Python – Numerical Analysis

The modules of interest are `numpy` and `numpy.scipy`.

```
import numpy as np
import scipy.optimize as resol
import scipy.integrate as integr
import matplotlib.pyplot as plt
```

1. Complex numbers. It is possible to declare and manipulate complex numbers in Python. The complex imaginary number i is denoted `1j`. The real and imaginary parts of a complex number can be obtained by the attributes `real` and `imag`. The modulus of a complex number can be obtained with the function `abs`.

```
>>> a = 2 + 3j
>>> b = 5 - 3j
>>> a*b
(19+9j)
>>> a.real
2.0
>>> a.imag
3.0
>>> abs(a)
3.6055512754639896
```

2. Mathematical functions. The constant π is obtained by the command `pi`. Many standard mathematical functions are available through the module `numpy.scipy`. The floor of the scalar x (the largest integer i such that $i \leq x$). is obtained by the command `floor`. The natural logarithm function ln is obtained by the command `log`.

```
>>> np.exp(1)
2.7182818284590451
>>> np.cos(np.pi)
-1.0
>>> np.log(np.exp(1))
1.0
>>> np.floor(3.4)
3
>>> np.floor(-3.7)
-4
```

3. Approximate solution of equations. To solve the algebraic equation $f(x) = 0$, where f is a real-valued function of a real variable x, it is possible to use the function `solve` of the module `scipy.optimize`. It is necessary to indicate the initial value x_0 for the search algorithm. The result will depend on the value of x_0 in case of multiple solutions of $f(x) = 0$.

```
def f(x):
    return x**2 - 2

>>> resol.fsolve(f, -2.)
array([-1.41421356])

>>> resol.fsolve(f, 2.)
array([ 1.41421356])
```

In case of a vector valued function, one uses the function `root`. For instance, to solve the nonlinear system of equations

$$\begin{cases} x^2 - y^2 = 1 \\ x + 2y - 3 = 0 \end{cases}$$

```
def f(v):
    return v[0]**2 - v[1]**2 - 1, v[0] + 2*v[1] - 3

>>> sol = resol.root(f, [0,0])
>>> sol.success
True
>>> sol.x
array([ 1.30940108,  0.84529946])

>>> sol=resol.root(f, [-5,5])
>>> sol.success
True
>>> sol.x
array([-3.30940108,  3.15470054])
```

3. Numerical determination of integrals. The function `quad` of the module `scipy.integrate` allows for the approximation calculation of definite integrals, with finite or infinite bounds. The function returns an approximate value of the integral and an error estimate.

```
def f(x):
    return np.exp(-x)

>>> integr.quad(f, 0, 1)
(0.6321205588285578, 7.017947987503856e-15)

>>> integr.quad(f, 0, np.inf)
(1.0000000000000002, 5.842607038578007e-11)
```

The function `quad` can be used for the definition of integrals with one or more parameters. For instance, to calculate approximate values of the gamma function $\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$ for $x > 0$, one can proceed as follows

```
def g(x):
    def f(t):
        return np.exp(-t)*t**(x-1)
    return integr.quad(f,0,np.inf)[0]

>>> g(2)
0.9999999999999998
```

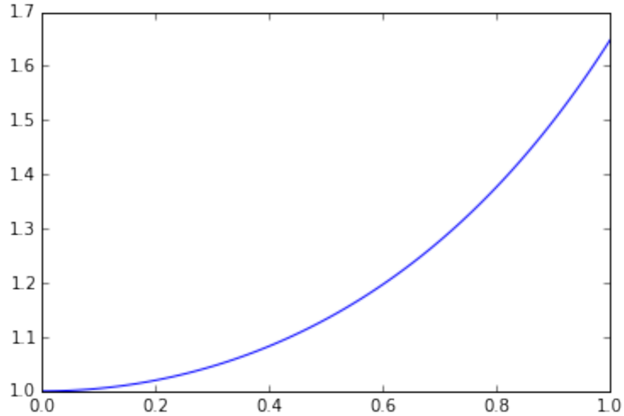
4. Numerical solution of ODEs.

To solve an ODE of the type $\dot{x} = f(x, t)$, one can use the function `odeint` of the module `scipy.integrate`. This requires a list of t values, starting with the initial time t_0, and an initial condition x_0. The function returns approximate values of $x(t)$ at the values t of the supplied list satisfying $\dot{x} = f(x, t)$ satisfying $x(t_0) = x_0$. For instance, to find the solution of equation $\dot{x} = tx(t)$ satisfying $x(0) = 1$ on the interval $0 \le x \le 1$, we can use the following code

```
def f(x, t):
    return t*x

>>> T = np.arange(0, 1.01, 0.01)
>>> X = integr.odeint(f, 1, T)
>>> X[0]
array([ 1.])
>>> X[-1]
array([ 1.64872143])
>>> plt.plot(T,X)
>>> plt.show()
```

Here is the graphical output:



As a second example, consider the following system on the interval $0 \leq t \leq 1$:

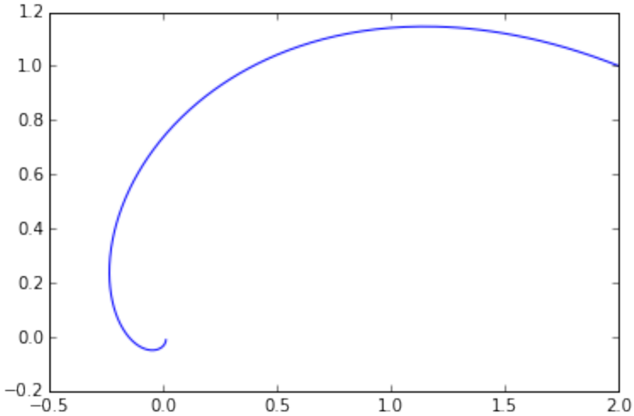
$$\begin{cases} \dot{x} = -x - y \\ \dot{y} = x - y \end{cases}$$

with the initial conditions $x(0) = 2$ and $y(0) = 1$. We can use the following code:

```
def f(x, t):
    return np.array([-x[0]-x[1], x[0]-x[1]])

>>> T = np.arange(0, 5.01, 0.01)
>>> X = integr.odeint(f, np.array([2.,1.]), T)
>>> X[0]
array([ 2.,  1.])
>>> plt.plot(X[:,0], X[:,1])
>>> plt.show()
```

Here is the graphical output of y vs x:



In order to solve a ODE of order 2 governing scalar function $x(t)$, the ODE must first be converted to an ODE of order 1 governing vector $\mathbf{X} = \begin{pmatrix} x(t) \\ \dot{x}(t) \end{pmatrix}$. For instance consider the ODE

$$\begin{cases} \ddot{x} + 2\dot{x} + 3x = \sin(t) \\ x(0) = 0, \dot{x}(0) = 1 \end{cases}$$

converted to the system

$$\begin{cases} \dot{X}_1 = X_2 \\ \dot{X}_2 = -2X_2 - 3X_1 + \sin(t) \end{cases} \qquad 0 \leq t \leq 3\pi$$

with the initial conditions $X_1(0) = 0$ and $X_2(0) = 1$. We can use the following code:

```
def f(x,t):
    return np.array([x[1], -2*x[1] - 3*x[0] + np.sin(t)])

T = np.arange(0, 3*np.pi + 0.01, 0.01)
X = integr.odeint(f, np.array([0,1]), T)
plt.plot(T, X[:,0])
plt.show()
```

Here is the graphical output of x vs t: