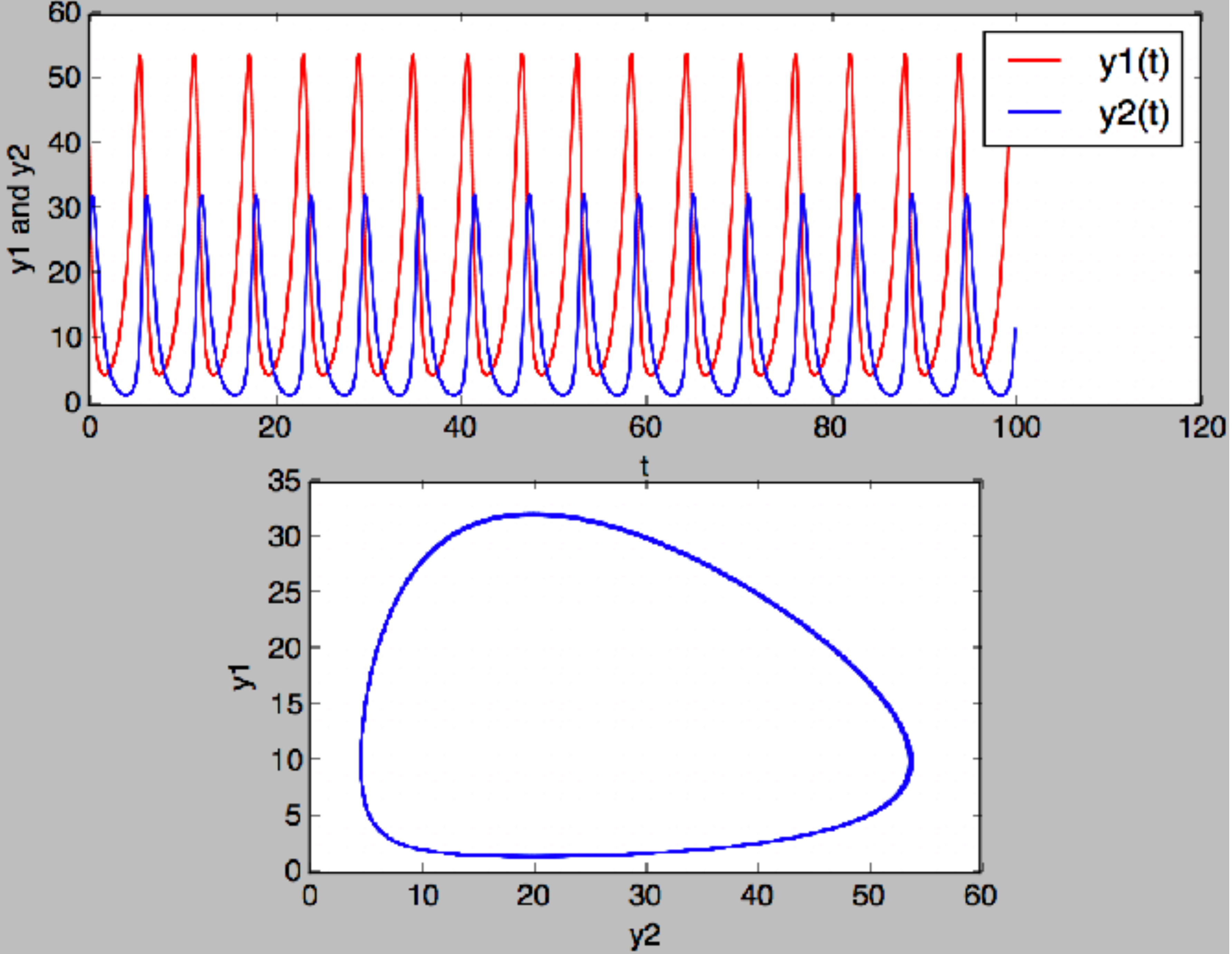
Scipy

SciPy.org Sponsored By ENTHOUGHT

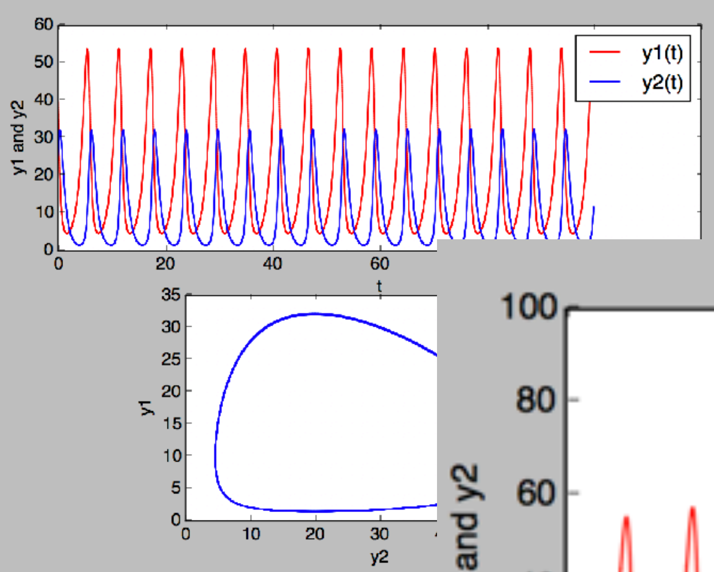
```python
def f(y,t):
    """du/dt =  a*u -   b*u*v"""
    return(a*y[0] - b*y[0]*y[1])

def g(y,t):
    """dy/dt = -c*v + d*b*u*v"""
    return(-c*y[1] + d*b*y[0]*y[1])


if __name__ == "__main__":

    a = 1.0
    b = 0.1
    c = 1.5
    d = 0.75
    rabbits0 = 40
    foxes0 = 25

    interval = [0.,100.]
    initial0 = [rabbits0,foxes0]
    maxiter  = 1000000
    rhs_list = [f,g]
    dt       = 0.001

    myode = ode.ODE(interval,initial0,maxiter,rhs_list,dt)
    solution = myode.solve()
    print solution[-4:]
    myode.plot()
    plt.show()
```

