Introduction to Tools - Python

August 6, 2014

1 The Python Language

When it comes to programming languages, there has been a long history of both compiler based languages and interpreter based languages. On the compiler side there was Fortran followed by Cobol and after quite a while, there was Pascal and then finally C. With each new language new features were added. Ofcourse the older languages did not stay put and so there is a Fortran2003 which has many of the features of C++ that are relavent to scientific programming.

Similarly, on the interpreter side, there were early attempts which included Flex and Basic, as well as the language of the command shell. When UNIX came on the scene, the shell came wtih a sophisticated programming language. But UNIX was special in that it added special power to this shell via other utilities such as awk, sed, yacc etc. Awk was the first text processor. Sed was a stream editor, that filtered a text file to give another. And Yacc was a build on the run compiler generator. These were utilities of power never before seen by casual users, and many inventive uses were found for them. Around the same time, on the IBM side, an interesting language called REXX also appeared which first invented one of the most important modern data structures, namely associative arrays.

With so many utilities and options things were a jumble and soon users began to demand for a more integrated language that had all these features built in. This ushered in the modern set of languages of which the premier one is PERL. PERL introduced many things into interpreted languages, so many that it is hard to enumerate them. But it tried to do too much and it is a sprawling language, one in which an expert can do magic, but one which requires quite a lot of studying to master.

Following PERL were some competitors of whom I consider Ruby and Python the best. Both are lovely languages from the user's point of view and both are very similar in their features. Ruby has developed into a terrific language for database programming. Python has become the primier language for science and engineering. This is why we will focus on Python in this course.

A third direction in which languages developed was the scratchpad type of language. It started in the sixties with early versions such as the "Online Math System" of UCLA. In the seventies, a powerful scientific development package called Basis was introduced. In the early eighties IBM had introduced scientific spreadsheets for laboratory use. This branch has led to modern tools such as Matlab, Mathematica and Labview.

In this lab we will study programming with Python. The particular focus will be, just as with Scilab, how to link C programs into Python.

2 Basic Features of Python

To start the interpreter, just type "python" at the command line. You can then enter commands at the prompt. For example

```
$ python
Python 2.5.2 (r252:60911, Sep 30 2008, 15:41:38)  [GCC 4.3.2 20080917 (Red Hat 4.3.
>>> a=range(10);a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> b=''This is a string'';b
```

```
'This is a string'
>>> a[2:5]
[2, 3, 4]
>>> quit()
$
```

The Python prompt is ">>> ". This short excerpt shows how to define a couple of variables and to quit the interpreter.

2.1 Basic Types

Python recognises a variety of types but doesn't require the user to declare any of them. They include:

- Integers
- floats
- complex numbers
- boolean values
- strings
- "tuples" and lists (these are arrays)
- dictionaries (these are associative arrays)

Pointers don't exist, as all memory management is handled invisibly. Python has the ability to define other types as well, such as matrices through the use of classes.

2.1.1 Examples

```
>>> a=4.5
```

This is an example of a real number.

```
>>> i=3
```

This is how we declare integers. The only thing to remember is that integer division can yield unexpected results:

```
>>> i=3;j=2;print i/j
1
```

The answer is not 1.5 but its truncated value. Also note that $3/(-2) = -2$! In Python version 3 onwards, integer division follows C syntax and returns the floating point answer.

```
>>> z=1+2j
```

Python handles complex numbers as native types. This is enormously useful to scientists and engineers which is one reason why it is popular with them. Note that "j" is identified as $\sqrt{-1}$ by context. So the following code will give peculiar answers:

```
>>> j=-2
>>> 1+2j
(1+2j)
>>> 1+j
-1
```

In the first complex assignment, Python knew that you meant a complex number. But in the second case it thought you meant the addition of 1 and the variable `j`. Also note that if you assign the answer to a variable Python does not print the answer. But if you do not assign it, the value is printed out.

```
>>> v=True
>>> j>2 or v
True
```

`True` and `False` (capitalized as shown) are the fundamental boolean values. They can be assigned to variables and they are also the result of logical expressions.

```
>>> ''This is the %dth sentence.'' % 4
'This is the 4th sentence.'
```

