031_Scipy

March 9, 2022

[1]:
```
from IPython.display import Image
```

1 Intermediate Numpy and Scipy

- Section **??**
- Section **??**

- Section **??**

- Section **??**

- Section **??**

2 Numpy and Scipy

As we have seen Numpy contains core routines for doing fast vector, matrix, and linear algebra-type operations in Python. Scipy contains additional routines for optimization, special functions, and so on. Both contain modules written in C and Fortran so that they're as fast as possible. Together, they give Python roughly the same capability that the Matlab program offers. (In fact, if you're an experienced Matlab user, there a guide to Numpy for Matlab users just for you.)

A recap of basic objects and (*universal*) functions we have seen includes:

numpy.ndarray

numpy.ones, numpy.zeros, numpy.eye

numpy.arange, numpy.linspace

numpy.diag

numpy.sort, numpy.argsort

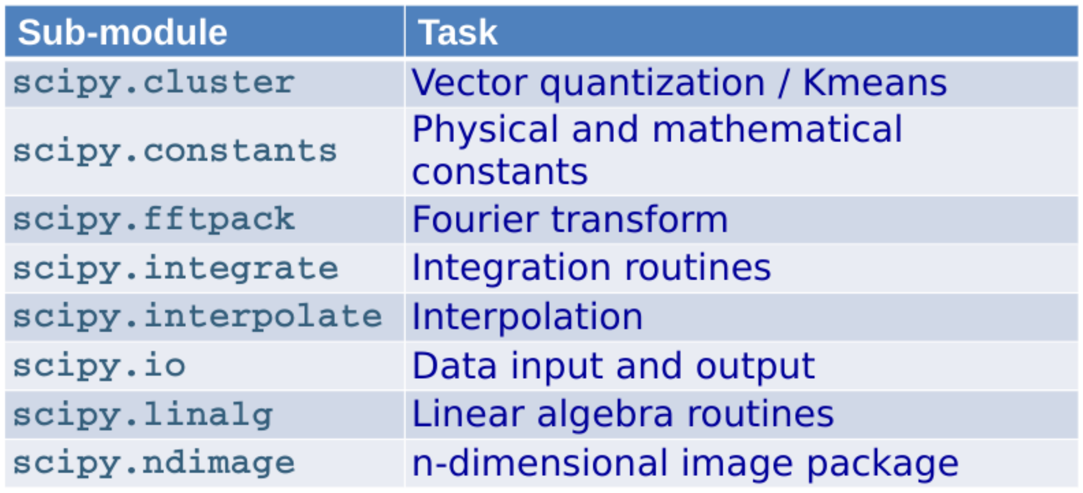
… and many more

[2]:
```
Image(filename="logos.png")
```

[2]:

```
[3]: Image(filename="scipysubm.png")
```

[3]:



Sub-module	Task
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

```
[4]: import math
     import numpy as np
     import scipy as sp
     import scipy.constants as CONST
     PI = CONST.pi
```

3 Matplotlib

```
[5]: import matplotlib.pyplot as plt
     %matplotlib inline
```

```
[6]: Image(filename="mpllogo.png")
```

[6]:

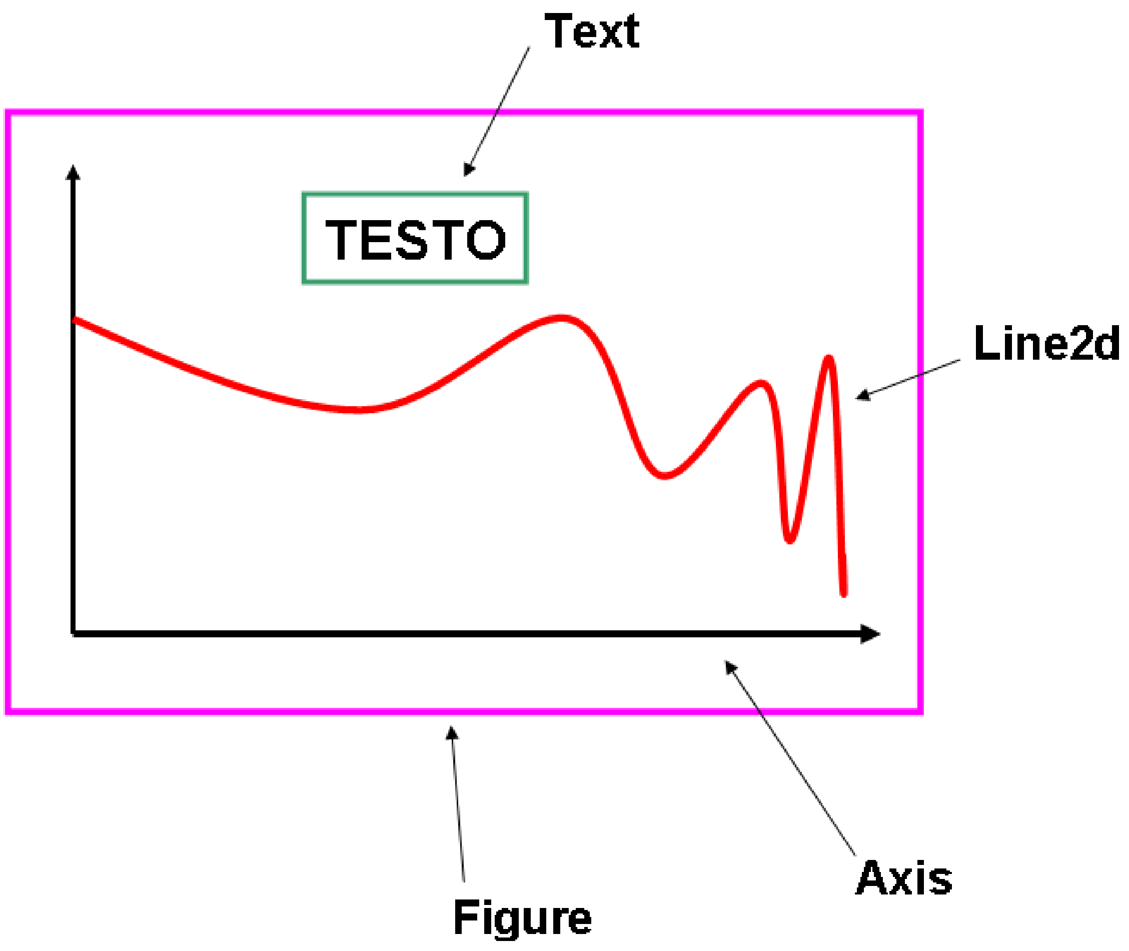
The base module for plotting 2D and 3D within the Scientific Python Ecosystem is *Matplotlib*.

