Vector Operations and Functions in Python

January 16, 2017

1 Python Assignment

1. Define $f(t) = 1/(1+t^2)$ as a function in *Python*. It should be able to take a vector argument.

2. Define a vector `x` that covers the region $0 \leq x \leq 5$ in steps of 0.1. (See `linspace`)

3. Plot $f(x)$ vs x using the Python function and the already defined vector x. Add a title like so:

   ```
   r"Plot of $1/(1+t^{2})$"
   ```

   and see what sort of title it creates. The expression within the "$" symbols is a LaTeX mathematical expression. For instance, Click on "view source" in the view menu and see how LyX translates the mathematical expression in question 1. You will see that it uses the same expression.

4. Create a "tan inverse" function from its integral definition, i.e., compute the function

   $$I(x) = \int_0^x \frac{dt}{1+t^2} = \int_0^x f(t)dt$$

   from 0 to x, accurate to 5 digits.

   (a) To carry out the integration, use the function `quad` in module `scipy.integrate`:

   ```
   from scipy.integrate import quad
   ```

   See `quad ?` for information on this function. To use it, you write

   ```
   quad(f,0,a)
   ```

   in order to compute the integral for $x = a$. Use a for loop to compute the vector of integrals.

   (b) Tabulate $\tan^{-1} x$ (see `arctan`) and the above numerical integral values for these values of `x`. This function is available in the `special` module. Do not use a for loop. Just apply tan to the vector `x`.

   (c) Plot the integral values as red circles and $\tan^{-1} x$ as a solid black line (versus x) in Figure 1.

This is a convention you should always follow. Theory has a value for every value of the independent variable. So it is a solid line. Observations or calculations yield values for certain values of the independent variable. They should be shown as points.

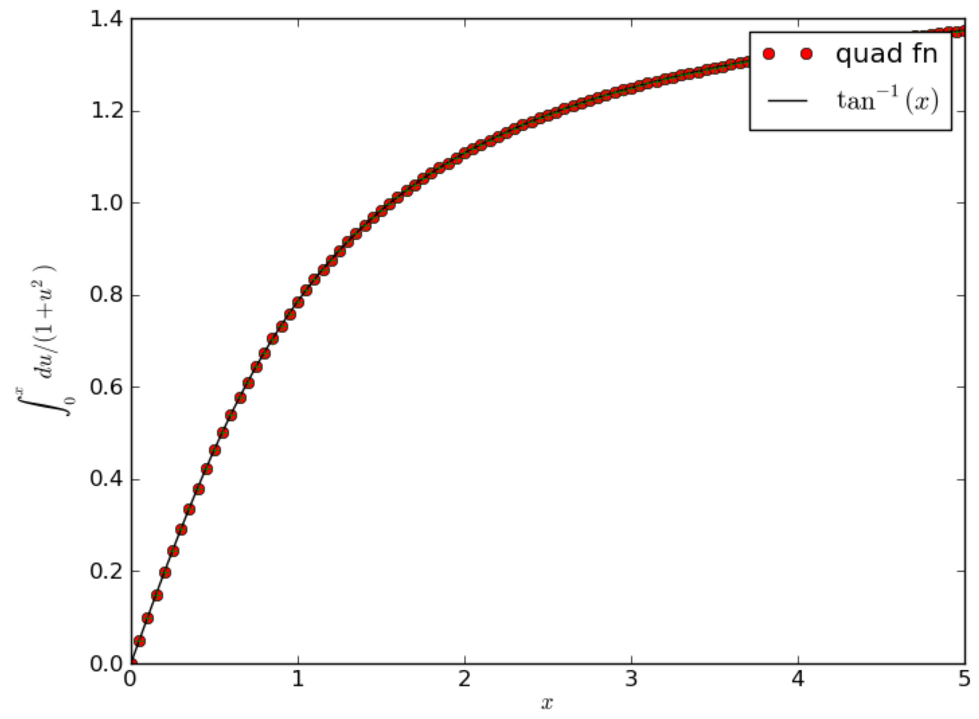
Add a legend to indicate which curve corresponds to what. The command for that is

