

Slides from INF3331 lectures - numerical Python

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2011



Numerical Python

Contents

- Efficient array computing in Python
- Creating arrays
- Indexing/slicing arrays
- Random numbers
- Linear algebra
- Plotting

More info

- Ch. 4 in the course book
- www.scipy.org
- The NumPy manual
- The SciPy tutorial

Numerical Python (NumPy)

- NumPy enables efficient numerical computing in Python
- NumPy is a package of modules, which offers efficient arrays (contiguous storage) with associated array operations coded in C or Fortran
- There are three implementations of Numerical Python
 - Numeric from the mid 90s (still widely used)
 - numarray from about 2000
 - numpy from 2006
- We recommend to use numpy (by Travis Oliphant)

```
from numpy import *
```

A taste of NumPy: a least-squares procedure

```
x = linspace(0.0, 1.0, n)           # coordinates
y_line = -2*x + 3
y = y_line + random.normal(0, 0.25, n) # line with noise

# goal: fit a line to the data points x, y

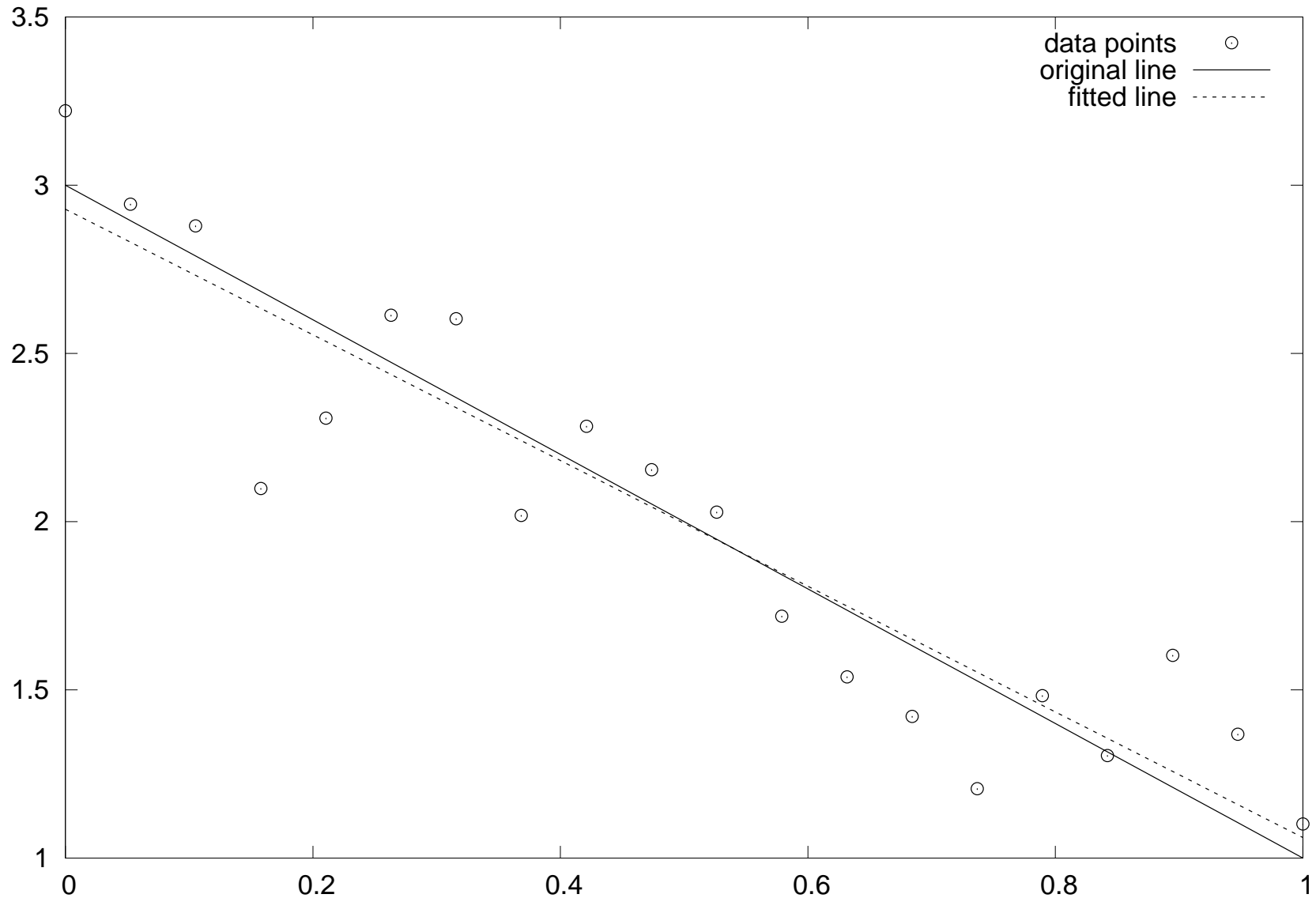
# create and solve least squares system:
A = array([x, ones(n)])
A = A.transpose()

result = linalg.lstsq(A, y)
# result is a 4-tuple, the solution (a,b) is the 1st entry:
a, b = result[0]

plot(x, y, 'o',           # data points w/noise
      x, y_line, 'r',     # original line
      x, a*x + b, 'b')   # fitted lines
legend('data points', 'original line', 'fitted line')
hardcopy('myplot.png')
```

