Exploring Python as Matlab alternative

A scientist view on python

Jochen Schulz

Georg-August Universität Göttingen

Aufbau

Programming for a Scientist

Usual tasks

- Create or gather data (simulation, experiment)
- postprocessing of data
- visualization and validation
- publish and communicate results

So we want a *high-level* programming language:

- programming is simple
- use elements which are there already
- good prototyping and debugging (interaction)
- hopefully only one tool for all problems

MATLAB

- MATLAB stands for Matrix laboratory; intially matrix calculations.
- Interactive system for numerical calculations and visualization (scripting language).

Advantages

- many tools for visualization.
- many additional toolboxes (Symb. Math T., PDE T., Wavelet T.)
- mature and integrated GUI.

Disadvantages

- costly.
- other tasks than matrix calculations can be difficult.

Python: NumPy, SciPy, SymPy

- modular scripting language.

Advantages

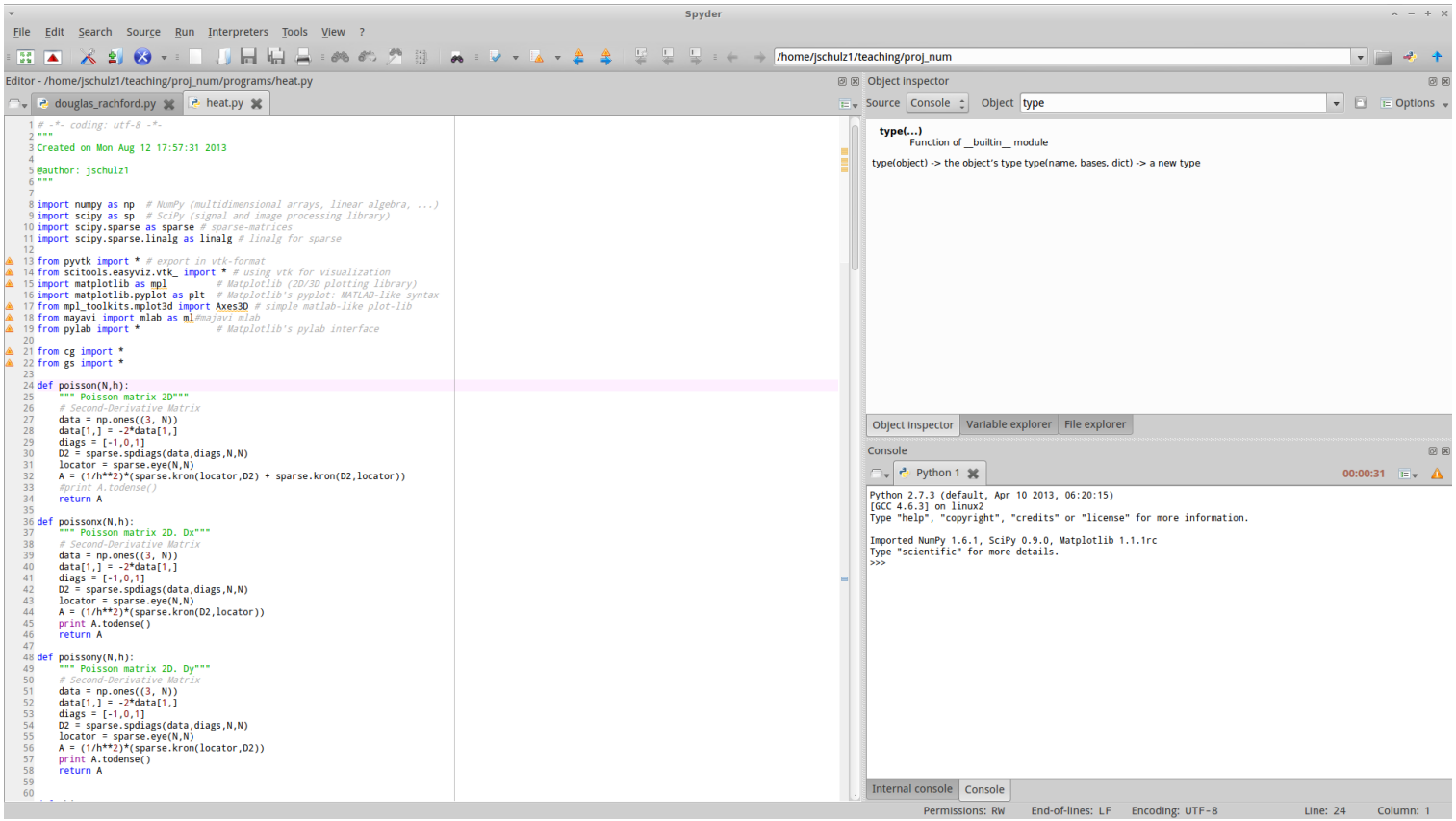
- many modules for scientific computations.
- clean code structure.
- much more modules for non-scientific tasks (for example helpful in I/O).
- support different coding paradigm (objectoriented, imperative, functional)
- free and open source.

Disadvantages

- Usage a little bit more complicated (Spyder,ipython).
- not all the special features of other software.
- with flexibility comes complexity.

Aufbau

Spyder GUI



- Editor: data manipulation.
- Console: command window and standard output.
- Object Inspector: help and list of variables.
- Grafik: separate windows.

Lists and tuples

- a list is marked with [..,..] (has order, mutable)

```
list = [21,22,24,23]
list.sort(); list
```

```
[21, 22, 23, 24]
```

- a tuple is marked with (..,..) (has structure, immutable)

```
tuple = (list[0], list[2])
tuple, tuple[0]
```

```
(( 21, 24), 21)
```

- Lists of integers ranging from a to b

```
range(a,b+1)
```

list comprehensions

```
[i**2 for i in range(4)]
```

```
[0, 1, 4, 9]
```

Dictionaries

- index can hold almost arbitrary objects.
- they are good for big data, because indexing is very fast.
- index is unique
- iteration:

```
d = {'a': 1, 'b':1.2, 'c':1j}
for key, val in d.iteritems():
  print key,val
```

```
a 1
c 1j
b 1.2
```

Functions

normal

```
def fun (arg1,arg2=<defaultvalue>,... ,*args,**kwargs)
    <code>
    return <returnvalue>
```

anonymous

```
lambda arg: <codeline>
```

- args: Tuple of input-arguments
- kwargs: Dictionary of named input-arguments
- *: unpacks tuple in comma separated list of arguments
- **: unpacks dictionary in list of named arguments
- arguments with defaultvalue are optional
- arguments with names can have arbitrarily order

a bit functional programming

iterators : objects which generates a next element when asked for.

```
import itertools
```

- ```
  imap(function,list)
  ```

  generates an iterator which calls function(x) for all elements from list .

- ```
  ifilter(function,list)
  ```

  generates an iterator which returns only elements x, for which function(x) returns True.

```
list(ifilter(lambda x: mod(x,3) == 0,[1,4,6,24]))
```

```
[6, 24]
```

non-iterators: map, filter

Vectors and matrixes - NumPy arrays

vectors

```
np.array([1,2,4])
```

matrix

```
np.array([[1,2,4],[2,3,4]])
np.matrix([[1,2,4],[2,3,4]])
```

- zeros((n,m)): $(n \times m)$- matrix entries all 0.
- ones((n,m)): $(n \times m)$- matrix entries all 1.
- tile (A,(n,m): block matrix with $(n \times m)$ blocks of A

Remark:

- matrix-multiply is achieved with dot(a,b) and * is elementwise!
- mostly use array not matrix, because it is the standard-object.