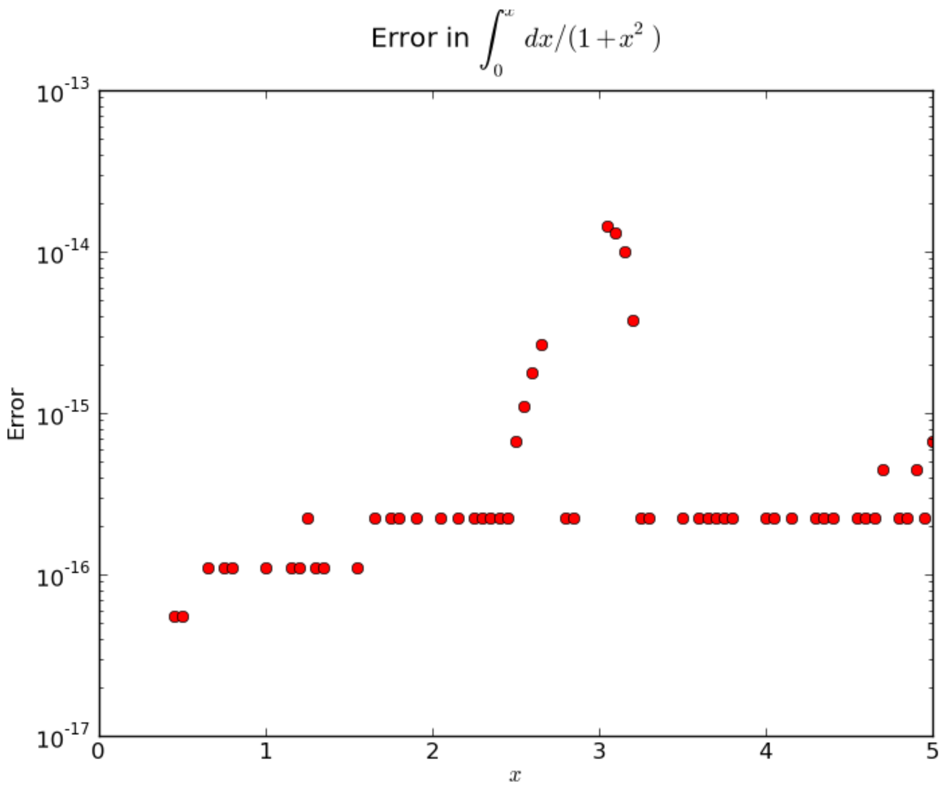
```
legend( (str1,str2) )
```

where the strings are the labels for each curve. As above, use expressions enclosed between “$” characters to produce a math expression.

Simple technique: Create a Lyx math expression containing what you want in the label, and view the source for that expression. The same source expression will do the job in Python. For example, if you want a label that says $e^{x^2/2}$, just add `r"$e^{x^{2}/2}$"` as the string.

(d) Plot the error of the integral method in Figure 2 in a semi log plot (see `semilogy` function). Label the plots and the axes appropriately. You should have something like the following plots:





5. Now we develop an algorithm to compute the integral ourselves. The approach is to compute the Trapezoidal algorithm for a running integral.

   According to the Trapezoidal Rule, if a function is known at points a, $a+h$, ..., b, its integral is given by

$$I = \begin{cases} 0 & x=a \\ 0.5\left(f(a)+f(x_i)\right)+\sum_{j=2}^{i-1} f(x_j) \quad , & x=a+ih \end{cases}$$

   This can be rewritten as

$$I_i = h\left(\sum_{j=1}^{i} f(x_j) - \frac{1}{2}\left(f(x_1)+f(x_i)\right)\right)$$

   The advantage of this is that previous work can be reused to compute successive integrals. Code this in Python, and then figure out how to estimate error and determine the "h" to use.