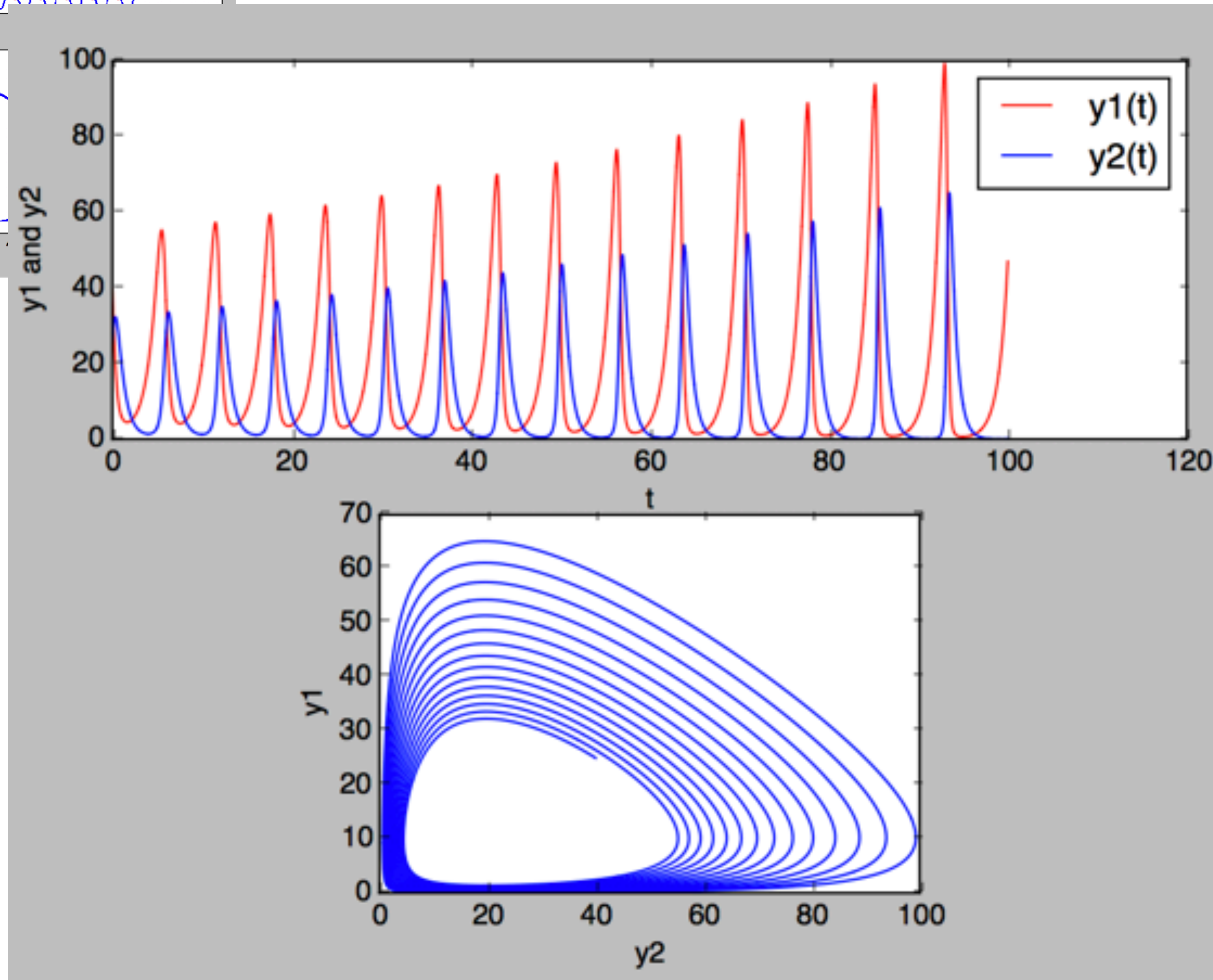
```python
class ODE(object):
    """
    Input:
    interval: [a,b] where a and b are bounds
    initial0: initial conditions for each function/y
    maxiter:  maximum iterations
    rhs_list: list of functions will take two arguments:
              an array y and a time t
    dt:       this is the interval length [default:0.01]
    """
    def __init__(self, interval,initial0,maxiter,rhs_list,dt=0.01):
        self.a,self.b = interval
        self.dt = dt
        self.t = self.a
        self.y = [[self.t,initial0]]
        self.maxiter = maxiter
        self.rhs = np.array(rhs_list)
        self.dim = len(initial0)
        self.count = 0
        if(self.dim != len(self.rhs)):
            print "right hand side and initial values"
            print "have inconsistent length",self.dim,len(self.rhs)
            raise Exception('Inconsistent Arguments')

    def solve(self):
        """ advance solution until maxiter iterations or boundary b is reached"""
        iter = 0
        while self.t < self.b and iter < self.maxiter:
            lasty = self.y[-1][1]
            rhs = np.array([func(lasty, self.t) for func in self.rhs])
            ynew = lasty + rhs * self.dt
            iter += 1
            self.t += self.dt
            self.y.append([self.t,ynew])
        self.count += 1
        return self.y
```



y1(t)
y2(t)
y1 and y2
t
y1
y2

Bad steppingsize **h**: $y_n = y_{n-1} + \mathbf{h}\, f(y,t)$





SciPy

- Is a collection of mathematical algorithms and convenience functions built on the Numpy extension of Python.

- It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data.

- Is the core package for scientific routines in Python

- Operate efficiently on numpy arrays, so that numpy and scipy work hand in hand.

- Contains various toolboxes dedicated to common issues in scientific computing such as:

interpolation, integration, optimization, image processing, statistics, special functions, etc.

The additional benefit of basing SciPy :

- Making a powerful programming language available for use in developing sophisticated programs and specialized applications.