Resulting plot

$y = -1.86794*x + 2.92875$: fit to $y = -2*x + 3.0 + \text{normal noise}$



Making arrays

```
>>> from numpy import *
>>> n = 4
>>> a = zeros(n)          # one-dim. array of length n
>>> print a
[ 0.  0.  0.  0.]
>>> a
array([ 0.,  0.,  0.,  0.])
>>> p = q = 2
>>> a = zeros((p,q,3))    # p*q*3 three-dim. array
>>> print a
[[[ 0.  0.  0.]
  [ 0.  0.  0.]]

 [[ 0.  0.  0.]
  [ 0.  0.  0.]]]
>>> a.shape              # a's dimension
(2, 2, 3)
```


Making float, int, complex arrays

```
>>> a = zeros(3)
>>> print a.dtype # a's data type
float64
>>> a = zeros(3, int)
>>> print a
[0 0 0]
>>> print a.dtype
int32
>>> a = zeros(3, float32) # single precision
>>> print a
[ 0.  0.  0.]
>>> print a.dtype
float32
>>> a = zeros(3, complex)
>>> a
array([ 0.+0.j,  0.+0.j,  0.+0.j])
>>> a.dtype
dtype('complex128')
```

>>> given an array a, make a new array of same dimension
>>> and data type:
>>> x = zeros(a.shape, a.dtype)

Array with a sequence of numbers

- `linspace(a, b, n)` generates n uniformly spaced coordinates, starting with a and ending with b

```
>>> x = linspace(-5, 5, 11)
>>> print x
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

- A special compact syntax is also available:

```
>>> a = r_[-5:5:11j] # same as linspace(-5, 5, 11)
>>> print a
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

- `arange` works like `range` (`xrange`)

```
>>> x = arange(-5, 5, 1, float)
>>> print x # upper limit 5 is not included!!
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.]
```

Warning: arange is dangerous

- arange's upper limit may or may not be included (due to round-off errors)
- Better to use a safer method: `seq(start, stop, increment)`

```
>>> from scitools.numpyutils import seq
>>> x = seq(-5, 5, 1)
>>> print x      # upper limit always included
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

- The package `scitools` is available at <http://code.google.com/p/scitools/>

Array construction from a Python list

- `array(list, [datatype])` generates an array from a list:

```
>>> p1 = [0, 1.2, 4, -9.1, 5, 8]
>>> a = array(p1)
```

- The array elements are of the simplest possible type:

```
>>> z = array([1, 2, 3])
>>> print z                                # array of integers
[1 2 3]
>>> z = array([1, 2, 3], float)
>>> print z
[ 1.  2.  3.]
```

- A two-dim. array from two one-dim. lists:

```
>>> x = [0, 0.5, 1]; y = [-6.1, -2, 1.2] # Python lists
>>> a = array([x, y]) # form array with x and y as rows
```

- From array to list: `alist = a.tolist()`

From “anything” to a NumPy array

- Given an object `a`,

```
a = asarray(a)
```

converts `a` to a NumPy array (if possible/necessary)

- Arrays can be ordered as in C (default) or Fortran:

```
a = asarray(a, order='Fortran')
isfortran(a) # returns True if a's order is Fortran
```

- Use `asarray` to, e.g., allow flexible arguments in functions:

```
def myfunc(some_sequence):
    a = asarray(some_sequence)
    return 3*a - 5
```

```
myfunc([1,2,3])           # list argument
myfunc((-1,1))           # tuple argument
myfunc(zeros(10))        # array argument
myfunc(-4.5)             # float argument
myfunc(6)                # int argument
```

Changing array dimensions

```
>>> a = array([0, 1.2, 4, -9.1, 5, 8])
>>> a.shape = (2,3)          # turn a into a 2x3 matrix
>>> print a
[[ 0.    1.2   4. ]
 [-9.1   5.    8. ]]
>>> a.size
6
>>> a.shape = (a.size,)     # turn a into a vector of length 6 again
>>> a.shape
(6,)
>>> print a
[ 0.    1.2   4.  -9.1   5.    8. ]
>>> a = a.reshape(2,3)     # same effect as setting a.shape
>>> a.shape
(2, 3)
```

Array initialization from a Python function

```
>>> def myfunc(i, j):  
...     return (i+1)*(j+4-i)  
...  
>>> # make 3x6 array where a[i,j] = myfunc(i,j):  
>>> a = fromfunction(myfunc, (3,6))  
>>> a  
array([[ 4.,  5.,  6.,  7.,  8.,  9.],  
       [ 6.,  8., 10., 12., 14., 16.],  
       [ 6.,  9., 12., 15., 18., 21.]])
```

Basic array indexing

Note: all integer indices in Python start at 0!

```
a = linspace(-1, 1, 6)
a[2:4] = -1      # set a[2] and a[3] equal to -1
a[-1] = a[0]    # set last element equal to first one
a[:] = 0        # set all elements of a equal to 0
a.fill(0)       # set all elements of a equal to 0

a.shape = (2,3) # turn a into a 2x3 matrix
print a[0,1]   # print element (0,1)
a[i,j] = 10    # assignment to element (i,j)
a[i][j] = 10   # equivalent syntax (slower)
print a[:,k]   # print column with index k
print a[1,:]   # print second row
a[:,:] = 0     # set all elements of a equal to 0
```


More advanced array indexing

```
>>> a = linspace(0, 29, 30)
>>> a.shape = (5,6)
>>> a
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.],
       [24., 25., 26., 27., 28., 29.]])

>>> a[1:3,::2]    # a[i,j] for i=1,2 and j=0,2,4
array([[ 6.,  8., 10.],
       [12., 14., 16.]])

>>> a[:,2::2]    # a[i,j] for i=0,3 and j=2,4
array([[ 2.,  4.],
       [20., 22.]])

>>> i = slice(None, None, 3); j = slice(2, None, 2)
>>> a[i,j]
array([[ 2.,  4.],
       [20., 22.]])
```

Slices refer the array data

- With a as list, `a[:]` makes a copy of the data
- With a as array, `a[:]` is a reference to the data

```
>>> b = a[2,:]          # extract 2nd row of a
>>> print a[2,0]
12.0
>>> b[0] = 2
>>> print a[2,0]
2.0                      # change in b is reflected in a!
```

- Take a copy to avoid referencing via slices:

```
>>> b = a[2,:].copy()
>>> print a[2,0]
12.0
>>> b[0] = 2          # b and a are two different arrays now
>>> print a[2,0]
12.0                  # a is not affected by change in b
```

Loops over arrays (1)

- Standard loop over each element:

```
for i in xrange(a.shape[0]):
    for j in xrange(a.shape[1]):
        a[i,j] = (i+1)*(j+1)*(j+2)
        print 'a[%d,%d]=%g ' % (i,j,a[i,j]),
    print # newline after each row
```

- A standard for loop iterates over the first index:

```
>>> print a
[[ 2.   6.  12.]
 [ 4.  12.  24.]]
>>> for e in a:
...     print e
...
[ 2.   6.  12.]
[ 4.  12.  24.]
```

Loops over arrays (2)

- View array as one-dimensional and iterate over all elements:

```
for e in a.ravel():  
    print e
```

Use `ravel()` only when reading elements, for assigning it is better to use `shape` or `reshape` first!

- For loop over all index tuples and values:

```
>>> for index, value in ndenumerate(a):  
...     print index, value  
...  
(0, 0) 2.0  
(0, 1) 6.0  
(0, 2) 12.0  
(1, 0) 4.0  
(1, 1) 12.0  
(1, 2) 24.0
```

Array computations

- Arithmetic operations can be used with arrays:

```
b = 3*a - 1    # a is array, b becomes array
```

1) compute $t1 = 3*a$, 2) compute $t2 = t1 - 1$, 3) set $b = t2$

- Array operations are much faster than element-wise operations:

```
>>> import time # module for measuring CPU time
>>> a = linspace(0, 1, 1E+07) # create some array
>>> t0 = time.clock()
>>> b = 3*a - 1
>>> t1 = time.clock() # t1-t0 is the CPU time of 3*a-1

>>> for i in xrange(a.size): b[i] = 3*a[i] - 1
>>> t2 = time.clock()
>>> print '3*a-1: %g sec, loop: %g sec' % (t1-t0, t2-t1)
3*a-1: 2.09 sec, loop: 31.27 sec
```

Standard math functions can take array arguments

```
# let b be an array

c = sin(b)
c = arcsin(c)
c = sinh(b)
# same functions for the cos and tan families

c = b**2.5 # power function
c = log(b)
c = exp(b)
c = sqrt(b)
```

Other useful array operations

```
# a is an array

a.clip(min=3, max=12) # clip elements
a.mean(); mean(a)    # mean value
a.var(); var(a)      # variance
a.std(); std(a)      # standard deviation
median(a)
cov(x,y)             # covariance
trapz(a)             # Trapezoidal integration
diff(a)              # finite differences (da/dx)

# more Matlab-like functions:
corrcoeff, cumprod, diag, eig, eye, fliplr, flipud, max, min,
prod, ptp, rot90, squeeze, sum, svd, tri, tril, triu
```

More useful array methods and attributes

```
>>> a = zeros(4) + 3
>>> a
array([ 3.,  3.,  3.,  3.]) # float data
>>> a.item(2) # more efficient than a[2]
3.0
>>> a.itemset(3,-4.5) # more efficient than a[3]=-4.5
>>> a
array([ 3. ,  3. ,  3. , -4.5])
>>> a.shape = (2,2)
>>> a
array([[ 3. ,  3. ],
       [ 3. , -4.5]])
>>> a.ravel() # from multi-dim to one-dim
array([ 3. ,  3. ,  3. , -4.5])
>>> a.ndim # no of dimensions
2
>>> len(a.shape) # no of dimensions
2
>>> rank(a) # no of dimensions
2
>>> a.size # total no of elements
4
>>> b = a.astype(int) # change data type
>>> b
array([3, 3, 3, 3])
```


Modules for curve plotting and 2D/3D visualization

- Matplotlib (curve plotting, 2D scalar and vector fields)
- PyX (PostScript/TeX-like drawing)
- Interface to Gnuplot
- Interface to Vtk
- Interface to OpenDX
- Interface to IDL
- Interface to Grace
- Interface to Matlab
- Interface to R
- Interface to Blender

Curve plotting with Easyviz

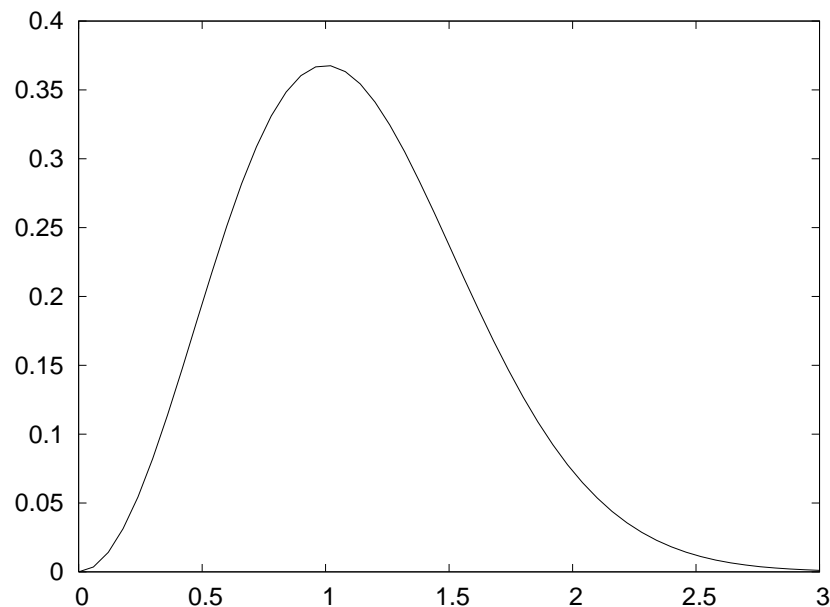
- Easyviz is a light-weight interface to many plotting packages, using a Matlab-like syntax
- Goal: write your program using Easyviz (“Matlab”) syntax and postpone your choice of plotting package
- Note: some powerful plotting packages (Vtk, R, matplotlib, ...) may be troublesome to install, while Gnuplot is easily installed on all platforms
- Easyviz supports (only) the most common plotting commands
- Easyviz is part of SciTools (Simula development)

```
from scitools.all import *
```

(imports all of numpy, all of easyviz, plus scitools)

Basic Easyviz example

