ . Wir definieren nun

$$\mathbf{x} = \mathbf{V}_1 \Sigma_r^+ \mathbf{U}_1^H \mathbf{b}, \quad (6.1.9)$$

das offensichtlich die Gleichung (6.1.8) erfüllt ( $\mathbf{V}_1^H \mathbf{V}_1 = \mathbf{I}_r$ ). Das so definierte  $x$  liegt in  $\text{Bild}(\mathbf{A}^H) = \text{span}\{\mathbf{v}_1, \dots, \mathbf{v}_r\}$  und somit steht orthogonal auf  $\text{Ker}(\mathbf{A}) = \text{span}\{v_{r+1}, \dots, v_n\}$ . Dass die Lösung des Minimierungsproblems (i) von (6.1.3) nicht eindeutig ist, sieht man bereits im Fundamentalsatz der Linearen Algebra: jedes  $x_g = x + x_n$  mit  $x_n$  in  $\text{Ker}(\mathbf{A})$  beliebig löst das Ausgleichsproblem. Die in (6.1.9) definierte Lösung steht orthogonal auf  $\text{Ker}(\mathbf{A}) = \text{span}\{v_{r+1}, \dots, v_n\}$  und somit minimiert sie auch die Euklidische Norm  $\|x\|_2$  und löst komplett die Lineare Ausgleichsrechnung (6.1.3). Die Bedingung (ii) legt also genau eine spezielle Lösung des einfachen Ausgleichsproblems fest.

**Definition 6.1.15.** Die Matrix  $\mathbf{V}_1 \boldsymbol{\Sigma}_r^+ \mathbf{U}_1^H$  in Gleichung (??) wird als *Pseudoinverse* der Matrix  $\mathbf{A}$  bezeichnet und formal definiert durch:

$$\mathbf{A}^+ = (\mathbf{A}^H \mathbf{A})^{-1} \mathbf{A}^H = \mathbf{V}_1 \boldsymbol{\Sigma}_r^+ \mathbf{U}_1^H.$$

Die Lösung von (6.1.3) mittels Singulärwertzerlegung wird also auf die Bestimmung der Pseudoinversen von  $\mathbf{A}$  und die Berechnung von  $\mathbf{x}$  mittels dieser zurückgeführt. Eine mögliche Implementierung, die auf dem *numerischen Rang*

$$r = \max\{i : \sigma_i / \sigma_1 > \text{tol}\}$$

einer Matrix beruht, ist:

**Code 6.1.16:** Lösung des linearen Ausgleichsproblems mit Singulärwertzerlegung

```

1 from numpy.linalg import svd
2 from numpy import dot, where
3
4 def lsqsvd(A,b,eps=1e-6):
5     U,s,Vh = svd(A) # python routine for singular value decomposition
6     r = 1+where(s/s[0]>eps)[0].max() # numerical rank
7     y = dot(Vh[:,r,:].T, dot(U[:,r].T,b)/s[r])
8     return y

```

Im Allgemeinen lassen sich die drei Verfahren in zwei Anwendungskategorien einordnen.

1.  $\mathbf{A} \in \mathbb{K}^{m \times n}$  ist voll besetzt und  $n$  ist klein ( $m \gg n$ ).
2.  $\mathbf{A} \in \mathbb{K}^{m \times n}$  ist dünn besetzt und  $m, n$  sind gross.

Im ersten Fall bieten sich QR-Zerlegung und Singulärwertzerlegung aufgrund ihrer numerischen Stabilität an. Da beide jedoch die spezifische Struktur dünn besetzter Matrizen nicht ausnutzen können, wendet man auf Probleme der zweiten Art Normalgleichungen an.

**Beispiel 6.1.17.** Wir kehren hier zurück zum zweiten Beispiel. Aus den Werten der Tabelle erhalten wir die Matrix  $\mathbf{A}$  und den Vektor  $\mathbf{b}$  als:

$$\mathbf{A} = \begin{bmatrix} 98.269 & 1 \\ 0 & 1 \\ -194.96 & 1 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 852.7 \\ 624.5 \\ 172.7 \end{bmatrix}.$$

Die QR-Zerlegung ergibt:

$$\mathbf{Q} = \begin{bmatrix} 0.45010 & 0.71625 \\ 0.00000 & 0.59720 \\ -0.89298 & 0.36102 \end{bmatrix} \quad \mathbf{R} = \begin{bmatrix} 218.32590 & -0.44287 \\ 0.00000 & 1.67447 \end{bmatrix}.$$

Durch Auflösen nach  $\tilde{\mathbf{b}} = \mathbf{Q}^T \mathbf{b}$  und Rückwärtselimination von  $\mathbf{R}\mathbf{x} = \tilde{\mathbf{b}}$  erhalten wir somit schliesslich für die Gleichung  $f(x) = mx + c$  die Koeffizienten

$$m = \mathbf{x}_1 = 2.3188$$

$$c = \mathbf{x}_2 = 624.7017.$$

Löst man dann noch  $f(t) = 0$  nach  $t$  auf, erhält man als Näherung für den absoluten Nullpunkt  $t = -269.41$ , was eine relativ gute Approximation ist ( $T_0 = -273.15$ )

**Code 6.1.18:** Lösung eines linearen Ausgleichsproblems mittels verschiedener Methoden

```

1 from numpy.linalg import qr, solve, lstsq
2 from numpy import array, dot
3
4 A = array([[98.269,1.0],[0.0,1.0],[-194.96,1.0]])
5 b = array([852.7,624.5,172.7])
6
7 #first: with qr-decomposition:
8 Q, R = qr(A)
9 b_tilde = dot(Q.T,b)
10 x_qr = solve(R,b_tilde)
11
12 print("QR-decomposition:")
13 print("m = x_1 = ", x_qr[0])
14 print("c = x_2 = ", x_qr[1])
15
16 #now with the pseudo inverse method, using the function that we defined in the
17 previous code:
18 x_lsqsvd = lsqsvd(A,b)
19
20 print("Pseudo inverse:")
21 print("m = x_1 = ", x_lsqsvd[0])
22 print("c = x_2 = ", x_lsqsvd[1])
23
24 #now with Numpy's function (least squares):
25 x_lstsq = lstsq(A,b)
26
27 print("Numpy - lstsq:")
28 print("m = x_1 = ", x_lstsq[0][0])
29 print("c = x_2 = ", x_lstsq[0][1])

```

### 6.1.3 Totale Ausgleichsrechnung

Was soll man tun, wenn auch die Matrix und die rechte Seite in  $\mathbf{Ax} = \mathbf{b}$  fehlerbehaftet sind? Eine Lösung existiert, auch im Sinne der kleinsten Quadrate, nur wenn  $\mathbf{b}$  in Bild  $\mathbf{A}$  liegt. Die Messfehler in  $\mathbf{A}$  und  $\mathbf{b}$  verhindern dann eine Lösung. In diesem Fall ersetzt man das System  $\mathbf{Ax} = \mathbf{b}$  mit einem neuen System  $\hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$ , das *so nah wie möglich* am ursprünglichen System liegt, so dass  $\hat{\mathbf{b}} \in \text{Bild } \hat{\mathbf{A}}$ . Die mathematische Formulierung lautet:

Gegeben die  $m \times n$ -Matrix  $\mathbf{A}$  mit Rang  $n$  und der Vektor  $\mathbf{b}$  der Länge  $m$ , finde  $\widehat{\mathbf{A}}$  und  $\widehat{\mathbf{b}} \in \text{Bild } \widehat{\mathbf{A}}$ , so dass folgende Norm minimal wird

$$\left\| [\mathbf{A} \ \mathbf{b}] - [\widehat{\mathbf{A}} \ \widehat{\mathbf{b}}] \right\|_F.$$

Wir notieren  $\mathbf{C} = [\mathbf{A} \ \mathbf{b}]$  und  $\widehat{\mathbf{C}} = [\widehat{\mathbf{A}} \ \widehat{\mathbf{b}}]$ . Die Bedingung  $\widehat{\mathbf{b}} \in \text{Bild } \widehat{\mathbf{A}}$  bedeutet dann, dass  $\widehat{\mathbf{C}}$  Rang  $n$  hat, so dass das Problem sich umschreiben lässt

$$\min_{\text{Rang } \widehat{\mathbf{C}}=n} \left\| \mathbf{C} - \widehat{\mathbf{C}} \right\|_F.$$

Das Theorem 5.1.50 weist uns den Weg zur Lösung. Die Singulärwertzerlegung

$$\mathbf{C} = \mathbf{U}\Sigma\mathbf{V}^H = \sum_{j=1}^{n+1} \sigma_j (\mathbf{u})_j (\mathbf{v})_j^H$$

gibt das Optimum

$$\widehat{\mathbf{C}} = \sum_{j=1}^n \sigma_j (\mathbf{u})_j (\mathbf{v})_j^H$$

und die Orthogonalität der Vektoren  $\mathbf{v}_j$  ergibt  $\widehat{\mathbf{C}}\mathbf{v}_{n+1} = 0$ . Falls  $v_{n+1,n+1} \neq 0$  dann haben wir auch die Lösung des Systems  $\widehat{\mathbf{A}}\widehat{\mathbf{x}} = \widehat{\mathbf{b}}$  als  $\widehat{\mathbf{x}} = \mathbf{v}_{n+1}/v_{n+1,n+1}$ .

## 6.2 Nichtlineare Ausgleichsrechnung

Wenn das zugrunde liegende Modell für das Ausgleichsproblem nicht linear in den Parametern ist, dann reden wir von *nichtlinearer Ausgleichsrechnung*. Im Folgenden werden wir die gesuchten Parameter mit  $\mathbf{x}$  und die Messpunkte mit  $t_i$  notieren.

**Aufgabe (nichtlineare Ausgleichsrechnung):** Finde  $\mathbf{x} \in \mathbb{R}^n$ , so dass

$$\mathbf{x} = \underset{\mathbf{u} \in \mathbb{R}^n}{\text{argmin}} \left( \sum_{i=1}^m |f(t_i, \mathbf{u}) - y_i|^2 \right) = \underset{\mathbf{u} \in \mathbb{R}^n}{\text{argmin}} \frac{1}{2} \|F(\mathbf{u})\|_2^2. \quad (6.2.1)$$

Hier ist  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$  von der Form:

$$F(\mathbf{x}) = \begin{bmatrix} f(t_1, \mathbf{x}) - y_1 \\ \vdots \\ f(t_m, \mathbf{x}) - y_m \end{bmatrix}.$$

Für das weitere Vorgehen definieren wir  $\Phi(\mathbf{x}) := \frac{1}{2} \|F(\mathbf{x})\|_2^2$ .

**Bemerkung 6.2.1.** Die Existenz und Eindeutigkeit einer Lösung von (6.2.1) ist im Allgemeinen nicht klar. Aus der Analysis ist bekannt (Satz über implizite Funktionen), dass es eine Umgebung  $U_\epsilon(\mathbf{x})$  gibt, in welcher  $\text{rang}(D\Phi(\mathbf{x})) = n$  ist,  $D\Phi(\mathbf{x})$  also in  $U_\epsilon(\mathbf{x})$  invertierbar ist. Da dies aber eine lokale Eigenschaft von  $\Phi(\mathbf{x})$  ist, lässt sich lediglich folgern, dass eine Lösung lokal eindeutig ist, wenn sie existiert.

### 6.2.1 Newton-Verfahren im mehreren Dimensionen

Gesucht wird  $\mathbf{x} \in \mathbb{R}^n$ , so dass gilt:  $\Phi(\mathbf{x}) = \min$ . Aus der Analysis ist bekannt, dass eine notwendige Bedingung dafür ist  $\text{grad } \Phi(\mathbf{x}) = 0$ . Wir haben das Problem (6.2.1) somit auf eine Nullstellensuche im  $\mathbb{R}^n$  reduziert. Diese lässt sich mit Hilfe des Newton-Verfahrens lösen:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - (D \text{grad } \Phi(\mathbf{x}))^{-1} \text{grad } \Phi(\mathbf{x}). \quad (6.2.2)$$

$$(6.2.3)$$

Wir rechnen dann aus:

$$\text{grad } \Phi(\mathbf{x}) = (DF(\mathbf{x}))^T F(\mathbf{x}) \quad (6.2.4)$$

$$D(\text{grad } \Phi)(\mathbf{x}) = (DF(\mathbf{x}))^T DF(\mathbf{x}) + \sum_{j=1}^m F_j(\mathbf{x}) D^2 F_j(\mathbf{x}). \quad (6.2.5)$$

In der Gleichung (6.2.5) taucht die Hesse-Matrix der Funktion  $\Phi$  auf:

$$H_{\Phi}(\mathbf{x}) := D(\text{grad } \Phi)(\mathbf{x}). \quad (6.2.6)$$

Aus (6.2.2), (6.2.3) und (6.2.5) mit der Notation (6.2.6) können wir die Iteration umschreiben in:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - (H_{\Phi}(\mathbf{x}))^{-1} \text{grad } \Phi(\mathbf{x}). \quad (6.2.7)$$

Andererseits kann (6.2.2) auch als Minimierung der Quadratischen Form  $Q(\mathbf{x}) := \Phi(\mathbf{x}^{(k)}) + (\text{grad } \Phi(\mathbf{x}^{(k)}))^T \mathbf{s} + \frac{1}{2} \mathbf{s}^T H_{\Phi}(\mathbf{x}^{(k)}) \mathbf{s}$ , die  $\Phi$  lokal approximiert (Taylor 2. Ordnung), betrachtet werden:

$$\text{grad } Q(\mathbf{x}) = 0 \Leftrightarrow H_{\Phi}(\mathbf{x}^{(k)}) \mathbf{s} + \text{grad } \Phi(\mathbf{x}^{(k)}) = 0,$$

woraus wiederum (6.2.7) folgt.

### 6.2.2 Gauss-Newton Verfahren

Das reine Newton-Verfahren für (6.2.2) ist aufwendig, selbst wenn man die Matrixinversion durch das Lösen eines LGS ersetzt. Zudem ist  $F(\mathbf{x})$  nicht immer zweimal stetig differenzierbar auf  $U_{\epsilon}(\mathbf{x}_0)$ , was das Aufstellen der Hesse-Matrix unmöglich macht. Um beide Probleme umzugehen, bedient man sich einer wohlbekannten Idee aus der Analysis.

Die komplizierte Funktion  $F(\mathbf{x})$  wird lokal (auf einer Umgebung der Lösung  $\mathbf{x}_0$ ) durch eine lineare Funktion approximiert, das heißt linearisiert:

$$\begin{aligned} F(\mathbf{x}) &\approx F(\mathbf{y}) + DF(\mathbf{y})(\mathbf{x} - \mathbf{y}) = F(\mathbf{y}) + \mathbf{J}_F(\mathbf{y})(\mathbf{x} - \mathbf{y}) \\ &= F(\mathbf{y}) + \mathbf{J}_F(\mathbf{y})\mathbf{x} - \mathbf{J}_F(\mathbf{y})\mathbf{y}. \end{aligned} \quad (6.2.8)$$

Definiert man  $\mathbf{A} := \mathbf{J}_F(\mathbf{y})$  und  $\mathbf{b} := \mathbf{J}_F(\mathbf{y})\mathbf{y} - F(\mathbf{y})$  so erhält man für (6.2.1) unter Zuhilfenahme von (6.2.8) ein lineares Ausgleichsproblem:

$$\operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} \frac{1}{2} \|F(\mathbf{x})\|_2^2 \approx \operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} \frac{1}{2} \|F(\mathbf{y}) + \mathbf{J}_F(\mathbf{y})\mathbf{x} - \mathbf{J}_F(\mathbf{y})\mathbf{y}\|_2^2 = \operatorname{argmin}_{\mathbf{x} \in \mathbb{R}^n} \frac{1}{2} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2^2,$$

wobei  $\mathbf{y}$  eine Näherung der Lösung  $\mathbf{x}$  von (6.2.1) ist.

Setzt man  $\mathbf{x} = \mathbf{x}^{k+1} = \mathbf{x}^k - \mathbf{s}$  und  $\mathbf{y} = \mathbf{x}^k$ , dann erhält man

$$\operatorname{argmin}_{\mathbf{z} \in \mathbb{R}^n} \frac{1}{2} \|F(\mathbf{x}^k - \mathbf{z})\| \approx \operatorname{argmin}_{\mathbf{z} \in \mathbb{R}^n} \frac{1}{2} \|F(\mathbf{x}^k) + \mathbf{J}_F(\mathbf{x}^k)(\mathbf{x}^k - \mathbf{z} - \mathbf{x}^k)\| \quad (6.2.9)$$

$$= \operatorname{argmin}_{\mathbf{z} \in \mathbb{R}^n} \frac{1}{2} \|F(\mathbf{x}^k) - \mathbf{J}_F(\mathbf{x}^k)\mathbf{z}\| \quad (6.2.10)$$

Damit ergibt sich die Vorschrift für die Gauss-Newton Iteration zu:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{s} \quad \text{mit} \quad \mathbf{s} := \operatorname{argmin}_{\mathbf{z} \in \mathbb{R}^n} \|F(\mathbf{x}^{(k)}) - \mathbf{J}_F(\mathbf{x}^{(k)})\mathbf{z}\|_2^2. \quad (6.2.11)$$

Wie auch beim Newton-Verfahren muss dem Gauss-Newton-Verfahren ein guter Startwert

$\mathbf{x}^{(0)} \in \mathbb{R}^n$  übergeben werden.

### Code 6.2.2: Gauss-Newton Verfahren

```

1 from numpy.linalg import lstsq, norm
3 def gn(x,F,J,tol):
4     #x is the starting vector that has to be intelligently chosen
5     #F: function
6     #J: Jacobi-Matrix
7     #tol: tolerance
8     s = lstsq(J(x),F(x))[0]
9     x = x-s
10    #now we perform the iteration
11    while norm(s) > tol*norm(x):
12        #every time we update x by subtracting s, found with the least square
method
13        s = lstsq(J(x),F(x))[0]
14        x = x-s
15    return x

```

Der Code 6.2.2 stellt eine beispielhafte Implementation des Gauss-Newton Verfahrens dar. Dabei wird der Funktion mit  $\mathbf{x}$  der Anfangswert  $\mathbf{x}^{(0)}$  übergeben,  $F$  ist eine Referenz auf die zu minimierende Funktion  $F$ ,  $J$  ist die Jakobi-Matrix von  $F$  und  $\text{tol}$  ist der Grenzwert für den Fehler.

Die zweite Ableitung von  $F(\mathbf{x})$  wird nicht benötigt, aber die Konvergenzordnung ist niedriger als die vom Newton-Verfahren ( $p \leq 2$ ).

**Beispiel 6.2.3.** Wir kehren zurück zum ersten Beispiel. Die Modellfunktion für den Heizvorgang ist  $I_1(t) = a_1 + b_1 e^{-c_1 t}$  und für den Abkühlvorgang  $I_2(t) = a_2 - b_2 e^{-c_2 t}$ . Wir stellen die jeweiligen Funktionen  $F_1, F_2$  nach (6.2.1) auf

$$F_1(\mathbf{a}_1) = \begin{bmatrix} a_1 + b_1 - 24.34 \\ a_1 + b_1 e^{-5c_1} - 18.93 \\ \vdots \\ a_1 + b_1 e^{-25c_1} - 15.91 \end{bmatrix}, \quad F_2(\mathbf{a}_2) = \begin{bmatrix} a_2 - b_2 - 9.66 \\ a_2 - b_2 e^{-5c_2} - 18.8 \\ \vdots \\ a_2 - b_2 e^{-25c_2} - 24.91 \end{bmatrix},$$

wobei  $\mathbf{a}_1 = (a_1, b_1, c_1)$ ,  $\mathbf{a}_2 = (a_2, b_2, c_2)$  sind. Die entsprechenden Jakobi-Matrizen sind dann

$$J_{F_1}(\mathbf{a}_1) = \begin{bmatrix} 1 & 1 & 0 \\ 1 & e^{-5c_1} & -5b_1 e^{-5c_1} \\ \vdots & \vdots & \vdots \\ 1 & e^{-25c_1} & -25b_1 e^{-25c_1} \end{bmatrix}, \quad J_{F_2}(\mathbf{a}_2) = \begin{bmatrix} 1 & 1 & 0 \\ 1 & -e^{-5c_2} & 5b_2 e^{-5c_2} \\ \vdots & \vdots & \vdots \\ 1 & -e^{-25c_2} & 25b_2 e^{-25c_2} \end{bmatrix}.$$

Verwendet man diese Matrizen und Vektoren als Eingabedaten für die Funktion aus Code 6.2.2, mit den Startvektoren  $x_1 = (10, 5, 0)$  für  $F_1$  und  $x_2 = (30, 10, 0)$  für  $F_2$ , so erhält man folgende Parameter:

$$\begin{aligned} a_1 &= 15.8489, b_1 = 8.4822, c_1 = 0.1991 \\ a_2 &= 25.0657, b_2 = 15.3954, c_2 = 0.1779. \end{aligned}$$

Das Ergebnis ist in Abbildung 6.2.5 dargestellt. Die Messwerte werden durch  $I_1, I_2$  mit den entsprechenden Parametern gut approximiert.

Der folgende Code wurde verwendet:

**Code 6.2.4:** Beispiel: Anwendung des Gauss-Newton Algorithmus

```

1 from numpy import arange, array, zeros, exp
3 t = arange(0,30,5)
  curr_heating = array([24.34,18.93,17.09,16.27,15.97,15.91])
5  curr_cooling = array([9.66,18.8,22.36,24.07,24.59,24.91])
7  n =len(t)
9  #define the functions that have to be minimized
  F_1 = lambda a: a[0] + a[1]*exp(-a[2]*t) - curr_heating
11 F_2 = lambda a: a[0] - a[1]*exp(-a[2]*t) - curr_cooling
13 #define the corresponding Jacobi matrices
  def J_1(a):
15     mat = zeros((n,3))
     for k in range(n):
17         mat[k,0] = 1.0
         mat[k,1] = exp(-t[k]*a[2])
19         mat[k,2] = -t[k]*a[1]*exp(-t[k]*a[2])

```

```

    return mat
21
def J_2(a):
23     mat = zeros((n,3))
    for k in range(n):
25         mat[k,0] = 1.0
         mat[k,1] = -exp(-t[k]*a[2])
27         mat[k,2] = t[k]*a[1]*exp(-t[k]*a[2])
    return mat
29
#guess starting vector
31 x_1 = array([10.,5.,0.])
    x_2 = array([30.,10.,0.])
33
#use the Gauss-Newton algorithm declared above
35 a_1 = gn(x_1,F_1,J_1,tol=10e-6)
    a_2 = gn(x_2,F_2,J_2,tol=10e-6)
37 print("Heating ", a_1)
    print("Cooling ", a_2)
39
#plot the data and the fit
41 ts = linspace(0,25,750) #for plotting
    from matplotlib.pyplot import figure, show, plot, xlabel, ylabel, legend, grid
43 figure(); xlabel("Zeit $t$"); ylabel("Strom $I(t)$"); grid(True)
    plot(t, curr_heating, "bv", label="Messdaten")
45 plot(t, curr_cooling, "rv", label="Messdaten")
    plot(ts,a_1[0] + a_1[1]*exp(-a_1[2]*ts), "b", label="$I_1(t)$")
47 plot(ts,a_2[0] - a_2[1]*exp(-a_2[2]*ts), "r", label="$I_2(t)$")
    legend(loc="best")
49 show()

```

In der gestellten Aufgabe ist die Spannung für  $t \rightarrow \infty$  gesucht, welche den konstanten Termen  $a_1, a_2$  entspricht. Mit Hilfe dieser beiden Spannungswerte ist es möglich, einen Näherungswert für die Wärmeleitfähigkeit von Nickel zu bestimmen.

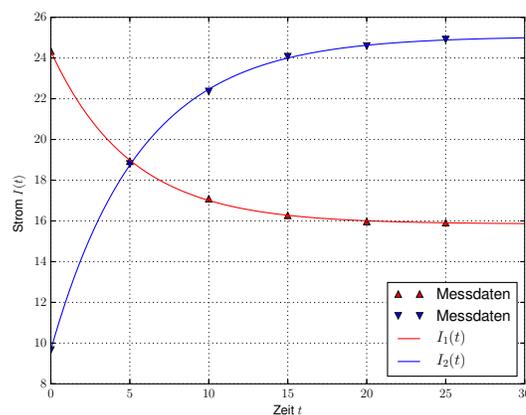


Abb. 6.2.5. Nichtlineare Funktionsanpassung und Messwerte

**Beispiel 6.2.6.** Für dasselbe nichtlineare Ausgleichsproblem wie vorhin:

$$f(t) = x_1 + x_2 \exp(-x_3 t),$$

untersuchen wir in einem numerischen Experiment die Konvergenz beider Verfahren für zwei verschiedene Startwerte. Hier ist der der Simulationscode:

**Code 6.2.7:** Initialisierung des Experiments

```

2 from numpy.linalg import lstsq, norm
3 from scipy.linalg import lu_solve, lu_factor
4 from numpy.random import seed
5 from numpy import arange, array, zeros, exp
6 from matplotlib.pyplot import figure, show, plot, xlabel, ylabel, legend, grid,
7 semilogy, subplots, twinx, title
8
9 random.seed(0) # reproducibility
10 x = array([1.,2.,0.15]); t = arange(1.,7.,0.3)
11 f = lambda x,t: x[0] + x[1]*exp(-x[2]*t)
12 y = f(x,t)
13 y += 0.1*(rand(len(t))-0.5)
14
15 F = lambda x: f(x,t) - y
16
17 DF = lambda x: array([ones(len(t)), exp(-x[2]*t), -t*x[1]*exp(-x[2]*t)].T
18 #JAvobi-Matrix
19
20 def HessPhi(x): #Hesse-Matrix of  $\Phi(x) = 1/2\|F(x)\|_2^2$ 
21     f = F(x)
22     df = DF(x)
23     tmp = t*exp(-x[2]*t)
24     H = array([\
25         [0., 0., 0.],\
26         [0.,0., -dot(f,tmp)],\
27         [0, -dot(f,tmp), x[1]*dot(f,t*tmp)]\
28     ])
29     H += dot(df.T,df)
30     return H
31
32 GradPhi = lambda x: dot( DF(x).T, F(x)) #gradient of  $\Phi(x) = 1/2\|F(x)\|_2^2$ 
33
34 x_0s = [[1.5,1.5,0.1],[1.8,1.8,0.1]]
35
36 #gauss-newton, we adapt the function to return also the corrections
37 def gn(x,F,J,tol):
38     #x is the starting vector that has to be intelligently chosen
39     #F: function
40     #J: Jacobi-Matrix
41     #tol: tolerance
42     s = lstsq(J(x),F(x),rcond=False)[0]
43     S = []; X = [] #container for corrections and temporary x values
44     S += [norm(s)]
45     X += [x]
46     x = x-s

```

```

44     #now we perform the iteration
45     while norm(s) > tol*norm(x):
46         #every time we update x by subtracting s, found with the least square
method
47         s = lstsq(J(x),F(x),rcond=False)[0]
48         S += [norm(s)]
49         X += [x]
50         x = x-s
51         #print(x)
52     return x, X, S

54 #now we plot the magnitude of the corrections
55 fig, ax1 = subplots()
56 ax2 = ax1.twinx()

58 for x_0 in x_0s:
59     x, X, S = gn(x_0, F, DF, 10**-12)
60     #print(S)
61     ax1.semilogy(arange(len(S)), [norm(F(x))**2 for x in X], "-x", label=x_0)
62     ax2.semilogy(arange(len(S)), S, "-x", label=x_0)
63     ax1.set_xlabel("No of step of Gauss-Newton method")
64     ax1.set_ylabel("$||F(x^{(k)})||_2^2$")
65     ax2.set_ylabel("Norm of the correction")
66     legend(loc="best")
67     show()
68

69 #now we finally plot the data together with the function that we obtained via
interpolation
70 ts = linspace(1.,7.,500) #for plot
71 figure()
72 xlabel("$t$"); ylabel("$y$"); title("Gauss-Newton fit")
73 plot(t,y,"ro")
74 plot(ts, f(x,ts), "g")
75 show()
76

77 #newton-method
78
79 def newton(x,GP,HP,tol):
80     #GP: gradient of  $\Phi$ . HP: Hesse Matrix of  $\Phi$ 
81     s = solve(HP(x),GP(x)) # Newton correction
82     S = []; X = [] #container for corrections and temporary x values
83     S += [norm(s)]
84     X += [x]
85     x -= s
86     while norm(s) > tol*norm(x):
87         s = solve(HP(x),GP(x)) # Newton correction
88         x -= s
89         S += [norm(s)]
90         X += [x]
91     return x, X, S
92

93 fig, ax1 = subplots()
94 ax2 = ax1.twinx()

96 for x_0 in x_0s:

```

```

    x, X, S = newton(x_0, GradPhi, HessPhi, 10**-12)
98     #print(S)
    ax1.semilogy(arange(len(S)), [norm(F(x))**2 for x in X], "-x", label=x_0)
100    ax2.semilogy(arange(len(S)), S, "-x", label=x_0)
    ax1.set_xlabel("No of step of Newton method")
102    ax1.set_ylabel("$||F(x^{(k)})||_2^2$")
    ax2.set_ylabel("Norm of the correction")
104    legend(loc="best")
    show()
106
    ts = linspace(1.,7.,500) #for plot
108    figure()
    xlabel("$t$"); ylabel("$y$"); title("Newton fit")
110    plot(t,y,"ro")
    plot(ts, f(x,ts), "g")
112    show()

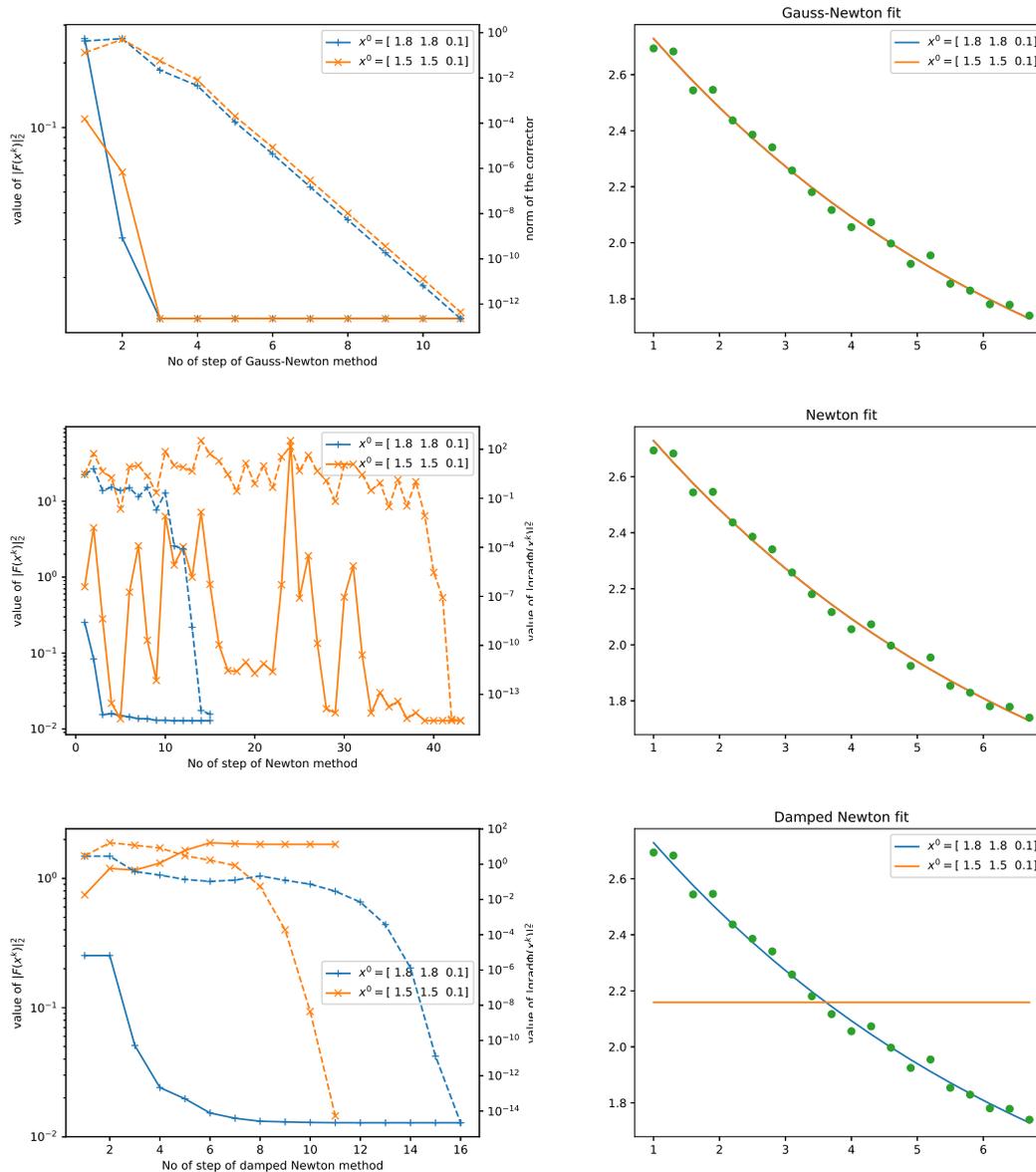
114    #finally, damped newton
    def dampnewton(x,GP,HP,q=0.5,tol=1e-10):
116
        lup = lu_factor(HP(x)) #LU decomposition
118        s = lu_solve(lup,GP(x)) #lu_solve is more efficient
        xn = x-s
120        lam = 1
        st = lu_solve(lup,GP(xn))
122        S = [] #container for corrections
        X = []
124        S += [norm(s)]
        X += [xn]
126        for k in range(15): #otherwise too many steps, d. newt. isn't a good
            technique for this problem...
            #while norm(st) > tol*norm(xn):
128            #(a posteriori termination criteria)
                while norm(st) > (1-lam*0.5)*norm(s): #natural monotonicity test
130                    lam *= 0.5 # reduce damping factor
                    if lam < 1e-10:
132                        print('DAMPED NEWTON: Failure of convergence')
                            return x, X, S
                    xn = x-lam*s #simplified Newton
                            st = lu_solve(lup,GP(xn))
136                x = xn
                    lup = lu_factor(HP(x))
138                    s = lu_solve(lup,GP(x))
                    lam = min(lam/q, 1.) #notice that we must have 1 - lam/2 > 0
140                    xn = x-lam*s #damped newton verfahren
                            S += [lam*norm(s)]
142                    X += [xn]
                            st = lu_solve(lup,GP(xn)) #simplified Newton, prepare st for the next
iteration
144
                x = xn
146                return x, X, S

148    fig, ax1 = subplots()
    ax2 = ax1.twinx()

```

```
150 for x_0 in x_0s:
152     x, X, S = dampnewton(x_0, GradPhi, HessPhi, 0.5, 10**-12)
154     print(cvg)
154     #print(S)
156     ax1.semilogy(arange(len(S)), [norm(F(x))**2 for x in X], "-x", label=x_0)
156     ax2.semilogy(arange(len(S)), S, "-x", label=x_0)
158     ax1.set_xlabel("No of step of Newton method")
158     ax1.set_ylabel("$||F(x^{\{k\}})||_2^2$")
160     ax2.set_ylabel("Norm of the correction")
160     legend(loc="best")
162     show()

162 ts = linspace(1., 7., 500) #for plot
164 figure()
164 xlabel("$t$"); ylabel("$y$"); title("Damped Newton fit (with second starting
164 vector, iteration with first one does not converge)")
166 plot(t, y, "ro")
166 plot(ts, f(x, ts), "g")
168 show()
```



**Abb. 6.2.8.** Für den Anfangswert  $(1.8, 1.8, 0.1)^T$  bleibt die Iteration des gedämpften Newton-Verfahrens in einem lokalen Minimum gefangen, so dass die gelieferte Lösung weit weg von den wahren Parametern liegt; für den Anfangswert  $(1.5, 1.5, 0.1)^T$  scheitert das gedämpfte Newton-Verfahren. Das ungedämpfte Newton-Verfahren konvergiert quadratisch. Das Gauss-Newton-Verfahren konvergiert linear in beiden Fällen.