Slicing

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
```

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

one entry

```
A[1,0]
```

```
4
```

blocks

```
A[1:3,0:2]
```

```
        4       5
        7       8
```

lines

```
A[,:]
```

```
        4       5       6
```

individual entries/lines

```
A[(0,-1),:]
```

```
        1       2       3
        7       8       9
```

Fancy indexing, copies and views

fancy indexing is indexing an array with integer or boolean arrays.

```
a = [i**2 for i in range(1,10)]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
a[a % 3 ==   0]
```

```
array([ 9, 36, 81])
```

Remark:

- normal indexing (slicing) creates views on the data. It shares the same memory, e.g.

  ```
  b = a[::3]
  ```

  you can force the copy:

  ```
  b = a[::3].copy()
  ```

- fancy indexing creates copies from the data.

Advanced arrays

Data types for arrays (dtypes)

- Types: int , uint , float , complex, string ...
- Character Codes : '<typechar><bytes>' : 'i4', 'u4', 'f8', 'c16', 'S25'
- typecast: <object>.astype(<type>)

Example:

```
b = np.array([5, 12345, 'test'],dtype='S4')
```

```
array(['5', '1234', 'test'], dtype='|S4')
```

Defining types

```
dt = np.dtype('i4',(2,2)) # 2x2 integer array
```

structured arrays

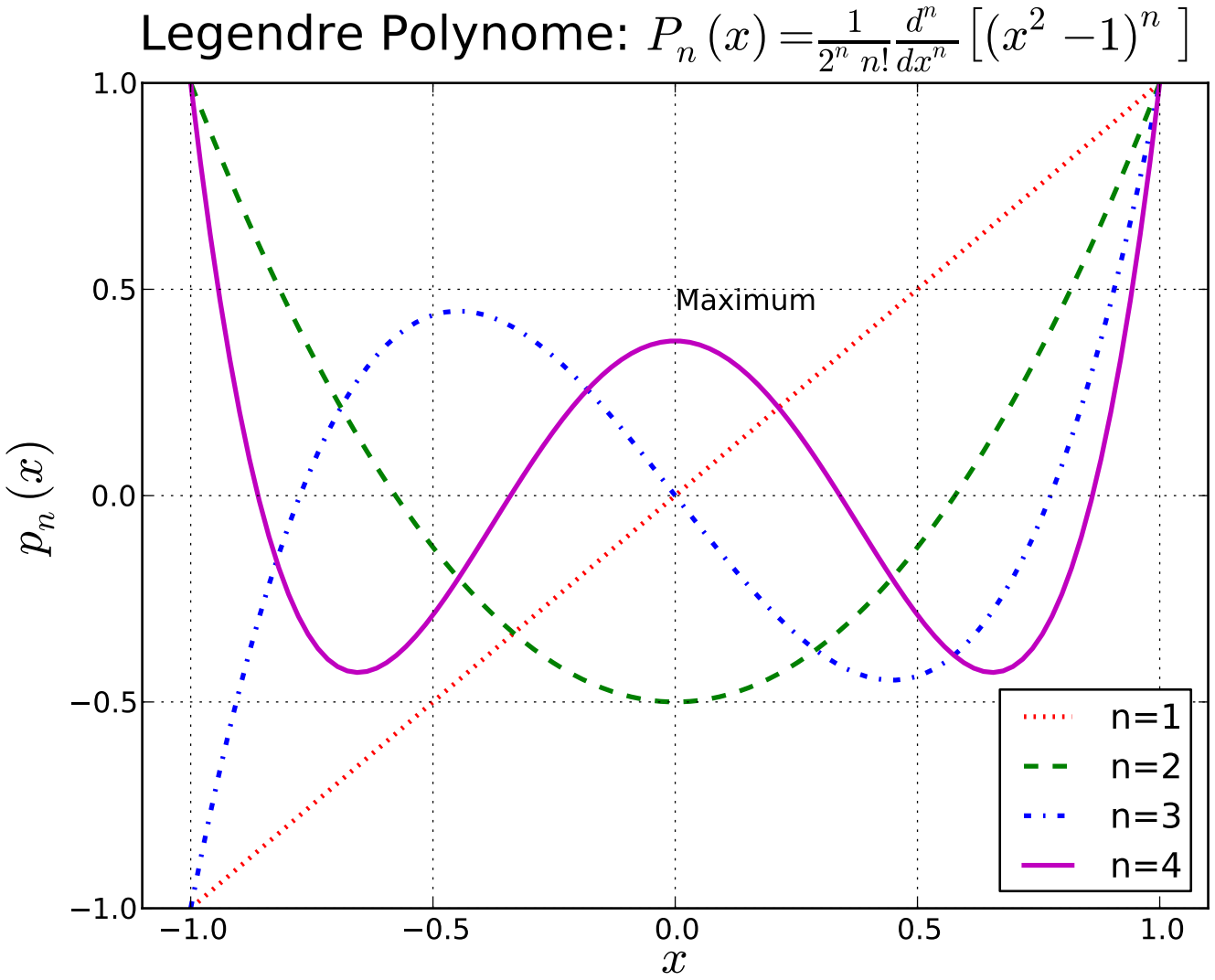
```
ct = zeros(6, dtype=[('name','S40'),('pop','u4') ])
ct[0]['name'] = 'Auckland'
ct[0]['pop'] = 1418000; ct[0]
```

```
('Auckland', 1418000L)
```

2D visualization via example legendre polynomials

```
import numpy as np  # NumPy
import matplotlib as mpl          # Matplotlib (2D/3D)
import matplotlib.pyplot as plt   # Matplotlibs pyplot
from pylab import *               # Matplotlibs pylab
x = linspace(-1,1,100)
p1 = x; p2 = (3./2)*x**2-1./2
p3 = (5./2)*x**3-(3./2)*x
p4 = (35./8)*x**4 -(15./4)*x**2 + 3./8
plot(x,p1,'r:',x,p2,'g--',x,p3,'b-.',x,p4,'m-',linewidth
    =2)
title('Legendre polynomials: $P_n(x) = \frac{1}{2^nn!}\
    frac{d^n}{dx^n}\left[(x^2-1)^n\right]$ ', fontsize=
    15)
xlabel( '$x$' , fontsize= 20)
ylabel ( '$p_n(x)$' , fontsize=20)
text( 0, 0.45 , 'Maximum' )
legend (('n=1', 'n=2', 'n=3', 'n=4'), loc='lower right')
grid('on'), box('on'), xlim( (-1.1 , 1.1) )
```

legendre polynomials



together with loadtxt - bye bye gnuplot

