scipy

July 17, 2019

1 SciPy - Library of scientific algorithms for Python

J.R. Johansson (jrjohansson at gmail.com) and **edited by Lento**

The latest version of this IPython notebook lecture is available at `http://github.com/jrjohansson/scientific-python-lectures`.

The other notebooks in this lecture series are indexed at `http://jrjohansson.github.io`.

[1]:
```
# what is this line all about? Answer in lecture 4
%matplotlib inline
import matplotlib.pyplot as plt
from IPython.display import Image
```

1.1 Introduction

The SciPy framework builds on top of the low-level NumPy framework for multidimensional arrays, and provides a large number of higher-level scientific algorithms. Some of the topics that SciPy covers are:

- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (scipy.optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (scipy.signal)
- Linear Algebra (scipy.linalg)
- Sparse Eigenvalue Problems (scipy.sparse)
- Statistics (scipy.stats)
- Multi-dimensional image processing (scipy.ndimage)
- File IO (scipy.io)

Each of these submodules provides a number of functions and classes that can be used to solve problems in their respective topics.

In this lecture we will look at how to use some of these subpackages.

To access the SciPy package in a Python program, we start by importing everything from the `scipy` module.

bad

```
from scipy import *
```

good

```
import scipy as sp
```

If we only need to use part of the SciPy framework we can selectively include only those modules we are interested in. For example, to include the linear algebra package under the name `la`, we can do:

[2]:
```
import numpy as np
import scipy as sp
import scipy.linalg as la
```

1.2 Special functions

A large number of mathematical special functions are important for many computional physics problems. SciPy provides implementations of a very extensive set of special functions. For details, see the list of functions in the reference documention at http://docs.scipy.org/doc/scipy/reference/special.html#module-scipy.special.

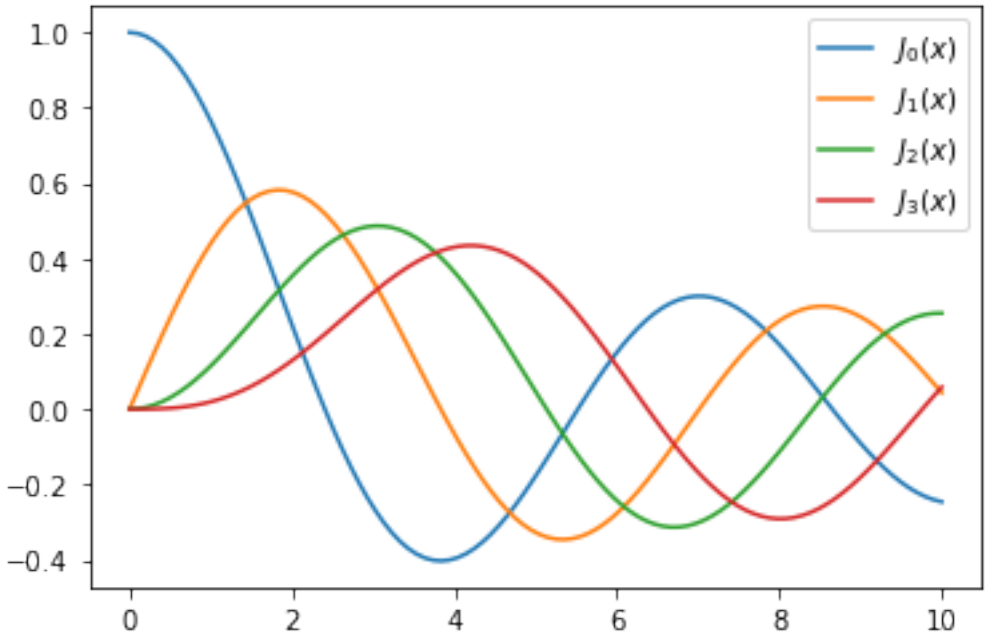
To demonstrate the typical usage of special functions we will look in more detail at the Bessel functions:

[3]:
```
#
# The scipy.special module includes a large number of Bessel-functions
# Here we will use the functions jn and yn, which are the Bessel functions
# of the first and second kind and real-valued order. We also include the
# function jn_zeros and yn_zeros that gives the zeroes of the functions jn
# and yn.
#
from scipy.special import jn, yn, jn_zeros, yn_zeros
```

[4]:
```
n = 0    # order
x = 0.0

# Bessel function of first kind
print("J_{}({}) = {}".format(n, x, jn(n, x)))

x = 1.0
# Bessel function of second kind
print("Y_{}({}) = {}".format(n, x, yn(n, x)))
```

```
J_0(0.0) = 1.0
Y_0(1.0) = 0.08825696421567697
```

[5]:
```
x = np.linspace(0, 10, 100)

fig, ax = plt.subplots()
for n in range(4):
    ax.plot(x, jn(n, x), label=r"$J_%d(x)$" % n)
ax.legend();
```



[6]:
```
# zeros of Bessel functions
n = 0 # order
m = 4 # number of roots to compute
jn_zeros(n, m)
```

[6]:
```
array([ 2.40482556,  5.52007811,  8.65372791, 11.79153444])
```

1.3 Integration

1.3.1 Numerical integration: quadrature

Numerical evaluation of a function of the type

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

is called *numerical quadrature*, or simply *quadature*. SciPy provides a series of functions for different kind of quadrature, for example the `quad`, `dblquad` and `tplquad` for single, double and triple integrals, respectively.

[7]:
```
from scipy.integrate import quad, dblquad, tplquad
```

The `quad` function takes a large number of optional arguments, which can be used to fine-tune the behaviour of the function (try `help(quad)` for details).