This is a most peculiar expression. We have seen strings already. But *any* string can be a `printf` string that follows C syntax. This must be followed by a % sign and a list of values.

```
>>> ``This is the %(n)dth %(type)s.'' % {'type':'sentence','n':4}
```

A string can be more peculiar. It can take variable names for its entries so that they are represented by labels. Note that this is the general string, and can be used anywhere, not just where printing is going to happen.

```
>>> ``"""This is a multi
line string"""''
'This is a multi\nline string'
>>> print _
This is a multi
line string
```

Sometimes we might want embedded new lines in strings. Then the above syntax helps us. Everything between the triple quotes is part of the string. Also note the print command. It prints "_" which is Python's variable that holds the last computed value that was not assigned to a variable.

Naturally strings are where a lot of built in functions exist and we will look at them soon.

2.1.2 Arrays

Arrays store items in order so that they can be accessed via their index. The indexing follows the C indexing (i.e., starts from 0). Unlike C, the elements of a list do not have to be identical in type.

```
>>> arr=[1,2,3,4,5];arr
[1, 2, 3, 4, 5]
```

Arrays can be created by just specifying the elements as in the first example above. The "constructor" is a set of comma separated values enclosed in square brackets.

```
>>> arr=range(10);arr
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> arr=range(2,5);arr
[2, 3, 4]
>>> range(2.2,10.7,1.5)
[2, 3, 4, 5, 6, 7, 8, 9]
```

A common use for arrays is to create a sequence of uniformly spaced points. This can be done with the `range` command. Without a second argument, it returns a sequence of integers starting with zero upto the given argument (but not including it). With two arguments, the sequence starts at the first argument and goes to the second. Given three arguments, the third argument is the step. **Note that range expects integer arguments.** This is seen in the last example above. When floats are given, they are "coerced" (i.e. converted) into integer and then sent to the function.

```
>>> arr[2:7:2]
[2, 4, 6]
```

We can refer to a portion of an array, using what is known as a "slice". `2:7:2` means "start with 2, increment by 2 upto but not including 7".

```
>>> arr=[1,2,[4,5,6],''This is a string'']
>>> arr[0]
1
>>> arr[2]
[4, 5, 6]
>>> arr[3]
'This is a string'
```

Arrays do not need to have elements of a single type. In the example above, we have mixed integers with another list and with a string.

```
>>> arr[2][1]
5
```

This shows the way to access the element of a list that is an element of a second list. The syntax is very similar to 2-dimensional arrays in C but clearly pointers are involved everywhere, but invisibly.

```
>>> hash={''a'':''argentina'',''b'':''bolivia'',''c'':[1,2,3]}
>>> hash[0]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 0
>>> hash["a"]
'argentina'
>>> hash["c"]
[1, 2, 3]
```

The first attempt `hash[0]` shows that this is a new beast. Its elements are not accessed by an index. The error message says that "0" is not a key found in the dictionary. A "dictionary" consists of a set of "key-value" pairs. Each pair has a string called the key and some value. The value could be another string or it could be anything else. In the above example it is a list of three elements. Dictionaries were (to my knowledge) first introduced in Rexx and perfected in Perl are invaluable in modern programming.

There is another type of list in Python called the "tuple". It merely means a constant list, i.e. a list whose values cannot change. It follows the same syntax except its values are enclosed in round brackets.