```
array = np.loadtxt(fname,delimiter=None,comments='#')
```

- fname: filename.
- delimiter : delimiter. e.g. ',' in comma separated tables. Default is space.
- comments: character for comments. In python e.g. '#'.
- array: return as (multidimensional) array.

more flexible: np.genfromtxt()

Saving figures:

```
plt.savefig('legendre.pdf')
```

Remarks:

- you can use TEX for rendering all Texts with plt.rc('text', usetex=True)
- You even can save the figures in pgf/TikZ !

Debugging

- pyflakes : syntax-checker (Spyder has it built-in)
- pdb: python debugger (Spyder has it built-in)

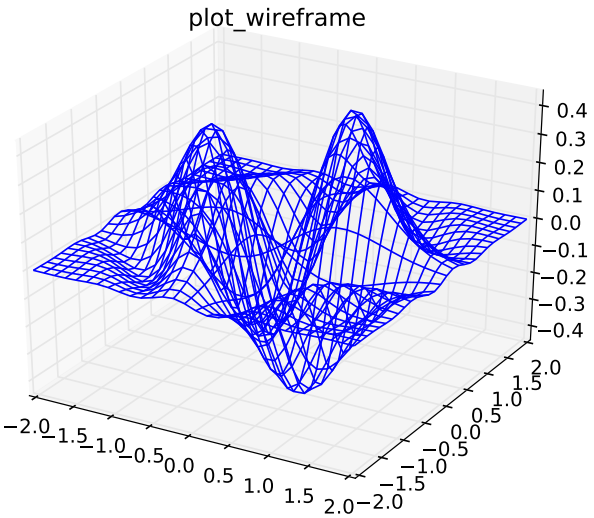
In IPython you can use

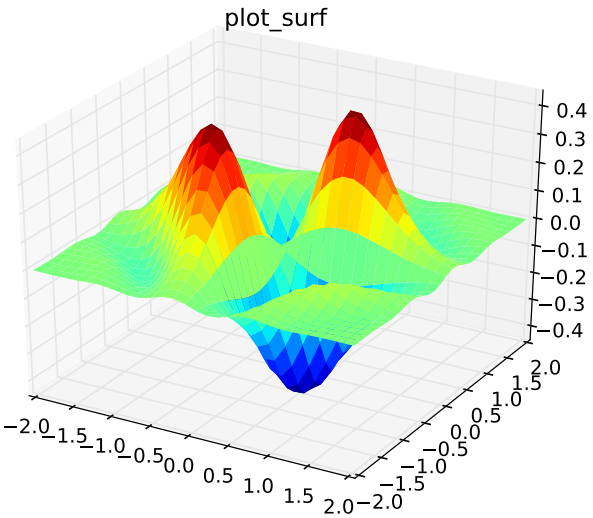
```
%run -d <script.py>
```

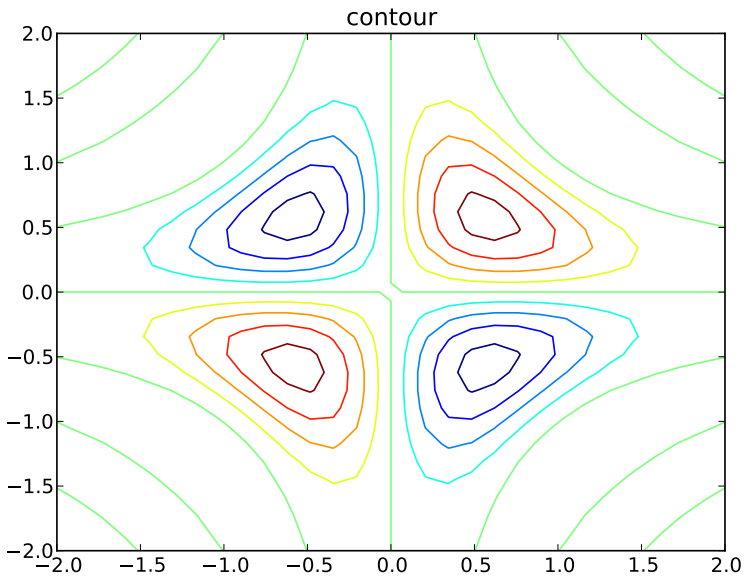
to run the debugger

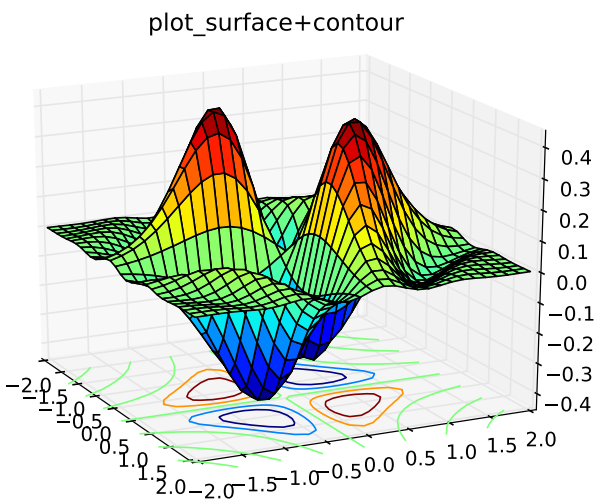
Aufbau

3D: functionplots (basic)







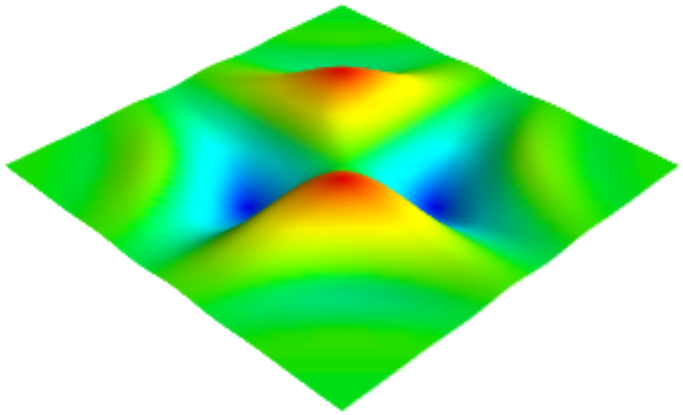


3D: functionsplots - implementation

```
x = linspace(-2,2,30)
y = linspace(-2,2,30)
[X,Y] = meshgrid(x,y)
Z = exp(-X**2-Y**2)*sin(pi*X*Y)
fig=figure()
ax = fig.add_subplot(2, 2, 1, projection='3d')
ax.plot_wireframe(X,Y,Z),title('plot_wireframe')

ax = fig.add_subplot(2, 2, 2, projection='3d')
ax.plot_surface(X,Y,Z,rstride=1,cstride=1,cmap=cm.jet,
    linewidth=0),title('plot_surface')

subplot(2, 2, 3)
contour(X,Y,Z,10), title('contour')

ax = fig.add_subplot(2, 2, 4, projection='3d')
ax.plot_surface(X,Y,Z,rstride=1,cstride=1,cmap=cm.jet)
ax.contour(X, Y, Z, zdir='z', offset=-0.5)
ax.view_init(20,-26),title('plot_surface + contour')
```

Mayavi mlab!