The basic usage is as follows:

[8]:
```
# define a simple function for the integrand
def f(x):
    return x
```

[9]:
```
x_lower = 0 # the lower limit of x
x_upper = 1 # the upper limit of x

val, abserr = quad(f, x_lower, x_upper)

print("integral value =", val, ", absolute error =", abserr)
```

```
integral value = 0.5 , absolute error = 5.551115123125783e-15
```

If we need to pass extra arguments to integrand function we can use the `args` keyword argument:

[10]:
```
def integrand(x, n):
    """
    Bessel function of first kind and order n.
    """
    return jn(n, x)


x_lower = 0  # the lower limit of x
x_upper = 10 # the upper limit of x

val, abserr = quad(integrand, x_lower, x_upper, args=(3,))

print(val, abserr)
```

```
0.7366751370811073 9.389126882496403e-13
```

For simple functions we can use a lambda function (name-less function) instead of explicitly defining a function for the integrand:

[11]:
```
val, abserr = quad(lambda x: np.exp(-x ** 2), -np.Inf, np.Inf)

print("numerical  =", val, abserr)

analytical = np.sqrt(np.pi)
print("analytical =", analytical)
```

```
numerical  = 1.7724538509055159 1.4202636780944923e-08
analytical = 1.7724538509055159
```

As show in the example above, we can also use 'Inf' or '-Inf' as integral limits.

Higher-dimensional integration works in the same way:

[12]:
```
def integrand(x, y):
    return np.exp(-x**2-y**2)

x_lower = 0
x_upper = 10
y_lower = 0
```

```
y_upper = 10

val, abserr = dblquad(integrand, x_lower, x_upper, lambda x : y_lower, lambda x:
 ↪ y_upper)

print(val, abserr)
```

```
0.7853981633974476 1.3753098510218528e-08
```

Note how we had to pass lambda functions for the limits for the y integration, since these in general can be functions of x.

1.4 Ordinary differential equations (ODEs)

SciPy provides two different ways to solve ODEs: An API based on the function `odeint`, and object-oriented API based on the class `ode`. Usually `odeint` is easier to get started with, but the `ode` class offers some finer level of control.

Here we will use the `odeint` functions. For more information about the class `ode`, try `help(ode)`. It does pretty much the same thing as `odeint`, but in an object-oriented fashion.

To use `odeint`, first import it from the `scipy.integrate` module

[13]:
```
from scipy.integrate import odeint, ode
```

A system of ODEs are usually formulated on standard form before it is attacked numerically. The standard form is:

$y' = f(y, t)$

where

$y = [y_1(t), y_2(t), ..., y_n(t)]$

and f is some function that gives the derivatives of the function $y_i(t)$. To solve an ODE we need to know the function f and an initial condition, $y(0)$.

Note that higher-order ODEs can always be written in this form by introducing new variables for the intermediate derivatives.

Once we have defined the Python function `f` and array `y_0` (that is f and $y(0)$ in the mathematical formulation), we can use the `odeint` function as:

```
y_t = odeint(f, y_0, t)
```

where `t` is and array with time-coordinates for which to solve the ODE problem. `y_t` is an array with one row for each point in time in `t`, where each column corresponds to a solution `y_i(t)` at that point in time.

We will see how we can implement `f` and `y_0` in Python code in the examples below.

Example: double pendulum Let's consider a physical example: The double compound pendulum, described in some detail here: http://en.wikipedia.org/wiki/Double_pendulum

[14]:
```
Image(url='http://upload.wikimedia.org/wikipedia/commons/c/c9/
 ↪Double-compound-pendulum-dimensioned.svg')
```

[14]:
```
<IPython.core.display.Image object>
```

The equations of motion of the pendulum are given on the wiki page:

$\dot{\theta}_1 = \frac{6}{m\ell^2}\frac{2p_{\theta_1} - 3\cos(\theta_1-\theta_2)p_{\theta_2}}{16-9\cos^2(\theta_1-\theta_2)}$

$\dot{\theta}_2 = \frac{6}{m\ell^2}\frac{8p_{\theta_2} - 3\cos(\theta_1-\theta_2)p_{\theta_1}}{16-9\cos^2(\theta_1-\theta_2)}$.

$\dot{p}_{\theta_1} = -\frac{1}{2}m\ell^2\left[\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) + 3\frac{g}{\ell}\sin\theta_1\right]$

$\dot{p}_{\theta_2} = -\frac{1}{2}m\ell^2\left[-\dot{\theta}_1\dot{\theta}_2\sin(\theta_1-\theta_2) + \frac{g}{\ell}\sin\theta_2\right]$

To make the Python code simpler to follow, let's introduce new variable names and the vector notation: $x = [\theta_1, \theta_2, p_{\theta_1}, p_{\theta_2}]$

$\dot{x}_1 = \frac{6}{m\ell^2}\frac{2x_3 - 3\cos(x_1-x_2)x_4}{16-9\cos^2(x_1-x_2)}$

$\dot{x}_2 = \frac{6}{m\ell^2}\frac{8x_4 - 3\cos(x_1-x_2)x_3}{16-9\cos^2(x_1-x_2)}$

$\dot{x}_3 = -\frac{1}{2}m\ell^2\left[\dot{x}_1\dot{x}_2\sin(x_1-x_2) + 3\frac{g}{\ell}\sin x_1\right]$

$\dot{x}_4 = -\frac{1}{2}m\ell^2\left[-\dot{x}_1\dot{x}_2\sin(x_1-x_2) + \frac{g}{\ell}\sin x_2\right]$

[15]:
```
g = 9.82
L = 0.5
m = 0.1

def dx(x, t):
    """
    The right-hand side of the pendulum ODE
    """
    x1, x2, x3, x4 = x[0], x[1], x[2], x[3]

    dx1 = 6.0/(m*L**2) * (2 * x3 - 3 * np.cos(x1-x2) * x4)/(16 - 9 * np.
↪cos(x1-x2)**2)
    dx2 = 6.0/(m*L**2) * (8 * x4 - 3 * np.cos(x1-x2) * x3)/(16 - 9 * np.
↪cos(x1-x2)**2)
    dx3 = -0.5 * m * L**2 * ( dx1 * dx2 * np.sin(x1-x2) + 3 * (g/L) * np.
↪sin(x1))
    dx4 = -0.5 * m * L**2 * (-dx1 * dx2 * np.sin(x1-x2) + (g/L) * np.sin(x2))

    return [dx1, dx2, dx3, dx4]
```