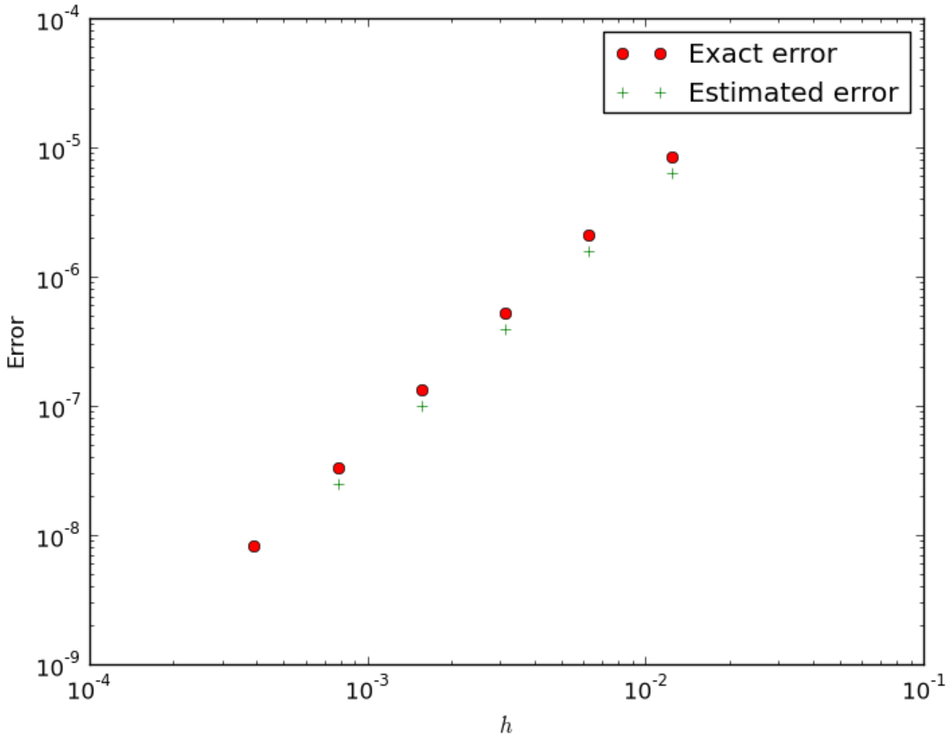
   (a) Use a for loop and implement the integrals.

   (b) Use vector operations and compute the running integral `I` given a vector of values. **Hint**: Look at the `cumsum` function. Your basic implementation should be a single line of code! Add this plot to Figure 0.

(c) The error is defined as the largest error for common points. For instance if the first integral was for $x = 0, 0.5, 1$ and the second time the integral was for $x = 0, 0.25, 0.5, 0.75, 1$, clearly we can compare the integral values at $x = 0, 0.5, 1$. Keep halving h till error falls below the desired tolerance. Apply this to the integral in Q 1 and compare to the `atan` function to see if the error is correctly calculated. Plot both the error according to this part and the actual error vs x.

Note: For this problem, assume that the desired tolerance is 10^{-8}.

Note: You should create the arrays to hold h and error ahead of time. Assume that a maximum of nine halving steps will happen (and write the `while` loop accordingly). Note that there are two errors - the true error and the "estimated" error.



Here is what you should obtain. Note that the estimated error (i.e., the maximum error on common points) is not equal to the actual error. This is an important thing to realise - numerical methods are always a little uncertain. Only in special cases do we know the exact error. With a function whose integral is not available, we have to depend on the estimated error to know when to stop.

$$\boxed{\text{No algorithm is worth anything without an error estimate.}}$$

6. Once you have the code working, make it part of a LyX report. The plots and table should be part of the report. And ofcourse the code should be in the “scrap” environment so that the Python file can be extracted and run.

Submit the lyx file to the Moodle site. Please note that the plots will only be linked to the LyX file and will not be a part of the file as is the case in a Word document. So you will have to submit the plot files as well. To do this, create the files in a subdirectory, zip the files and submit the zip file to Moodle.

1.1 Introduction to Scientific Python

Python has some packages (called modules) that make Python very suitable for scientific use. These are `numpy`, `scipy` and `matplotlib.` `numpy` makes available numerical (and other useful) routines for Python. `scipy` adds special functions and complex number handling for all functions. `matplotlib` adds sophisticated plotting capabilities. All of these, combined together, are in the `pylab` module.

Here is a simple program to plot $J_0(x)$ for $0 < x < 10$. (type it in and see)

```
In [48]: from pylab import *
In [49]: x=arange(0,10,.1)
In [50]: y=jv(0,x)
In [51]: plot(x,y)
In [52]: show()
```

The `import` keyword imports a module as discussed above. The “pylab” module is a super module that imports everything needed to make python seem to be like Matlab.

The actual code is four lines. One defines the x values. The second computes the Bessel function. The third plots the curve while the last line displays the graphic.

1.2 Plotting

Plotting is very simple. Include the plot command and at the end of the script add a `show()`. The plot command has the following syntax:

```
plot(x1,y1,style1,x2,y2,style2,...)
```

just as in Matlab.

1.3 Array operations

With these modules, we can now create arrays of greater complexity and manipulate them. For example,

```
In [56]: from pylab import *
In [57]: A=[[1,2,3],[4,5,6],[7,8,9]]
In [58]: print A
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
In [59]: A=array([[1,2,3],[4,5,6],[7,8,9]]);A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
In [60]: B=ones((3,3));B
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
In [61]: C=A*B;C
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
In [62]: D=dot(A,B);D
array([[  6.,  6.,  6.],
       [ 15., 15., 15.],
       [ 24., 24., 24.]])
```

As can be seen in the examples above, operations on arrays are carried out element by element. If we meant matrix multiplication instead, we have to use `dot` instead. This is a problem that is unavoidable since these modules are built on top of python, which does not permit "dot" operators.