### 6.2.3 Levenberg-Marquardt-Methode

Wie bereits im Falle des Newton-Verfahrens für nichtlineare Gleichungssysteme gesehen, gibt es auch bei Gauss-Newton-Verfahren für die nichtlineare Ausgleichsrechnung das “over-shooting”-Phänomen, was einer Änderung bedarf. Analog der Technik beim Newton-Verfahren verwendet die Levenberg-Marquardt-Methode eine

heuristische Dämpfung mittels eines Strafterms. Anstatt  $\|F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})\mathbf{s}\|^2$  zu minimieren wird

$$\|F(\mathbf{x}^{(k)}) + DF(\mathbf{x}^{(k)})\mathbf{s}\|^2 + \lambda \|\mathbf{s}\|_2^2$$

minimiert. Der Dämpfungsparameter  $\lambda$  bestraft also grosse Schritte. Die modifizierte Gleichung für die Korrektur lautet:

$$(DF(\mathbf{x}^{(k)})^T DF(\mathbf{x}^{(k)}) + \lambda \mathbf{I}) \mathbf{s} = -DF(\mathbf{x}^{(k)})^T F(\mathbf{x}^{(k)}) . \quad (6.2.12)$$

In python verwendet man die `scipy.optimize.leastsq` Funktion.

# Chapter 7

## Eigenwerte

### 7.1 Motivation und Theorie

**Beispiel 7.1.1.** In der Vorlesung Physik I sah man die Bewegungsgleichung eines Atoms mit der Masse  $m$  im Feld eines anderen Atoms für eine harmonische (d.h. quadratische) Approximation des Potentials:

$$\ddot{x} + \omega_0^2 x = 0, \text{ mit } \omega_0 = \sqrt{\frac{k}{m}} \text{ und } k = D^2U(x^*).$$

Für die Berechnung des Infrarotspektrums eines Moleküls mit  $n$  Freiheitsgraden betrachtet man die Position  $x \in \mathbb{R}^n$  und Geschwindigkeit  $\dot{x} = \frac{dx}{dt} \in \mathbb{R}^n$ . Wir setzen voraus, dass das Molekül nur um ein lokales Minimum des Potentials  $U$  vibriert:

$\mathbf{x}^* \in \mathbb{R}^n$ , so dass  $DU(\mathbf{x}^*) = 0$  und  $\mathbf{H} = D^2U(\mathbf{x}^*)$  symmetrisch positiv definit ist.

Dann können wir  $U$  lokal durch seine (Taylor) quadratische Approximation ersetzen und uns nur um die Abweichung  $\mathbf{a}$  vom Equilibrium  $\mathbf{x}^*$  kümmern:

$$U(\mathbf{x}^* + \mathbf{a}) = U(\mathbf{x}^*) + \frac{1}{2} \mathbf{a}^T \mathbf{H} \mathbf{a}.$$

Wir schreiben die Newtonsche Bewegungsgleichung in der neuen Variable  $\mathbf{a}$  um:

$$m\ddot{\mathbf{x}} = m \frac{d^2}{dt^2}(\mathbf{x}^* + \mathbf{a}) = m\ddot{\mathbf{a}} = -D_{\mathbf{a}}U(\mathbf{x}^* + \mathbf{a}).$$

Da wir nah am Minimum sind, haben wir:

$$m\ddot{\mathbf{a}} = -D_{\mathbf{a}} \left( U(\mathbf{x}^*) + \frac{1}{2} \mathbf{a}^T \mathbf{H} \mathbf{a} \right) = -\mathbf{H} \mathbf{a}.$$

Somit erhalten wir die Bewegungsgleichung für kleine Vibrationen um Equilibrium:

$$\ddot{\mathbf{a}} + \frac{1}{m} \mathbf{H} \mathbf{a} = 0, \text{ mit } \mathbf{H} = D^2U(\mathbf{x}^*).$$

Die Matrix  $\mathbf{H}$  besitzt reelle Einträge und ist symmetrisch. Ihre Eigenvektoren  $\mathbf{w}^1, \mathbf{w}^2, \dots, \mathbf{w}^n$  sind also orthonormal und bilden somit eine zur Darstellung allgemeiner Vektoren geeignete Basis:

$$\mathbf{a} = c_1(t)\mathbf{w}^1 + \dots + c_n(t)\mathbf{w}^n .$$

Wenn wir diese Darstellung in der Bewegungsgleichung verwenden, erhalten wir

$$m(\ddot{c}_1\mathbf{w}^1 + \dots + \ddot{c}_n\mathbf{w}^n) = -(c_1\mathbf{H}\mathbf{w}^1 + \dots + c_n\mathbf{H}\mathbf{w}^n) = -(c_1\lambda_1\mathbf{w}^1 + \dots + c_n\lambda_n\mathbf{w}^n) ,$$

wobei  $\mathbf{H}\mathbf{w}^j = \lambda_j\mathbf{w}^j$ , d.h.  $\lambda_j$  sind die Eigenwerte zu den Eigenvektoren  $\mathbf{w}^j$ . Jetzt bilden wir den Skalarprodukt mit  $\mathbf{w}^k$ . Die Orthogonalität der Vektoren ergibt das entkoppelte System für die Koeffizienten

$$m\ddot{c}_k = -\lambda_k c_k ,$$

das die Frequenzen  $\omega_k = \sqrt{\lambda_k/m}$  aufweist.

Wenn die involvierten Massen verschieden sind, dann enden wir mit dem System für die Abweichung  $\mathbf{a}$ :

$$\mathbf{M}\ddot{\mathbf{a}} = -\mathbf{H}\mathbf{a} ,$$

wobei die Massen-Matrix  $\mathbf{M}$  symmetrisch und positiv-definit ist, aber nicht unbedingt diagonal. Dann müssen wir ein verallgemeinertes Eigenwertproblem lösen:

$$\mathbf{M}^{-1}\mathbf{H}\mathbf{w}^j = \lambda_j\mathbf{w}^j \iff \mathbf{H}\mathbf{w}^j = \lambda_j\mathbf{M}\mathbf{w}^j .$$

**Beispiel 7.1.2.** In der Vorlesung "Physik I" sah man die Gleichung für die erzwungene Schwingung:

$$\ddot{x} + 2\rho\dot{x} + \omega_0^2 x = \frac{F_0}{m} \cos \Omega t ,$$

mit Resonanz für  $\Omega = \omega_0\sqrt{1 - 2\rho^2/\omega_0^2}$ . Die Verallgemeinerung für den mehrdimensionalen Fall lautet:

$$\mathbf{M}\ddot{\mathbf{x}} + 2\mathbf{B}\dot{\mathbf{x}} + \mathbf{C}\mathbf{x} = \mathbf{f} ,$$

mit Massenmatrix  $\mathbf{M}$ , Dämpfungsmatrix  $\mathbf{B}$ , Steifigkeitsmatrix  $\mathbf{C}$  und Kraft  $\mathbf{f}$ . Die Eigenfrequenzen sind durch die Lösung des verallgemeinerten Eigenwertproblems gegeben:

$$\mathbf{C}\mathbf{x} = \omega^2\mathbf{M}\mathbf{x} .$$

**Beispiel 7.1.3.** Eine autonome, homogene, lineare, gewöhnliche Differentialgleichung

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} , \quad \mathbf{A} \in \mathbb{C}^{n,n}$$

mit der diagonalisierbaren Matrix  $\mathbf{A}$  lässt sich mittels Diagonalisierung lösen:

$$\mathbf{A} = \mathbf{S} \underbrace{\begin{bmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{bmatrix}}_{=: \mathbf{D}} \mathbf{S}^{-1}, \quad \mathbf{S} \in \mathbb{C}^{n,n} \text{ regulär.}$$

Der Variablenwechsel  $\mathbf{z} = \mathbf{S}^{-1}\mathbf{y}$  entkoppelt das System:

$$\dot{\mathbf{z}} = \mathbf{D}\mathbf{z} \text{ und } \mathbf{z}(0) = \mathbf{S}^{-1}\mathbf{y}_0.$$

Dann:

$$\mathbf{z}_i(t) = \exp(\lambda_i t)(\mathbf{z}_0)_i = \exp(\lambda_i t) ((\mathbf{S}^{-1})_{i,:} \mathbf{y}_0),$$

und

$$\mathbf{y}(t) = \mathbf{S} \begin{bmatrix} e^{\lambda_1 t} & & \\ & \ddots & \\ & & e^{\lambda_n t} \end{bmatrix} \mathbf{S}^{-1} \mathbf{y}(0). \quad (7.1.1)$$

Diese Art, lineare gewöhnliche Differentialgleichungen zu lösen ist **nicht geeignet** für numerische Berechnungen. Die Methoden für die numerische Lösung linearer gewöhnlichen Differentialgleichungen werden weiter im Abschnitt 2.1.3 beschrieben.

**Beispiel 7.1.4.** Die Vibration einer Saite, die an beiden Enden befestigt ist und unter gleichmässiger Spannung  $T$  steht, wird durch die folgende Differentialgleichung beschrieben:

$$\frac{\partial^2 u(x, t)}{\partial t^2} = \frac{T}{m(x)} \frac{\partial^2 u(x, t)}{\partial x^2},$$

wobei  $m(x)$  die Masse ist. Die Methode der Separation der Variablen liefert

$$\frac{T}{m(x)} \frac{d^2 y(x)}{dx^2} + \omega^2 y(x) = 0,$$

wo  $\omega$  durch die Randbedingungen bestimmbar ist. Für die Vibration einer Saite ist das ein eindimensionales Problem, das analytisch lösbar ist; in mehreren Dimensionen ist aber eine numerische Lösung unumgänglich. Eine Möglichkeit ist, finite Differenzen für die Approximation der Ableitungen anzusetzen. Wir nehmen die äquidistanten Punkte  $x_i = x_1 + ih$  und wir approximieren die Werte der exakten Lösung an diesen Punkten mit  $y_i \approx y(x_i)$ . Die Differentialgleichung in  $x$  wird dann durch ein diskretes Problem approximiert:

$$\frac{T}{m_i} \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2} + \omega^2 y_i = 0, \text{ für } i = 1, 2, \dots, N-1,$$

was zum Eigenwertproblem

$$\mathbf{A}\mathbf{y} = \omega^2 \mathbf{y}$$

führt. Dabei ist  $\mathbf{A}$  eine dünnbesetzte Matrix und  $\mathbf{y} = (y_1, \dots, y_{N-1})^T$ .

Wir betrachten in diesem Kapitel nur quadratische Matrizen. Aus der Vorlesung über Lineare Algebra sollten die folgenden Definitionen und Sätze bekannt sein.

**Definition 7.1.5** (Eigenwerte und Eigenvektoren).

- $\lambda \in \mathbb{C}$  heisst *Eigenwert* von  $\mathbf{A} \in \mathbb{K}^{n,n}$  wenn  $\det(\lambda \mathbf{I} - \mathbf{A}) = 0$ . Das heisst, dass es ein  $\mathbf{v}$  gibt, sodass  $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$ .
- *Spektrum* von  $\mathbf{A} \in \mathbb{K}^{n,n}$  heisst die Menge  $\sigma(\mathbf{A}) := \{\lambda \in \mathbb{C} : \lambda \text{ Eigenwert von } \mathbf{A}\}$ .
- Der *Eigenraum* zum Eigenwert  $\lambda \in \sigma(\mathbf{A})$  ist  $\text{Eig } \mathbf{A}_\lambda := \text{Kern}(\lambda \mathbf{I} - \mathbf{A})$ .
- Jedes Element  $\mathbf{x} \in \text{Eig } \mathbf{A}_\lambda \setminus \{0\}$  heisst *Eigenvektor*.
- Die *Geometrische Vielfachheit* eines Eigenwertes  $\lambda \in \sigma(\mathbf{A})$  ist  $m(\lambda) := \text{Dim } \text{Eig } \mathbf{A}_\lambda$ .

**Bemerkung 7.1.6.** Für  $\lambda \in \sigma(\mathbf{A})$  ist die Dimension des entsprechenden Eigenraums nicht Null. Ausserdem haben die Matrizen  $A$  und  $A^T$  dasselbe Spektrum.

**Definition 7.1.7.** Der *Spektralradius* einer Matrix  $A$  ist

$$\rho(\mathbf{A}) := \max\{|\lambda| : \lambda \in \sigma(\mathbf{A})\}$$

**Theorem 7.1.8.** Für jede von einer Vektornorm induzierte Matrixnorm gilt

$$\rho(\mathbf{A}) \leq \|\mathbf{A}\| .$$

**Theorem 7.1.9.** Das Spektrum einer Matrix  $A$  wird von den Gerschgorin-Kreisen abgedeckt:

$$\sigma(\mathbf{A}) \subset \bigcup_{j=1}^n \{z \in \mathbb{C} : |z - a_{jj}| \leq \sum_{i \neq j} |a_{ji}|\} .$$

**Lemma 7.1.10.** Ähnliche Matrizen haben die gleichen Eigenwerte:

$$\sigma(\mathbf{S}^{-1}\mathbf{A}\mathbf{S}) = \sigma(\mathbf{A}) \text{ für alle reguläre } \mathbf{S} \in \mathbb{K}^{n,n} .$$

**Theorem 7.1.11** (Schur-Zerlegung). Jede Matrix kann mittels unitärer Transformationen auf eine obere Dreiecksmatrix gebracht werden:

$$\forall \mathbf{A} \in \mathbb{K}^{n,n} : \exists \mathbf{U} \in \mathbb{C}^{n,n} \text{ unitär} : \mathbf{U}^H \mathbf{A} \mathbf{U} = \mathbf{T} \text{ mit } \mathbf{T} \in \mathbb{C}^{n,n} \text{ obere Dreiecksmatrix} .$$

**Bemerkung 7.1.12.** Die Diagonaleinträge in der oberen Dreiecksmatrix  $\mathbf{T}$  aus der Schur-Zerlegung sind die Eigenwerte von  $\mathbf{A}$ . Die Spalten der unitären Matrix  $\mathbf{U}$  bilden eine orthonormale Basis von Eigenvektoren von  $\mathbf{A}$ .

**Bemerkung 7.1.13.** Normale Matrizen (d.h. Matrizen  $\mathbf{A}$ , die  $\mathbf{A}\mathbf{A}^H = \mathbf{A}^H\mathbf{A}$  erfüllen) lassen sich via unitäre Transformationen diagonalisieren. Die Hermite-symmetrischen ( $\mathbf{A}^H = \mathbf{A}$ ), die unitären ( $\mathbf{A}^H = \mathbf{A}^{-1}$ ) und die Hermite-schiefsymmetrischen ( $\mathbf{A}^H = -\mathbf{A}$ ) Matrizen sind normale Matrizen.

## Wissenswertes über Eigenwerte

Matrix	Eigenwerte	Eigenvektoren
Symmetrisch $\mathbf{A}^T = \mathbf{A}$	alle $\lambda \in \mathbb{R}$	orthogonal $\mathbf{w}_i^T \mathbf{w}_j = 0, i \neq j$
Orthogonal $\mathbf{Q}^T = \mathbf{Q}^{-1}$	alle $ \lambda  = 1$	orthogonal $\overline{\mathbf{w}}_i^T \mathbf{w}_j = 0, i \neq j$
Schief-symmetrisch $\mathbf{A}^T = -\mathbf{A}$	alle $\lambda \in i\mathbb{R}$	orthogonal $\overline{\mathbf{w}}_i^T \mathbf{w}_j = 0$
Hermite-symmetrisch $\mathbf{A}^H := \overline{\mathbf{A}}^T = \mathbf{A}$	alle $\lambda \in \mathbb{R}$	orthogonal $\overline{\mathbf{w}}_i^T \mathbf{w}_j = 0$
Positiv definit $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$	alle $\lambda > 0$	orthogonal
ähnliche $\mathbf{B} = \mathbf{M}^{-1} \mathbf{A} \mathbf{M}$	$\lambda(\mathbf{B}) = \lambda(\mathbf{A})$	$\mathbf{w}(\mathbf{B}) = \mathbf{M}^{-1} \mathbf{w}(\mathbf{A})$
Projektion $P = P^2 = P^T$	$\lambda = 1; 0$	Bild, Kernel
Spiegelung $\mathbf{I} - 2\mathbf{u}\mathbf{u}^T$	$\lambda = -1; 1, \dots, 1$	$\mathbf{u}; \mathbf{u}^\perp$
Rang 1: $\mathbf{u}\mathbf{v}^T$	$\lambda = \mathbf{v}^T \mathbf{u}; 0, \dots, 0$	$\mathbf{u}; \mathbf{v}^\perp$
Inverse $\mathbf{A}^{-1}$	$\lambda(\mathbf{A})^{-1}$	$\mathbf{w}(\mathbf{A})$
Shift $\mathbf{A} + c\mathbf{I}$	$\lambda(\mathbf{A}) + c$	$\mathbf{w}(\mathbf{A})$
Diagonalisierbar $\mathbf{A} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1}$	$\text{diag}(\mathbf{\Lambda})$	Spalten von $\mathbf{S}$ sind unabhängig
Symmetrisch $\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$	$\text{diag}(\mathbf{\Lambda}), \text{reell}$	Spalten von $\mathbf{Q}$ sind orthonormal

## 7.2 “Direktes” Lösen

Das Eigenwertproblem  $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$  lässt sich als Nullstellen-Problem  $p(\lambda) = 0$  umformen, wobei  $p(\lambda)$  das charakteristische Polynom  $p(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I})$  ist. Auch umgekehrt gilt es: gegeben

$$p(\lambda) = \lambda^n + c_{n-1}\lambda^{n-1} + \dots + c_1\lambda + c_0$$

gibt es die Matrix

$$\begin{bmatrix} 0 & & & -c_0 \\ 1 & 0 & & -c_1 \\ & 1 & \ddots & \vdots \\ & & \ddots & 0 & -c_{n-2} \\ & & & 1 & -c_{n-1} \end{bmatrix},$$

sodass die Nullstellen von  $p$  die Eigenwerte von  $\mathbf{A}$  sind. Abel und Galois zeigten 1824, dass es keine Formel für die direkte Berechnung der Nullstellen von Polynomen

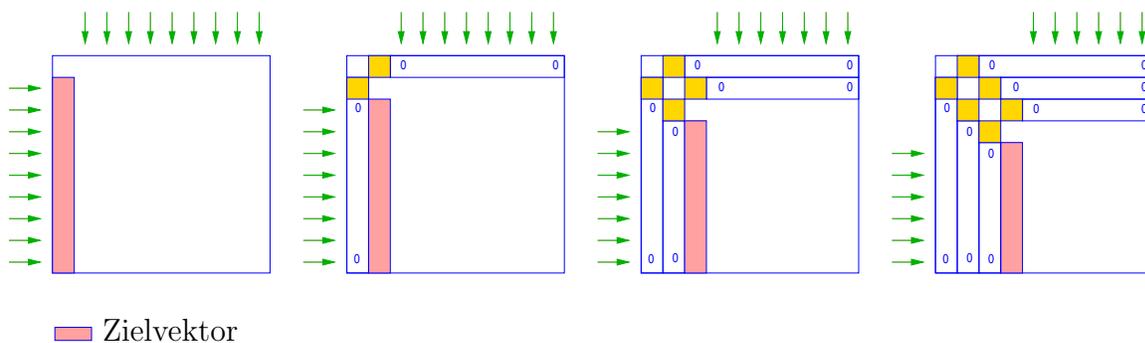
vom Grad grösser als 4 gibt. Dies bedeutet für uns, dass es Matrizen gibt, deren Eigenwerte sich nicht exakt berechnen lassen, auch bei exakter Arithmetik. Mit anderen Worten: **ein Algorithmus für Eigenwertprobleme muss iterativ** sein. Die Algorithmen für Eigenwertprobleme verwenden die Schur-Zerlegung und bestehen aus zwei Phasen: zuerst wird die Matrix  $\mathbf{A}$  durch orthogonale Transformationen auf Hessenberg-Form

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & & \times & \times \end{bmatrix}$$

gebracht. Dann produziert eine zweite Iteration eine Folge von Hessenberg-Matrizen, die zu einer oberen Dreiecksform konvergiert. Der Algorithmus für eine  $n \times n$ -Matrix kostet  $O(n^3)$  Operationen, was für grosse Matrizen sehr teuer ist.

Wenn  $A$  Hermite-symmetrisch ist, dann ist die zweite Phase deutlich günstiger durchzuführen, da die Hessenberg-Matrizen in diesem Fall tridiagonale Matrizen sind.

Die erste Phase wird mittels Householder-Transformationen realisiert, hier schematisch für  $A = A^H$ :



Der  $QR$ -Algorithmus mit Shift ist das Standard-Verfahren; er konvergiert quadratisch und sogar kubisch für normale Matrizen.

Code 7.2.1: QR-algorithm with shift

```

1  ...
2  QR-algorithm with shift
3  ...
4  from numpy import mat, argmin, eye, tril, random
5  from numpy.linalg import norm, eig, qr, eigvals
6  import time
7
8  def eigqr(A,tol):
9      n = A.shape[0]

```

```

10     while (norm(tril(A,-1), ord=2) > tol*norm(A, ord=2)):
11         # shift by ew of lower right 2x2 block closest to (A)n,n
12         sc, dummy = eig(A[n-2:n,n-2:n]) #sc: eigenvalues, dummy: eigenvectors
13         (we are not going to use them)
14         k = argmin(abs(sc - A[n-1,n-1]))
15         shift = sc[k]
16         Q, R = qr( A - shift * eye(n))
17         A = mat(Q).H*mat(A)*mat(Q) #mat() interprets the input as a matrix
18         d = A.diagonal()
19         return d
20
21 if __name__ == "__main__":
22     #we want to run some experiments and study the efficiency of the algorithms
23     #A = mat('1,2,3; 4,5,6; 7,8,9')
24     N = 5
25     #we first generate an N×N matrix with random components
26     A = random.rand(N,N); A = mat(A)
27     ti = time.time()
28     print('numpy.linalg.eigvals:', eigvals(A))
29     tf = time.time()
30     print(tf-ti)
31     ti = time.time()
32     print('with our eigqr::', eigqr(A,10**-6))
33     tf = time.time()
34     print(tf-ti)

```

**Bemerkung 7.2.2.** Professionelle Implementierungen sind schneller, aber trotzdem teuer:

Eigenwerte & Eigenvektoren von $\mathbf{A} \in \mathbb{K}^{n,n}$	$\sim 25n^3 + O(n^2)$
nur Eigenwerte von $\mathbf{A} \in \mathbb{K}^{n,n}$	$\sim 10n^3 + O(n^2)$
Eigenwerte & Eigenvektoren von $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim 6n^3 + O(n^2)$
nur Eigenwerte von $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim O(n^2)$
nur Eigenwerte von <b>tridiagonalem</b> $\mathbf{A} = \mathbf{A}^H \in \mathbb{K}^{n,n}$	$\sim n^2 + O(n)$

Code 7.2.3: Laufzeit von eig

```

1  import numpy as np
2  from scipy.sparse import spdiags
3  import scipy.sparse.linalg as sparsela
4  import time
5  from scipy.sparse.linalg.eigen.arpack import arpack
6  from eigqr import eigqr
7
8  N = 500 #N×N will be the dimension of the matrix
9  A = np.random.rand(N,N); A = np.mat(A) #N×N matrix with random coefficients
10 B = A.H*A #A.H returns the hermite-conjugate of the matrix A
11 z = np.ones(N) #N×N identity matrix

```

```

13 t0 = time.time()
   C = spdiags([3*z,z,z],[0,1,-1],N,N,format='lil') #spdiags() returns a sparse
   matrix from diagonals
15 t1 = time.time()
   print('C constructed in', t1-t0, 'seconds')
17 D = C.tocsr() # tocsr() returns a copy of this matrix in Compressed Sparse Row
   format (necessary for the arpack wrapper)

19 nexp = 4 #number of experiments to be performed in order to get the average
   timing
   ns = np.arange(5,N+1,5) #for every n in ns we will consider matrices of
   dimension n×n
21 times = []
   for n in ns:
23     print('n=', n)
       An = A[:n,:n]; Bn = B[:n,:n]; Dn = D[:n,:n]

25
       #first we extract only the eigenvalues with the standard algorithm
27     ts = np.zeros(nexp)
       for k in range(nexp):
29         ti = time.time()
           w = np.linalg.eigvals(An)
31         tf = time.time()
           ts[k] = tf-ti
33     t1 = ts.sum()/nexp #average execution time
       print('t1 =', t1)

35
       #now we repeat the computation, requiring also the eigenvectors
37     ts = np.zeros(nexp)
       for k in range(nexp):
39         ti = time.time()
           w, V = np.linalg.eig(An)
41         tf = time.time()
           ts[k] = tf-ti
43     t2 = ts.sum()/nexp
       print('t2 =', t2)

45
       #for hermite-symmetric matrices one can use a more efficient algorithm. We
       test it with B
47
       #here we only look for the eigenvalues
49     ts = np.zeros(nexp)
       for k in range(nexp):
51         ti = time.time()
           w = np.linalg.eigvalsh(Bn)
53         tf = time.time()
           ts[k] = tf-ti
55     t3 = ts.sum()/nexp
       print('t3 =', t3)

57
       #here we look for eigenvalues and eigenvectors
59     ts = np.zeros(nexp)
       for k in range(nexp):
61         ti = time.time()
           w, V = np.linalg.eigh(Bn)
63         tf = time.time()

```

```

        ts[k] = tf-ti
65     t4 = ts.sum()/nexp
        print('t4 =', t4)
67
        #arpack wrapper for hermite-symmetric matrices
69     ts = np.zeros(nexp)
        for k in range(nexp):
71         ti = time.time()
            w, V = arpack.eigsh(Dn, k=n-2)
73         tf = time.time()
            ts[k] = tf-ti
75     t6 = ts.sum()/nexp
        print('t6 =', t6)
77
        times += [np.array([t1, t2, t3, t4, t6])]
79
    times = np.array(times)
81
    #finally we want to plot the execution times as a function of the matrix
    dimension for the different methods
83    from matplotlib import pyplot as plt
    plt.loglog(ns, times[:,0], 'r+')
85    plt.loglog(ns, times[:,1], 'm*')
    plt.loglog(ns, times[:,2], 'cp')
87    plt.loglog(ns, times[:,3], 'b^')
    plt.loglog(ns, times[:,4])
89    plt.xlabel('matrix size n')
    plt.ylabel('time [s]')
91    plt.title('eig runtimes')
    plt.legend(('w = eig(A)', 'w, V = eig(A)', 'w = eig(B)', 'w, V = eig(B)', 'w =
    arpack.eigsh(C)'), loc='upper left')
93    plt.savefig('eigtimingall.eps')
    plt.show()
95
    plt.clf() #clf clears the current figure (we could also produce a new figure
    with figure())
97    plt.loglog(ns, times[:,0], 'r+')
    plt.loglog(ns, times[:,1], 'm*')
99    plt.loglog(ns, ns**3./(ns[0]**3.)*times[0,1], 'k-')
    plt.xlabel('matrix size n')
    plt.ylabel('time [s]')
101    plt.title('nXn random matrix')
    plt.legend(('w = eig(A)', 'w, V = eig(A)', 'O(n^3)'), loc='upper left')
103    plt.savefig('eigtimingA.eps')
    plt.show()
105
    plt.clf()
107    plt.loglog(ns, times[:,2], 'cp')
    plt.loglog(ns, times[:,3], 'b^')
109    plt.loglog(ns, ns**3./(ns[0]**3.)*times[0,2], 'k-')
    plt.xlabel('matrix size n')
    plt.ylabel('time [s]')
111    plt.title('nXn random Hermitian matrix')
    plt.legend(('w = eig(B)', 'w, V = eig(B)', 'O(n^3)'), loc='upper left')
113    plt.savefig('eigtimingB.eps')
115

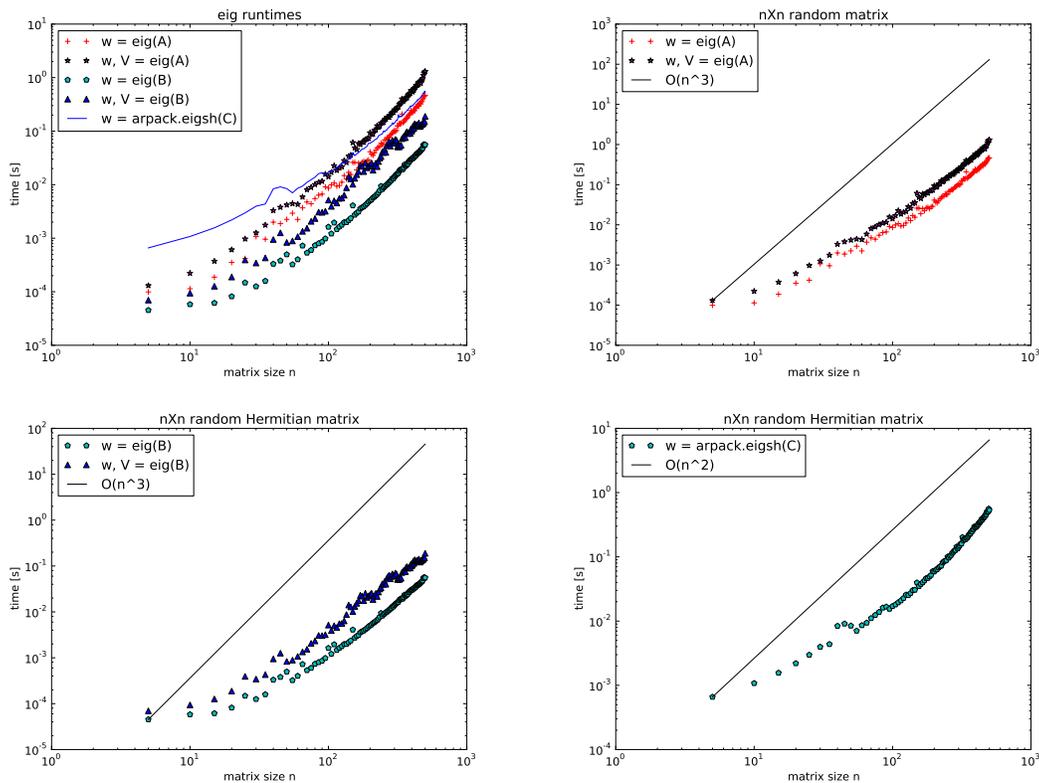
```

```

plt.show()
117
plt.clf()
119 plt.loglog(ns, times[:,4], 'cp')
plt.loglog(ns, ns**2./(ns[0]**2.)*times[0,4], 'k-')
121 plt.xlabel('matrix size n')
plt.ylabel('time [s]')
123 plt.title('nXn random Hermitian matrix')
plt.legend(('w = arpack.eigsh(C)', 'O(n^2)'), loc='upper left')
125 plt.savefig('eigtimingC.eps')
plt.show()

```

**Beispiel 7.2.4.** Hier sind die Ergebnisse, die der Code 7.2.3 vor wenigen Jahren auf einem Laptop lieferte:



**Abb. 7.2.5.** Laufzeit verschiedener Eigensolver für verschiedene Matrizen.

Effizientere Methoden werden wir für dünnbesetzte Matrizen im Folgenden besprechen. Sparse eigensolvers: ARPACK

scipy 0.17: `scipy.sparse.linalg.eigs`

## 7.3 Potenzmethoden

**Definition 7.3.1.** Der *Rayleigh-Quotient* von  $\mathbf{x}$  ist definiert als

$$\rho_{\mathbf{A}}(\mathbf{x}) = \frac{\mathbf{x}^H \mathbf{A} \mathbf{x}}{\mathbf{x}^H \mathbf{x}}.$$

**Bemerkung 7.3.2.** Wenn  $\mathbf{x}$  ein Eigenvektor ist, dann ist  $\mathbf{A} \mathbf{x} = \lambda \mathbf{x}$  und  $\rho_{\mathbf{A}}(\mathbf{x}) = \lambda$ . Wenn  $\mathbf{x}$  allgemein ist, dann ist

$$\rho_{\mathbf{A}}(\mathbf{x}) = \arg \min_{\alpha} \|\mathbf{A} \mathbf{x} - \alpha \mathbf{x}\|_2.$$

**Bemerkung 7.3.3.** Für  $\mathbf{A} \in \mathbb{R}^{n \times n}$ ,  $\mathbf{A}^T = \mathbf{A}$  und  $\lambda_1, \dots, \lambda_n \in \mathbb{R}$ ,  $\mathbf{q}_1, \dots, \mathbf{q}_n \in \mathbb{R}^n$  sind orthonormale Eigenvektoren. Wir berechnen die Ableitung von  $\rho_{\mathbf{A}} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ :

$$\frac{\partial \rho_{\mathbf{A}}}{\partial x_j} = \frac{2}{\mathbf{x}^T \mathbf{x}} (\mathbf{A} \mathbf{x} - \rho_{\mathbf{A}}(\mathbf{x}) \mathbf{x})_j,$$

sodass  $\nabla \rho_{\mathbf{A}}(\mathbf{x}) = \frac{2}{\|\mathbf{x}\|^2} (\mathbf{A} \mathbf{x} - \rho_{\mathbf{A}}(\mathbf{x}) \mathbf{x})$ .

Somit sind die Eigenwerte von  $\mathbf{A}$  die Stationärpunkte von  $\rho_{\mathbf{A}}$ . Der Satz von Taylor für  $\rho_{\mathbf{A}}$  gibt dann

$$\rho_{\mathbf{A}}(\mathbf{x}) - \rho_{\mathbf{A}}(\mathbf{q}_j) = O(\|\mathbf{x} - \mathbf{q}_j\|^2) \text{ für } \mathbf{x} \rightarrow \mathbf{q}_j,$$

was bedeutet, dass der Rayleigh-Quotient quadratisch akkurat ist.

### 7.3.1 Direkte Potenzmethode

**Aufgabe:**

Gegeben  $\mathbf{A}$ , finde den (im Betrag) grössten Eigenwert von  $\mathbf{A}$  und einen dazugehörigen Eigenvektor.