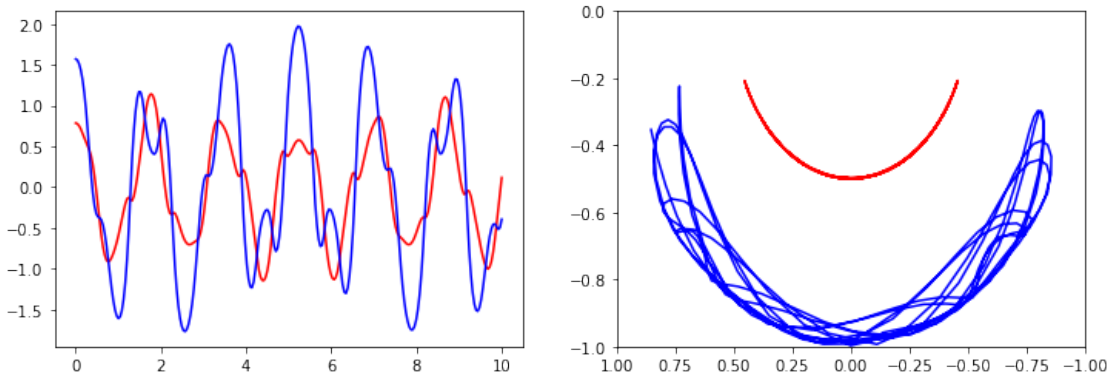
[16]:
```
# choose an initial state
x0 = [np.pi/4, np.pi/2, 0, 0]
```

[17]:
```
# time coodinate to solve the ODE for: from 0 to 10 seconds
t = np.linspace(0, 10, 250)
```

[18]:
```
# solve the ODE problem
x = odeint(dx, x0, t)
```

[19]:
```
# plot the angles as a function of time

fig, axes = plt.subplots(1,2, figsize=(12,4))
axes[0].plot(t, x[:, 0], 'r', label="theta1")
axes[0].plot(t, x[:, 1], 'b', label="theta2")
```

```
x1 = + L * np.sin(x[:, 0])
y1 = - L * np.cos(x[:, 0])

x2 = x1 + L * np.sin(x[:, 1])
y2 = y1 - L * np.cos(x[:, 1])

axes[1].plot(x1, y1, 'r', label="pendulum1")
axes[1].plot(x2, y2, 'b', label="pendulum2")
axes[1].set_ylim([-1, 0])
axes[1].set_xlim([1, -1]);
```



Simple annimation of the pendulum motion. We will see how to make better animation in Lecture 4.

[20]:
```
from matplotlib import animation, rc
from IPython.display import HTML
```

[21]:
```
# First set up the figure, the axis, and the plot element we want to animate
fig, ax = plt.subplots(figsize=(4,4))

ax.set_ylim([-1.5, 0.5])
ax.set_xlim([1, -1])


line1, = ax.plot([], [], 'r.-')
line2, = ax.plot([], [], 'b.-')

# initialization function: plot the background of each frame
def init():
    line1.set_data([], [])
    line2.set_data([], [])
    return (line1,line2)

# animation function. This is called sequentially
def animate(i):
```

```
    x1 = + L * np.sin(x[i, 0])
    y1 = - L * np.cos(x[i, 0])

    x2 = x1 + L * np.sin(x[i, 1])
    y2 = y1 - L * np.cos(x[i, 1])

    line1.set_data([0, x1], [0, y1])
    line2.set_data([x1, x2], [y1, y2])

    return (line1,line2)


# call the animator. blit=True means only re-draw the parts that have changed.
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=200, interval=40, blit=True)

plt.close()
```

[22]:
```
HTML(anim.to_html5_video())
```

[22]:
```
<IPython.core.display.HTML object>
```

Example: Damped harmonic oscillator ODE problems are important in computational physics, so we will look at one more example: the damped harmonic oscillation. This problem is well described on the wiki page: http://en.wikipedia.org/wiki/Damping

The equation of motion for the damped oscillator is:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} + 2\zeta\omega_0\frac{\mathrm{d}x}{\mathrm{d}t} + \omega_0^2 x = 0$$

where x is the position of the oscillator, ω_0 is the frequency, and ζ is the damping ratio. To write this second-order ODE on standard form we introduce $p = \frac{\mathrm{d}x}{\mathrm{d}t}$:

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -2\zeta\omega_0 p - \omega_0^2 x$$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = p$$

In the implementation of this example we will add extra arguments to the RHS function for the ODE, rather than using global variables as we did in the previous example. As a consequence of the extra arguments to the RHS, we need to pass an keyword argument `args` to the `odeint` function:

[23]:
```
def dy(y, t, zeta, w0):
    """
    The right-hand side of the damped oscillator ODE
    """
    x, p = y[0], y[1]

    dx = p
    dp = -2 * zeta * w0 * p - w0**2 * x
```

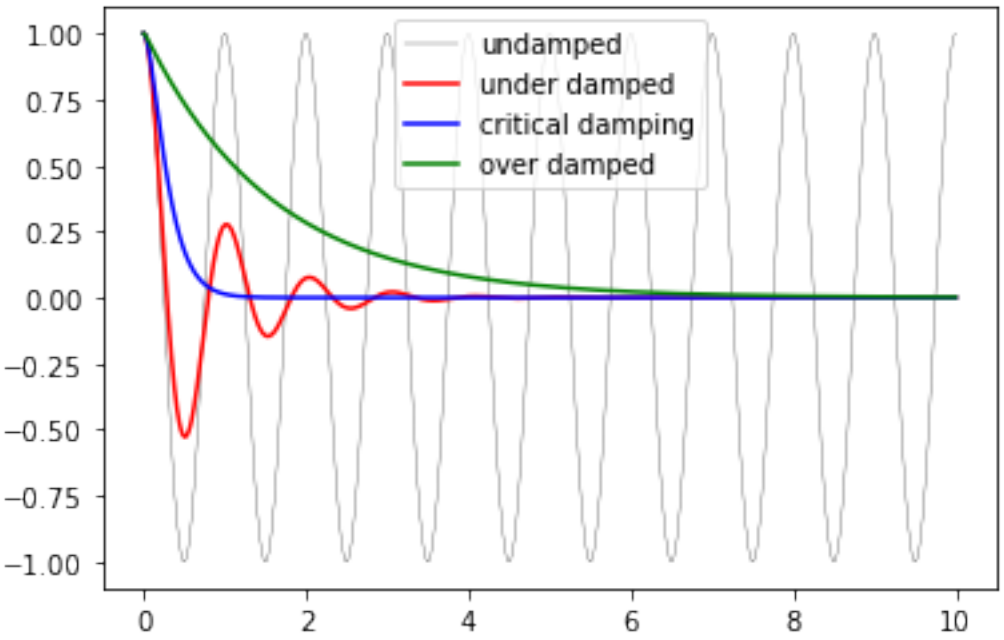
```
    return [dx, dp]
```

[24]:
```
# initial state:
y0 = [1.0, 0.0]
```

[25]:
```
# time coodinate to solve the ODE for
t = np.linspace(0, 10, 1000)
w0 = 2*np.pi*1.0
```