```
from mayavi import mlab as ml #majavi mlab
ml.surf(X.T,Y.T,Z)
title('plot_surf (mayavi)')
```



4D: Mayavi mlab

Slices

```
ml.pipeline.image_plane_widget(ml.pipeline.scalar_field(V
    ), plane_orientation=<'x_axes'|'y_axes'|'z_axes'>,
    slice_index=<idx>)
```

- V: function values $V(i)$ for $(X(i), Y(i), Z(i))$.
- plane_orientation : slices through x-/y-/z- axes
- slice_index : index in matrices (no direct coordinates)

Volume rendering

```
ml.pipeline.volume(ml.pipeline.scalar_field(V))
```

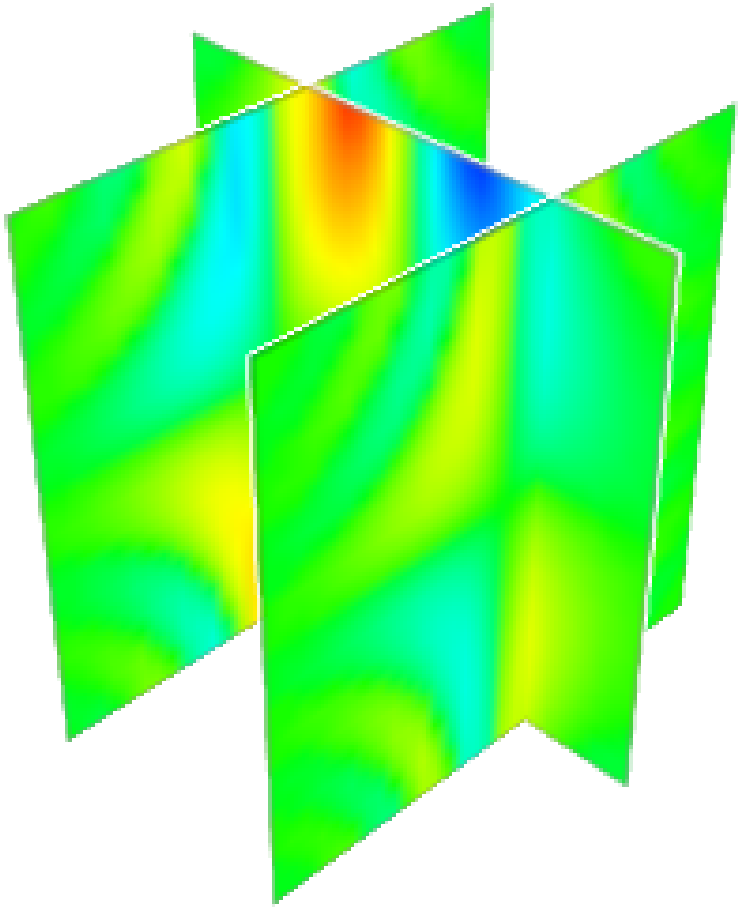
isosurfaces

```
ml.contour3d(V)
```

Example (grid-generating using broadcasting):

```
X, Y, Z = np.ogrid[-2:2:20j,-2:2:20j,-2:2:20j]
V = exp(-X**2-Y**2) * sin(pi*X*Y*Z)
```

4D: Example slice



Aufbau

system of linear equations

Let $A \in \mathbb{C}^{n \times n}$ and $b \in \mathbb{C}^n$. the system of linear equations

$$Ax = b$$

is solved (directly) with solve(A,b).
some iterative methods:

- gmres (generalized minimum residual)
- cg (preconditioned conjugate gradient)
- bicgstab (biconjugate gradients stabilized)
- ...

100 Digits-Challenge

Problem

Let A be a 20.000×20.000 matrix, which entries are all zero except of the primes $2, 3, 5, 7, \ldots, 224737$ on the diagonal and ones in all entries a_{ij} with $|i - j| = 1, 2, 4, 8, \ldots, 16384$.
What is the $(1, 1)$ entry of A^{-1}?

```
n = 20000
primes = [x for x in range(2,224738) if isPrime(x)]
A = sparse.spdiags(primes,0,n,n)\
+sparse.spdiags(np.ones((15,n)),[2**x for x in range
    (0,15)],n,n)\
+sparse.spdiags(np.ones((15,n)),[-2**x for x in range
    (0,15)],n,n)
b = np.zeros(n)
b[0] = 1
x0 = sparse.linalg.cg(A,b)
```

Ordinary differential equations