Sei  $\mathbf{A} \in \mathbb{R}^{n \times n}$  diagonalisierbar:  $\mathbf{S}^{-1} \mathbf{A} \mathbf{S} = \text{diag}(\lambda_1, \dots, \lambda_n)$  und wir nehmen an, dass

$$|\lambda_1| \leq |\lambda_2| \leq \dots \leq |\lambda_{n-1}| < |\lambda_n|$$

und die Eigenvektoren sind die Spalten von  $\mathbf{S}$ :

$$\|\mathbf{S}_j\|_2 = 1 \text{ für } j = 1, 2, \dots, n.$$

Sei

$$\mathbf{z} = \sum_{j=1}^n \zeta_j \mathbf{S}_j \text{ mit } \zeta_n \neq 0.$$

Dann:

$$\mathbf{A} \mathbf{z} = \sum_{j=1}^n \zeta_j \mathbf{A} \mathbf{S}_j = \sum_{j=1}^n \zeta_j \lambda_j \mathbf{S}_j$$

und

$$\mathbf{A}^2 \mathbf{z} = \sum_j \zeta_j \lambda_j \mathbf{A} \mathbf{S}_j = \sum_j \zeta_j \lambda_j^2 \mathbf{S}_j.$$

Im allgemeinen folgt:

$$\mathbf{A}^k \mathbf{z} = \sum_{j=1}^n \zeta_j \lambda_j^k \mathbf{S}_j = \lambda_n^k \left( \sum_{j=1}^{n-1} \zeta_j \left( \frac{\lambda_j}{\lambda_n} \right)^k \mathbf{S}_j + \zeta_n \mathbf{S}_n \right).$$

Da  $|\lambda_j| < |\lambda_n|$ , gilt  $\left( \frac{\lambda_j}{\lambda_n} \right)^k \rightarrow 0$  für  $k \rightarrow \infty$  und somit

$$\frac{\mathbf{A}^k \mathbf{z}}{\|\mathbf{A}^k \mathbf{z}\|} \rightarrow \pm \mathbf{S}_n \text{ für } k \rightarrow \infty.$$

Das ergibt den folgenden **Algorithmus**: Potenzmethode:

$$\begin{aligned} & \mathbf{z}^{(0)} \text{ zufällig, } \|\mathbf{z}^{(0)}\| = 1 \\ & \text{für } 1, 2, \dots \\ & \quad \mathbf{w} = \mathbf{A} \mathbf{z}^{(k-1)} \\ & \quad \lambda^{(k-1)} = \mathbf{w}^H \mathbf{z}^{(k-1)} \\ & \quad \mathbf{z}^{(k)} = \mathbf{w} / \|\mathbf{w}\|. \end{aligned}$$

**Beispiel 7.3.4.** (Direkte Potenzmethode)

Wir bauen uns verschiedene Matrizen  $\mathbf{A}$ , deren Eigenwerte und Eigenvektoren wir kennen (sie sind die Einträge des Vektors  $\mathbf{d}$ ):

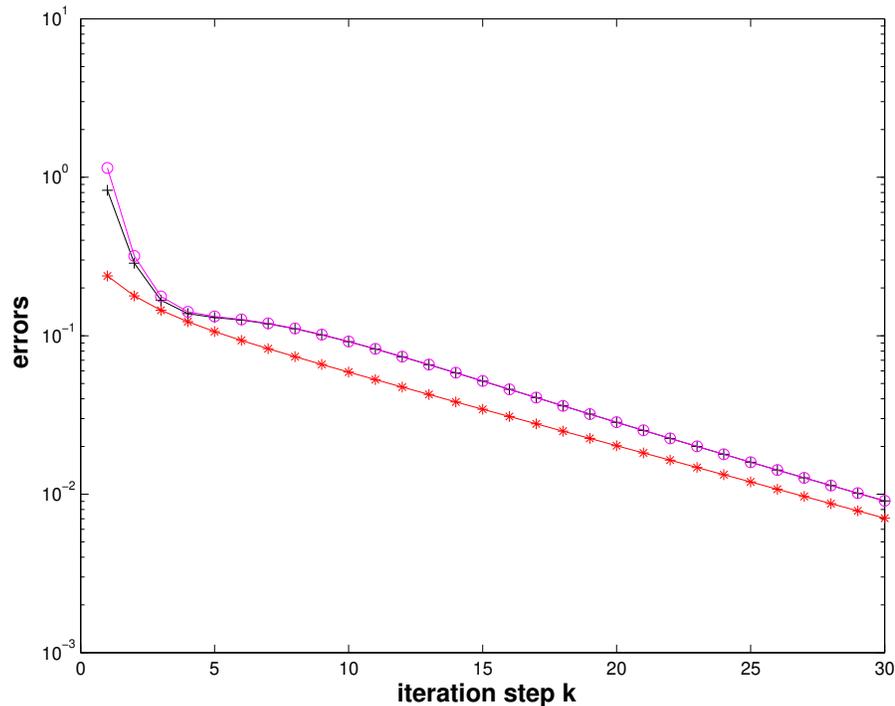
```

1 n = len(d) # size of the matrix
2 S = triu(diag(r_[n:0:-1]) + ones((n,n)))
3 A = dot(S, dot(diag(d), inv(S)) )

```

Wir erhalten damit Testmatrizen für verschiedene Vektoren  $\mathbf{d}$ :

$$\begin{aligned} \mathbf{d} &= 1. + \mathbf{r}_-[0:n] \quad \text{hat } |\lambda_{n-1}| : |\lambda_n| = 0.9 \\ \mathbf{d} &= \text{ones}(n); \mathbf{d}[-1] = 2. \quad \text{hat } |\lambda_{n-1}| : |\lambda_n| = 0.5 \\ \mathbf{d} &= 1. - 0.5 * \mathbf{r}_-[1:5.1:0.5] \quad \text{hat } |\lambda_{n-1}| : |\lambda_n| = 0.9866 \end{aligned}$$



**Abb. 7.3.5.** Fehler in der Eigenwertberechnung mittels:

- o = Rayleigh-Quotienten  $|\lambda_n - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$ ,
- \* = Eigenvektor  $\|\mathbf{z}^{(k)} - \mathbf{s}_{\cdot,n}\|$ ,
- + = Potenzformel  $\left| \lambda_n - \frac{\|\mathbf{A}\mathbf{z}^{(k-1)}\|_2}{\|\mathbf{z}^{(k-1)}\|_2} \right|$ .

Diesen Graphen (und mehr) können wir mit dem folgenden Code erhalten:

**Code 7.3.6:** Power Iteration für die Berechnung von  $\lambda_{\max}(\mathbf{A})$  und entsprechendem Eigenvektor

```

1  from numpy import r_, triu, diag, ones, dot, arange, shape
2  from numpy.linalg import inv, norm
3  from numpy.random import rand, seed
4  from matplotlib.pyplot import figure, show, semilogy, xlabel, ylabel, title,
5  legend
6
7  #the algorithm from the theory, adapted in order to get the results of each
8  iteration:
9  #exercise: complete the algorithm to include a tolerance. Hint: if norm(xnew -
10 xold) < tol ...
11 #you should update xnew and xold in every iteration, how?
12 #hint: in case you want to use this algorithm in your code, modify it and return
13 only the data you need
14 def power_method(A, x, maxit):
15     #arrays for storing the approximation of the maximal eigenvector at every
16     iteration
17     EW_k = []; EWrayleigh_k = []
18     x_k = [] #container for x at every iteration
19     k = 1; x = x/norm(x)
20     while(k <= maxit):

```

```

17     w = dot(A,x)
18     EWrayleigh_k += [dot(x,w)]
19     x = w/norm(w)
20     k = k+1
21     EW_k += [norm(w)]
22     x_k += [x]
23     EV = x #eigenvector
24     EW = EW_k[-1] #eigenvalue
25     return EV, EW, EW_k, EWrayleigh_k, x_k
26
27 n = 9 # size of the matrix
28 n_iter = 30 #number of iterations
29
30 #define possible sets of eigenvalues
31 d_1 = 1.+r_[0:n]
32 d_2 = ones(n); d_2[-1] = 2
33 d_3 = 1.-0.5**((0.5*r_[2:(n+2)])) #1,1.5,2,2.5,
34 D = [d_1, d_2, d_3]
35
36 S = triu(diag(r_[n:0:-1]) + ones((n,n)))
37
38 seed(0)
39 x_0 = rand(n) #choose a random starting vector
40
41 #now we plot the errors
42 figure()
43 title("Error on eigenvalues")
44 xlabel("Iteration step k"); ylabel("errors")
45 for d in D:
46     A = dot(S, dot(diag(d), inv(S)) )
47     errors_eval = abs(power_method(A, x_0, n_iter)[2]-max(abs(d))) #the errors
48     semilogy(arange(n_iter), errors_eval,"-+")
49     show()
50
51 figure()
52 title("Error on eigenvalues with Rayleigh quotients")
53 xlabel("Iteration step k"); ylabel("errors")
54 for d in D:
55     A = dot(S, dot(diag(d), inv(S)) )
56     errors_rayl = abs(power_method(A, x_0, n_iter)[3]-max(abs(d))) #the errors
57     semilogy(arange(n_iter), errors_rayl,"-o")
58     show()
59
60 figure()
61 title("Error on eigenvectors")
62 xlabel("Iteration step k"); ylabel("errors")
63 for d in D:
64     A = dot(S, dot(diag(d), inv(S)) )
65     #S[:, -1]/norm(S[:, -1]) is the (normed) eigenvector corresponding to the
66     biggest eigenvalue
67     errors_ev = [norm(power_method(A, x_0, n_iter)[4][k]-S[:, -1]/norm(S[:, -1]))
68     for k in range(n_iter)] #the
69     errors
70     semilogy(arange(n_iter), errors_ev,"-*")
71     show()

```

```

69 #finally, we compare the errors for d1
    figure()
71 title("d_1 - comparison of error estimations")
    xlabel("Iteration step k"); ylabel("errors")
73 A = dot(S, dot(diag(d_1), inv(S)) )
    errors_eval = abs(power_method(A, x_0, n_iter)[2]-max(abs(d_1)))
75 errors_rayl = abs(power_method(A, x_0, n_iter)[3]-max(abs(d_1)))
    errors_ev = [norm(power_method(A, x_0, n_iter)[4][k]-S[:, -1])/norm(S[:, -1])] for
    k in range(n_iter)] #the
    errors
77 semilogy(arange(n_iter), errors_eval, "--+", label = "Eigenvalues")
    semilogy(arange(n_iter), errors_rayl, "-o", label = "Rayleigh")
79 semilogy(arange(n_iter), errors_ev, "--*", label = "Eigenvectors")
    legend(loc="best")
81 show()

```

Ausserdem (Übung!) können wir die Konvergenzraten mittels

$$\rho_{\text{EV}}^{(k)} := \frac{\|\mathbf{z}^{(k)} - \mathbf{s}_{\cdot, n}\|}{\|\mathbf{z}^{(k-1)} - \mathbf{s}_{\cdot, n}\|},$$

$$\rho_{\text{EW}}^{(k)} := \frac{|\rho_{\mathbf{A}}(\mathbf{z}^{(k)}) - \lambda_n|}{|\rho_{\mathbf{A}}(\mathbf{z}^{(k-1)}) - \lambda_n|}.$$

schätzen. Wir erhalten folgende Ergebnisse:

$k$	$\rho_{\text{EV}}^{(k)}$	$\rho_{\text{EW}}^{(k)}$	$\rho_{\text{EV}}^{(k)}$	$\rho_{\text{EW}}^{(k)}$	$\rho_{\text{EV}}^{(k)}$	$\rho_{\text{EW}}^{(k)}$
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

Wir beobachten numerische lineare Konvergenz mit Rate  $\frac{|\lambda_{n-1}|}{|\lambda_n|}$ .

**Bemerkung 7.3.7.** (Startvektor)

Dieser Algorithmus fängt mit einem Zufallsvektor an; theoretisch ist es also möglich, ein  $\zeta_n = 0$  zu bekommen, was aber in praktischen Anwendungen meistens nicht der Fall ist.

## 7.3.2 Inverse Iteration

**Aufgabe:**

Gegeben  $\mathbf{A}$  regulär, finde den (im Betrag) kleinsten Eigenwert von  $\mathbf{A}$  und einen dazugehörigen Eigenvektor.

**Bemerkung 7.3.8.** Sei  $\mathbf{A}$  regulär, dann gilt: der betragskleinste Eigenwert von  $\mathbf{A}$  ist das Inverse vom betragsgrössten Eigenwert von  $\mathbf{A}^{-1}$ .

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \iff \mathbf{A}^{-1}\mathbf{x} = \frac{1}{\lambda}\mathbf{x}.$$

**Code 7.3.9:** Inverse Iteration für die Berechnung von  $\lambda_{\min}(\mathbf{A})$  und entsprechendem Eigenvektor

```

import numpy as np
import scipy.linalg as splalg

def invit(A,tol):
    LUP = splalg.lu_factor(A) #pivoted LU decomposition of A
    n = A.shape[0] #dimension of the vector space
    x = np.random.rand(n) #random starting vector
    x /= np.linalg.norm(x) #normalize x
    y = splalg.lu_solve(LUP, x)
    a = y @ x
    lmin = 1/a
    x = y/norm(y)
    while abs(lmin-lold) > tol*abs(lmin0):
        lold = lmin
        y = splalg.lu_solve(LUP, x)
        a = y @ x
        lmin = 1/a
        x = y/norm(y)
    return lmin

```

**Bemerkung 7.3.10.** Wir berechnen niemals  $\mathbf{A}^{-1}$ , sondern wir lösen lineare Gleichungssysteme mittels LU-Zerlegungen! Hier wird die LU-Zerlegung nur ein einziges Mal durchgeführt und in jedem Iterationsschritt wieder verwendet.

**Aufgabe:** Gegeben  $\alpha \in \mathbb{C}$ , finde den Eigenwert von  $\mathbf{A}$ , der  $\alpha$  am nächsten liegt:

$$|\alpha - \lambda| = \min\{|\alpha - \mu|; \mu \in \sigma(\mathbf{A})\}.$$

**Bemerkung 7.3.11.** Die Eigenvektoren von  $(\mathbf{A} - \alpha\mathbf{I})^{-1}$  sind dieselben von  $\mathbf{A}$  und die Eigenwerte sind  $(\lambda_f - \alpha)^{-1}$ :

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \iff (\mathbf{A} - \alpha\mathbf{I})\mathbf{x} = (\lambda - \alpha)\mathbf{x}.$$

**Algorithmus:** *shifted inverse Iteration*

$$z^{(0)} \text{ zufällig, } \|z^{(0)}\| = 1$$

für  $1, 2, \dots$

$$w = (\mathbf{A} - \alpha\mathbf{I})^{-1}z^{(k-1)}$$

$$z^{(k)} = w/\|w\|_2.$$

Dabei wird nur ein einziges Mal die LU-Zerlegung von  $(\mathbf{A} - \alpha\mathbf{I})$  durchgeführt. Im letzten Schritt wird der Vektor normalisiert.

**Bemerkung 7.3.12.** Der Algorithmus ist sehr schnell für  $\alpha \approx \lambda_j$ . Daher können wir einen adaptiven Shift verwenden, z.B.  $\alpha := \rho_A(Z^{(k-1)})$  im  $k$ -ten Schritt um die Konvergenz gegen  $\lambda_1$  zu beschleunigen. Wegen (7.3.2) konvergiert die Iteration immer noch gegen den selben Eigenvektor, wird aber in jedem Schritt schneller.

Code 7.3.13: Rayleigh-Quotient-Iteration für  $\lambda_{\min}(\mathbf{A})$  und Eigenvektor

```

1  import numpy as np
2  from matplotlib.pyplot import figure, show, semilogy, xlabel, ylabel, title,
3  legend
4
5  def rqi(A,maxit):
6      # For calculating the errors for every step, the "correct" eigenvalues are
7      # calculated here with eig
8      w, V = np.linalg.eig(A)
9      t = np.where(w == abs(w).min())
10     k = t[0];
11     if len(k) > 1:
12         print('Error: no single smallest EV')
13         raise ValueError
14     ev = V[:,k[0]]; ev /= np.linalg.norm(ev) #eigenvector to max eigenvalue.
15     normalized.
16     ew = w[k[0]] #max. eigenvalue found with eig
17     #
18     n = A.shape[0]
19     alpha = 0.
20     z = np.random.rand(n); z /= np.linalg.norm(z) #random starting vector.
21     normalized.
22     ea_k = []; eb_k = [] #containers for the errors
23     for k in range(maxit):
24         z = np.linalg.solve(A-alpha*np.eye(n), z) #we solve a linear system
25         instead of computing A-1
26         z /= np.linalg.norm(z) #normalization
27         alpha = np.dot(np.dot(A,z),z) #rayleih weight for acceleration of algo
28         ea = abs(alpha-ew)
29         eb = np.minimum(np.linalg.norm(z-ev), np.linalg.norm(z+ev))
30         ea_k += [ea]; eb_k += [eb]
31         print('ea, eb =', ea, eb)
32     return ea_k, eb_k
33
34 #wrapper for running our function in simulation
35 def runit(d, tol, func):
36     n = len(d) # size of the matrix
37     Z = np.diag(np.sqrt(np.r_[n:0:-1])) + np.ones((n,n))
38     Q,R = np.linalg.qr(Z) #QR decomposition
39     A = np.dot(Q, np.dot(np.diag(d), np.linalg.inv(Q)) )
40     res = func(A,tol) #in this case res[0] = eak and res[1] = ebk
41     ea_k = res[0]; eb_k = res[1]
42     figure(); xlabel("k"); ylabel("error")
43     semilogy(np.arange(len(ea_k)),ea_k,"-*",label="$error - eigenvalue")
44     semilogy(np.arange(len(eb_k)),eb_k,"-o",label="$error - eigenvector")
45     legend(loc="best"); show()
46
47 # -----

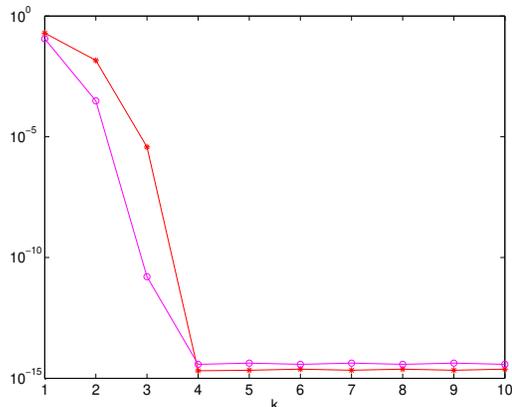
```

```

43 if __name__=='__main__':
    n = 10 # size of the matrix
45   d = 1.+np.r_[0:n] # eigenvalues
    print('d =', d, 'd[-2]/d[-1] =', d[-2]/d[-1])
47   runit(d, 10, rqi)

```

### Beispiel 7.3.14. (Rayleigh-Quotient-Iteration)



$k$	$ \lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)}) $	$\ \mathbf{z}^{(k)} - \mathbf{x}_j\ $
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

Abb. 7.3.15.  $\circ$  :  $|\lambda_{\min} - \rho_{\mathbf{A}}(\mathbf{z}^{(k)})|$   
 $*$  :  $\|\mathbf{z}^{(k)} - \mathbf{x}_j\|$ ,  $\lambda_{\min} = \lambda_j$ ,  $\mathbf{x}_j \in \text{Eig } \mathbf{A}\lambda_j$

**Theorem 7.3.16.** Wenn  $\mathbf{A} = \mathbf{A}^H$ , dann konvergiert  $\rho_{\mathbf{A}}(\mathbf{z}^{(k)})$  lokal mit Ordnung 3 gegen einen Eigenwert, wenn  $\mathbf{z}^{(k)}$  von der Rayleigh-Quotient-Iteration generiert wird.

**Bemerkung 7.3.17.** Der Startvektor hat einen sehr grossen Einfluss auf die inverse Iteration mit Shift: ein Shift zu einem Eigenwert hin zwingt die Iteration, diesen Eigenwert zu reproduzieren. Man kann auch sagen, die Iteration bleibt gefangen in dem Unterraum, der dem nächstliegenden Eigenwert entspricht.

## 7.3.3 Vorkonditionierte inverse Iteration

### Aufgabe:

Gegeben  $\mathbf{A}$  regulär suchen wir den betragskleinsten Eigenwert und einen Eigenvektor dazu. Hierzu kommt aber die Bedingung, dass  $\mathbf{A}\mathbf{x} = \mathbf{b}$  nicht möglich zu lösen ist, zum Beispiel weil  $\mathbf{A}$  sehr gross und dünnbesetzt ist und die LU-Faktoren von  $\mathbf{A}$  nicht dünnbesetzt sind. Die Matrix  $\mathbf{A}$  kann eventuell auch nur als Matrix mal Vektor-Prozedur vorhanden sein.

Die Idee ist,  $\mathbf{A}^{-1}$  durch eine günstige Matrix  $\mathbf{B}^{-1} \approx \mathbf{A}^{-1}$  zu ersetzen. Wir definieren das *Residuum* für das Eigenwertproblem:

$$r(\mathbf{z}) = \mathbf{Ax} - \rho_A(\mathbf{z}) \mathbf{z}.$$

**Algorithmus:** Vorkonditionierte inverse Iteration:

$$\begin{aligned} & \mathbf{z}^{(0)} \text{ zufällig, } \|\mathbf{z}^{(0)}\| = 1 \\ & \text{für } 1, 2, \dots \\ & \quad \mathbf{w} := \mathbf{z}^{(k-1)} - \mathbf{B}^{-1}(\mathbf{Az}^{(k-1)} - \rho_A(\mathbf{z}^{(k-1)})\mathbf{z}^{(k-1)}) \\ & \quad \mathbf{z}^{(k)} = \mathbf{w} / \|\mathbf{w}\|_1. \end{aligned}$$

Code 7.3.18: Vorkonditionierte inverse Iteration

```

1  from numpy import dot, ones, tile, array, r_
2  from numpy.linalg import norm
3  from scipy.sparse import spdiags
4  from scipy.sparse.linalg.eigen.arpack import arpack
5  from scipy.sparse.linalg import spsolve

7  def pinvit(evA, invB, tol, maxit):
8      n = A.shape[0]
9      z = ones(n)/n
10     res = []; rho = 0
11     for k in range(maxit):
12         v = evA(z)
13         rhon = dot(v,z) # Rayleigh quotient
14         r = v - rhon*z # residual
15         z = z - invB(r) # iteration (??)
16         z /= norm(z) # normalization
17         res += [rhon] # tracking iteration
18         if abs(rho-rhon) < tol*abs(rhon): break
19         else: rho = rhon
20     lmin = dot((evA(z)),z)
21     res += [lmin]
22     return lmin, z, res

23 def getdiag(A,k):
24     # A in diagonal matrix format
25     ind = abs(k+A.offsets).argmin()
26     return A.data[ind,:]

29 from numpy import dot, ones, tile, array, linspace
30 from numpy.linalg import norm
31 from scipy.sparse import spdiags
32 from scipy.sparse.linalg.eigen.arpack import arpack
33 from scipy.sparse.linalg import spsolve

35 def pinvit(evA, invB, tol, maxit):
36     n = A.shape[0] # n×n is the dimension of the matrix

```

```

37     # z = ones(n)/n
38     z = linspace(1,n,n)
39     z = z/norm(z)

41     res = []; rho = 0
42     for k in range(maxit):
43         v = evA(z)
44         rhon = dot(v,z) # Rayleigh quotient
45         r = v - rhon*z # residual
46         z = z - invB(r) # iteration
47         z /= norm(z) # normalization
48         res += [rhon] # tracking iteration
49         if abs(rho-rhon) < tol*abs(rhon): break
50         else: rho = rhon
51     lmin = dot((evA(z)),z)
52     res += [lmin] #store the result
53     return lmin, z, res

55 def getdiag(A,k):
56     # A in diagonal matrix format
57     ind = abs(k+A.offsets).argmin()
58     return A.data[ind,:]

59 # -----
60 if __name__=='__main__':
61     k = 1; tol = 1e-13; maxit = 50
62     errC = []; errB = []
63     import matplotlib.pyplot as plt
64     for n in [20, 50, 100, 200]: #we test the algorithm with 4 different values
65 of n
66         a = tile([1./n, -1., 2*(1+1./n), -1., 1./n], (n,1))
67         #numpy.tile(A, reps) constructs an array by repeating A the number of
68 times given by reps (see docs.scipy.org)
69         A = spdiags(a.T, [-n/2,-1,0,1,n/2],n,n)
70         C = A.tocsr() # more efficient sparse format
71         lam = min(abs(arpack.eigsh(C,which='SM')[0])) #minimal eigenvalue found
72 with arpack
73         evC = lambda x: C*x
74         # inverse iteration
75         invB = lambda x: spsolve(C,x) #we solve a linear system instead of
76 computing  $B^{-1}$ 
77         lmin, z, rn = pinvit(evC, invB, tol, maxit)
78         err = abs(rn - lam)/lam
79         errC += [err]
80         txt = 'n='+str(n)
81         plt.semilogy(err, '+', label='$A^{-1}$', '+ txt)
82         print(lmin, lam, err)
83         # preconditioned inverse iteration
84         b = array([getdiag(A,-1), A.diagonal(), getdiag(A,1)])
85         B = spdiags(b, [-1,0,1], n,n)
86         B = B.tocsr()
87         invB = lambda x: spsolve(B,x)
88         lmin, z, rn = pinvit(evC, invB, tol, maxit)
89         err = abs(rn - lam)/lam
90         errB += [err]

```

```

89     plt.semilogy(err,'o', label='$B^{-1}$', '+ txt)
90     print(lmin, lam, err)
91
92     plt.xlabel('# iteration step')
93     plt.ylabel('error in approximation for $\lambda_{\min}$')
94     plt.legend()
95     plt.savefig('pinvitD.eps')
96     plt.show()
97 # -----
98 if __name__=='__main__':
99     k = 1; tol = 10**-4; maxit = 50
100    itnumC = []; itnumB = []
101    import matplotlib.pyplot as plt
102    import time
103    vn = 2**r_[4:11]
104    tCs = []; tBs = []
105    for n in vn:
106        a = tile([1./n, -1., 2*(1+1./n), -1., 1./n], (n,1))
107        A = spdiags(a.T, [-n/2,-1,0,1,n/2],n,n)
108        C = A.tocsr() # more efficient sparse format
109        lam = min(abs(arpack.eigsh(C,n-1)[0]))
110        evC = lambda x: C*x
111        # inverse iteration
112        invB = lambda x: spsolve(C,x)
113        t1 = time.time()
114        lmin, z, rn = pinvit(evC, invB, tol, maxit)
115        t2 = time.time() - t1
116        tCs += [t2]
117        itnumC += [len(rn)-1]
118        print(n, lmin, lam, len(rn)-1, t2)
119        # preconditioned inverse iteration
120        b = array([getdiag(A,-1), A.diagonal(), getdiag(A,1)])
121        B = spdiags(b, [-1,0,1], n,n)
122        B = B.tocsr()
123        invB = lambda x: spsolve(B,x)
124        t1 = time.time()
125        lmin, z, rn = pinvit(evC, invB, tol, maxit)
126        t2 = time.time() -t1
127        tBs += [t2]
128        itnumB += [len(rn)-1]
129        print(n, lmin, lam, len(rn)-1, t2)
130
131    plt.semilogx(vn,itnumC,'+', label='invit')
132    plt.semilogx(vn,itnumB,'+r', label='pinvit')
133    plt.xlabel('$n$')
134    plt.ylabel('#iteration steps')
135    plt.legend()
136    plt.savefig('pinvitDlong.eps')
137    plt.show()

```

Beispiel 7.3.19.

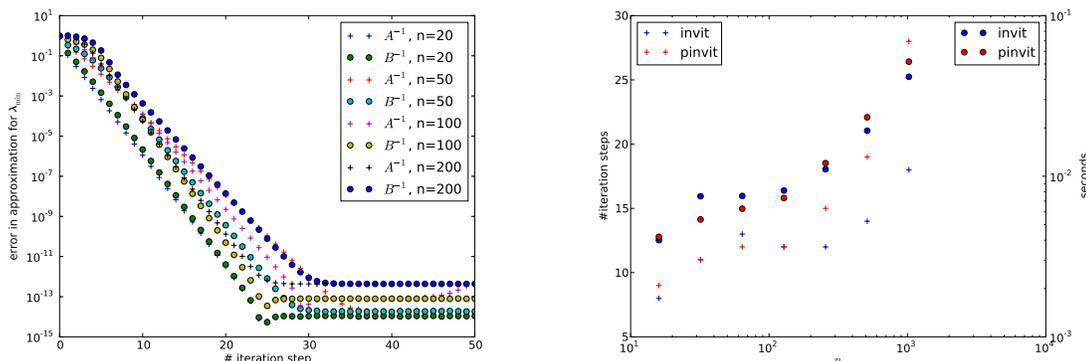


Abb. 7.3.20. Ergebnisse der tridiagonalen vorkonditionierten Iteration

Wir beobachten lineare Konvergenz; man kann beweisen, dass die Konvergenz schnell ist, wenn die Konditionszahl der Matrix  $B^{-1}A$  klein ist.

## 7.4 Krylov-Verfahren

Für kleine Matrizen ist das “direkte” QR-Verfahren für die Eigenwertberechnung hinreichend schnell. Für grosse dünnbesetzte Matrizen dauern aber die Berechnungen mit dem QR-Verfahren sehr lange. Wenn wir aber nicht an allen Eigenwerten interessiert sind, dann sind die Krylov-Methoden die richtige Wahl.

Es gibt grundsätzlich zwei Methoden, um die QR-Zerlegung einer Matrix  $A$  zu berechnen: entweder orthogonale Transformationen (Householder-Spiegelungen oder Givens-Rotationen) bis zu einer oberen Dreiecksmatrix, oder die Orthogonalisierung der Spalten via Transformationen mit oberen Dreiecksmatrizen (Gram-Schmidt). Die letzte Methode ist weniger stabil, aber der Algorithmus kann jederzeit abgebrochen werden mit dem Ergebnis in Form einer unvollständigen QR-Zerlegung. Die orthogonalisierten Vektoren können verwendet werden um eine partielle Reduktion auf die Hessenberg-Form von  $A$  zu erzielen.

Im Vergleich zu den Potenzmethoden verwendet ein Krylov-Verfahren die Information aus allen vorigen Schritten.

**Definition 7.4.1.** Sei  $0 \neq \mathbf{z} \in \mathbb{C}^n$ ,  $\mathbf{A} \in \mathbb{C}^{n \times n}$ .

Der  $\ell$ -te Krylov-Raum ist definiert als

$$\mathcal{K}_\ell(\mathbf{A}, \mathbf{z}) = \text{span}\{\mathbf{z}, \mathbf{A}\mathbf{z}, \dots, \mathbf{A}^{\ell-1}\mathbf{z}\}.$$

**Bemerkung 7.4.2.** Der  $\ell$ -te Krylov-Raum kann auch als die Menge

$$\{p(\mathbf{A})\mathbf{z}; p = \text{Polynom vom Grad } \leq \ell - 1\}$$

gesehen werden.

Eine Orthonormalbasis in  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{z})$  lässt sich mittels der Gram-Schmidt-Verfahrens iterativ bauen:

Gegeben  $\mathbf{v}_1, \dots, \mathbf{v}_\ell$  Orthonormalbasis von  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{z})$ , baue  $\mathbf{v}_{\ell+1} \in \mathcal{K}_{\ell+1}(\mathbf{A}, \mathbf{z})$ , sodass  $\mathbf{v}_{\ell+1} \perp \mathcal{K}_\ell(\mathbf{A}, \mathbf{z})$ :

$$\tilde{\mathbf{v}}_{\ell+1} := \mathbf{A}\mathbf{v}_\ell - \sum_{j=1}^{\ell} (\mathbf{v}_j^H \mathbf{A}\mathbf{v}_\ell) \cdot \mathbf{v}_j$$

$$\mathbf{v}_{\ell+1} := \tilde{\mathbf{v}}_{\ell+1} / \|\tilde{\mathbf{v}}_{\ell+1}\|_2.$$

Wir notieren

$$h_{j\ell} := \langle \mathbf{v}_j, \mathbf{A}\mathbf{v}_\ell \rangle = \mathbf{v}_j^H \mathbf{A}\mathbf{v}_\ell.$$

Um zu sehen, dass  $\tilde{\mathbf{v}}_{\ell+1} \perp \mathbf{v}_i$  für  $i = 1, 2, \dots, \ell$ , berechnen wir

$$\langle \mathbf{v}_i, \tilde{\mathbf{v}}_{\ell+1} \rangle = \langle \mathbf{v}_i, \mathbf{A}\mathbf{v}_\ell \rangle - \sum_{j=1}^{\ell} h_{j\ell} \langle \mathbf{v}_i, \mathbf{v}_j \rangle = \langle \mathbf{v}_i, \mathbf{A}\mathbf{v}_\ell \rangle - h_{i\ell} = 0,$$

denn  $\mathbf{v}_i \perp \mathbf{v}_j$  für  $i, j \in \{1, 2, \dots, \ell\}$ , wenn  $i \neq j$ .

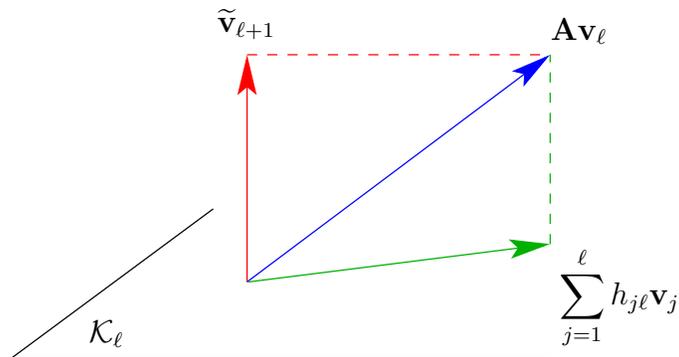


Abb. 7.4.3. Der Gram-Schmidt-Schritt für  $\mathbf{A}\mathbf{v}_\ell$

**Algorithmus** Das *Arnoldi-Verfahren* als modifiziertes Gram-Schmidt-Verfahren ist:

$$\begin{aligned} \mathbf{v}_1 &= \frac{1}{\|\mathbf{z}\|} \mathbf{z} \\ \text{für } \ell &= 1, 2, \dots, k-1 \\ \tilde{\mathbf{v}} &= \mathbf{A} \mathbf{v}_\ell \\ \text{für } j &= 1, 2, \dots, \ell : \\ h_{j\ell} &:= \mathbf{v}_j^H \tilde{\mathbf{v}} \\ \tilde{\mathbf{v}} &= \tilde{\mathbf{v}} - h_{j\ell} \mathbf{v}_j \\ h_{\ell+1,\ell} &:= \|\tilde{\mathbf{v}}\| \\ \mathbf{v}_{\ell+1} &:= \tilde{\mathbf{v}}/h_{\ell+1,\ell} \end{aligned}$$

Code 7.4.4: Arnoldi-Iteration

```

1 from numpy import dot
2
3 def arnoldi(A, v0, k):
4     V = mat(zeros((v0.shape[0],k+1), dtype=complex))
5     V[:,0] = v0.copy()/norm(v0)
6     H = mat(zeros((k+1,k), dtype=complex))
7     for m in range(k):
8         vt = dot(A, V[:, m])
9         for j in range(m+1):
10            H[j, m] = (V[:, j].H * vt)[0,0]
11            vt -= H[j, m] * V[:, j]
12        H[m+1, m] = norm(vt);
13        V[:,m+1] = vt.copy()/H[m+1, m]
14    return V, H

```

**Bemerkung 7.4.5.** Die Arnoldi-Iteration muss abgebrochen werden, wenn ein  $h_{\ell+1,\ell}$  Null wird. In diesem Fall liegt  $\mathbf{v}_{\ell+1}$  in  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{z})$ .