[26]:
```
# solve the ODE problem for three different values of the damping ratio

y1 = odeint(dy, y0, t, args=(0.0, w0)) # undamped
y2 = odeint(dy, y0, t, args=(0.2, w0)) # under damped
y3 = odeint(dy, y0, t, args=(1.0, w0)) # critial damping
y4 = odeint(dy, y0, t, args=(5.0, w0)) # over damped
```

[27]:
```
fig, ax = plt.subplots()
ax.plot(t, y1[:,0], 'k', label="undamped", linewidth=0.25)
ax.plot(t, y2[:,0], 'r', label="under damped")
ax.plot(t, y3[:,0], 'b', label=r"critical damping")
ax.plot(t, y4[:,0], 'g', label="over damped")
ax.legend();
```



1.5 Fourier transform

Fourier transforms are one of the universal tools in computational physics, which appear over and over again in different contexts. SciPy provides functions for accessing the classic FFTPACK

library from NetLib, which is an efficient and well tested FFT library written in FORTRAN. The SciPy API has a few additional convenience functions, but overall the API is closely related to the original FORTRAN library.

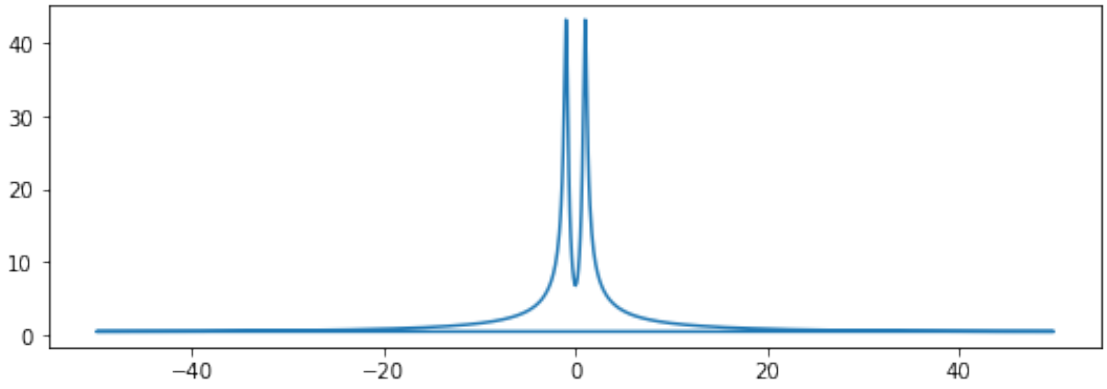
To use the `fftpack` module in a python program, include it using:

[28]:
```
from numpy.fft import fftfreq
from scipy.fftpack import fft
```

To demonstrate how to do a fast Fourier transform with SciPy, let's look at the FFT of the solution to the damped oscillator from the previous section:

[29]:
```
N = len(t)
dt = t[1]-t[0]

# calculate the fast fourier transform
# y2 is the solution to the under-damped oscillator from the previous section
F = fft(y2[:,0])

# calculate the frequencies for the components in F
w = fftfreq(N, dt)
```

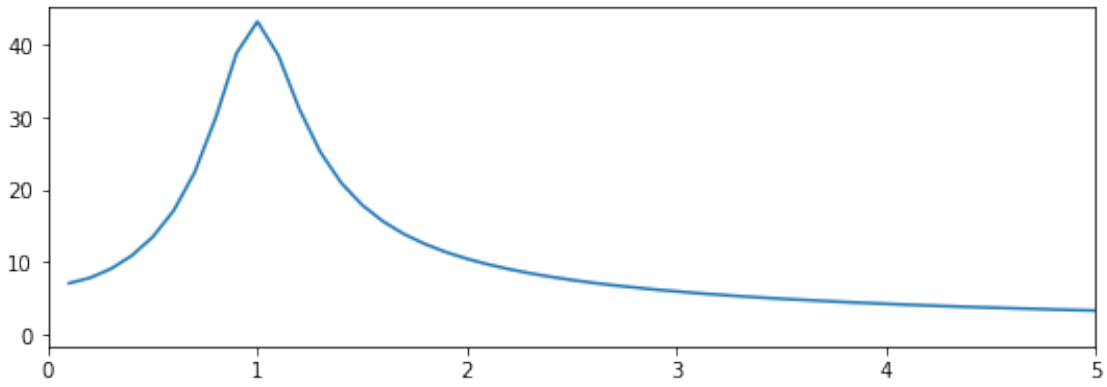
[30]:
```
fig, ax = plt.subplots(figsize=(9,3))
ax.plot(w, abs(F));
```



Since the signal is real, the spectrum is symmetric. We therefore only need to plot the part that corresponds to the postive frequencies. To extract that part of the `w` and `F` we can use some of the indexing tricks for NumPy arrays that we saw in Lecture 2:

[31]:
```
indices = np.where(w > 0) # select only indices for elements that corresponds␣
 ↪to positive frequencies
w_pos = w[indices]
F_pos = F[indices]
```

[32]:
```
fig, ax = plt.subplots(figsize=(9,3))
ax.plot(w_pos, abs(F_pos))
ax.set_xlim(0, 5);
```



As expected, we now see a peak in the spectrum that is centered around 1, which is the frequency we used in the damped oscillator example.

1.6 Linear algebra

The linear algebra module contains a lot of matrix related functions, including linear equation solving, eigenvalue solvers, matrix functions (for example matrix-exponentiation), a number of different decompositions (SVD, LU, cholesky), etc.