```
r = scipy.integrate.ode(f[,jac])
```

- f: right hand side: $y'(t) = f(t, y)$
- jac: (optional) jacobian matrix
- r.set_integrator(<name>[,<params>]): sets solver <name> with parameters <params>.
- r.set_initial_value(y[, t]): sets initial value.
- r.integrate(t): solves for $y(t)$ and sets new initial value.
- r.succesful(): boolean value for success.

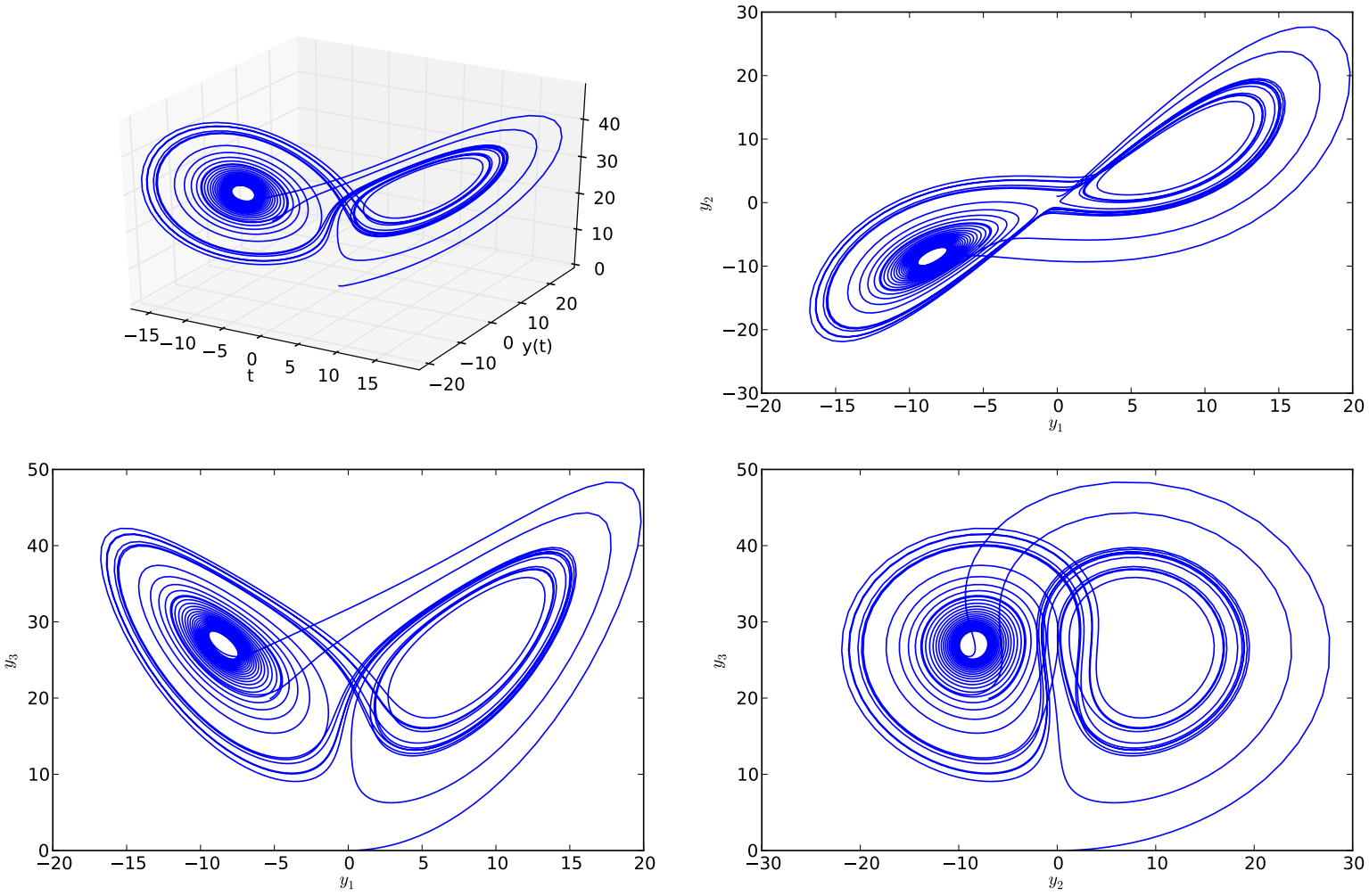
Lorenz equations

$$
\begin{aligned}
\frac{d}{dt}y_1(t) &= 10(y_2(t) - y_1(t)) \\
\frac{d}{dt}y_2(t) &= 28y_1(t) - y_2(t) - y_1(t)y_3(t) \\
\frac{d}{dt}y_3(t) &= y_1(t)y_2(t) - 8y_3(t)/3
\end{aligned}
$$

Lorenz equations

```
def lorenz_rhs(t,y):
    return array([[10*(y[1]-y[0])], [28*y[0]-y[1]-y[0]*y
        [2]], [y[0]*y[1]-8*y[2]/3]])
y = array([0,1,0])
r = ode(lorenz_rhs)
r.set_initial_value(y, 0)
r.set_integrator('dopri5',atol=1e-7,rtol=1e-4)
tmax = 30,dt = 0.01,t=[]
while r.successful() and r.t < tmax:
    r.integrate(r.t+dt)
    t.append(r.t)
    y = vstack( (y, r.y) )
fig = figure(figsize=(16,10))
ax = fig.add_subplot(2, 2, 1, projection='3d')
ax.plot(y[:,0],y[:,1],y[:,2]),xlabel('t'), ylabel('y(t)')
subplot(2,2,2),plot(y[:,0],y[:,1]), xlabel('y_1')
subplot(2,2,3),plot(y[:,0],y[:,2]), xlabel('y_1')
subplot(2,2,4),plot(y[:,1],y[:,2]), xlabel('y_2')
```

Lorenz-equations



PDE - Heat equation

Given a rectangular Domain $\Omega \subset \mathbb{R}^2$ and a time dependent function $u(x,t), x \in \Omega, t \in \mathbb{R}^+$ the heat equation is given as:

$$\frac{\partial u}{\partial t} - \alpha \Delta u = 0 \text{ in } \Omega$$

with a constant $\alpha \in \mathbb{R}$. Given are the dirichlet boundary conditions

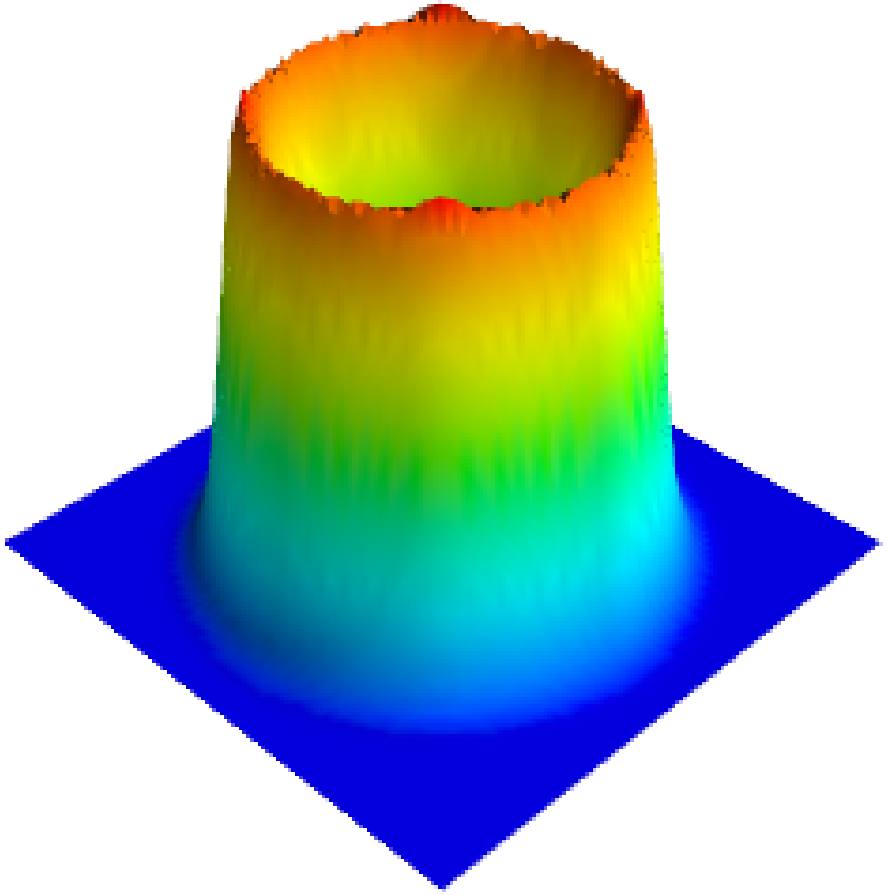
$$u = R, \text{ on } \partial\Omega$$

with a function $R : \partial\Omega \mapsto C(\partial\Omega)$. At $t = 0$ is

$$u(x,0) = f(x), \forall x \in \Omega.$$

with a arbitrary but fixed initial function $f : \mathbb{R}^2 \mapsto \mathbb{R}$.

heat equation - Exercise!



general non-linear solver

