Some Help on Scipy Functions

1 How to use scipy.integrate.odeint

result=scipy.integrate.odeint(func, solve, time, args=() ....)

Consider a second order differential equation e.g.

$$\frac{d^2x}{dt^2} = f(x) \tag{1}$$

By introducing a variable xp which is simply the differential of x this can be replaced by two coupled first order equations:

$$\dot{x} = \frac{dx}{dt} \tag{2}$$

$$\frac{d(\dot{x})}{dt} = \frac{d^2x}{dt^2} = f(x) \tag{3}$$

One call to **scipy.integrate.odeint** will simultaneously solve these to give x and $\dot{x}$ as a function of t. The parameters you need to supply are as follows:

func - this is the name of a user supplied function (i.e. you write it!) which must return $\frac{dx}{dt}$ (which is just the current $\dot{x}$) and $\frac{d\dot{x}}{dt}$. The inputs to **func** are (in order) **solve**, t, and any extra arguments passed by **args=(...)**. Note that **solve** will contain the current values (i.e. at time t) of x and $\dot{x}$. You should store the return values in the same length of vector as **solve**.

solve - For our simple example, this is a vector (1D array) of two elements containing the values of x and $\dot{x}$ at time $t = 0$ (i.e. the initial values). During the calculation, this is used to store the values of x and $\dot{x}$ at time t.

time - this is simply a 1D array containing all the values of t (in numerical order) at which we wish to calculate the values of x.

args=(...) - any variables inside the brackets (it is a tuple, so they need to be separated by commas) are passed to **func** as extra parameters after **solve** and t (so, for example, you would pass here any constants you need in the calculation of $f(x)$).

On exit, **odeint** will return an $N \times 2$ array, where N is the number of output times you specified (len(**time**)), containing the solved values of x and dx/dt at each time t.

There are other possible parameters which control the internal workings of **odeint** but you probably won't need to use these.

If you have more than two coupled equations then simply increase the length of **solve** to accommodate the extra values you wish to solve for. So if you had a three dimensional problem with coupled equations in x, y and z you would store x, y, z, $\dot{x}$, $\dot{y}$, $\dot{z}$ in **solve** and **func** would have to return $\frac{dx}{dt}, \frac{dy}{dt}, \frac{dz}{dt}, \frac{d\dot{x}}{dt}, \frac{d\dot{y}}{dt}, \frac{d\dot{z}}{dt}$.

An example

Let's solve the equation for simple harmonic motion: $\ddot{x} = -\omega^2 x$

```
"""this is an example of how to use odeint to solve 2nd order equation.
It stores variables as x[0]=position, x[1]=velocity."""

import numpy as np
import scipy.integrate as SP
mport matplotlib.pyplot as plt


def myfunc(x, t, k):       # must include t, even though it is not used
    """returns dx/dt and dxdot/dt for SHM. k is the spring constant"""
    return  (x[1],  -1.0 * k**2 * x[0])

def solve_it(t, k):
    """solves the equation and returns solution"""
    solve_me = np.array( [0., 1.])              # start position and velocity
    return SP.odeint( myfunc, solve_me, t, args=( k, ) )  # note "," after k

if __name__ == "__main__" :
    const = 2.0
    times = np.arange( 100.0 ) /10.
    answer = solve_it( times, const )
    plt.plot( times, answer[:,0] )    # plot position vs time
    plt.plot( times, np.sin(const*times)/const +0.05)  # analytic solution, shifted up
    plt.show()
```

2 How to use scipy.linalg.eig

result= scipy.linalg.eig(M))

A Matrix **M** has associated eigen values, λ and eigen vectors **E**, that are defined by

$$\mathbf{M}.\mathbf{E} = \lambda \mathbf{E}.$$

Scipy provides routines to calculate λ and $\mathbf{E}$. For example, if

$$\mathbf{M} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

We can compute the eigenvalues and vectors eig(M). This returns a list of eigenvalues and a list of eigenvectors, as the following example makes clear.

There is no particular order to the eigenvalues, you may need to use argsort to find the smallest one, or to put them in the right order.

```
""" example of how to solve eigen values and vectors from set of
linear equations."""

import numpy as np
import scipy.linalg as la

M = np.array( [ [1., 2.], [3., 4.] ] )

(eval, evec) = la.eig(M)

# note that the returned eigen value/vector may be complex.
# in this case, it is real, so uncomment if you wish...
##  eval = np.real(eval)

isort=np.argsort( np.abs(eval) )     # sort by magnitude of eigen value

i = isort[0]
print "smallest eigen value and vector", eval[i], evec[:,i]

# check that it really works.
check = np.dot(M, evec[:,i]) - eval[i] * evec[:,i]
print "    solution accuracy = ", check

i = isort[1]
print "largest eigen value and vector", eval[i], evec[:,i]
# check that it really works.
check = np.dot(M, evec[:,i]) - eval[i] * evec[:,i]
print "    solution accuracy = ", check
```

Some things to note. The eigen values/vectors may be complex. Use eigh if your matrix is symmetric etc. If you do not want to compute the vectors, just the values it may be simpler to use the eigvals or eigvalsh functions.

Numpy includes routines to manipulate matrices. For example numpy.dot(M, N) will multiply matrices M and N. We can use this to check the eigen values/vectors really work. There are other important matrix functions too: inv takes the inverse of a matrix, det computes the determinant, etc. This is very powerful: matrix equations are solved very easily. More help is available at

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

3 How to use Spherical Bessel Functions

Scipy provides many different special functions, including routines to calculate Bessel functions and their derivatives. These are briefly described at

https://docs.scipy.org/doc/scipy/reference/special.html

In the resonant scattering project, you need to be able to calculate the Spherical Bessel functions and their derivatives. The recommended way to do this is to use the sph_jnyn function. Here is an example to show you how to do this.

```
"""Example of how to compute the spherical Bessel functions and
derivatives"""

import numpy as np
import scipy.special as ss

order = 5                        # order must be an integer
radius = 1.5                     # radius at which to evaluate

# sph_jnjy return a list of arrays, I use numpy to turn it into a
# 2d array. It returns all the orders up to 5, you have to select
# the one you want.

jnyn = np.array( ss.sph_jnyn( order, radius ) )
print "this creates an array of shape ", np.shape(jnyn)

print "Computed Bessel functions of order 5 at radius %5g" % radius
print  "value       j5(%5g ) = %7.3g"  %  ( radius,  jnyn[ 0, order ] )
print  "derivative j'5(%5g ) = %7.3g"  %  ( radius,  jnyn[ 1, order ] )
print  "value       y5(%5g ) = %7.3g"  %  ( radius,  jnyn[ 2, order ] )
print  "derivative y'5(%5g ) = %7.3g"  %  ( radius,  jnyn[ 3, order ] )
```