SciPyTutorial Documentation

Release 0.0.4

VG

February 06, 2015

CONTENTS

CHAPTER ONE

BASIC STUFF

1.1 A Dialog on Python for Scientific Computing

Why should I learn Python?

If you consider the possibility that you will work in a different domain than pure mathematics, then there is a good chance that you will need it at a given moment of your career.

It is used in the industry?

Yes, it is used in many companies and by research institutions that use computers as an instrument.

Why? I thought there is Java and C++ and so on...

With Python you can develop and prototype your products much more quickly. If needed, parts of your production code can later be migrated to C/C++/Fortran/Java or whatever.

Whatever?

Yes. One of the qualities that qualified Python to large industrial application is its affinity to many programming languages. This makes it easy to embed applications in both directions.

Aha, so you glue together tools.

Yes, Python is a perfect scripting language, but here I meant that you can really mix programming languages in an easy and consistent way.

What is scripting?

Scripting is a way to let very different applications play together to achieve your aim. This is another quality that makes Python attractive for the industry.

What about research and scientific work?

As a researcher, you would like to know quite soon if your idea makes sense. You would like to run simulations as soon as possible, so you want quickly a correct prototype and not a prototype that runs quickly...

But that was one of the attractions of matlab, so what’s new?

```
If the idea works and the code gives you correct results, but you want it more
efficiently or you want to scope it to other codes or professional visualization
tools, then it is much simpler to migrate parts of python code than to re-write
the code from scratch, as usually happens with matlab projects.
```

I see. So with python I can glue applications together and can migrate critical numerical parts to C/Fortran. But if I do not need to? Isn't matlab enough?

```
Enough? This is a good word!
It depends. Earlier I used to say: if you are satisfied with matlab and you do
not feel you need something more, then stay with matlab.
```

And now?

```
Now I would be more careful. I noticed that people prefer to use what they
learned first than to learn something new. This means that when people get a
difficult new problem, they make great efforts to solve it with the tools they
know better. This gives complicated solutions after great efforts, where
simple rush solutions were possible with new knowledge. You have to hit the
wall, and the hit must really hurt, in order to be really willing to learn
something new. I am sure that people are more efficient and more likely to
obtain good and rush solutions to hard problems if they learn python (and
scientific tools) first.
```

You mean that you can solve more difficult problems with python than with matlab?

```
I mean that with python et al you have better chances, especially for large projects.
```

Why should we learn or use python for teaching numerical methods? Isn't it much too complex? Isn't it simpler to use matlab for this aim?

```
When you start from zero, you need the same effort to learn matlab and Python.

matlab is a very good software. It has also its price, which is very high,
if you leave the university.  Most of our students will not remain in a
university, so working in a small company would be a clear handicap for
our alumni.

matlab is a good software because it was developed over several years under
very strict guidelines. Hence, it is more stable than the on-going development
versions of python et al.

For the very same reasons that make matlab very stable, matlab is very rigid.
Having to cope with the main ideas of 1970-80 makes improvements very hard
to realize. Object-orientation is the best example in this sense. It had to
be introduced in order to make matlab survive, but apart from its marketing
effect, it is useless, since it rather hampers than facilitates development
and code-reuse. Respecting old guidelines makes matlab strong, but rigid and
rather past-oriented. On the contrary, our students will live in the future,
let us give them a chance to make a better world!
```

1.2 Introduction

The aim of these pages is to provide enough information in order to help students to start using python's tools for computational science, numerical methods and implementation of numerical algorithms.

Python is not a Matlab clone, but a very flexible and very powerful (be aware!) high-level programming language. Still, it is so simple to use, that we'll learn it by doing our own job... We focus on scientific computing now, but you might use it for very different tasks, from controlling your experimental machines, by making high-performance visualizations, to sorting your mp3's on your mobile phone.

While coming with 'batteries included' python is only a small kernel that should be completed by modules specialized for your aims.

We'll use the following python modules:

- numpy
- scipy
- matplotlib
- ipython

For the numerics lecture ("Numerische Methoden D-PHYS") we will use the very same Virtual Box machine which was used during the computer science ("Informatik") course. All python related tools we will need during this lecture are already installed there and ready for use. (Note that some of the tools shown in the *advanced* chapter might be missing.) For details on the installation and usage of this virtual machine, we refer to: Anleitung zur Installation von VirtualBox. This machine was kindly prepared by people taking care of the computer science lecture. For the rest of this tutorial we will assume you have a working installation.

1.2.1 Work flow

The two recommended ways of using *ipython* are:

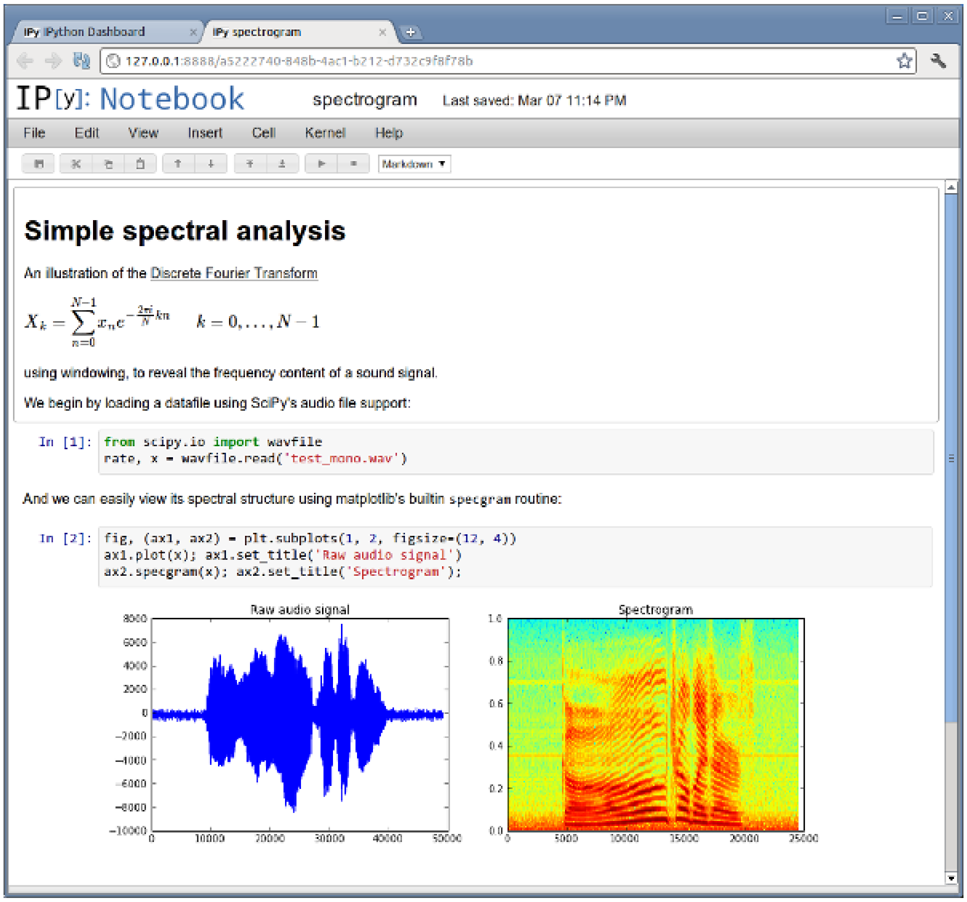
- Your favorite text editor and the *ipython* interpreter running in a shell (*terminal*). This is also the *only* supported work flow possible during the computer exam. The tutorial assumes this working style.
- The ipython-notebook. It's main advantage is that it unifies Code, Output, inline graphics and text/formulas. Similar to Mathematica. Furthermore it provides inline documentation, i.e. function signatures and help, directly under the cursor.

> **Warning:** Please be aware that the *only* supported setup on the exam computers consists of an editor (*gedit*) and the *ipython* interpreter running in a shell. Hence you should be familiar with that too. Other tools might or might not be installed and their use is at your own risk.

The *ipython*-notebook

Inside the Virtual-Box installation you can run the notebook with:

```
python -m IPython notebook --no-mathjax --matplotlib=inline --notebook-dir=/home/ifmp14/
```



Simple spectral analysis

An illustration of the Discrete Fourier Transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \qquad k = 0, \dots, N-1$$

using windowing, to reveal the frequency content of a sound signal.

We begin by loading a datafile using SciPy's audio file support:

```
In [1]: from scipy.io import wavfile
        rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib's builtin `specgram` routine:

```
In [2]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
        ax1.plot(x); ax1.set_title('Raw audio signal')
        ax2.specgram(x); ax2.set_title('Spectrogram');
```

1.3 First introduction to Python

1.3.1 Fast-track Python

The python source code is run by an `interpreter` either interactively or as script. The common python interpreter can be launched by the `python` command from any shell window. In general we recommend not to use the bare python interpreter but an enhanced and much more user friendly version called *IPython*. Start an interactive IPython session by running the `ipython` command:

```
ifmp14@ifmp14:~$ ipython
Python 2.7.6 (default, Mar 22 2014, 22:59:38)
Type "copyright", "credits" or "license" for more information.

IPython 2.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython’s features.
%quickref -> Quick reference.
help      -> Python’s own help system.
object?   -> Details about ’object’, use ’object??’ for extra details.

In [1]:
```

In Python everything is an object and the types are dynamic

```
In [1]: a = 1.5   # The rest of this line is a comment

In [2]: type(a)
Out[2]: float

In [3]: a = 1 # redefine a as an integer

In [4]: type(a)
Out[4]: int

In [5]: a = 1e-10 # redefine a as a float with scientific notation

In [6]: type(a)
Out[6]: float

In [7]: a = 1+5j # redefine a as complex

In [8]: type(a)
Out[8]: complex

In [9]: print("a="+str(a))
a=(1+5j)

In [10]: print(a)
(1+5j)

In [11]: a
Out[11]: (1+5j)

In [12]: print(2**6)
64

In [13]: a**8
Out[13]: (-3824-456960j)

In [14]: 2**6
Out[14]: 64
```

str, unicode - string types (immutable):

```
In [15]: s1 = ’hello’

In [16]: s2 = u’φ(α) + ψ(α+β+γ)’  # prepend u, gives unicode string

In [17]: s1[0]
Out[17]: ’h’

In [18]: s1[0], s1[1]
Out[18]: (’h’, ’e’)

In [19]: type(s1)
Out[19]: str

In [20]: type(s2)
Out[20]: unicode

In [21]: print(s2)  # The ’print’ shows the string using the actual special characters
φ(α) + ψ(α+β+γ)
```

```
In [22]: s2         # This way we just see the unicode encoding
Out[22]: u’\u03c6(\u03b1) + \u03c8(\u03b1+\u03b2+\u03b3)’
```

list - mutable sequence:

```
In [23]: ell = [1,2,’three’] # make list

In [24]: type(ell)
Out[24]: list

In [25]: type(ell[0])
Out[25]: int

In [26]: type(ell[2])
Out[26]: str

In [27]: ell[2] = 3

In [38]: ell.append(4)

In [29]: ell
Out[29]: [1, 2, 3, 4]
```

tuple - immutable sequence:

```
In [30]: tempty = ()

In [31]: toneelement = (1,)  # Note the trailing comma here (and only here)

In [32]: ttwoelement = (1, 2)

In [33]: tempty
Out[33]: ()

In [34]: toneelement
Out[34]: (1,)

In [35]: ttwoelement
Out[35]: (1, 2)

In [36]: t = (1, 2, ’four’)

In [37]: t[2]
Out[37]: ’four’

In [38]: t[2] = 4
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-34-56c309f1aa91> in <module>()
----> 1 t[2] = 4

TypeError: ’tuple’ object does not support item assignment
```

python - loop:

```
In [39]: for element in xrange(1,21,2):
   ....:     print(element)
   ....:
1
```

```
3
5
7
9
11
13
15
17
19
```

```
In [40]: for element in xrange(1,21,2):
   ....: print(element)
   ....:
  File "<ipython-input-43-ff776174c35a>", line 2
    print(element)
        ^
IndentationError: expected an indented block
```

hence indentation is the delimiter for blocks. Usually IPython takes care for you. Pressing Enter is enough.

python - functions:

A pythonic way of adding first n numbers:

```
def sf(n):
    k = 0
    result = 0

    while True:
        print("current number to add =" + str(k))
        result += k
        k += 1
        if k > n:
            break

    return result
```

In the above example note that the indentation delimits the blocks and that no semi-colons are needed.

We can enter this in our interactive ipython session

```
In [50]: def sf(n):
   ....:     k = 0
   ....:     result = 0
   ....:     while True:
   ....:         print("current number to add = " + str(k))
   ....:         result += k
   ....:         k += 1
   ....:         if k > n:
   ....:             break
   ....:     return result
   ....:

In [51]: sf(10)
current number to add = 0
current number to add = 1
current number to add = 2
current number to add = 3
current number to add = 4
current number to add = 5
current number to add = 6
current number to add = 7
current number to add = 8
current number to add = 9
current number to add = 10
Out[51]: 55
```

Note how nicely ipython handles indentation for us when we start new nested blocks. On the other hand we can not make empty lines in this interactive mode because this would be interpreted as end of the input. (Actually, we *can* make empty lines but only directly after opening a new nested block.)

For larger pieces of code there is much advantage in copying the above definition of the function `sf` into a file, let us call it `sumy.py`. Python source code files have the extention `.py`. Then we start ipython in the same directory, and type:

```
In [1]: from sumy import *
```

to load the code. Then we can call the function like:

```
In [2]: sf(3)
current number to add = 0
current number to add = 1
current number to add = 2
current number to add = 3
Out[2]: 6
```

1.3.2 Mutable and immutable objects

Warning: Arguments are always passed by assignement.
Python's pass-by-assignment scheme isn't quite the same as C++'s reference parameters option, but it turns out to be very similar to the C language's argument-passing model in practice:

- Immutable arguments are effectively passed "by value." Objects such as integers and strings are passed by object reference instead of by copying, but because you can't change immutable objects in-place anyhow, the effect is much like making a copy.
- Mutable arguments are effectively passed "by pointer." Objects such as lists and dictionaries are also passed by object reference, which is similar to the way C passes arrays as pointers – mutable objects can be changed in-place in the function, much like C arrays.

Of course, if you've never used C, Python's argument-passing mode will seem simpler still – it involves just the assignment of objects to names, and it works the same whether the objects are mutable or not.

```
In [1]: def f(a):     # a is assigned to (references) the passed object
   ...:     a = 99    # changes local variable a only: here simply resets 'a' to a completely differen
   ...:

In [2]: b = 88

In [3]: f(b)          # 'a' and 'b' both reference same 88 initially

In [4]: print(b)      # b not changed
88
```

Assignment to an argument name inside a function (e.g., `a=99`) does not magically change a variable like `b` in the scope of the function call. Argument names may share passed objects initially (they are essentially pointers to those objects), but only temporarily, when the function is first called. As soon as an argument name is reassigned, this relationship ends. This is also related to the *copy-on-write* semantics.

That is the case for assignment to argument names themselves. When arguments are passed mutable objects like lists and dictionaries, we also need to be aware that inplace changes to such objects may live on after a function exits, and hence impact callers. Here's an example that demonstrates this behavior:

```
In [5]: def changer(a,b):    # Arguments assigned references to objects
   ...:     a = 2            # Changes local name's value only
   ...:     b[0] = 'spam'    # Changes shared object in-place
   ...:

In [6]: X = 1

In [7]: L = [1, 2]

In [8]: changer(X,L)         # Pass immutable and mutable objects

In [9]: X                    # X is unchanged, L is different!
Out[9]: 1

In [10]: L
Out[10]: ['spam', 2]
```

If we don't want in-place changes within functions to impact objects we pass to them, though, we can simply make explicit copies of mutable objects. For function arguments, we can always copy the list at the point of call:

```
In [11]: L = [1, 2]

In [12]: changer(X, L[:])    # Pass a copy, so our 'L' does not change

In [13]: X
Out[13]: 1

In [14]: L
Out[14]: [1, 2]
```

We can also copy within the function itself, if we never want to change passed-in objects, regardless of how the function is called:

```
In [15]: def changer(a, b):
   ....:     b = b[:]         # Copy input list so we don't impact caller
   ....:     a = 2
   ....:     b[0] = 'spam'    # Changes our list copy only
   ....:

In [16]: L = [1, 2]

In [17]: changer(X, L)

In [18]: X
Out[18]: 1

In [19]: L
Out[19]: [1, 2]
```

Both of these copying schemes don't stop the function from changing the object – they just prevent those changes from impacting the caller. To really prevent changes, we can always convert to immutable objects to force the issue. Tuples, for example, throw an exception when changes are attempted:

```
In [20]: changer(X, tuple(L))    # Pass a tuple, so changes are errors
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-26-f94776bffe6b> in <module>()
----> 1 changer(X, tuple(L)) # Pass a tuple, so changes are errors

<ipython-input-15-a06105498d20> in changer(a, b)
      2     b = b[:]
      3     a = 2
----> 4     b[0] = 'spam' # Changes our list copy only
      5

TypeError: 'tuple' object does not support item assignment
```

Elements of writing good Python **style** are on Google Python Style Guide or PythonStyle.

1.3.3 Number formatting

You generally do not want to display a floating point result of a calculation in its raw form, often with an enormous number of digits after the decimal point, like 23.457413902458498. You are likely to prefer rounding it to something like 23.46.

Python features sprintf()-style formatting, which you might know already from Matlab or C. The most common format specifiers are `%d` for integers, `%f` for floating point numbers, `%e` for scientific notation and `%s` for strings.

Examples

A float rounded to 2 digits after the comma with 6 digits in total:

```
In [1]: '%+6.2f' % 0.2
Out[1]: ' +0.20'
```

Scientific notation with 2 digits after the comma:

```
In [2]: s = '%.2e' % 0.2
Out[2]: 2.00e-01
```

Integer with a fixed size of 5 digits and left-padded with zeros:

```
In [3]:
s = '%05d' % 34
print(s)
Out[3]: 00034
```

It is also possible to replace more than one number in a string:

```
In [4]: 'GMRES solver converged to a rel. residual of %.3e after %d iterations in %.2f seconds.' % (
Out[4]: 'GMRES solver converged to a rel. residual of 1.232e-03 after 1000 iterations in 5.10 seconds
```

An exhaustive list of options can be found in section 5.6.2. String Formatting Operations: of the python documentation.

1.3.4 Formatting tables with `str.format()` (Optional)

This section is optional and mainly intended as a reference for formatting plain text tables.

The `str.format()` is an alternative way in python for string formatting, it replaces *fields* contained in a string marked with curly braces `{}`. A *field* is specified by the following syntax:

```
{[keyword] : [alignment][format specifier]}
```

The format specifier is identical to the previous section, except that the `%` has to be omitted now. Valid alignment options are >, <, ^, resp. left, right and center.

- Example:

  ```
  In [1]: '{Class:5s} has {NStudents:d} students'.format(Class='Numerische Methoden', NStudents=20
  Out[1]: 'Numerische Methoden has 200 students'
  ```

  `keyword` is optional and can be omitted completely or replaced by the argument number:

  ```
  In [43]: '{:5s} has {:d} students'.format('Numerische Methoden', 200)
  Out[43]: 'Numerische Methoden has 200 students'
  ```

- Example: formatting a table with 3 columns and proper alignment

  ```
  In [2]: title = '{0:>15s}\t{1:>15s}\t{2:>15s}'.format('Refinement','Mesh width', 'Abs. error')
     ...: print(''.join(len(title)*['-'])) # separator '-------'
     ...: print(title)
     ...: print(''.join(len(title)*['-']))
     ...: for i in xrange(5):
     ...:     h = 2**(-i)
     ...:     err = h**5
     ...:     row = '{0:15d}\t{1:15.2f}\t{2:15.2e}'
     ...:     print(row.format(i, h, err))
     ...: print(''.join(len(title)*['-']))
  ```

  ```
  Out [2]:

  ----------------------------------------------
       Refinement         Mesh width       Abs. error
  ----------------------------------------------
                0               1.00         1.00e+00
                1               0.50         3.12e-02
                2               0.25         9.77e-04
                3               0.12         3.05e-05
                4               0.06         9.54e-07
  ----------------------------------------------
  ```

For more options see section 7.1.3. Format String Syntax of the python documentation.

1.3.5 Some nice extra specialities (Optional)

python - list comprehensions:

Often we want to construct lists in a very systematic way, for example a list of all square numbers. This can be done easily by a so called *list comprehension*:

```
In [5]: L = [k**2 for k in xrange(10)]

In [6]: L
Out[6]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

It is even possible to attach a condition, for example we can compute squares of even numbers only:

```
In [83]: L = [k**2 for k in xrange(10) if k%2 == 0]
```

```
In [84]: L
Out[84]: [0, 4, 16, 36, 64]
```

python - generators:

The python documentation states:

> They [generators] are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called, the generator resumes where it left-off [...]

Let's just make a trivial example:

```
In [37]: def squares(n):
   ....:     for i in xrange(n):
   ....:         yield i**2
   ....:

In [38]: S = squares(10)

In [39]: S.next()
Out[39]: 0

In [40]: S.next()
Out[40]: 1

In [41]: S.next()
Out[41]: 4

In [42]: S.next()
Out[42]: 9
```

Until we reach the end, where it raises a `StopIteration` exception

```
In [48]: S.next()
Out[48]: 81

In [49]: S.next()
^[[A
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-49-8d33773269b4> in <module>()
----> 1 S.next()

StopIteration:
```

This way we can iterate over generated values, and for example combine this with a list comprehension to find the first square numbers:

```
In [55]: L = [s for s in squares(10)]

In [56]: L
Out[56]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Generators can be useful to represent infinite data structures lazily. For example we can compute *all* square numbers, one after the other:

```
In [71]: def squares():
   ....:     i = 0
   ....:     while True:
   ....:         yield i
   ....:         i = i+1
   ....:

In [72]: S = squares()

In [73]: S.next()
Out[73]: 0

In [74]: S.next()
Out[74]: 1

In [75]: S.next()
Out[75]: 2
```

ad infinitum ...

python - Anonymous functions (lambda functions):

A *lambda function* is a (usually small) function without an explicit name. The are defined like regular function but use the `lambda` keyword instead of `def` and have no name. Let's look at an example:

```
In [90]: lambda x,y : x + y - 2
Out[90]: <function __main__.<lambda>>
```

Of course we can assign this function object to any variable name we like:

```
In [91]: f = lambda x,y : x + y - 2
```

This is now equivalent to a regular definition but involes less typing:

```
In [55]: def f(x, y):
   ....:     return x+y-2
   ....:
```

Lambda functions have a few restrictions, for example we can not do any variable assignment or control blocks inside. But still it's possible to do some fancy stuff:

```
In [59]: f = lambda n: [i for i in xrange(n)]

In [60]: f(10)
Out[60]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Finally, the Zen of Python (hat you might find useful outside of Pyhton, too):

```
In [35]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
```

```
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.3.6 More on the Python language

You can find more on Python in on-line tutorials and books. I enjoyed *Learning Python* by M. Lutz and *Python in a Nutshell* by A. Martelli.

There as the very good official python tutorial covering most of the important stuff and also some things not so important for us:

> http://docs.python.org/2/tutorial/

You should definitely read at least the following three chapters:

- An Informal Introduction to Python Covers all the basic information that did not fit into this tutorial document.
- More Control Flow Tools Control structures like `if` blocks, `for` and `while` loops and function definitions.
- Data Structures The basic data structures like lists, tuples, dictionaries (hash tables) and sets.

For the ones of you interested in object-oriented programming in the python laguage, there is also a chapter on classes and inheritance.

- Classes Covers the topics related to object-oriented programming in python.

Finally, python comes with a huge standard library included.

- The Python Standard Library Reference for the python standard library.

And if this is still not enough, there is a central storage called *PyPI* containing thousands of python packages for almost any job

- PyPI - the Python Package Index A python package for any job.

1.4 First introduction to NumPy

The most important and central python package for doing scientific computing is *NumPy*. This package provides convenient and fast arbitrary-dimensional array manipulation routines. The home of NumPy as well as some other scientific packages is:

> http://www.scipy.org

We will give a very short introduction to NumPy now.

The main python object is a N-dimensional array data structure. In the following sections we show different core aspects of this array object.

1.4.1 Basic properties

The NumPy arrays have three fundamental properties.

- **`shape`: The shape attribute of any arrys describes its size** along all of its dimensions
- `ndim`: The number of dimensions (often also called *directions* or *axes*)

- `dtype`: The data type of the array elements.

Lets look at some examples to get a feeling for these concepts. At the very beginning we make sure to import (load) the `numpy` python module (package):

```
In [1]: from numpy import *    # Import all (top-level) functions from numpy
```

We can create arrays from (nested) python lists or tuples:

```
In [2]: A = array([1,2,3,4,5])

In [3]: A
Out[3]: array([1, 2, 3, 4, 5])

In [4]: type(A)
Out[4]: numpy.ndarray
```

This is a simple vector of 5 elements. The type of the array *itself* is `numpy.ndarray`. The array is one-dimensional as expected:

```
In [5]: A.ndim
Out[5]: 1
```

and has the following shape (shapes are always return as python tuples)

```
In [6]: A.shape
Out[6]: (5,)
```

The data type of the *elements* of `A` is:

```
In [7]: A.dtype
Out[7]: dtype('int64')
```

But most of the time we don't need to bother us with types. Let's create a simple matrix:

```
In [10]: M = array([[1,2,3],[4,5,6]])

In [11]: M
Out[11]:
array([[1, 2, 3],
       [4, 5, 6]])
```

We should think of matrices simply as two-dimensional arrays:

```
In [12]: M.ndim
Out[12]: 2
```

This one having two rows and three columns:

```
In [13]: M.shape
Out[13]: (2, 3)
```

We could define higher-dimensional arrays too but will do this rarely.

The full specification of the `ndarray` object can be read at:

> http://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html

1.4.2 Array creation

There are several other methods to construct arrays beside the initialization by nested lists. Remember that the list can also be a list comprehension:

```
In [23]: S = array([i**2 for i in xrange(10)])

In [24]: S
Out[24]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

We will learn a much better way later to achieve the same. It is possible to use lambda functions together with nested list comprehensions for advanced matrix constructions. In the following we build a so-called Toeplitz matrix:

```
In [25]: arc = lambda r,c: r-c

In [26]: T = array([[arc(r,c) for c in xrange(4)] for r in xrange(5)])

In [27]: T
Out[27]:
array([[ 0, -1, -2, -3],
       [ 1,  0, -1, -2],
       [ 2,  1,  0, -1],
       [ 3,  2,  1,  0],
       [ 4,  3,  2,  1]])
```

Often we want to initialize arrays for later use. For example we can create arrays of `zeros` or `ones` by these two functions of same name:

```
In [14]: Z = zeros((12,))

In [15]: Z
Out[15]: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])

In [16]: O = ones((3,3))

In [17]: O
Out[17]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Identity matrices can be made with `eye`:

```
In [34]: I = eye(3)

In [35]: I
Out[35]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

If we need `N` equally spaced points between two endpoints `a` and `b` we can call `linspace(a, b, N)` like:

```
In [38]: points = linspace(0, 10, 15)

In [39]: points.shape
Out[39]: (15,)

In [40]: points
Out[40]:
```

```
array([  0.        ,   0.71428571,   1.42857143,   2.14285714,
         2.85714286,   3.57142857,   4.28571429,   5.        ,
         5.71428571,   6.42857143,   7.14285714,   7.85714286,
         8.57142857,   9.28571429,  10.        ])
```

If we need only some integer ranges, we can either convert the result of Python's `range` or directly call NumPy's `arange` (remember that the last point is always *excluded* in ranges but *included* in the result of `linspace`):

```
In [53]: array(range(5,13))
Out[53]: array([ 5,  6,  7,  8,  9, 10, 11, 12])

In [54]: arange(5,13)
Out[54]: array([ 5,  6,  7,  8,  9, 10, 11, 12])
```

For computing a two-dimensional grid of points there is the `meshgrid` command. We show only a small example here:

```
In [43]: x = linspace(0, 3, 4)

In [44]: x
Out[44]: array([ 0.,  1.,  2.,  3.])

In [45]: y = linspace(0, 2, 3)

In [46]: y
Out[46]: array([ 0.,  1.,  2.])

In [47]: X,Y = meshgrid(x,y)

In [48]: X
Out[48]:
array([[ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.],
       [ 0.,  1.,  2.,  3.]])

In [49]: Y
Out[49]:
array([[ 0.,  0.,  0.,  0.],
       [ 1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.]])
```

Meshgrids are commonly used for plotting in three dimensions.

Sometimes we already have an array and want to create another one with *same shape*. For this we have the functions `zeros_like` and `ones_like` at hand:

```
In [50]: zeros_like(T)
Out[50]:
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])

In [52]: ones_like(A)
Out[52]: array([1, 1, 1, 1, 1])
```

Finally, from time to time we need random numbers. There are powerful methods for filling arrays with random numbers [1]. The most simple call looks like:

```
In [58]: random.random((3,5))
Out[58]:
array([[ 0.79921796,  0.01877047,  0.56102347,  0.57946688,  0.37414866],
       [ 0.84593219,  0.44403091,  0.89057679,  0.11296198,  0.72045779],
       [ 0.40033809,  0.81665445,  0.07307056,  0.24462193,  0.8721881 ]])
```

and gives us values according to the continuous uniform distribution on $[0, 1[$.

More on array creation can be found in the NumPy reference:

> http://docs.scipy.org/doc/numpy/reference/routines.array-creation.html

1.4.3 Indexing and Slicing

Indexing some single value or even a whole set of values from a given array is the next important operation we need to learn about.

Indexing for retrieving single elements works as usual in python and indexing by negative numbers starts counting from the end:

```
In [63]: l = arange(9)

In [64]: l
Out[64]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])

In [65]: l[3]
Out[65]: 3

In [66]: l[-2]
Out[66]: 7
```

We run into `IndexError` exceptions if we try to access elements not available:

```
In [69]: l[9]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-69-622e6bac6342> in <module>()
----> 1 l[9]

IndexError: index out of bounds

In [74]: l[-10]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-74-3f63644bd4e3> in <module>()
----> 1 l[-10]

IndexError: index out of bounds
```

In general we have to index an array `A` with as many indices as there are dimensions (There are some more or less esoteric exceptions).

```
In [75]: A = eye(4)

In [76]: A
Out[76]:
```

[1] http://docs.scipy.org/doc/numpy/reference/routines.random.html

```
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])

In [77]: A[0,0]
Out[77]: 1.0

In [78]: A[0,3]
Out[78]: 0.0

In [80]: A[1,1,1]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-80-97042a7c163c> in <module>()
----> 1 A[1,1,1]

IndexError: invalid index
```

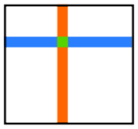
In principle we can also index A by just one value and get:

```
In [79]: A[1]
Out[79]: array([ 0.,  1.,  0.,  0.])
```

we will learn later how to interpret this result. In any case you should be very careful with such things.

Slicing

The term *slicing* is used for extracting whole rows, columns or other rectangular (hypercubic) subregions from an arbitrary dimensional array. One can easily select the colored regions from the following images.



We can extract (slice out) single rows or columns by indexing with colons;

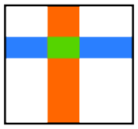
```
In [130]: A = array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

In [131]: A
Out[131]:
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

In [132]: r1 = A[1,:]    # Get the second row (indexing starts at 0)

In [133]: r1
Out[133]: array([5, 6, 7, 8])

In [134]: r1.shape
Out[134]: (4,)

In [135]: c2 = A[:,2]    # Get the third column (indexing starts at 0)
In [136]: c2
Out[136]: array([ 3,  7, 11])

In [137]: c2.shape
Out[137]: (3,)
```

Now we can also understand what `A[1]` does:

```
In [158]: A[1]
Out[158]: array([5, 6, 7, 8])
```

If we provide too less indices, then NumPy appends as many `:` as necessary to the *end* of the given index (tuple).

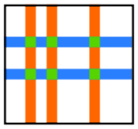
Its possible to select more that one row or column at once.



The syntax stays the same:

```
In [142]: A[:,0:3]    # The first three columns
Out[142]:
array([[ 1,  2,  3],
       [ 5,  6,  7],
       [ 9, 10, 11]])

In [144]: A[1:3,:]    # The second two rows
Out[144]:
array([[ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

In [146]: A[1:3,1:3]    # A submatrix slice
Out[146]:
array([[ 6,  7],
       [10, 11]])
```

What if we want to select a non-contiguous block, like shown in the next image.



Select all orange columns? Select all blue rows? Or even all green elements? This is also possible, we just have to use tuple notation:

```
In [152]: A[:, (0,2,3)]
Out[152]:
array([[ 1,  3,  4],
       [ 5,  7,  8],
       [ 9, 11, 12]])

In [153]: A[(0,1),:]
```

```
Out[153]:
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

The major caveat here is the following:

```
In [154]: A[(0,1),(0,3)]   # What? Why did we not get 4 elements?
Out[154]: array([1, 8])

In [155]: A[(0,1),(0,2,3)]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-155-3c3d973d8ee2> in <module>()
----> 1 A[(0,1),(0,2,3)]

ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

It is even possible to use *arrays* to index other arrays like in the following example:

```
In [180]: i = array([0,2])

In [181]: A[:,i]
Out[181]:
array([[ 1,  3],
       [ 5,  7],
       [ 9, 11]])

In [182]: A[i,:]
Out[182]:
array([[ 1,  2,  3,  4],
       [ 9, 10, 11, 12]])
```

This is the key to solve our failed index example from above. We can indeed select the 6 non-contiguous single elements and get the result packet into a two-dimensional array:

```
In [160]: u=array([[0,1]])

In [161]: u
Out[161]: array([[0, 1]])

In [162]: v=array([[0],[2],[3]])

In [163]: A[u,v]
Out[163]:
array([[1, 5],
       [3, 7],
       [4, 8]])
```

As a last example lets look at so called *boolean indexing* where we select all elements having some common property:

```
In [188]: a
Out[188]: array([[0, 1, 2, 3, 4]])

In [190]: A = a*transpose(a)

In [191]: A
Out[191]:
array([[ 0,  0,  0,  0,  0],
       [ 0,  1,  2,  3,  4],
       [ 0,  2,  4,  6,  8],
       [ 0,  3,  6,  9, 12],
       [ 0,  4,  8, 12, 16]])

In [192]: A[A>=6]   # Select all elements from A that are larger than 6
Out[192]: array([ 6,  8,  6,  9, 12,  8, 12, 16])
```

A special case of indexing is iteration where we iterate in a loop over all elements of an array (in row-wise fashion) like over the elements of a simple python list. For multi-dimensional array we have to flatten it first.

```
In [198]: [i**2 for i in A.flatten()]
Out[198]:
[0,
 0,
 0,
 0,
 0,
 0,
 1,
 4,
 9,
 16,
 0,
 4,
 16,
 36,
 64,
 0,
 9,
 36,
 81,
 144,
 0,
 16,
 64,
 144,
 256]
```

If you still want to learn more about the NumPy indexing and slicing features, there is a whole section in the reference:

> http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

1.4.4 Array manipulation

We can stack together arrays (with matching shapes of course) along the first (rows) or second (columns) axes:

```
In [13]: A = arange(4)

In [14]: B = vstack([A,A,A])

In [15]: B
Out[15]:
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])

In [8]: C = array([[1,2],[3,4]])
In [9]: C
Out[9]:
array([[1, 2],
       [3, 4]])

In [10]: D = hstack([C,C])

In [11]: D
Out[11]:
array([[1, 2, 1, 2],
       [3, 4, 3, 4]])
```

Another useful operation is *reshaping* of arrays. The condition for this operation is that both shapes will result in the same number of elements.

```
In [17]: B.reshape((6,2))
Out[17]:
array([[0, 1],
       [2, 3],
       [0, 1],
       [2, 3],
       [0, 1],
       [2, 3]])
```

Note that the elements get filled in in row-wise order or more general the elements are taken along the last axes first. In the extreme case we can make the array with a singular dimension:

```
In [23]: B.reshape((12,1))
Out[23]:
array([[0],
       [1],
       [2],
       [3],
       [0],
       [1],
       [2],
       [3],
       [0],
       [1],
       [2],
       [3]])

In [24]: E = B.reshape((12,1))

In [25]: E.shape
Out[25]: (12, 1)

In [26]: E.ndim
Out[26]: 2
```

Note that `E` is still two-dimensional but with shape 1 along the second axis (this is called a singular dimension). This can be inconvenient in some cases (for example for plotting as we will see later). The solution to this is called `squeeze` and removes all singular dimensions of an array:

```
In [28]: F = squeeze(E)

In [29]: F
Out[29]: array([0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3])
```

```
In [30]: F.ndim
Out[30]: 1

In [31]: F.shape
Out[31]: (12,)

In [53]: oneelem = array([[5]])    # A matrix with one single element

In [54]: oneelem
Out[54]: array([[5]])

In [55]: oneelem.ndim
Out[55]: 2

In [56]: oneelem.shape
Out[56]: (1, 1)

In [57]: scalar = squeeze(oneelem)

In [58]: scalar.shape
Out[58]: ()

In [59]: scalar.ndim
Out[59]: 0
```

Finally there is the important *transpose* operation which acts as a transpose in the mathematical sense:

```
In [61]: B
Out[61]:
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])

In [70]: B.shape
Out[70]: (3, 4)

In [62]: transpose(B)
Out[62]:
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2],
       [3, 3, 3]])

In [68]: TB = transpose(B)

In [69]: TB.shape
Out[69]: (4, 3)
```

It’s important to note that we can only transpose arrays of dimension larger than 1.

```
In [63]: A
Out[63]: array([0, 1, 2, 3])

In [65]: A.shape
Out[65]: (4,)

In [66]: TA = transpose(A)

In [71]: TA
```

```
Out[71]: array([0, 1, 2, 3])

In [67]: TA.shape
Out[67]: (4,)
```

An option would be to reshape `A` to `(1,4)` and then take the transpose. Because this is a very common operation there is a small helper function called `atleast_2d` which does the job:

```
In [72]: TA2 = transpose(atleast_2d(A))

In [73]: TA2
Out[73]:
array([[0],
       [1],
       [2],
       [3]])

In [74]: TA2.shape
Out[74]: (4, 1)
```

An overview of many more useful methods to manipulate arrays as a whole is given here:

> http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

1.4.5 Operation on Arrays

Until now we have only worked with arrays as structured data containers. Usually we want to work with the the data inside too.

Arrays support all basic arithmetic operation out of the box:

```
In [113]: a = arange(4)

In [114]: a
Out[114]: array([0, 1, 2, 3])

In [115]: a+a
Out[115]: array([0, 2, 4, 6])

In [116]: a-a
Out[116]: array([0, 0, 0, 0])

In [117]: a*a
Out[117]: array([0, 1, 4, 9])

In [118]: a/a
/usr/bin/ipython:1: RuntimeWarning: divide by zero encountered in divide
  #!/usr/bin/python
Out[118]: array([0, 1, 1, 1])
```

Note that we got a warning for the division by zero but nevertheless obtained a valid array.

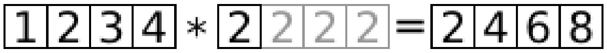
Remember the computation of squares from above? Now we know how to do this in the most simple way:

```
In [184]: arange(10)**2
Out[184]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

In these examples all arrays were of matching shape. But what happens if this is not the case?

Broadcasting

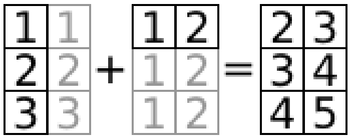
If we try to perform some operation where the shapes of the operands do not match, NumPy still tries to do some computation if possible. The method applied to resolve the issue is called *broadcasting* and shown in the following pictures. First a simple example, we want to multiply the array a by a scalar number:



NumPy now makes an array of the same shape of a by repeating the element as many times as necessary (gray boxes). (Of course internally no memory is wasted and this second array never constructed explicitly.)

```
In [121]: a = arange(1,5)

In [122]: a
Out[122]: array([1, 2, 3, 4])

In [123]: a.shape
Out[123]: (4,)

In [124]: a*2
Out[124]: array([2, 4, 6, 8])

In [125]: b = a*2

In [126]: b.shape
Out[126]: (4,)
```

As a second example we add a row and a column vector of different length like shown in this figure:



Let’s try to do this with NumPy arrays now:

```
In [145]: a = arange(1,4).reshape((3,1))

In [146]: a
Out[146]:
array([[1],
       [2],
       [3]])

In [147]: a.shape
Out[147]: (3, 1)

In [148]: b = array([[1,2]])

In [149]: b
Out[149]: array([[1, 2]])

In [150]: b.shape
Out[150]: (1, 2)

In [151]: c = a+b

In [152]: c
Out[152]:
```

```
array([[2, 3],
       [3, 4],
       [4, 5]])

In [153]: c.shape
Out[153]: (3, 2)
```

Some more explanations and examples of broadcasting can be found in the user guide:

> http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

The exact details when broadcasting is applied are stated in the reference manual:

> http://docs.scipy.org/doc/numpy/reference/ufuncs.html#broadcasting

Universal functions

NumPy provides a rich set of functions that operate on the elements of an array. These functions are called *ufuncs* (short for universal functions) and the manual says:

> Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs.

where the *core function* can be for example some math function like `sin`:

```
In [81]: x = linspace(0, 2*pi, 10)

In [82]: x
Out[82]:
array([ 0.        ,  0.6981317 ,  1.3962634 ,  2.0943951 ,  2.7925268 ,
        3.4906585 ,  4.1887902 ,  4.88692191,  5.58505361,  6.28318531])

In [83]: y = sin(x)

In [84]: y
Out[84]:
array([  0.00000000e+00,   6.42787610e-01,   9.84807753e-01,
         8.66025404e-01,   3.42020143e-01,  -3.42020143e-01,
        -8.66025404e-01,  -9.84807753e-01,  -6.42787610e-01,
        -2.44929360e-16])
```

In the introduction we said that we don't care about data types too much. There are some exceptions like the following example shows:

```
In [104]: o = -ones((2,))

In [105]: o
Out[105]: array([-1., -1.])

In [106]: sqrt(o)
Out[106]: array([ nan,  nan])

In [107]: oc = -ones((2,), dtype=complexfloating)

In [108]: oc
Out[108]: array([-1.-0.j, -1.-0.j])

In [109]: sqrt(oc)
Out[109]: array([ 0.-1.j,  0.-1.j])
```

```
In [110]: o.dtype
Out[110]: dtype('float64')

In [111]: oc.dtype
Out[111]: dtype('complex128')
```

A good overview of the many available ufucs can be found at:

> http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs

More information about universal functions in general:

> http://docs.scipy.org/doc/numpy/reference/ufuncs.html

1.4.6 Basic linear algebra

All functions we know by now operate element-wise on arrays. For linear algebra we need scalar, matrix-vector and matrix-matrix products. The function for all three is called `dot` and takes two operands:

```
In [164]: A = array([[1,2],[3,4]])

In [165]: v = array([5,6])

In [166]: dot(v, v)
Out[166]: 61

In [167]: dot(A, v)
Out[167]: array([17, 39])

In [168]: dot(A, A)
Out[168]:
array([[ 7, 10],
       [15, 22]])
```

In this example the vector has a shape of `(2,)`. For one-dimensional arrays as vectors everything is fine (remember that transpose has no effect for one-dimensional arrays). We can also work with (two-dimensional) explicit column vectors:

```
In [172]: v = v.reshape((2,1))

In [173]: v.shape
Out[173]: (2, 1)

In [174]: A.shape
Out[174]: (2, 2)
```

This time the dot product failes because the shapes do not match:

```
In [175]: dot(v, v)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-175-093f352c32e0> in <module>()
----> 1 dot(v, v)

ValueError: objects are not aligned
```

This is a very typical error message. Whenever you see this you should go and check the shapes of your arrays again.

We have to take the transpose of the first argument, equivalent to the mathematical formulation $a^T a$:

```
In [176]: dot(transpose(v), v)
Out[176]: array([[61]])
```

Note that the result is still a two-dimensional array of shape `(1, 1)`. If this is inconvenient we could use `squeeze` on the result. We can even built outer products aa^T this way:

```
In [179]: dot(v, transpose(v))
Out[179]:
array([[25, 30],
       [30, 36]])
```

In priciple this is all very straight-forward, we just have to be careful enough with the shapes of all operands.

1.4.7 Further information on NumPy

- NumPy User Guide
- NumPy Reference Manual

1.5 First introduction to SciPy

The second very important python package we look at is SciPy. The SciPy documentation says:

> SciPy is a collection of mathematical algorithms and convenience functions built on the Numpy extension for Python. It adds significant power to the interactive Python session by exposing the user to high-level commands and classes for the manipulation and visualization of data.

You can read a longer informal introduction at:

> http://docs.scipy.org/doc/scipy/reference/tutorial/general.html

The home of the SciPy python package is:

> http://www.scipy.org

In the following we will make some examples for a few selected parts of SciPy. But SciPy is a huge library and we can not make an example or point you to every useful function from it. There also is no good reason to duplicate its documentation here. If you encounter a (scientific programming) problem you should go to the library reference:

> http://docs.scipy.org/doc/scipy/reference/#reference

and search for the tools you need. The documentation is very good and well structured.

1.5.1 Linear Algebra

At the end of the last chapter we only looked at the very basics of linear algebra. In this section we show some more features.

First we have to import the `linalg` submodule of `scipy`

```
In [1]: from numpy import *

In [2]: from scipy.linalg import *
```

Let's do a first computation:

```
In [5]: A = array([[1,2,3],[4,5,6],[7,8,8]])

In [6]: det(A)
Out[6]: 3.0
```

next we can do a LU decomposition of `A`:

```
In [11]: P, L, U = lu(A)

In [12]: L
Out[12]:
array([[ 1.        ,  0.        ,  0.        ],
       [ 0.14285714,  1.        ,  0.        ],
       [ 0.57142857,  0.5       ,  1.        ]])

In [13]: U
Out[13]:
array([[ 7.        ,  8.        ,  8.        ],
       [ 0.        ,  0.85714286,  1.85714286],
       [ 0.        ,  0.        ,  0.5       ]])

In [14]: dot(L,U)
Out[14]:
array([[ 7.,  8.,  8.],
       [ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])

In [15]: dot(P,dot(L,U))
Out[15]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8.,  8.]])
```

We can find out the eigenvalues and eigenvectors of this matrix:

```
In [17]: EW, EV = eig(A)

In [18]: EW
Out[18]: array([ 15.55528261+0.j,  -1.41940876+0.j,  -0.13587385+0.j])

In [19]: EV
Out[19]:
array([[-0.24043423, -0.67468642,  0.51853459],
       [-0.54694322, -0.23391616, -0.78895962],
       [-0.80190056,  0.70005819,  0.32964312]])

In [20]: dot(A, EV[:,0])
Out[20]: array([ -3.74002233,  -8.50785632, -12.47378976])

In [21]: n = dot(A, EV[:,0]) - EW[0]*EV[:,0]

In [22]: n
Out[22]: array([ -3.55271368e-15+0.j,  -1.24344979e-14+0.j,  -8.88178420e-15+0.j])

In [23]: norm(n)
Out[23]: 1.5688358818848954e-14
```

Indeed, we found the eigenvalues and eigenvectors.

Solving systems of linear equations is not more difficult:

```
In [26]: v = array([[2,3,5]])

In [27]: s = solve(A, transpose(v))

In [28]: s
Out[28]:
array([[-2.33333333],
       [ 3.66666667],
       [-1.        ]])
```

We can check the solution:

```
In [29]: dot(A, s) - v
Out[29]:
array([[  4.44089210e-16,  -1.00000000e+00,  -3.00000000e+00],
       [  1.00000000e+00,   8.88178420e-16,  -2.00000000e+00],
       [  3.00000000e+00,   2.00000000e+00,   3.55271368e-15]])
```

Oups, broadcasting happened! We have to transpose the vector `v` (or squeeze the result of `dot`):

```
In [30]: dot(A, s) - transpose(v)
Out[30]:
array([[  4.44089210e-16],
       [  8.88178420e-16],
       [  3.55271368e-15]])
```

If you want you can read on here

> http://docs.scipy.org/doc/scipy/reference/tutorial/linalg.html

to see some more examples on how to do linear algebra with SciPy.

For a complete overview of the (dense) linear algebra functionalities of SciPy please see:

> http://docs.scipy.org/doc/scipy/reference/linalg.html

1.5.2 Sparse linear algebra

SciPy has some routines for computing with sparse and potentially very large matrices. The necessary tools are in the submodule `scipy.sparse`.

We make one example on how to construct a large matrix:

```
In [1]: from scipy.sparse import *

In [2]: from scipy import rand

In [3]: A = lil_matrix((1000, 1000))

In [4]: A[0,:100] = rand(100)

In [5]: A[1,100:200] = A[0,:100]

In [6]: A.setdiag(rand(1000))

In [7]: A
Out[7]:
<1000x1000 sparse matrix of type '<type 'numpy.float64'>'
        with 1199 stored elements in LInked List format>
```

Next we solve a linear equation:

```
In [8]: from scipy.sparse.linalg import *

In [9]: A = A.tocsr()

Out[10]:
<1000x1000 sparse matrix of type '<type 'numpy.float64'>'
        with 1199 stored elements in Compressed Sparse Row format>

In [11]: b = rand(1000)

In [12]: x = spsolve(A, b)
```

The two largest and smallest eigenvalues of `A`:

```
In [26]: EWs, EVs = eigs(A, 2, which="SM")

In [27]: EWs
Out[27]: array([ 0.00107174+0.j,  0.00211868+0.j])

In [28]: EWl, EVl = eigs(A, 2, which="LM")