The module main parts are:

1. figure: this objects contains a plot and all its atributes (resolution, ticks …)
2. Line2d: object to plot lines, markers ….
3. Text: for everything about text (plain or math, using Tex)
4. Axis: to manage axes

[7]:
```
Image(filename="plskeleton.png")
```

[7]:



Pylab is a module in matplotlib that gets installed alongside matplotlib; and matplotlib.pyplot is a module in matplotlib.

Pyplot provides the state-machine interface to the underlying ploting library in matplotlib. This means that figures and axes are implicitly and automatically created to achieve the desired plot. Setting a title will then automatically set that title to the current axes object.

It is a collection of command style functions that make matplotlib work like MATLAB. Each pyplot unction makes some change to a figure: eg, create a figure, create a ploting area in a figure, plot some lines in a ploting area, decorate the plot with labels, etc….

Pylab combines the pyplot functionality (for ploting) with the numpy functionality (for mathematics and for working with arrays) in a single namespace, For example, one can call the sin and cos functions just like you could in MATLAB, as well as having all the features of pyplot.

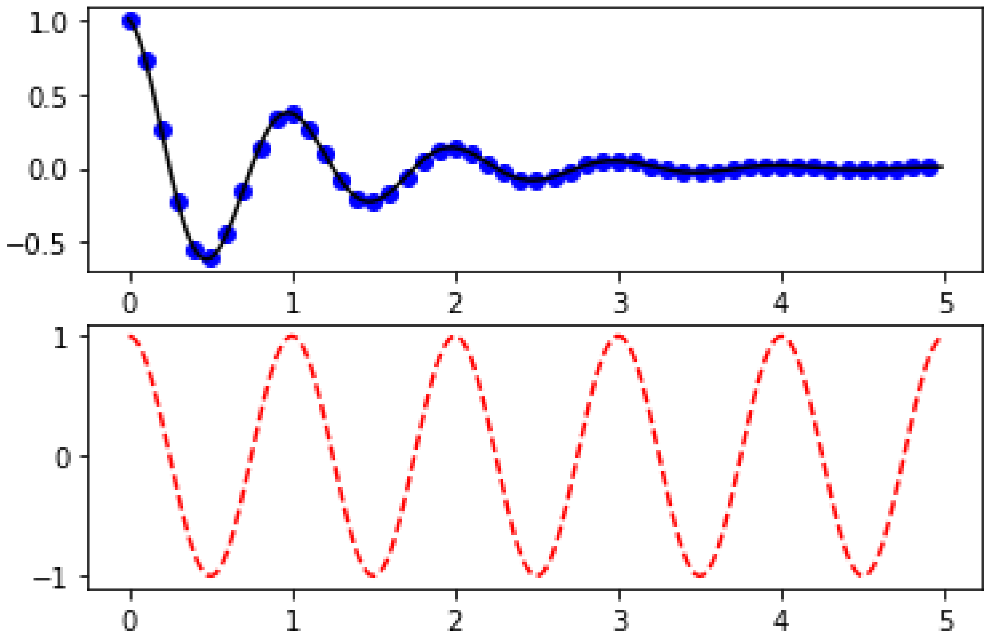
The *pyplot interface used to be generally preferred for non-interactive ploting* (i.e., scripting). The *pylab interface used to be convenient for interactive calculations and plotting*, as it minimizes typing.