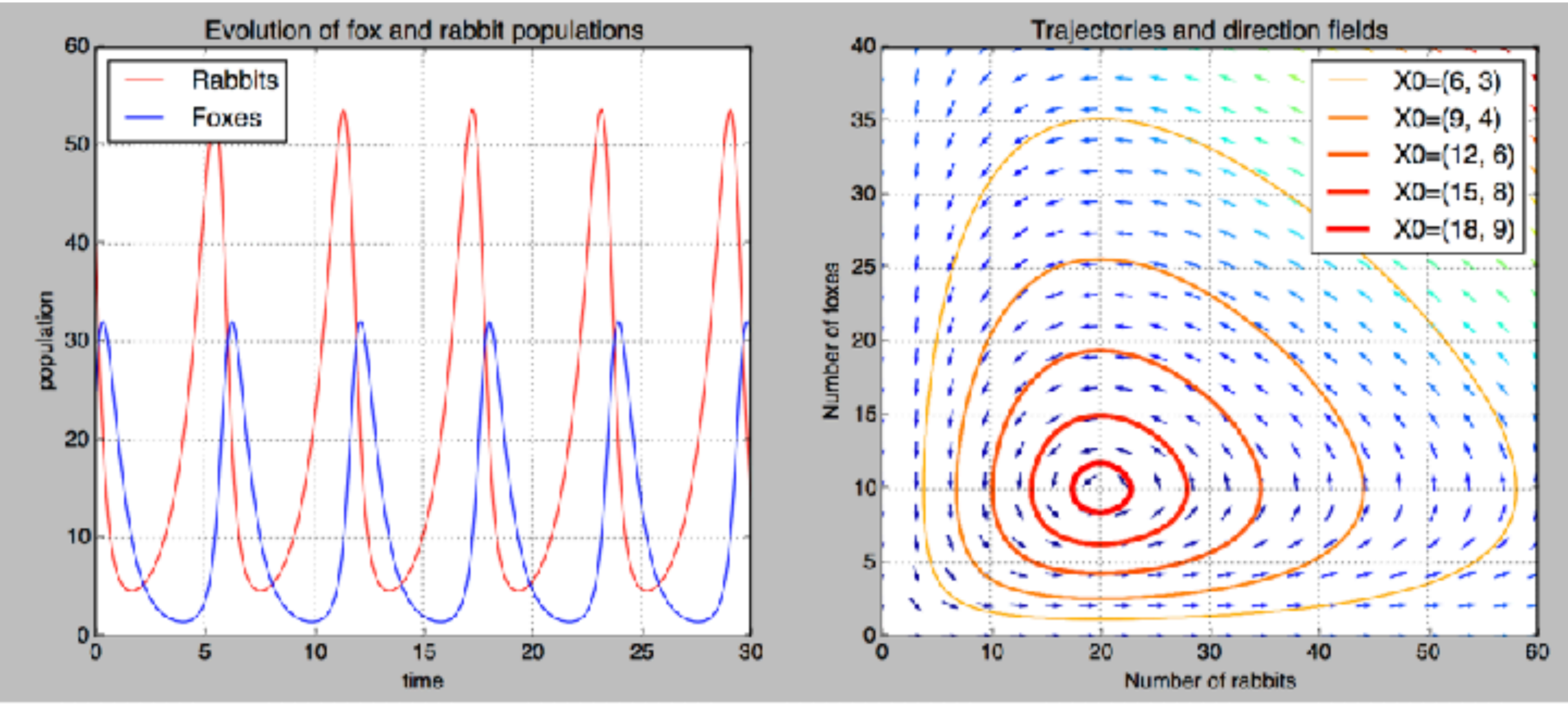
- Everything from parallel programming to web and data-base subroutines and classes have been made available to the Python programmer.

- scipy.cluster : Vector quantization / Kmeans
- scipy.constants : Physical and mathematical constants
- scipy.fftpack: Fourier transform
- scipy.integrate: Integration routines
- scipy.interpolate: Interpolation
- scipy.io: Data input and output
- scipy.linalg: Linear algebra routines
- scipy.ndimage: n-dimensional image package
- scipy.odr: Orthogonal distance regression
- scipy.optimize: Optimization
- scipy.signal: Signal processing
- scipy.sparse: Sparse matrices
- scipy.spatial: Spatial data structures and algorithms
- scipy.special: Any special mathematical functions
- scipy.stats: Statistics

Solving ODEs using scipy

```
a = 1.0
b = 0.1
c = 1.5
d = 0.75
rabbits0 = 40
foxes0=25

def myodes(X, t=0):
    """ Return the growth rate of fox and rabbit populations. "
    return np.array([ a*X[0] -   b*X[0]*X[1] ,
                  -c*X[1] + d*b*X[0]*X[1] ])

t = np.linspace(0, 30,  1000)    # time
X0 = np.array([rabbits0,foxes0]) # initials conditions: 10
rabbits and 5 foxes
X,infodict = integrate.odeint(myodes, X0, t, full_output=True)
infodict['message']              #>>>'Integration successful.'


rabbits, foxes = X.T
```

Solving ODEs using scipy



File input/output: scipy.io

matlab files:

sio.loadmat
sio.savemat
sio.whosmat

```
from scipy import io as spio
a = np.ones((3, 3))
spio.savemat('file.mat', {'a': a}) # savemat expects a dictionary
print data['a']
```

```
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
```

Savemat: Save a dictionary of names and arrays into a MATLAB-style .mat file.

- Reading Image:

```
from scipy import misc
misc.imread('fname.png')
# Matplotlib also has a similar function
import matplotlib.pyplot as plt
plt.imread('fname.png')
```

Linear Algebra: scipy.linalg

- The scipy.linalg.det() function computes the determinant of a square matrix

```
from scipy import linalg
arr = np.array([[1, 2],[3, 4]])
d=linalg.det(arr)
print d
arr = np.array([[3, 2],[6, 4]])
d=linalg.det(arr)
print d
```

```
-2.0
0.0
```

```
linalg.det(np.ones((3, 4)))
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-15-4d4672bd00a7> in <module>()
----> 1 linalg.det(np.ones((3, 4)))

/Library/Python/2.7/site-packages/scipy-0.10.1-py2.7-macosx-10.7-x86_64.egg/scipy/linalg/basic.pyc in det(a, overwrite_a)
    352     a1 = np.asarray_chkfinite(a)
    353     if len(a1.shape) != 2 or a1.shape[0] != a1.shape[1]:
--> 354         raise ValueError('expected square matrix')
    355     overwrite_a = overwrite_a or _datacopied(a1, a)
    356     fdet, = get_flinalg_funcs(('det',), (a1,))

ValueError: expected square matrix
```