In [29]: EWl
Out[29]: array([ 0.99854009+0.j,  0.99786107+0.j])
```

For further information please see the reference for sparse matrices:

> http://docs.scipy.org/doc/scipy/reference/sparse.html

and sparse linear algebra:

> http://docs.scipy.org/doc/scipy/reference/sparse.linalg.html

1.5.3 Fourier Transforms

There are plenty of different Fourier and related transformations available.

Some function are available in the `numpy.fft` submodue:

> http://docs.scipy.org/doc/numpy/reference/routines.fft.html

others in the `scipy.fftpack` submodule:

> http://docs.scipy.org/doc/scipy/reference/fftpack.html

It's a bit unfortunate that some of these functions live in NumPy rather than SciPy.

1.5.4 Optimization and root finding

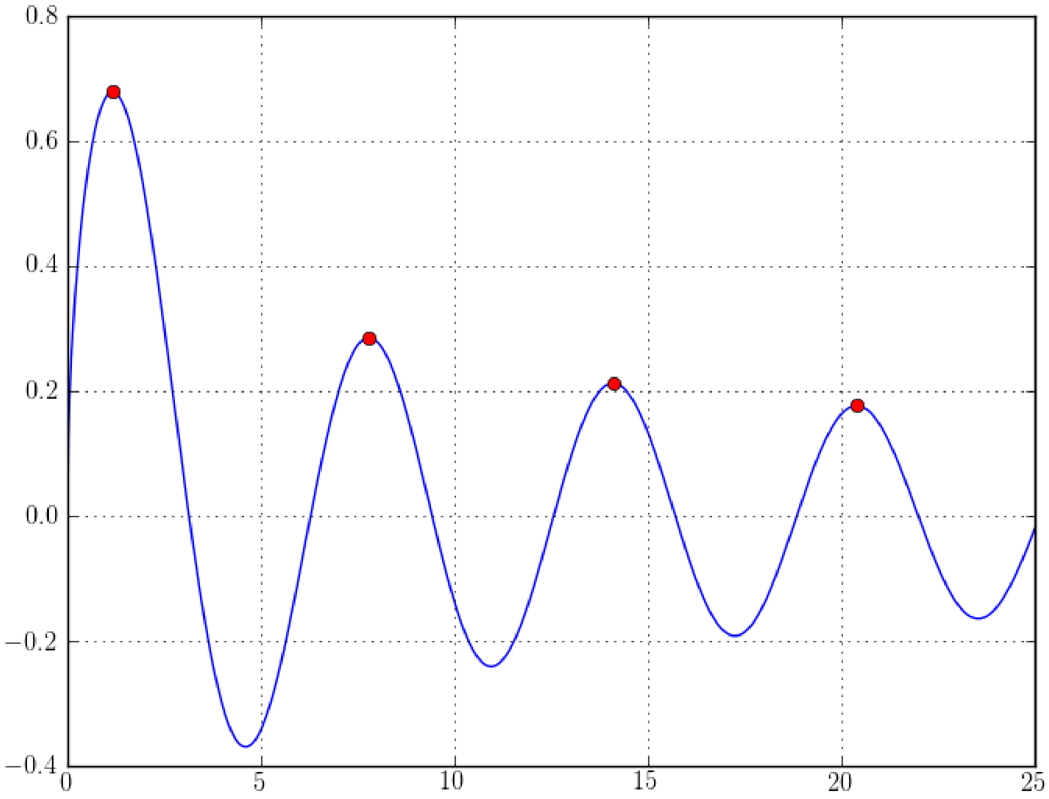
SciPy has functions for univariate and multivariate as well as unconstrained and constrained optimization techniques.

As an example we compute the first 4 maxima of the Bessel function $J_{\frac{1}{2}}(x)$ given an initial guess for each one and a bounded interval to search in:

```
In [1]: from scipy.special import jv     # Import the Bessel J_n special function

In [2]: from scipy.optimize import fminbound

In [3]: approx = [1, 6, 12, 18]
In [4]: for guess in approx:
   ...:     f = lambda x: -jv(0.5, x)
   ...:     xmax = fminbound(f, guess-4, guess+4)
   ...:     print(xmax)
   ...:
```

```
1.16556123601

7.78988391603

14.1017257582

20.3958424877
```

This specific Bessel function looks like:



Consult the documentation for more details on the various different routines of the `scipy.optimize` submodule:

> http://docs.scipy.org/doc/scipy/reference/optimize.html

1.5.5 Other important submodules of SciPy

- http://docs.scipy.org/doc/scipy/reference/special.html All the ususal higher special functions from mathematics and physics. Almost all of them are implemented as universal functions operating on NumPy arrays.

- http://docs.scipy.org/doc/scipy/reference/stats.html Large collection statistical functions and random distributions.
- http://docs.scipy.org/doc/scipy/reference/constants.html Physical and mathematical constants and units. The values are taken from the "CODATA Recommended Values of the Fundamental Physical Constants 2010".
- http://docs.scipy.org/doc/scipy/reference/integrate.html Some functions for numerical integration and methods for solving ODEs.

1.5.6 Further information on SciPy

There are many other submodules of SciPy and we have no place to introduce all. (Additionally not all are relevant for our purpose.) Anyway, if you are interested or need more information, look into the reference guide:

- Scipy Reference Guide

1.6 First introduction to Matplotlib

The last big topic to look at is plotting. Thanks to a very nice plotting library called `matplotlib` plotting of (scientific) data from within python has become almost trivial.

The webpage of Matplotlib is located at:

> http://matplotlib.org/index.html

Matplotlib is mainly for two-dimensional plots (although there are some limited capabilities for 3 dimensional stuff.) We will concentrate on 2D and only show a few very basic examples.

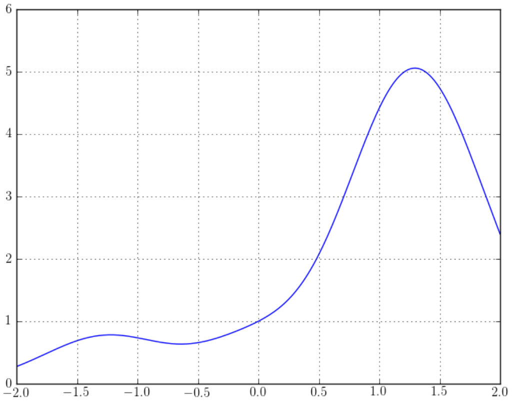
Matplotlib can do a few things more than just plotting. We will use only one submodule, namely `matplotlib.pyplot`. This is what we need to import before we can do any plotting.

1.6.1 First steps

Now let start with plotting and make a first simple example:

```
In [1]: from numpy import *

In [2]: from matplotlib.pyplot import *

In [3]: x = linspace(-2, 2, 200)

In [4]: y = (x**5 + 4*x**4 + 3*x**3 + 2*x**2 + x + 1)*exp(-x**2)

In [5]: figure()    # Make a new figure
Out[5]: <matplotlib.figure.Figure at 0x3636250>

In [6]: plot(x, y)    # Plot some data
Out[6]: [<matplotlib.lines.Line2D at 0x36e5550>]

In [7]: grid(True)    # Set the thin grid lines

In [8]: savefig("plot_1.png")    # Save the figure to a file "png",
                                 # "pdf", "eps" and some more file types are valid
```

And the result then looks like:



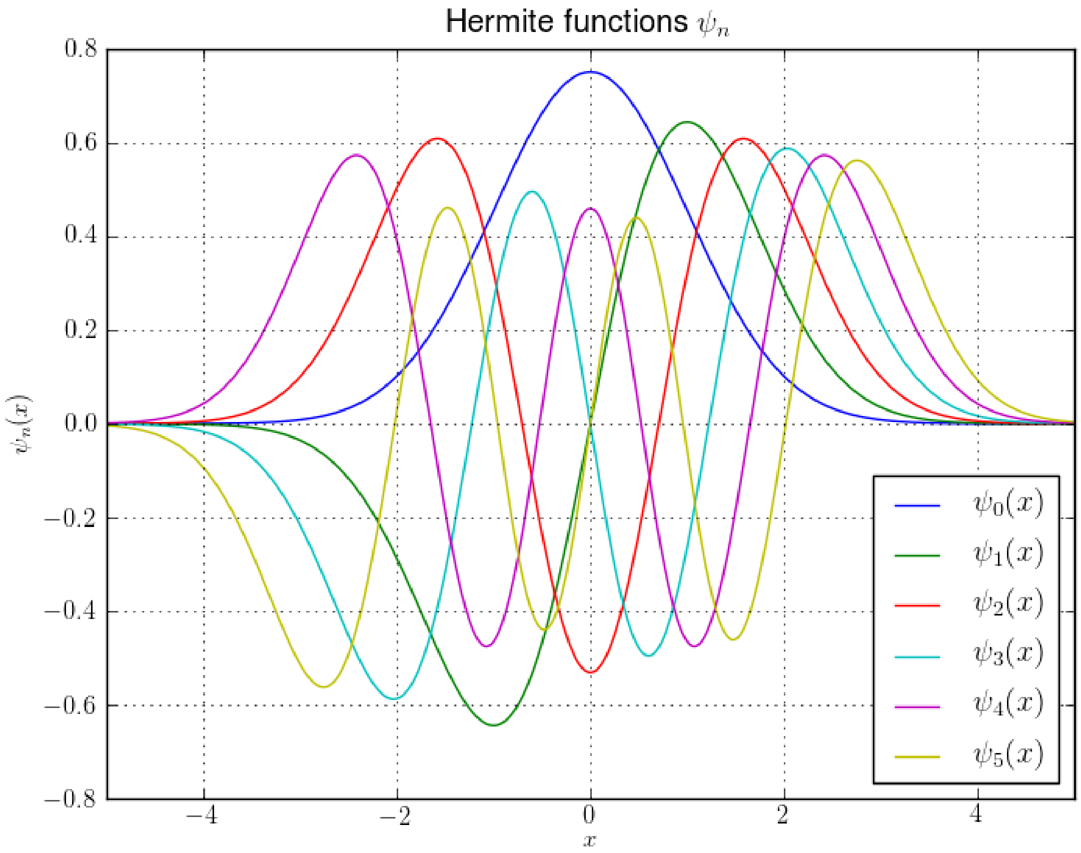
With only a couple of lines of python we get such a nice plot!

Additionally to saving the figure to a file we can also show it right from the python interpreter by using the command `show()` instead of `savefig`.

Now there are many options available to change the look or other features of the resulting figure. Usually most have reasonable defaults. In the next example we show some useful things:

```
In [9]: from scipy.special import gamma
In [10]: from scipy.special.orthogonal import eval_hermite

In [11]: x = linspace(-5, 5, 1000)

In [12]: psi = lambda n,x: 1.0/sqrt((2**n*gamma(n+1)*sqrt(pi))) * exp(-x**2/2.0) * eval_hermite(n, x

In [13]: figure()
Out[13]: <matplotlib.figure.Figure at 0x3a9be50>

In [14]: for n in xrange(6):
   ....:     plot(x, psi(n,x), label=r"$\psi_"+str(n)+r"(x)$")    # The ’label’ to put into the legen
   ....:

In [15]: grid(True)

In [16]: xlim(-5,5)    # Specify the range of the x axis shown
Out[16]: (-5, 5)

In [17]: ylim(-0.8,0.8)    # Specify the range of the y axis shown
Out[17]: (-0.8, 0.8)

In [18]: xlabel(r"$x$")    # Put a label on the x axis
Out[18]: <matplotlib.text.Text at 0x3d06210>

In [19]: ylabel(r"$\psi_n(x)$")    # Put a label on the y axis
Out[19]: <matplotlib.text.Text at 0x3d0a190>

In [20]: legend(loc="lower right")    # Add a legend table
Out[20]: <matplotlib.legend.Legend at 0x3d32e50>

In [21]: title(r"Hermite functions $\psi_n$")    # And a title on top of the figure
Out[21]: <matplotlib.text.Text at 0x3d11590>

In [22]: savefig("plot_2.png")
```

Which results in:

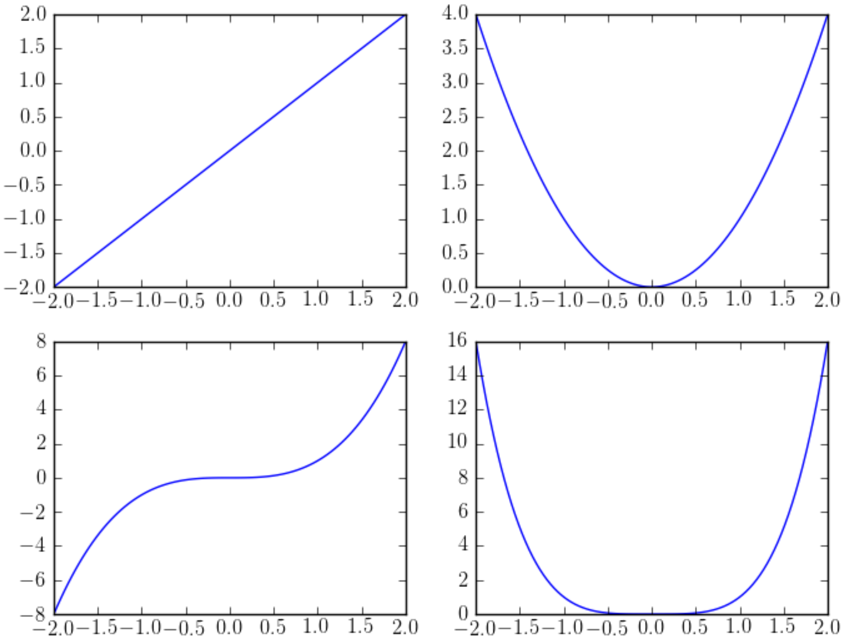


We can easily make a plot that contains several so called *subplots*

```
In [3]: x = linspace(-2, 2, 100)

In [4]: figure()
Out[4]: <matplotlib.figure.Figure at 0x362b350>

In [5]: subplot(2,2,1)    # Two rows and two cols of plots, select first one
Out[5]: <matplotlib.axes.AxesSubplot at 0x362ba90>

In [6]: plot(x, x)
Out[6]: [<matplotlib.lines.Line2D at 0x27b2cd0>]

In [7]: subplot(2,2,2)    # Two rows and two cols of plots, select second one
Out[7]: <matplotlib.axes.AxesSubplot at 0x36d7b10>

In [8]: plot(x, x**2)
Out[8]: [<matplotlib.lines.Line2D at 0x38937d0>]

In [9]: subplot(2,2,3)    # Two rows and two cols of plots, select third one
Out[9]: <matplotlib.axes.AxesSubplot at 0x3893c90>

In [10]: plot(x, x**3)
```

```
Out[10]: [<matplotlib.lines.Line2D at 0x38b8a10>]

In [11]: subplot(2,2,4)    # Two rows and two cols of plots, select fourth one
Out[11]: <matplotlib.axes.AxesSubplot at 0x38b8e10>

In [12]: plot(x, x**4)
Out[12]: [<matplotlib.lines.Line2D at 0x362b8d0>]

In [13]: savefig("plot_subplots.png")
```

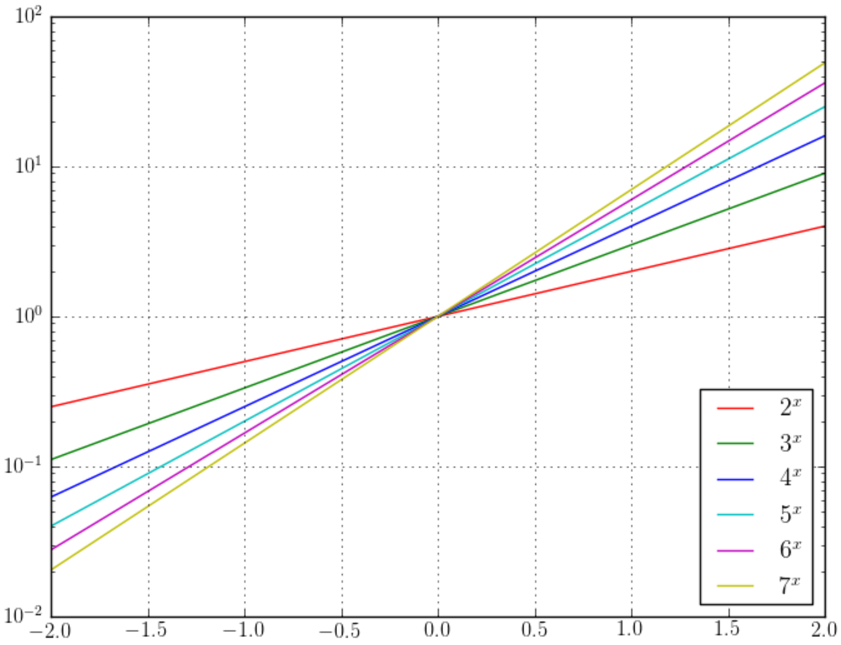
and we get the following image



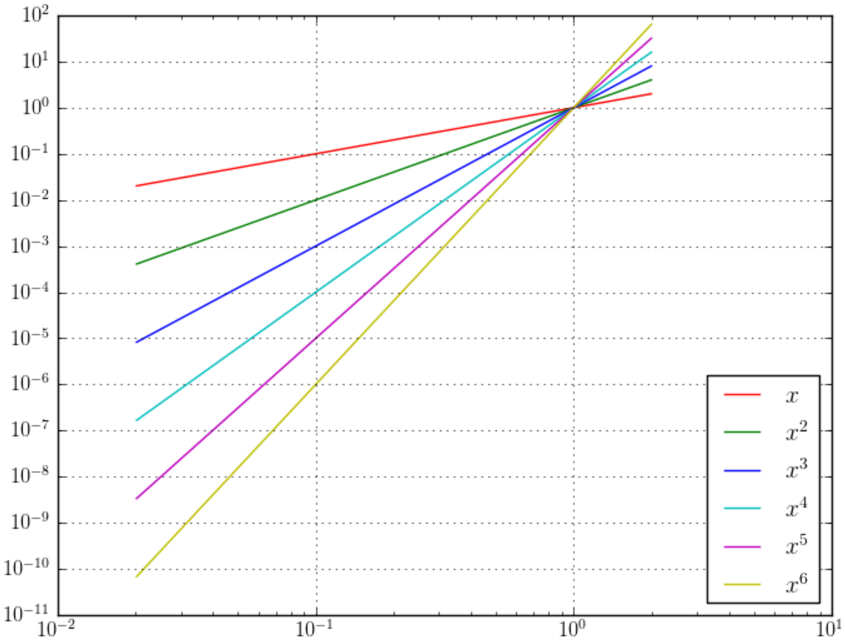
We can make one (command `semilogx`, `semilogy`) or both (command `loglog`) axes log scale

```
In [70]: figure()
Out[70]: <matplotlib.figure.Figure at 0x3f28850>

In [71]: semilogy(x, 2**x, "r-", label=r"$2^x$")
Out[71]: [<matplotlib.lines.Line2D at 0x54c9410>]

In [72]: semilogy(x, 3**x, "g-", label=r"$3^x$")
Out[72]: [<matplotlib.lines.Line2D at 0x54cf090>]

In [73]: semilogy(x, 4**x, "b-", label=r"$4^x$")
Out[73]: [<matplotlib.lines.Line2D at 0x54bf410>]

In [74]: semilogy(x, 5**x, "c-", label=r"$5^x$")
Out[74]: [<matplotlib.lines.Line2D at 0x5690f50>]

In [75]: semilogy(x, 6**x, "m-", label=r"$6^x$")
Out[75]: [<matplotlib.lines.Line2D at 0x5690f10>]
```

```
In [76]: semilogy(x, 7**x, "y-", label=r"$7^x$")
Out[76]: [<matplotlib.lines.Line2D at 0x5690e50>]

In [77]: grid(True)

In [78]: legend(loc="lower right")
Out[78]: <matplotlib.legend.Legend at 0x54d9fd0>

In [79]: savefig("plot_semilogy.png")
```



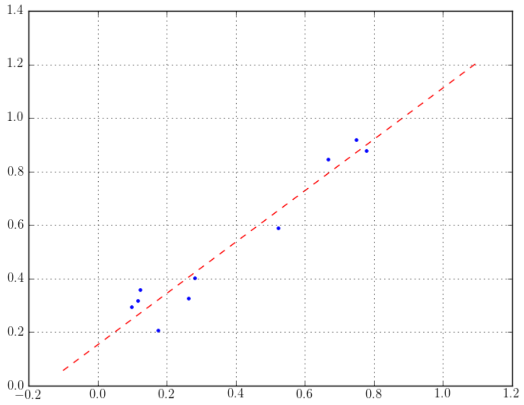
```
In [35]: figure()
Out[35]: <matplotlib.figure.Figure at 0x44f9510>

In [36]: loglog(x, x, "r-", label=r"$x$")
Out[36]: [<matplotlib.lines.Line2D at 0x47f9310>]

In [37]: loglog(x, x**2, "g-", label=r"$x^2$")
Out[37]: [<matplotlib.lines.Line2D at 0x480ce90>]

In [38]: loglog(x, x**3, "b-", label=r"$x^3$")
Out[38]: [<matplotlib.lines.Line2D at 0x480ced0>]

In [39]: loglog(x, x**4, "c-", label=r"$x^4$")
Out[39]: [<matplotlib.lines.Line2D at 0x480f250>]

In [40]: loglog(x, x**5, "m-", label=r"$x^5$")
Out[40]: [<matplotlib.lines.Line2D at 0x480f350>]

In [41]: loglog(x, x**6, "y-", label=r"$x^6$")
Out[41]: [<matplotlib.lines.Line2D at 0x480fe50>]

In [42]: grid(True)
In [43]: legend(loc="lower right")
Out[43]: <matplotlib.legend.Legend at 0x4521710>

In [44]: savefig("plot_loglog.png")
```



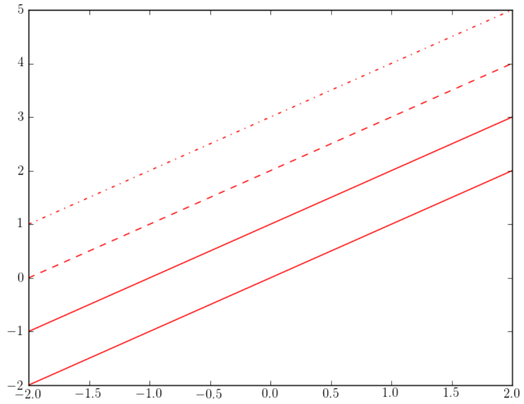
Log scale plots are commonly used for convergence plots as we will see later.

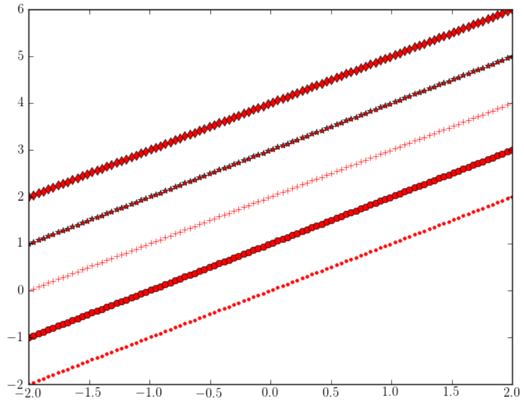
Often one wants to fit some polynomials and plot the result. This is what the following example does. (For fitting in log scaled plots one has to apply `log` manually before doing a `polyfit`. The result has to be transformed back again.)