[8]:
```python
def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
print(t1,t2)
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7
 1.8 1.9 2.  2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.  3.1 3.2 3.3 3.4 3.5
 3.6 3.7 3.8 3.9 4.  4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9] [0.   0.02 0.04 0.06
0.08 0.1  0.12 0.14 0.16 0.18 0.2  0.22 0.24 0.26
 0.28 0.3  0.32 0.34 0.36 0.38 0.4  0.42 0.44 0.46 0.48 0.5  0.52 0.54
 0.56 0.58 0.6  0.62 0.64 0.66 0.68 0.7  0.72 0.74 0.76 0.78 0.8  0.82
 0.84 0.86 0.88 0.9  0.92 0.94 0.96 0.98 1.   1.02 1.04 1.06 1.08 1.1
 1.12 1.14 1.16 1.18 1.2  1.22 1.24 1.26 1.28 1.3  1.32 1.34 1.36 1.38
 1.4  1.42 1.44 1.46 1.48 1.5  1.52 1.54 1.56 1.58 1.6  1.62 1.64 1.66
 1.68 1.7  1.72 1.74 1.76 1.78 1.8  1.82 1.84 1.86 1.88 1.9  1.92 1.94
 1.96 1.98 2.   2.02 2.04 2.06 2.08 2.1  2.12 2.14 2.16 2.18 2.2  2.22
 2.24 2.26 2.28 2.3  2.32 2.34 2.36 2.38 2.4  2.42 2.44 2.46 2.48 2.5
 2.52 2.54 2.56 2.58 2.6  2.62 2.64 2.66 2.68 2.7  2.72 2.74 2.76 2.78
 2.8  2.82 2.84 2.86 2.88 2.9  2.92 2.94 2.96 2.98 3.   3.02 3.04 3.06
 3.08 3.1  3.12 3.14 3.16 3.18 3.2  3.22 3.24 3.26 3.28 3.3  3.32 3.34
 3.36 3.38 3.4  3.42 3.44 3.46 3.48 3.5  3.52 3.54 3.56 3.58 3.6  3.62
 3.64 3.66 3.68 3.7  3.72 3.74 3.76 3.78 3.8  3.82 3.84 3.86 3.88 3.9
 3.92 3.94 3.96 3.98 4.   4.02 4.04 4.06 4.08 4.1  4.12 4.14 4.16 4.18
 4.2  4.22 4.24 4.26 4.28 4.3  4.32 4.34 4.36 4.38 4.4  4.42 4.44 4.46
 4.48 4.5  4.52 4.54 4.56 4.58 4.6  4.62 4.64 4.66 4.68 4.7  4.72 4.74
 4.76 4.78 4.8  4.82 4.84 4.86 4.88 4.9  4.92 4.94 4.96 4.98]
```

[9]:
```python
plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```



4 Scipy Special functions

A cool thing about Scipy is it's wide availability of definition for commonly used functions and polynomials, including *Jacobi*, *Legendre* or *Chebyshev* polynomials, *Bessel* functions and many more. A complete list is available at Scipy Special Functions Page.

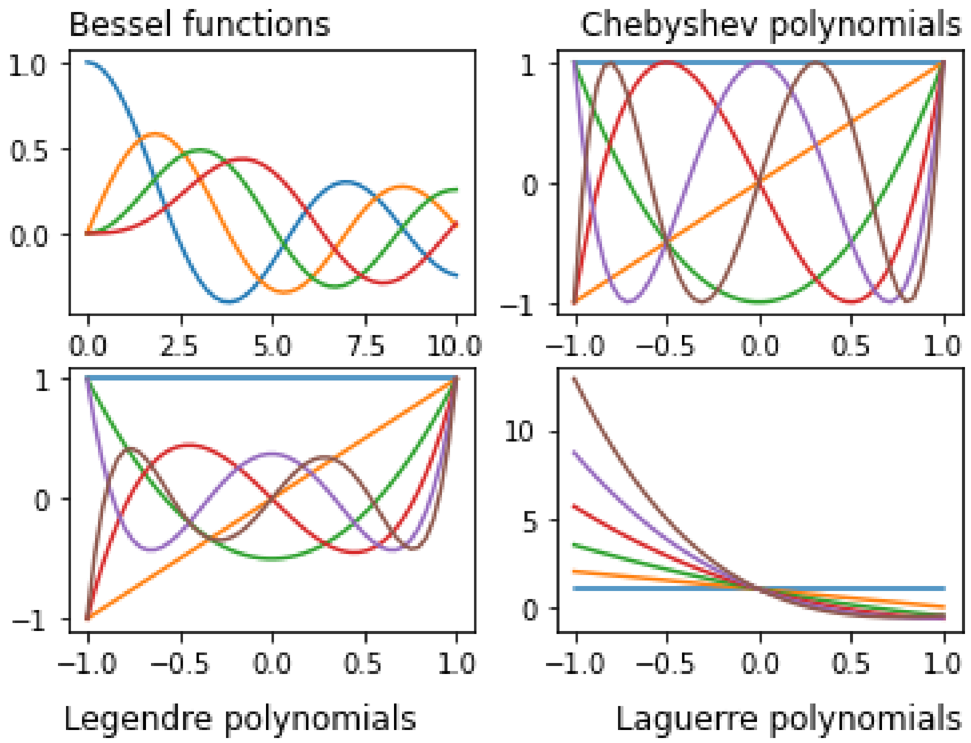
[10]:
```python
import scipy.special as Special
```

[11]:
```python
plt.subplot(2,2,1)
x = np.linspace(0,10)
for i in range(4):
    plt.plot(x,Special.jn(i,x))
plt.title("Bessel functions",loc="left")
plt.subplot(2,2,2)
x = np.linspace(-1,1)
for i in range(6):
    plt.plot(x,Special.eval_chebyt(i,x))
plt.title("Chebyshev polynomials",loc="right")

plt.subplot(2,2,3)
x = np.linspace(-1,1)
for i in range(6):
    plt.plot(x,Special.eval_legendre(i,x))
plt.figtext(0.12,0.,"Legendre polynomials",size="large")

plt.subplot(2,2,4)
x = np.linspace(-1,1)
for i in range(6):
    plt.plot(x,Special.eval_laguerre(i,x))
plt.figtext(0.60,0.,"Laguerre polynomials",size="large")
```