**Bemerkung 7.4.6.** Das Arnoldi-Verfahren baut eine Orthonormalbasis  $\mathbf{v}_1, \dots, \mathbf{v}_k$  in  $\mathcal{K}_k(\mathbf{A}, \mathbf{z})$ , sodass

$$\mathbf{v}_{\ell+1} \perp \mathcal{K}_\ell(\mathbf{A}, \mathbf{z}) \text{ für alle } \ell = 1, 2, \dots, k-1.$$

**Bemerkung 7.4.7.** Wenn wir die erhaltenen orthogonalen Vektoren  $\mathbf{v}_1, \dots, \mathbf{v}_\ell$  nebeneinander in einer Matrix schreiben:

$$\mathbf{V}_\ell := [\mathbf{v}_1, \dots, \mathbf{v}_\ell] \in \mathbb{C}^{n \times \ell}$$

und die quadratische  $(\ell \times \ell)$  Hessenberg-Matrix

$$\mathbf{H}_\ell = [h_{ij}]_{1 \leq i, j \leq \ell} \text{ mit } h_{jj} = 0 \text{ für } i > j + 1,$$

und die  $(\ell + 1) \times \ell$  Hessenberg-Matrix

$$\tilde{\mathbf{H}}_\ell = [\tilde{h}_{ij}]_{\substack{1 \leq i \leq \ell+1 \\ 1 \leq j \leq \ell}} \text{ mit } \tilde{h}_{ij} = \begin{cases} h_{ij} = \mathbf{v}_i^H \mathbf{A} \mathbf{v}_j, & \text{für } i \leq j \\ \|\tilde{\mathbf{v}}_j\|_2, & \text{für } i = j + 1 \\ 0, & \text{sonst} \end{cases}$$

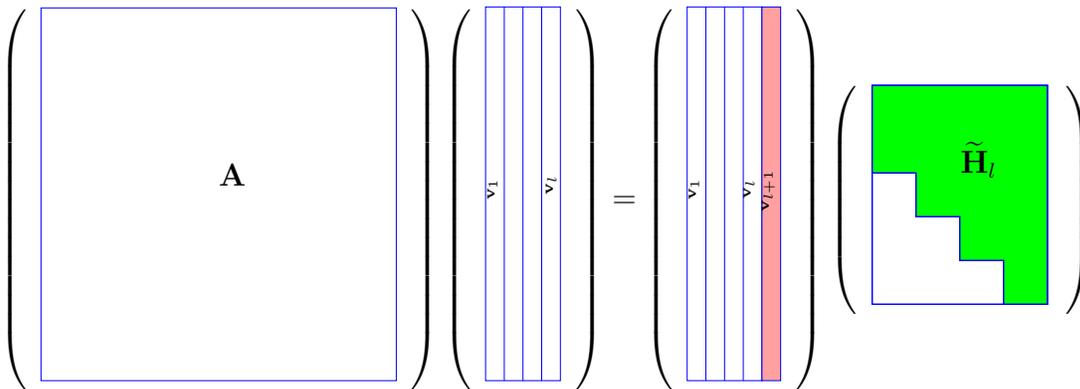
notieren, dann können wir

$$\mathbf{A} \mathbf{v}_k = h_{k+1,k} \mathbf{v}_{k+1} + \sum_{j=1}^k h_{jk} \mathbf{v}_j \text{ für } k = 1, 2, \dots, \ell$$

umschreiben in

$$\mathbf{A} \mathbf{V}_\ell = \mathbf{V}_{\ell+1} \tilde{\mathbf{H}}_\ell = \mathbf{V}_\ell \mathbf{H}_\ell + \mathbf{v}_{\ell+1} \tilde{\mathbf{h}}_{\ell+1,:} = \mathbf{V}_\ell \mathbf{H}_\ell + [\mathbf{0} \quad \tilde{h}_{\ell+1,\ell} \mathbf{v}_{\ell+1}],$$

oder graphisch ( $\ell = 4$ ):



**Bemerkung 7.4.8.**

- 1)  $\mathbf{V}_\ell^H \mathbf{V}_\ell = \mathbf{I}_\ell$ , da  $\mathbf{v}_1, \dots, \mathbf{v}_\ell$  orthonormal sind.
- 2)  $\mathbf{H}_\ell = \mathbf{V}_\ell^H \mathbf{A} \mathbf{V}_\ell$ , denn  $\mathbf{V}_\ell^H \mathbf{A} \mathbf{V}_\ell = \mathbf{V}_\ell^H \mathbf{V}_{\ell+1} \tilde{\mathbf{H}}_\ell = \mathbf{V}_\ell^H [\mathbf{V}_\ell \mathbf{v}_{\ell+1}] \tilde{\mathbf{H}}_\ell = [\mathbf{I}_\ell \mathbf{0}] \tilde{\mathbf{H}}_\ell = \mathbf{H}_\ell$ . Dabei ist  $\mathbf{H}_\ell$  eine (kleine)  $\ell \times \ell$  obere Hessenberg-Matrix und die (langen)  $\ell$  Spalten in  $\mathbf{V}_\ell$  sind orthonormal.
- 3) Das Arnoldi-Verfahren bricht ab, falls  $h_{\ell+1,\ell} = 0$  und dann:

$$\mathbf{A} \mathbf{V}_\ell = \mathbf{V}_\ell \mathbf{H}_\ell \text{ und } \mathcal{K}_{\ell+1} = 0.$$

- 4) Falls  $\mathbf{A}$  Hermite-symmetrisch ist ( $\mathbf{A}^H = \mathbf{A}$ ), dann:

$$\mathbf{H}_\ell^H = (\mathbf{V}_\ell^H \mathbf{A} \mathbf{V}_\ell)^H = \mathbf{V}_\ell^H \mathbf{A}^H \mathbf{V}_\ell = \mathbf{H}_\ell,$$

und somit ist  $\mathbf{H}_\ell$  eine tridiagonale Matrix.



*Proof.*

- 1) Sei  $\lambda$  Eigenwert zum Eigenvektor  $\mathbf{y} \neq 0$  von  $\mathbf{H}_\ell$ :

$$\mathbf{H}_\ell \mathbf{y} = \lambda \mathbf{y}.$$

Wir multiplizieren von links mit  $\mathbf{V}_\ell$  und erhalten

$$\mathbf{V}_\ell \mathbf{H}_\ell \mathbf{y} = \lambda \mathbf{V}_\ell \mathbf{y}.$$

Aber  $\mathbf{A} \mathbf{V}_\ell = \mathbf{V}_\ell \mathbf{H}_\ell$  und somit  $\mathbf{A} \mathbf{V}_\ell \mathbf{y} = \lambda \mathbf{V}_\ell \mathbf{y}$ .

- 2)  $\mathbf{A}$  regulär  $\implies 0$  ist kein Eigenwert von  $\mathbf{A} \implies 0$  ist kein Eigenwert von  $\mathbf{H}_\ell \implies \mathbf{H}_\ell$  regulär. Sei  $\mathbf{V}_\ell$  gebaut für  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{b})$ . Dann:

$$\mathbf{b} = \beta \mathbf{v}_1 = \mathbf{V}_\ell \beta \mathbf{e}_1 \quad \text{mit} \quad \mathbf{e}_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Sei  $\mathbf{y}$  so, dass  $\mathbf{H}_\ell \mathbf{y} = \beta \mathbf{e}_1$ . Dann:

$$\mathbf{A} \mathbf{V}_\ell \mathbf{y} = \mathbf{V}_\ell \mathbf{H}_\ell \mathbf{y} = \mathbf{V}_\ell \beta \mathbf{e}_1 = \mathbf{b}.$$

□

**Bemerkung 7.4.11.** Betrachten wir den linearen Operator von  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{z})$  zu  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{z})$ , definiert durch  $\mathbf{v} \mapsto$  orthogonale Projektion von  $\mathbf{A} \mathbf{v}$  auf  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{z})$ .

Da  $\mathbf{V}_\ell \mathbf{V}_\ell^H$  die orthogonale Projektion von  $\mathbb{C}^n$  auf  $\mathcal{K}_\ell(\mathbf{A}, \mathbf{z})$  ist, lässt sich der von uns betrachtete Operator in der kanonischen Basis von  $\mathbb{C}$  als  $\mathbf{V}_\ell \mathbf{V}_\ell^H \mathbf{A}$  schreiben, und in der Basis  $\{\mathbf{v}_1, \dots, \mathbf{v}_\ell\}$  als  $\mathbf{V}_\ell^H \mathbf{A} \mathbf{V}_\ell$ .

Das ist die Rayleigh-Ritz-Projektion und das ist eine wiederkehrende Idee in der angewandten Mathematik: ersetze ein Problem in einem Raum mit sehr grosser (oder unendlicher) Dimension durch ein (approximiertes) Problem in einem Raum kleiner (endlicher) Dimension. Die Eigenwerte von  $\mathbf{H}_\ell$  heissen die Ritz-Werte.

**Theorem 7.4.12.** Seien  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$  und  $\mu_1^{(\ell)} \geq \mu_2^{(\ell)} \geq \dots \geq \mu_\ell^{(\ell)}$  die Eigenwerte der Hermite-symmetrischen Matrix  $\mathbf{A} \in \mathbb{C}^{n \times n}$ , bzw. von  $\mathbf{H}_\ell = \mathbf{V}_\ell^H \mathbf{A} \mathbf{V}_\ell$  für  $\ell = 1, 2, \dots$ . Dann gelten für  $1 \leq j \leq \ell$  die Ungleichungsketten

$$\lambda_{n-j+1} \leq \mu_{\ell+1-j+1}^{(\ell+1)} \leq \mu_{\ell-j+1}^{(\ell)}$$

und

$$\mu_j^{(\ell)} \leq \mu_j^{(\ell+1)} \leq \lambda_j.$$

Mit anderen Worten, mit wachsendem  $\ell$  wächst der  $j$ -grösste Ritz-Wert monoton gegen den  $j$ -grössten Eigenwert, und der  $j$ -kleinste Ritz-Wert fällt monoton gegen den  $j$ -kleinsten Eigenwert. Für allgemeine Matrizen ist das Konvergenzverhalten komplizierter, aber im Grunde genommen ähnlich (siehe Trefethen und Bau, Lecture 34). Professionelle Codes verwenden die Bibliothek ARPACK.

Code 7.4.13: Arnoldi Eigenwertapproximation

```

1  import scipy.linalg
   import scipy.io
3  from scipy.linalg import norm
   from scipy import mat, eye, zeros, array, arange, sqrt, real, imag
5  from arnoldi import arnoldi, randomvector
   #from scipy.sparse.linalg.eigen.arnoldi import arnoldi
7  from scipy.sparse.linalg.eigen.arnoldi import arnoldi
   from numpy.random import rand
9  import time

11 def delta(i,j):
    if i==j: return 1
13     else: return 0

15 # construct matrix to be used in simulations
   def ConstructMatrix(N, case='minij'):
17     H = mat(zeros([N,N]))
    for i in range(N):
19         for j in range(N):
            if case=='sqrts':
21                 H[i,j] = 1 * sqrt((i+1)**2+(j+1)**2)+(i+1)*delta(i,j)
            else:
23                 H[i, j] = float( 1+min(i, j) )
    return H

25 # sort eigenvalues from smallest to biggest, together with their eigenvectors
27 def sortEwEV(d,v, focus=abs):
    u = v.copy()
29     #I = abs(d).argsort()
    I = focus(d).argsort() # argsort returns the indices that sort an array
31     a = d[I]
    u = u[:,I]
33     return a,u

35 if __name__ == "__main__":
    # construct/import matrix
37     #sparseformat = True
    sparseformat = False
39     if sparseformat:
        # look at http://math.nist.gov/MatrixMarket/
41         L = scipy.io.mmread('SomeMatrices/qc324.mtx.gz')
    else:
43         #L = scipy.io.loadmat('Lmat.mat')['L']
        #L = ConstructMatrix(1000)
45         L = ConstructMatrix(1000,case='sqrts')

```

```

#-----
#print L
47 #-----
N = L.shape[0]
49 nev = 10 # compute the largest nev eigenvalues and vectors
print('full matrix approach')
51 t0 = time.clock()
if sparseformat:
53     d0,V0 = scipy.linalg.eig(L.todense()) # todense() returns a dense matrix
representation of the matrix
else:
55     d0,V0 = scipy.linalg.eig(L)
print(time.clock() - t0 , 'sec')
57 V0 = mat(V0) # math () interprets the input as a matrix (see docs.scipy.org)
# let us check if the eigenvectors are orthogonal
59 print(abs(V0.H*V0 - eye(V0.shape[1]))).max())
# consider only the largest in abs.value
61 a0,U0 = sortEwEV(d0,V0)
na = U0.shape[0]
63 a0 = a0[na-nev:]
U0 = U0[:,na-nev:]
65 print('a0', a0)
#-----
67 print('my arnoldi procedure')
# compute the largest na eigenvalues and vectors
69 # but only the first eigenvectors are relevant
#v = randomvector(N)
71 v = rand(N)
na = 3*nev+1 # take 3 times more steps
73 if sparseformat:
    Lt = L.todense()
75     t0 = time.clock()
    A,H = arnoldi(Lt, v, na)
77 else:
    t0 = time.clock()
79     A,H = arnoldi(L, v, na)
print(time.clock() - t0, 'sec')
81 d1,V1 = scipy.linalg.eig(H[:-1,:])
Z = A[:, :-1]*V1
83 a1,U1 = sortEwEV(d1, Z)
a1 = a1[na-nev:]
85 U1 = U1[:,na-nev:]
print('a1', a1)
87 #-----
print('same with ARPACK')
89 if sparseformat:
    L = L.tocsr()
91     t0 = time.clock()
    d2,V2 = arpack.eigs(L,k=nev, which='LM')
93 else:
    t0 = time.clock()
95     d2,V2 = arpack.eigs(L, k=nev, ncv=na, which='LM')
print(time.clock() - t0 , 'sec')
97 a2,U2 = sortEwEV(d2,V2)
U2 = mat(U2)

```

```

99     print('a2', a2)
    #-----
101     e1 = abs(a0-a1)#/abs(a0)
    e2 = abs(a0-a2)#/abs(a0)
103     print('error in eigenvalues:\n' , e1, '\n', e2)
    print('residuals in eigenvectors:')
105     z0 = [norm(L*U0[:,k] - a0[k]*U0[:,k]) for k in range(nev) ]
    print('eig(L)', z0)
107     z1 = [norm(L*U1[:,k] - a1[k]*U1[:,k]) for k in range(nev) ]
    print('arnoldi(L)', z1)
109     z2 = [norm(L*U2[:,k] - a2[k]*U2[:,k]) for k in range(nev) ]
    print('arpack(L)', z2)
111     #-----
    # plot residuals in eigenvectors
113     from pylab import loglog, semilogy, plot, show, legend, xlabel, ylabel,
savefig, rcParams#, xlim,
ylim
    plotf = loglog
115     #plotf = semilogy

117     params = {'backend': 'ps',          # 'png' appears to be invalid.
    'axes.labelsize': 20,
119     'text.fontsize': 20,
    'legend.fontsize': 20,
121     'xtick.labelsize': 16,
    'lines.markersize' : 12,
123     'ytick.labelsize': 16#,
    #'text.usetex': True
125     }
    rcParams.update(params)
127     #-----
    plotf(abs(a0), z0, 'g-+')
129     plotf(abs(a0), z1, 'r-o')
    plotf(abs(a0), z2, 'b-v')
131     legend(('eig', 'arnoldi', 'arpack'),loc=2)
    xlabel('|Eigenvalue|')
133     ylabel('Residual of Eigenvector')
    savefig('resEVarpack.eps')
135     show()
    #-----
137     plotf(abs(a0), e1, 'ro')
    plotf(abs(a0), e2, 'bv')
139     legend(('arnoldi', 'arpack'),loc=2)
    xlabel('|Eigenvalue|')
141     ylabel('Error')
    savefig('errEWarpack.eps')
143     show()

```

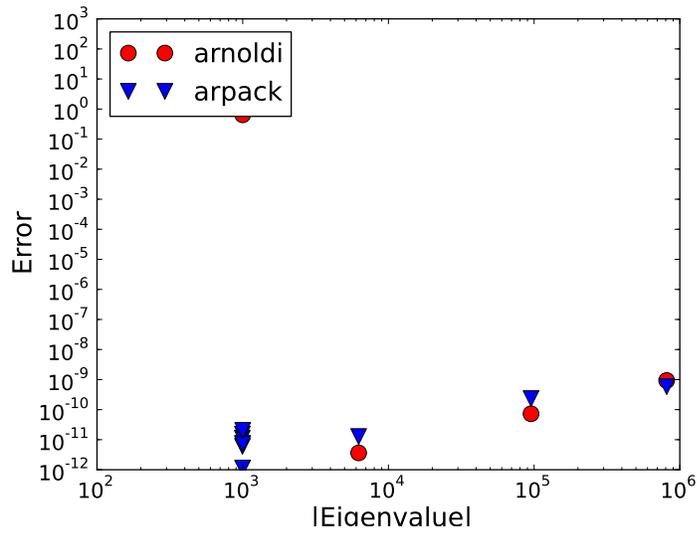


Abb. 7.4.14. Die Betragsgrossten Eigenwerte wurden korrekt approximiert, aber arpack ist besser.

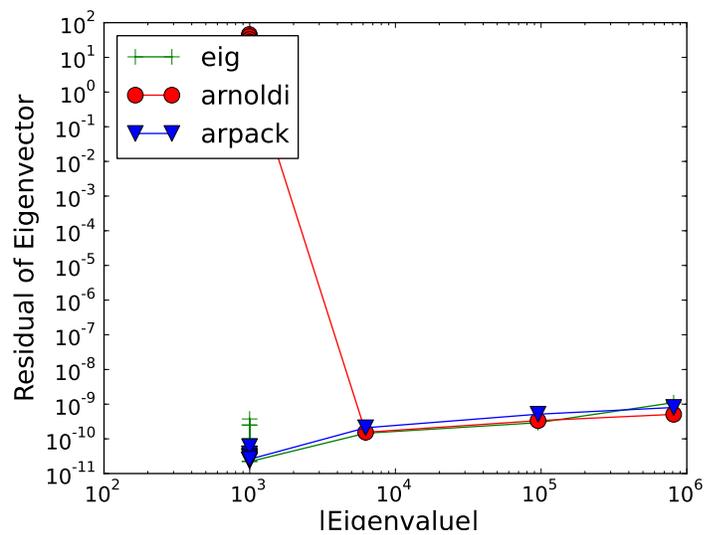


Abb. 7.4.15. Arpack ist auch für die Berechnung der entsprechenden Eigenvektoren besser als unsere Implementierung.

Beispiel 7.4.16. (Lanczos-Verfahren)

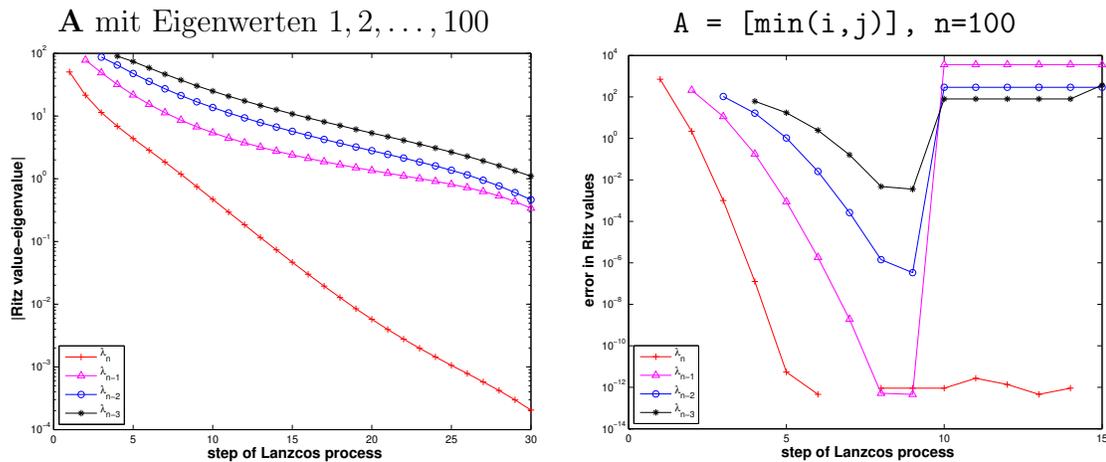


Abb. 7.4.17. Für manche Matrizen funktioniert das Lanczos-Verfahren nicht.

**Beobachte:** lineare Konvergenz der Ritz-Werte gegen die Eigenwerte.

Für  $\mathbf{A} \in \mathbb{R}^{10,10}$ ,  $a_{ij} = \min\{i, j\}$  gibt es nach anfänglicher guter Konvergenz einen abrupten Sprung der Ritz-Werte weg von den Eigenwerten! Schauen wir uns den Fall einer  $10 \times 10$ -Matrix an:

$\mathbf{A} \in \mathbb{R}^{10,10}$ ,  $a_{ij} = \min\{i, j\}$ .  $\mathbf{A} = [\min(i, j)], n=10$   
 berechnet mit `lanczos(A, n, ones(n))`, Code 7.4.9:

$$\mathbf{H}_{10} = \begin{pmatrix} 38.500000 & 14.813845 & & & & & & & & & \\ 14.813845 & 9.642857 & 2.062955 & & & & & & & & \\ & 2.062955 & 2.720779 & 0.776284 & & & & & & & \\ & & 0.776284 & 1.336364 & 0.385013 & & & & & & \\ & & & 0.385013 & 0.826316 & 0.215431 & & & & & \\ & & & & 0.215431 & 0.582380 & 0.126781 & & & & \\ & & & & & 0.126781 & 0.446860 & 0.074650 & & & \\ & & & & & & 0.074650 & 0.363803 & 0.043121 & & \\ & & & & & & & 0.043121 & 3.820888 & 11.991094 & \\ & & & & & & & & 11.991094 & 41.254286 & \end{pmatrix}$$

$$\sigma(\mathbf{A}) = \{0.255680, 0.273787, 0.307979, 0.366209, 0.465233, 0.643104, 1.000000, 1.873023, 5.048917, 44.766069\}$$

$$\sigma(\mathbf{H}_{10}) = \{0.263867, 0.303001, 0.365376, 0.465199, 0.643104, 1.000000, 1.873023, 5.048917, \mathbf{44.765976}, 44.766069\}$$

Wir sehen einen Geister-Eigenwert: 44.769976 ist ein Ritz-Wert, aber kein Eigenwert. Das widerspricht dem Theorem 7.4.10. Was ist passiert?

Die Spalten von  $V$  sollten der Theorie nach orthonormal sein, aber:

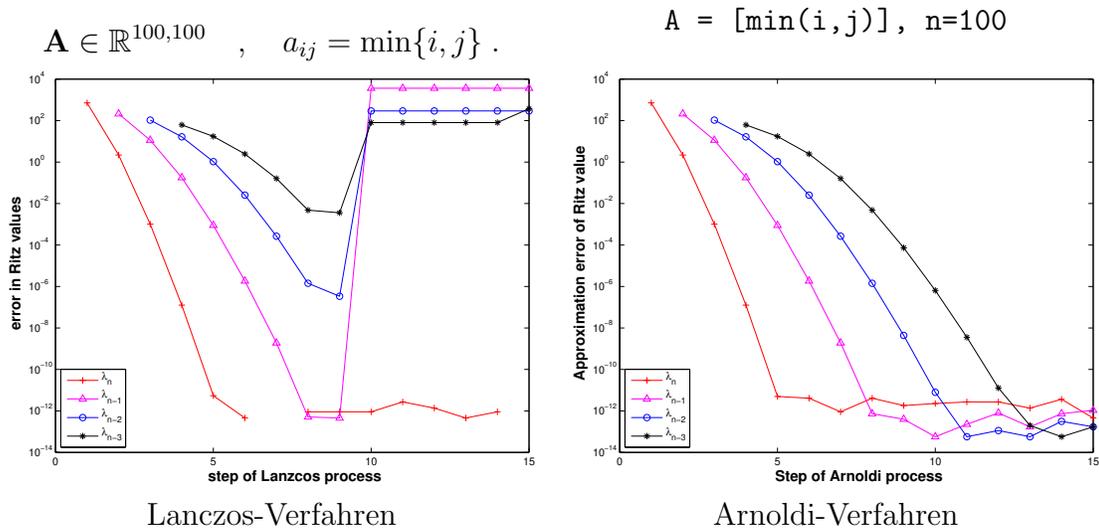
$$\mathbf{V}^H \mathbf{V} = \begin{pmatrix} 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000251 & 0.258801 & \mathbf{0.883711} \\ 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000106 & 0.109470 & 0.373799 \\ 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000005 & 0.005373 & 0.018347 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000096 & 0.000328 \\ 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000001 & 0.000003 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 & 0.000000 \\ 0.000251 & 0.000106 & 0.000005 & 0.000000 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & -0.000000 & 0.000000 \\ 0.258801 & 0.109470 & 0.005373 & 0.000096 & 0.000001 & 0.000000 & 0.000000 & -0.000000 & 1.000000 & 0.000000 \\ \mathbf{0.883711} & \mathbf{0.373799} & 0.018347 & 0.000328 & 0.000003 & 0.000000 & 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{pmatrix}$$

was eben nicht die Identitätsmatrix ist; die Spalten von  $\mathbf{V}$  sind nun nicht orthogonal zueinander. Die Rundungsfehler haben die theoretische Orthogonalität von  $\mathbf{v}_1, \dots, \mathbf{v}_\ell$  zerstört. Wenn dies passiert, dann erscheint ein Geister-Eigenwert:

$l$	$\sigma(\mathbf{H}_l)$									
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			0.276160	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	<b>44.765976</b>	44.766069

Das Arnoldi-Verfahren ist hingegen deutlich stabiler.

**Beispiel 7.4.18.** (Stabilität des Arnoldi-Verfahrens)



**Abb. 7.4.19.** Fehler in den Ritz-Werte mittels Lanczos- und Arnoldi-Verfahren

Ritz-Werte während der Arnoldi-Iteration für  $\mathbf{A} = [\min(i, j)]$ ,  $n = 10$ ;

$l$	$\sigma(\mathbf{H}_l)$									
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

**Beobachte:** Fast perfekte Approximation des Spektrums von  $\mathbf{A}$ .

In den vorherigen Beispielen sind das Arnoldi- und das Lanczos-Verfahren *algebraisch äquivalent*, denn es handelt sich um symmetrische Matrizen  $\mathbf{A} = \mathbf{A}^T$ . Trotzdem liefern sie unterschiedliche Ergebnisse, was bedeutet, dass sie **nicht numerisch äquivalent** sind.

Das Arnoldi-Verfahren ist viel weniger von den Rundungsfehlern beeinflusst als das Lanczos-Verfahren, denn das Arnoldi-Verfahren nimmt nicht automatisch an, dass die entstehenden Vektoren orthogonal sind, sondern orthogonalisiert diese in jedem Schritt. Das macht das Verfahren teurer, aber sicherer.

Da das Arnoldi-Verfahren teuer ist und Lanczos-Verfahren anfällig an Rundungsfehlern ist, wird in den professionellen Codes (ARPACK) einen Mittelweg mittels Reorthogonalisierung gewählt.

**Beispiel 7.4.20.** Eigenwertberechnung für eine nicht-symmetrische Matrix mittels Arnoldi-Verfahren:

```

1 n=100
  M = 2.*eye(n) + -0.5*eye(n,k=-1) + 1.5*eye(n, k=1); M = mat(M)
3 A = M*(diag(r_[1:n+1]))*M.I

```

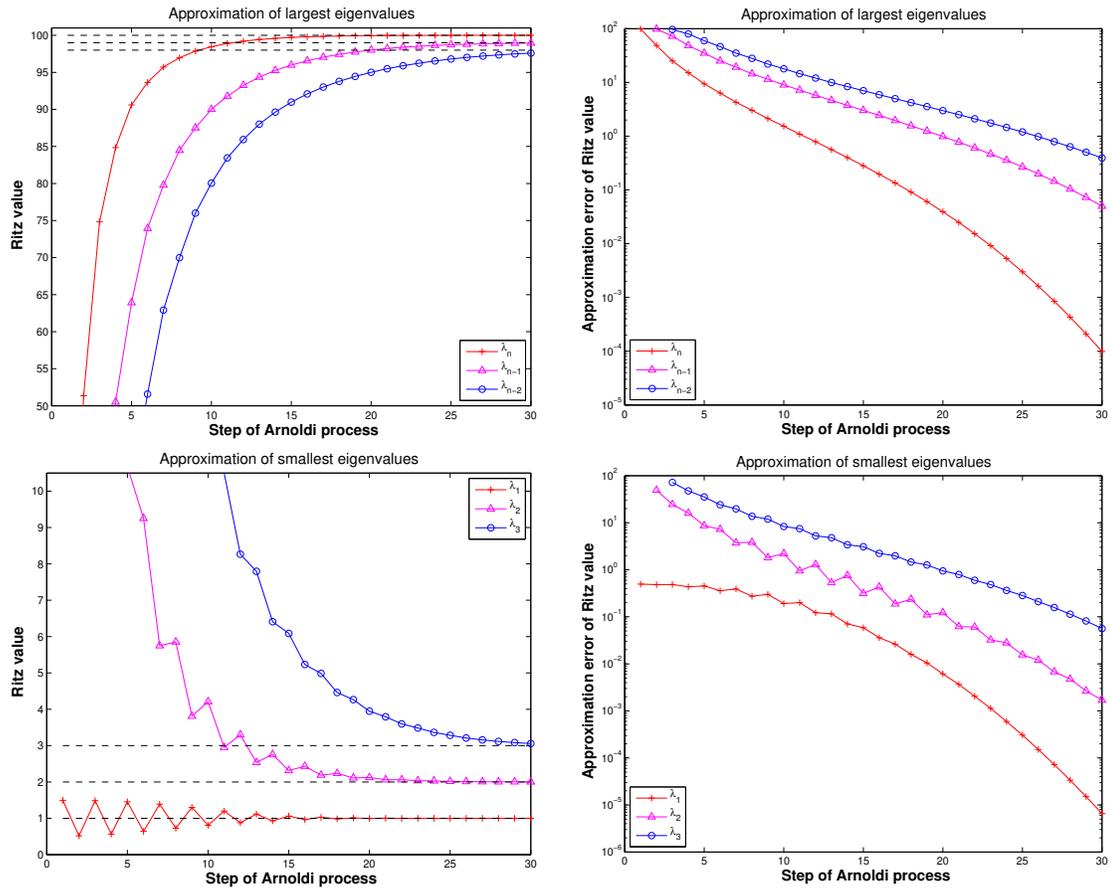


Abb. 7.4.21.

Wir beobachten eine gewisse Konvergenz für die grössten und kleinsten Eigenwerte.

# Chapter 8

## Polynominterpolation

### 8.1 Interpolation und Polynome

In diesem und im nächsten Kapitel wird es um die Interpolation von Funktionen gehen. Wir nehmen an, dass wir eine Funktion haben, die z.B. einen physikalischen Vorgang beschreibt, aber deren analytische Form sehr verschachtelt und unhandlich oder gar nicht bekannt ist. Meistens ist es aber nützlich, Extremstellen, Ableitungen oder Integrale davon zu bestimmen. Hier bietet es sich an, die komplizierte Funktion durch eine wesentlich einfachere Funktion zu approximieren. Je nach Interpolationsverfahren erhalten wir möglicherweise sogar andere Einsichten in die Funktion; z.B. können wir Informationen über die Hauptschwingung erhalten. Wir wollen also in diesem Kapitel einige Methoden zur Interpolation erarbeiten.

#### Definitionen

Das Ziel der Interpolation ist es, eine Funktion  $\tilde{f}$  zu gegebenen Datenpunkten (bzw. Funktionswerten) zu finden:

$$\begin{bmatrix} x_0 & x_1 & \cdots & x_n \\ y_0 & y_1 & \cdots & y_n \end{bmatrix}, \quad x_i \in \mathbb{R}, y_i \in \mathbb{R}.$$

Dabei heissen  $x_i$  **Stützstellen** oder **Knoten** und  $y_i = f(x_i)$  sind die dazugehörigen Datenpunkte (bzw. Funktionswerte). Die **Interpolationsbedingung**

$$\tilde{f}(x_i) = y_i \quad \text{für alle } i = 1, 2, \dots, n \quad (8.1.1)$$

sollte exakt (und nicht im Sinne der Ausgleichsrechnung) erfüllt werden.

Die interessante und vorgegebene Funktion  $f$  liegt normalerweise in einem unendlich-dimensionalen Vektorraum  $\mathcal{V}$  von Funktionen gewisser Glattheit. Wir suchen also zu  $f \in \mathcal{V}$  eine Approximation  $\tilde{f}$  in einem linearen Unterraum  $\mathcal{V}_n$  (von  $\mathcal{V}$ ) der endlichen Dimension  $\dim \mathcal{V}_n = n$ . Dabei werden wir verschiedene Basisvektoren des Raumes  $\mathcal{V}_n$  verwenden, sodass wir schreiben können:

$$f(x) \approx f_n(x) = \sum_{j=1}^n \alpha_j b_j(x).$$

Hier bilden die  $b_j(x)$  eine Basis in  $\mathcal{V}_n$ :  $\text{span}\{b_1, \dots, b_n\} = \mathcal{V}_n$ .

**Bemerkung 8.1.1.** Hier handelt es sich *nicht* um eine Ausgleichsrechnung. In der Tat bauen wir eine Interpolationsfunktion, die durch *alle* Datenpunkte  $y_i = \tilde{f}(t_i)$  laufen soll.

**Bemerkung 8.1.2.** Für einen linearen Raum  $\mathcal{V}$  kann man Unterräume  $\mathcal{V}_n$  von verschiedenen Arten von Funktionen betrachten, oder sogar für eine Art von Unterraum  $\mathcal{V}_n$  verschiedene Basen verwenden. Zunächst wollen wir uns aber ausschliesslich mit Polynomen beschäftigen. Wir werden im folgenden häufig  $p(x)$  für  $\tilde{f}(x)$  schreiben, um diesen Sachverhalt zu verdeutlichen.

**Beispiel 8.1.3.** Beispiele von möglichen  $b_j(x)$  sind:

- |   |                                 |
|---|---------------------------------|
| (1) $b_j(x) = x^{j-1}$  | polynomiale Interpolation       |
| (2) $b_j(x) = \cos((j-1) \arccos(x))$                         | Chebyshev Interpolation         |
| (3) $b_j(x) = e^{i2\pi jx} = \cos(2\pi jx) + i \sin(2\pi jx)$ | trigonometrische Interpolation. |

**Beispiel 8.1.4.** (Taylor-Formel)

Aus der Analysis ist die Taylorformel (um den Punkt  $a$ ) bereits bekannt:

$$f(x) = p_n(x) + R_n(x) \quad \text{mit Restglied} \quad R_n(x) = \frac{(x-a)^{n+1}}{(n+1)} f^{(n+1)}(\xi) \quad (8.1.2)$$

mit  $\xi$  zwischen  $a$  und  $x$ . Die Taylorformel wird häufig in der Mathematik und Physik verwendet um Funktionen lokal zu approximieren. Für numerische Berechnungen ist sie aber nicht gut genug, da sie nur lokal approximiert und auch Ableitungen benötigt werden, die wir eventuell nur schwer oder gar nicht ausrechnen können.

## Existenz

Es stellt sich natürlich die Frage, ob unsere Interpolationsfunktionen überhaupt gute Approximationen sind oder ob es überhaupt möglich ist, jede Funktion zu approximieren. Dazu kennen wir aber den aus der Analysis bekannten Satz:

**Theorem 8.1.5.** (Satz von Peano)

Angenommen  $f$  ist stetig, dann gilt:

Es existiert ein Polynom  $p$ , welches die Funktion  $f$  in der  $\|\cdot\|_\infty$ -Norm beliebig gut approximiert.

**Bemerkung 8.1.6.** Je glatter  $f$  ist, desto bessere Konvergenz erhalten wir im Allgemeinen mit den einzelnen Verfahren.

## Polynome und Monobasis

**Definition 8.1.7.** (Raum der Polynome)

Der Raum der Polynome von Grad  $\leq k$ ,  $k \in \mathbb{N}$  ist gegeben durch:

$$\mathcal{P}_k := \{x \mapsto \alpha_k x^k + \alpha_{k-1} x^{k-1} + \cdots + \alpha_1 x + \alpha_0, \alpha_j \in \mathbb{K}\}. \quad (8.1.3)$$

**Definition 8.1.8.** (Monom)

Die Funktionen der Form  $x \mapsto x^k$ ,  $k \in (\mathbb{N} \cup \{0\})$  heissen *Monome*.

$x \mapsto \alpha_k x^k + \alpha_{k-1} x^{k-1} + \cdots + \alpha_0$  ist die *Monom-Darstellung* eines Polynoms.

**Theorem 8.1.9.** (Eigenschaften des Raumes der Polynome)

$\mathcal{P}_k$  ist ein Vektorraum der Dimension  $k+1$ . Zusätzlich gilt  $\mathcal{P}_k \subset C^\infty(\mathbb{R})$ .

*Proof.* Die Monome vom Grad  $\leq k$  bilden eine Basis des Vektorraumes der Polynome, also gilt  $\dim \mathcal{P}_k = k+1$ . Ausserdem sind alle Polynome  $\infty$ -Mal stetig differenzierbar und liegen somit in  $C^\infty(\mathbb{R})$  (man schreibt  $\mathcal{P}_k \subset C^\infty(\mathbb{R})$ ).  $\square$

## Berechnung mit der Monobasis

**Theorem 8.1.10.** (Eindeutigkeit)

Ein Polynom  $p(x)$   $k$ -ten Grades ist eindeutig durch  $k+1$  Punkte  $y_i = p(t_i)$  bestimmt.

Damit haben wir bereits eine Methode zur Polynomdarstellung kennengelernt. Wir müssen lediglich die Funktion, die wir approximieren wollen, an einer bestimmten Anzahl an Stützstellen auswerten. Danach lassen sich die Koeffizienten in der Monobasis durch Lösen eines Gleichungssystems leicht bestimmen.