`Numpy` and `scipy` also define "matrix" objects for which "*" means matrix multiplication. However, I think it is not worth it. Just use the array objects.

Some important things to know about arrays:

- Array elements are all of one type, unlike lists. This is precisely to improve the speed of computation.
- An array of integers is different from an array of reals or an array of doubles. So you can also use the second argument to create an array of the correct type. Eg:

  ```
  x=array([[1,2],[3,4]]),dtype=complex)
  ```

- Arrays are stored row wize by default. This can be changed by setting some arguments in numpy functions. This storage is consistent with C.
- The `size` and `shape` methods give information about arrays. In above examples,

  ```
  D.size   # returns 9
  D.shape  # returns (3, 3)
  len(D)   # returns 3
  ```

  So `size` gives the number of elements in the array. `Shape` gives the dimensions while `len` gives only the number of rows.
- Arrays can be more than two dimensional. This is a big advantage over Matlab and its tribe. Here arrays are intrinsically multi-dimensional.

- The `dot` operator does tensor contraction. The sum is over the last dimension of the first argument and the first dimension of the second argument. In the case of matrices and vectors, this is exactly matrix multiplication.

1.4 Finding elements

Sometimes we want to know the indices in a matrix that satisfy some condition. The method to do that in Python is to use the `where` command. To find the even elements in the above matrix we can do

```
from pylab import *
A=array([[1,2,3],[4,5,6],[7,8,9]]);A
i,j=where(A%2==0)   # returns the coords of even elements
print A[i,j]
```

The ouptut is `[[2 4 6 8]]` which has no memory of the shape of A, but does contain the values. Note that the row and column indices start with zero, not one.

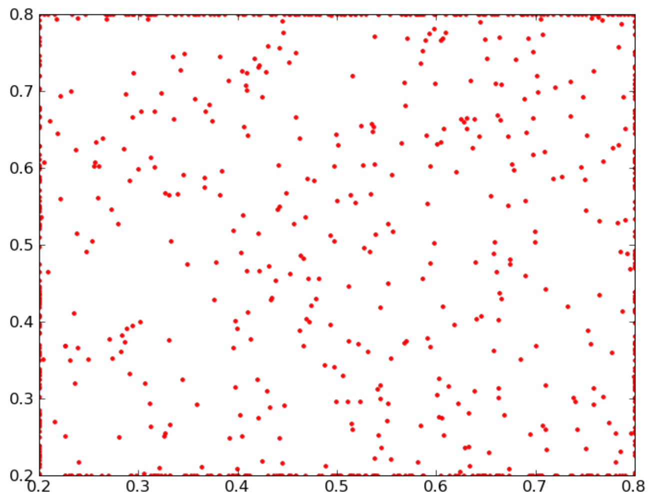
```
B=array([[6,6,6],[4,4,4],[2,2,2]])
i,j=where((A>3) & (B<5))
```

Here we have a new trick. We want to know those elements of A and B where the element of A is greater than 3 while the corresponding element of B is less than 5.

Python has two sets of logical operators. 'And', 'or', and 'not' are boolean operators that work on boolean values. Unfortunately, here (A>3) is a boolean matrix, and 'and' has no idea what to do with it. Python also has '&', '|' and '~' which are the same logical operators, but these work element by element. These are what you should use with matrices to combine compound conditions.

How do we use the `where` command? Suppose we want to clip values to between 0.2 and 0.8. We execute the following code:

```
A=random.rand(1000,2)
i,j=where(A<0.2)
A[i,j]=0.2
i,j=where(A>0.8)
A[i,j]=0.8
plot(A[:,0],A[:,1],'r.')
show()
```