Detailed documetation is available at: http://docs.scipy.org/doc/scipy/reference/linalg.html

Here we will look at how to use some of these functions:

1.6.1 Linear equation systems

Linear equation systems on the matrix form

$Ax = b$

where A is a matrix and x, b are vectors can be solved like:

[33]:
```
A = np.array([[3,1,1], [1,2,2], [1,8,9]])
b = np.array([9,8,1])
```

[34]:
```
x = np.linalg.solve(A, b)

x
```

[34]:
```
array([  2.,  28., -25.])
```

[35]:
```
# check
np.dot(A, x) - b
```

[35]:
```
array([ 0.00000000e+00, -7.10542736e-15, -2.84217094e-14])
```

[36]:
```
# Checking if floating point is close
np.allclose(np.dot(A,x),b)
```

[36]:
```
True
```

We can also do the same with

$AX = B$

where A, B, X are matrices:

[37]:
```
A = np.random.rand(3,3)
B = np.random.rand(3,3)
```

[38]:
```
X = np.linalg.solve(A, B)
```

[39]:
```
X
```

[39]:
```
array([[-0.57775233,  1.26903468, -0.4692516 ],
       [ 0.51624969, -3.38856026, -0.93183552],
       [ 0.97494037,  1.45678915,  1.2215408 ]])
```

[40]:
```
# check
np.linalg.norm(np.dot(A, X) - B)
```

[40]:
```
1.401057714162961e-16
```

1.6.2 Eigenvalues and eigenvectors

The eigenvalue problem for a matrix A:

$Av_n = \lambda_n v_n$

where v_n is the nth eigenvector and λ_n is the nth eigenvalue.

To calculate eigenvalues of a matrix, use the `eigvals` and for calculating both eigenvalues and eigenvectors, use the function `eig`:

[41]:
```
evals = np.linalg.eigvals(A)
```

[42]:
```
evals
```

[42]:
```
array([ 1.51256384+0.j        , -0.00380993+0.24213219j,
       -0.00380993-0.24213219j])
```

[43]:
```
evals, evecs = np.linalg.eig(A)
```

[44]:
```
evals
```

[44]:
```
array([ 1.51256384+0.j        , -0.00380993+0.24213219j,
       -0.00380993-0.24213219j])
```

[45]:
```
evecs
```

[45]:
```
array([[ 0.45289941+0.j        ,  0.00272523+0.45992831j,
         0.00272523-0.45992831j],
       [ 0.59526195+0.j        ,  0.75139976+0.j        ,
         0.75139976-0.j        ],
       [ 0.66373589+0.j        , -0.35705594-0.31043192j,
        -0.35705594+0.31043192j]])
```

The eigenvectors corresponding to the nth eigenvalue (stored in `evals[n]`) is the nth *column* in `evecs`, i.e., `evecs[:,n]`. To verify this, let's try mutiplying eigenvectors with the matrix and compare to the product of the eigenvector and the eigenvalue:

[46]:
```
n = 1

np.linalg.norm(np.dot(A, evecs[:,n]) - evals[n] * evecs[:,n])
```

[46]:
```
1.9094028487886839e-16
```

There are also more specialized eigensolvers, like the `eigh` for Hermitian matrices.

1.6.3 Matrix operations

[47]:
```
# the matrix inverse
np.linalg.inv(A)
```

[47]:
```
array([[-0.71782794,  2.08727512, -0.93101281],
       [-4.87168696,  1.27198777,  2.77635012],
       [ 3.23377696, -1.6975985 , -0.0229681 ]])
```

[48]:
```
# determinant
np.linalg.det(A)
```

[48]:
```
0.08870054268660271
```

[49]:
```
# norms of various orders
np.linalg.norm(A, ord=2), np.linalg.norm(A, ord=np.Inf)
```

[49]:
```
(1.5779077020461605, 1.6803009387497139)
```

1.6.4 Sparse matrices

Sparse matrices are often useful in numerical simulations dealing with large systems, if the problem can be described in matrix form where the matrices or vectors mostly contains zeros. Scipy has a good support for sparse matrices, with basic linear algebra operations (such as equation solving, eigenvalue calculations, etc).

There are many possible strategies for storing sparse matrices in an efficient way. Some of the most common are the so-called coordinate form (COO), list of list (LIL) form, and compressed-sparse column CSC (and row, CSR). Each format has some advantanges and disadvantages. Most computational algorithms (equation solving, matrix-matrix multiplication, etc) can be efficiently implemented using CSR or CSC formats, but they are not so intuitive and not so easy to initialize. So often a sparse matrix is initially created in COO or LIL format (where we can efficiently add elements to the sparse matrix data), and then converted to CSC or CSR before used in real calcalations.

For more information about these sparse formats, see e.g. http://en.wikipedia.org/wiki/Sparse_matrix

When we create a sparse matrix we have to choose which format it should be stored in. For example,

[50]:
```
import scipy.sparse as spsparse
```

[52]:
```
# dense matrix
M = np.array([[1,0,0,0], [0,3,0,0], [0,1,1,0], [1,0,0,1]])
M
```

[52]:
```
array([[1, 0, 0, 0],
       [0, 3, 0, 0],
       [0, 1, 1, 0],
       [1, 0, 0, 1]])
```

[53]:
```
# convert from dense to sparse
A = spsparse.csr_matrix(M)
A
```

[53]:
```
<4x4 sparse matrix of type '<class 'numpy.int64'>'
        with 6 stored elements in Compressed Sparse Row format>
```

[54]:
```
# convert from sparse to dense
A.todense()
```

[54]:
```
matrix([[1, 0, 0, 0],
        [0, 3, 0, 0],
        [0, 1, 1, 0],
        [1, 0, 0, 1]], dtype=int64)
```

More efficient way to create sparse matrices: create an empty matrix and populate with using matrix indexing (avoids creating a potentially large dense matrix)

[55]:
```
A = spsparse.lil_matrix((4,4)) # empty 4x4 sparse matrix
A[0,0] = 1
A[1,1] = 3
A[2,2] = A[2,1] = 1
A[3,3] = A[3,0] = 1
A
```

[55]:
```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in LInked List format>
```

[56]:
```
A.todense()
```

[56]:
```
matrix([[1., 0., 0., 0.],
        [0., 3., 0., 0.],
        [0., 1., 1., 0.],
        [1., 0., 0., 1.]])
```

Converting between different sparse matrix formats:

[57]:
```
A
```

[57]:
```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in LInked List format>
```

[58]:
```
A = spsparse.csr_matrix(A)
A
```

[58]:
```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in Compressed Sparse Row format>
```

[59]:
```
A = spsparse.csc_matrix(A)
A
```

[59]:
```
<4x4 sparse matrix of type '<class 'numpy.float64'>'
        with 6 stored elements in Compressed Sparse Column format>
```

We can compute with sparse matrices like with dense matrices:

[60]:
```
A.todense()
```

[60]:
```
matrix([[1., 0., 0., 0.],
        [0., 3., 0., 0.],
        [0., 1., 1., 0.],
        [1., 0., 0., 1.]])
```

[61]:
```
(A * A).todense()
```

[61]:
```
matrix([[1., 0., 0., 0.],
        [0., 9., 0., 0.],
        [0., 4., 1., 0.],
        [2., 0., 0., 1.]])
```

[62]:
```
A.todense()
```

[62]:
```
matrix([[1., 0., 0., 0.],
        [0., 3., 0., 0.],
        [0., 1., 1., 0.],
        [1., 0., 0., 1.]])
```

[63]:
```
A.dot(A).todense()
```

[63]:
```
matrix([[1., 0., 0., 0.],
        [0., 9., 0., 0.],
        [0., 4., 1., 0.],
        [2., 0., 0., 1.]])
```

[64]:
```
v = np.array([1,2,3,4])[:,np.newaxis]
v
```

[64]:
```
array([[1],
       [2],
       [3],
       [4]])
```

[65]:
```
# sparse matrix - dense vector multiplication
A * v
```

[65]:
```
array([[1.],
       [6.],
       [5.],
       [5.]])
```

[66]:
```
# same result with dense matrix - dense vector multiplcation
A.todense() * v
```

[66]:
```
matrix([[1.],
        [6.],
        [5.],
        [5.]])
```

1.7 Optimization

Optimization (finding minima or maxima of a function) is a large field in mathematics, and optimization of complicated functions or in many variables can be rather involved. Here we will only look at a few very simple cases. For a more detailed introduction to optimization with SciPy see: http://scipy-lectures.github.com/advanced/mathematical_optimization/index.html

To use the optimization module in scipy first include the `optimize` module:

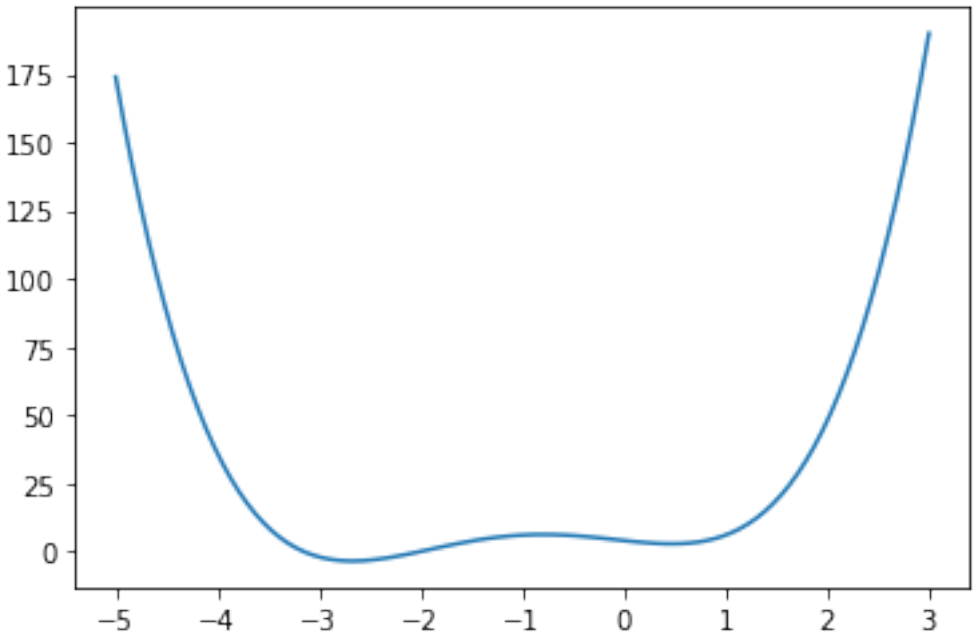
[67]:
```
from scipy import optimize
```

1.7.1 Finding a minima

Let's first look at how to find the minima of a simple function of a single variable:

[68]:
```
def f(x):
    return 4*x**3 + (x-2)**2 + x**4
```

[69]:
```
fig, ax  = plt.subplots()
x = np.linspace(-5, 3, 100)
ax.plot(x, f(x));
```



We can use the `fmin_bfgs` function to find the minima of a function:

[70]:
```
x_min = optimize.fmin_bfgs(f, -2)
x_min
```

```
Optimization terminated successfully.
         Current function value: -3.506641
         Iterations: 5
```

```
         Function evaluations: 24
         Gradient evaluations: 8
```

[70]:
```
array([-2.67298151])
```

[71]:
```
optimize.fmin_bfgs(f, 0.5)
```

```
Optimization terminated successfully.
         Current function value: 2.804988
         Iterations: 3
         Function evaluations: 15
         Gradient evaluations: 5
```

[71]:
```
array([0.46961745])
```

We can also use the `brent` or `fminbound` functions. They have a bit different syntax and use different algorithms.