**Beispiel 8.1.11.** (Berechnung der Koeffizienten in der Monobasis)

Wir möchten das Polynom durch die Punkte  $y_0, \dots, y_n$  berechnen:

Das gesuchte Polynom  $p_n \in \mathcal{P}_n$  hat in der Monobasis die Form:

$$p_n(x) = \alpha_n x^n + \cdots + \alpha_1 x^1 + \alpha_0.$$

Wir finden also das lineare Gleichungssystem:

$$\begin{bmatrix} 1 & t_0 & \cdots & t_0^n \\ 1 & t_1 & \cdots & t_1^n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & t_n & \cdots & t_n^n \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

**Bemerkung 8.1.12.** Dieses Gleichungssystem weist die Vandermonde matrix auf; deswegen ist es sehr schlecht konditioniert (die Vandermonde Matrix haben wir bereits in einem Beispiel bei der QR-Zerlegung gesehen, cf. Abbildung 5.1.30). Würden die Koeffizienten  $\alpha_0, \dots, \alpha_n$  daraus berechnet, würden diese sehr ungenau ausfallen. Die Funktion `polyfit` kann für die bequeme Berechnung von  $\alpha_0, \dots, \alpha_n$  verwendet werden, sie gibt aber auch die Warnung `RankWarning: Polyfit may be poorly conditioned`.

### Auswertung und Bemerkungen

Allgemein lassen sich Polynome, wie im Beispiel oben, über die Matrixdarstellung auswerten: Mit bekannten  $\alpha_i$  lassen sich aus den Auswertungstellen  $t_i$  die dazugehörigen  $y_i$  bestimmen. Eine effizientere Möglichkeit der Auswertung ist aber das sogenannte Horner-Schema:

**Definition 8.1.13.** (Horner-Schema)

$$p(x) = (x \dots x (x (\alpha_n x + \alpha_{n-1}) + \dots + \alpha_1) + \alpha_0). \quad (8.1.4)$$

**Bemerkung 8.1.14.** das Polynom  $p_n(x) = \alpha_n x^n + \dots + \alpha_1 x^1 + \alpha_0$  wird in `python` durch ein array `p = [\alpha_n, \alpha_{n-1}, \dots, \alpha_1, \alpha_0]` der Länge  $n + 1$  repräsentiert. Das Modul `numpy` stellt die Funktion `polyval(p, x)` zur Verfügung, um ein solches Polynom via Horner-Schema schnell auszuwerten. Mit unserer Notation haben wir `p[0] = \alpha_n`, `p[1] = \alpha_{n-1}`,  $\dots$ , `p[n] = \alpha_0` und eine mögliche Implementierung ist:

#### Code 8.1.15: Horner-Schema

```

1  def horner(p, x):
2      y = p[0]
3      for i in range(1, len(p)):
4          y = x * y + p[i]
5      return y

```

`p` ist hier ein array mit den Koeffizienten  $\alpha_i$  des Interpolationspolynoms.

**Bemerkung 8.1.16.** Die Monombasis ist zwar sehr anschaulich, hat aber keine nützlichen numerischen Eigenschaften, sondern eher Nachteile. Daher wird diese auch nur sehr selten verwendet. Das Horner-Schema wirkt aber stabilisierend: obwohl `polyfit` grosse Fehler in die Berechnung der Koeffizienten  $\alpha_0, \dots, \alpha_n$  produziert, liefert die Auswertung über `polyval` wieder relativ gute Ergebnisse, wie der folgende Code zeigt.

#### Code 8.1.17: Horner-Schema/polyval ist besser

```

1  import numpy as np
2  t = np.linspace(-5., 5.)
3  runge = lambda x: 1./(1+x**2) # function to be interpolated
4  y = runge(t)
5  N = len(y)
6  p = np.polyfit(t, y, N-1) #here we used numpy's polyfit
7
8  # the 'foolish' evaluation method:
9  def evalmyp(p, x):
10     res = 0.
11     n = len(p)
12     for j in range(n):
13         res += p[N-1-j]*x**j
14     return res

```

```

15     z = evalmyp(p,t)
17
18     # now we compare polyfit with the 'foolish' method
19     print('polyfit + foolish evaluation gives an error:', abs(z-y).max())
20     w = np.polyval(p,t)
21     print('polyfit + polyval gives an error:', abs(w-y).max())

```

p ist hier ein array mit den Koeffizienten  $\alpha_i$  des Interpolationspolynoms.

## 8.2 Newton Basis und Dividierte Differenzen

### Grundlagen

In der Monombasis können neu gewonnene Datenpunkte nur durch Neuberechnung des Interpolationspolynoms eingebunden werden. Die Newton-Basis erlaubt das Hinzufügen von Datenpunkte, ohne die bereits vorhandene Informationen zu verworfen. Betrachten wir hierzu zunächst den einfachen Fall, wo wir nur eine Stützstelle haben und eine weitere hinzufügen möchten:

#### Beispiel 8.2.1. (Motivation Newtonbasis)

Angenommen, wir haben zunächst nur eine Stützstelle  $(x_0, y_0)$  zur Verfügung. Daraus ergibt sich trivialerweise als Interpolationspolynom z.B.  $p_0(x) = y_0$ . Nun möchten wir eine zweite Stützstelle  $(x_1, y_1)$  hinzufügen, ohne das bisher berechnete Polynom zu verwerfen. Mithilfe der Interpolationsbedingungen  $p_0(x_0) = y_0$  und  $p_1(x_1) = y_1$  finden wir folgendes Polynom:

$$p_0(x_0) = y_0 \quad \xrightarrow{(x_1, y_1)} \quad p_1(x) = p_0(x) + (x - x_0) \frac{(y_1 - y_0)}{(x_1 - x_0)},$$

Nun möchten wir noch eine weitere Stützstelle  $(x_2, y_2)$  hinzufügen. Wir wenden also das selbe Schema an:

$$p_2(x) = p_1(x) + \frac{\left(\frac{y_2 - y_1}{x_2 - x_1}\right) - \left(\frac{y_1 - y_0}{x_1 - x_0}\right)}{x_2 - x_0} (x - x_0)(x - x_1).$$

Dieses Schema lässt sich beliebig fortführen.

**Bemerkung 8.2.2.** Nach dem Theorem 8.1.10 sind die im obigen Beispiel gefundenen Polynome eindeutig und entsprechen das Polynomen, die wir mit der "naiven Herangehensweise" erhalten würden.

### Newton Basis und Dividierten Differenzen

#### Theorem 8.2.3. (Newtonbasis)

Die Newtonbasis:

$$N_0(x) := 1, \quad N_1(x) := x - x_0, \quad N_2(x) := (x - x_0)(x - x_1), \dots, \quad N_n(x) := \prod_{i=0}^{n-1} (x - x_i) \quad (8.2.5)$$

ist eine Basis des Raumes  $\mathcal{P}_n$ .

Wie erhalten wir im Allgemeinen die Koeffizienten  $\beta_j$  zur Darstellung eines Polynoms in der Newtonbasis?

$$p_n(x) = \beta_0 N_0(x) + \beta_1 N_1(x) + \dots + \beta_n N_n(x), \quad \text{mit } \beta_j \in \mathbb{R}.$$

Wir können diese Koeffizienten  $\beta_j$  durch das Lösen eines linearen Gleichungssystems erhalten:

$$\begin{aligned} \beta_0 N_0(x_j) + \beta_1 N_1(x_j) + \dots + \beta_n N_n(x_j) &= y_j, \quad j = 0, \dots, n \\ \Leftrightarrow \begin{bmatrix} 1 & 0 & \dots & 0 \\ 1 & (x_1 - x_0) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & (x_n - x_0) & \dots & \prod_{i=0}^{n-1} (x_n - x_i) \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix} &= \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{bmatrix}. \end{aligned}$$

Dabei haben wir verwendet:

$$N_n(x) = \prod_{i=0}^{n-1} (x - x_i) = 0 \quad \text{für } x = x_0, \dots, x_{n-1}.$$

**Bemerkung 8.2.4.** Beim Lösen dieses Gleichungssystems fällt auf, dass immer wieder die selben Ausdrücke berechnet werden. Die Dividierten Differenzen nutzen diesen Sachverhalt und stellen damit eine effizientere Methode zur Berechnung der Koeffizienten dar.

**Definition 8.2.5.** (Dividierte Differenzen)

Die dividierten Differenzen sind definiert durch:

$$y[x_i] = y_i \quad \text{und} \quad y[x_i, \dots, x_{i+k}] = \frac{y[x_{i+1}, \dots, x_{i+k}] - y[x_i, \dots, x_{i+k-1}]}{x_{i+k} - x_i}. \quad (8.2.6)$$

Zur Berechnung verwendet man folgendes Schema:

$$\begin{array}{l|l} x_0 & y[x_0] \\ & > y[x_0, x_1] \\ x_1 & y[x_1] \\ & > y[x_0, x_1, x_2] \\ & > y[x_1, x_2] \\ & > y[x_0, x_1, x_2, x_3] \\ x_2 & y[x_2] \\ & > y[x_1, x_2, x_3] \\ & > y[x_2, x_3] \\ x_3 & y[x_3], \end{array} \quad (8.2.7)$$

wobei jedes ”>” für eine in (8.2.6) definierte Rekursion steht.

**Bemerkung 8.2.6.** (Äquidistante Stützstellen)

Falls die  $x_j$  äquidistant mit Abstand  $h$  sind, d.h. falls  $x_j = x_0 + j \cdot h$ , so können wir schreiben:

$$\begin{aligned} y[x_0, x_1] &= \frac{y_1 - y_0}{h} &:= \frac{1}{h} \Delta y_0 \\ y[x_0, x_1, x_2] &= \frac{\frac{1}{h} \Delta y_1 - \frac{1}{h} \Delta y_0}{2h} &:= \frac{1}{2! h^2} \Delta^2 y_0 \\ &\vdots & \\ y[x_0, \dots, x_n] &= \frac{1}{n! h^n} \Delta^n y_0. \end{aligned} \tag{8.2.8}$$

Beachte: Das  $n$  in  $\Delta^n$  ist als Index zu verstehen, nicht als Potenz!

**Code 8.2.7:** Dividierte Differenzen

```

1  def divdiff(x, y):
2      n = y.size
3      T = np.zeros((n, n))
4      T[:,0] = y
5      for level in range(1, n):
6          for i in range(n-level):
7              T[i, level] = (T[i+1, level-1] - T[i, level-1]) / (x[i+level]-x[i])
8
9      return T[0, :]

```

**Theorem 8.2.8.** (Newton, 1676)

Es gibt ein Polynom  $n$ -ten Grades, das durch  $(x_0, y_0), \dots, (x_n, y_n)$  geht. Dieses lässt sich wie folgt berechnen:

$$p(x) = y[x_0] + y[x_0, x_1](x - x_0) + \dots + y[x_0, x_1, \dots, x_n](x - x_0) \cdots (x - x_{n-1}).$$

Die Koeffizienten  $\beta_j$  in der Newtonbasis sind also gegeben durch  $\beta_j = y[x_0, x_1, \dots, x_j]$ .

**Auswertung, Kosten und Fehler**

Für die Auswertung des Interpolationspolynoms bietet sich wieder ein abgewandeltes Horner-Schema an:

Wir verwenden  $p$  im folgenden Algorithmus als "Speichervariable":

$$\begin{aligned} p &:= \beta_n \\ p &:= (x - t_{n-1})p + \beta_{n-1} \\ p &:= (x - t_{n-2})p + \beta_{n-2}. \end{aligned}$$

**Bemerkung 8.2.9.** Der Aufwand (gegliedert nach den Teilschritten) ist gegeben durch:

- (1)  $O(n^2)$  für die Berechnung der dividierten Differenzen;
- (2)  $O(n)$  jeweils für jede Auswertung.

Bisher haben wir uns noch nicht mit dem Fehler  $\|f(x) - p(x)\|$  des Interpolationspolynoms beschäftigt. Insbesondere zur Entwicklung besserer Interpolationsverfahren möchten wir den Fehler nun genauer abschätzen.

**Theorem 8.2.10.** Angenommen,  $f$  ist  $n$ -mal stetig differenzierbar und  $y_i = f(x_i)$  für  $i = 0, \dots, n$ , dann existiert ein  $\xi \in ]\min_i x_i, \max_i x_i[$ , so dass

$$y[x_0, x_1, \dots, x_n] = \frac{f^{(n)}(\xi)}{n!}. \quad (8.2.9)$$

**Theorem 8.2.11.** (Fehler)

Sei  $f : [a, b] \rightarrow \mathbb{R}$   $(n+1)$ -mal stetig differenzierbar und  $p$  das Interpolationspolynom zu  $f$  in  $x_0, x_1, \dots, x_n \in [a, b]$ . Dann gilt:  
für jedes  $x \in [a, b]$  gibt es ein  $\xi \in ]a, b[$  mit

$$f(x) - p(x) = (x - x_0) \cdots (x - x_n) \frac{f^{(n+1)}(\xi)}{(n+1)!}. \quad (8.2.10)$$

**Beispiel 8.2.12.** Als Beispiel einer Anwendung dieses Satzes wollen wir uns die Funktion  $f(x) = \log_{10} x = \frac{\ln x}{\ln 10}$  auf dem Intervall  $[55, 58]$  anschauen. Als Stützstellen verwenden wir  $\{55, 56, 57, 58\}$ . Als Fehler an der Stelle  $x = 56.5$  ergibt sich:

$$|\log_{10}(56.5) - p(56.5)| \leq 1.5 \cdot 0.5 \cdot 0.5 \cdot 1.5 \cdot \frac{6}{(55)^4 \ln(10)} \cdot \frac{1}{4!} \approx 6.7 \cdot 10^{-9}.$$

Der geringe Fehler, den wir erhalten haben, lässt sich vor allem auf das kleine Intervall und die geringe Oszillation der Funktion zurückführen.

Aus der Formel zur Abschätzung des Fehlers geht hervor, dass die Wahl der Stützstellen massgeblichen Einfluss auf den Fehler hat. Also stellt sich natürlich die Frage, welche Wahl  $x_1, \dots, x_n$  der Stützstellen ist am besten?

D.h. wann ist  $\max_{x \in [a, b]} |(x - x_0) \cdots (x - x_n)|$  minimal?

Eine andere Frage ist, wie wirken sich Fehler in den Funktionswerten (z.B. Messfehler) auf das Interpolationspolynom aus?

$$\left. \begin{array}{l} (x_i, y_i) \quad \text{für } i = 0, 1, \dots, n \rightarrow p(x) \\ (x_i, \tilde{y}_i) \quad \text{für } i = 0, 1, \dots, n \rightarrow \tilde{p}(x) \end{array} \right\} p(x) - \tilde{p}(x) = ?$$

Um diese Fragen zu beantworten, wechseln wir in eine andere Basis des Raumes der Polynome  $\mathcal{P}$ , die sogenannten Lagrange-Polynome. Die Lagrange-Basis hat einige nützliche Eigenschaften, die uns helfen werden, diese Fragen zu beantworten.

## 8.3 Lagrange- und baryzentrische Interpolationsformeln

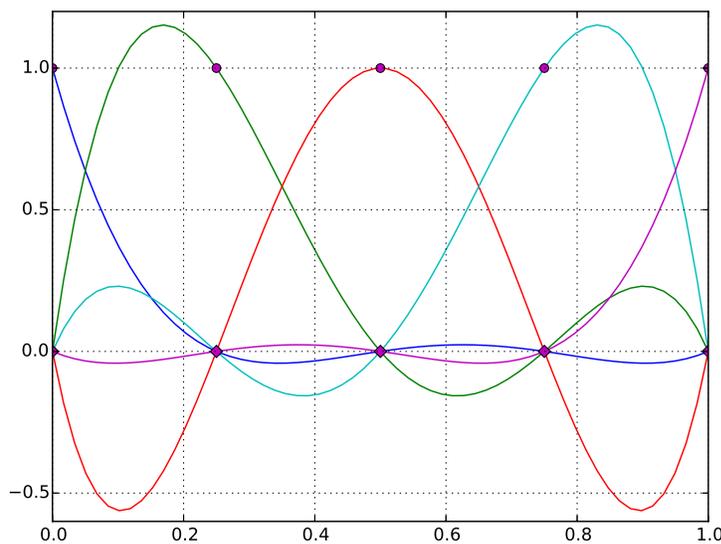
Im letzten Abschnitt haben wir uns mit dem Fehler im Interpolationspolynom beschäftigt. Wir möchten hier eine Basis einführen, die uns die Untersuchung des

Fehlers deutlich vereinfacht. Diese Basis ist zur praktischen Berechnung von Interpolationspolynomen ungeeignet. Stattdessen wird die baryzentrische Formel verwendet.

**Theorem 8.3.1.** (Lagrange Interpolationsformel)

Die Lagrange-Polynome  $l_i(x)$  zu den Stützstellen  $(x_0, y_0), \dots, (x_n, y_n)$  bilden eine Basis des Raumes der Polynome  $\mathcal{P}_{n+1}$ , und wir können schreiben:

$$p(x) = \sum_{i=0}^n y_i l_i(x) \quad \text{mit} \quad l_i(x) = \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}. \quad (8.3.11)$$



**Abb. 8.3.2.** Die 5 Lagrange Polynome zu Knoten  $0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1$ .

**Bemerkung 8.3.3.** Die ersten drei Lagrange-Polynome für die Stützstellen  $(x_0, x_1, x_2)$  sind:

$$l_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)},$$

$$l_1(x) = \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)},$$

$$l_2(x) = \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

Es gelten folgende Eigenschaften:

1.  $l_i(x_j) = 0$  für alle  $j \neq i$ ,
2.  $l_i(x_i) = 1$  für alle  $i$ ,

3.  $\text{grad } l_i = n$  für alle  $i$ .

4.  $\sum_{k=0}^n \ell_k(x) = 1$  und  $\sum_{k=0}^n \ell_k^{(m)}(x) = 0$  für  $m > 0$ .

Wenn wir  $L(x) = (x - x_0)(x - x_1) \cdots (x - x_n)$  und

$$\lambda_k = \prod_{j \neq k} \frac{1}{x_k - x_j} = \frac{1}{(x_k - x_0) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}$$

dann lässt sich  $p$  umschreiben in

$$p(x) = L(x) \sum_{k=0}^n \frac{\lambda_k}{x - x_k} y_k.$$

Für das konstante Polynom 1 ergibt das

$$1 = L(x) \sum_{k=0}^n \frac{\lambda_k}{x - x_k}$$

was zur baryzentrischen Formel für das Interpolationspolynom führt:

$$p(x) = \frac{\sum_{k=0}^n \frac{\lambda_k}{x - x_k} y_k}{\sum_{k=0}^n \frac{\lambda_k}{x - x_k}}. \tag{8.3.12}$$

Diese Formel ist skalierungsinvariant bezüglich des Interpolationsintervalls  $[a, b]$  mit  $a = x_0 < \dots, x_n = b$ ; die Auswertung der baryzentrischen Formel ist extrem stabil für  $x \in [a, b]$  und lässt sich mit  $\mathcal{O}(n)$  Operationen durchführen, wenn man die  $\lambda_k$  im Voraus ausrechnet (in  $\mathcal{O}(n^2)$  Operationen). Auch einige zusätzliche Stützstellen lassen sich leicht hinzufügen; die darauf basierte Neuauswertung des entstandenen Interpolationspolynoms kostet nur  $\mathcal{O}(n)$  Operationen. Für eine spezielle Wahl der Stützstellen ( $n + 1$  Chebyshev-Abszissen  $x_k = \cos(k\pi/n)$ ) sind  $\lambda_k = (-1)^k$ . Wenn wir andere  $\lambda_k$  zulassen, dann ergibt die baryzentrische Formel kein Polynom mehr, so dass sich der Weg zu einer Verallgemeinerung zur Interpolation mittels rationaler Funktionen öffnet.

Die baryzentrische Formel kann auch als Startpunkt zur spektralen Methode für die Differentialgleichungen genommen werden. Wenn wir die unbekannte Funktion  $u$  durch dessen Interpolationspolynom ersetzen und die Ableitungen ausrechnen, dann brauchen wir  $\ell'_j(x_i)$  und  $\ell''_j(x_i)$  um die Differentialgleichung der Form  $F(u'', u', x) = 0$  zu einem linearen Gleichungssystem in den Unbekannten  $y_i = u(x_i)$  umzuschreiben. Eine geschickte Rechnung ergibt für  $i \neq j$ :

$$\ell'_j(x_i) = \frac{\lambda_j/\lambda_i}{x_i - x_j} \tag{8.3.13}$$

$$\ell''_j(x_i) = -2 \frac{\lambda_j/\lambda_i}{x_i - x_j} \left( \sum_{k \neq j} \frac{\lambda_k/\lambda_i}{x_k - x_j} - \frac{1}{x_i - x_j} \right) \tag{8.3.14}$$

während für  $i = j$  gilt

$$\begin{aligned}\ell_j'(x_j) &= -\sum_{k \neq j} \frac{\lambda_k / \lambda_j}{x_k - x_j} \\ \ell_j''(x_j) &= -\sum_{k \neq j} \ell_k''(x_j).\end{aligned}$$

### Fehlerbetrachtung

Wenden wir uns zunächst dem Fall zu, dass, z.B. durch ungenaue Messungen, die Werte  $\tilde{y}_i$  an den Stützstellen  $x_i$  mit Fehlern behaftet sind. Wir möchten nun den Fehler zwischen dem korrekten Interpolationspolynom  $p(x)$  und dem mit Fehlern behafteten Polynom  $\tilde{p}(x)$  berechnen. Dazu schreiben wir  $p$  und  $\tilde{p}$  in der Lagrange-Basis:

$$p(x) = \sum_{i=0}^n y_i l_i(x) \quad \text{und} \quad \tilde{p} = \sum_{i=0}^n \tilde{y}_i l_i(x).$$

Dann ergibt sich für den Fehler:

$$|p(x) - \tilde{p}(x)| = \left| \sum_{i=0}^n (y_i - \tilde{y}_i) l_i(x) \right| \leq \max_{i=0, \dots, n} |y_i - \tilde{y}_i| \cdot \sum_{i=0}^n |l_i(x)|.$$

### Definition 8.3.4. (Lebesgue-Konstante)

Die Lebesgue-Konstante  $\Lambda$  zu den Stützstellen  $x_0, \dots, x_n$  im Intervall  $[a, b]$  ist definiert durch

$$\Lambda_n := \max_{x \in [a, b]} \sum_{i=0}^n |l_i(x)|. \quad (8.3.15)$$

**Bemerkung 8.3.5.** Die Lebesgue-Konstante hängt offensichtlich nur von der Wahl der Stützstellen ab.

### Theorem 8.3.6. (Auswirkung von Messfehlern)

Es gilt:

$$\max_{x \in [a, b]} |p(x) - \tilde{p}(x)| \leq \Lambda_n \max_{i=0, \dots, n} |y_i - \tilde{y}_i|, \quad (8.3.16)$$

wobei die Lebesgue-Konstante  $\Lambda_n$  die bestmögliche Konstante in dieser Ungleichung ist.

### Theorem 8.3.7. (Fehler)

Sei  $f : [a, b] \rightarrow \mathbb{R}$  und  $p$  das Interpolationspolynom zu  $f$ , Seien  $x_0, \dots, x_n$  die Stützstellen, dann gilt:

$$\|f(x) - p(x)\|_\infty = \max_{x \in [a, b]} |f(x) - p(x)| \leq (1 + \Lambda_n) \max_{x \in [a, b]} |f(x) - q(x)| \quad \text{für alle } q \in \mathcal{P}_n.$$

*Proof.* Sei  $q \in \mathcal{P}_n$ , dann gilt:

$$|f(x) - p(x)| = |[f(x) - q(x)] + [q(x) - p(x)]| \leq |f(x) - q(x)| + |q(x) - p(x)| .$$

Seien  $y_i = f(x_i)$  und  $\tilde{y}_i = q(x_i)$ . Wir wissen, dass  $p$  eine Approximation von  $f$  ist und dass  $q$  sein eigenes Interpolationspolynom zu sich selbst ist. Dann:

$$\max_{x \in [a,b]} |p(x) - q(x)| \leq \Lambda_n \max_{i=0,1,\dots,n} \|f(x_i) - q(x_i)\| \leq \Lambda_n \max_{x \in [a,b]} |f(x) - q(x)| .$$

Die letzte zwei Ungleichungen ergeben dann:

$$\max_{x \in [a,b]} |f(x) - p(x)| \leq \max_{x \in [a,b]} |f(x) - q(x)| + \Lambda_n \max_{x \in [a,b]} |f(x) - q(x)| .$$

□

**Beispiel 8.3.8.** (Lebesgue-Konstante)

Auf dem Intervall  $[-1, 1]$  ist die Lebesgue-Konstante für eine äquidistante Verteilung der Stützstellen  $x_j = -1 + \frac{2}{n}j$ :

$$\Lambda_{10} \approx 40, \quad \Lambda_{20} \approx 30000, \quad \Lambda_{40} \approx 10^{10} .$$

Wir sehen also, dass schon für etwa 40 Stützstellen der Fehler sehr gross werden kann!

**Bemerkung 8.3.9.** Für äquidistant verteilte Stützstellen gilt:

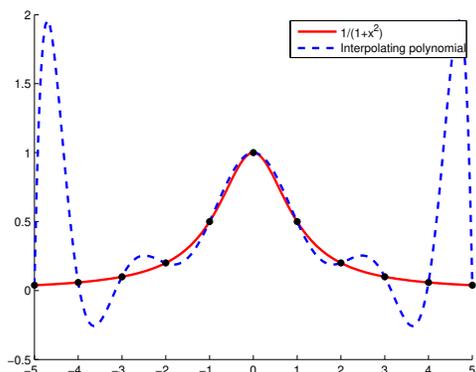
$$\Lambda_n \approx \frac{2^{n+1}}{n \log n} .$$

**Beispiel 8.3.10.** (Runge-Funktion)

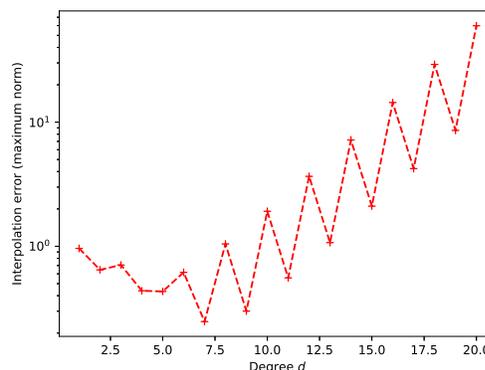
Als Beispiel betrachten wir die sogenannte Runge-Funktion auf dem Intervall  $[-5, 5]$ :

$$f(x) = \frac{1}{1 + x^2} .$$

Zur Berechnung des Interpolationspolynoms wählen wir äquidistant verteilte Stützstellen. Das Interpolationspolynom ist in Theorem 8.3.11 dargestellt. Wir sehen, dass in den Randbereichen starke Oszillationen auftreten. Die Theorem 8.3.12 zeigt, dass wir durch höheren Grad des Interpolationspolynoms sogar noch höhere Fehler erhalten.



**Abb. 8.3.11.** Interpolationspolynom



**Abb. 8.3.12.**  $\|f - p\|_\infty$  auf  $[-5, 5]$

Die Wahl der Stützstellen hat erheblichen Einfluss auf den Fehler. Aus dem obigen Beispiel lässt sich zudem eine einfache Grundregel für die Interpolation ableiten: **Verwende für eine Interpolation hohen Grades *niemals* äquidistant verteilte Stützstellen!**

Wie können wir genauere Interpolationen durch höheren Grad des Interpolationspolynoms erhalten?

Mögliche Ansatzpunkte könnten sein:

1. das Intervall in kleinere Intervalle zerteilen und für jeden Teil ein eigenes Interpolationspolynom berechnen.
2. die ideale Verteilung der Stützstellen wählen, die diesen Nachteil nicht haben.

Die erste Idee bringt uns zu Splines und Finite Elemente, die aber nicht Teil dieser Vorlesung sind. Im nächsten Kapitel werden wir uns mit der zweiten Idee näher beschäftigen und eine bessere Verteilung der Stützstellen kennenlernen.

## 8.4 Chebyshev-Interpolation

Hier möchten wir durch geschickte Wahl der Stützstellen den Interpolationsfehler verkleinern. Bisher haben wir immer äquidistant verteilte Stützstellen verwendet, in diesem Kapitel werden wir die Chebyshev-Knoten als die ideale Verteilung kennenlernen.

**Definition 8.4.1.** (Chebyshev-Polynome erster Art)

Die Chebyshev-Polynome (erster Art) sind:

$$T_n(x) = \cos(n \arccos(x)), \quad x \in [-1, 1]. \quad (8.4.17)$$

**Definition 8.4.2.** (Chebyshev-Polynome zweiter Art)

Die Chebyshev-Polynome zweiter Art sind:

$$U_n(x) = \frac{\sin((n+1) \arccos(x))}{\sin(\arccos(x))}, \quad x \in [-1, 1]. \quad (8.4.18)$$

Auf dem ersten Blick ist es nicht sofort einzusehen, dass die  $T_n(x)$  überhaupt Polynome sind! Im folgenden Satz wird die Aussage aber klar.

**Theorem 8.4.3.** (Eigenschaften)

Das  $n$ -te Chebyshev-Polynom ist ein Polynom von Grad  $n$  und es gilt für  $x \in [-1, 1]$ :

1.  $T_0(x) = 1, T_1(x) = x, T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$ ;
2.  $|T_n(x)| \leq 1$ ;
3.  $T_n(\cos(\frac{k\pi}{n})) = (-1)^n$  für  $k = 0, 1, \dots, n$ ;

4.  $T_n(\cos(\frac{(2k+1)\pi}{2n})) = 0$  für  $k = 0, 1, \dots, n - 1$ .

*Proof.* Es gilt:

$$\cos((n + 1)t) = 2 \cos(nt) \cos(t) - \cos((n - 1)t).$$

Für  $\cos(t) = x$  folgt die Rekursion.

Daraus folgt auch direkt, dass  $T_n(x)$  ein Polynom ist, da  $T_n(x)$  nach der Rekursion nur aus Polynomen besteht.

Die anderen Punkte ergeben sich durch einfaches Nachrechnen. □

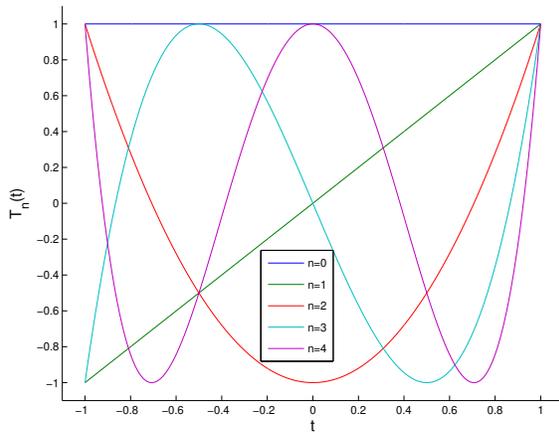


Abb. 8.4.4.  $T_0, \dots, T_4$

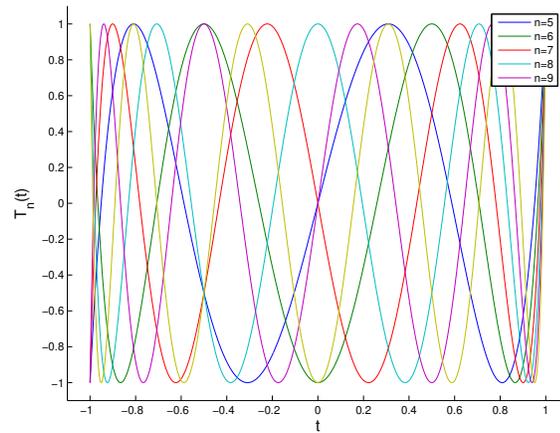


Abb. 8.4.5.  $T_5, \dots, T_9$

**Definition 8.4.6.** (Chebyshev-Knoten)

Die  $n + 1$  Chebyshev-Knoten  $x_0, \dots, x_n$  im Intervall  $[-1, 1]$  sind die Nullstellen von  $T_{n+1}(x)$ .

**Bemerkung 8.4.7.** Für ein beliebiges Intervall  $[a, b]$  sind die Chebyshev-Knoten gegeben durch:

$$x_k = a + \frac{1}{2}(b - a) \left( \cos \left( \frac{2k + 1}{2(n + 1)} \pi \right) + 1 \right) \quad k = 0, \dots, n. \quad (8.4.19)$$

**Definition 8.4.8.** (Chebyshev-Abszissen)

Die  $n - 1$  Chebyshev-Abszissen  $x_0, \dots, x_{n-2}$  im Intervall  $[-1, 1]$  sind die Extrema des Chebyshev-Polynoms  $T_n(x)$ . Gleichzeitig sind sie jedoch auch die Nullstellen von  $U_{n-1}(x)$ . Man nimmt je nach Kontext noch die Punkte  $-1$  und  $1$  hinzu und erhält dann  $n + 1$  Chebyshev-Abszissen.

**Bemerkung 8.4.9.** Für ein beliebiges Intervall  $[a, b]$  sind die Chebyshev-Abszissen gegeben durch:

$$x_k = a + \frac{1}{2}(b - a) \left( \cos \left( \frac{k}{n} \pi \right) + 1 \right) \quad k = 0, \dots, n. \quad (8.4.20)$$

Schliesst man die Endpunkte  $a$  und  $b$  aus, so ist  $k = 1, \dots, n - 1$ .

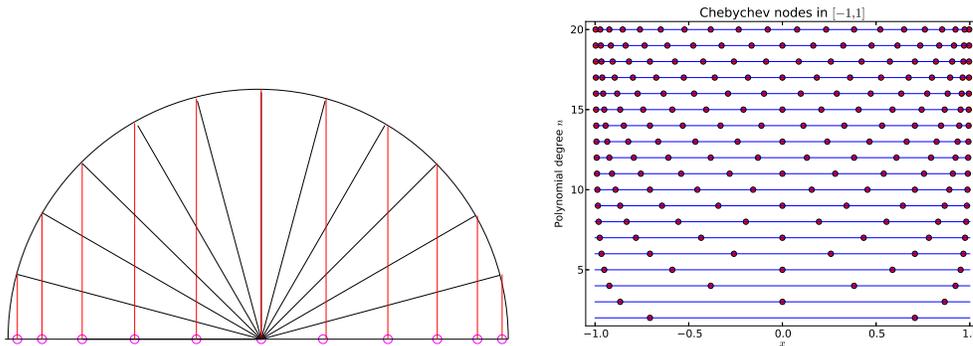


Abb. 8.4.10. Chebyshev-Knoten: links für  $n = 11$ , rechts für verschiedene  $n$

**Bemerkung 8.4.11.** In der Theorem 8.4.10 sehen wir, dass die Chebyshev-Knoten zu den Rändern des Intervalls dichter werden.

Zusätzlich zu diesen Eigenschaften haben die Chebyshev-Polynome eine weitere wichtige Eigenschaft im Raum das Polynome:

**Theorem 8.4.12.** (Orthogonalität I)

Die Chebyshev-Polynome sind orthogonal bezüglich des Skalarprodukts

$$\langle f, g \rangle = \int_{-1}^1 f(x)g(x) \frac{1}{\sqrt{1-x^2}} dx. \quad (8.4.21)$$

*Proof.*

$$\begin{aligned} \langle T_k, T_j \rangle &= \int_{-1}^1 \frac{1}{\sqrt{1-x^2}} T_k(x) T_j(x) dx \\ &\quad \text{Substitution } x = \cos(\phi) \Rightarrow dx = -\sin(\phi) d\phi = -\sqrt{1-x^2} d\phi \\ &= -\int_0^\pi \cos(k\phi) \cos(j\phi) d\phi \\ &= \int_0^\pi \frac{1}{2} [\cos((j-k)\phi) + \cos((j+k)\phi)] d\phi \\ &= \begin{cases} 0 & \text{für } j \neq k \\ \frac{\pi}{2} & \text{für } j = k \neq 0 \\ \pi & \text{für } j = k = 0. \end{cases} \end{aligned}$$

□

**Theorem 8.4.13.** (Orthogonalität II)

Die Chebyshev-Polynome  $T_0, T_1, \dots, T_n$  sind orthogonal bezüglich des diskreten Skalarprodukts im Raum das Polynome von Grad  $\leq n$ :

$$(f, g) = \sum_{l=0}^n f(x_l)g(x_l), \quad (8.4.22)$$

wobei  $(x_0, x_1, \dots, x_n)$  die Nullstellen von  $T_{n+1}$  sind.