```
[11]: Text(0.6, 0.0, 'Laguerre polynomials')
```



5 Numerical derivatives

We have seen how manipulate array, compute statistical descriptors and extract and plot meaningful information. However, this is not enough if we want to test a physical model described by some (differential) equations. We may want, e.g., to compute the derivative of a function $f(x)$:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

if h is small but finite the equation above may be used as a *discrete approximation* of f':

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

if f is defined in the interval (a, b) we can test the approximation evaluating f' in that interval; since h must be small, we divide it in small bin, creating a *grid* (or *mesh*) of n *nodes* and n-1 *intervals* (this is what numpy.linspace does).

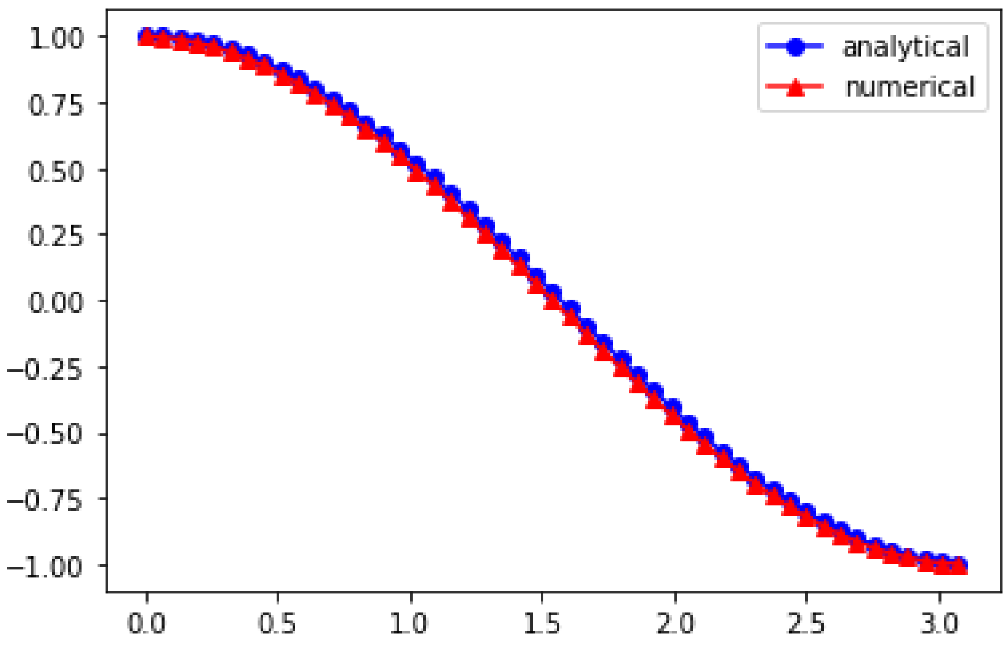
Let's try:

[12]:
```python
def fwd1(x,f):
    """
    Forward finite difference for f'
    """
    n = len(x)
    fprime = list()
    for i in range(n-1):
        fprime.append((f[i+1]-f[i])/(x[i+1]-x[i]))
    return np.asarray(fprime)
```

[13]:
```python
x = PI*np.linspace(0,1)
f = lambda x: np.sin(x)
fzero = f(x)
fprime = fwd1(x,fzero)
test = np.cos(x[:-1])
print(fzero.shape,fprime.shape)
```

```
(50,) (49,)
```

[14]:
```python
plt.plot(x[:-1],test,marker='o',color='b',label='analytical')
plt.plot(x[:-1],fprime,marker='^',color='r',label='numerical')
plt.legend()
plt.show()
```



So, the *forward finite difference* seems to work. However, using a for loop ain't very *numpythonic*. What about exploiting numpy arrays? Sadly, *numpy.vectorize* (read the documentation later on) provides for readble code, not speed. But for the fwd difference we use the **numpy.subtract** *binary universal function*:

[15]:
```
def fwd1_uf(x,f):
    """
    Forward finite difference for f'
    using numpy arrays
    assume h to be constant
    """
    n = len(x)
    h = x[1]-x[0]
    left  = f[:-1]
    right = f[1:]
    return np.subtract(right,left)/h
```

[16]:
```
fprime_uf = fwd1_uf(x,fzero)
fprime_test = np.abs(fprime - fprime_uf)
fprime_test = fprime_test[fprime_test>1e-5]
fprime_test
```

[16]:
```
array([], dtype=float64)
```

Which is faster?

```
[17]: x = PI*np.linspace(0,1,10000)
      fzero = f(x)
```

```
[18]: %timeit fwd1(x,fzero)
```

```
14 ms ± 33.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
[19]: %timeit fwd1_uf(x,fzero)
```

```
79.7 µs ± 58.6 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

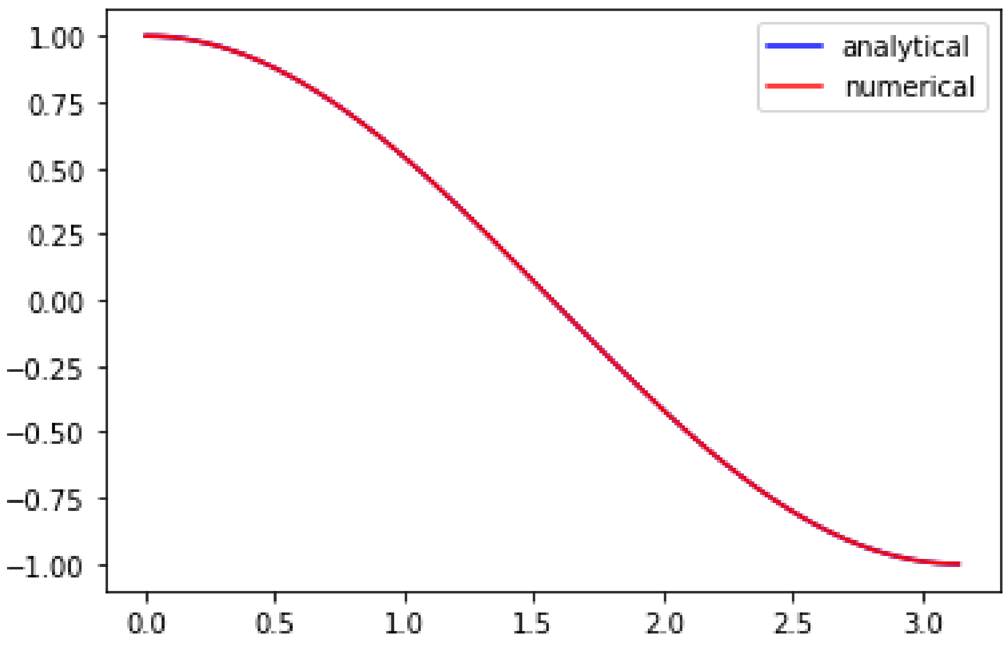
However, we are discarding the last point; we may also use also the *right finite difference*, just for the last point:

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

and have a more flexible numerical derivative:

```
[20]: def d1_uf(x,f):
          """
          Finite difference for f'
          using numpy arrays
          assume h to be constant
          """
          n = len(x)
          h = x[1]-x[0]
          left  = f[:-1]
          right = f[1:]
          last = np.array((right[-1]-right[-2]))
          return np.append(np.subtract(right,left),last)/h
```

```
[21]: fprime = d1_uf(x,fzero)
      plt.plot(x,np.cos(x),color='b',label='analytical')
      plt.plot(x,fprime,color='r',label='numerical')
      plt.legend()
      plt.show()
```



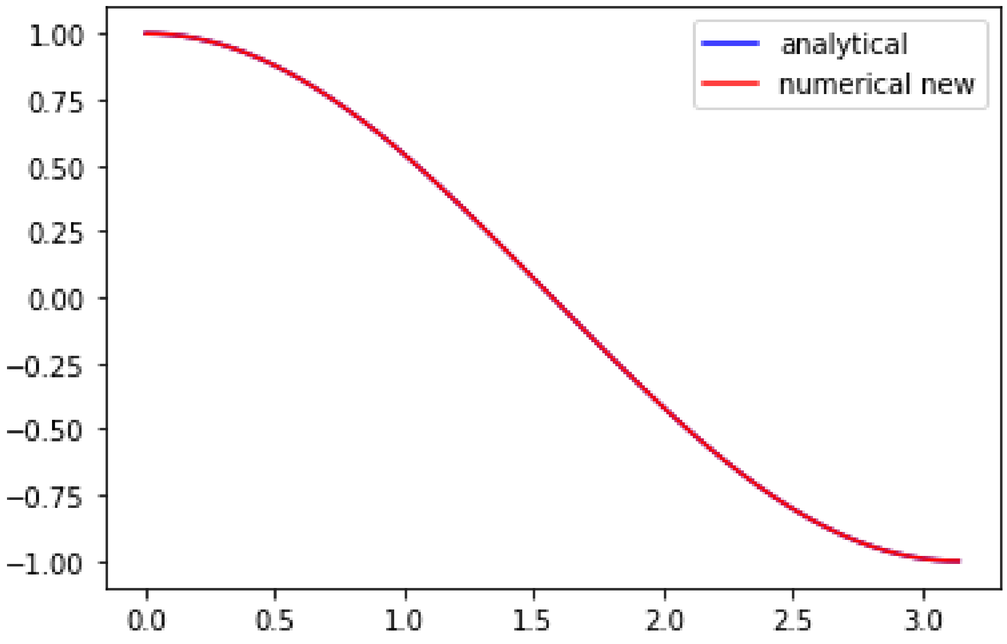
Using taylor expansion you can verify that the forwad and backward finite difference error is proportional to h. Further you can derive a *central finite difference* with an error proportional to h^2:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

so that we can combine the three methods in a unique flexible way:

[22]:
```python
def D1(x,f):
    """
    Finite difference first derivative
    """
    n = x.shape[0]
    h = x[1]-x[0]
    first = (f[1] - f[0])/h
    last  = (f[-1] - f[-2])/h
    left  = f[:-2]
    right = f[2:]
    central = np.subtract(right,left)/(2*h)
    return np.append(np.append(first,central),last)
```

[23]:
```python
fprime = D1(x,fzero)
fprime.shape
plt.plot(x,np.cos(x),color='b',label='analytical')
plt.plot(x,fprime,color='r',label='numerical new')
plt.legend()
plt.show()
```