The scipy.linalg.inv() function computes the inverse of a square matrix:

```
arr = np.array([[1, 2],[3, 4]])
iarr = linalg.inv(arr)
print iarr
```

```
[[-2.   1. ]
 [ 1.5 -0.5]]
```

```
arr = np.array([[3, 2],[6, 4]])
linalg.inv(arr)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-18-b961f864356e> in <module>()
      1 arr = np.array([[3, 2],[6, 4]])
----> 2 linalg.inv(arr)

/Library/Python/2.7/site-packages/scipy-0.10.1-py2.7-macosx-10.7-x86_64.egg/scipy/linalg/basic.pyc in inv(a, overwrite_a)
    325             inv_a, info = getri(lu, piv, overwrite_lu=1)
    326     if info > 0:
--> 327         raise LinAlgError("singular matrix")
    328     if info < 0:
    329         raise ValueError('illegal value in %d-th argument of internal '

LinAlgError: singular matrix
```

SVD: Singular Value Decomposition

```
arr = np.arange(9).reshape((3, 3))
print arr
print "------"
U, S, V = linalg.svd(arr)
print U
print "------"
print S
print "------"
print V
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
------
[[-0.13511895  0.90281571  0.40824829]
 [-0.49633514  0.29493179 -0.81649658]
 [-0.85755134 -0.31295213  0.40824829]]
------
[  1.42267074e+01   1.26522599e+00   5.89938022e-16]
------
[[-0.4663281  -0.57099079 -0.67565348]
 [-0.78477477 -0.08545673  0.61386131]
 [-0.40824829  0.81649658 -0.40824829]]
```

- The original matrix can be re-composed by matrix multiplication of the outputs of svd with np.dot:

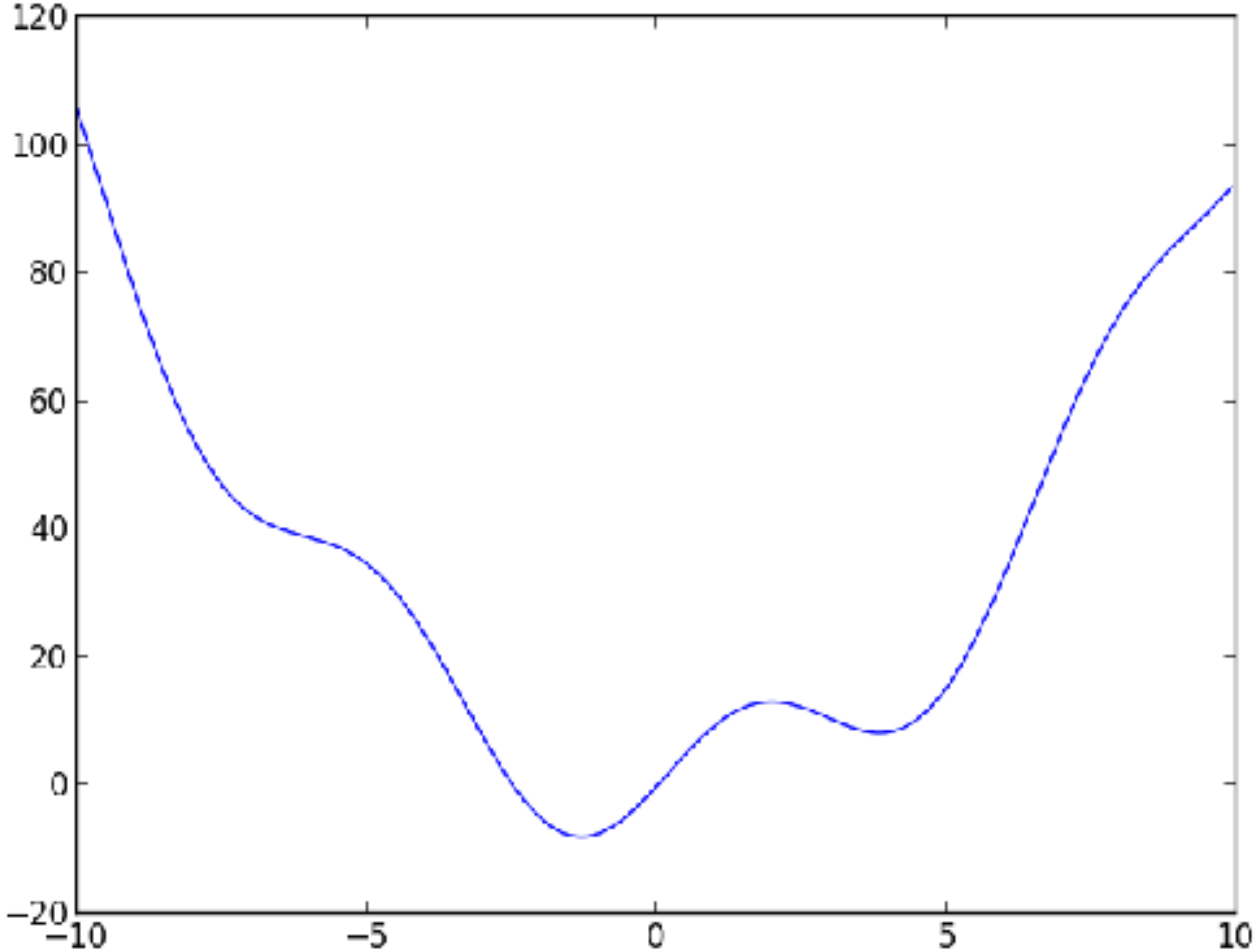
```
sarr = np.diag(S)
svd_mat = U.dot(sarr).dot(V)
print svd_mat
```

```
[[ 1.   1.   2.]
 [ 3.   4.   5.]
 [ 6.   7.   9.]]
```

Optimization and fit: scipy.optimize

Optimization is the problem of finding a numerical solution to a minimization or equality. The scipy.optimize module provides useful algorithms for function minimization (scalar or multidimensional), curve fitting and root finding.

```
import pylab as plt
from scipy import optimize

def f(x):
    return x**2 + 10*np.sin(x)

x = np.arange(-10, 10, 0.1)
plt.plot(x, f(x))
plt.show()
```



This function has a global minimum around -1.3 and a local minimum around 3.8

- The general and efficient way to find a minimum for this function is to conduct a gradient descent starting from a given initial point. The BFGS algorithm is a good way of doing this:

From an initial guess $\mathbf{x}_0$ and an approximate Hessian matrix B_0 the following steps are repeated as $\mathbf{x}_k$ converges to the solution.