```
from scitools.all import * # import numpy and plotting
t = linspace(0, 3, 51)     # 51 points between 0 and 3
y = t**2*exp(-t**2)       # vectorized expression
plot(t, y)
hardcopy('tmp1.eps')     # make PostScript image for reports
hardcopy('tmp1.png')    # make PNG image for web pages
```



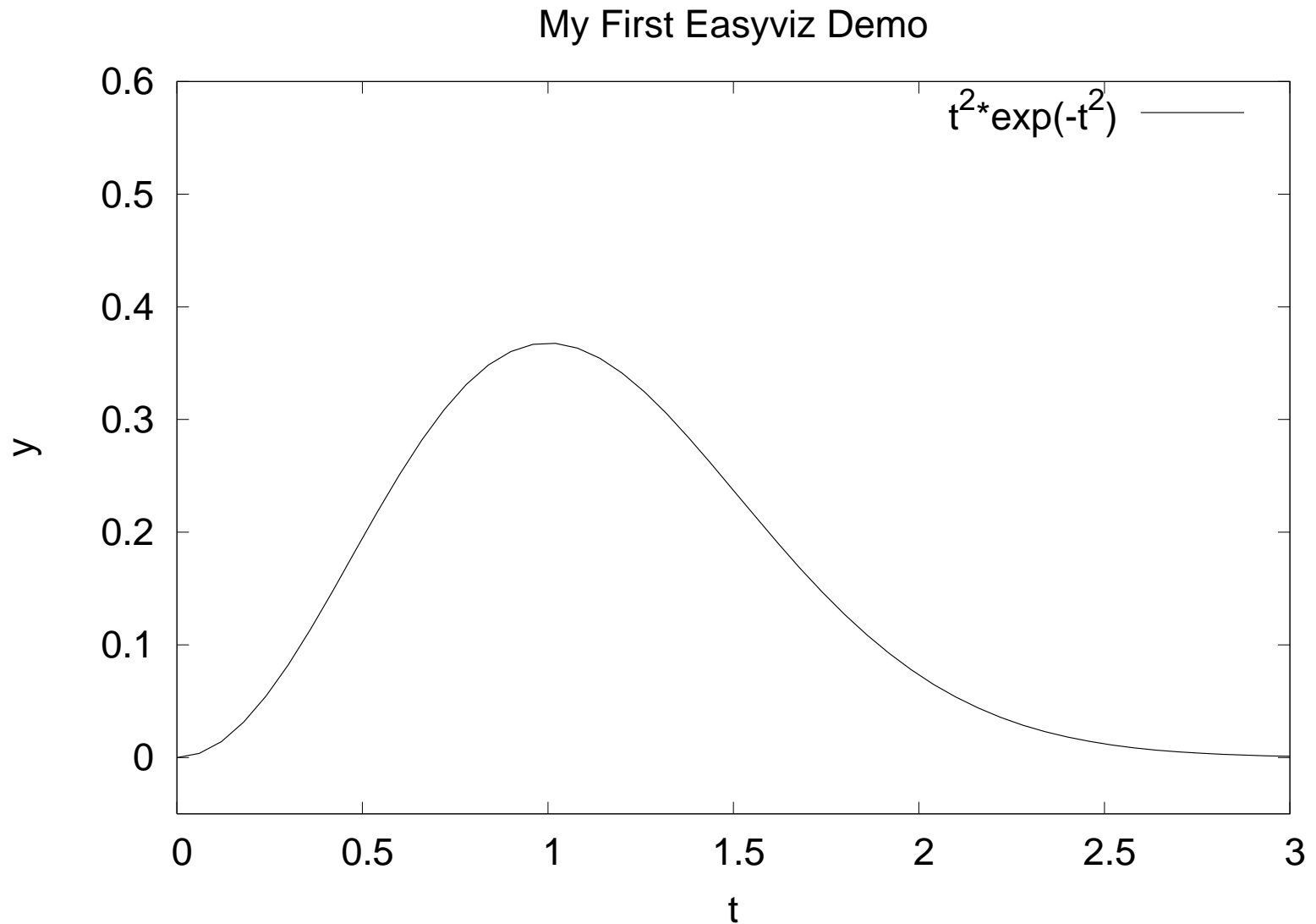
Decorating the plot

```
plot(t, y)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)')
axis([0, 3, -0.05, 0.6]) # [tmin, tmax, ymin, ymax]
title('My First Easyviz Demo')

# or
plot(t, y, xlabel='t', ylabel='y',
      legend='t^2*exp(-t^2)',
      axis=[0, 3, -0.05, 0.6],
      title='My First Easyviz Demo',
      hardcopy='tmp1.eps',
      show=True) # display on the screen (default)
```

The resulting plot



Plotting several curves in one plot

Compare $f_1(t) = t^2 e^{-t^2}$ and $f_2(t) = t^4 e^{-t^2}$ for $t \in [0, 3]$