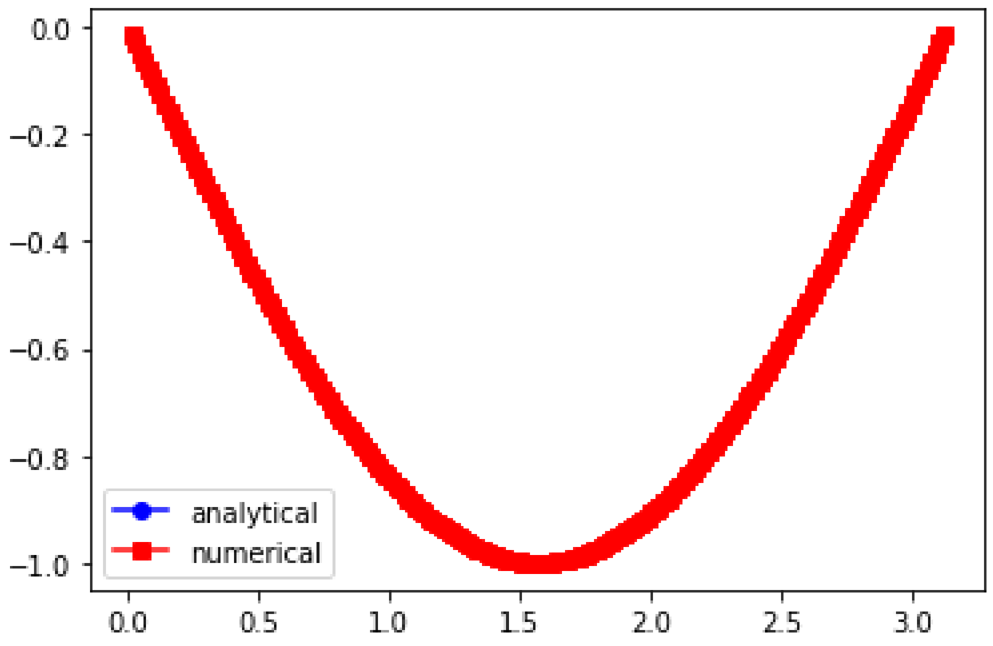
and, we can use Taylor expansion to compute a finite difference second derivative:

$$f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$$

[24]:
```
def D2(x,f):
    """
    Finite difference second derivative
    """
    n = x.shape[0]
    h = x[1]-x[0]
    left = np.subtract(f[2:],2*f[1:-1])
    fsec = np.add(left,f[:-2])/(h*h)
    return fsec
```

[25]:
```
x = PI*np.linspace(0,1,200)
f = lambda x: np.sin(x)
fzero = f(x)
test  = -fzero
fsec  = D2(x,fzero)
```

```
plt.plot(x[1:-1],test[1:-1],marker='o',color='b',label='analytical')
plt.plot(x[1:-1],fsec,marker='s',color='r',label='numerical')
plt.legend()
plt.show()
```



5.1 One-Dimensional Harmonic Oscillator using Finite Difference (from PyQuante)

Let's see if we can use Finite Differences this to solve the one-dimensional harmonic oscillator, i.e. want to solve the time-independent Schrodinger equation

$$-\frac{\hbar^2}{2m}\frac{\partial^2\psi(x)}{\partial x^2} + V(x)\psi(x) = E\psi(x)$$

for $\psi(x)$ when

$$V(x) = \frac{1}{2}m\omega^2x^2$$

is the harmonic oscillator potential. We're going to transform the differential equation into a matrix equation by multiplying both sides the complex conjugate $\psi^*(x)$ and integrating over x. This yields

$$-\frac{\hbar}{2m}\int\psi^*(x)\frac{\partial^2}{\partial x^2}\psi(x)dx + \int\psi^*(x)V(x)\psi(x)dx = E$$

We can think of the first term in the Schrodinger equation as the overlap of the wave function $\psi(x)$ with its second derivative. Given the finite difference expression for f'', we can see if we take the overlap of the states $y_1, \dots, y_n$ with the second derivative, we will only have three points where the overlap is nonzero, at y_{i-1}, y_i, and y_{i+1}. In matrix form, this leads to the tridiagonal Laplacian matrix, which has -2's along the diagonals, and 1's along the diagonals above and below the main diagonal.

The second term turns leads to a diagonal matrix with $V(x_i)$ on the diagonal elements. Putting all of these pieces together, we get:

[ ]:
```
def Laplacian(x,h):
    n = len(x)
    M = -2*np.identity(n,'d')
    for i in range(1,n):
        M[i,i-1] = M[i-1,i] = 1
    print(M[:,n/2-4:n/2+4][n/2-4:n/2+4,:])
    return M/h**2
```

[ ]:
```
x = np.linspace(-2,2)
m = 1.0
omega = 1.0
h = x[1]-x[0]
```

[ ]:
```
T = (-0.5/m)*Laplacian(x,h)
V = 0.5*(omega**2)*(x**2)
H =  T + np.diag(V)
E,U = LA.eigh(H)
```

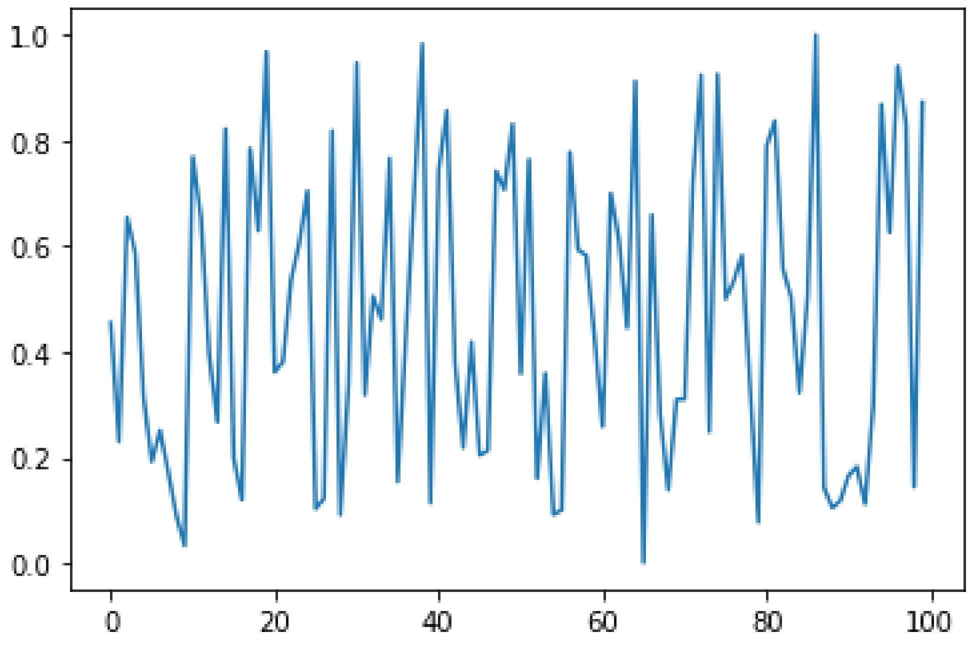
[ ]:
```
plt.plot(x,V,'k--',linewidth=2.0)
for i in range(4):
    # For each of the first few solutions, plot the energy level:
    plt.axhline(y=E[i],color='k',ls=":")
    # as well as the eigenfunction, displaced by the energy level so they don't
    # all pile up on each other
    #reflect the phase of U[:,0]
    plt.plot(x,-U[:,i]/math.sqrt(h)+E[i])
plt.title("Eigenfunctions of the Quantum Harmonic Oscillator")
plt.xlabel("Displacement (bohr)")
plt.ylabel("Energy (hartree)")
plt.show()
```