```
In [6]: x = random.rand(10)       # Some random points
In [7]: offset = random.rand(10)
In [8]: y = x + 0.25*offset

In [9]: p = polyfit(x, y, 1)    # Fit a polynomial of first degree (line)

In [10]: u = linspace(-0.1, 1.1, 10)    # Evaluate the polynomial at (other) points
In [11]: v = polyval(p, u)

In [12]: figure()
Out[12]: <matplotlib.figure.Figure at 0x2e87f10>
In [13]: plot(x, y, ".")
Out[13]: [<matplotlib.lines.Line2D at 0x334ac10>]
In [14]: plot(u, v, "--r")
Out[14]: [<matplotlib.lines.Line2D at 0x3271e10>]
In [15]: grid(True)
In [16]: show()
```



Matplotlib knows many different line and marker styles. The following plot show only a small subset of what's available.



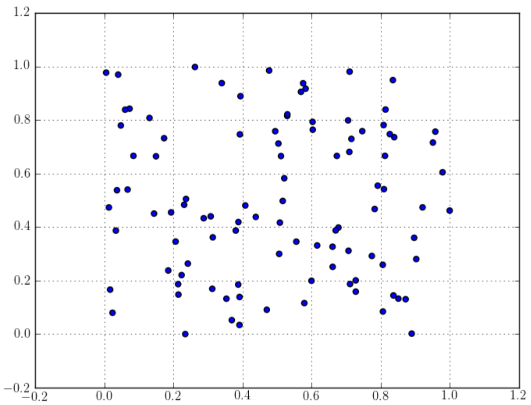


For some more examples see http://matplotlib.org/examples/pylab_examples/line_styles.html and http://matplotlib.org/examples/pylab_examples/filledmarker_demo.html and the reference plots:

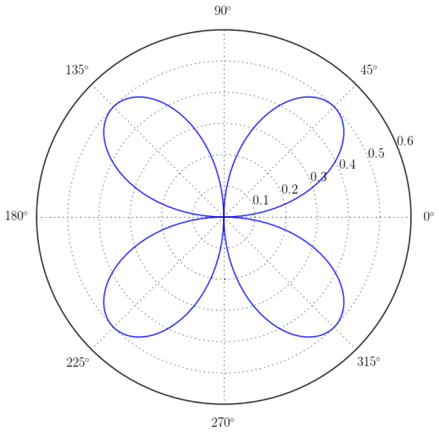
- Line Styles
- Markers

There are several other plot types too, for example scatter plots and polar plots

```
In [141]: x = random.rand(100)

In [142]: y = random.rand(100)

In [143]: figure()
Out[143]: <matplotlib.figure.Figure at 0x5df78d0>

In [144]: scatter(x, y)
Out[144]: <matplotlib.collections.CircleCollection at 0x6794b10>

In [145]: grid(True)

In [146]: savefig("plot_scatter.png")
```

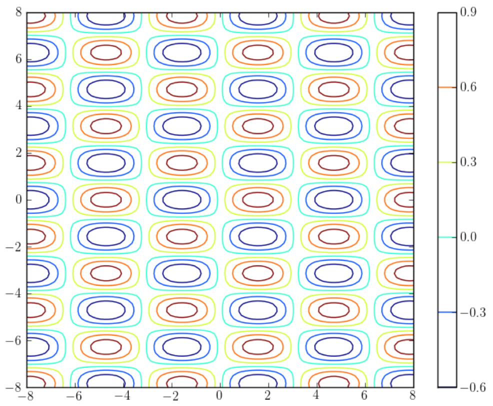


```
In [148]: phi = linspace(0, 2*pi, 200)

In [149]: figure()
Out[149]: <matplotlib.figure.Figure at 0x5cbda50>

In [150]: polar(phi, sin(phi)*cos(phi))
Out[150]: [<matplotlib.lines.Line2D at 0x67a29d0>]

In [151]: grid(True)

In [152]: savefig("plot_polar.png")
```



A simple contour plot of an implicit function

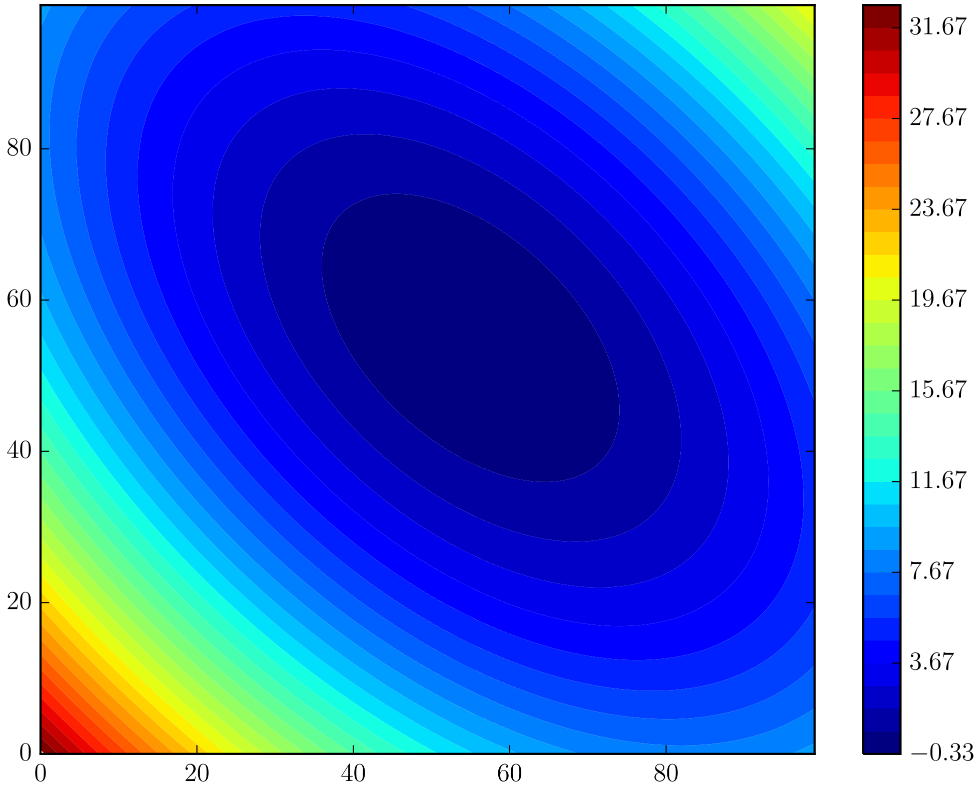
```
In [2]: x = linspace(-8,8,100)

In [3]: y = linspace(-8,8,100)

In [4]: X, Y = meshgrid(x, y)

In [5]: f = lambda u,v: sin(u)*cos(2*v)+0.1

In [6]: Z = f(X, Y)

In [7]: contour(X,Y,Z)    # Plot the contour lines
Out[7]: <matplotlib.contour.QuadContourSet instance at 0x4948b48>

In [8]: colorbar()    # Make a legend for the colors
Out[8]: <matplotlib.colorbar.Colorbar instance at 0x49574d0>

In [9]: show()
```



Matplotlib knows many more plot types like advanced contour and density plots, matrix and discrete plots, stem plots, stream lines, vector fields, various types of charts and histograms as well as some (limited) 3D plotting.

To end this chapter we show a somehwat longer contour plot example

```
from numpy import *
from matplotlib.pyplot import *

# Data of the quadratic form
A = array([
        [2., 1.],
        [1., 2.]])

b = array([1., 1.])

# Mesh
Nx = 100
Ny = 100

X, Y = ogrid[-3.:3:Nx*1j, -3.:3:Ny*1j]   # ogrid is similar to meshgrid

z = zeros(2)
Z = zeros(Nx*Ny)

# compute values
k = 0
for x in X[:,0]:
    z[0] = x
    for y in Y[0,:]:
         z[1] = y
         Z[k] = 0.5*dot(z,dot(A,z)) - dot(z,b)
         k += 1

Z = Z.reshape(Nx,Ny)

# Plot
levels = arange(Z.min(), Z.max()+0.01, 1.0)   # How many contour levels and where
colormap = cm.get_cmap('jet', len(levels)-1)   # Add a nice color map

figure()
contourf(Z, levels, cmap=colormap)
colorbar()   # Add the color bar legend
show()
```



The official tutorial is also quite a good introduction. You should continue reading:

http://matplotlib.org/users/pyplot_tutorial.html

if you want to know more right now. Another very good tutorial with many examples is this one:

http://www.labri.fr/perso/nrougier/teaching/matplotlib/

1.6.2 The Matplotlib Gallery

It is always a good idea to look at the Matplotlib gallery

http://matplotlib.org/gallery.html

when you intend to do a new kind of plot and are not sure how to continue best. The gallery contains many examples with full source code of almost every plot imaginable. You might get either your ploting program already tipped or good visualization ideas. There are also two less prominent spots to look for examples:

http://matplotlib.org/users/screenshots.html

http://matplotlib.org/examples/index.html

1.6.3 Further information on Matplotlib

- User's Guide

1.7 IPython tips

IPython is an interactive computing environment; not only it is an enhanced interactive Python shell, but it is also an architecture for interactive parallel computing and provides (in recent versions) a very nice notebook like interface.

In case you use the pythonxy distribution under Windows, it is a good idea to start the IPython with mlab-mode enabled; you find several modes in the menu-entry "enhanced console" of pythonxy.

For a good introduction to IPython, you should read the tutorial on using the interactive features:

> http://ipython.org/ipython-doc/stable/interactive/tutorial.html

We resume here the commands that we use frequently in our work. One of the most important features is **TAB-completion**, you can start typing any function or variable name and by pressing <TAB> obtain a list of suggestions or the unique match for completion.

```
In [1]: from scipy import linalg

In [2]: linalg.<TAB>
```

where <TAB> refers to the TAB-key. You should see something like:

```
In [2]: linalg.
linalg.LinAlgError       linalg.clapack           linalg.eigvals
linalg.Tester            linalg.companion         linalg.eigvals_banded
linalg.all_mat           linalg.coshm             linalg.eigvalsh
linalg.basic             linalg.cosm              linalg.expm
linalg.bench             linalg.decomp            linalg.expm2
linalg.blas              linalg.decomp_cholesky   linalg.expm3
linalg.block_diag        linalg.decomp_lu         linalg.fblas
linalg.calc_lwork        linalg.decomp_qr         linalg.flapack
linalg.cblas             linalg.decomp_schur      linalg.flinalg
linalg.cho_factor        linalg.decomp_svd        linalg.funm
linalg.cho_solve         linalg.det               linalg.get_blas_funcs
linalg.cho_solve_banded  linalg.diagsvd           linalg.hadamard
linalg.cholesky          linalg.eig               linalg.hankel
linalg.cholesky_banded   linalg.eig_banded        linalg.hessenberg
linalg.circulant         linalg.eigh              linalg.hilbert
[...]
```

IPython examined the linalg submodule, and returned all the possible completions. Once you see an interesting function, you'd like to know how to use it:

```
In[3]: linalg.eig?<ENTER>
```

In order to view the actual source code, use two question marks instead of one.

The magic commands (all commands starting by a `%` sign are called magic because they are *not* python code but ipython specific control codes.) `%pdoc`, `%pdef`, `%psource` and `%pfile` will respectively print the docstring, function definition line, full source code and the complete file for any object (when they can be found). If automagic is on (it is by default), you don't need to type the `%` explicitly:

```
np.linalg.svd?
pdoc np.linalg.svd
pdef np.linalg.svd
psource np.linalg.svd
```

All output results are automatically stored in a global dictionary named `Out` and variables named `_1`, `_2`, etc. alias them. For example, the result of input line 4 is available either as `Out[4]` or as `_4`. The last three objects in output history are also kept in variables named _, __ and ___.

Put a `;` at the end of a line to suppress the printing of output. The `_*` variables and the `Out[*]` list do get updated with the contents of the output, even if it is not printed. You can thus still access the generated results this way for further processing.

The `%history` command can show you all previous input, without line numbers if desired (option -n) so you can directly copy and paste code either back in IPython or in a text editor. You can also save all your history by turning on logging via `%logstart`; these logs can later be either reloaded as IPython sessions or used as code for your programs.

Related to the history:

- Up and down arrows (Ctrl-p/Ctrl-n)
- PgUp and PgDn keys
- Search: Ctrl-r and start typing
- Ctrl-a or `Home` key: go to start of line
- Ctrl-e or `End` key: end of line
- Ctrl-k: kill to end of line
- Ctrl-L: clear screen

Consider now the files sattelite.py that implements the simplified trajectory of a sattelite:

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

op = 2*np.pi; os = 14.*np.pi
R = 4.; r = 0.25
t = np.linspace(0, 2, 2000)
xp = np.cos(op*t)
yp = np.sin(op*t)
xs = xp + r*np.cos(os*t)
ys = yp + r*np.sin(os*t)
plt.plot(xs,ys); plt.show()
```

The `%run` magic command:

```
run sattelite.py
run -t sattelite.py
run -p sattelite.py
run -d sattelite.py
```

`%run` also has special flags for timing the execution of your scripts (-t) and for executing them under the control of either Python's pdb debugger (-d) or profiler (-p). With all of these, `%run` can be used as the main tool for efficient interactive development of code which you write in your editor of choice.

IPython is a command line-oriented program, without full control of the terminal. Therefore, it doesn't fully support true multiline editing. However, it has a number of useful tools to help you in dealing effectively with more complex editing.:

```
%edit 10-20 24 28 # opens an editor with these lines pre-loaded for modification
%save <filename> # saves the lines directly to a named file on disk
```

Here comes an example showing the working flow with python.

Suppose we want to plot the tajectory of a sattelite of the earth as seen from the sun. After thinking about the mathematics of the problem, we might start by trying an implementation interactively in ipython:

```
[gradinar@localhost TutCodes]$ ipython
Python 2.6.4 (r264:75706, Jul 14 2010, 09:36:06)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: import numpy as np

In [2]: op = 2*np.pi; os = 14.*np.pi

In [3]: R = 4.; r = 2

In [4]: t = np.linspace(0, 2, 100)

In [5]: xp = np.cos(op*t)

In [6]: yp = np.sin(op*t)

In [7]: xs = xp + r*np.cos(os*t)

In [8]: ys = yp + r*np.sin(os*t)

In [9]: from matplotlib import pyplot as plt

In [10]: plt.plot(xs,ys); plt.show()
/usr/lib64/python2.6/site-packages/matplotlib/backends/backend_gtk.py:621: DeprecationWarning: Use th
self.tooltips = gtk.Tooltips()
Out[10]: [<matplotlib.lines.Line2D object at 0x7f5d8c822050>]
```

The result is far from satisfactory. Something went wrong with the radia:

```
In [11]: R = 4.; r = 0.25

In [12]: plt.plot(xs,ys); plt.show()
Out[12]: [<matplotlib.lines.Line2D object at 0x7f5d8c853510>]
```

Nothing changed? Of course, because we haven't recomputed the trajectories yet! We do not want to retype everything, so look at the history:

```
In [13]: hist
1 : import numpy as np
2 : op = 2*np.pi; os = 14.*np.pi
3 : R = 4.; r = 2
4 : t = np.linspace(0, 2, 100)
5 : xp = np.cos(op*t)
6 : yp = np.sin(op*t)
7 : xs = xp + r*np.cos(os*t)
8 : ys = yp + r*np.sin(os*t)
9 : from matplotlib import pyplot as plt
10: plt.plot(xs,ys); plt.show()
11: R = 4.; r = 0.25
12: plt.plot(xs,ys); plt.show()
13: _ip.magic("hist ")
```

OK, so lines 6 to 10 has to be repeated, but I forgot how it works:

```
In [14]: rep?
  Type:             Magic function
  Base Class:       <type 'instancemethod'>
  String Form:   <bound method InteractiveShell.rep_f of <IPython.iplib.InteractiveShell object at 0x
  Namespace:        IPython internal
  File:             /usr/lib/python2.6/site-packages/IPython/history.py
  Definition:       rep(self, arg)
  Docstring:
                   Repeat a command, or get command to input line for editing

                   - %rep (no arguments):

                   Place a string version of last computation result (stored in the special '_'
                   variable) to the next input prompt. Allows you to create elaborate command
                   lines without using copy-paste::

                       $ l = ["hei", "vaan"]
                       $ "".join(l)
                       ==> heivaan
                       $ %rep
                       $ heivaan_ <== cursor blinking

                   %rep 45

                   Place history line 45 to next input prompt. Use %hist to find out the
                   number.
```

Aha!

```
In [15]: rep 6-10
lines [u'yp = np.sin(op*t)\nxs = xp + r*np.cos(os*t)\nys = yp + r*np.sin(os*t)\nfrom matplotlib impor
Out[20]: [<matplotlib.lines.Line2D object at 0x7f5d8c629b90>]
```

Hmm, still not good, what if we take more points?

```
In [21]: t = np.linspace(0, 2, 2000)

In [22]: rep 6-10
lines [u'yp = np.sin(op*t)\nxs = xp + r*np.cos(os*t)\nys = yp + r*np.sin(os*t)\nfrom matplotlib impor
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

/home/gradinar/LaTeX_doc/Numcourses/Numcourses/SciPyTools/Start/TutCodes/<ipython console> in <module

ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Upps, one of the variables is longer. What was there?

```
In [25]: hist
1 : import numpy as np
2 : op = 2*np.pi; os = 14.*np.pi
3 : R = 4.; r = 2
4 : t = np.linspace(0, 2, 100)
5 : xp = np.cos(op*t)
6 : yp = np.sin(op*t)
7 : xs = xp + r*np.cos(os*t)
8 : ys = yp + r*np.sin(os*t)
9 : from matplotlib import pyplot as plt
10: plt.plot(xs,ys); plt.show()
```

```
11: R = 4.; r = 0.25
12: plt.plot(xs,ys); plt.show()
13: _ip.magic("hist ")
14: #?rep
15: _ip.magic("rep 6-10")
16: yp = np.sin(op*t)
17: xs = xp + r*np.cos(os*t)
18: ys = yp + r*np.sin(os*t)
19: from matplotlib import pyplot as plt
20: plt.plot(xs,ys); plt.show()
21: t = np.linspace(0, 2, 2000)
22: _ip.magic("rep 6-10")
23: yp = np.sin(op*t)
24: xs = xp + r*np.cos(os*t)
25: _ip.magic("hist ")
```

Aha, I rerun the wrong lines; let’s do it again:

```
In [26]: rep 5-10
lines [u’xp = np.cos(op*t)\nyp = np.sin(op*t)\nxs = xp + r*np.cos(os*t)\nys = yp + r*np.sin(os*t)\nfr
Out[32]: [<matplotlib.lines.Line2D object at 0x7f5d8c556750>]
```

It looks well; let us save our valuable work in a file. What to save?

```
In [33]: hist
1 : import numpy as np
2 : op = 2*np.pi; os = 14.*np.pi
3 : R = 4.; r = 2
4 : t = np.linspace(0, 2, 100)
5 : xp = np.cos(op*t)
6 : yp = np.sin(op*t)
7 : xs = xp + r*np.cos(os*t)
8 : ys = yp + r*np.sin(os*t)
9 : from matplotlib import pyplot as plt
10: plt.plot(xs,ys); plt.show()
11: R = 4.; r = 0.25
12: plt.plot(xs,ys); plt.show()
13: _ip.magic("hist ")
14: #?rep
15: _ip.magic("rep 6-10")
16: yp = np.sin(op*t)
17: xs = xp + r*np.cos(os*t)
18: ys = yp + r*np.sin(os*t)
19: from matplotlib import pyplot as plt
20: plt.plot(xs,ys); plt.show()
21: t = np.linspace(0, 2, 2000)
22: _ip.magic("rep 6-10")
23: yp = np.sin(op*t)
24: xs = xp + r*np.cos(os*t)
25: _ip.magic("hist ")
26: _ip.magic("rep 5-10")
27: xp = np.cos(op*t)
28: yp = np.sin(op*t)
29: xs = xp + r*np.cos(os*t)
30: ys = yp + r*np.sin(os*t)
31: from matplotlib import pyplot as plt
32: plt.plot(xs,ys); plt.show()
33: _ip.magic("hist ")
```

Aha, let us save the lines 1 31 2 11 21 5-8 10 in this order:

```
In [34]: save testsat.py 1 31 2 11 21 5-8 10
The following commands were written to file `testsat.py`:
import numpy as np
from matplotlib import pyplot as plt
op = 2*np.pi; os = 14.*np.pi
R = 4.; r = 0.25
t = np.linspace(0, 2, 2000)
xp = np.cos(op*t)
yp = np.sin(op*t)
xs = xp + r*np.cos(os*t)
ys = yp + r*np.sin(os*t)
plt.plot(xs,ys); plt.show()

In [35]:
```

`%who`/`%whos`: These functions give information about identifiers you have defined interactively (not things you loaded or defined in your configuration files). `%who` just prints a list of identifiers and `%whos` prints a table with some basic details about each identifier.

The point about `%` with an example on the magic comand `%cd`:

```
[gradinar@localhost Start]$ ipython
Python 2.6.4 (r264:75706, Jul 14 2010, 09:36:06)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?          -> Introduction and overview of IPython’s features.
%quickref -> Quick reference.
help       -> Python’s own help system.
object?    -> Details about ’object’. ?object also works, ?? prints more.

In [1]: cd ..
/home/gradinar/Documents/SciPyTutorial

In [2]: cd = 5

In [3]: cd
Out[3]: 5

In [4]: cd Start
------------------------------------------------------------
File "<ipython console>", line 1
    cd Start
       ^
SyntaxError: invalid syntax


In [5]: %cd Start
/home/gradinar/Documents/SciPyTutorial/Start

In [6]:
```

More nice documentation on IPython here.

1.8 Examples

Here we put a few complete examples that should be relevant for the numerics lectures.

1.8.1 First example

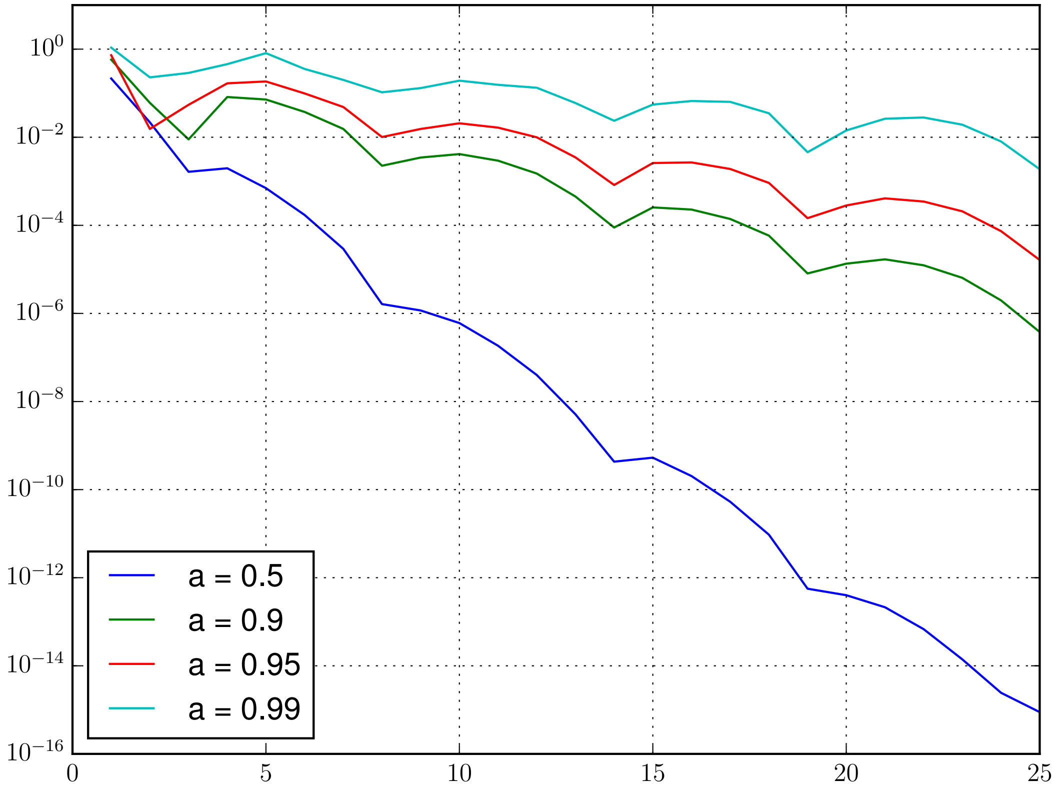
Define $f(x) = \frac{1}{\sqrt{1-a \ \sin(2\pi x-1)}}$.

We want to plot the error made by using the trapezoidal rule when approximating $\int_0^1 f(x)dx$ and $\int_0^{0.5} f(x)dx$.

As exact value we take the value of the integral as produced by the function `quad` from the sub-package `integrate` of `scipy`. This function is just a wrapper to the Fortran library *QUADPACK*. Please, use IPython facility to consult the documentation of it.

The aim of the plots is to put in evidence the exponential convergence of the trapezoidal rule when the smoothness of the function extends above its periodical domain. The plots should be done for several possible parameters $0 < a < 1$.