```
from scitools.all import *      # for curve plotting

def f1(t):
    return t**2*exp(-t**2)

def f2(t):
    return t**2*f1(t)

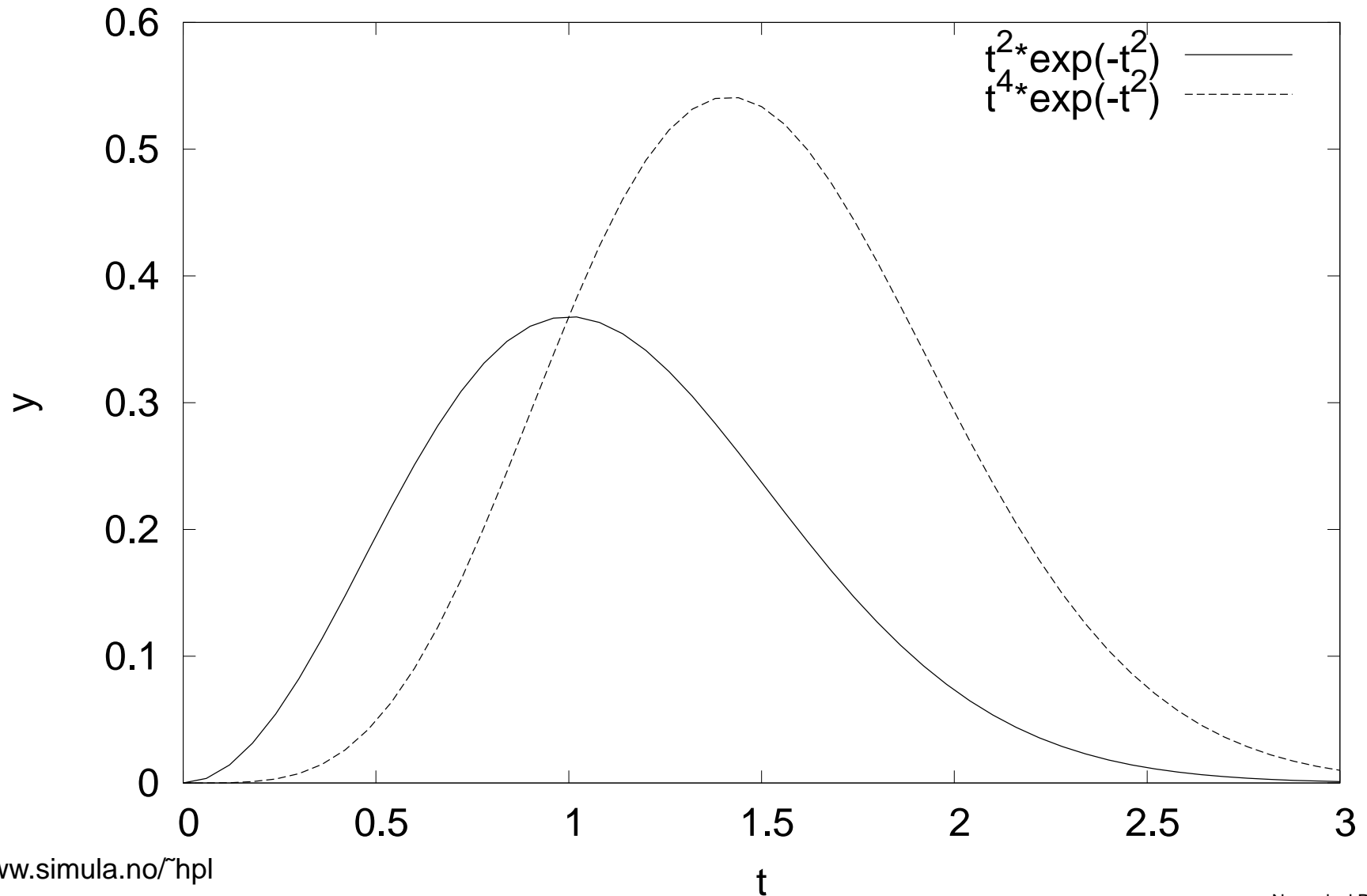
t = linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plot(t, y1)
hold('on')      # continue plotting in the same plot
plot(t, y2)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
hardcopy('tmp2.eps')
```

The resulting plot

Plotting two curves in the same plot



Example: plot a function given on the command line

- Task: plot (e.g.) $f(x) = e^{-0.2x} \sin(2\pi x)$ for $x \in [0, 4\pi]$

- Specify $f(x)$ and x interval as text on the command line:

```
Unix/DOS> python plotf.py "exp(-0.2*x)*sin(2*pi*x)" 0 4*pi
```

- Program:

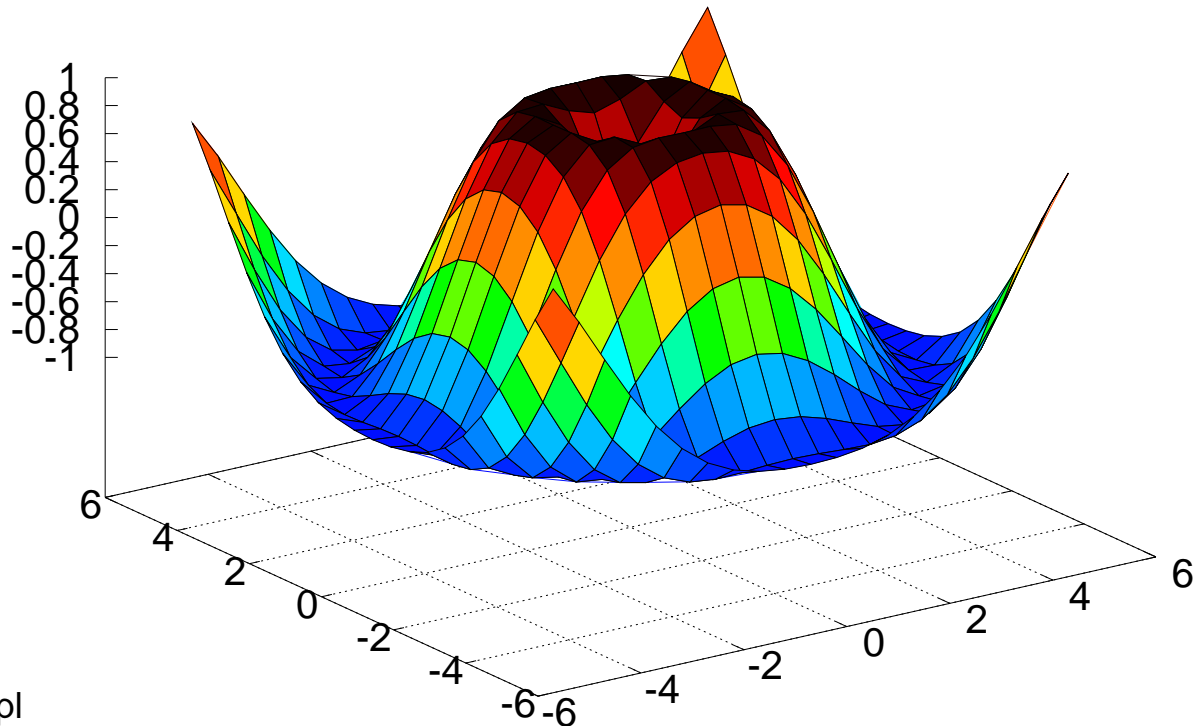
```
from scitools.all import *
formula = sys.argv[1]
xmin = eval(sys.argv[2])
xmax = eval(sys.argv[3])

x = linspace(xmin, xmax, 101)
y = eval(formula)
plot(x, y, title=formula)
```

- Thanks to `eval`, input (text) with correct Python syntax can be turned to running code on the fly

Plotting 2D scalar fields