*Proof.*

$$\begin{aligned}
 & \cos\left(k\frac{2l+1}{n+1}\frac{\pi}{2}\right) = \cos\left(k\left(l+\frac{1}{2}\right)h\right) \text{ mit } h = \frac{\pi}{n+1} \\
 (T_k, T_j) &= \sum_{l=0}^n \cos\left(k\left(l+\frac{1}{2}\right)h\right) \cos\left(j\left(l+\frac{1}{2}\right)h\right) \\
 &= \frac{1}{2} \sum_{l=0}^n \left[ \cos\left((k-j)\left(l+\frac{1}{2}\right)h\right) + \cos\left((k+j)\left(l+\frac{1}{2}\right)h\right) \right] \\
 &= \frac{1}{2} \operatorname{Re} \left[ \sum_{l=0}^n e^{i(k-j)(l+\frac{1}{2})h} + e^{i(k+j)(l+\frac{1}{2})h} \right] \\
 &= \begin{cases} 0 & \text{für } k \neq j \\ \frac{1}{2}(n+1) & \text{für } k = j \neq 0 \\ n+1 & \text{für } k = j = 0. \end{cases}
 \end{aligned}$$

□

### 8.4.1 Fehlerbetrachtung

Wir wollen nun untersuchen, welche Vorteile wir durch die Verteilung der Stützstellen erhalten haben. Dazu betrachten wir nochmals die Fehlerabschätzung, die wir im letzten Kapitel kennengelernt haben. Der Fehler war gegeben durch:

$$f(x) - p(x) = (x - x_0) \cdots (x - x_n) \frac{f^{(n+1)}(\xi)}{(n+1)!}.$$

**Theorem 8.4.14.** (Fehlerabschätzung)

Unter allen  $(x_0, \dots, x_n)$  mit  $x_i \in \mathbb{R}$  wird

$$\max_{x \in [-1, 1]} |(x - x_0) \cdots (x - x_n)| \quad \text{minimal für} \quad x_k = \cos\left(\frac{2k+1}{n+1} \frac{\pi}{2}\right),$$

$x_k$  sind Nullstellen von  $T_{n+1}$ .

**Bemerkung 8.4.15.** Die Nullstellen der Chebyshev-Polynome  $T_n$  (d.h. Chebyshev-Knoten) sind also die bestmögliche Wahl für die Stützstellen. Die Extrema des Chebyshev-Polynoms  $T_{2n}$  (d.h. Chebyshev-Abszissen) beinhalten die Nullstellen von  $T_n$ . Zwischen jede zwei nebeneinander liegenden Chebyshev-Abszissen gibt es eine Nullstelle von  $T_{2n}$ . Aufgrund dieser Eigenschaften und der einfacheren Verbindung in der FFT werden eher die Chebyshev-Abszissen statt der Chebyshev-Knoten für praktische Berechnungen verwendet.

**Theorem 8.4.16.** (Lebesgue-Konstante)

Die Lebesgue-Konstante  $\Lambda_n$  für die Chebyshev-Interpolation ist gegeben durch

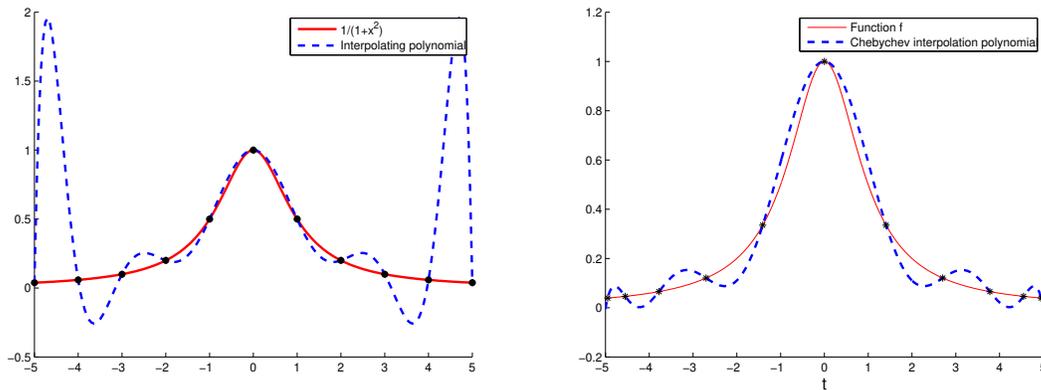
$$\Lambda_n \approx \frac{2}{\pi} \log n \quad \text{für } n \rightarrow \infty.$$

**Beispiel 8.4.17.** (Runge-Funktion)

Zur Veranschaulichung wollen wir die Runge-Funktion im Intervall  $[-5, 5]$  nochmals betrachten. Zur Erinnerung, die Runge-Funktion war gegeben durch:

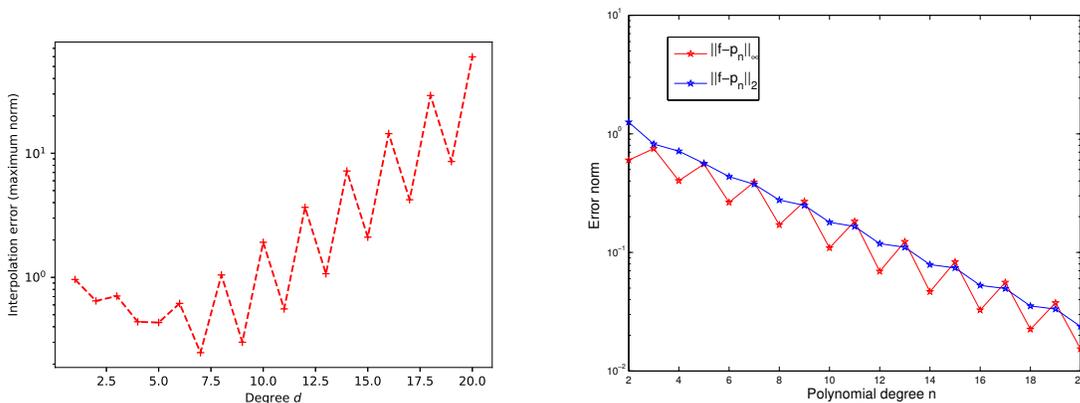
$$f(x) = \frac{1}{x^2 + 1}.$$

Diesmal wählen wir die Chebyshev-Knoten zur Interpolation. Zum Vergleich machen wir dieselbe Interpolation mit einer äquidistanten Verteilung der Stützstellen. Die beiden Interpolationspolynome sind in Theorem 8.4.18, bzw. Theorem 8.4.19 dargestellt.



**Abb. 8.4.18.** Äquidistante Stützstellen **Abb. 8.4.19.** Chebyshev-Knoten

Der Interpolationsfehler der Chebyshev-Interpolation ist eindeutig geringer. Aber wie verhält sich der Fehler als Funktion des Grades der Interpolationsfunktion im Falle der Chebyshev-Interpolation?



**Abb. 8.4.20.** Äquidistante Stützstellen **Abb. 8.4.21.** Chebyshev-Knoten

Aus Theorem 8.4.21 wird deutlich, dass wir einen geringeren Fehler bei der Chebyshev-Interpolation erhalten, wenn wir den Grad des Interpolationspolynoms erhöhen. Die äquidistante Verteilung der Stützstellen hingegen verursacht immer stärker werdenden Oszillationen an den Rändern des Intervalls und einen grösser werdenden Fehler (Theorem 8.4.20).

### 8.4.2 Interpolation und Auswertung

Neben den schon kennengelernten nützlichen Eigenschaften wollen wir uns nun mit der praktischen Anwendung der Chebyshev-Interpolation beschäftigen.

**Theorem 8.4.22.** (Interpolationspolynom)

Das Interpolationspolynom  $p$  zu  $f$  mit den Chebyshev-Knoten  $x_0, x_1, \dots, x_n$ , also den Nullstellen von  $T_{n+1}$ , ist gegeben durch:

$$p(x) = \frac{1}{2}c_0 + c_1T_1(x) + \dots + c_nT_n(x), \quad (8.4.23)$$

wobei für die  $c_i$  gilt:

$$c_k = \frac{2}{n+1} \sum_{l=0}^n f \left( \underbrace{\cos \left( \frac{2l+1}{n+1} \frac{\pi}{2} \right)}_{=x_i(\text{Knoten})} \right) \cos \left( k \frac{2l+1}{n+1} \frac{\pi}{2} \right). \quad (8.4.24)$$

**Bemerkung 8.4.23.** Für grosse  $n$  ( $n \geq 15$ ) berechnet man die  $c_k$  mithilfe der Fast Fourier Transformation (FFT). Diese wird im nächsten Kapitel behandelt. Damit ergibt sich für den Aufwand zur Berechnung des Interpolationspolynoms:

Direkte Berechnung der $c_k$	$(n+1)^2$ Operationen
Dividierte Differenzen	$\frac{n(n+1)}{2}$ Operationen (zum Vergleich)
$c_k$ mittel FFT	$n \log n$ Operationen.

Auch die Auswertung des Interpolationspolynomes an beliebigen Punkten gestaltet sich durch den sogenannten Clenshaw-Algorithmus sehr einfach:

**Theorem 8.4.24.** (Clenshaw Algorithmus)

Das Interpolationspolynom sei, wie oben, gegeben durch

$$p(x) = \frac{1}{2}c_0 + c_1T_1(x) + \dots + c_nT_n(x).$$

Seien  $d_{n+2} = d_{n+1} = 0$ . Setze nun:

$$d_k = c_k + (2x)d_{k+1} - d_{k+2} \quad \text{für } k = n, n-1, \dots, 0. \quad (8.4.25)$$

Dann gilt:  $p(x) = \frac{1}{2}(d_0 - d_2)$  ( $p(x)$  ist also über eine Rückwärtsrekursion gegeben.

*Proof.* Wie wissen bereits, dass gilt  $T_{k+1}(x) = 2xT_k(x) - T_{k-1}$ , also folgt mit mehrmaligem Einsetzen:

$$\begin{aligned} p(x) &= \frac{1}{2}c_0 + c_1T_1(x) + \dots + \underbrace{c_n}_{=d_n} T_n(x) \\ &= \frac{1}{2}c_0 + c_1T_1(x) + \dots + (c_{n-2} - d_n)T_{n-2}(x) + \underbrace{(c_{n-1} + 2xd_n)}_{=d_{n-1}} T_{n-1}(x) \\ &= \frac{1}{2}c_0 + c_1T_1(x) + \dots + (c_{n-3} - d_{n-1})T_{n-3}(x) + \underbrace{(c_{n-2} + 2xd_{n-1} - d_n)}_{=d_{n-2}} T_{n-2}(x) \\ &= \dots = \frac{1}{2}c_0 + xd_1 - d_2 = \frac{1}{2} \underbrace{(c_0 + 2xd_1 - d_2)}_{=d_0} - d_2 \\ &= \frac{1}{2}(d_0 - d_2). \end{aligned}$$

□

## Code 8.4.25: Clenshaw Algorithmus

```

1  def clenshaw(a, x):
2      # Grad n des Polynoms
3      n = a.shape[0] - 1
4      d = tile( reshape(a,n+1,1), (x.shape[0], 1) )
5      d = d.T
6      for j in range(n, 1, -1):
7          d[j-1,:] = d[j-1,:] + 2.0*x*d[j,:]
8          d[j-2,:] = d[j-2,:] - d[j,:]
9      y = d[0,:] + x*d[1,:]
10     return y

```

**Bemerkung 8.4.26.** Der Clenshaw-Algorithmus ist sehr stabil. Das dies nicht immer der Fall sein muss zeigt die folgende instabile Rekursion:

$$x_{n+1} = 10x_n - 9.$$

Wenn  $x_0 = 1$ , dann ist  $x_n = 1$  für alle  $n$ . Bei einem Mess- oder Rundungsfehler am Anfang der Rekursion

$$x_0 = 1 + \varepsilon$$

erhalten wir nach  $n$  Schritte

$$x_n = 1 + 10^n \varepsilon.$$

Mit andern Worten, der relative Fehler wird in jedem Schritt mit 10 multipliziert, sodass er nach  $n$  Schritten  $10^n$  grösser ist.

Showing a Fourier Transform to a physics student generally produces the same reaction as showing a crucifix to Count Dracula.

---

J.F. James - "A Student's Guide to Fourier Transforms"

## Chapter 9

# Trigonometrische Interpolation

### 9.1 Motivation

Wir haben bereits gesehen, dass man Funktionen durch Polynome und die Taylorentwicklung approximieren kann. Allerdings gibt es auch weitere und bessere Möglichkeiten.

Der französische Mathematiker und Physiker Fourier<sup>1</sup> machte im Jahre 1822 mit seinem Werk "Théorie analytique de la chaleur" die Idee, Funktionen durch trigonometrische Polynome zu approximieren, bekannt. Er benutzte (bereits in 1804-1807) Reihen von trigonometrischen Funktionen, um die Wärmeleitungsgleichung zu lösen. Weiterhin behauptete er, dass es für alle Funktionen eine solche Entwicklung nach trigonometrischen Funktionen gibt. Die Behauptung, jede Funktion durch eine periodische und unendlich glatte Funktion zu approximieren, war damals dermaßen verblüffend, dass Lagrange, Laplace, Biot und Poisson dieser sehr kritisch am Anfang gegenüber waren. Die Mathematik war damals einfach noch nicht so weit entwickelt, um diese Ideen mit der Eleganz von heute zu erklären. Tatsache ist, dass wichtige wissenschaftliche und technologische Fortschritte auf diese einfache Idee entstanden sind. Hier werden wir einen eher intuitiven Zugang wählen und auch nur Teile dieser gigantischen Theorie betrachten. Mathematische Sätze, die wir verwenden werden, sind zum Beispiel in *Königsberger, Analysis I, Kapitel 16* und *Amann, Escher, Analysis II, Kapitel VI.7* zu finden.

**Definition 9.1.1.** Wir definieren als *trigonometrisches Polynom vom Grad  $\leq 2m$*  die Funktion

$$p_{2m}(t) := t \rightarrow \sum_{j=-m}^m \gamma_j e^{2\pi i j t} \text{ wobei } \gamma_j \in \mathbb{C} \text{ und } t \in \mathbb{R}. \quad (9.1.1)$$

**Bemerkung 9.1.2.** Die Funktion  $p_{2m} : \mathbb{R} \rightarrow \mathbb{C}$  ist periodisch mit Periode 1. Wenn  $\gamma_{-j} = \overline{\gamma_j}$  für alle  $j$ , dann nimmt  $p_{2m}$  nur reelle Werte an und man kann  $p_{2m}$

---

<sup>1</sup>Jean Baptiste Joseph Fourier (1768-1830)

auch folgendermassen darstellen:

$$p_{2m}(t) = \frac{a_0}{2} + \sum_{j=1}^m (a_j \cos(2\pi jt) + b_j \sin(2\pi jt)) \quad (9.1.2)$$

mit  $a_0 = 2\gamma_0$ ,  $a_j = 2\operatorname{Re}(\gamma_j)$  und  $b_j = -2\operatorname{Im}(\gamma_j)$ .

Trigonometrische Polynome wurden in der Vorlesung Physik II unter den Namen "Überlagerung von Schwingungen" angetroffen.

**Definition 9.1.3.** Wir definieren die  $L^2$ -Funktionen auf dem Intervall  $(0, 1)$  als

$$L^2(0, 1) := \{f : (0, 1) \rightarrow \mathbb{C} : \|f\|_{L^2(0,1)} < \infty\}. \quad (9.1.3)$$

Die  $L^2$  - Norm auf  $(0, 1)$  wird durch das Skalarprodukt

$$\langle g, f \rangle_{L^2(0,1)} := \int_0^1 \overline{g(x)} f(x) dx \quad (9.1.4)$$

über  $\|f\|_{L^2(0,1)} = \sqrt{\langle f, f \rangle_{L^2(0,1)}}$  induziert.

**Bemerkung 9.1.4.**  $L^2(a, b)$  lässt sich analog definieren mit  $\langle g, f \rangle_{L^2(a,b)} := \int_a^b \overline{g(x)} f(x) dx = (b-a) \int_0^1 \overline{g(a+(b-a)t)} f(a+(b-a)t) dt$ .

**Bemerkung 9.1.5.**  $L^2(0, 1)$  ist ein linearer Raum mit Skalarprodukt  $\langle f, g \rangle_{L^2(0,1)}$ . Er enthält auch Funktionen, die nur stückweise stetig sind und auch welche, die deutlich unglattere Funktionen sind.

**Bemerkung 9.1.6.** Die Funktionen  $\varphi_k(x) = \exp(2\pi i k x)$  sind orthogonal bezüglich dem  $L^2(0, 1)$ -Skalarprodukt

$$\langle \varphi_k, \varphi_j \rangle_{L^2(0,1)} = \begin{cases} 0, & \text{wenn } k \neq j \\ 1, & \text{wenn } k = j \end{cases}$$

und bilden somit eine Basis für den Unterraum der trigonometrischen Polynome.

**Definition 9.1.7.** Eine Funktion  $f$  ist der  $L^2$  - Grenzwert einer Folge von Funktion  $f_n \in L^2(0, 1)$ , wenn für  $n \rightarrow \infty$  gilt, dass  $\|f - f_n\|_{L^2(0,1)} \rightarrow 0$  geht.

**Bemerkung 9.1.8.** Dieser Konvergenzbegriff ergänzt entsprechende Begriffe, die bereits aus Analysis I bekannt sind. Es geht um eine schwächere Form der Konvergenz als die punktweise und die gleichmässige Konvergenz, da es hier nur um eine Art Mittelwert geht.

**Theorem 9.1.9.** Jede Funktion  $f \in L^2(0, 1)$  ist der  $L^2$ -Grenzwert ihrer Fourier-Reihe:

$$f(t) = \sum_{k=-\infty}^{\infty} \widehat{f}(k) e^{2\pi i k t}, \quad (9.1.5)$$

wobei die Fourier-Koeffizienten  $\widehat{f}(k)$  als

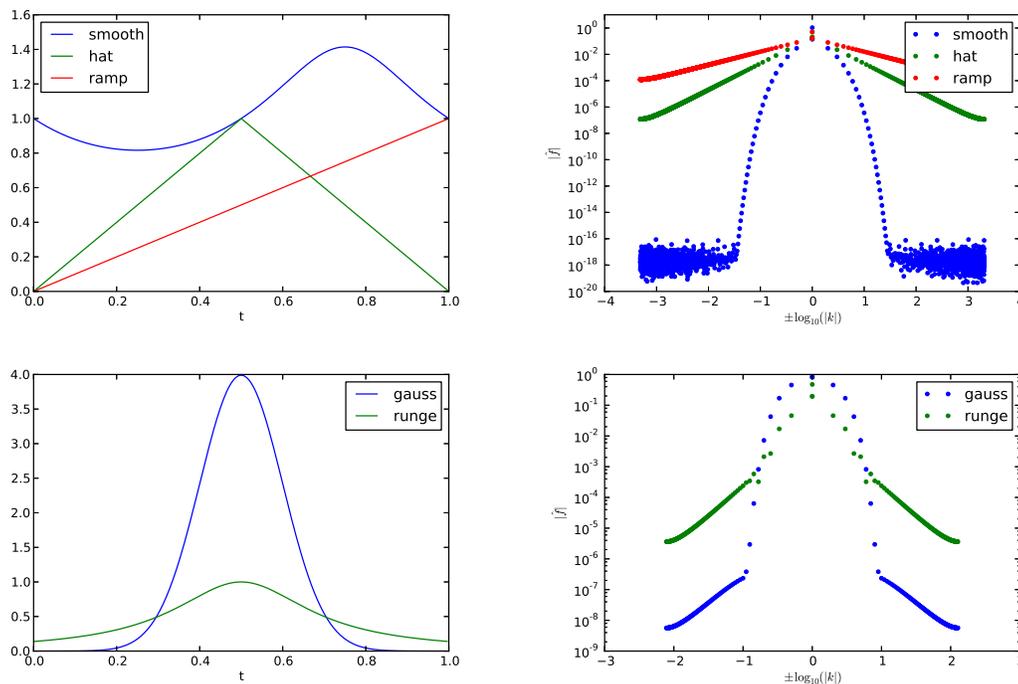
$$\widehat{f}(k) = \int_0^1 f(t)e^{-2\pi ikt} dt, \quad k \in \mathbb{Z}. \quad (9.1.6)$$

definiert sind. Dazu gilt die Parseval'sche Gleichung:

$$\sum_{k=-\infty}^{\infty} |\widehat{f}(k)|^2 = \|f\|_{L^2(0,1)}^2. \quad (9.1.7)$$

**Bemerkung 9.1.10.** Wir können dieses Theorem auch kürzer formulieren: Die Funktionen  $\varphi_k(x) = \exp(2\pi i k x)$  bilden eine **vollständige orthonormale Basis** in  $L^2(0, 1)$ .

**Bemerkung 9.1.11.** Dieser Satz rechtfertigt, dass man sich eine Funktion entweder im Zeitraum  $t \rightarrow f(t)$  oder im Frequenzraum  $k \rightarrow \widehat{f}(k)$  vorstellen kann. Es ist eventuell sinnvoll zwischen den Darstellungen zu wechseln, wenn man in einer Darstellung gewisse Eigenschaften sehr einfach ablesen kann. Dafür gibt es unzählige Anwendungen: NMR, IR-Spektroskopie, etc.



**Abb. 9.1.12.** Darstellungen im Zeitraum (links) und im Frequenzraum (rechts)

**Bemerkung 9.1.13.** Für Funktionen mit reellem Wertebereich können wir diese

Ergebnisse auch noch anders darstellen:

$$f(t) = \frac{a_0}{2} + \sum_{k=1}^{\infty} (a_k \cos(2\pi kt) + b_k \sin(2\pi kt))$$

$$a_k = \int_0^1 f(t) \cos(2\pi kt) dt$$

$$b_k = \int_0^1 f(t) \sin(2\pi kt) dt.$$

**Bemerkung 9.1.14.** Die Parseval'sche Gleichung sagt zum einen aus, dass die Fourierkoeffizienten für  $L^2(0, 1)$  schneller als  $\frac{1}{n}$  abklingen, da die rechte Seite einen definierten Wert und die linke Reihe damit konvergieren muss. Zum anderen sagt sie, dass die  $L^2$ -Norm der Funktion (die manche als "Energie" benennen) statt aus einem Integral aus einer Summe (der Fourier-Koeffizienten im Betragsquadrat) berechnet werden kann. In Analysis wird dies verwendet um gewisse Reihen zu summieren.

Leiten wir (9.1.5) formal nach  $t$  ab, erhalten wir

$$f'(t) = \sum_{k=-\infty}^{\infty} 2\pi ik \widehat{f}(k) e^{2\pi ikt}.$$

**Theorem 9.1.15.** Sei  $f$  und  $f'$  integrierbar auf  $(0, 1)$ , dann gilt  $\widehat{f'}(k) = 2\pi ik \widehat{f}(k)$  für  $k \in \mathbb{Z}$ .

Wenn die Operationen erlaubt sind, dann:

$$\widehat{f^{(n)}}(k) = (2\pi ik)^n \widehat{f}(k) \text{ und } \|f^{(n)}\|_{L^2}^2 = (2\pi)^{2n} \sum_{k=-\infty}^{\infty} k^{2n} |\widehat{f}(k)|^2,$$

was eine direkte Verbindung zwischen Glattheit einer Funktion und der Abfall ihrer Fourier-Koeffizienten zeigt: je glatter die Funktion, desto schneller ist der Abfall der Fourier-Koeffizienten. Diese Tatsache wird verwendet um die Glattheit über das Konvergenzverhalten der Fourier-Reihen zu definieren (was wir hier nicht tun).

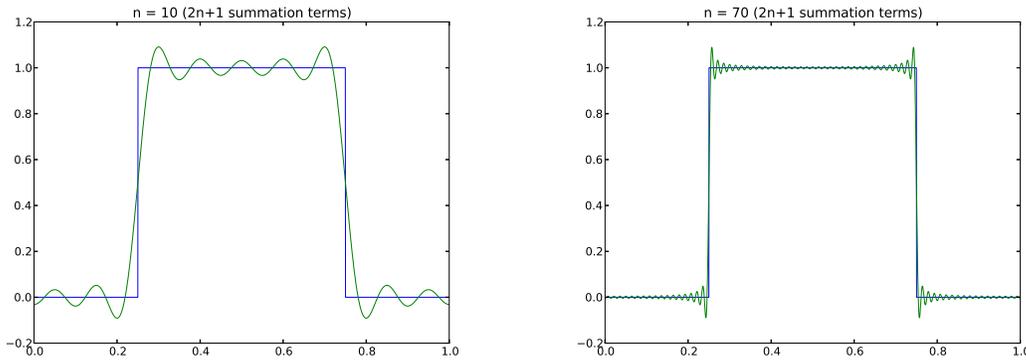
**Theorem 9.1.16.** Wenn  $\int_0^1 |f^{(n)}(t)| dt < \infty$ , dann  $\widehat{f}(k) = \mathcal{O}(k^{-n})$ .

Interessant ist auch, was passiert, wenn die Funktion eben nicht so glatt ist. Was macht die Fourier Reihe an einer Sprungstelle der Funktion? Dort kommt es zu Überschwingungen, die immer näher an die Sprünge kommen, aber die nicht kleiner werden, wenn wir mehr Terme in der Fourier-Reihe aufsummieren: das ist das Gibbs-Phänomen; in der Sprache der Konvergenz: wir haben  $L^2$ -Konvergenz aber keine punktweise Konvergenz an der Sprungstelle.

**Beispiel 9.1.17.** Die Fourier-Reihe der Charakteristischen Funktion des Intervalls  $[a, b] \subset ]0, 1[$  lässt sich analytisch berechnen:

$$b - a + \frac{1}{\pi} \sum_{|k|=1}^{\infty} e^{-ikc} \frac{\sin kd}{k} e^{i2\pi kt}, \quad t \in [0, 1],$$

mit  $c = \pi(a + b)$  und  $d = \pi(b - a)$ . Die Fourier-Koeffizienten fallen sehr langsam ab, so dass wir eine sehr langsame Konvergenz der Fourier-Reihe erwarten. In den Bildern beobachten wir auch das Gibbs-Phänomen; wir haben nur  $L^2$ -Konvergenz der Fourier-Reihe.



**Bemerkung 9.1.18.** Normalerweise können wir die Fourier-Koeffizienten nicht analytisch ausrechnen; stattdessen verwenden wir die Funktionswerte an den Knoten  $x_\ell = \frac{\ell}{N}$  für  $\ell = 0, 1, \dots, N$  und die Trapezregel um die exakten Fourier-Koeffizienten numerisch zu approximieren:

$$\hat{f}(k) \approx \frac{1}{N} \sum_{\ell=0}^{N-1} f(x_\ell) e^{-2\pi i k x_\ell} =: \hat{f}_N(k), \quad (9.1.8)$$

wobei wir hier  $f(1) = f(0)$  vorausgesetzt haben.

Bevor wir zur diskreten Fourier-Transformation kommen, lassen Sie uns die wichtigsten Ideen dieser Sektion zusammenfassen.

Das Ziel unserer Untersuchungen sind (eventuell unglatte) Funktionen:

$$f : (0, 1) \longrightarrow \mathbb{C}, \text{ mit } \int_0^1 |f(x)|^2 dx < \infty.$$

Für solche Funktionen definieren wir die (kontinuierlichen) Fourier-Koeffizienten:

$$\hat{f}(k) = \int_0^1 f(x) e^{-2\pi i k x} dx, \text{ für } k = 0, \pm 1, \pm 2, \dots$$

Dann bauen wir die Summen

$$p_{2m}(x) = \sum_{k=-m}^m \hat{f}(k) e^{2\pi i k x}$$

und wir lassen  $m \rightarrow \infty$ . Fourier sagt:

$$\lim_{m \rightarrow \infty} \|p_{2m}(\cdot) - f(\cdot)\|_2 = \lim_{m \rightarrow \infty} \sqrt{\int_0^1 |p_{2m}(x) - f(x)|^2 dx} = 0$$

und Parseval sagt:

$$\|f\|_2^2 = \int_0^1 |f(x)|^2 dx = \lim_{m \rightarrow \infty} \sum_{k=-m}^m |\widehat{f}(k)|^2 = \|\widehat{f}\|_2^2,$$

wobei  $\widehat{f} = \left(\widehat{f}(k)\right)_k$  als eine Folge mit entsprechender Norm interpretiert wird. Die Konvergenz der Reihe  $\sum_{k=-m}^m |\widehat{f}(k)|^2$  steht in direktem Zusammenhang mit dem Abfall seiner Mitglieder  $|\widehat{f}(k)|$ . Es stellt sich heraus: je glatter  $f$ , desto schneller ist der Abfall seiner Fourier-Koeffizienten  $|\widehat{f}(k)|$ , also desto schneller ist die Konvergenz der Fourier-Reihe. Für die Approximation bedeutet das, dass glatte Funktionen sehr gut durch kurze Fourier-Summen sich approximieren lassen, während unglatte Funktionen einfach längere Fourier-Summen brauchen. Anwendungen dafür befinden sich in der Kompression, Rauschunterdrückung, Autofokus, etc. Z.B. Rauschen ist unglatt und entspricht also grossen Fourier-Koeffizienten  $|\widehat{f}(k)|$  für grosse  $k$ .

## 9.2 Diskrete Fouriertransformation

**Definition 9.2.1.** Die  $N$ -te Einheitswurzel ist  $\omega_N := \exp\left(\frac{-2\pi i}{N}\right)$ .

**Bemerkung 9.2.2.** Wir finden durch Nachrechnen und Ausnutzen der Eigenschaften der komplexen Exponentialfunktion:

$$\omega_N^{k+N} = \omega_N^k, \text{ für alle } k \in \mathbb{Z}, \quad (9.2.9)$$

$$\omega_N^{t+kN} = \omega_N^t, \text{ für alle } k \in \mathbb{Z}, \text{ und } t \in \mathbb{R} \quad (9.2.10)$$

$$\omega_N^N = 1, \quad (9.2.11)$$

$$\omega_N^{N/2} = -1, \quad (9.2.12)$$

$$\sum_{k=0}^{N-1} \omega_N^{kj} = \begin{cases} N, & \text{wenn } j = 0 \\ 0, & \text{wenn } j \neq 0. \end{cases} \quad (9.2.13)$$

In der Tat, wir erinnern uns:

$$\exp(z) = e^{\Re z} (\cos \Im z + i \sin \Im z),$$

und

$$\omega_N = \cos \frac{2\pi}{N} - i \sin \frac{2\pi}{N}.$$

Die Potenzfunktion ist definiert als

$$z^\alpha = \exp(\alpha \ln z),$$

also

$$\omega_N^\alpha = \exp(\alpha \ln \omega_N) = \exp\left(-\frac{2\pi i}{N} \alpha\right) = \cos \frac{2\pi \alpha}{N} - i \sin \frac{2\pi \alpha}{N}.$$

Wir erhalten dann

$$\omega_N^{t+kN} = \exp\left(-\frac{2\pi i}{N}(t+kN)\right) = \cos \frac{2\pi t + 2\pi kN}{N} - i \sin \frac{2\pi t + 2\pi kN}{N} = \omega_N^t.$$

Sei  $w_N = \exp(\frac{2\pi i}{N}) = \overline{\omega_N}$  und

$$\mathbf{v}_k = \begin{bmatrix} w_N^{0 \cdot k} \\ w_N^{1 \cdot k} \\ w_N^{2 \cdot k} \\ \vdots \\ w_N^{(N-1) \cdot k} \end{bmatrix}$$

**Definition 9.2.3.** Wir definieren als *trigonometrische Basis* die Vektoren

$$\mathbf{v}_0 = \begin{bmatrix} w_N^{0 \cdot 0} \\ w_N^{1 \cdot 0} \\ w_N^{2 \cdot 0} \\ \vdots \\ w_N^{(N-1) \cdot 0} \end{bmatrix}, \dots, \mathbf{v}_k = \begin{bmatrix} w_N^{0 \cdot k} \\ w_N^{1 \cdot k} \\ w_N^{2 \cdot k} \\ \vdots \\ w_N^{(N-1) \cdot k} \end{bmatrix}, \dots, \mathbf{v}_{N-1} = \begin{bmatrix} w_N^{0 \cdot (N-1)} \\ w_N^{1 \cdot (N-1)} \\ w_N^{2 \cdot (N-1)} \\ \vdots \\ w_N^{(N-1) \cdot (N-1)} \end{bmatrix}. \quad (9.2.14)$$

Die vorigen Eigenschaften zeigen, dass es um paarweise orthogonale Vektoren in  $\mathbb{C}^N$  handelt. Mit der Notation

$$\mathbf{V} = [\mathbf{v}_0 \dots \mathbf{v}_{N-1}]$$

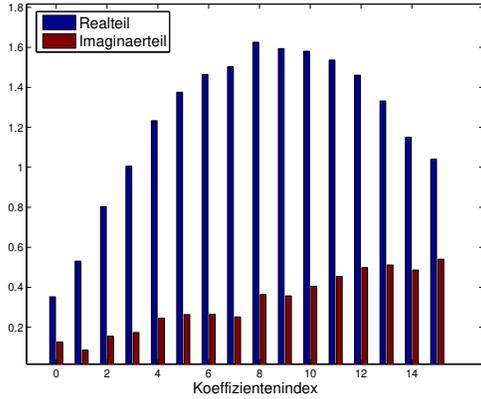
berechnet man leicht

$$\mathbf{V}^H \mathbf{V} = N \mathbf{I}_N.$$

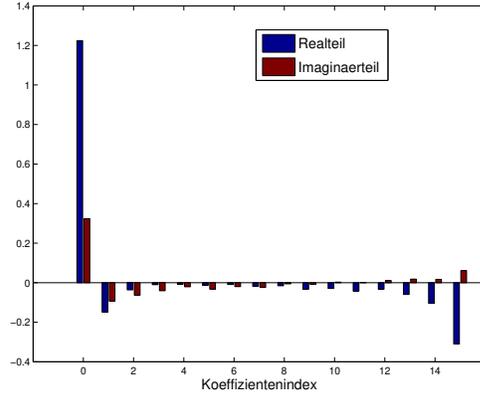
$\mathbf{V}$  ist eine symmetrische, nicht Hermite-symmetrische  $N$  mal  $N$  Matrix mit komplexe Einträge. Die Basiswechsellmatrix von der Standardbasis zu der trigonometrischen Basis ist  $\mathbf{V}$ , denn  $\mathbf{v}_k = \mathbf{V} \mathbf{e}_k$ . Für den Koordinatenwechseln haben wir also

$$\mathbf{y} = \sum_{k=0}^{N-1} y_k \mathbf{e}_k = \sum_{k=0}^{N-1} z_k \mathbf{v}_k = \sum_{k=0}^{N-1} z_k \begin{bmatrix} w_N^{0 \cdot k} \\ w_N^{1 \cdot k} \\ w_N^{2 \cdot k} \\ \vdots \\ w_N^{(N-1) \cdot k} \end{bmatrix}.$$

d.h.  $\mathbf{y} = \mathbf{V} \mathbf{z}$  und  $\mathbf{z} = \mathbf{V}^{-1} \mathbf{y} = \frac{1}{N} \mathbf{V}^H \mathbf{y} = \frac{1}{N} \mathbf{F}_N \mathbf{y}$  mit  $\mathbf{F}_N = \mathbf{V}^H$ . Damit können wir einen Vektor aus der trigonometrischen Basis durch Matrixmultiplikation mit  $\mathbf{V}$  in der Standardbasis darstellen. Umgekehrt müssen wir mit der Inversen von  $\mathbf{V}$  multiplizieren. Eventuell ist die Darstellung eines Vektors in einer der beiden Basen wesentlich einfacher; wir können dann in dieser Basis Rechnungen vornehmen und am Ende transformieren wir zurück.