6 Monte Carlo, random numbers, and computing π

Section ?? integration: a sequence of (pseudo) random numbers are used to approximate the integral of a function. Python has good random number generators in the standard library. The **random()** function gives pseudorandom numbers uniformly distributed between 0 and 1:

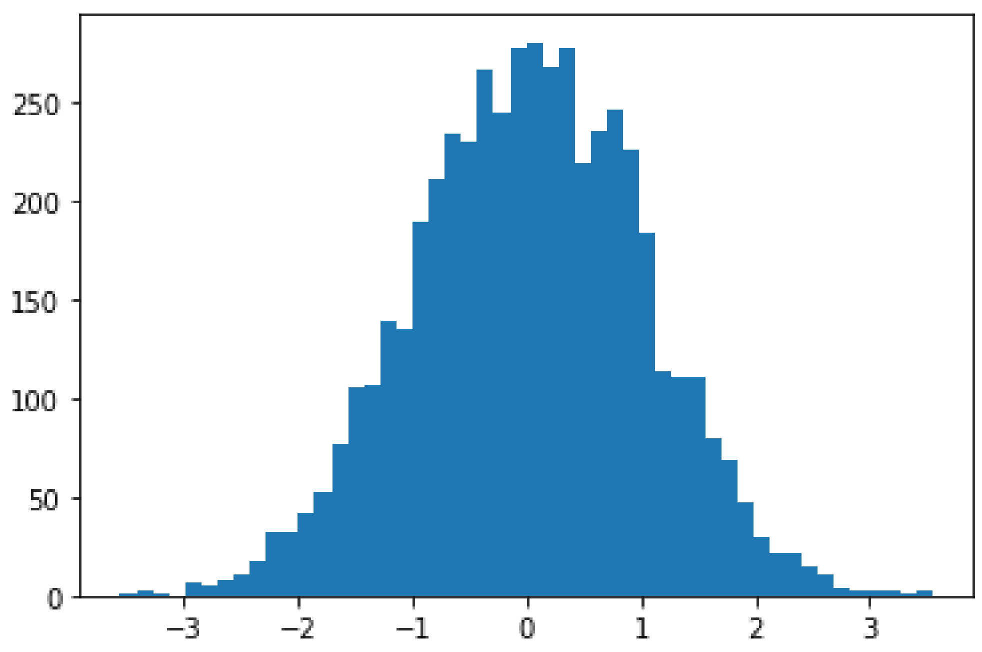
```
[26]: import random
      rands = [random.random() for i in range(100)]
      plt.plot(rands)
```

```
[26]: [<matplotlib.lines.Line2D at 0x7f1c8f469fa0>]
```



Looking at the documentation page you can read details about random implementation using the Mersenne Twister algorithm. There are also functions to generate random integers, to randomly shuffle a list, and functions to pick random numbers from a particular distribution, like the normal distribution:

```
[27]: gaussian_rands = [random.gauss(0,1) for i in range(5000)]
      hist, nodes,p = plt.hist(gaussian_rands,50)
```



Unsurprsingly, Numpy provides a function to generate an array of random numbers:

[28]:
```
np.random.rand(100)
```

[28]:
```
array([0.64927696, 0.36274726, 0.22279323, 0.37172376, 0.7027024 ,
       0.59299899, 0.1093975 , 0.08658675, 0.80563164, 0.43351056,
       0.71924413, 0.46298774, 0.69120794, 0.26205685, 0.6744396 ,
       0.2706521 , 0.46801865, 0.31978817, 0.80235089, 0.41043544,
       0.02921168, 0.4250264 , 0.96659513, 0.99449097, 0.97980147,
       0.01812078, 0.95126122, 0.10796657, 0.95948227, 0.62194687,
       0.39352254, 0.73158824, 0.15265736, 0.45646283, 0.67830372,
       0.31714336, 0.11740426, 0.06120548, 0.2388941 , 0.36429891,
       0.32394881, 0.8970562 , 0.32137448, 0.43585502, 0.02234761,
       0.03272666, 0.03940384, 0.82750721, 0.55098254, 0.94060723,
       0.48953392, 0.18927304, 0.21869452, 0.84402124, 0.40899559,
       0.53869991, 0.23616256, 0.83118443, 0.399579  , 0.80498429,
       0.37332146, 0.89799469, 0.04874984, 0.33139325, 0.62998133,
       0.85905746, 0.98143718, 0.2761416 , 0.70443412, 0.82161091,
       0.08742889, 0.79923023, 0.12342111, 0.43507197, 0.8546673 ,
       0.00241201, 0.96417687, 0.62882403, 0.4007365 , 0.66884428,
       0.21089854, 0.52156586, 0.43619803, 0.29195846, 0.62763309,
       0.71602395, 0.98810482, 0.37422105, 0.97105059, 0.08565808,
       0.9816108 , 0.07894548, 0.00165622, 0.92607719, 0.60062802,
       0.56042562, 0.34406459, 0.04859206, 0.27716165, 0.50173152])
```

A common example of MC integration is to compute π by sampling random numbers as x and y coordinates, and counting how many of them were in the unit circle. The basic idea is that the ratio of the area of the unit circle to the square that inscribes it is $\pi/4$, so by counting the fraction of the random points in the square that are inside the circle, we may estimate $\pi$.