```python
from numpy import *
from matplotlib.pyplot import *
from scipy import integrate

def trapr(func, a, b, N):
    x = linspace(a,b, N+1)
    h = x[1] - x[0]
    res = sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
    res = res * h
    return res

# number of quadrature points used
N = 25

# integration limits
left = 0.0
right = 1.0

# alocate space for results
res = zeros(N)

lista = [0.5, 0.9, 0.95, 0.99]

for a in lista:
    myf = lambda x: 1./sqrt(1.0 - a*sin(2*pi*x-1))
    for n in xrange(N):
        res[n] = trapr(myf, left, right, n+1)

    exact, aerr = integrate.quad(myf, left, right)

    print("a: "+str(a))
    print("quad: "+str(exact))
    print("aerr: "+str(aerr))

    err = abs(res-exact)
    semilogy(arange(1,N+1), err, label="a = "+str(a))

grid(True)
legend(loc="lower left")
show()
```



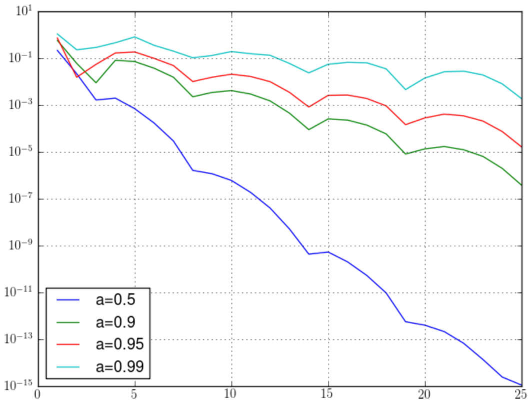
At lines 1-3 we import the libraries we are using. At lines 5-10 we define our quadrature rule function. Note that there are several possibilities to prepare the array of sampling points. The integrand is evaluated once at all points (vectorial) and the function `sum` of numpy is used in order to add the necessary values in most efficient way possible.

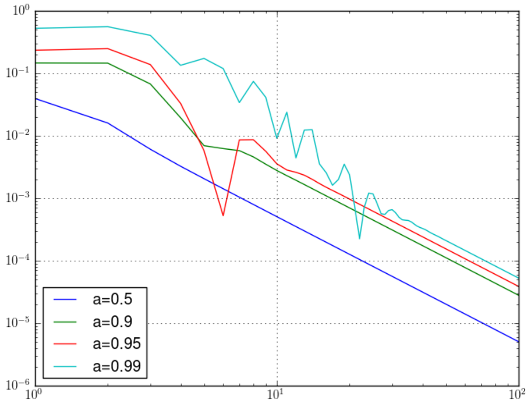
After preparing the data, we run the loop over the interesting values of the parameter `a`.

At line 25 we define the function to be integrated as a *lambda function*. Note that if we define it as a function of two parameters, we'd have difficulties in the later call of the `trapr` function.

We would like to have a program that computes both integrals and makes both plots at one. Here's a possibility:

```
from numpy import *
from matplotlib.pyplot import *
from scipy import integrate

def trapr(func, a, b, N):
    """
    Compute the integral from a to b of func via trap. rule with N points
    """
    x = linspace(a,b, N+1)
    h = x[1] - x[0]
    res = sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
    res = res * h
    return res

def ploterr(left, right, N, lista, pf):
    # allocate space for results
    res = zeros(N)

    for a in lista:
        myf = lambda x: 1.0/sqrt(1.0- a*sin(2*pi*x-1))

        for n in xrange(N):
            res[n] = trapr(myf, left, right, n+1)

        exact, aerr = integrate.quad(myf, left, right)

        print("a: "+str(a))
        print("quad: "+str(exact))
        print("aerr: "+str(aerr))

        err = abs(res-exact)
        pf(arange(1,N+1), err, label="a = "+str(a))

    grid(True)
    legend(loc="lower left")
    savefig("err"+str(right)+".png")
    close()


# not really necessary here, but you see how to do modules and test them
if __name__ == "__main__":
    lista = [0.5, 0.9, 0.95, 0.99]

    ploterr(0., 1., 25, lista, pf=semilogy)
    ploterr(0., 0.5, 100, lista, pf=loglog)
```

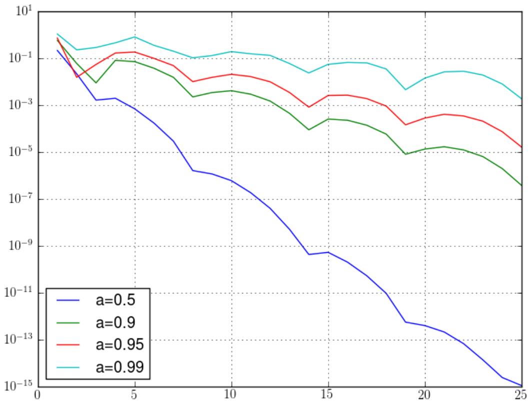


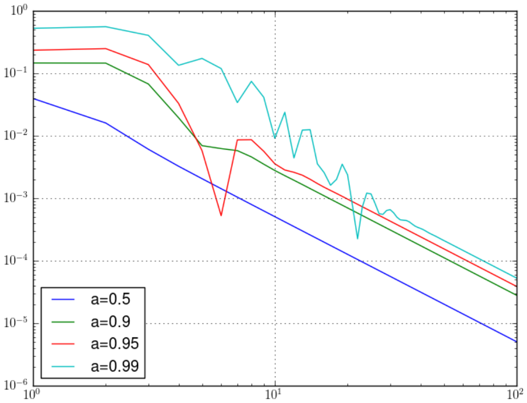


In last code, we incorporated the loop in a function that gets the integration range, the list of wished parameters and the preferred plot function (semilogy is better for the identification of the exponential convergence). The type of the file is decided by the extension of the name of the file. Note also the string concatenation used in the construction of the file name.

Finally, we would like to rewrite the code such that we avoid the trick based on inlining and locality of the variable a. We have now the occasion to see a very powerful trait of python: class function.

```
from numpy import *
from matplotlib.pyplot import *
from scipy import integrate

def trapr(func, a, b, N):
    """
    Compute the integral from a to b of func via trap. rule with N points
    """
    x = linspace(a,b, N+1)
    h = x[1] - x[0]
    y = func(x)
    res = sum(y[1:-1]) + 0.5*(y[0]+y[-1])
    res = res * h
    return res

class Myf(object):
    """
    models a parametric function suited for evidencing
    exponential convergence of trap. rule
    """
    def __init__(self, ina):
        self.a = ina

    def __str__(self):
        return str(self.a)

    def __call__(self, x):
        return 1.0/sqrt(1.0- self.a*sin(2*pi*x-1))

def ploterr(f, left, right, N, lista, pf):
    # allocate space for results
    res = zeros(N)

    for a in lista:
        f.a = a
        print(f)

        for n in xrange(N):
            res[n] = trapr(f, left, right, n+1)

        exact, aerr = integrate.quad(f, left, right)

        print("a: "+str(a))
        print("quad: "+str(exact))
        print("aerr: "+str(aerr))

        err = abs(res-exact)
        pf(arange(1,N+1), err, label="a="+str(a))

    grid(True)
    legend(loc="lower left")
    savefig("err"+str(right)+".png")
    close()


# not really necessary here, but you see how to do modules and test them
if __name__ == "__main__":
    lista = [0.5, 0.9, 0.95, 0.99]

    # initialize the function
    myf = Myf(0.5)

    ploterr(myf, 0., 1., 25, lista, pf=semilogy)
    ploterr(myf, 0., 0.5, 100, lista, pf=loglog)
```



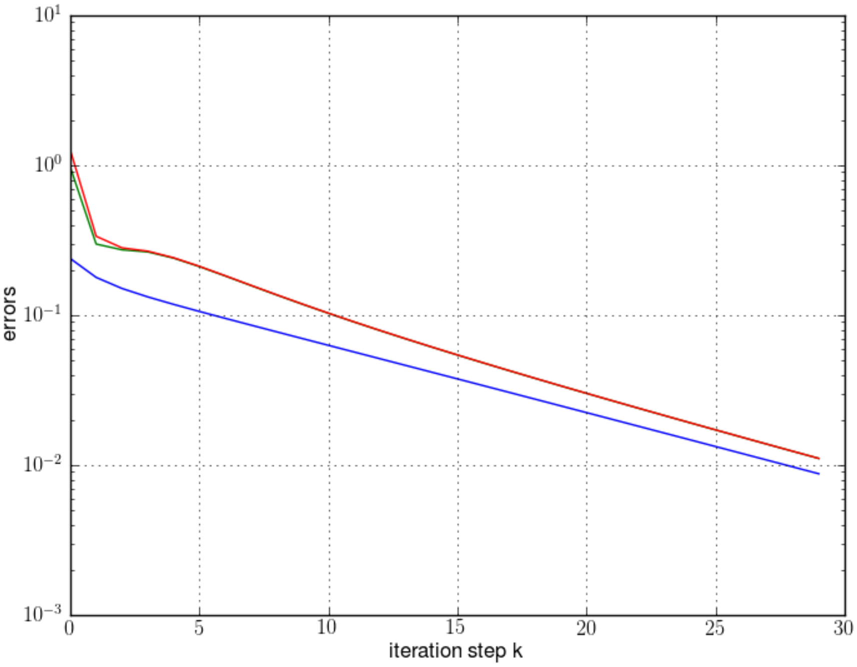


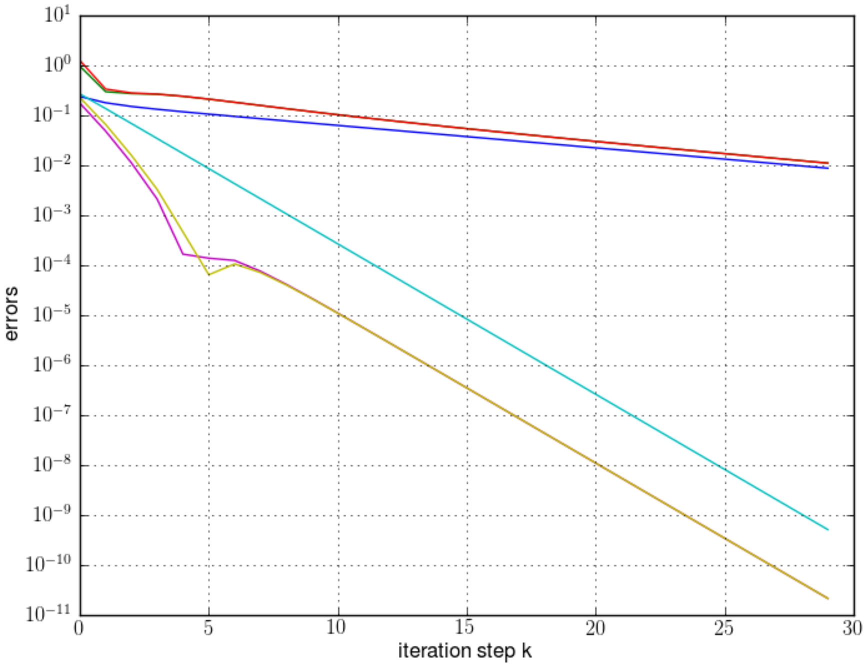
The class `Myf` creates a callable objects. On line 61 we create one such object that models our parameter function, having the parameter set by the first element of the list of interesting parameters. During the loop at line 34, we modify the parameter of the object, which we use as a function to be integrated by our quadrature routines at lines 39 and 41.

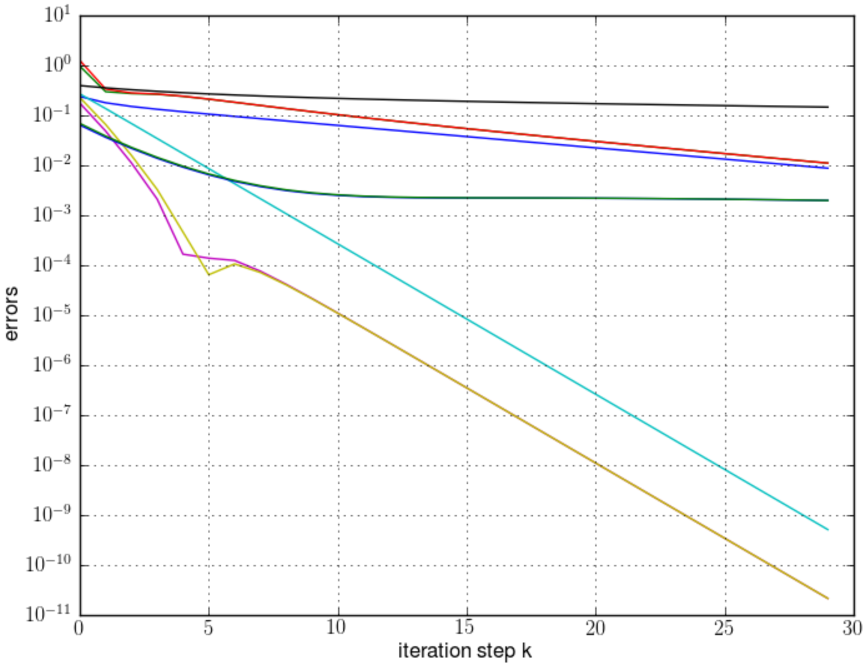
This is a very simple example, but imagine the power at your hands given by the class function when you have to optimize a very complicated function (e.g. given only implicitly as the result of complicated algorithms) depending on several parameters.

1.8.2 Second example

```
"""
Calculates eigenvalues through direct power method

Originaly, there were the input parameters d and maxit,
but in the skript, d = [1:10] and maxit = 30 are used.
"""

import numpy as np
from matplotlib import pyplot as plt

def pm(A, z, sgn, ev, ew, maxit=30):
    """Performs ’maxit’ iterations of the power method
    for the matrix ’A’ and the initial vector ’z’
    and computes the error for the largest eigenvalue ’ew’
    having the sign ’sgn’ and the corresponding eigenvector ’ev’
    """
    res = np.zeros((maxit,5))
    s = 1;
    for k in xrange(maxit):
        w = np.dot(A,z)
        no = np.linalg.norm(w)
        rq = np.real(np.vdot(w,z))
        z = w/no
        err_aew = abs(no-abs(ew))
        err_ew = abs(sgn*rq-ew)
        err_ev = np.linalg.norm(s*z-ev)
        res[k] = np.array([k, no, err_ev, err_aew, err_ew])
        s = s*sgn
    return res

def runpm(d, fname):
    """Performs the analysis of the power iteration starting from
    the array of eigenvalues d; the picture with the results goes
    to the file indicate by fname
    """
    n = len(d) # size of the matrix
    S = np.triu(np.diag(np.r_[n:0:-1]) + np.ones((n,n)))
    A = np.dot(S, np.dot(np.diag(d), np.linalg.inv(S)))

    # need exact ew and ew for error calculation
    w, V = np.linalg.eig(A)
    t = np.where(w == abs(w).max())
    k = t[0]
    if len(k) > 1:
        raise ValueError("Error: no single larges EV")
    ev = V[:,k[0]]
    ev /= np.linalg.norm(ev)
    ew = w[k[0]]

    sgn = 1
    if ew < 0:
        sgn = -1

    print("ew:")
    print(ew)
    print("sgn:")
    print(sgn)
    print("ev =")
    print(ev)

    z = np.random.rand(n)
    z /= np.linalg.norm(z)
    res = pm(A,z,sgn,ev,ew)

    plt.semilogy(res[:,0], res[:,2])
    plt.semilogy(res[:,0], res[:,3])
    plt.semilogy(res[:,0], res[:,4])
    plt.grid(True)
    plt.xlabel("iteration step k")
    plt.ylabel("errors")
    plt.savefig(fname)
    plt.show()


if __name__=="__main__":
    # size of the matrix
    n = 10

    d = 1.0 + np.arange(n) # eigenvalues
    print("d:")
    print(d)
    print("d[-2]/d[-1]:")
    print(d[-2]/d[-1])
    runpm(d, "pm1.png")

    d = np.ones(n)
    d[-1] = 2.0 # eigenvalues
    print("d:")
    print(d)
    print("d[-2]/d[-1]:")
    print(d[-2]/d[-1])
    runpm(d, "pm2.png")

    d = 1.0 - 0.5**np.linspace(1,5,9) # eigenvalues
    print("d:")
    print(d)
    print("d[-2]/d[-1]:")
    print(d[-2]/d[-1])
    runpm(d, "pm3.png")
```







1.8.3 Third example

Serial Monte-Carlo integration via IPython. Serial code:

```
"""
 Computes integral
 I0(1) = (1/pi) int(z=0..pi)  exp(-cos(z)) dz by raw MC.
 Abramowitz and Stegun give I0(1) = 1.266066
"""

import numpy as np
import time

t1 = time.time()

# number of times we run our MC inetgration
M = 100
asval = 1.266065878

ex = np.zeros(M)

print("A and S tables:  I0(1) = "+str(asval))
print("sample       variance        MC I0 val")
print("------       --------        ---------")

# how many experiments
k = 5
N = 10**np.arange(1,k+1)
v = []
e = []

for n in N:
    for m in xrange(M):
        x = np.random.rand(n)
        x = np.exp(np.cos(-np.pi*x))
        ex[m] = sum(x)/n

    ev = sum(ex)/M
    vex = np.dot(ex,ex)/M
    vex -=  ev**2
    v += [vex]
    e += [ev]
    print((n, vex, ev))

t2 = time.time()
t = t2 - t1

print("Serial calculation completed, time = "+str(t)+" s")
```

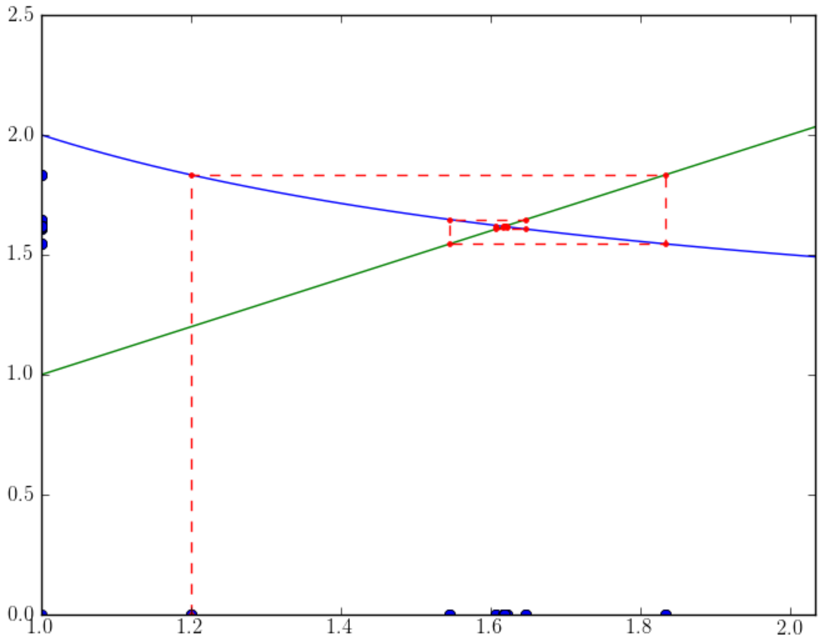
Output:

```
A and S tables:  I0(1) = 1.266065878
sample       variance        MC I0 val
------       --------        ---------
(10, 0.069908462650839942, 1.2525905567351583)
(100, 0.0055876950073416864, 1.2709643005937217)
(1000, 0.0006511488699125767, 1.262876852977836)
(10000, 5.7940507106613026e-05, 1.2664260011112918)
(100000, 6.7123607208063873e-06, 1.265782558306026)
Serial calculation completed, time = 5.22518420219 s
```

1.8.4 Fourth example

A nice plot showing the fixed-point iteration in action finding the golden ration.

```
from numpy import *
from matplotlib.pyplot import *

phi = lambda x: (1.0 + 1.0/(1.0*x))
#phi = lambda x: sqrt(x+1.0)

x0 = 1.2

x = [x0]
y = [0]

for i in xrange(10):
    x1 = phi(x0)

    x.append(x0)
    y.append(x1)

    x.append(x1)
    y.append(x1)

    x0 = x1

x = array(x)
y = array(y)

xmin = x.min() - 0.2
xmax = x.max() + 0.2

t = linspace(xmin, xmax, 50)

figure()
plot([xmin, xmax], [xmin, xmax], "g")
plot(t, phi(t), "b")

plot(x, zeros_like(x), "bo")
plot(xmin*ones_like(y), y, "bo")

plot(x, y, "r--")
plot(x, y, "r.")

xlim(xmin, xmax)
savefig("spiral1.png")

print(x)
print(y)
```



1.9 Some frequent errors

In this section we collect some frequent errors typically found in beginner's `numpy` code. We try to show where the problems come from by some easy examples and explain typical fixes.

1.9.1 Error: `data type not understood`

We wish to construct an array, for example filled with zeros, but only get a `data type not understood` error message like this:

```
In [11]: zeros(3,4)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-dc85bc86d3a9> in <module>()
----> 1 zeros(3,4)

TypeError: data type not understood
```

The reason is that the call of `zeros` is slightly wrong. The correct call syntax is:

```
In [12]: zeros((3,4))
Out[12]:
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

where we have to provide the *shape* as a python tuple (as always). There are similar pitfalls with `ones`, `array` and `eye`. The error occurs because `zeros` understands more arguments than the `shape` parameter only:

```
zeros(...)
    zeros(shape, dtype=float, order='C')

    Return a new array of given shape and type, filled with zeros.

    Parameters
    ----------
    shape : int or sequence of ints
        Shape of the new array, e.g., ``(2, 3)``.
    dtype : data-type, optional
        The desired data-type for the array, e.g., `numpy.int8`.  Default is
        `numpy.float64`.
```

In the above example, `shape` would be `3` and `dtype` is `4` which is of course an invalid type description. The following are all valid examples:

```
In [21]: zeros((3,4), "int")
Out[21]:
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])

In [22]: zeros((3,4), "float")
Out[22]:
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])

In [23]: zeros((3,4), "complex")
Out[23]:
array([[ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  0.+0.j,  0.+0.j]])
```

1.9.2 Error: `setting an array element with a sequence`

The error `setting an array element with a sequence` happens if we try to write something into a single place (*array cell*, *matrix entry*) of an array and this *something* is not a scalar value.

```
In [29]: A = zeros((3,3))

In [30]: A[1,1] = array([1,2])
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-30-3f21ee5db7a1> in <module>()
----> 1 A[1,1] = array([1,2])

ValueError: setting an array element with a sequence.
```

This happens most often when we store the result of some computation which we think is scalar but in fact is not due to an error inside that computation. In the following example we compute a scalar product and store its value inside a matrix:

```
In [34]: v = array([[1,2,3]])

In [35]: A[1,1] = dot(transpose(v), v)
---------------------------------------------------------------------------
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-35-cebb0c885c8d> in <module>()
----> 1 A[1,1] = dot(transpose(v), v)

ValueError: setting an array element with a sequence.
```

Why does this fail? Well, the result of `dot` is in this case not a scalar at all:

```
In [37]: v.shape
Out[37]: (1, 3)

In [38]: dot(transpose(v), v)
Out[38]:
array([[1, 2, 3],
       [2, 4, 6],
       [3, 6, 9]])
```

we started with a row vector and put the `transpose` on the wrong side. The other way round we get what we wanted:

```
In [39]: A[1,1] = dot(v, transpose(v))

In [40]: A
Out[40]:
array([[  0.,   0.,   0.],
       [  0.,  14.,   0.],
       [  0.,   0.,   0.]])
```

1.9.3 Error: `matrices are not aligned`

This error message comes from a whole class of more or less generic errors occurring during array manipulation. They can sometimes be very hard to find:

```
In [111]: A = ones((3,3))

In [112]: b = array([[1, 2, 3]])

In [113]: dot(A, b)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-113-ed12d1ad26b4> in <module>()
----> 1 dot(A, b)

ValueError: matrices are not aligned
```

Why does this fail? We want to multiply a 3 times 3 matrix with a vector of 3 elements, everything should be fine from the mathematical point of view, no? If we look at the operands more closely we see it's not:

```
In [114]: A
Out[114]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

In [115]: b
Out[115]: array([[1, 2, 3]])

In [116]: A.shape
Out[116]: (3, 3)
```

```
In [117]: b.shape
Out[117]: (1, 3)
```

The vector is really a *row vector*! We need to transpose it first:

```
In [118]: dot(A, transpose(b))
Out[118]:
array([[ 6.],
       [ 6.],
       [ 6.]])
```

1.9.4 Error: `operands could not be broadcast`

The error message we look at in this section tells us that we violated the *broadcasting rules*. This is not uncommon as broadcasting can give very unexpected results if done wrong or not understood properly. Remember that we can (for example) add a matrix and a vector but *only* if their shapes fulfill certain criteria:

```
In [124]: A = ones((3,5))

In [125]: A
Out[125]:
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])

In [126]: v = ones((4,1))

In [127]: A + v
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-127-cf7178c26507> in <module>()
----> 1 A + v

ValueError: operands could not be broadcast together with shapes (3,5) (4,1)
```

If we look at `v`:

```
In [128]: v
Out[128]:
array([[ 1.],
       [ 1.],
       [ 1.],
       [ 1.]])
```

we see that this column vector is one element to long. Let’s construct another one with matching length:

```
In [129]: w = ones((3,1))

In [130]: w
Out[130]:
array([[ 1.],
       [ 1.],
       [ 1.]])

In [131]: A + w
Out[131]:
array([[ 2.,  2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.,  2.],
       [ 2.,  2.,  2.,  2.,  2.]])
```

Now it works as expected.