```
from scitools.all import *  
  
x = y = linspace(-5, 5, 21)  
xv, yv = ndgrid(x, y)  
values = sin(sqrt(xv**2 + yv**2))  
surf(xv, yv, values)
```

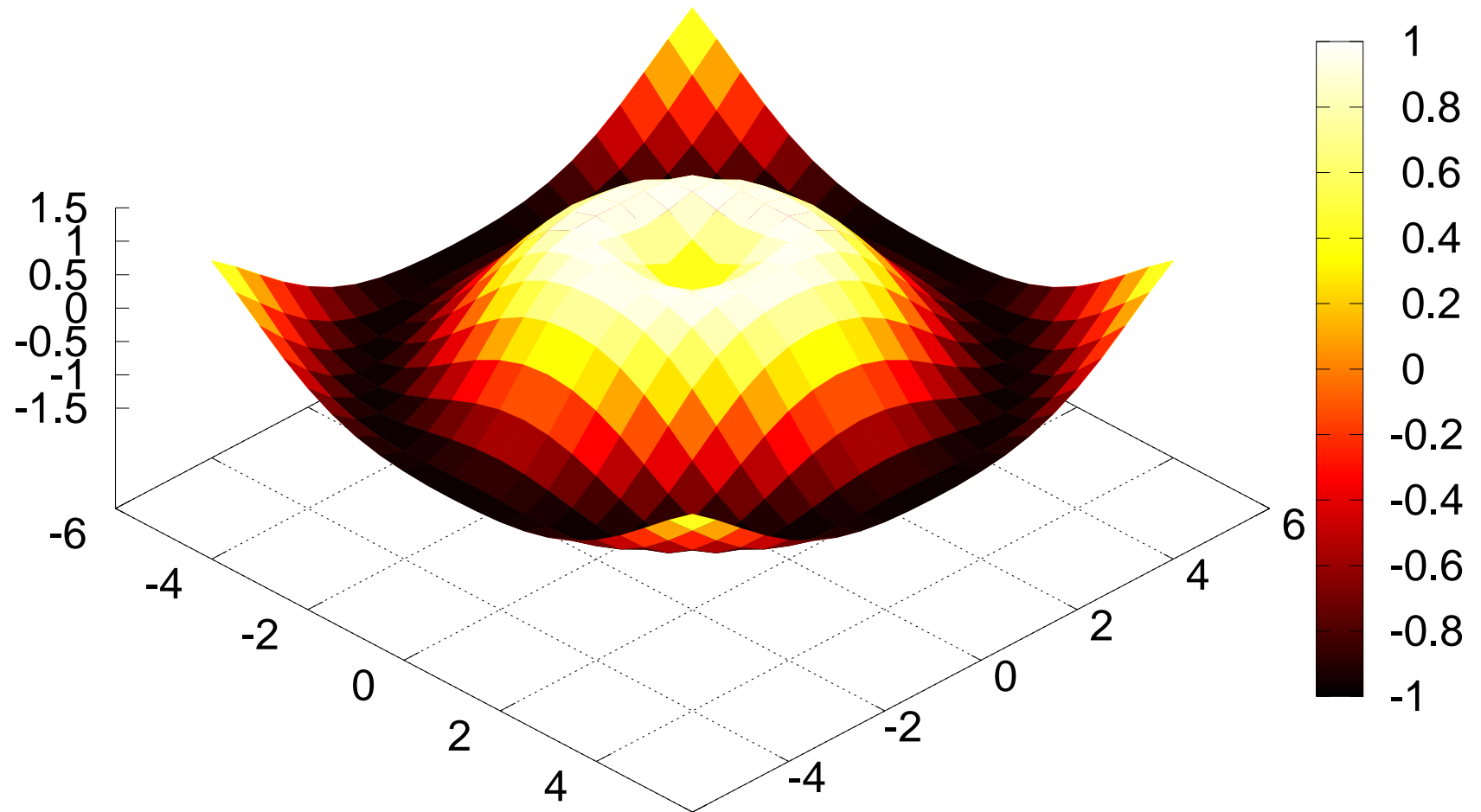


Adding plot features

```
# Matlab style commands:  
setp(interactive=False)  
surf(xv, yv, values)  
shading('flat')  
colorbar()  
colormap(hot())  
axis([-6,6,-6,6,-1.5,1.5])  
view(35,45)  
show()
```

```
# Optional Easyviz (Pythonic) short cut:  
surf(xv, yv, values,  
      shading='flat',  
      colorbar='on',  
      colormap=hot(),  
      axis=[-6,6,-6,6,-1.5,1.5],  
      view=[35,45])
```

The resulting plot



Other commands for visualizing 2D scalar fields

- `contour` (standard contours), `contourf` (filled contours), `contour3` (elevated contours)
- `mesh` (elevated mesh), `meshc` (elevated mesh with contours in the xy plane)
- `surf` (colored surface), `surfc` (colored surface with contours in the xy plane)
- `pcolor` (colored cells in a 2D mesh)

Commands for visualizing 3D fields

Scalar fields:

- `isosurface`
- `slice_` (colors in slice plane),
`contourslice` (contours in slice plane)

Vector fields:

- `quiver3` (arrows), (`quiver` for 2D vector fields)
- `streamline`, `streamtube`, `streamribbon` (flow sheets)

More info about Easyviz

- A plain text version of the Easyviz manual:

```
pydoc scitools.easyviz
```

- The HTML version:

```
http://code.google.com/p/scitools/wiki/EasyvizDocumentation
```

- Download SciTools (incl. Easyviz):

```
http://code.google.com/p/scitools/
```