1. Obtain a direction $\mathbf{p}_k$ by solving: $B_k \mathbf{p}_k = -\nabla f(\mathbf{x}_k).$
2. Perform a line search to find an acceptable stepsize α_k in the direction found in the first step, then update $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k.$
3. Set $\mathbf{s}_k = \alpha_k \mathbf{p}_k.$
4. $\mathbf{y}_k = \nabla f(\mathbf{x}_{k+1}) - \nabla f(\mathbf{x}_k).$
5. $$B_{k+1} = B_k + \frac{\mathbf{y}_k \mathbf{y}_k^{\mathrm{T}}}{\mathbf{y}_k^{\mathrm{T}} \mathbf{s}_k} - \frac{B_k \mathbf{s}_k \mathbf{s}_k^{\mathrm{T}} B_k}{\mathbf{s}_k^{\mathrm{T}} B_k \mathbf{s}_k}.$$

```
optimize.fmin_bfgs(f, 0)
```

```
Optimization terminated successfully.
         Current function value: -7.945823
         Iterations: 5
         Function evaluations: 24
         Gradient evaluations: 8

array([-1.30644003])
```

A possible issue with this approach is that, if the function has local minima the algorithm may find these local minima instead of the global minimum depending on the initial point:

```
optimize.fmin_bfgs(f, 3, disp=0)
```

```
array([ 3.83746663])
```

If we don't know the neighborhood of the global minimum to choose the initial point, we need to resort to costlier global optimization. To find the global minimum, the simplest algorithm is the brute force algorithm, in which the function is evaluated on each point of a given grid:

```
grid = (-10, 10, 0.1)
xmin_global = optimize.brute(f, (grid,))
print xmin_global
```

```
[-1.30641113]
```

Brute Force algorithm becomes quite slow for larger grid sizes. Simulated annealing can be a good alternative:

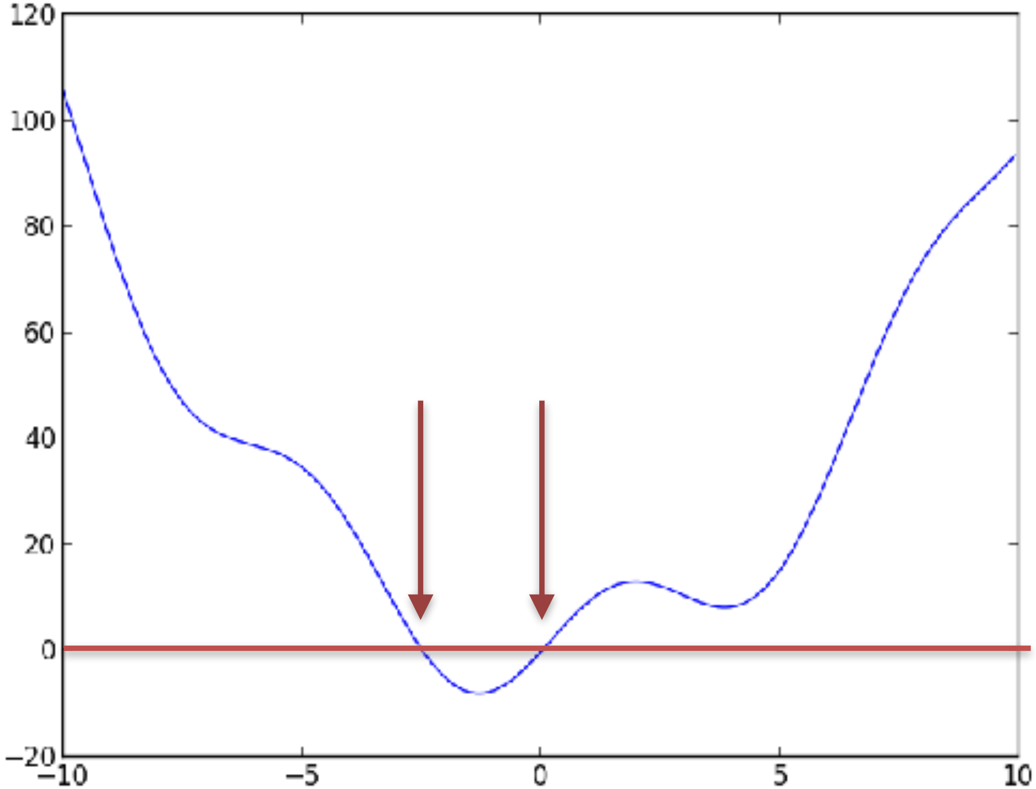
Scipy.optimize.anneal()

For local minimum, we can constraint the variable to the interval and use:

```
xmin_local = optimize.fminbound(f, 0, 10)
```

Root finding

To find a root, a point where f(x) = 0



scipy.optimize.fsolve()

```
root = optimize.fsolve(f, 1) # our initial guess is 1
print root
```

```
[ 0.]
```

Only one root is found. But there is a second root around -2.5. We find
the exact value of it by adjusting our initial guess:

```
root2 = optimize.fsolve(f, -2.5)
print root2
```

```
[-2.47948183]
```

Other modules:

Scalar functions

brentq(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in given interval.
brenth(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of f in [a,b].
ridder(f, a, b[, args, xtol, rtol, maxiter, ...])	Find a root of a function in an interval.
bisect(f, a, b[, args, xtol, rtol, maxiter, ...])	Find root of a function within an interval.
newton(func, x0[, fprime, args, tol, ...])	Find a zero using the Newton-Raphson or secant method.

Fixed point finding:

fixed_point(func, x0[, args, xtol, maxiter])	Find a fixed point of the function.

Multidimensional

General nonlinear solvers:

root(fun, x0[, args, method, jac, tol, ...])	Find a root of a vector function.
fsolve(func, x0[, args, fprime, ...])	Find the roots of a function.
broyden1(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's first Jacobian approximation.
broyden2(F, xin[, iter, alpha, ...])	Find a root of a function, using Broyden's second Jacobian approximation.