Vektor in Standardbasis  
Der  $\ell$ -te Eintrag in  $\mathbf{y}$  ist



Vektor in der trigonometrische Basis

$$\begin{aligned}
 y_\ell &= \sum_{k=0}^{N-1} z_k w_N^{\ell \cdot k} = \sum_{k=0}^{N/2-1} z_k w_N^{\ell \cdot k} + \sum_{k=N/2}^{N-1} z_k w_N^{\ell \cdot k} = \sum_{k=0}^{N/2-1} z_k w_N^{\ell \cdot k} + \sum_{k=-N/2}^{-1} z_{N+j} w_N^{\ell \cdot j} \\
 &= \sum_{k=-N/2}^{N/2-1} \gamma_k \exp\left(\frac{2\pi i}{N} \ell k\right),
 \end{aligned}$$

was einem Teil der Fourier-Reihe ausgewertet im Punkt  $\frac{\ell}{N} \in [0, 1[$  entspricht, wobei die (diskreten) Fourier-Koeffizienten durch eine Umsortierung der Koeffizienten in der trigonometrischen Basis gegeben sind:

$$\gamma_k = \begin{cases} z_k, & \text{für } 0 \leq k \leq n-1 = \frac{N}{2} - 1 \\ z_{k+N}, & \text{für } -\frac{N}{2} = -n \leq k < 0 \end{cases}. \quad (9.2.15)$$

**Definition 9.2.4.** Wir definieren die *Fouriermatrix*, als die Basistransformationsmatrix von der trigonometrischen Basis zur Standardbasis (bis auf dem Faktor  $1/N$ ) (Kanonische Basis):

$$\mathbf{F}_N = \mathbf{V}^H = \begin{bmatrix} \omega_N^0 & \omega_N^0 & \dots & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \dots & \omega_N^{N-1} \\ \omega_N^0 & \omega_N^2 & \dots & \omega_N^{2N-2} \\ \vdots & \vdots & & \vdots \\ \omega_N^0 & \omega_N^{N-1} & \dots & \omega_N^{(N-1)^2} \end{bmatrix} = (\omega_N^{ij})_{i,j=0}^{N-1} \in \mathbb{C}^{N \times N}. \quad (9.2.16)$$

**Definition 9.2.5.** Sei  $\mathcal{F}_N : \mathbb{C}^N \rightarrow \mathbb{C}^N$  die lineare Abbildung  $\mathcal{F}_N(\mathbf{y}) := \mathbf{F}_N \mathbf{y}$  für  $\mathbf{y} \in \mathbb{C}^N$ . Wir nennen diese Abbildung *Diskrete Fourier-Transformation*, kurz DFT. Äquivalent dazu ist die Formulierung, dass für  $\mathbf{c} = \mathcal{F}_N(\mathbf{y})$  gilt:

$$c_k = \sum_{j=0}^{N-1} y_j \omega_N^{kj}, \quad k = 0, 1, \dots, N-1. \quad (9.2.17)$$

**Theorem 9.2.6.** Die skalierte Fouriermatrix  $\frac{1}{\sqrt{N}} \mathbf{F}_N$  ist unitär:

$$\mathbf{F}_N^{-1} = \frac{1}{N} \mathbf{F}_N^H = \frac{1}{N} \overline{\mathbf{F}_N}.$$

**Bemerkung 9.2.7.** Es gilt  $\frac{1}{N}\mathbf{F}_N^2 = -I$  und  $\frac{1}{N^2}\mathbf{F}_N^4 = I$ , also die Eigenwerte von  $\frac{1}{\sqrt{N}}\mathbf{F}_N$  sind in  $\{1, -1, i, -i\}$ .

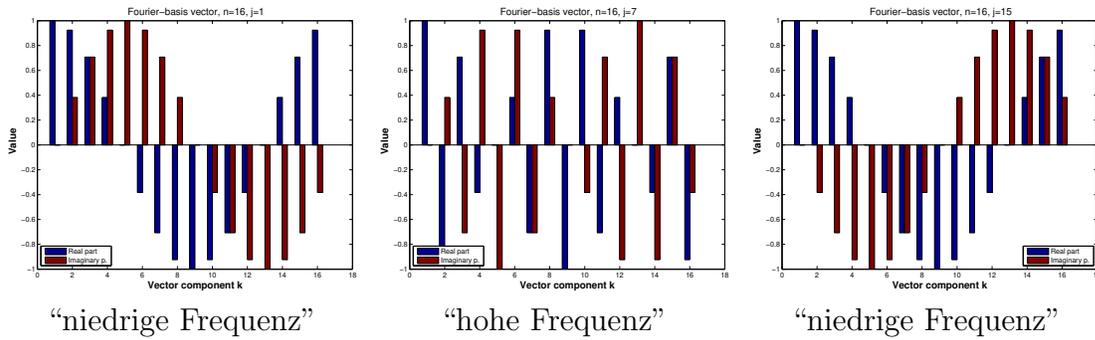
**Bemerkung 9.2.8.** numpy stellt die Funktionen

$$\mathbf{c} = \text{fft}(\mathbf{y}) \text{ und } \mathbf{y} = \text{ifft}(\mathbf{c})$$

für

$$\mathbf{c} = \mathbf{F}_N \mathbf{y} \text{ und } \mathbf{y} = \frac{1}{N} \mathbf{F}_N^H \mathbf{c}$$

zur Verfügung.



**Abb. 9.2.9.** Einige Vektoren der Fourier-Basis ( $n = 16$ )

In der Abbildung 9.2.9 sind einige Vektoren der Fourier-Basis ( $n = 16$ ); Die Real- und Imaginärteile oszillieren weniger für die ersten ( $j = 0, 1, 2$ ) und die letzten ( $j = 13, 14, 15$ ), und viel mehr für die mittleren (um  $j = 7$ ) Vektoren der trigonometrischen Basis. In diesem Sinne stehen die hohen Frequenzen in der Mitte des trigonometrischen Basis im Kontext der diskreten Fourier-Transformation, im Kontrast zur Formel (9.1.1), die ein trigonometrisches Polynom definiert.

**Bemerkung 9.2.10.** Wenn wir eine Darstellung analog zu (9.1.1) wollen, dann müssen wir die Basisfunktionen und die Koeffizienten umsortieren (zum Beispiel mit `fft.fftshift`). Hier ist das Argument und die Umsortierung für eine gerade Anzahl  $N = 2n$  von Basisfunktionen. Für  $k = 0, 1, 2, \dots, N - 1$  gilt:

$$\begin{aligned} y_k &= \sum_{j=-n}^{n-1} \gamma_j \omega_N^{-jk} = \sum_{j=-n}^{-1} \gamma_j \omega_N^{-jk} + \sum_{j=0}^{n-1} \gamma_j \omega_N^{-jk} \\ &= \sum_{p=n}^{2n-1} \gamma_{p-2n} \omega_N^{-(p-N)k} + \sum_{j=0}^{n-1} \gamma_j \omega_N^{-jk} \\ &= \sum_{p=n}^{2n-1} \gamma_{p-2n} \omega_N^{-pk+kN} + \sum_{j=0}^{n-1} \gamma_j \omega_N^{-jk} = \sum_{j=0}^{2n-1} z_j \omega_N^{-jk}, \end{aligned} \quad (9.2.18)$$

wobei

$$z_j = \begin{cases} \gamma_j, & \text{für } 0 \leq j \leq n-1 = \frac{N}{2} - 1 \\ \gamma_{j-N}, & \text{für } n \leq j \leq 2n-1 = N-1 \end{cases}. \quad (9.2.19)$$

Somit erhalten wir  $\mathbf{z} = \frac{1}{N} \mathbf{F}_N \mathbf{y}$  und

$$\gamma_k = \begin{cases} z_k, & \text{für } 0 \leq k \leq n-1 = \frac{N}{2} - 1 \\ z_{k+N}, & \text{für } -\frac{N}{2} = -n \leq k < 0 \end{cases} .$$

Ausgehend von den Werten

$$y_k = f\left(\frac{k}{N}\right), \text{ für } k = 0, 1, \dots, N-1,$$

berechnen wir

$$\mathbf{z} = \frac{1}{N} \mathbf{F}_N \mathbf{y}$$

via `fft.fft` und erhalten laut (9.1.8) eine Approximation von  $f$  mittels trigonometrisches Polynom

$$f(x) \approx \sum_{j=0}^{\frac{N}{2}-1} z_j e^{2\pi i j x} + \sum_{j=\frac{N}{2}}^{N-1} z_j e^{2\pi i (j-N)x},$$

oder

$$f(x) \approx \sum_{k=0}^{\frac{N}{2}-1} z_k e^{2\pi i k x} + \sum_{k=-\frac{N}{2}}^{-1} z_{k+N} e^{2\pi i k x},$$

oder mit  $\zeta = \text{fft.fftshift}(\mathbf{z})$ :

$$f(x) \approx \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \zeta_k e^{2\pi i k x} .$$

**Beispiel 9.2.11.** Wenn wir `fft.fft` auf einen Vektor  $y$  der Länge 8 anwenden, erhalten wir den Vektor  $c[0], \dots, c[7]$ , und `fft.fftshift` stellt diese Einträge in der Reihenfolge  $c[4], c[5], c[6], c[7], c[0], c[1], c[2], c[3]$ , die der  $\gamma_{-4}, \gamma_{-3}, \gamma_2, \gamma_1, \gamma_0, \gamma_1, \gamma_2, \gamma_3$  entsprechen.

**Bemerkung 9.2.12.** Diese Approximation kann für die numerische Berechnung der  $L^2$ -Norm der Funktion  $f$  oder deren Ableitungen verwendet werden, z.B.

$$\|f\|_2^2 \approx \left\| \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} \zeta_k e^{2\pi i k x} \right\|_2^2 = \|z\|_2^2,$$

oder auch für das numerische Lösen von Differentialgleichungen.

**Beispiel 9.2.13** (Frequenzanalyse mittels DFT). Nun nehmen wir ein periodisches Signal mit zwei Frequenzen (1 und 7); dazu addieren wir eine zufällige Störung. Wir transformieren dann dieses gestörte Signal in die Fourier-Basis und wir schauen uns das sogenannte “power spectrum” (die spektrale Leistungsdichte) an. Die ursprünglichen zwei Frequenzen sind eindeutig sichtbar in der Abbildung 9.2.14. In diesem Beispiel sind die Periodizität und die Abtastpunkte des Signals allerdings wichtig; wir schauen uns auch nur die erste Hälfte der Koeffizienten. Probieren Sie auch `bar(t[:]-64/2,fft.fftshift(p[:]))!`

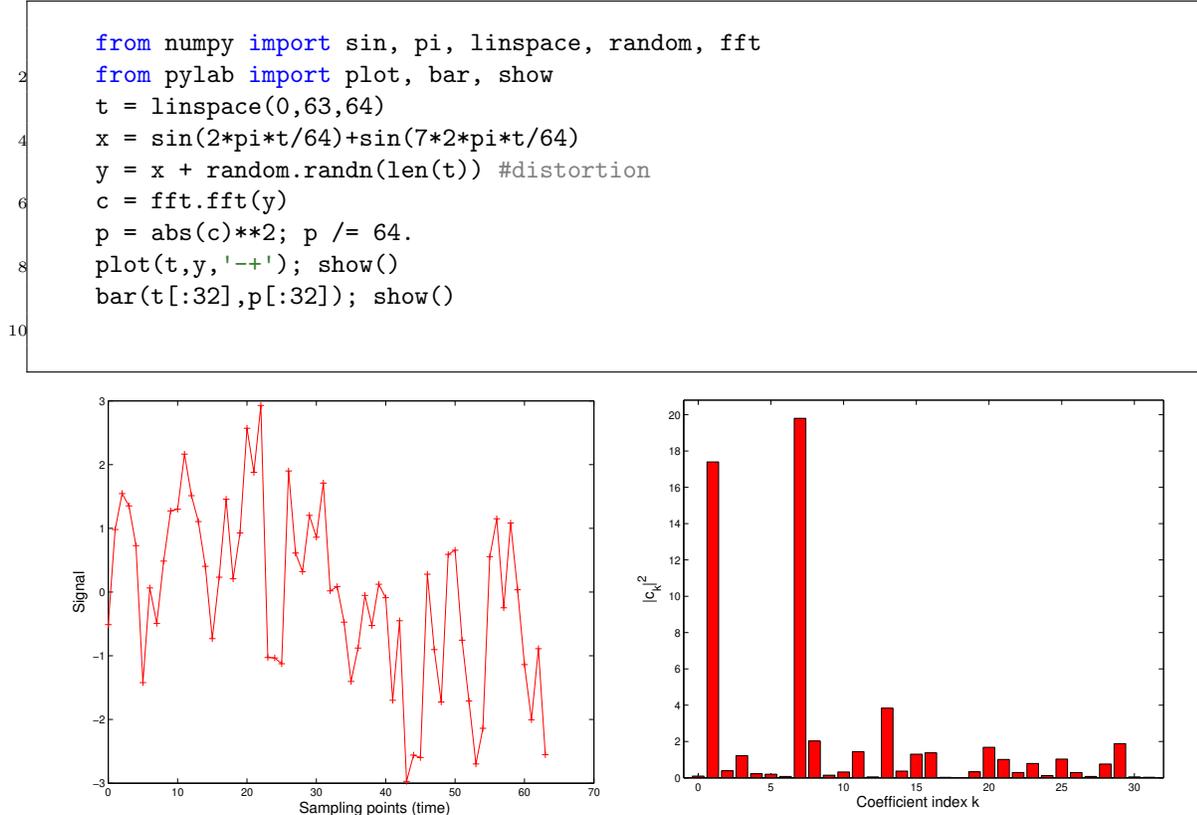


Abb. 9.2.14. Frequenzanalyse mittels DFT

### 9.3 FFT - Schnelle Fouriertransformation

Nach *Science*, "Random Samples", p.799, 4.2.2000 gehört die schnelle Fouriertransformation, kurz FFT, zu den 10 wichtigsten Algorithmen in Naturwissenschaft und Technik im 20. Jahrhundert.

Warum ist dieser Algorithmus aber so wichtig? Auf den ersten Blick scheint es, dass der DFT-Algorithmus für einen Vektor aus dem  $\mathbb{C}^n$  eine Komplexität von  $\mathcal{O}(n^2)$  zu haben, da es sich um die Multiplikation einer vollbesetzten Matrix mit einem Vektor handelt. Die Stärke der FFT liegt darin, dass sie die asymptotische Komplexität von  $\mathcal{O}(n \log n)$  hat, was um Welten besser ist.

**Beispiel 9.3.1.** Wir können die DFT via einer naiven, auf Schleifen basierten Implementierung, oder via Multiplikation mit der Matrix  $\mathbf{F}_n$ , oder via `fft.fft` realisieren. Hier sind die Ergebnisse, die für sich sprechen:

Code 9.3.2: Laufzeit verschiedener Implementierungen der DFT

```

1  from numpy import zeros, random, exp, pi, meshgrid, r_, dot, fft
2  import timeit
3
4  def naiveFT():
5      global y, n

```

```

7     c = 0.*1j*y
      # naive
9     omega = exp(-2*pi*1j/n)
      c[0] = y.sum(); s = omega
11    for jj in range(1,n):
        c[jj] = y[n-1]
13        for kk in range(n-2,-1,-1): c[jj] = c[jj]*s+y[kk]
        s *= omega
15    #return c

17    def matrixFT():
        global y, n
19
        # matrix based
21        I, J = meshgrid(r_[0:n], r_[0:n])
        F = exp(-2*pi*1j*I*J/n)
23        c = dot(F,y)
        #return c

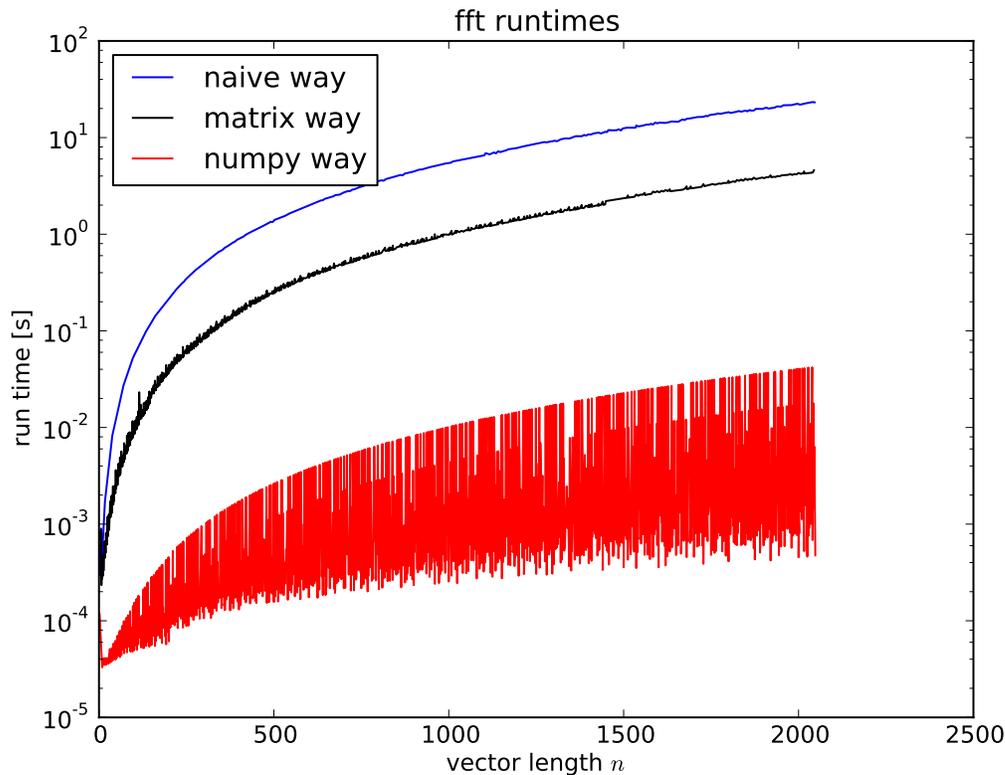
25    def fftFT():
27        global y,n
        c = fft.fft(y)
29        #return c

31    nrexp = 5
        N = 2**11 # how large the vector will be
33    res = zeros((N,4))
        for n in range(1,N+1):
35        y = random.rand(n)

37        t = timeit.Timer('naiveFT()', 'from __main__ import naiveFT')
        tn = t.timeit(number=nrexp)
39        t = timeit.Timer('matrixFT()', 'from __main__ import matrixFT')
        tm = t.timeit(number=nrexp)
41        t = timeit.Timer('fftFT()', 'from __main__ import fftFT')
        tf = t.timeit(number=nrexp)
43        #print n, tn, tm, tf
        res[n-1] = [n, tn, tm, tf]
45

47    print(res[:,3])
        from pylab import semilogy, show, plot, savefig
        semilogy(res[:,0], res[:,1], 'b-')
49        semilogy(res[:,0], res[:,2], 'k-')
        semilogy(res[:,0], res[:,3], 'r-')
51    #savefig('ffftime.eps')
        show()

```



**Abb. 9.3.3.** Die Funktion `fft` ist viel besser als alle anderen Implementierungen.

Es gibt viele Algorithmen zur Durchführung einer FFT. Der heute am häufigsten verwendete Algorithmus wurde zuerst von Gauss<sup>2</sup> im Jahr 1805 entdeckt und dann 1965 von Cooley und Tukey<sup>3</sup> wiederentdeckt.

Dieser Algorithmus heisst auch Cooley & Turkey - Algorithmus und beruht auf dem Prinzip “Divide & conquer”. Die Idee dahinter ist, dass man das Problem einer DFT von einer Länge  $n$  zu berechnen in die Berechnung vieler DFT mit wesentlich kleineren Längen zurückführt und dann diese Ergebnisse wieder zusammenführt. Die Idee kann in vier Schritten beleuchtet werden:

1. Vektoren der Länge  $n = 2m$
2. Vektoren der Länge  $n = 2^L$
3. Vektoren der Länge  $n = pq$  mit  $p, q$  ganzen Zahlen
4. Vektoren der Länge  $n$ , wobei  $n$  eine Primzahl ist

<sup>2</sup>“Nachlass: Theoria interpolationis methodo nova tractata”, Werke, Band 3, 265 - 327 (Königliche Gesellschaft der Wissenschaften, Göttingen, 1866)

<sup>3</sup>An algorithm for the machine calculation of complex Fourier series, Math. Comput. 19, 297-301 (1965)

Für den ersten Schritt führen wir eine Umformung der diskreten Fouriertransformation durch:

$$\begin{aligned}
 c_k &= \sum_{j=0}^{n-1} y_j e^{-\frac{2\pi i}{n} jk} \\
 &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n} 2jk} + \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n} (2j+1)k} \\
 &= \sum_{j=0}^{m-1} y_{2j} e^{-\frac{2\pi i}{n/2} jk} + e^{-\frac{2\pi i}{n} k} \sum_{j=0}^{m-1} y_{2j+1} e^{-\frac{2\pi i}{n} 2jk}
 \end{aligned}$$

Die Summen stellen nun die geraden und ungeraden Koeffizienten der Diskreten Fouriertransformation dar. Diese bekommt man nun wieder durch eine diskrete Fouriertransformation. Dies kann man nun prinzipiell solange wiederholen, bis man die DFT von einem Vektor der Länge eins berechnen muss, was dann ein einfach ist, da die entsprechende Transformationsmatrix eine  $1 \times 1$  - Matrix mit dem Eintrag  $\omega_n^0 = 1$  ist. Damit ist das Problem eigentlich für Vektoren der Länge  $n = 2^L$  gelöst, der folgende Python-Code zeigt, wie man dies in Python realisieren kann.

**Code 9.3.4:** FFT Implementation für Vektoren der Länge  $n = 2^L$

```

1  from numpy import linspace, random, hstack, pi, exp
2  def fftrec(y):
3      n = len(y)
4      if n == 1:
5          c = y
6          return c
7      else:
8          c0 = fftrec(y[::2])
9          c1 = fftrec(y[1::2])
10         r = (-2.j*pi/n) * linspace(0,n-1,n)
11         c = hstack((c0,c0)) + exp(r) * hstack((c1,c1))
12     return c

```

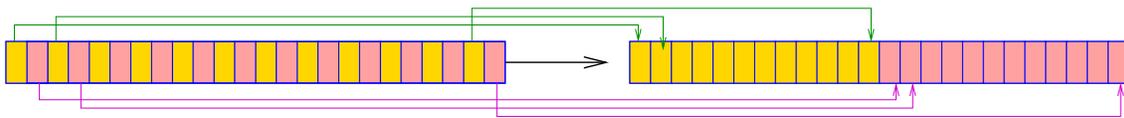
Die Kosten dieser Rekursion sind:



und jede Stufe der Rekursion braucht  $\mathcal{O}(2^L)$  Operationen.

**Bemerkung 9.3.5.** Wir können die FFT auch als Matrix-Zerlegung anschauen. Für  $n = 2m$  sei die Permutation

$$P_m^{\text{OE}}(1, \dots, n) = (1, 3, \dots, n-1, 2, 4, \dots, n).$$



Da aber  $\omega_n^{2j} = \omega_m^j$  haben wir für diese Permutation der Spalten:

$$P_m^{\text{OE}} \mathbf{F}_n = \begin{bmatrix} \mathbf{F}_m & \mathbf{F}_m \\ \mathbf{F}_m \begin{bmatrix} \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \end{bmatrix} & \mathbf{F}_m \begin{bmatrix} \omega_n^{n/2} & & & \\ & \omega_n^{n/2+1} & & \\ & & \ddots & \\ & & & \omega_n^{n-1} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \mathbf{F}_m & & & \\ & \mathbf{I} & & \\ & & \mathbf{I} & \\ & & & \mathbf{F}_m \end{bmatrix} \begin{bmatrix} & & & \\ \omega_n^0 & & & \\ & \omega_n^1 & & \\ & & \ddots & \\ & & & \omega_n^{n/2-1} \\ & & & & -\omega_n^0 & & & \\ & & & & -\omega_n^1 & & & \\ & & & & & \ddots & & \\ & & & & & & & -\omega_n^{n/2-1} \end{bmatrix}$$

Nun ist es allerdings so, dass man in der Numerik nicht nur Vektoren mit solcher Länge schnell transformieren möchte. Was macht man nun, wenn man einen Vektor der Länge  $n = pq$  hat? Dann verwendet man den Cooley - Tukey - Algorithmus.

Zusammenfassend lässt sich sagen, dass die FFT auf dem Prinzip beruht, dass es effizienter ist, viele einfache kleine Probleme zu lösen, als eine Matrixmultiplikation mit  $\mathcal{O}(n^2)$  auszuführen. Man beobachtet in der Praxis, dass die FFT für Zweierpotenzen am schnellsten ist, Zahlen mit kleinen Primfaktoren sind auch noch relativ schnell, für Zahlen mit grossen Primfaktoren oder Primzahlen doch deutlich langsamer.

## 9.4 Trigonometrische Interpolation und rechen-technische Aspekte

Unser Zugang zur Fourier-Approximation war: gegeben  $f \in L^2(0, 1)$ , können wir  $f$  als eine 1-periodische Funktion auf  $\mathbb{R}$  sehen:

$$f(t) = \sum_{k=-\infty}^{\infty} \widehat{f}(k) e^{2\pi i k t} \text{ in } L^2(0, 1),$$

wobei die exakten Fourier-Koeffizienten durch ein Integral gegeben sind:

$$\widehat{f}(k) = \int_0^1 f(t) e^{-2\pi i k t} dt.$$

Die Fourier-Approximation erhält man, wenn man die unendliche Fourier-Reihe abbricht; analog der Taylor-Reihe (die abgebrochen zur Approximation durch Taylor-Polynom führt) erhalten wir eine Approximation mit trigonometrischen Polynome:

$$f_{2n+1}(t) = \sum_{k=-n}^n \widehat{f}(k) e^{2\pi i k t} \text{ was } 2n + 1 \text{ Koeffizienten braucht,}$$

bzw.

$$f_{2n}(t) = \sum_{k=-n}^{n-1} \widehat{f}(k) e^{2\pi i k t} \text{ was } 2n \text{ Koeffizienten braucht,}$$

welche eine 1-periodische Funktion ist, obwohl  $f$  nicht unbedingt periodisch war. Die Approximation gilt im Sinne der  $L^2(0, 1)$ -Konvergenz und die Konvergenzgeschwindigkeit hängt entscheidend von der Glattheit der Periodisierung von  $f$  ab. Die Orthogonalität der  $e^{2\pi i k t}$ -Funktionen bezüglich des  $L^2(0, 1)$ -Skalarprodukts macht diese Approximation auch eine orthogonale Projektion im Raum der Trigonometrischer Polynome vom Grad  $\geq 2n$  bzw.  $\geq 2n - 1$ .

Wenn wir die Fourier-Koeffizienten  $\widehat{f}(k)$  nicht exakt kennen, müssen wir diese auch approximieren. Eine Option, die wir bereits gesehen haben, ist das Integral durch die Trapezregel (siehe dazu das Kapitel über die Quadratur) anzunähern

$$\widehat{f}(k) = \int_0^1 f(t) e^{-2\pi i k t} dt \approx \frac{1}{N} \sum_{\ell=0}^{N-1} f\left(\frac{\ell}{N}\right) e^{-2\pi i k \frac{\ell}{N}} =: \widehat{f}_N(k)$$

für eine 1-periodische Funktion  $f$  (d.h.  $f(0) = f(1)$ ). Somit sind die diskreten Fourier-Koeffizienten  $\widehat{f}_N(k)$  für  $k = 0, 1, \dots, N-1$  durch die Anwendung der Fourier-Matrix auf dem Vektor der Funktionswerte in den Punkten  $\frac{\ell}{N}$  zu berechnen. Die richtige Zuordnung zu den Frequenzen im trigonometrischen Polynom geschieht mit dem `fftshift` wie in der Theorem 9.2.10 beschrieben; hier ist sie nochmals (für gerade  $N$ ):

$$\begin{aligned}\widehat{f}_N(0) &= \gamma_0 \approx \widehat{f}(0), \dots, \widehat{f}_N\left(\frac{N}{2} - 1\right) = \gamma_{\frac{N}{2}-1} \approx \widehat{f}\left(\frac{N}{2} - 1\right) \\ \widehat{f}_N(N-1) &= \gamma_{-1} \approx \widehat{f}(-1), \dots, \widehat{f}_N\left(\frac{N}{2}\right) = \gamma_{-\frac{N}{2}} \approx \widehat{f}\left(-\frac{N}{2}\right).\end{aligned}$$

**Bemerkung 9.4.1.** Lassen Sie uns diese diskrete Fourier-Approximation mit  $2n$  Koeffizienten in den Punkten  $\frac{\ell}{N}$  für  $\ell = 0, 1, \dots, N-1$  auswerten. Zuerst wollen wir uns um den Shift kümmern:

$$f_{2n}(x) = \sum_{k=-n}^{n-1} \gamma_k e^{2\pi i k x} = \sum_{k=0}^{n-1} \gamma_k e^{2\pi i k x} + \sum_{k=-n}^{-1} \gamma_k e^{2\pi i k x}.$$

Für  $k \geq 0$  ist  $\gamma_k = \widehat{f}_N(k) = \frac{1}{N} \sum_{j=0}^{N-1} f\left(\frac{j}{N}\right) e^{-2\pi i k \frac{j}{N}}$  und für  $k < 0$  ist  $\gamma_k = \widehat{f}_N(N+k) = \frac{1}{N} \sum_{j=0}^{N-1} f\left(\frac{j}{N}\right) e^{-2\pi i (N+k) \frac{j}{N}}$ . Da  $\omega_N^{(N+k) \frac{j}{N}} = (\omega_N^{N+k})^{\frac{j}{N}} = (\omega_N^k)^{\frac{j}{N}} = \omega_N^{k \frac{j}{N}}$ , erhalten wir

$$f_{2n}(x) = \frac{1}{N} \sum_{j=0}^{N-1} f\left(\frac{j}{N}\right) \sum_{k=-n}^{n-1} e^{2\pi i k (x - \frac{j}{N})},$$

welche wir nun in  $\frac{\ell}{N}$  auswerten:

$$\begin{aligned}f_{2n}\left(\frac{\ell}{N}\right) &= \frac{1}{N} \sum_{j=0}^{N-1} \sum_{k=-n}^{n-1} f\left(\frac{j}{N}\right) \omega_N^{k(j-\ell)} = \frac{1}{N} \sum_{j=0}^{N-1} f\left(\frac{j}{N}\right) \sum_{k=-n}^{n-1} \omega_N^{k(j-\ell)} \\ &= f\left(\frac{\ell}{N}\right),\end{aligned}$$

denn die Orthogonalität der diskreten Fourier-Vektoren gibt  $\sum_{k=-n}^{n-1} \omega_N^{k(j-\ell)} = 0$ , ausser für  $j = \ell$ .

Somit erfüllt unsere diskrete Fourier-Approximation die Interpolationsbedingung in den Punkten  $\frac{\ell}{N}$ :

$$f_N\left(\frac{\ell}{N}\right) = f\left(\frac{\ell}{N}\right) \text{ für } \ell = 0, 1, \dots, N-1,$$

dass heisst, sie löst auch die Interpolationsaufgabe

finde  $p_N$  das trigonometrische Polynom vom Grad  $N$ , so dass

$$p_N\left(\frac{\ell}{N}\right) = f\left(\frac{\ell}{N}\right) \text{ für } \ell = 0, 1, \dots, N-1.$$

Für allgemeine  $N \in \mathbb{N}$  definieren wir nun die Räume der trigonometrischen Polynome

$$\mathcal{T}_N = \text{span}\{e^{2\pi ijt} ; j = -n, -n+1, \dots, 0, 1, \dots, n\}, \text{ falls } N = 2n$$

und

$$\mathcal{T}_N = \text{span}\{e^{2\pi ijt} ; j = -n, -n+1, \dots, 0, 1, \dots, n-1\}, \text{ falls } N = 2n-1.$$

Für die Lösung der Interpolationsaufgabe im Raum  $\mathcal{T}_N$  in den Punkten  $t_\ell \in [0, 1]$  haben wir zwei Möglichkeiten:

- (1) via Lösung eines linearen Gleichungssystems:

$$\sum_{j=-n}^n \gamma_j e^{2\pi ijt_\ell} = f(t_\ell) \text{ für } \ell = 0, 1, \dots, 2n.$$

Das kostet  $\mathcal{O}(N^3)$  Operationen, aber funktioniert für beliebige  $t_\ell \in [0, 1]$ .

- (2) via FFT in  $\mathcal{O}(N \log N)$  Operationen, falls die Punkte äquidistant sind  $t_\ell = \frac{\ell}{N}$ , denn die Matrix des obigen Gleichungssystems ist  $\mathbf{F}_N^{-1}$ .

#### Code 9.4.2: Berechnung der Koeffizienten des trigonometrischen Polynoms

```

1 from numpy import hstack, arange, sin, cos, pi, outer, exp, linspace, fft,
  random, conj
2 from numpy.linalg import solve
3 import time
4
5 def trigpolycoeff(t, y):
6     N = y.shape[0]
7
8     if N%2 == 1:
9         n = (N-1)/2.
10        M = exp(2*pi*1j*outer(t,arange(-n,n+1)))
11    else:
12        n = N/2.
13        M = exp(2*pi*1j*outer(t,arange(-n,n)))
14        c = solve(M, y)
15    return c
16
17 N = 2**12
18 t = linspace(0,1,N, endpoint=False)
19 y = random.rand(N)
20 tic = time.time()
21 direkt = trigpolycoeff(t,y)
22 toc = time.time() - tic
23 print('time for a direkt transform',toc, ' s')
24 tic = time.time()
25 viafft = fft.fftshift(fft.fft(y)/N)
  toc = time.time() - tic

```

```

27 print('time for a transform via fft',toc, ' s')
    tic = time.time()
29 viaIfft = conj(fft.fftshift(fft.ifft(y)))
    toc = time.time() - tic
31 print('time for a transform via ifft',toc, ' s')
    print(abs(direkt-viafft).max(), abs(direkt-viaIfft).max())

```

Eine andere Frage ist, wie wir ein solches trigonometrisches Polynom

$$p(t) = \sum_{j=-n}^{n-1} \gamma_j e^{2\pi i j t}$$

effizient an den äquidistanten Punkten  $\frac{k}{M}$  mit  $M = 2m > N = 2n$  für  $k = 0, 1, 2, \dots, M - 1$  auswerten können. Wir schreiben dafür  $p_{2n} \in \mathcal{T}_{2n} \subset \mathcal{T}_{2m}$  in der trigonometrischen Basis von  $\mathcal{T}_{2m}$ , was einfach die Einfügung von Nullen im Koeffizientenvektor an der Stelle der höheren Frequenzen bedeutet. Direkt nach der Anwendung der Funktion `fft` oder `ifft` müssen wir die Nulle in der Mitte des Koeffizientenvektors einfügen. Die folgenden zwei Codes geben zwei Möglichkeiten diese Auswertung zu tun, wobei der zweite einige Multiplikationen mehr braucht.