The exact broadcasting rules can be found at:

- http://docs.scipy.org/doc/numpy/reference/ufuncs.html#broadcasting

1.9.5 Error: `array dimensions must agree`

Here another simple case of a shape mismatch. We try to stack some matrices together in the usual block-matrix manipulation fashion:

```
In [41]: v
Out[41]: array([[1, 2, 3]])

In [42]: w = array([[4,5,6,7]])

In [43]: vstack([v,w])
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-43-83da883f09da> in <module>()
----> 1 vstack([v,w])

/usr/lib64/python2.7/site-packages/numpy/core/shape_base.pyc in vstack(tup)
    224
    225     """
--> 226     return _nx.concatenate(map(atleast_2d,tup),0)
    227
    228 def hstack(tup):

ValueError: array dimensions must agree except for d_0
```

We can not stack together arrays if their shapes do not match! Let's correct our mistake and build a matrix from two individual rows:

```
In [44]: w = array([[4,5,6]])

In [45]: w
Out[45]: array([[4, 5, 6]])

In [46]: vstack([v, w])
Out[46]:
array([[1, 2, 3],
       [4, 5, 6]])
```

1.9.6 Error: `LinAlgError: ...`

Errors originating from true mathematical linear algebra issues raise an exception called `LinAlgError`. Here we show two typical examples:

```
In [72]: O = ones((3,3))

In [73]: inv(O)
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
```

```
<ipython-input-73-bdcade699894> in <module>()
----> 1 inv(O)

/usr/lib64/python2.7/site-packages/numpy/linalg/linalg.pyc in inv(a)
    443     """
    444     a, wrap = _makearray(a)
--> 445     return wrap(solve(a, identity(a.shape[0], dtype=a.dtype)))
    446
    447

/usr/lib64/python2.7/site-packages/numpy/linalg/linalg.pyc in solve(a, b)
    326     results = lapack_routine(n_eq, n_rhs, a, n_eq, pivots, b, n_eq, 0)
    327     if results['info'] > 0:
--> 328         raise LinAlgError, 'Singular matrix'
    329     if one_eq:
    330         return wrap(b.ravel().astype(result_t))

LinAlgError: Singular matrix
```

Indeed the matrix has no inverse:

```
In [74]: det(O)
Out[74]: 0.0
```

Another one is:

```
In [75]: R = ones((3,4))

In [76]: inv(R)
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-76-eb140fe92b28> in <module>()
----> 1 inv(R)

/usr/lib64/python2.7/site-packages/numpy/linalg/linalg.pyc in inv(a)
    443     """
    444     a, wrap = _makearray(a)
--> 445     return wrap(solve(a, identity(a.shape[0], dtype=a.dtype)))
    446
    447

/usr/lib64/python2.7/site-packages/numpy/linalg/linalg.pyc in solve(a, b)
    310         b = b[:, newaxis]
    311     _assertRank2(a, b)
--> 312     _assertSquareness(a)
    313     n_eq = a.shape[0]
    314     n_rhs = b.shape[1]

/usr/lib64/python2.7/site-packages/numpy/linalg/linalg.pyc in _assertSquareness(*arrays)
    158     for a in arrays:
    159         if max(a.shape) != min(a.shape):
--> 160             raise LinAlgError, 'Array must be square'
    161
    162 def _assertFinite(*arrays):

LinAlgError: Array must be square
```

This can of course never work:

```
In [77]: R.shape
Out[77]: (3, 4)
```

A failing matrix decomposition:

```
In [9]: A = diag([1,0,-1])

In [10]: A
Out[10]:
array([[ 1,  0,  0],
       [ 0,  0,  0],
       [ 0,  0, -1]])

In [11]: cholesky(A)
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-11-59f690b5723f> in <module>()
----> 1 cholesky(A)

/usr/lib64/python2.7/site-packages/numpy/linalg/linalg.pyc in cholesky(a)
    529     if results['info'] > 0:
    530         raise LinAlgError, 'Matrix is not positive definite - \
--> 531         Cholesky decomposition cannot be computed'
    532     s = triu(a, k=0).transpose()
    533     if (s.dtype != result_t):

LinAlgError: Matrix is not positive definite -         Cholesky decomposition cannot be computed
```

By construction the matrix `A` was not a valid input for Cholesky factorization.

1.9.7 Error: `x and y must have same first dimension`

This errors happens when plotting things goes wrong. It is a very typical error thrown by `matplotlib`. Let's plot a simple graph with a few points:

```
In [87]: x = arange(5)

In [88]: y = array([[2,4,6,7,1]])

In [89]: figure()
Out[89]: <matplotlib.figure.Figure at 0x4e77550>

In [90]: plot(x, y)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-90-500b81ce9ab8> in <module>()
----> 1 plot(x, y)

/usr/lib64/python2.7/site-packages/matplotlib/pyplot.pyc in plot(*args, **kwargs)
   2284         ax.hold(hold)
   2285     try:
-> 2286         ret = ax.plot(*args, **kwargs)
   2287         draw_if_interactive()
   2288     finally:

/usr/lib64/python2.7/site-packages/matplotlib/axes.pyc in plot(self, *args, **kwargs)
   3783         lines = []
   3784
-> 3785         for line in self._get_lines(*args, **kwargs):
   3786             self.add_line(line)
   3787             lines.append(line)

/usr/lib64/python2.7/site-packages/matplotlib/axes.pyc in _grab_next_args(self, *args, **kwargs)
    315                 return
    316             if len(remaining) <= 3:
--> 317                 for seg in self._plot_args(remaining, kwargs):
    318                     yield seg
    319                 return

/usr/lib64/python2.7/site-packages/matplotlib/axes.pyc in _plot_args(self, tup, kwargs)
    292             x = np.arange(y.shape[0], dtype=float)
    293
--> 294         x, y = self._xy_from_xy(x, y)
    295
    296         if self.command == 'plot':

/usr/lib64/python2.7/site-packages/matplotlib/axes.pyc in _xy_from_xy(self, x, y)
    232         y = np.atleast_1d(y)
    233         if x.shape[0] != y.shape[0]:
--> 234             raise ValueError("x and y must have same first dimension")
    235         if x.ndim > 2 or y.ndim > 2:
    236             raise ValueError("x and y can be no greater than 2-D")

ValueError: x and y must have same first dimension
```

The reason for this failure is that the two arrays `x` and `y` have the same number of elements but still a *different shape*:

```
In [91]: x
Out[91]: array([0, 1, 2, 3, 4])

In [92]: y
Out[92]: array([[2, 4, 6, 7, 1]])

In [93]: x.shape
Out[93]: (5,)

In [94]: y.shape
Out[94]: (1, 5)

In [95]: x.ndim
Out[95]: 1

In [96]: y.ndim
Out[96]: 2
```

This is very nasty and happens now and then not to beginners only. Because we were not careful enough, one array is two-dimensional! Mostly this actually happens when one part is a matrix having one row or column. The solution is to call `squeeze` to remove the singular dimension(s):

```
In [97]: figure()
Out[97]: <matplotlib.figure.Figure at 0x515fd50>

In [98]: plot(x, squeeze(y))
Out[98]: [<matplotlib.lines.Line2D at 0x5206050>]
```

1.10 Useful links

1.10.1 Tutorials, Manuals and Books

- Python Scientific Lecture Notes
- Documentations on NumPy and SciPy
- Tentative NumPy Tutorial
- Scipy tutorial
- NumPy for Matlab Users
- IPython documentation
- SymPy documentation
- h5py project website and h5py user guide
- A good book on the subject is H. Langtangen, "Python Scripting for Computational Science", available electronically on NEBIS, too.
- A French overview
- Using IPython for parallel computing

1.10.2 Additional software

- The spyder IDE
- winpdb, a python debugger
- mayavi2, 3D scientific data visualization
- image magic, batch image processing
- mplayer, versatile video player and encoder
- hdfview, HDF file editor

CHAPTER

TWO

ADVANCED STUFF (OPTIONAL)

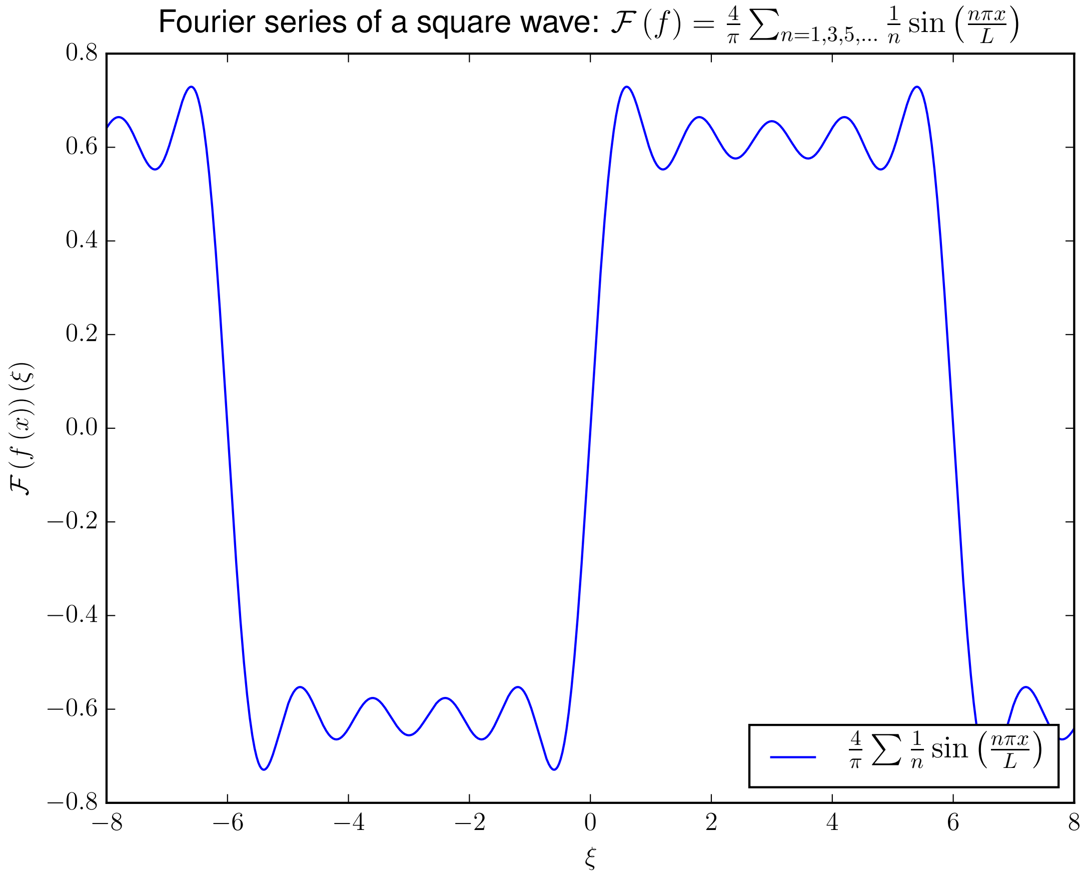
2.1 Matplotlib tips

2.1.1 True TeX labels

The plotting utilities of matplotlib can use the power of a TeX system if one is installed. We just have to tell matplotlib that it should generate the labels with the help of TeX. We then can use anything that is a valid TeX expression we would otherwise write between the dollar signs in a TeX file. The following code presents a small example showing TeX labels in action.

NOTE: you may have issues with the following usetex-options in case that latex is bad configured on MacOS or Windows.

```
from numpy import *
from pylab import *
from matplotlib import rc

# Enable usage of real TeX for labels and captions
rc('text', usetex=True)

# Build a complicated function
x = linspace(-8, 8, 5000)

y = zeros(x.shape)
for n in xrange(1,11,2):
    y += 1.0 / n * sin(n * pi * x / 6.0)
y = pi / 4.0 * y

# Plot and label with full TeX support
figure()
plot(x, y, label=r"$\frac{4}{\pi} \sum \frac{1}{n} \sin\left( \frac{n \pi x}{L} \right)$")
xlabel(r"$\xi$")
ylabel(r"$\mathcal{F}\left(f\left(x\right)\right)\left(\xi\right)$")
legend(loc="lower right")
title(r"Fourier series of a square wave: $\mathcal{F}\left(f\right) = \frac{4}{\pi} \sum_{n=1,3,5,\lo
savefig("tex_labels.png")
```



Note that we have to use so called raw strings, i.e. strings with the character `r` as a prefix. Additionally we use the dollar signs inside the strings for delimiting formulas as usual.

2.2 Making of: animations

Plots are nice but sometimes an animation is even better. Thus we take the opportunity and explain in this section how to generate simple animations. An animation is nothing else than a pile of images which are shown one after the other.

This is also exactly the way we will create animations: in a first step we draw each image (we call this a frame) and after we got all the frames, we will glue them together.

There is more than one way to make animations. We describe here two possibilities which probably are used most.

- Animated images: use the common `.gif` image file format (so called *animgifs*).
- Videos: use some video compression algorithm and are placed inside a data container.

Both of these options have their respective advantages and disadvantages. Animated gif images tend to be larger in file size, but will yield good results without much technicalities. If you only have a couple of images, say up to 50 frames, then animgifs are probably the easiest way to go. (The file *animwave.gif* from the example below has a size of 750 KB)

For longer animations you should of course use the advantages of modern video compression techniques. But mature video encoding tools usually have a huge number of options interacting in a complex manner. It's not easy to find the parameters which give you the best result. For example the video compression algorithms are tuned for areas and can introduce bad artefacts if the image has only a bunch sharp lines in front of a uniform background. But this is what plots look like most of the time! (By the way, the file *animwave.flv* from the example below has a size of 700 KB.)

2.2.1 External tools

We additionaly need two programs which are not part of the python installation as proposed at the beginning of this tutorial. But they may be already preinstalled on a common Linux system today.

The first of these tools is called `convert` which is itself part of the well known `ImageMagic` batch image processor. We use this tool for creating animated gif images.

- ImageMagic can be found at http://www.imagemagick.org/

(This is not the only tool that can compose gif files, you may also try `gifsicle` which can be found at http://www.lcdf.org/gifsicle/. My own experience is that this tool sometimes is faster, uses less memory and the resulting files are smaller in size. But don't take this as a given.)

To encode videos we use the famous `mplayer` / `mencoder` toolbox. This is one of the most versatile tools for working with video data. Mplayer can play almost any file format and has a huge set of options. Mencoder is responsible for encoding videos and does a very good job.

- Mplayer / Mencoder can be found at http://www.mplayerhq.hu/design7/news.html

Describing these tools is far beyond the scope of this document. You should take a look at the documentation available on the webpages mentioned above. The *man pages* also provide a clean description of the parameters and show some useful examples at the very end.

2.2.2 Command lines

For creating the animations we relay on these external tools. We just call them from inside the python scripts. You may also call them directly at your favourite shell. The commands used in the example below are for animated gifs:

```
convert -delay 20 filebasename*.png animfilename.gif
```

where `filebasename` is a prefix that is common to all files you want to include in the animation. Instead of `filebasename*.png` you can plug in any regular expression here. Similar `animfilename` if the name the output file will have. The parameter `delay` specifies the frame rate, a value of 20 means a time delay of 20 ms between each two frames.

The command for encoding videos is much more complex (note that this is all **one** single line):

```
mencoder mf://./ filenamebase*.png  -fps 5 -ofps 5 -o animfilename.flv -of lavf
-oac mp3lame -lameopts abr:br=64 -srate 22050 -ovc lavc
-lavcopts vcodec=flv:keyint=50:vbitrate=700:mbd=2:mv0:trell:v4mv:cbp:last_pred=3
-vf yadif -sws 9
```

where `filebasename` and `animfilename` and as above. The resulting video is of file type `.flv` (flash video) [1]. You should change the other parameters only if you really know what you do. (These values here are passed on since the early universe.)

2.2.3 Example Code

The following example shows a simple wave packet propagating in a one-dimensional space.

[1] For more information on this particular video format see http://en.wikipedia.org/wiki/Flv.

The function `make_frames()` is responsible for plotting the frames. Notice how we start a new figure with the command `figure()` in each iteration of the for loop. This is to avoid plotting into the last frame. Also we need some bits of intelligence while saving the images. Because we use regular expressions for selecting all images that will be part of the animation, we are subject to the way the shell sorts filenames. To avoid wrong sorting we use a 5-digit numbering with zero padding here:

```
"wave_%05d" % i
```

results in file names like the following ones:

```
wave_00000.png wave_00001.png wave_00002.png wave_00003.png ...
```

and assures the files will be in the correct order. The other two functions:

```
make_animation_animgif(filebasename, animfilename)
```

and:

```
make_animation_video(filenamebase, animfilename)
```

just compose a string containing the shell commands shown above. Then they will execute this command on the system's shell. You can copy these two functions to your own code and use them to generate animations. The rest of the code should be self-explanatory.

By the way, with code similar to the one shown here you can dynamically compose and execute **any** command on the system's shell! Therefore you should be very careful with this powerful mechanism.

Note: this example do not work on windows (as long as "convert" and "mencoder" are not installed)

```
from numpy import *
from pylab import *

import os
import subprocess as sp


def make_frames():
    """This function generates the images (frames) of the animation.
    """

    wave = lambda x, f: sin(f*x)
    envelope = lambda x, l: exp(-(x-l)**2)

    x = linspace(-5,5,2000)
    positions = linspace(-5,5,50)

    # Plot all the individual frames
    for i, pos in enumerate(positions):
        y = wave(x,10) * envelope(x, pos)
        z = envelope(x, pos)

        # Plot the frame
        figure()
        plot(x, z, color=(0.5,0,0))
        plot(x,-z, color=(0.5,0,0))
        plot(x, y, color=(0,0,1))
        ylim(-1.3, 1.3)

        # Save the with a filename that ensures correct order on the filesystem
        savefig("wave_"+(5-len(str(i)))*"0"+str(i)+".png")
```

```
def make_animation_video(filenamebase, animfilename):
    """A function that takes frames and composes them into a video.
    """

    # The shell command controlling the ’mencoder’ tool.
    # Please refer to the man page for the parameters.
    # Warning: Use of multiline strings with EOL escaping to reduce line length!
    command = """mencoder mf://./""" + filenamebase + """*.png\
    -fps 5 -ofps 5 -o """ + animfilename + """.flv\
    -of lavf -oac mp3lame -lameopts abr:br=64 -srate 22050 -ovc lavc\
    -lavcopts vcodec=flv:keyint=50:vbitrate=700:mbd=2:mv0:trell:v4mv:cbp:last_pred=3\
    -vf yadif -sws 9"""

    # Execute the command on the system’s shell
    proc = sp.Popen(command, shell=True)
    os.waitpid(proc.pid, 0)



def make_animation_animgif(filebasename, animfilename):
    """Take a couple of images and compose them into an animated gif image.
    """

    # The shell command controlling the ’convert’ tool.
    # Please refer to the man page for the parameters.
    command = "convert -delay 20 " + filebasename + "*.png " + animfilename + ".gif"

    # Execute the command on the system’s shell
    proc = sp.Popen(command, shell=True)
    os.waitpid(proc.pid, 0)



if __name__ == "__main__":
    make_frames()
    make_animation_video("wave_", "animwave")
    make_animation_animgif("wave_", "animwave")
```

2.3 Other noteworthy plot commands

While you saw several plot command and know how to do different 2D and 3D plots, we will list here some other plot command that might be interesting.

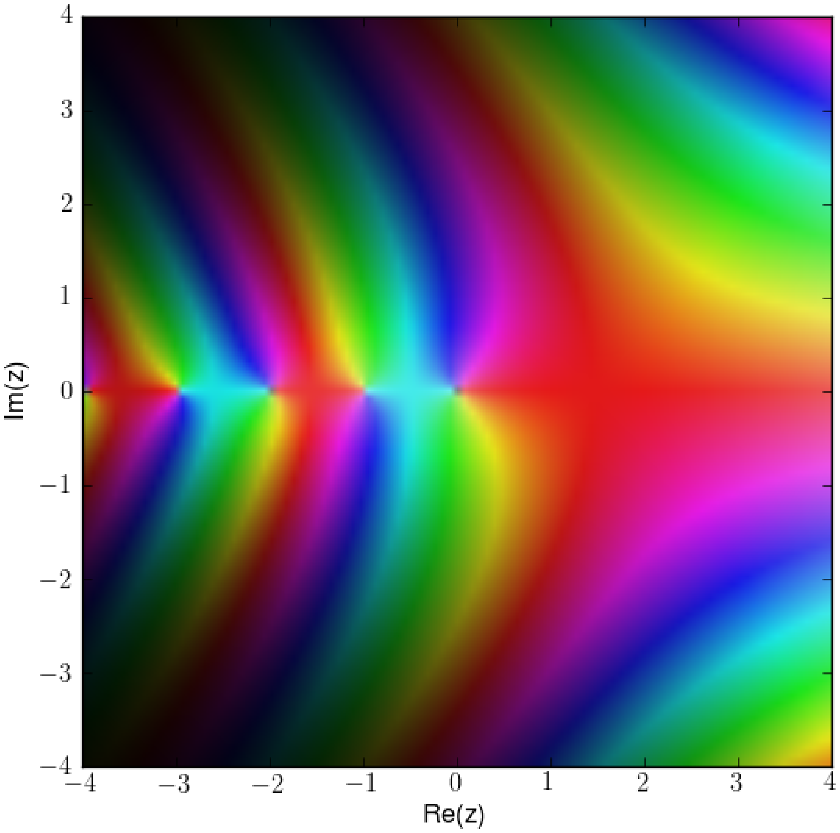
2.3.1 Plots in the complex plane

Plots of a possibly complex function in the complex plane can be easily done with plot routines from the `mpmath` python package.

The example plots the gamma function over the complex plane:

```
import mpmath as mp

mp.cplot(mp.gamma, re=[-4,4], im=[-4,4], points=10000, file="gamma.png")
```

This is all for plotting the gamma function in the rectangle -4 < Re(z) < 4 and -4 < Im(z) < 4. The above call saves the result to the file “gamma.png” which looks like this:



2.4 Mayavi2 tips

The *mayavi.mlab* module, that we call mlab, provides an easy way to visualize data in a script or from an interactive prompt with one-liners as done in the matplotlib `pyplot` interface but with an emphasis on 3D visualization using Mayavi2. This allows users to perform quick 3D visualization while being able to use Mayavi’s powerful features. For more details see:

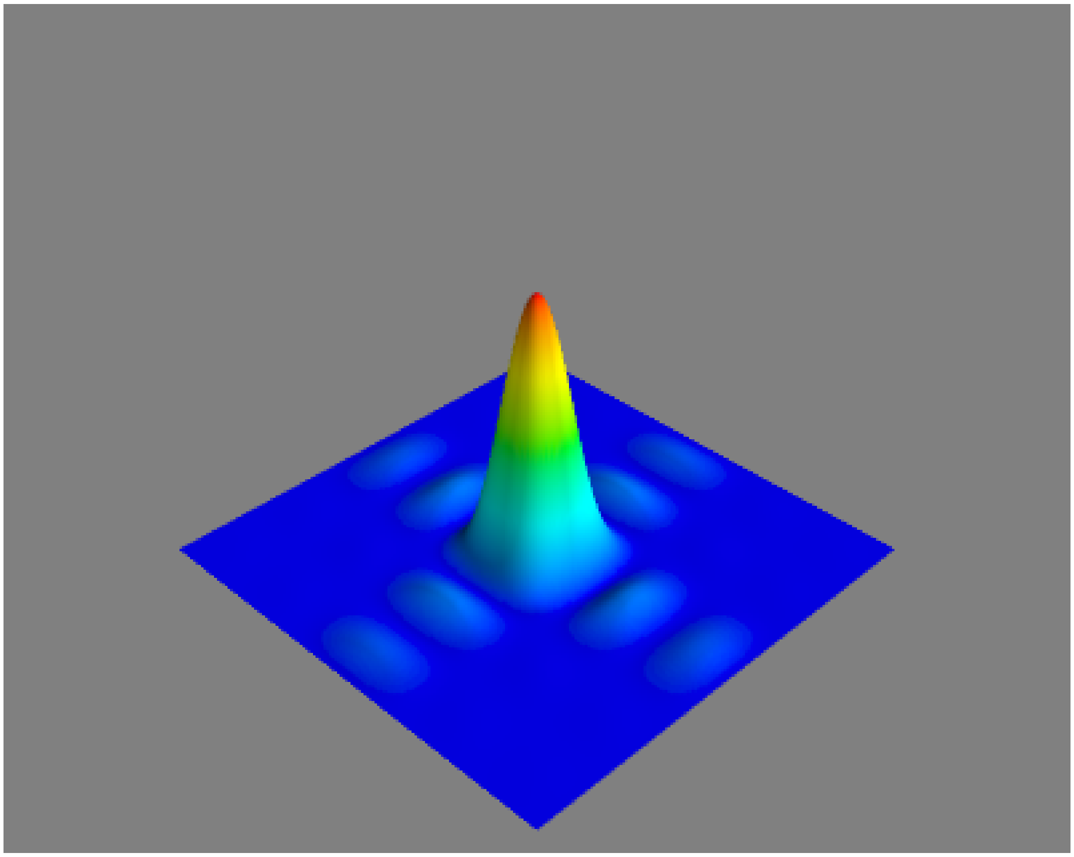
- Scripting with mayavi2
- Reference manual

Try:

```
In [1]: import numpy as np

In [2]: from mayavi import mlab
In [3]: t = np.linspace(0, 2*np.pi, 50)

In [4]: u = np.cos(t)*np.pi

In [5]: x, y, z = np.sin(u), np.cos(u), np.sin(t)

In [6]: mlab.points3d(x,y,z)

In [7]: mlab.show()

In [8]: mlab.points3d(x,y,z,t, scale_mode=’none’)

In [9]: mlab.show()

In [10]: mlab.points3d?

In [11]: x, y = np.mgrid[-3:3:100j, -3:3:100j ]

In [12]: z = np.sin(x*x + y*y)

In [13]: mlab.surf(x,y,z)

In [14]: mlab.show()

In [15]: mlab.mesh(x,y,z)

In [16]: mlab.show()
```

A simple example showing a 3D plot done with the help of mayavi2. The following code shows what it takes to produce such an image. The plotting is essentially only two lines of code!

```python
from numpy import mgrid, real, conj, ones, zeros
from numpy.fft.fftpack import fft2
from numpy.fft.helper import fftshift
from mayavi import mlab

# A mesh grid
X,Y = mgrid[-100:100, -100:100]

# The initial function: a 2D unit step
Z = zeros((200,200))
Z[0:6,0:6] = 0.3*ones((6,6))

# The fourier transform: a 2D sinc(x,y)
W = fftshift(fft2(Z))
W = real(conj(W)*W)

# Display the data with mayavi
# Plot the original function
#mlab.mesh(X, Y, Z)
# Plot the fourier transformed function
mlab.mesh(X, Y, W)
mlab.savefig("mayavi_fft_plot.png")
```



And now an animated example:

```
from mayavi import mlab
from scipy import sin, ogrid, array
from pylab import plot, show

# prepare data, hence test scipy elements
x , y = ogrid[-3:3:100j , -3:3:100j]
z = sin(x**2 + y**2)

# test matplotlib
plot(x, sin(x**2)); show()

#now mayavi2
obj = mlab.surf(x,y,z)
P = mlab.pipeline
scalar_cut_plane = P.scalar_cut_plane(obj, plane_orientation='y_axes')
scalar_cut_plane.enable_contours = True
scalar_cut_plane.contour.filled_contours = True
scalar_cut_plane.implicit_plane.widget.origin = array([  0.00000000e+00,   1.46059210e+00,  -2.026557

scalar_cut_plane.warp_scalar.filter.normal = array([ 0.,  1.,  0.])
scalar_cut_plane.implicit_plane.widget.normal = array([ 0.,  1.,  0.])
f = mlab.gcf()
f.scene.camera.azimuth(10)

f.scene.show_axes = True
f.scene.magnification = 4 # or 4
mlab.axes()
```

```python

# Now animate the data.
dt = 0.01; N = 40
ms = obj.mlab_source
for k in xrange(N):
    x = x + k*dt
    scalars = sin(x**2 + y**2)
    ms.set(x=x, scalars=scalars)
```

The next example demonstrates how to use scalar cut planes in order to evidence subtle characteristics of a plot. Here, we would like to show the Gibbs oscillations in the Fourier series approximation of jumps.