Curve fitting:

Suppose we have data sampled from f with some noises.

if we know the functional form of the function from which the sample, were but not the amplitudes of the terms, we can find those by least squares curve fitting.

1. we have to define the function to fit:
2. We have data sampled from f with some noise:
3. can use scipy.optimize.curve_fit() to find a and b

```
def f2(x, a, b):
    return a*x**2 + b*np.sin(x)
xdata = np.linspace(-10, 10, num=20)
ydata = f(xdata) + np.random.randn(xdata.size)
guess = [2, 2]
params, p_cov = optimize.curve_fit(f2, xdata, ydata, guess)
print params
```

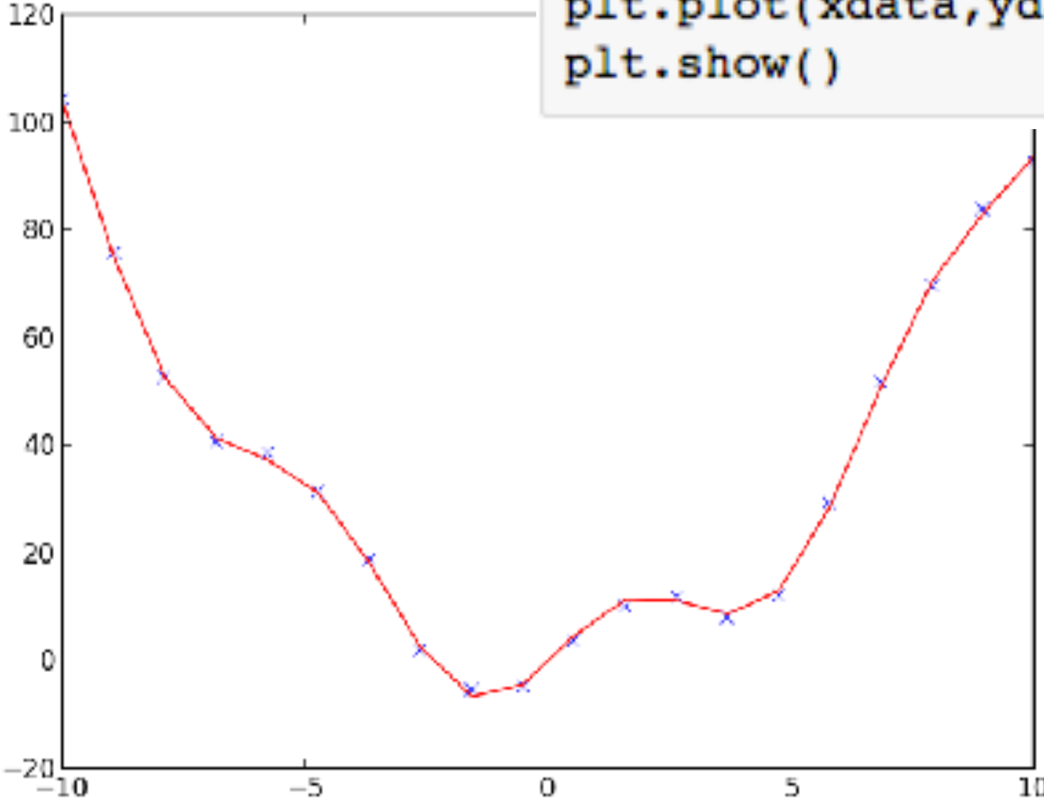
```
[ 0.98797075  9.04290946]
```

Plot the curve and fitted points:

```
def f2(x, a, b):
    return a*x**2 + b*np.sin(x)
xdata = np.linspace(-10, 10, num=20)
ydata = f(xdata) + np.random.randn(xdata.size)
guess = [2, 2]
params, p_cov = optimize.curve_fit(f2, xdata, ydata, guess)
print params
```

```
[ 0.98797075  9.04290946]
```

```
ynew = f2(xdata, params[0], params[1])
plt.plot(xdata,ydata,'x',xdata,ynew,'r-')
plt.show()
```



Histogram and probability density function

Given observations of a random process, their histogram is an estimator of the random process's PDF (probability density function):

```
a = np.random.normal(size=1000)
bins = np.arange(-4, 5)
print bins
histogram = np.histogram(a, bins=bins, normed=True)
print histogram[0]
```

```
[-4 -3 -2 -1  0  1  2  3  4]
[ 0.001  0.022  0.136  0.337  0.351  0.126  0.026  0.001]
```

```
a = np.random.normal(size=1000)
bins = np.arange(-4, 5)
print bins
histogram = np.histogram(a, bins=bins)
print histogram[0]
```