```
fsolve(func, x0, args=(), fprime=None, full_output=0,
    xtol=1.49012e-08, maxfev=0 )
```

Example:

```
from scipy import optimize
x0 = -5  # initial
f = lambda x: abs(2*x - exp(-x))
res,info,i,mesg = optimize.fsolve(f,x0,xtol=1e-5,
    full_output=True)
print ("res: {} \nnfev: {} \nfvec: {}".format(res,info['
    nfev'],info['fvec']) )
```

```
res: [ 0.35173371]
nfev: 13
fvec: [ -1.50035540e-12]
```

gradient-free optimizer w/o constraints

Find minima wo constraints, multidimensional (Nelder-Mead-Simplex):

```
fmin(func, x0, args=(), xtol=0.0001, ftol=0.0001,
    full_output=0, disp=1)
```

- func: Function handle
- x0: initial vector
- xtol, ftol : tolerance in x and *func*.

Example:

```
optimize.fmin(f,x0)
```

```
Optimization terminated successfully.
         Current function value: 0.000032
         Iterations: 21
         Function evaluations: 42
```

with constraints, multidimensional:

```
fminbound(func, x1, x2, args=(), xtol=1e-05, full_output
    =0, disp=1)
```

Aufbau

profiler and line profiler

the line_profiler (http://pythonhosted.org/line_profiler/) gives you informationen about how long each individual line took.
use the decorator @profile for the function you want to check.

```
@profile
def function ():
```

Then call your python-script with the profiler:

```
kernprof.py -l -v <file.py>
```

or use the profiler which gives informationen about how long each function took.

```
python -m cProfile <script.py>
```

NumPy-tricks

Use sliced arrays instead of loops.

Example: 1D-distance:

```
a = randn(1000000)
d = zeros(a.shape[0]-1)
```

Loop:

```
for i in range(0,len(d)):
    d[i] = a[i]-a[i+1]
```

Numpy-slices:

```
d[:] = a[:-1]-a[1:]
```

pure python loop	0.993635177612 s
numpy slicing	0.00627207756042 s
matlab loop	0.053599 s

Cython

Cython is an extension, which allows to...

- call C functions (from libraries).
- use C++ objects.
- give C-types to variables (and gain performance)
- compile Python code (and thus can speedup parts of your code)

Basic usage:

- generate C-code from Python/Cython-code.
- compile the generated C-code (and use the shared lib as module in Python).

Minimal example

File csin.pyx

```
cdef extern from "math.h":
    double sin(double x)
def csin(arg):
    return sin(arg)
```

- cdef: declare C-types or -functions.
- extern: usage of external library functions.

Remark: the normal python-function is only needed to be able to use it as module in python.

Compile

compiling is done with Distutils
Crate a File 'setup.py' and put the following in (with replaced <extensionname> and <filename>.pyx):

```
from distutils.core import setup, Extension
from Cython.Distutils import build_ext

setup(
        cmdclass={'build_ext': build_ext},
        ext_modules=[Extension("<extensionname>", ["<
            filename>.pyx"])]
    )
```

Generate c-code and compile

```
python setup.py build_ext --inplace
```

Usage

Load module...

```
import csin
```

...and use it

```
print csin.csin(2)
```

Mandelbrot-set

the Mandelbrot-set is the set of points $c \in \mathbb{C}$ for which the sequence $(z_n)_n$ given by

$$z_0 := c, \qquad z_{n+1} = z_n^2 + c, \quad n \in \mathbb{N}$$

is finite.

- Is $|z_n| \geq 2$ then the sequence diverges.
- we use this as stopping criteria.

Mandelbrot: Python

```
def mandel():
    x1 = linspace(-2.1, 0.6, 3001)
    y1 = linspace(-1.1, 1.1, 2001)
    [X,Y] = meshgrid(x1,y1)

    it_max = 50
    Anz = zeros(X.shape)

    C = (X + 1j*Y)
    Z = copy(C) # beware: otherwise it wouldn't be a copy

    for k in range(1,it_max):
        Z = Z**2+C
        Anz += isfinite(Z)
    imshow(Anz)
```

Mandelbrot: Cython

Calculation for one point. will be completely optimized.

```
import numpy as np
cimport numpy as np #  cython-support of numpy
import scipy as sp
from pylab import *
def cython_mandel(double x,double y):
    cdef double z_real = 0.
    cdef double z_imag = 0.
    cdef int i
    cdef int max_iterations=50
    for i in range(0, max_iterations):
        z_real, z_imag = ( z_real*z_real - z_imag*z_imag
            + x,2*z_real*z_imag + y )
        if (z_real*z_real + z_imag*z_imag) >= 4:
            return i
    return max_iterations
```

Mandelbrot: Cython II

Python with C-types. This is much faster than standard python.

```
def mandel_cy(int pointsx, int pointsy):
    cdef np.ndarray[double,ndim=1] x = linspace(-2.1,1.2,
        pointsx)
    cdef np.ndarray[double,ndim=1] y = linspace(-1.1,1.1,
        pointsy)
    cdef np.ndarray[double,ndim=2] z = np.zeros([pointsx,
        pointsy])
    for i in range(0,len(x)):
        for j in range(0,len(y)):
            z[i,j] = cython_mandel(x[i],y[j])
    return z
```

pure python	9.4 s
python-loop	6.0 s
cython-loop	1.3 s
pure matlab	6.6 s

Aufbau

Ways of parallelism

ordered by simplicity and/or most effect.

- under the hood (e.g. BLAS-library while using `dot()`)
- multiple processes (`import multiprocessing`)
- MPI (the most powerful) (e.g. `import mpi4py`)
- multiple threads (but this only viable for non-CPU-bound work)

Remark: You can use MPI and multiprocesses in an interactive way in IPython!

multiprocessing

Quick introduction. Starting 2 processes in a pool:

```
from multiprocessing import Pool
p = Pool(2)
```

use the pool to do a loop in parallel:

```
def f(x):
    return sin(x) + x + exp(x)
p.map(f,x)
```

Remark: this example is academic: this is much slower than a normal loop/map (because of overhead). You need much more work in `f` to be more efficient with this.

Use multiprocessing for Mandelbrot

A little less academic. Remember the cython-enhanced calculation of the Mandelbrot-set.