[72]:
```
optimize.brent(f)
```

[72]:
```
0.46961743402759754
```

[73]:
```
optimize.fminbound(f, -4, 2)
```

[73]:
```
-2.6729822917513886
```

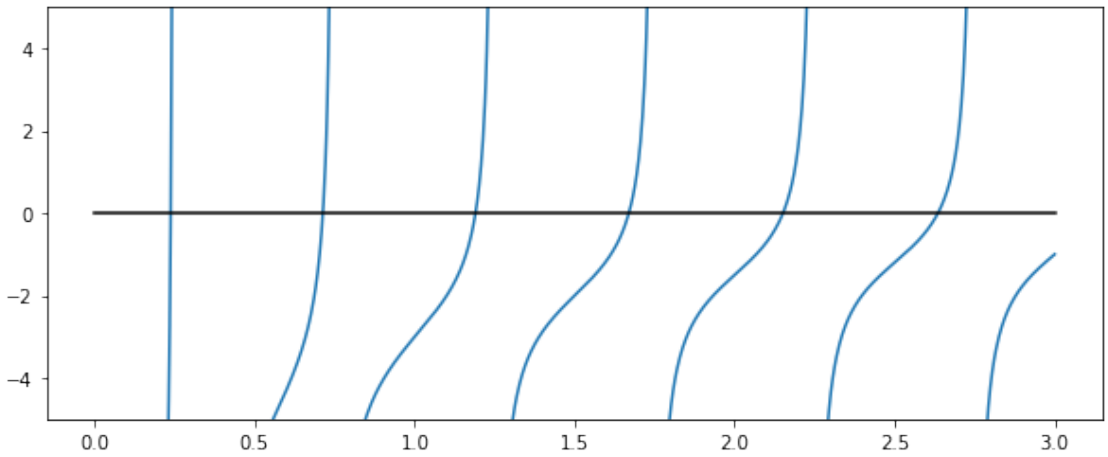
1.7.2 Finding a solution to a function

To find the root for a function of the form $f(x) = 0$ we can use the `fsolve` function. It requires an initial guess:

[74]:
```
omega_c = 3.0
def f(omega):
    # a transcendental equation: resonance frequencies of a low-Q SQUID␣
 ↪terminated microwave resonator
    return np.tan(2*np.pi*omega) - omega_c/omega
```

[75]:
```
fig, ax  = plt.subplots(figsize=(10,4))
x = np.linspace(0, 3, 1000)
y = f(x)
mask = np.where(abs(y) > 50)
x[mask] = y[mask] = np.NaN # get rid of vertical line when the function flip␣
 ↪sign
ax.plot(x, y)
ax.plot([0, 3], [0, 0], 'k')
ax.set_ylim(-5,5);
```

```
/home/lento/apps/anaconda3/envs/tutorial/lib/python3.7/site-
packages/ipykernel_launcher.py:4: RuntimeWarning: divide by zero encountered in
true_divide
  after removing the cwd from sys.path.
```



[76]:
```
optimize.fsolve(f, 0.1)
```

[76]:
```
array([0.23743014])
```

[77]:
```
optimize.fsolve(f, 0.6)
```

[77]:
```
array([0.71286972])
```

[78]:
```
optimize.fsolve(f, 1.1)
```

[78]:
```
array([1.18990285])
```

1.8 Interpolation

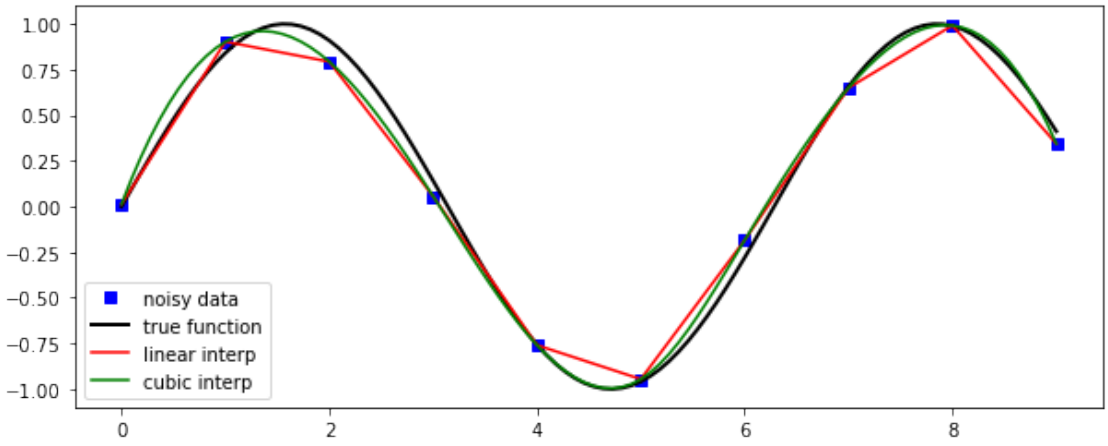
Interpolation is simple and convenient in scipy: The `interp1d` function, when given arrays describing X and Y data, returns and object that behaves like a function that can be called for an arbitrary value of x (in the range covered by X), and it returns the corresponding interpolated y value:

[79]:
```
import scipy.interpolate as spinterp
```

[80]:
```
def f(x):
    return np.sin(x)
```

[81]:
```
n = np.arange(0, 10)
x = np.linspace(0, 9, 100)

y_meas = f(n) + 0.1 * np.random.randn(len(n)) # simulate measurement with noise
y_real = f(x)

linear_interpolation = spinterp.interp1d(n, y_meas)
y_interp1 = linear_interpolation(x)

cubic_interpolation = spinterp.interp1d(n, y_meas, kind='cubic')
y_interp2 = cubic_interpolation(x)
```

[82]:
```
fig, ax = plt.subplots(figsize=(10,4))
ax.plot(n, y_meas, 'bs', label='noisy data')
ax.plot(x, y_real, 'k', lw=2, label='true function')
ax.plot(x, y_interp1, 'r', label='linear interp')
ax.plot(x, y_interp2, 'g', label='cubic interp')
ax.legend(loc=3);
```



1.9 Statistics

The `scipy.stats` module contains a large number of statistical distributions, statistical functions and tests. For a complete documentation of its features, see http://docs.scipy.org/doc/scipy/reference/stats.html.