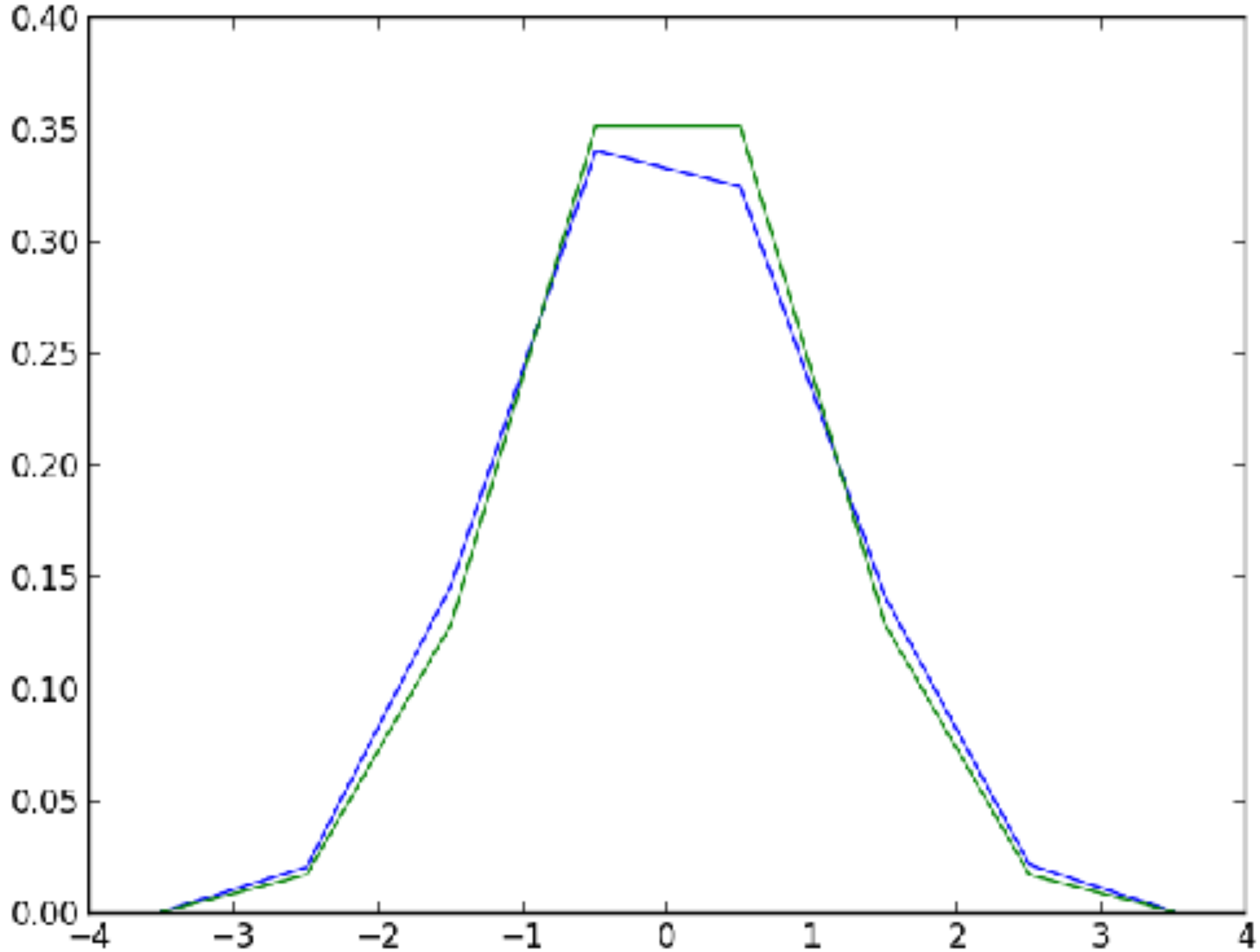
```
[-4 -3 -2 -1  0  1  2  3  4]
[  1  20 137 349 325 149  18   1]
```

Scipy statistic

```
from scipy import stats
b = stats.norm.pdf(bins) # norm is a distribution
print b
```

```
[ 0.00087268  0.0175283   0.1295176   0.35206533  0.35206533  0.1295176
  0.0175283   0.00087268]
```

```
bins = np.arange(-4, 5)
bins = 0.5*(bins[1:] + bins[:-1])

plt.plot(bins, histogram[0])
plt.plot(bins, b)
plt.show()
```

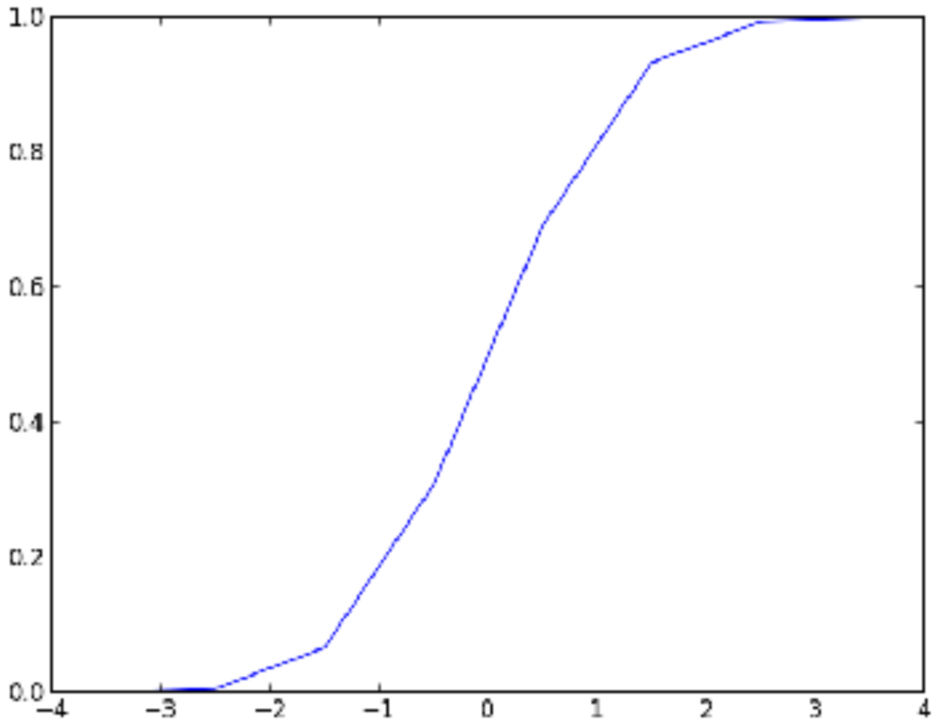

0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4
−3
−2
−1
0
1
2
3
4

cdf: Cumulative Distribution Function

```
c=stats.norm.cdf(bins)
print c
```

```
[  2.32629079e-04   6.20966533e-03   6.68072013e-02   3.08537539e-01
   6.91462461e-01   9.33192799e-01   9.93790335e-01   9.99767371e-01]
```

```
plt.plot(bins,c)
plt.show()
```



Interpolation: scipy.interpolate

The scipy.interpolate is useful for fitting a function from experimental data and thus evaluating points where no measure exists.

```
measured_time = np.linspace(0, 1, 10)
noise = (np.random.random(10)*2 - 1) * 1e-1
measures = np.sin(2 * np.pi * measured_time) + noise
print measures
```

```
[-0.01763823  0.54905976  0.91035641  0.95797226  0.34247089 -0.26868721
 -0.84500759 -0.95461638 -0.63040057  0.02453659]
```

The scipy.interpolate.interp1d class can build a linear interpolation function:

```
from scipy.interpolate import interp1d
linear_interp = interp1d(measured_time, measures)
print linear_interp
```

```
<scipy.interpolate.interpolate.interp1d object at 0x109f5b090>
```

Then the scipy.interpolate.linear_interp instance needs to be evaluated at the time of interest:

```
computed_time = np.linspace(0, 1, 50)
linear_results = linear_interp(computed_time)
print linear_results
```

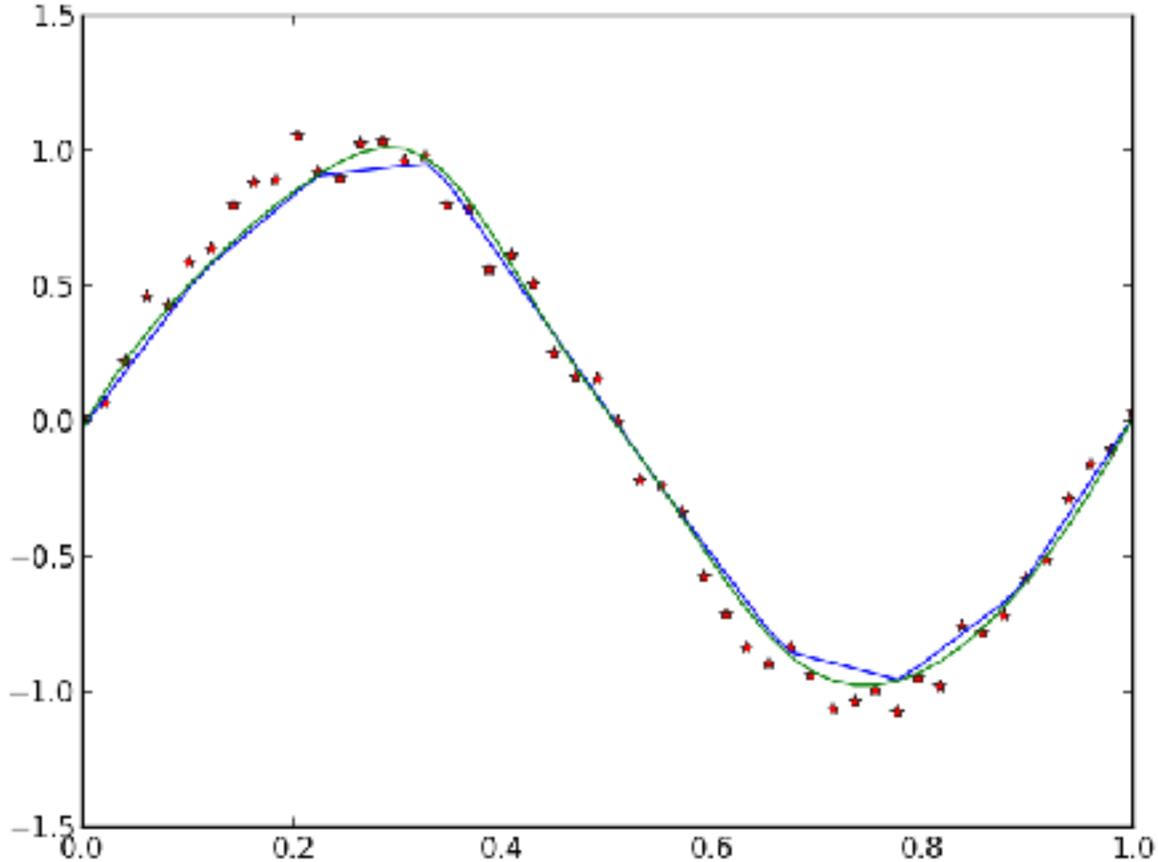
```
[-0.01763823  0.08644915  0.19053654  0.29462393  0.39871131  0.5027987
  0.58592676  0.65228737  0.71864798  0.78500859  0.8513692   0.91132816
  0.92007393  0.9288197   0.93756547  0.94631124  0.955057    0.88260474
  0.76955347  0.6565022   0.54345093  0.43039966  0.31752566  0.20527213
  0.0930186  -0.01923492 -0.13148845 -0.24374198 -0.35101869 -0.45687346
 -0.56272822 -0.66858299 -0.77443775 -0.85171834 -0.87185056 -0.89198279
 -0.91211501 -0.93224724 -0.95237947 -0.90168319 -0.84213334 -0.7825835
 -0.72303366 -0.66348381 -0.57693631 -0.45664173 -0.33634715 -0.21605257
 -0.09575799  0.02453659]
```

A cubic interpolation can also be selected by providing the kind optional keyword argument:

```
cubic_interp = interp1d(measured_time, measures, kind='cubic')
cubic_results = cubic_interp(computed_time)
print cubic_results
```

```
[-0.01763823  0.11354749  0.22989075  0.33369805  0.4272759   0.51293082
  0.59289566  0.6680814   0.73825556  0.80314799  0.86248853  0.91600586
  0.9622944   0.99668571  1.0139283   1.00877064  0.97596123  0.91091701
  0.81682573  0.70188856  0.57439023  0.44261547  0.31482824  0.19591967
  0.08370803 -0.02487896 -0.13291362 -0.24346824 -0.35897128 -0.47673792
 -0.59164496 -0.69855417 -0.79232734 -0.86786676 -0.92250524 -0.95734291
 -0.97380397 -0.9733126  -0.957293   -0.92708041 -0.88351218 -0.82725262
 -0.75896586 -0.67931601 -0.58895386 -0.48812566 -0.37661    -0.25415939
 -0.12052635  0.02453659]
```

```
plt.plot(measured_time ,measures ,'r*')
plt.plot(measured_time,linear_results,'b')
plt.plot(measured_time ,cubic_results,'g')
plt.show()
```



Geometrical transformations on images

```
import numpy as np
import scipy.misc
from scipy import ndimage
import matplotlib.pyplot as plt

face = scipy.misc.face(gray=True)
lx, ly = face.shape
# Cropping
crop_face = face[lx//4:-lx//4, ly//4:-ly//4]
# up <-> down flip
flip_ud_face = np.flipud(face)
# rotation
rotate_face = ndimage.rotate(face, 45)
rotate_face_noreshape = ndimage.rotate(face,
45, reshape=False)
```

```
plt.imshow(rotate_face_noreshape, cmap=plt.cm.gray)
```

http://www.scipy-lectures.org/advanced/image_processing/auto_examples/plot_geom_face.html