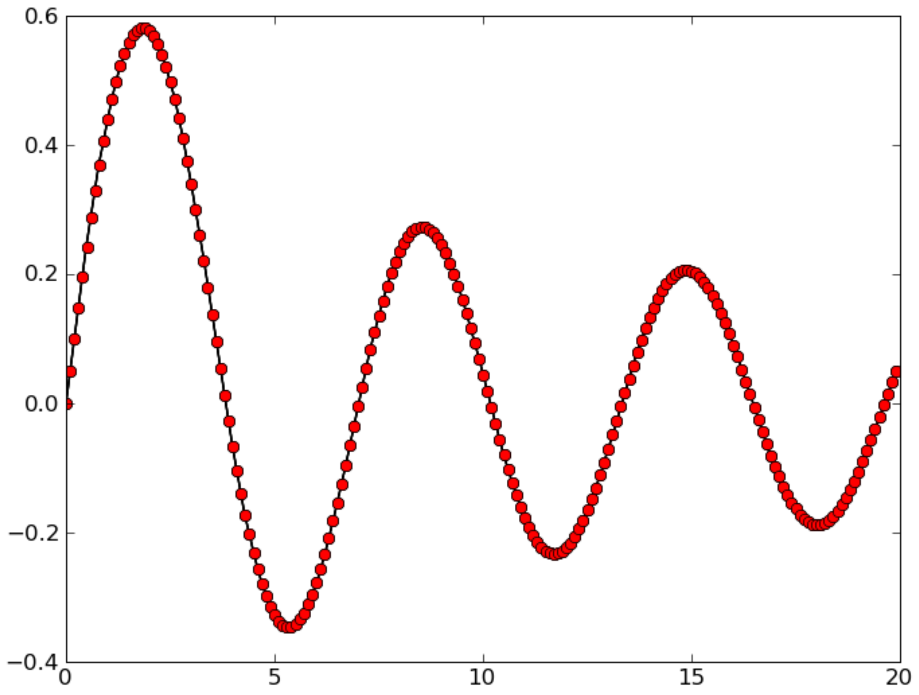


As can be seen, the points are all clipped to between 0.2 and 0.8.

1.5 Simple File I/O with Numpy

There are two simple functions that do most of what we want with numbers, namely `loadtxt()` and `savetxt()`. Both are in the `numpy` module. I used one of them to create the `plot.py` script. Here is a simple way to use them. (I will drop the python prompt from here one. That way you can cut and paste more easily)

```
from scipy import *
import scipy.special as sp
from matplotlib.pyplot import *
x=arange(0,20,.1)    # x values
y=sp.jv(1,x)            # y values
plot(x,y,'k')
savetxt('test.dat',(x,y),delimiter=' ')
w,z=loadtxt('test.dat')
plot(w,z,'ro')
show()
```



As can be seen, the red dots (the plot of w vs z) lie on the black line (the plot x vs y). So the reading and writing worked correctly.

`savetxt` simply writes out vectors and arrays. In the above example, if you look at `test.dat`, you will see two rows of data each with 200 columns. `loadtxt` reads in the objects in the file. Again, in this case, it reads in a 2 by 200 matrix, which is assigned to two row vectors.

In the assignment of this week, we directly use file i/o to access the contents of a file. That shows the power of Python that is available at any time.

Certain things which are very intuitive in matlab like languages are less easy in Python. To stack two vectors as columns we use

```
A=[x,y]
```

in matlab, provided the vectors are column vectors. In python we must write

```
A=c_[x,y]
```

since `[x,y]` defines a list with two elements and not a matrix. The function `c_[...]` takes the arguments and stacks them as columns to form a numpy array.

Computation Speed in Python

Last time I said that scientific python was invented to make fast computing possible. And I said that “arrays” were *n*-dimensional matrices of numbers that were all of a single type so that calculation is faster. What these statements all mean is that with scientific python modules added, python is about as fast as matlab (but not really - matlab is still faster).

There remains the question - why use vectors at all, in Python or Matlab? For instance, suppose I wanted to generate a lakh of random numbers uniformly distributed between 0 and 1, and find those whose value is greater than 0.999. The following code could do it:

```
# import modules
from pylab import *
import time as t
# time the calculations with for loop
t1=t.time()
n=0
w=zeros(200) # we don’t expect to see more than 200 randoms
for i in range(100000):
  x=rand(1)
  if x>0.999:
    w[n]=x
    n+=1
t2=t.time()
w=w[0:n]
print w
print "the program took %.3f seconds to find %d randoms" % (t2-t1,len(w))
```

On a particular machine I tried this code, it took about 0.55 seconds. Now let us write the same code with vector operations and time the same.

```
# import modules
from scipy import *
from matplotlib.pylab import *
import time as t
# time the block
t3=t.time()
x=rand(100000)
ii=where(x>0.999) # find locations where x>0.999
w=x[ii] # create new vector with only those values
print w
t4=t.time()
print "the program took %.3f seconds to find %d randoms" % (t4-t3,len(w))
```

This code took between 0.005 and 0.006 seconds to run. A speed up factor of 100.

$\boxed{\text{Vectorizing code is worth it!}}$