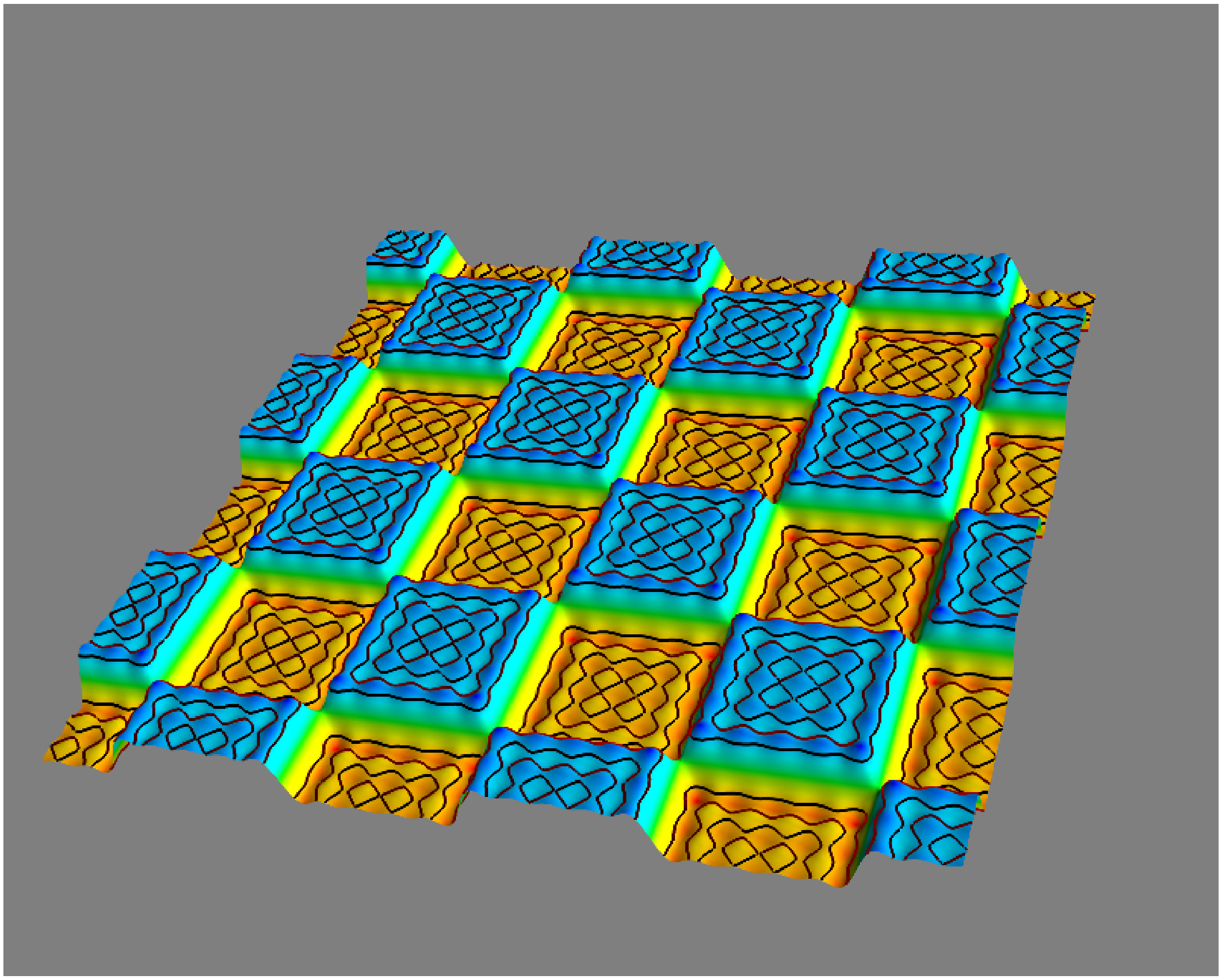
The workflow for such a problem is:

- write a simple program (let us call it pl.py) that plots your object
- run mayavi2 -x pl.py (python pl.py works too for just executing the script)
- use the 'spion' of mayavi2: click on the red point in the menu: the protocol of everything you do visually with the mouse is python-scripted in the opened window
- play with the possibilities and variables that mayavi2 offers, till the image that you see satisfies your exigences
- copy from the 'spion'-window the relevant information for your result into your pl.py file
- save the picture, if you wish; CAUTION: the pictures that mayavi2 produces became huge, if saved in vector graphics ... so eps is not recommeded here!
- re-run mayavi2 -x pl.py to see if this is really what you wanted
- iterate the above process, if wished
- improve speed and quality of the plot by avoiding rendering and increasing size

In the following code (that was developped following this receipt) you find hints for your formulations.

```python
from numpy import mgrid, zeros, sin, pi, array
from mayavi import mlab

Lx = 4
Ly = 4

S = 10
dx = 0.05
dy = 0.05

# A mesh grid
X,Y = mgrid[-S:S:dx, -S:S:dy]

# The initial function:
#Z = zeros(X.shape)
#Z = sin(pi*X/Lx)*sin(pi*Y/Ly)
#Z[Z>0] = 1
#Z[Z<0] = -1

# The Fourier series:
W = zeros(X.shape)
m = 10
for i in xrange(1,m,2):
    for j in xrange(1,m,2):
        W += 1.0 / (i*j) * sin(i * pi * X / Lx) * sin(j * pi * Y / Ly)
```

```
W *= pi / 4.0


# prepare scene
scene = mlab.gcf()
# next two lines came at the very end of the design
scene.scene.magnification = 4

# plot the object
obj = mlab.mesh(X, Y, W)
P = mlab.pipeline

# first scalar_cut_plane
scalar_cut_plane = P.scalar_cut_plane(obj, plane_orientation=’z_axes’)
scalar_cut_plane.enable_contours = True
scalar_cut_plane.contour.filled_contours = True
scalar_cut_plane.implicit_plane.widget.origin = array([-0.025, -0.025,  0.48])
scalar_cut_plane.warp_scalar.filter.normal = array([ 0.,  0.,  1.])
scalar_cut_plane.implicit_plane.widget.normal = array([ 0.,  0.,  1.])
scalar_cut_plane.implicit_plane.widget.enabled = False # do not show the widget

# second scalar_cut-plane
scalar_cut_plane_2 = P.scalar_cut_plane(obj, plane_orientation=’z_axes’)
scalar_cut_plane_2.enable_contours = True
scalar_cut_plane_2.contour.filled_contours = True
scalar_cut_plane_2.implicit_plane.widget.origin = array([-0.025, -0.025,  -0.48])
scalar_cut_plane_2.warp_scalar.filter.normal = array([ 0.,  0.,  1.])
scalar_cut_plane_2.implicit_plane.widget.normal = array([ 0.,  0.,  1.])
scalar_cut_plane_2.implicit_plane.widget.enabled = False # do not show the widget

# see it from a closer view point
scene.scene.camera.position = [-31.339891336951567, 14.405281950904936, -27.156389988308661]
scene.scene.camera.focal_point = [-0.025000095367431641, -0.025000095367431641, 0.0]
scene.scene.camera.view_angle = 30.0
scene.scene.camera.view_up = [0.63072643330371414, -0.083217283169475589, -0.77153033000256477]
scene.scene.camera.clipping_range = [4.7116394000124906, 93.313165745624019]
scene.scene.camera.compute_view_plane_normal()

scene.scene.render()
```



2.5 Symbolic calculations

Even if we want to deal with numerical computations mainly, I'd like to give a short introduction into symbolic calculation. There are several reasons why we want to include some symbolic manipulations in a numerical code. For example we want to do an error analysis and would like to be able to evaluate the known exact solution on various different grids. Or we have a closed form solution for some right hand side. Beyond the toy examples there exist also industrial strength codes that use symbolic computation in a preparation step before going into the numerics [2].

In any case, we could put the symbolic expression literally into the source code. But sometimes it would be useful if we could do a few more things with the expression beside numerical evaluation, for example we may need the first derivative. The benefit of symbolic calculation is now that we can compute this derivation without explicitly put it into the source code or falling back to numerical techniques.

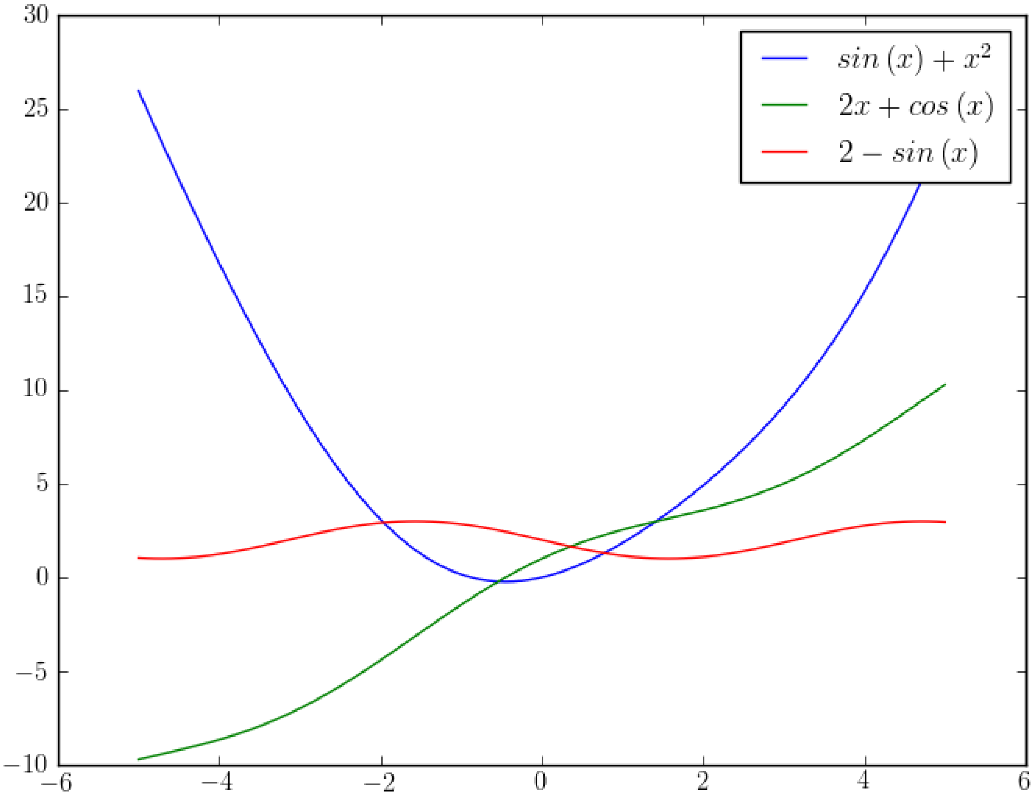
In the following section I'll show a really simple example of how to use the power of the `sympy` python module. This module acts as a library for symbolic calculation and is quite easy to use yet surprisingly powerful for it's complexity.

```
# Use explicit namespaces to make clear from which package a function comes.
import sympy as sp
import numpy as np
import matplotlib.pyplot as pl
```

[2] See fore example SyFi which is part of the FEniCS project http://www.fenicsproject.org/

```python
# Define a symbol
x = sp.Symbol("x")

# A symbolic expression
expr = sp.sin(x) + x**2

# We can evaluate it for any value of x:
print(expr.subs({x:sp.pi**2}))


# A function for evaluating the expression by symbolic
# manipulations. We have to assign the function parameter
# y to the symbolic variable x of the expression.
fs = lambda y: expr.subs({x:y})

# Again we can evaluate the expression for any value of x easily:
print(fs(2))
print(fs(sp.pi))


# But we want the function to work with the arrays from numpy too.
# Thus we have to lambdify the expression.
fn = sp.lambdify(x, expr, "numpy")

xvals = np.linspace(-5,5,100)
yvals = fn(xvals)


# Compute the derivative of our expression
dexpr = sp.diff(expr, x)

# And again lambdify it
fnd = sp.lambdify(x, dexpr, "numpy")

# Evaluate the derivative on the given grid
dyvals = fnd(xvals)


# Compute the second derivative of our expression
ddexpr = sp.diff(expr, x, 2)

# And again lambdify it
fndd = sp.lambdify(x, ddexpr, "numpy")

# Evaluate the derivative on the given grid
ddyvals = fndd(xvals)


# And plot the function sampled at the grid points xvals
# Note how we reuse the symbolic expressions for generating the plot labels too!
pl.figure()
pl.plot(xvals, yvals, label=r"$%s$" % sp.latex(expr))
pl.plot(xvals, dyvals, label=r"$%s$" % sp.latex(dexpr))
pl.plot(xvals, ddyvals, label=r"$%s$" % sp.latex(ddexpr))
pl.legend()
pl.savefig("symbolics_demo.png")
```

And this is how the output looks like



For many other simple examples, please look at the introductory level `sympy` tutorial [3].

Note that you should use the **computationaly expensive symbolical calculations** only in the preparation phase of a numerical simulation program and never in a main computation loop.

2.6 Optimization of Python Code

Good style for speed [4]:

1. Avoid explicit loops: use vectorizd numpy expressions instead. If you really need a loop, then try `list comprehensions` as they are in general much faster. Example:

```
# Slow:
x = []
for i in xrange(8):
    x.append( i**2 )

# Better:
x = [0, 0, 0, 0, 0, 0, 0, 0]
for i in xrange(len(x)):
    x[i] = i**2

```

[3] http://docs.sympy.org/latest/tutorial/index.html

[4] See also http://wiki.python.org/moin/PythonSpeed/PerformanceTips

```
# Best:
x = [ i**2 for i in xrange(8) ]
```

2. Avoid module prefix in frequently called funcions:

   ```
   import mod
   func = mod.func
   for x in hugelist:
        func(x)
   ```

   Or even import the function into the global namespace:

   ```
   from mod import func
   for x in hugelist:
        func(x)
   ```

3. Plain functions are called faster than class methods: trick:

   ```
   f = myobj.__call__
   f(x)
   ```

4. Inlining functions speeds-up the code
5. Avoid usieng numpy functions with scalar arguments
6. Use xrange instead of range (but be aware that xrange only supports a small part of the interface of `range()` and similar containers. [5])
7. if-else is faster than try-except (never use exceptions for program flow!)
8. Avoid resizing of numpy arrays
9. Callbacks to Python from Fortran/C/C++ are expensive
10. Avoid unnecessary allocation of large data structures

11. Be careful with type mactching in mixed Python-Fortran software (e.g. flat/ real*8): if the array entry types do not match, arrays will be copied!

12. Use asarray with parameter order='Fortran' when sending arrays from numpy to Fortran routines!

2.7 Timing Your Code

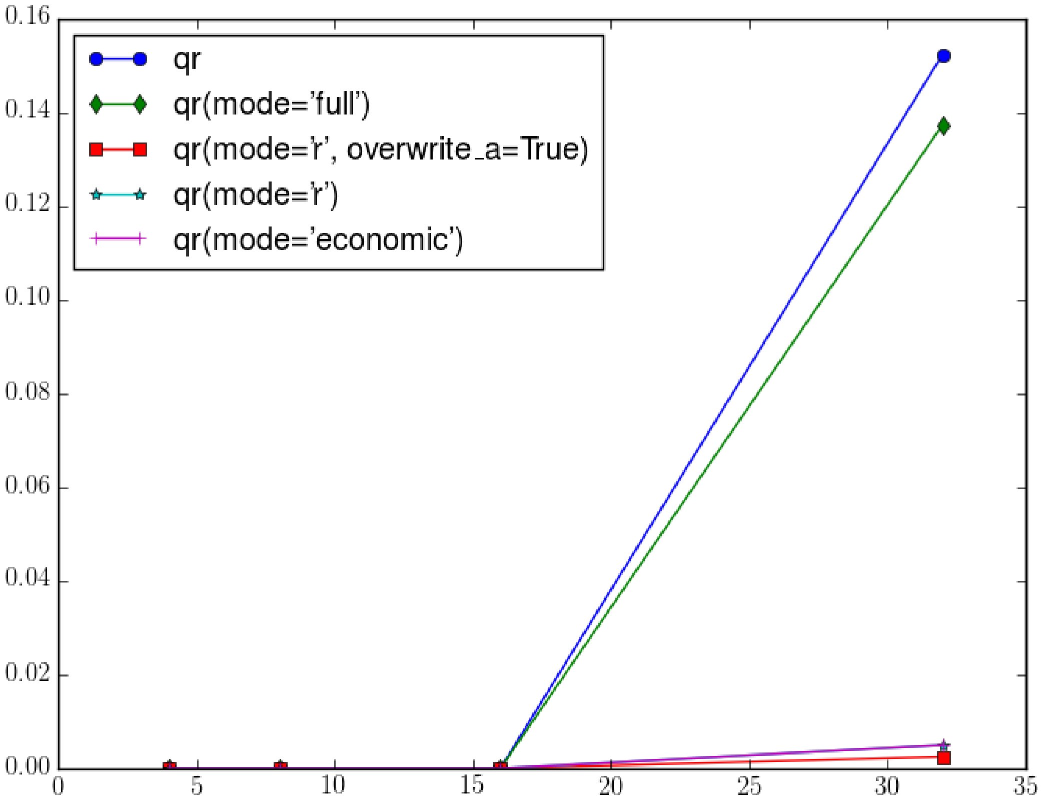
Timing your code is a more complex task you imagine. If the code to run takes long and you are not fond in very precise measurements, then the module time is a very simple way to do it. Otherwise you should use the module timeit. See the two examples below.

```
from numpy import r_, mat, vstack, eye, ones, zeros
from scipy.linalg import qr
import time
nrEXP = 4
sizes = 2**r_[2:6]
qrtimes = zeros((5,sizes.shape[0]))
k = 0

for n in sizes:
```

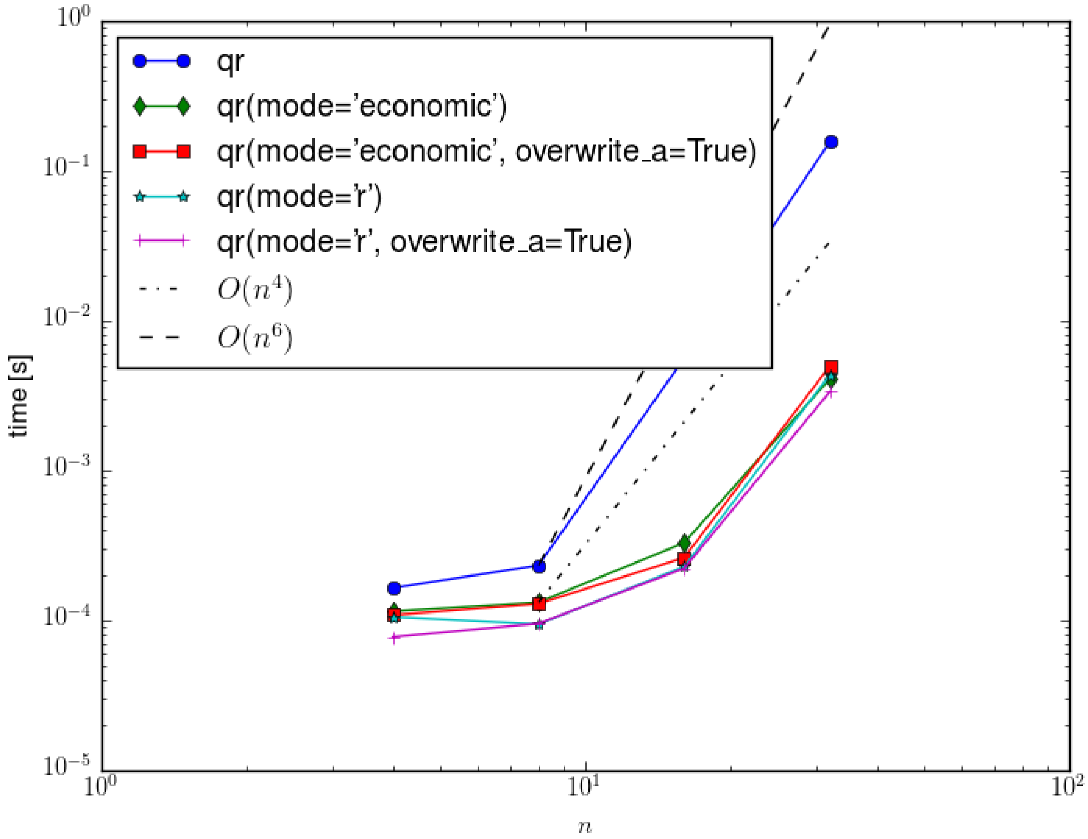
[5] Objects of type xrange don't support slicing, concatenation or repetition, xrange objects have very little behavior: they only support indexing, iteration, and the len() function. The advantage of the xrange type is that an xrange object will always take the same amount of memory, no matter the size of the range it represents. (This is some kind of lazy evaluation.) http://docs.python.org/library/functions.html#xrange

```
    print 'n=', n
    m = n**2
    A = mat(1.*r_[1:m+1]).T*mat(1.*r_[1:n+1])
    A += vstack((eye(n), ones((m-n,n)) ))
    avqr = 0.
    print(A.shape)
    for k in xrange(nrEXP):
        t1 = time.clock()
        Q, R = qr(A)
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[0,k] = avqr
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        Q, R = qr(A, mode='full')
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[1,k] = avqr
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        R = qr(A, mode='r', overwrite_a=True)
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[2,k] = avqr
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        R = qr(A, mode='r')
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[3,k] = avqr
    avqr = 0.
    for k in xrange(nrEXP):
        t1 = time.clock()
        Q, R = qr(A, mode='economic')
        t2 = time.clock()-t1
        avqr += t2

    avqr /= nrEXP
    print avqr
    qrtimes[4,k] = avqr
    k += 1

import matplotlib.pyplot as plt
```

```
plt.plot(sizes, qrtimes[0], "-o", label=r"qr")
plt.plot(sizes, qrtimes[1], "-d", label=r"qr(mode='full')")
plt.plot(sizes, qrtimes[2], "-s", label=r"qr(mode='r', overwrite\_a=True)")
plt.plot(sizes, qrtimes[3], "-*", label=r"qr(mode='r')")
plt.plot(sizes, qrtimes[4], "-+", label=r"qr(mode='economic')")
plt.legend(loc="upper left")
plt.savefig("time_qr.png")
```



```
from numpy import r_, mat, vstack, eye, ones, zeros
from scipy.linalg import qr
import timeit

def qr_full():
    global A
    Q, R = qr(A)

def qr_econ():
    global A
    Q, R = qr(A, mode='economic')

def qr_ovecon():
    global A
    Q, R = qr(A, mode='economic', overwrite_a=True)

def qr_r():
    global A
    R = qr(A, mode='r')

def qr_ovr():
    global A
    R = qr(A, mode='r', overwrite_a=True)

nrEXP = 4
sizes = 2**r_[2:6]
qrtimes = zeros((5,sizes.shape[0]))
k = 0

for n in sizes:
    print 'n=', n
    m = n**2#4*n
    A = mat(1.*r_[1:m+1]).T*mat(1.*r_[1:n+1])
    A += vstack((eye(n), ones((m-n,n)) ))

    t = timeit.Timer('qr_full()','from __main__ import qr_full')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[0,k] = avqr

    t = timeit.Timer('qr_econ()','from __main__ import qr_econ')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[1,k] = avqr

    t = timeit.Timer('qr_ovecon()','from __main__ import qr_ovecon')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[2,k] = avqr

    t = timeit.Timer('qr_r()','from __main__ import qr_r')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[3,k] = avqr


    t = timeit.Timer('qr_ovr()','from __main__ import qr_ovr')
    avqr = t.timeit(number=nrEXP)/nrEXP
    print avqr
    qrtimes[4,k] = avqr

    k += 1

import matplotlib.pyplot as plt

v4 =  qrtimes[1,1]* (sizes/sizes[1])**4
v6 =  qrtimes[0,1]* (sizes/sizes[1])**6

plt.loglog(sizes, qrtimes[0], "-o", label=r"qr")
plt.loglog(sizes, qrtimes[1], "-d", label=r"qr(mode='economic')")
plt.loglog(sizes, qrtimes[2], "-s", label=r"qr(mode='economic', overwrite\_a=True)")
plt.loglog(sizes, qrtimes[3], "-*", label=r"qr(mode='r')")
plt.loglog(sizes, qrtimes[4], "-+", label=r"qr(mode='r', overwrite\_a=True)")
plt.loglog(sizes, v4, "-.k", label=r"$O(n^4)$")
plt.loglog(sizes, v6, "--k", label=r"$O(n^6)$")
plt.legend(loc=2)
plt.xlabel(r"$n$")
plt.ylabel("time [s]")
plt.savefig("time_qr_2.png")
```



2.8 Gluing Stand-Alone Applications

This section follows closely the approach of Langtangen: Python Scripting for Computational Science and Engineering, chapter 2.3.

2.8.1 Scripting

Suppose you have a stand-alone code for solving the ODE:

$m\frac{d^2u}{dt^2} + b\frac{du}{dt} + cf(u) = A\cos\omega t$

and you wish to have simulation results for varoius choices of the parameters. For instance, the provided

oscillator.f inp run

(gfortran -O3 -o oscillator oscillator.f)

reads the following parameters from the standard input, in the listed order: $m, b, c,$ name of f(y) function, A, $\omega, y_0, t_{stop}, \Delta t$

The values of the parameters may be placed in a file inp: and the program can be run as

oscillator < inp

The output consists of data points $t_i u(t_i)$, a suitable format to plot, e.g. with gnuplot.

Under the link you’ll find fortran, C and python versions of this stand-alon code.

Our first goal is to simplify the user’s interaction with the program; with the script simviz1.py it is possible to adjust the parameters through command-line options:

```
python simviz1.py -m 3.3 -b 0.9 -dt 0.05
```

The result should be png-files and an otional plot on the screen. The files for each such experiment should go in a subdirectory.

```
#!/usr/bin/env python
import sys, math

# default values of input parameters:
m = 1.0; b = 0.7; c = 5.0; func = ’y’; A = 5.0; w = 2*math.pi
y0 = 0.2; tstop = 30.0; dt = 0.05; case = ’tmp1’; screenplot = 1

# read variables from the command line, one by one:
while len(sys.argv) > 1:
    option = sys.argv[1];             del sys.argv[1]
    if   option == ’-m’:
        m = float(sys.argv[1]);      del sys.argv[1]
    elif option == ’-b’:
        b = float(sys.argv[1]);      del sys.argv[1]
    elif option == ’-c’:
        c = float(sys.argv[1]);      del sys.argv[1]
    elif option == ’-func’:
        func = sys.argv[1];          del sys.argv[1]
    elif option == ’-A’:
        A = float(sys.argv[1]);      del sys.argv[1]
    elif option == ’-w’:
        w = float(sys.argv[1]);      del sys.argv[1]
    elif option == ’-y0’:
        y0 = float(sys.argv[1]);     del sys.argv[1]
    elif option == ’-tstop’:
        tstop = float(sys.argv[1]); del sys.argv[1]
    elif option == ’-dt’:
        dt = float(sys.argv[1]);     del sys.argv[1]
    elif option == ’-noscreenplot’:
        screenplot = 0
    elif option == ’-case’:
        case = sys.argv[1];          del sys.argv[1]
    else:
        print sys.argv[0],’: invalid option’,option
        sys.exit(1)

# create a subdirectory:
d = case                # name of subdirectory
import os, shutil
if os.path.isdir(d):    # does d exist?
    shutil.rmtree(d)    # yes, remove old directory
os.mkdir(d)             # make new directory d
os.chdir(d)            # move to new directory d

# make input file to the program:
f = open('%s.i' % case, 'w')
# write a multi-line (triple-quoted) string with
# variable interpolation:
f.write("""
        %(m)g
        %(b)g
        %(c)g
        %(func)s
        %(A)g
        %(w)g
        %(y0)g
        %(tstop)g
        %(dt)g
        """ % vars())
f.close()
# run simulator:
cmd = '../oscillator < %s.i' % case  # command to run
failure = os.system(cmd)
if failure:
    print 'running the oscillator code failed'; sys.exit(1)

# make file with gnuplot commands:
f = open(case + '.gnuplot', 'w')
f.write("""
set title '%s: m=%g b=%g c=%g f(y)=%s A=%g w=%g y0=%g dt=%g';
""" % (case, m, b, c, func, A, w, y0, dt))
if screenplot:
    f.write("plot 'sim.dat' title 'y(t)' with lines;\n")
f.write("""
set size ratio 0.3 1.5, 1.0;
# define the postscript output format:
set term postscript eps monochrome dashed 'Times-Roman' 28;
# output file containing the plot:
set output '%s.ps';
# basic plot command:
plot 'sim.dat' title 'y(t)' with lines;
# make a plot in PNG format:
set term png small;
set output '%s.png';
plot 'sim.dat' title 'y(t)' with lines;
""" % (case, case))
f.close()
# make plot:
cmd = 'gnuplot -geometry 800x200 -persist ' + case + '.gnuplot'
failure = os.system(cmd)
if failure:
    print 'running gnuplot failed'; sys.exit(1)
```

One can easily generate an automatic LaTeX-report or HTML-report containing the values of the parameters and the corresponding pictures.

2.8.2 Conducting Numerical Experiments

Suppose we want to keep most of the parameters fixed and vary one parameter between a minimum and a maximum value; the results should be collected in a HTML-report:

```
python loop4simviz1.py m_min m_max m_increment [ simviz1.py options ]
```

```
#!/usr/bin/env python
"""calls simviz1.py with different m values (in a loop)"""
import sys, os
usage = 'Usage: %s m_min m_max m_increment [ simviz1.py options ]' \
        % sys.argv[0]

try:
    m_min = float(sys.argv[1])
    m_max = float(sys.argv[2])
    dm    = float(sys.argv[3])
except IndexError:
    print usage;  sys.exit(1)

simviz1_options = ' '.join(sys.argv[4:])

html = open('tmp_mruns.html', 'w')
html.write('<HTML><BODY BGCOLOR="white">\n')
psfiles = []  # plot files in PostScript format

m = m_min
while m <= m_max:
    case = 'tmp_m_%g' % m
    cmd = 'python simviz1.py %s -m %g -case %s' % \
          (simviz1_options, m, case)
    print 'running', cmd
    os.system(cmd)
    html.write('<H1>m=%g</H1> <IMG SRC="%s">\n' \
               % (m,os.path.join(case,case+'.png')))
    psfiles.append(os.path.join(case,case+'.ps'))
    m += dm
html.write('</BODY></HTML>\n')
```

or let vary any parameter:

```
loop4simviz2.py parameter min max increment [ simviz2.py options ]
```

```
#!/usr/bin/env python
"""
As loop4simviz1.py, but here we call simviz2.py, make movies,
and also allow any simviz2.py option to be varied in a loop.
"""
import sys, os
usage = 'Usage: %s parameter min max increment '\
        '[ simviz2.py options ]' % sys.argv[0]
try:
    option_name = sys.argv[1]
    min = float(sys.argv[2])
    max = float(sys.argv[3])
    incr = float(sys.argv[4])
except:
    print usage;  sys.exit(1)

```

```
simviz2_options = ' '.join(sys.argv[5:])

html = open('tmp_%s_runs.html' % option_name, 'w')
html.write('<HTML><BODY BGCOLOR="white">\n')
psfiles = []    # plot files in PostScript format
pngfiles = []   # plot files in PNG format

value = min
while value <= max:
    case = 'tmp_%s_%g' % (option_name,value)
    cmd = 'python simviz2.py %s -%s %g -case %s' % \
          (simviz2_options, option_name, value, case)
    print 'running', cmd
    os.system(cmd)
    psfile = os.path.join(case,case+'.ps')
    pngfile = os.path.join(case,case+'.png')
    html.write('<H1>%s=%g</H1> <IMG SRC="%s">\n' \
               % (option_name, value, pngfile))
    psfiles.append(psfile)
    pngfiles.append(pngfile)
    value += incr
cmd = 'convert -delay 50 -loop 1000 %s tmp_%s.gif' \
      % (' '.join(pngfiles), option_name)
print 'converting PNG files to animated GIF:\n', cmd
os.system(cmd)
html.write('<H1>Movie</H1> <IMG SRC="tmp_%s.gif">\n' % option_name)
html.write('</BODY></HTML>\n')
html.close()
```

2.9 Input / Output of numerical data

In this section we would like to focus on a last but very important point once you will do real world numerical simulations. It's all about how to store numerical simulation data into files in an efficient and portable way. We will give a short introduction to a general purpose file format called HDF, the *Hierarchical Data Format*, which is carefully designed to store and organize large amounts of numerical data [6].

2.9.1 Introduction to HDF

When we said that HDF is a file format this is not the whole truth. HDF is rather some kind of scheme for creating file formats. Maybe we can say that in some aspects it's similar to e.g. html which is not a file format but a language for describing data.

A HDF file has the following file structure and includes only two major types of objects:

- Datasets, which are multidimensional arrays of a homogenous type
- Groups, which are container structures which can hold datasets and other groups

This results in a truly hierarchical, filesystem-like data format. In fact, resources (for now read: data sets) in an HDF file are even accessed using the POSIX-like syntax "/path/to/resource". Metadata is stored in the form of user-defined, named attributes attached to groups and datasets.

[6] The people responsible for taking care of the HDF standard have their home at http://www.hdfgroup.org/.

In addition HDF includes an type system for specifying data types (are the numbers here 32 bit real floats or 64 bit complex floats or ...), and dataspace objects which represent selections over dataset regions. The API is object-oriented with respect to datasets, groups, attributes, types, dataspaces and property lists.

By the way, currently there exist two major versions of HDF, HDF4 and HDF5, which differ significantly in design and API. We will only deal with HDF5 here.

2.9.2 Handling HDF files with python

Until now we just talked about the ideas of HDF. Let's now look how to work with HDF files. The library that handles HDF files has interfaces to many programing languages and there are python bindings too. Here we will present the `h5py` python package. Let's just cite the official description from the `h5py` developers:

"The HDF5 library is a versatile, mature library designed for the storage of numerical data. The h5py package provides a simple, Pythonic interface to HDF5. A straightforward high-level interface allows the manipulation of HDF5 files, groups and datasets using established Python and NumPy metaphors."

The `h5py` project is located at http://www.h5py.org/ where you can find the python package as well as a very detailed user reference manual and many examples of usage.

2.9.3 A first example

Ok, enough theory by now, let's go to a simple hands-on example. Hopefully the code is self-explanatory with all these comments in place.

```python
import numpy as np
import scipy as sp
import pylab as pl
import h5py as hdf

# A grid
x = np.linspace(-sp.pi,sp.pi,100)
X, Y = np.meshgrid(x,x)

# Some values
U = sp.sin(X)
V = sp.cos(X)


# Save the data into a hdf5 file
filename = "data.hdf5"

# Create and open the file with the given name
outfile = hdf.File(filename)

# Store some data under /axisgrid with direct assignment of the data x
outfile.create_dataset("axisgridx", data=x)

# Create a group for storing further data under /grid/*
grp_grid = outfile.create_group("grid")

# Store some data under /grid/grid* with direct assignment of the data X and Y
grp_grid.create_dataset("gridx", data=X)
grp_grid.create_dataset("gridy", data=Y)

# Store the vector field values under another group called /values
```

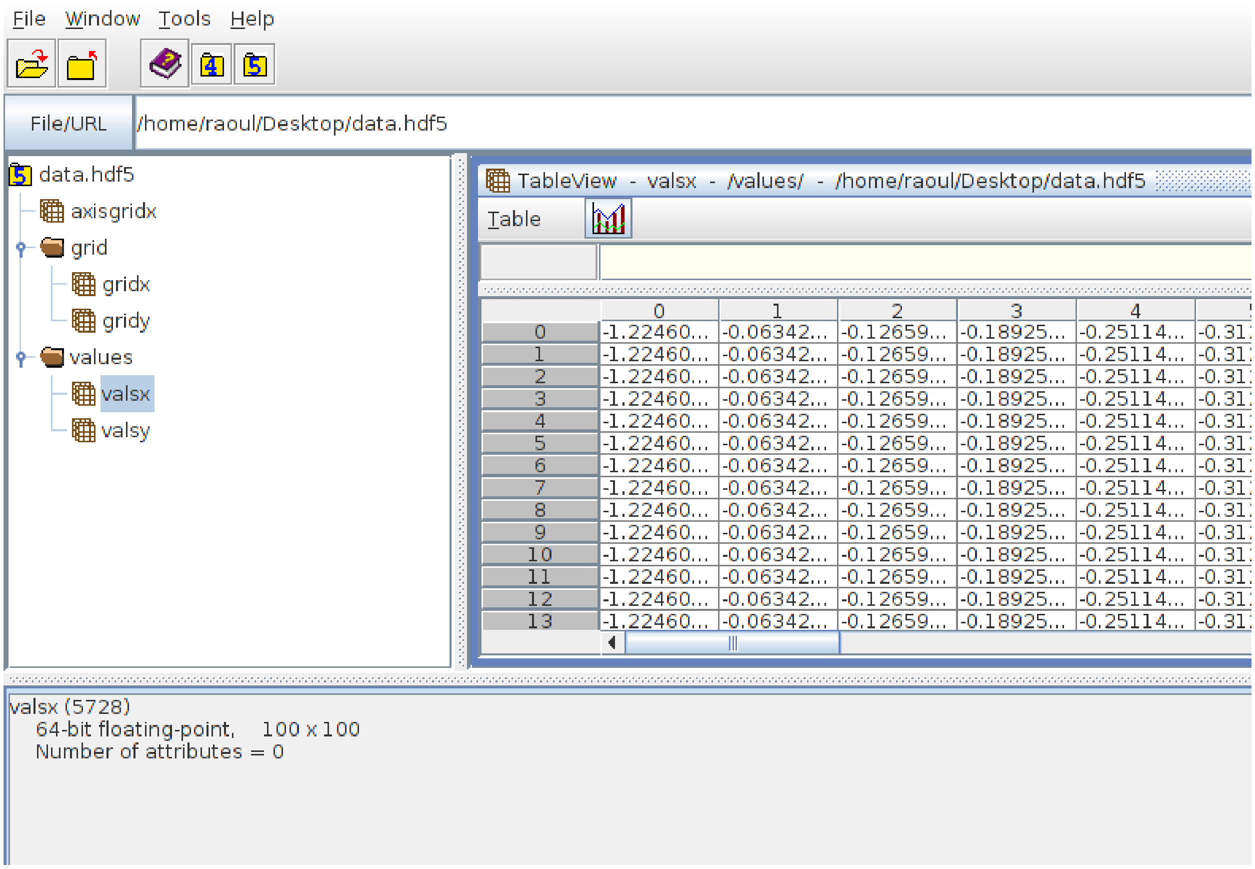
```
grp_vals = outfile.create_group("values")

# Prepare space for storing the values but do not assign values yet
grp_vals.create_dataset("valsx", U.shape, np.floating)
grp_vals.create_dataset("valsy", V.shape, np.floating)

# Now assign the data for U and V and show slicing via ":" and ellipsis "..."
outfile["/values/valsx"][:] = U
outfile["/values/valsy"][...] = V

# And close the file
outfile.close()


# Open the file again, now in read only mode (we don’t want to write data anymore!)
infile = hdf.File(filename, "r")

# Get the data, but omit every second value, just to show off some slicing
a = infile["/grid/gridx"][::2,::2]
b = infile["/grid/gridy"][::2,::2]
c = infile["/values/valsx"][::2,::2]
d = infile["/values/valsy"][::2,::2]

# Plot the data as usual
pl.figure()
pl.quiver(a,b, c,d)
pl.savefig("hdf_example.png")
```

2.9.4 HDFView

The program HDFView is a tool for browsing and editing HDF files. Using HDFView, you can view the internal file hierarchy in a tree structure, create new files, add or delete groups and datasets, view and modify the content of a dataset and much more. There are even some basic plotting facilities which can serve for a first glance at the numerical data.

You can download hdfview from the URL http://www.hdfgroup.org/hdf-java-html/hdfview/. The installation should pose no problems, the software is written in the Java language and available for various operating systems.

If we open the file created by the above example with the program hdfview then we will see something like the following image. On the left there is the structural tree shown with its two groups grid and *values*‘ and the five leaves containing the data. On the right the actual values of the data node valsx is shown as a two-dimensional table (remember that the data array was two-dimensional). And on the bottom we see some information about the data currently examined node, for example the data type (64 bit floats in this case) and the size of the array (here 100 times 100 entries).



File/URL: /home/raoul/Desktop/data.hdf5

TableView - valsx - /values/ - /home/raoul/Desktop/data.hdf5

	0	1	2	3	4	5
0	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
1	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
2	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
3	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
4	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
5	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
6	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
7	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
8	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
9	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
10	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
11	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
12	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31
13	-1.22460...	-0.06342...	-0.12659...	-0.18925...	-0.25114...	-0.31

```
valsx (5728)
    64-bit floating-point,    100 x 100
    Number of attributes = 0
```

Of course you can do much more with the HDF data format and we can only show the tip of this iceberg. For example, HDF supports transparent compression of the data, further descriptive attributes for nodes, special sorting and indexing of data for fast retrieval and many things more. For more information you should read the `h5py` user guide http://docs.h5py.org/en/2.3/.

2.10 Combining Python with Fortran, C and C++

This section follows closely the approach of Langtangen: Python Scripting for Computational Science and Engineering, chapter 5.

When is integration of Python with Fortran, C, or C++ of interest?

- migration of slow parts of a Python code to Fortran or C/C++
- access to existing numerical code (from Python)

Typically: user interfaces, i/O, report generation and management is in Python, while the computationally intensive functions are in Fortran/C/C++

A source of troubles: Fortran/C/C++ has strong typing rules, while in Python variables are typeless...

Needs writing wrapper code: each function need a Python wrapper. Automatic generation: SWIG for C/C++, f2py for Fortran

2.10.1 Scientific Hello World Examples

Pure Python implementation:

```python
#!/usr/bin/env python
"""Pure Python Scientific Hello World module."""
import math, sys

def hw1(r1, r2):
    s = math.sin(r1 + r2)
    return s

def hw2(r1, r2):
    s = math.sin(r1 + r2)
    print 'Hello, World! sin(%g+%g)=%g' % (r1,r2,s)
```

with application script:

```python
#!/usr/bin/env python
"""Scientific Hello World script using the module hw."""
import sys
from hw import hw1, hw2
try:
    r1 = float(sys.argv[1]);  r2 = float(sys.argv[2])
except IndexError:
    print 'Usage:', sys.argv[0], 'r1 r2'; sys.exit(1)
print 'hw1, result:', hw1(r1, r2)
print 'hw2, result: ',
hw2(r1, r2)
```

We intend to migrate the functions hw1 and hw2 in the hw module to Fortran/C/C++. We eill also onvolve a third function hw3, which is a version of hw1 where s is an output argument, in call by reference style, and not a return variable. A pure implementation of hw3 has no meaning.

2.10.2 Combining Python and Fortran

```fortran
      program hwtest
      real*8 r1, r2 ,s
      r1 = 1.0
      r2 = 0.0
      s = hw1(r1, r2)
      write(*,*) 'hw1, result:',s
      write(*,*) 'hw2, result:'
      call hw2(r1, r2)
      call hw3(r1, r2, s)
      write(*,*) 'hw3, result:', s
      end

      real*8 function hw1(r1, r2)
      real*8 r1, r2
      hw1 = sin(r1 + r2)
      return
      end

      subroutine hw2(r1, r2)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      write(*,1000) 'Hello, World! sin(',r1+r2,')=',s
 1000 format(A,F6.3,A,F8.6)
      return
      end
```

```
C     special version of hw1 where the result is
C     returned as an argument:

      subroutine hw3_v1(r1, r2, s)
      real*8 r1, r2, s
      s = sin(r1 + r2)
      return
      end

C     F2py treats s as input arg. in hw3_v1; fix this:

      subroutine hw3(r1, r2, s)
      real*8 r1, r2, s
Cf2py intent(out) s
      s = sin(r1 + r2)
      return
      end

C     test case sensitivity:

      subroutine Hw4(R1, R2, s)
      real*8 R1, r2, S
Cf2py intent(out) s
      s = SIN(r1 + r2)
      ReTuRn
      end

C end of F77 file
```

f2py comes automaticaly with numpy; we make a subdirectory f2py-hw and run f2py in this subdirectory:

```
f2py -m hw -c ../hw.f
```

The reuslt is the module hw.so which may be loaded into Python by an ordinary import statement.

- -m option specifies the name of the extension module
- -c indicates that f2py should compile and link the module
- –fcompiler=gfortran specifies the compiler to use (less error messages)
- f2py -c –help-fcompiler to see a list of Fortran compilers installed
- f2py -c –help-compiler to see a list of C compilers installed

Test if everything went fine:

```
python -c 'import hw'
```

We can test now the new module with the previous hwa.py program, who does not know if the module hw is written in Python or Fortran.

When dealing with more complicated libraries, one may want to create Python interfaces to only some of the functions, e.g.:

```
f2py -m hw -c --fcompiler=gfortran ../hw.f only: hw1 hw2 :
```

-h hw.pyf –overwrite-signature options makes f2py to write the module interfaces to a file hw.pyf that you can adjust to your needs:

```
f2py -m hw -h hw.pyf --overwrite-signature ../hw.f
f2py -m hw -h hw.pyf --overwrite-signature ../hw.f only: hw1 hw2 :
```

The function hw3_v1 enables us to see one difficulty of mixing Fortran and Python. The rsult of the computations is stored in the output variable s. Since Fortran employs the call by reference for all the arguments, any change to an argument is visible in the calling code. Let us see how the interface looks like:

Bei dfault, f2py treates r1, r2, and s as input argumnets; calling hw3_v1 shows that the value of the Fortran s variable is not returned to the Python s variable in the call. The remdey is to tell f2py that s is an output parameter. We do this by replacing in the hw.pyf file

```
real*8 :: s
```

with:

```
real*8 intent(out) :: s
```

as in subroutine hw3. The directives intent(in) and intent(out) specify input or output variables, while intent(in,out) are employed for variables for both input and output.

Compiling and linking the hw module with th emodified interface specification in hw.pyf are now performed by:

```
f2py -c --fcompiler=gfortran hw.pyf ../hw.f
```

Note thet f2py changes the interface to become more Pythonic, i.e. we write in Python:

```
s = hw3(r1,r2)
```

For a Fortran routine:

```
subroutine somef(in1, in2, o1, o2, o3, o4, io1)
```

where in1, in2 are input variables, o1, o2, o3, o4 are output variables and io1 is an input/output variable, the generated Python interface will have i1, i2, io1 as input arguments and o1, o2, o3, o4, io1 is returend as a tuple:

```
o1, o2, o3, o4, io1 = somef(i1, i2, io1)
```

As an alternative to the modification of the .pyf file, we may insert an intent specification as a special Cf2py comment in the Fortran source code file. The safest way of writing hw3 is to specify the input/output nature of all the function arguments:

```
      subroutine hw3(r1,r2,s)
      real*8 :: r1, r2, s
Cf2py intent(in) r1
Cf2py intent(in) r2
Cf2py intent(out) s
      s = sin(r1 + r2)
      return
      end
```

2.10.3 Building an Extension Module with Distutils

The standard way for buiilding and installing Python modules uses Python's Distutils tool which comes with the standard Python distribution. An enhanced version of Distutils with better support for Fortran code comes with numpy. We have to create a script setup.py which calls a function steup in Distutils. Then, in order to build the extension module:

```
python setup.py build
```

or to build and install:

```
python steup.py install
```

To build in the current working directory:

```
python setup.py build build_ext --inplace
```

An example of steup.py file

```
from numpy.distutils.core import Extension, setup

setup(name='hw',
      ext_modules=[Extension(name='hw', sources=['../hw.f'])],
      )
```

2.11 Integrated development environment

Personally, I use my preferred editor *emacs* and the improved interactive environment *ipython* for the design and test of the scripts; the actual tutorial assumes this working style. You can of course use any text editor you like. You might consider also one of the following alternatives to (editor and *ipython* shell):

- spyder (similar to the Matlab-environment)
- ipython notebook (Notebook similar in style to Mathematica)
- eric (similar to the Eclipse-environment)
- PyCharm (another python IDE)

The latter two are not specially focussed on scientific computing.

> **Warning:** Please be aware that the *only* supported setup on the exam computers consists of an editor (probably *gedit*) and the *ipython* interpreter running in a shell. Hence you should be familiar with that too. Other tools might or might not be installed and their use is at your own risk.

This document as pdf

CHAPTER

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*