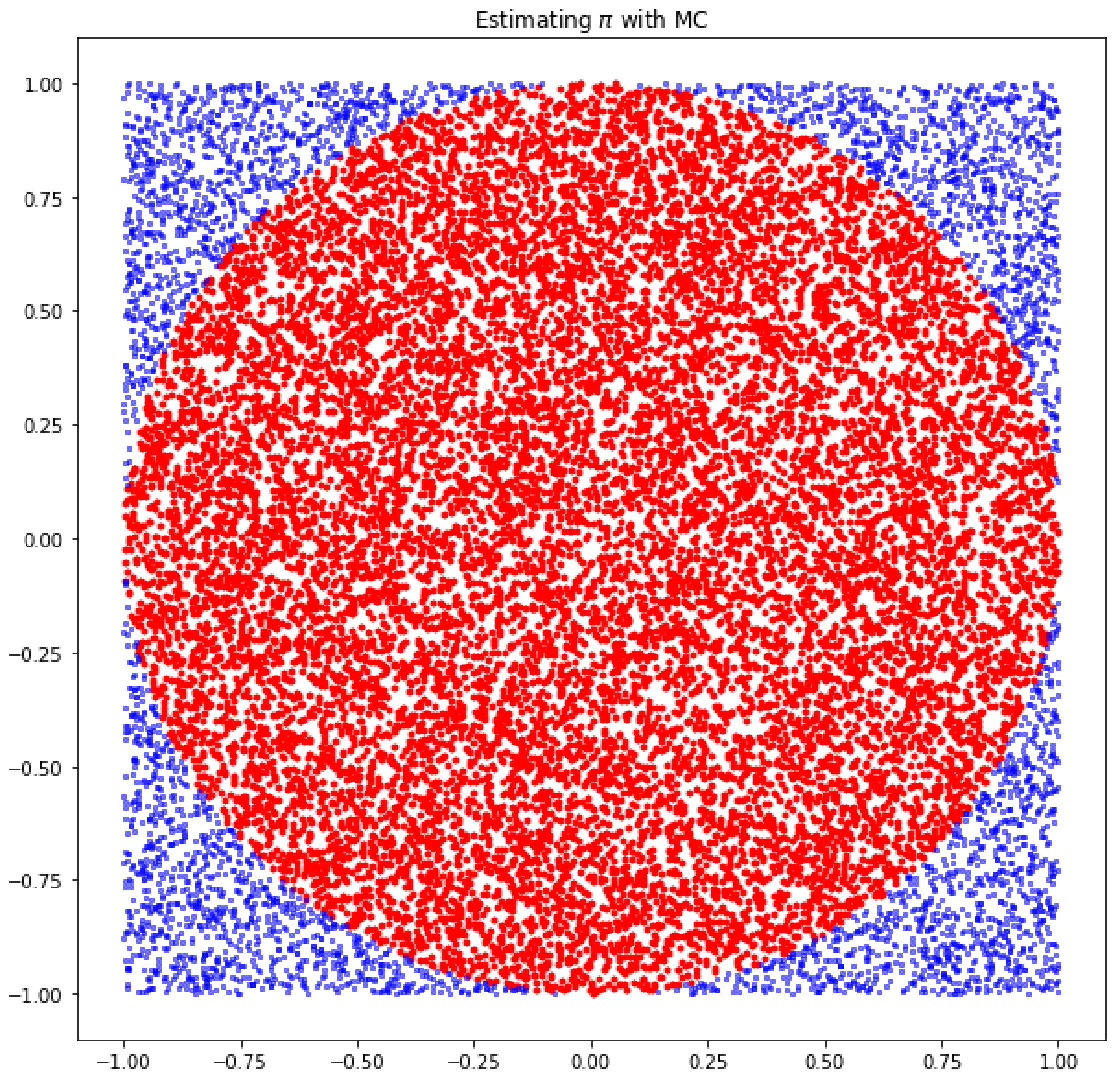
```
[29]: npoints = 20000
      #unit circle centered in .0, .0
      x = 2.*np.random.rand(npoints)-1
      y = 2.*np.random.rand(npoints)-1
      r = x**2+y**2
      #
      # fancy indexing
      inside = r[r<1.]
      print("# of points inside ",inside.shape)
```

```
# of points inside  (15717,)
```

Let's see how it works:

```
[30]: plt.figure(figsize=(10,10))
      plt.title("Estimating $\pi$ with MC")
      plt.scatter(x[r<1],y[r<1],marker='h',s=10,color='r')
      plt.scatter(x[r>1],y[r>1],marker='s',s=5,color='b',alpha=0.5)
```

```
[30]: <matplotlib.collections.PathCollection at 0x7f1c8f38fc40>
```



[31]:
```
4.*inside.shape[0]/npoints
```

[31]:
```
3.1434
```

compare it with Srinivasa Ramanujan fromula in notebook 01b