```
def f(xi):
    zrow = zeros(y.shape[0])
    for j,yj in enumerate(y):
        zrow[j] = mandel.cython_mandel(xi,yj)
    return zrow
p = Pool(2) # start Pool of 2 processes
zdum = p.map(f,x) # parallel map
z = vstack(zdum) # create the final matrix
```

We are nearly as fast as the cython-loop in this case.

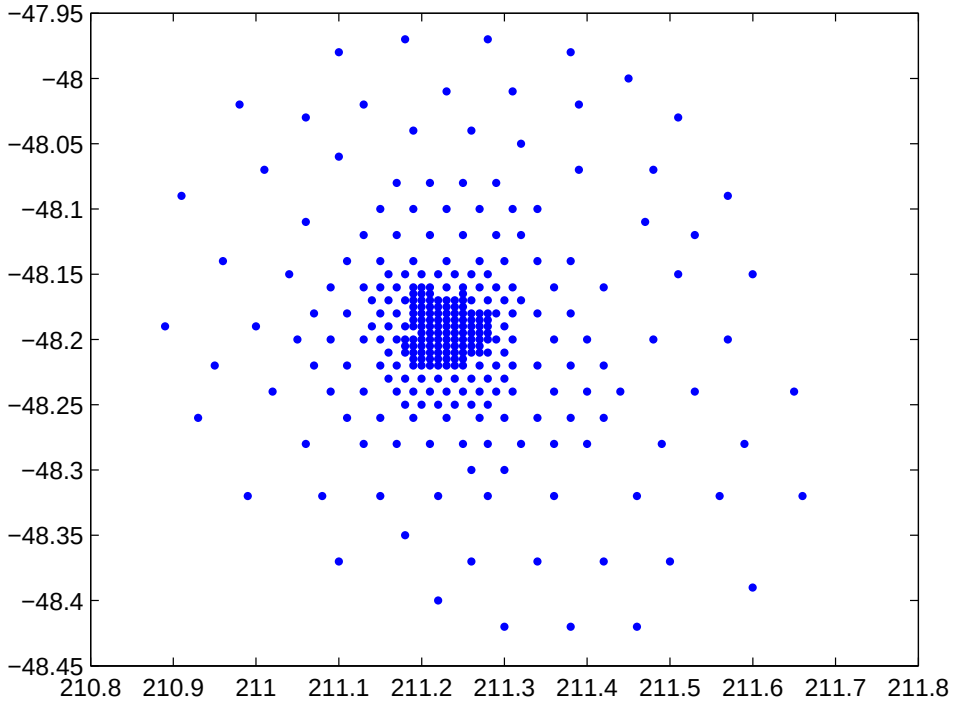
```
z = mandel.mandel_cy(pointsx,pointsy)
```

Aufbau

Literature

- NumPy, SciPy SciPy developers (http://scipy.org/),
- SciPy-lectures, F. Perez, E. Gouillart, G. Varoquaux, V. Haenel (http://scipy-lectures.github.io),
- Matplotlib (http://matplotlib.org)
- scitools (https://code.google.com/p/scitools/)
- enthought tools (http://docs.enthought.com/)
- mayavi (http://docs.enthought.com/mayavi/mayavi/mlab.html)
- Traits user manual (http://docs.enthought.com/traits/traits_user_manual/index.html)
- Cython documentation Cython developers (http://docs.cython.org/),

Interpolate scattered data

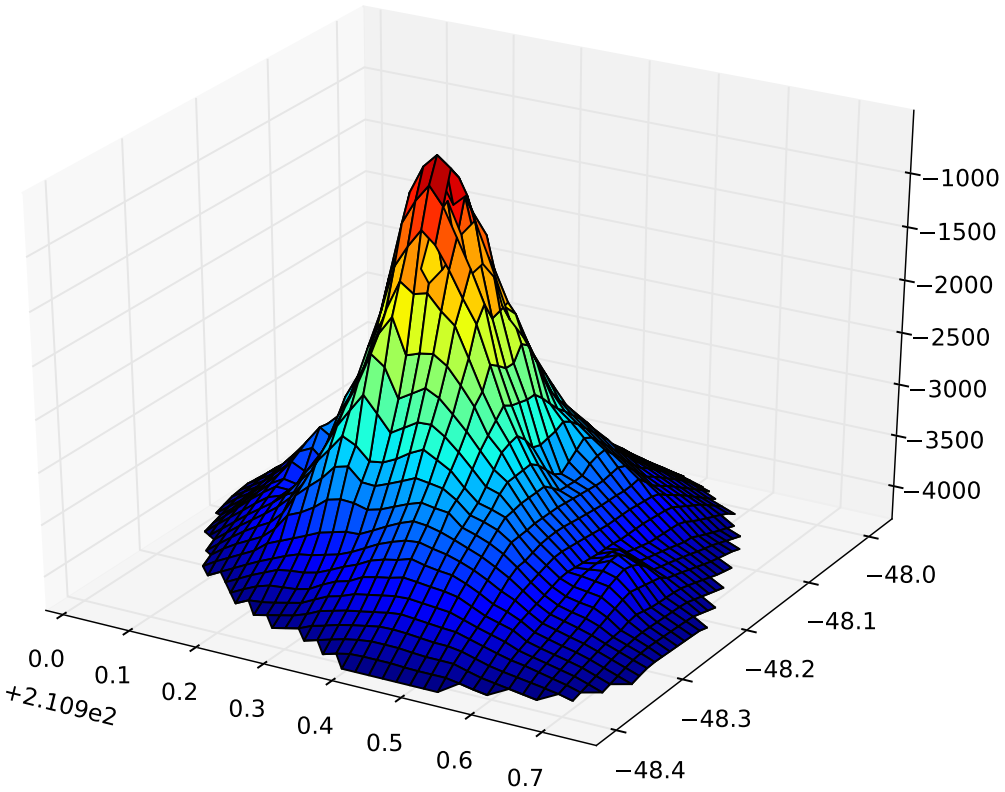



```
ZI = sp.interpolate.griddata ((<x>,<y>),<z>,(<XI>,<YI>),
    method='<method>')
```

- data: vectors x, y, z with $(x(i), y(i), z(i))$.
- matrices XI, YI for interpolated points $(XI(i,j), YI(i,j))$.
- method: `'nearest'`: piecewise constant; `'linear'`: linear
- interpolates in convex hull of the points $(x(i), y(i))$ otherwise `NaN`.

Interpolate scattered data