**Code 9.4.3:** Schnelle Auswertung des trigonometrischen Interpolationspolynoms auf äquidistanten Punkten

```

1  from numpy import zeros, fft, sqrt, pi, real, linalg, linspace, sin
3  def evaliptrig(y,N):
    n = len(y)
5     if (n%2) == 0:
        c = fft.ifft(y)
7         a = zeros(N, dtype=complex)
        a[:n//2] = c[:n//2]
9         a[N-n//2:] = c[n//2:]
        v = fft.fft(a);
11        return v
    else: raise TypeError('odd length')
13
14 f = lambda t: 1./sqrt(1.+0.5*sin(2*pi*t))
15
16 vlinf = []; vl2 = []; vn = []
17 N = 4096; n = 2
18 while n < 1+2**7:
19     t = linspace(0,1,n, endpoint=False)
    y = f(t)
21     v = real(evaliptrig(y,N))
    t = linspace(0,1,N, endpoint=False)
23     fv = f(t)
    d = abs(v-fv); linf = d.max()
25     l2 = linalg.norm(d)/sqrt(N)
    vlinf += [linf]; vl2 += [l2]; vn += [n]
27     n *= 2

```

```

29 from pylab import semilogy, show
    semilogy(vn,vl2,'-+'); show()

```

**Code 9.4.4:** Schnelle Auswertung des trigonometrischen Interpolationspolynoms auf äquidistanten Punkten

```

1  from numpy import zeros, fft, sqrt, pi, real, linalg, linspace, sin
3  def evaliptrig(y,N):
4      n = len(y)
5      if (n%2) == 0:
6          c = fft.fft(y)*1./n
7          a = zeros(N, dtype=complex)
8          a[:n//2] = c[:n//2]
9          a[N-n//2:] = c[n//2:]
10         v = fft.ifft(a)*N;
11         return v
12     else: raise TypeError('odd length')
13
14 f = lambda t: 1./sqrt(1.+0.5*sin(2*pi*t))
15
16 vlinf = []; vl2 = []; vn = []
17 N = 4096; n = 2
18 while n < 1+2**7:
19     t = linspace(0,1,n, endpoint=False)
20     y = f(t)
21     v = real(evaliptrig(y,N))
22     t = linspace(0,1,N, endpoint=False)
23     fv = f(t)
24     d = abs(v-fv); linf = d.max()
25     l2 = linalg.norm(d)/sqrt(N)
26     vlinf += [linf]; vl2 += [l2]; vn += [n]
27     n *= 2
28
29 from pylab import semilogy, show
    semilogy(vn,vl2,'-+'); show()

```

## 9.5 Trigonometrische Interpolation und Fehlerabschätzungen

Wir wollen nun den Fehler betrachten, wenn wir  $f : [0, 1] \rightarrow \mathbb{R}$  mittels trigonometrischer Interpolation in den äquidistanten Punkten  $\frac{k}{N}$ , für  $k = 0, 1, \dots, N - 1$  approximieren. Wir motivieren diese Fehlerbetrachtung durch drei Beispiele:

(1) Stufenfunktion: die periodische Fortsetzung dieser Funktion  $f$ :

$$f : [0, 1] \rightarrow \mathbb{R}$$

$$f(t) = \begin{cases} 0 & \text{für } |t - \frac{1}{2}| > 1/4 \\ 1 & \text{für } |t - \frac{1}{2}| \leq 1/4. \end{cases} \quad (9.5.20)$$

(2) Periodische glatte Funktion:

$$h : \mathbb{R} \rightarrow \mathbb{R}$$

$$h(t) = \frac{1}{\sqrt{1 + \frac{1}{2} \sin(2\pi t)}}. \quad (9.5.21)$$

(3) Hutfunktion: die periodische Fortsetzung der Funktion  $g$ :

$$g : [0, 1] \rightarrow \mathbb{R}$$

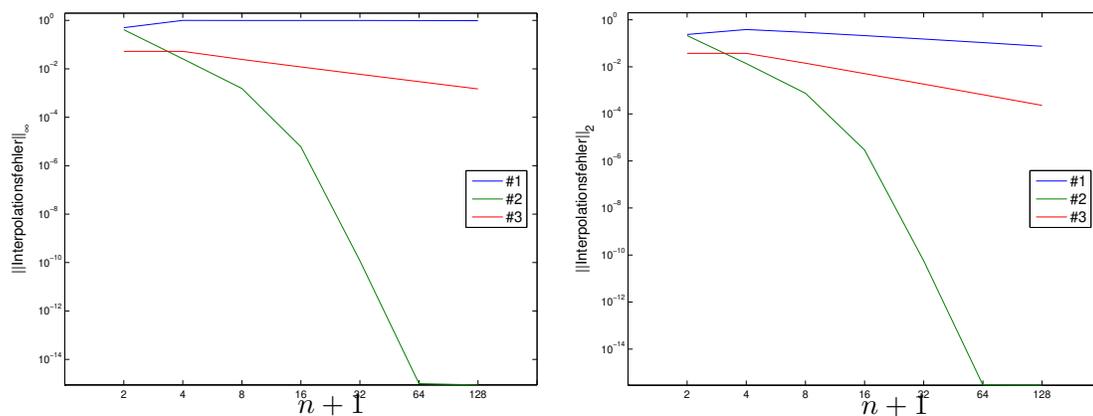
$$g(t) = |t - \frac{1}{2}|. \quad (9.5.22)$$

In den folgenden Plots sehen wir die Konvergenz dieser Funktionen in Abhängigkeit vom Grad des interpolierenden trigonometrischen Polynoms dargestellt.

```

1  t = linspace(0,1,n+1); y = f(t[:-1])
3  v = real(evaliptrig(y,N))
5  t = linspace(0,1,N+1); fv = f(t)
   d = abs(v-fv[:-1]); linf = d.max()
   l2 = linalg.norm(d) / sqrt(N)

```



**Abb. 9.5.1.** Fall (1) und (3): algebraische Konvergenz, Fall (2): exponentielle Konvergenz

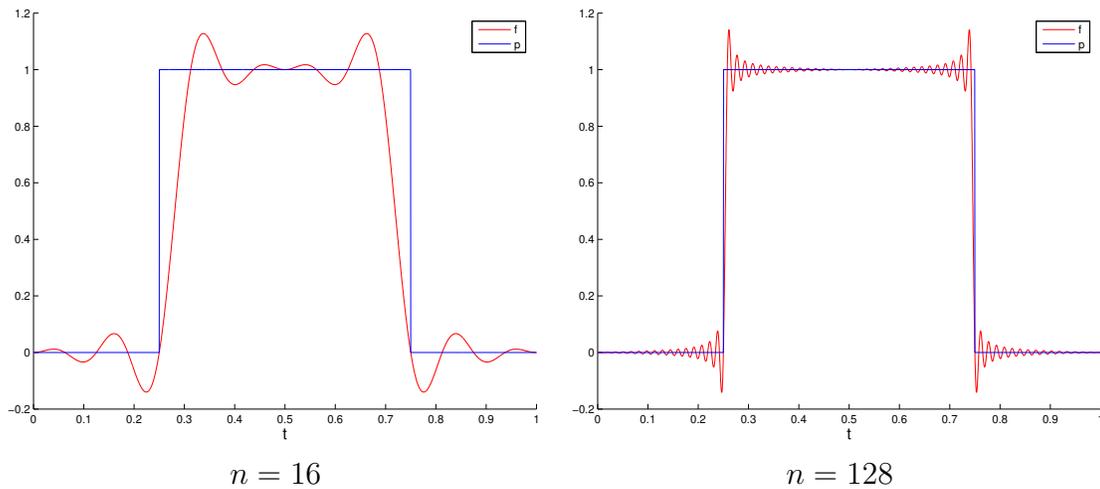


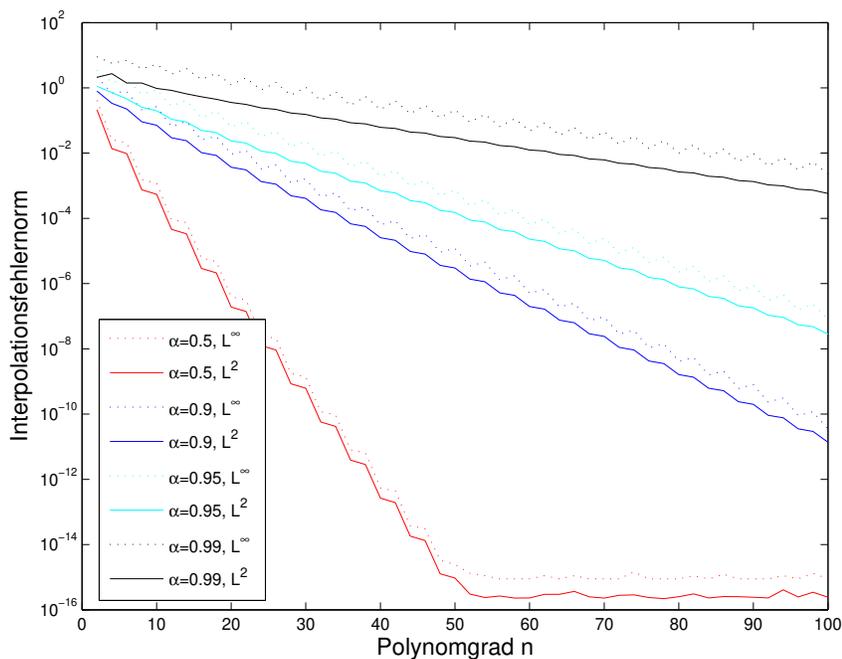
Abb. 9.5.2. Gibb's Phänomen an Sprungstellen

In diesem Beispiel ist es eindeutig zu erkennen, dass die Stellen, wo die Funktion unglatt ist, für die Verlangsamung der Konvergenz zuständig sind. Noch interessanter ist das Beispiel, wo wir eine sehr glatte Funktion mit einem zusätzlichen Parameter haben:

**Beispiel 9.5.3.** Für  $\alpha \in [0, 1)$  definieren wir:

$$f(t) = \frac{1}{\sqrt{1 - \alpha \sin(2\pi t)}} \text{ auf } I = [0, 1],$$

und wir beobachten exponentielle Konvergenz in  $n$ , schneller für kleineres  $\alpha$ .



**Abb. 9.5.4.** Fehler in der trigonometrischer Interpolation

**Theorem 9.5.5** (Aliasing). Der  $k$ -te Fourierkoeffizient des  $N$ -ten trigonometrischen Interpolationspolynoms unterscheidet sich vom  $k$ -ten Fourierkoeffizienten von  $f$  gerade um die Summe aller Fourierkoeffizienten (ausser  $j = 0$ ), die um das ganze Vielfache von  $N$  um den  $k$ -ten Fourierkoeffizienten verschoben sind:

$$\widehat{p}_N(k) - \widehat{f}(k) = \sum_{j \in \mathbb{Z}, j \neq 0} \widehat{f}(k + jN). \quad (9.5.23)$$

Die erste Aussage des folgenden Korollars folgt aus der Bemerkung über die Fourierkoeffizienten der Ableitung. Wenden wir dieses Lemma  $n$ -mal an und benutzen dann auf beiden Seiten die Identität von Parseval, so folgt:

$$\|f^{(n)}\|_{L_2(0,1)}^2 = (2\pi)^{2n} \sum_{k=-\infty}^{\infty} k^{2n} |\widehat{f}(k)|^2. \quad (9.5.24)$$

Falls  $|\widehat{f}^{(n)}(k)| \leq \int_0^1 |f^{(n)}(t)| dt \leq \infty$  erfüllt ist, gilt  $|\widehat{f}^{(n)}(k)| = \mathcal{O}(k^{-n})$  für  $|k| \rightarrow \infty$ .

**Korollar 9.5.6.** Für  $f \in C^p$  mit  $p \geq 2$  und 1-periodisch, gilt:

$$\widehat{p}_N(k) - \widehat{f}(k) = \mathcal{O}(N^{-p}) \quad \text{für } |k| \leq \frac{N}{2}. \quad (9.5.25)$$

Aus Folgerung kann man aus diesem Satz ziehen, dass die Approximation von den Fourierkoeffizienten von  $f$  bei kleinen Frequenzen, d.h. hier konkret  $|k| < \frac{N}{2}$  gut durch die Fourierkoeffizienten der trigonometrischen Interpolationspolynome approximiert wird.

Der folgende Satz stellt einen interessanten Zusammenhang zwischen den Fourierkoeffizienten einer Funktion  $f$  und der Güte der Approximation durch trigonometrische Funktionen dar.

**Theorem 9.5.7** (Fehler der trigonometrischen Interpolation). Sei  $f$  eine 1-periodische Funktion und konvergiere die Reihe  $\sum_{k \in \mathbb{Z}} \widehat{f}(k)$  absolut, dann kann man

$$|p_N(x) - f(x)| \leq 2 \sum_{|k| \geq N/2} |\widehat{f}(k)|, \quad \forall x \in \mathbb{R} \quad (9.5.26)$$

für den Approximationsfehler schreiben.

Der Fehler bei der Interpolation mit trigonometrischen Polynomen wird also durch die Beträge der schlecht approximierbaren Fourierkoeffizienten  $\widehat{f}(k)$  nach oben beschränkt. Es stellt sich die Frage, wann man Funktionen mittels trigonometrischer Polynome perfekt approximieren kann. Mit Polynomen  $n$ -ten Grades kann man ja alle Polynome vom kleineren Grad perfekt approximieren. Bei periodischen Funktionen ist nicht der Grad ausschlaggebend, sondern die "Frequenzen" der Funktion.

**Korollar 9.5.8** (Perfekte Approximation, Abtasttheorem). Sei  $f$  eine 1-periodische Funktion mit der maximalen Frequenz  $m$ , das heisst  $\widehat{f}(k) = 0$  für alle  $|k| > m$ . Falls  $N > 2m$ , gilt dann  $p_N(x) = f(x)$  für alle  $x$ .

Man sieht also, dass falls die Fourierkoeffizienten ab einem gewissen  $|k|$  identisch verschwinden, dann wird die Funktion perfekt approximiert. Was bedeutet dies anschaulich? Wenn in unserem trigonometrischen Interpolationspolynom Summanden vorkommen, die mehr als doppelt so schnell wie die zu interpolierende Funktion oszillieren, ist perfekte Interpolation möglich.

Diese Sätze haben aber eine erstaunliche Konsequenz für die Anwendung. Stellen wir uns vor, dass wir ein analoges Musiksignal haben und dieses digitalisieren wollen. Wir messen also das Signal auf einer diskreten Menge:

$$\mathcal{T} = \left\{ \frac{j}{n} \right\}_0^{n-1} \quad (9.5.27)$$

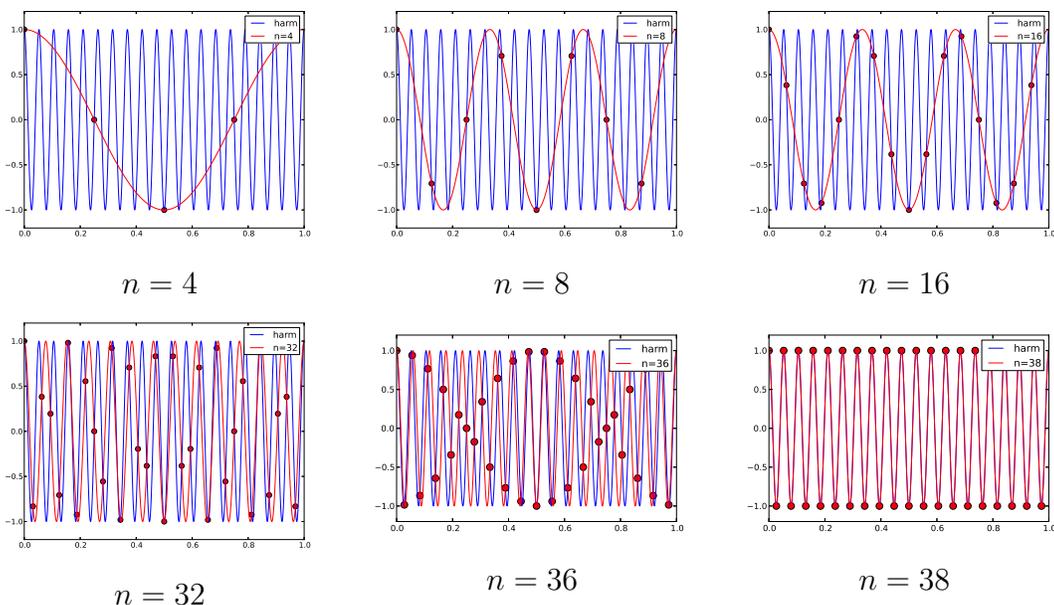
Sei  $N$  nun die maximale Frequenz (d.h.  $N$ , so dass  $\hat{f}(k) = 0$  für alle  $|k| > N$ ) mit der die Funktion schwingt, und  $n$  unsere Abtastfrequenz mit  $n < N$ . Dann gilt für alle Punkte aus unserer Messung:

$$\exp(2\pi i N t) = \exp(2\pi i (N - n) t) \quad (9.5.28)$$

Interpolieren wir diese beiden Funktionen, so erhält man, dass folgende Gleichheit gilt:

$$p_n(e^{2\pi i N}) = p_n(e^{2\pi i (N \bmod n)}) \quad (9.5.29)$$

Die interpolierenden Polynome sind also gleich für die beiden unterschiedlichen Funktionen, d.h. die wesentlich langsamer schwingende Funktion erzeugt die eine genau gleich gute Interpolation. Wenn man also ein Signal mit zu wenig Punkten abtastet, bekommt man durch diesen Effekt den sog. "Aliasing - Effekt". Das gleiche interpolierende Polynom für zwei klar unterscheidliche Funktionen, die in ihrer Periode klar abweichen. Daraus folgt dann, dass viele Details der zu interpolierenden Funktion verloren gehen. Für  $f(t) = \sin(2\pi \cdot 19t)$  und  $N = 38$  entstehen bei  $n = 4, 8, 16$  Funktionen, die wenig mit dem Original zu tun haben, wie in der Abbildung 9.5.9 zu sehen ist.



**Abb. 9.5.9.** Aliasing für  $f(t) = \cos(2\pi \cdot 19t)$

Auf unser Musikstück übertragen bedeutet dies, dass es nicht mehr so wie vorher klingt. Ein weiteres interessantes Beispiel für diesen Effekt ist die scheinbare Rückwärtsbewegung von Autorädern, die in alten Filmen bei zu niedriger Bildrate pro Sekunde zu sehen ist.

Man kann mit genauer Analyse der Situation den folgenden Satz beweisen, der die Regularität der Funktion direkt in Verbindung mit der Güte der Approximation der trigonometrischen Interpolation setzt.

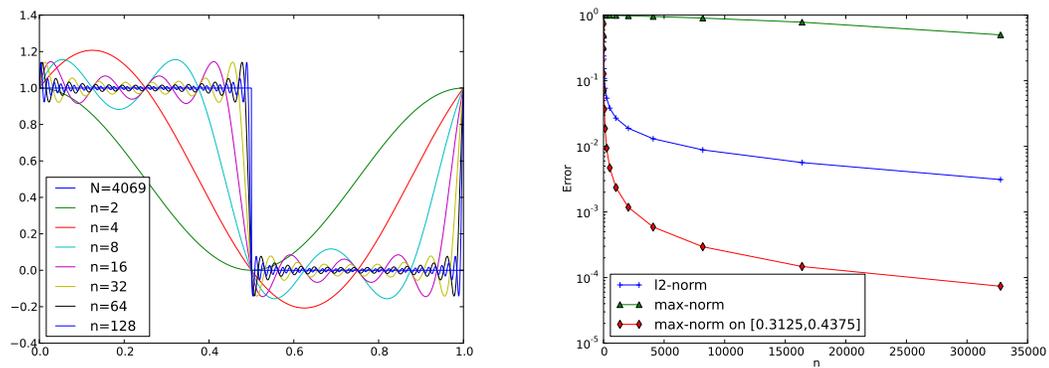
**Theorem 9.5.10** (Fehlerschätzung für trigonometrische Interpolation). Sei für  $k \in \mathbb{N}$   $f^{(k)} \in L^2(0, 1)$ , dann:

$$\|f - p_N(f)\|_{L^2(0,1)} \leq \sqrt{1 + c_k} N^{-k} \|f^{(k)}\|_{L^2(0,1)} \tag{9.5.30}$$

mit

$$c_k = 2 \sum_{l=1}^{\infty} (2l - 1)^{-2k} . \tag{9.5.31}$$

Je mehr Ableitungen in  $L^2(0, 1)$  liegen, desto kleiner ist der Fehler. Man beachte, dass man hier eine Konvergenz im Sinne der  $L^2$  - Norm hat, eine solche Konvergenz ist schwächer als eine Konvergenz im Sinne der Supremumsnorm. Zur Illustartion vom Konvergenzverhalten schauen wir uns die folgenden Figuren an.



**Abb. 9.5.11.** Trigonometrische Approximation der Stufenfunktion (9.5.20)

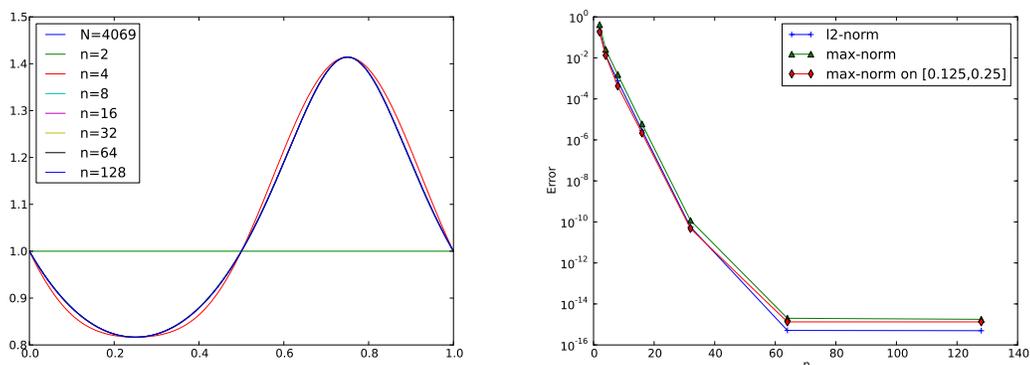


Abb. 9.5.12. Trigonometrische Approximation der glatten Funktion (9.5.21)

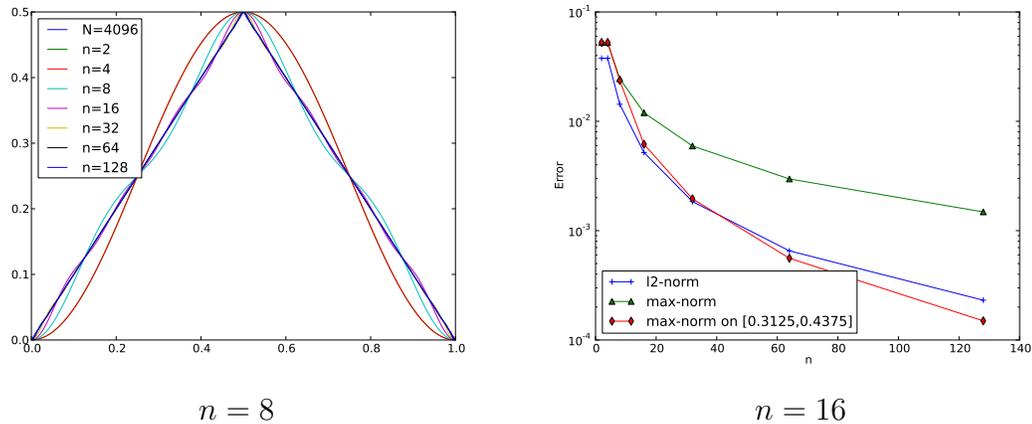


Abb. 9.5.13. Trigonometrische Approximation der Hutfunktion (9.5.22)

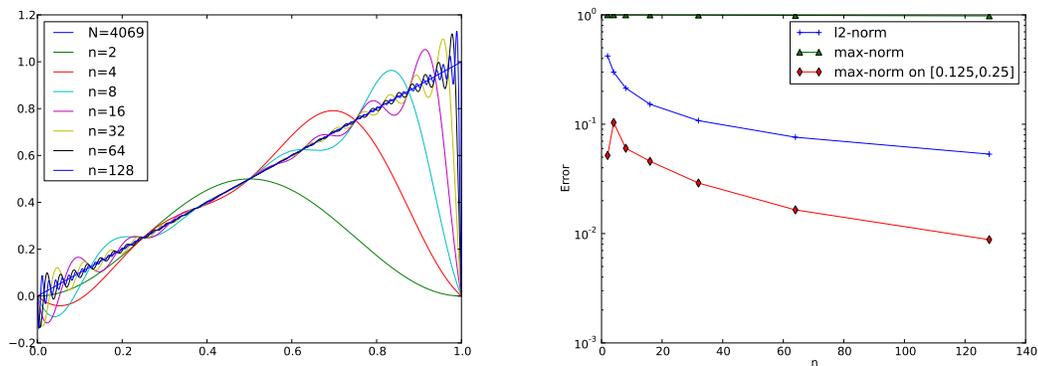


Abb. 9.5.14. Trigonometrische Approximation einer Rampenfunktion; durch Periodisierung entstehen Sprünge an 0 und 1.

## 9.6 DFT und Chebyshev-Interpolation

Wir haben im vorherigen Kapitel gesehen, dass man glatte Funktion numerisch gut durch Chebyshev-Polynome interpolieren kann. Mit der DFT bekommt man einen sehr effizienten Weg diese Interpolation durchzuführen. Wir haben nun ein Chebyshev Interpolationspolynom  $p \in \mathcal{P}_n$  auf  $[-1, 1]$  zu einer Funktion  $f : [-1, 1] \rightarrow \mathbb{C}$ . Nach der dortigen Konstruktion gilt dann für die Chebyshev - Knoten  $t_k$  die Gleichheit  $f(t_k) = p(t_k)$ , wobei die die Chebyshev - Knoten definiert sind als:

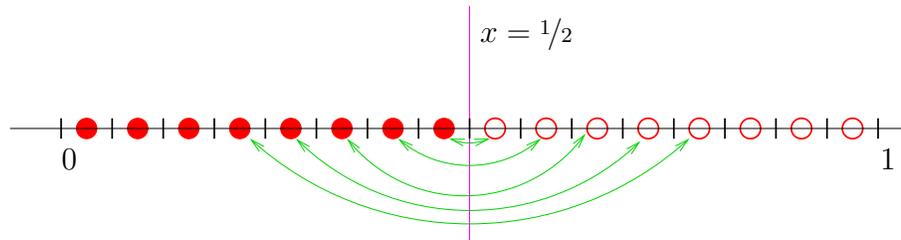
$$t_k := \cos \left( \frac{2k + 1}{2(n + 1)} \pi \right), \quad k = 0, \dots, n.$$

Wir definieren uns die beiden Hilfsfunktion  $g$  und  $q$ :

$$\begin{aligned} g : [-1, 1] &\rightarrow \mathbb{C} : & g(s) &:= f(\cos 2\pi s) \\ q : [-1, 1] &\rightarrow \mathbb{C} : & q(s) &:= p(\cos 2\pi s). \end{aligned}$$

Dann übersetzt unsere Interpolationsbedingung in den Chebyshev - Knoten wie folgt:

$$p(t_k) = f(t_k) \Leftrightarrow q\left(\frac{2k+1}{4(n+1)}\right) = g\left(\frac{2k+1}{4(n+1)}\right)$$



Wir wenden nun die Translation  $s^* = s + \frac{1}{4(n+1)}$  an und definieren wieder neue Hilfsfunktionen:

$$g^*(s) := g(s^*) = g\left(s + \frac{1}{4(n+1)}\right)$$

$$q^*(s) := q(s^*) = q\left(s + \frac{1}{4(n+1)}\right)$$

Wir zeigen nun, dass  $q(s)$  ein trigonometrische Polynom ist, genauer zeigen wir, dass  $q(s) \in \mathcal{T}_{2n+1}$  ist:

$$\begin{aligned} q(s) &= p(\cos 2\pi s) \\ &= \sum_{j=0}^n a_j T_j(\cos(2\pi s)) \\ &= \sum_{j=0}^n a_j \cos(2\pi j s) \\ &= \sum_{j=0}^n \frac{1}{2} a_j (e^{2\pi i j s} + e^{-2\pi i j s}) \\ &= \sum_{j=-n}^{n+1} b_j e^{-2\pi i j s}, \end{aligned}$$

wobei

$$b_j := \begin{cases} 0 & , \text{ for } j = n + 1 , \\ \frac{1}{2} a_j & , \text{ for } j = 1, \dots, n , \\ a_0 & , \text{ for } j = 0 , \\ \frac{1}{2} a_{n-j} & , \text{ for } j = -n, \dots, -1 , \end{cases}$$

Die Symmetrie  $q(s) = q(1 - s)$  und  $q\left(\frac{2k+1}{4(n+1)}\right) = f(t_k)$  für  $k = 0, \dots, n$  ergeben

zusammen die Interpolationsbedingungen

$$q\left(\frac{k}{2(n+1)} + \frac{1}{4(n+1)}\right) = z_k := \begin{cases} y_k & , \text{ for } k = 0, \dots, n, \\ y_{2n+1-k} & , \text{ for } k = n+1, \dots, 2n+1, \end{cases} \quad (9.6.32)$$

und somit ist  $q^*$  das trigonometrische Interpolationspolynom von  $g^*$ . Man kann also die Chebyshev-Interpolation durch eine DFT durchführen. Weiterhin übertragen sich die Fehlerabschätzungen von der trigonometrischen Interpolation direkt auf die Tschebyscheff-Interpolation hier.

Für die konkrete Implementierung ist es nützlich, die Rechnungen anders zu gestalten. Die Nullstellen von  $T_{n+1}$ , d.h. die Chebyshev-Knoten  $t_k$ , lassen sich auch schreiben als

$$t_k = \cos(2\pi\xi_k) \text{ mit } \xi_k = \frac{2k+1}{4(n+1)}.$$

Wir sahen bereits

$$q(s) = p(\cos 2\pi s) = \sum_{j=-n}^{n+1} b_j \exp(-2\pi i j s).$$

Nun ist

$$y_k = p(t_k) = p(\cos 2\pi t_k) = q(t_k) = q\left(\frac{2k+1}{4(n+1)}\right) = \sum_{j=-n}^{n+1} b_j \exp\left(-2\pi i j \frac{2k+1}{4(n+1)}\right).$$

Jetzt unterscheidet sich die Rechnung von der vorigen, denn wir isolieren  $\exp\left(\frac{-2\pi i j}{4(n+1)}\right)$ :

$$y_k = \sum_{j=-n}^{n+1} b_j \exp\left(\frac{-2\pi i j}{4(n+1)}\right) \exp\left(\frac{-2\pi i j k}{2n+2}\right) \text{ für } k = 0, 1, \dots, n+1$$

und mit der Notation

$$\zeta_j = b_j \exp\left(\frac{-2\pi i j}{4(n+1)}\right) \text{ für } j = -n, \dots, -1, 0, 1, \dots, n+1$$

erhalten wir

$$y_k = \sum_{j=-n}^{n+1} \zeta_j \exp\left(\frac{-2\pi i j k}{2n+2}\right) \text{ für } k = 0, 1, \dots, n.$$

Der Nenner  $2n+2$  in dieser Formel verleitet zu einer Fourier-Transformation doppelter Länge. Wie in (9.6.32) definieren wir

$$z_j = y_j \text{ für } j = 0, 1, \dots, n$$

und per Symmetrie

$$z_j = z_{2n+1-j} \text{ für } j = n+1, \dots, 2n+1,$$

und somit haben wir

$$z_\ell = \sum_{j=-n}^{n+1} \zeta_j \exp\left(-2\pi i j \tilde{\xi}_\ell\right) \text{ mit } \tilde{\xi}_\ell = \frac{\ell}{2n+2} \text{ für } \ell = 0, 1, \dots, 2n+1.$$

Jetzt kommt  $\omega_{2n+2} = \exp\left(\frac{-2\pi i}{2n+2}\right)$  ins Spiel:

$$z_\ell = \sum_{j=-n}^{n+1} \zeta_j \omega_{2n+2}^{j\ell} = \sum_{k=0}^{2n+1} \zeta_{k-n} \omega_{2n+2}^{(k-n)\ell} = \omega_{2n+2}^{-n\ell} \sum_{j=0}^{2n} \zeta_{j-n} \omega_{2n+2}^{j\ell}$$

oder

$$\omega_{2n+2}^{n\ell} z_\ell = \sum_{j=0}^{2n} \zeta_{j-n} \omega_{2n+2}^{j\ell}.$$

Somit

$$\mathbf{F}_{2n+2} \zeta = \left[ \exp\left(\frac{-\pi i n \ell}{n+1}\right) z_\ell \right].$$

### Code 9.6.1: Effiziente Interpolation in den Chebyshev-Knoten

```

1 from numpy import exp, pi, real, hstack, arange
2 from scipy import ifft
3
4
5 def chebexp(y):
6     """Efficiently compute coefficients  $\alpha_j$  in the Chebychev expansion
7      $p = \sum_{j=0}^n \alpha_j T_j$  of  $p \in \mathbb{C}_p$  based on values  $y_k$ ,
8      $k=0, \dots, n$ , in Chebychev nodes  $t_k$ ,  $k=0, \dots, n$ . These values are
9     passed in the row vector  $y$ .
10    """
11
12    # degree of polynomial
13    n = y.shape[0] - 1
14
15    # create vector z by wrapping and componentwise scaling
16
17    # r.h.s. vector
18    t = arange(0, 2*n+2)
19    z = exp(-pi*1.0j*n/(n+1.0)*t) * hstack([y,y[::-1]])
20
21    # Solve linear system for  $\zeta$  with effort  $O(n \log n)$ 
22    c = ifft(z)
23
24    # recover  $\gamma_j$ 
25    t = arange(-n, n+2)
26    b = real(exp(0.5j*pi/(n+1.0)*t) * c)
27
28    # recover coefficients  $c$  of Chebyshev expansion
29    a = hstack([ b[n], 2*b[n+1:2*n+1] ])
30
31    return a

```

**Bemerkung 9.6.2.** Die Chebyshev–Abszissen sind als die Extrema des Chebyshev–Polynoms  $T_{2n}$  definiert:

$$\zeta_k = \cos\left(\frac{k\pi}{2n}\right) \text{ für } k = 0, 1, \dots, 2n.$$

Die Chebyshev–Abszissen mit geradem  $k$  sind genau die Nullstellen von  $T_n$ , d.h. die  $n$  Chebyshev–Knoten. Wir können auch an die Chebyshev–Abszissen sehr effizient und stabil interpolieren.

**Code 9.6.3:** Effiziente Interpolation in den Chebyshev–Abszissen

```

1 from numpy import exp, pi, real, hstack, arange, zeros, flipud, cos, linspace
2 from scipy import ifft, fft
3
4 def chebexp(y):
5     """Efficiently compute coefficients  $\alpha_j$  in the Chebyshev expansion
6      $p = \sum_{j=0}^n \alpha_j T_j$  of  $p \in \mathbb{C}_p$  based on values  $y_k$ ,
7      $k=0, \dots, n$ , in Chebyshev abscissas  $t_k$ ,  $k=0, \dots, n$ . These values
8     are
9     passed in the row vector y.
10    """
11
12    # degree of polynomial
13    n = y.shape[0]-1
14    # create vector z by wrapping
15    z = zeros(2*n, dtype=complex)
16    z[:n+1] = 1.*y
17    z[n+1:] = flipud(y)[1:-1]
18    c = fft(z)/(2.*n)
19
20    a = 1.*c[:n+1]
21    a[0] *= 0.5
22    a[-1] *= 0.5
23
24    return a
25
26 def evalchebexp(a,N):
27     n = a.shape[0]-1
28     c = zeros(2*N, dtype=complex)
29     c[:n+1] = 1.*a
30     c[0] *= 2
31     c[n] *= 2
32     c[2*N-n+1:] = c[n-1:0:-1]
33     z = ifft(c)*2*N
34     y = 1.*z[:N+1]
35     x = cos(arange(N+1)*pi/N)
36     return x,y
37
38 def rungef(x):
39     """Compute the value of the function of Runge's counterexample.
40     """

```

```
    return 1.0 / (1.0 + x**2)
42
def step(x):
44     return where(abs(x-0.5)<0.25, 1., 0.)
def ramp(x):
46     return where(x<1., x, 0.)
def hat(x):
48     return where(abs(x)<=1., 1-abs(x), 0.)
def morf(x):
50     return where(abs(x)<=1, 0.5*(1+cos(pi*x)), 0.)
def smooth(x):
52     return 1./sqrt(1.+0.5*sin(2*pi*x))

54 if __name__ == "__main__":

56     f = rungef
        n = 10
58     x = 5*cos(arange(n+1)*pi/n) # Chebyshev points
        y = f(x)
60     c = chebexp(y)

62     t = linspace(-5.0, 5.0, 201)
        N = 200
64     xN, yN = evalchebexp(c,N)

66     from pylab import plot, legend, xlabel, show
        plot(t, f(t), "r")
68     plot(5*xN,yN,"b--")
        plot(x, y, "k*")
70     legend(["Function","Chebychev"])
        xlabel("t")
72     show()
```