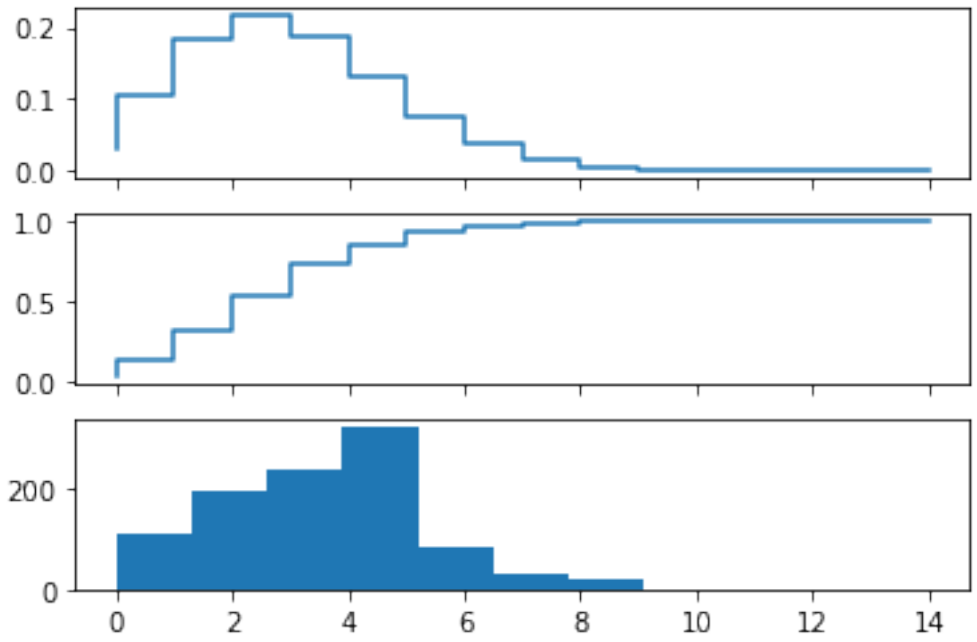
There is also a very powerful python package for statistical modelling called statsmodels. See http://statsmodels.sourceforge.net for more details.

[83]:
```
from scipy import stats
```

[84]:
```
# create a (discreet) random variable with poissionian distribution

X = stats.poisson(3.5) # photon distribution for a coherent state with n=3.5␣
 ↪photons
```

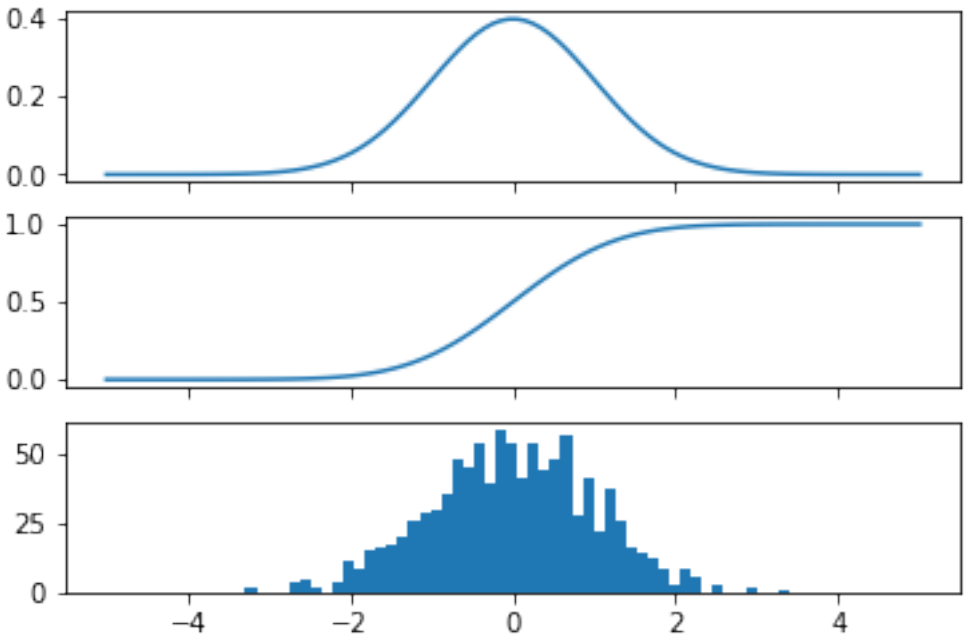
[85]:
```
n = np.arange(0,15)

fig, axes = plt.subplots(3,1, sharex=True)

# plot the probability mass function (PMF)
axes[0].step(n, X.pmf(n))

# plot the commulative distribution function (CDF)
axes[1].step(n, X.cdf(n))

# plot histogram of 1000 random realizations of the stochastic variable X
```

```
axes[2].hist(X.rvs(size=1000));
```



[86]:
```
# create a (continous) random variable with normal distribution
Y = stats.norm()
```

[87]:
```
x = np.linspace(-5,5,100)

fig, axes = plt.subplots(3,1, sharex=True)

# plot the probability distribution function (PDF)
axes[0].plot(x, Y.pdf(x))

# plot the commulative distributin function (CDF)
axes[1].plot(x, Y.cdf(x));

# plot histogram of 1000 random realizations of the stochastic variable Y
axes[2].hist(Y.rvs(size=1000), bins=50);
```



Statistics:

[88]:
```
X.mean(), X.std(), X.var() # poission distribution
```

[88]:
```
(3.5, 1.8708286933869707, 3.5)
```

[89]:
```
Y.mean(), Y.std(), Y.var() # normal distribution
```

[89]:
```
(0.0, 1.0, 1.0)
```

1.9.1 Statistical tests

Test if two sets of (independent) random data comes from the same distribution:

[90]:
```
t_statistic, p_value = stats.ttest_ind(X.rvs(size=1000), X.rvs(size=1000))

print("t-statistic =", t_statistic)
print("p-value =", p_value)
```

```
t-statistic = -0.845809665741376
p-value = 0.39776021333800704
```

Since the p value is very large we cannot reject the hypothesis that the two sets of random data have *different* means.

To test if the mean of a single sample of data has mean 0.1 (the true mean is 0.0):

[91]:
```
stats.ttest_1samp(Y.rvs(size=1000), 0.1)
```

[91]:
```
Ttest_1sampResult(statistic=-3.9574432022750643, pvalue=8.111617569847899e-05)
```

Low p-value means that we can reject the hypothesis that the mean of Y is 0.1.

[92]:
```
Y.mean()
```

[92]:
```
0.0
```

[93]:
```
stats.ttest_1samp(Y.rvs(size=1000), Y.mean())
```

[93]:
```
Ttest_1sampResult(statistic=-0.6733533924415327, pvalue=0.5008782354150199)
```

1.10 Further reading

- http://www.scipy.org - The official web page for the SciPy project.
- http://docs.scipy.org/doc/scipy/reference/tutorial/index.html - A tutorial on how to get started using SciPy.
- https://github.com/scipy/scipy/ - The SciPy source code.