```
XI, YI = mgrid[min(x):max(x):40j,min(y):max(y):40j]
ZI = sp.griddata ((x,y),z,(XI,YI),method='linear')
fig = figure() , ax = Axes3D(fig)
ax.plot_surface(XI ,YI ,ZI,rstride=1,cstride=1)
```

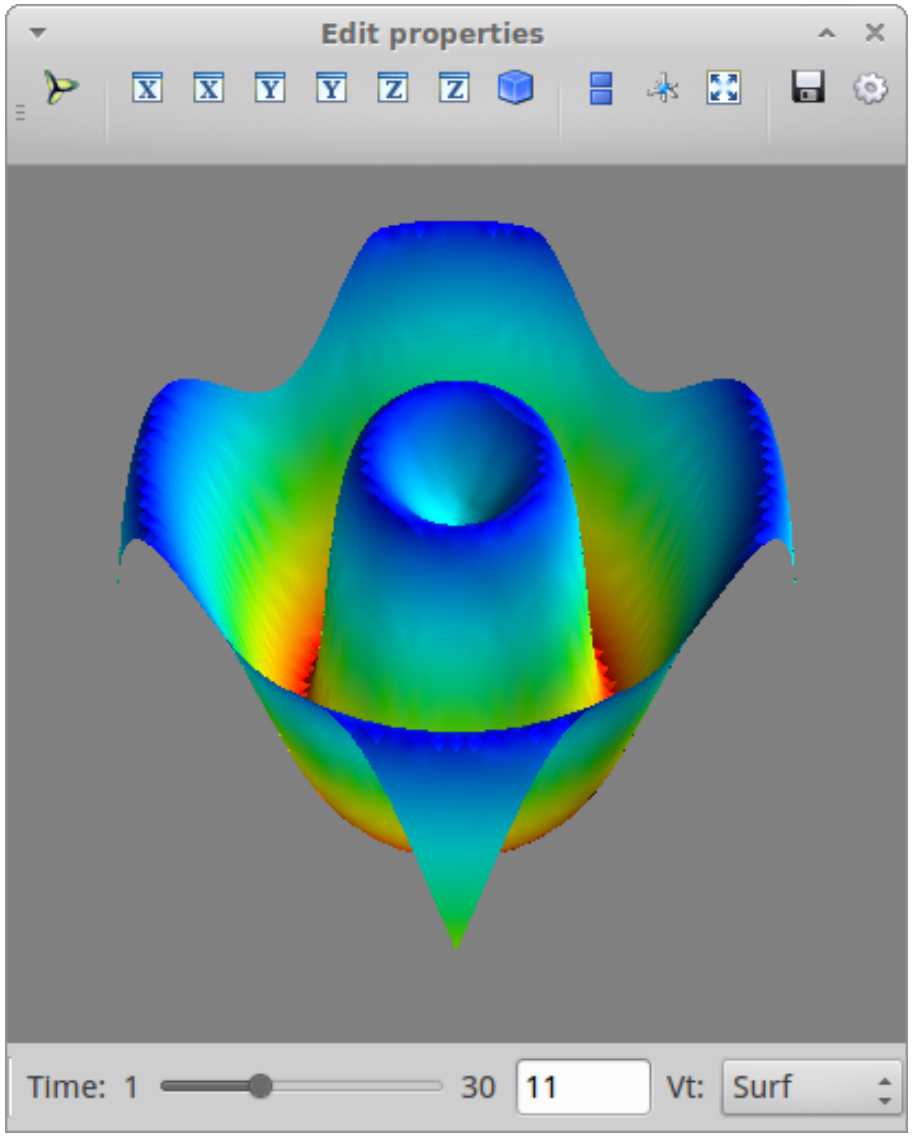


Traits

Extensions to normal python-objects with the following properties:

- initialization
- validation
- visualization
- notification
- documentation

lets do something like this:



Creation

Imports and classdefinition (cookbook)

```
from traits.api import HasTraits, Range, Enum
class Visualization(HasTraits):
```

create needed variables with traits:

```
        time = Range(1, 30, 1) # Range-slider
        vt = Enum ('Surf','Mesh','Contour') #Enumeration
```

Some Types

Trait	Python Type
Bool	Boolean
Float	Floating point number
Str	String
Array	Array of values
Enum	Enum of values
Range	Slider with values from/to some constants

(complete list: `http://docs.enthought.com/traits/traits_user_manual/defining.html#other-predefined-traits`)

Graphical representation - TraitsUI

use Traits UI and mayavi (cookbook)

```
from traitsui.api import View, Item, Group
from tvtk.pyface.scene_editor import SceneEditor
from mayavi.tools.mlab_scene_model import MlabSceneModel
from mayavi.core.ui.mayavi_scene import MayaviScene
```

create View; simple arrangement (item + group)

```
scene = Instance(MlabSceneModel, ()) # cookbook
view = View(Item('scene', editor=SceneEditor(scene_class=
    MayaviScene), height=250, width=300, show_label=False
    ), #first Item
            Group( #second Item
                  'time','vt'
                  orientation='horizontal',layout='normal
                     '),
            kind='live', title='simple GUI'
            )
```

Views

View configuration of a view of given traits. it contains

- Items
- Groups (groups of Items)

```
View (<itemORgroup>[, <itemORgroup>, <settings>])
```

settings of window

- height,width, title
- kind: type of window/dialog
  - ' live ' : normal window
  - 'wizard': is updated after pressing 'finish'
  - 'panel': embeds inside other windows

View: Items and Groups

an Item is a representation of a Trait:

```
Item (<traitname>[, <settings>])
```

settings of widget

- height, width, padding, tooltip , show_label

a Group is a visual or logical unit. it can hold Items and Groups.

```
Group(<item>[,<item>,<settings>])
```

settings of groups

- orientation
- layout: type of grouping
  - 'normal': one after another.
  - 'flow': as before but wraps around.
  - ' split ': split-bars between elements.
  - 'tabbed': tabbed elements.

Callbacks

initialization of data and classes.

```
x,y,t = np.mgrid[-1:1:(2.0/50),-1:1:(2.0/50),1:31]
Z = cos(pi*t**0.5*exp(-x**2-y**2))
def __init__(self):
    HasTraits.__init__(self)
    self.plot = self.scene.mlab.surf(self.x[:,:,0],
        self.y[:,:,0],self.Z[:,:,0])
```

Change camera viewpoint when first activated

```
@on_trait_change('scene.activated')
def create_plot(self):
    self.scene.mlab.view(45,210)
```

Decorator: @on_trait_change('variable' calls the given function, when the given variable has changed.

Callbacks II

functions for update the plot

```
    @on_trait_change('time,vt')
    def update_plot(self):
        self.plot.remove() # remove last image
        if self.vt == 'Surf':
            self.plot = self.scene.mlab.surf(self.x
                [:,:,0],self.y[:,:,0],self.Z[:,:,self.
                time-1])
        elif self.vt == 'Mesh':
            self.plot = self.scene.mlab.surf(self.x
                [:,:,0],self.y[:,:,0],self.Z[:,:,self.
                time-1],representation='wireframe')
        elif self.vt == 'Contour':
            self.plot = self.scene.mlab.contour_surf(self
                .x[:,:,0],self.y[:,:,0],self.Z[:,:,self.
                time-1],contours=15)
        else:
            print "error in choice of plot-type"
```