2.2 Basic Programming

The basic structures in Python are the if block, the for block and functions. There are others, but these are the important ones.

```
>>> if j>2 or v:
...     print ''Too many inputs: %d'' % j
...
Too many inputs: -2
>>> sum=0
```

Note that blocks start with a statement ending in a colon. The code in the block must be indented evenly - the extent of indentation is how python knows which block the statement belongs to. The above code is the standard `if` block. The general structure is

```
if condition:
    commands
elif condition:
    commands
elif condition:
    commands
...
else:
    commands
```

There is no switch command in Python, and this construction has to do the job of the select - case statement as well.

```
>>> for i in range(20):
...     sum += i
...
>>> print ‘‘The sum of 0, 1, ... 19 = %d’’ % sum
The sum of 0, 1, ... 19 = 190
```

The for block in Python iterates over an array of values. Here it is iterating over a set of integers.

```
>>> words=[‘‘Water’’, ‘‘water’’, ‘‘everywhere’’, ‘‘but’’, ‘‘not’’, ‘‘a’’, ‘‘drop’’, ‘‘to’’, ‘‘drink’’
>>> for w in words:
...     print ‘‘%s’’ % w,
...
Water water everywhere but not a drop to drink
```

This for block prints out all the values of array words. Note that the print command ends in a comma, which replaces the carriage return with a single space.

```
>>> sum=0;i=0
>>> while i<20:
...     i += 1
...     if i%3 == 0: continue
...     sum += i
...
>>> print ‘‘The sum = %d’’ % sum
The sum = 147
```

The while construct is self-evident. It iterates till the condition fails. The if command skips to the next iteration every time a multiple of 3 is encountered. Also note that the “continue” statement can be placed on the if line itself if desired. This is not actually recommended though.

```
>>> def factorial(n=0):
...     ‘‘’’’ Computes the factorial of an integer ‘‘’’’
...     fact=1
...     for i in range(1,n+1):
...         fact *= i
...     return(fact)
...
>>> factorial(8)
40320
>>> factorial()
1
>>> factorial(4.5)
```

```
__main__:4: integer argument expected, got float
>>> factorial.__doc__
' Computes the factorial of an integer '
```

Functions are defined using blocks as well. Arguments are like C and can have "default" values. Here the default value of `n` is zero, which is the value assumed for `n` if no argument is passed. If a float is passed or a complex, an error is generated. This is not an argument mismatch error - python is very forgiving about those. Rather it is saying that `range(1,n+1)` does not make sense unless `n` is an integer.

The final line shows the self-documenting nature of Python functions. Anything entered as a string at the beginning of a function is retained as documentation for the function. The help string for a function can always be retrieved through the "`__doc__`" method. For example,

```
>>> import os
>>> print os.system.__doc__
system(command) -> exit_status

Execute the command (a string) in a subshell.
```

2.3 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.

Here is a simple program to plot $J_0(x)$ for $0 < x < 10$.

```
>>> import numpy as np
>>> import scipy.special as sp
>>> from matplotlib.pyplot import *
>>> x=np.arange(0,10,.1)
>>> y=sp.jv(0,x)
>>> plot(x,y)
>>> show()
```

The `import` keyword imports a module. The "`as np`" bit gives an alias. In the case of the plotting routines we could have had the following line instead:

```
>>> import matplotlib.pyplot as plt
```

But we don't want to keep saying `plt.plot()` etc. So we import those functions into our local space so that all those functions can be called directly. That is what the `from ... import ...` statement does.

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.

2.4 Help on scientific functions

The numpy library has an "info" command that gives information on the usage of built in commands. For example, to know how to use `arange`, we type

```
>>> info(arange)
```

to get three pages of information on its usage, complete with examples. As another example,

```
>>> info(sp.jv)
y = jv(x1,x2)
y=jv(v,z) returns the Bessel function of real order v at complex z.
```

tells us about how to use the bessel function available in `scipy.special`.

Note that info also returns the standard help on python built in functions. For example, we can get information on the `range` command:

```
>>> info(range)
range([start,] stop[, step]) -> list of integers
Return a list containing an arithmetic progression of integers. range(i, j) returns
```

2.5 Array operations

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

```
>>> from scipy import *
>>> A=[[1,2,3],[4,5,6],[7,8,9]];A
>>> print A
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> A=array([[1,2,3],[4,5,6],[7,8,9]]);A
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> B=ones((3,3));B
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
>>> C=A*B;C
array([[1., 2., 3.],
       [4., 5., 6.],
       [7., 8., 9.]])
>>> 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. Scilab has hypermatrices, but those are slow. 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.

2.6 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 scipy 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]
```

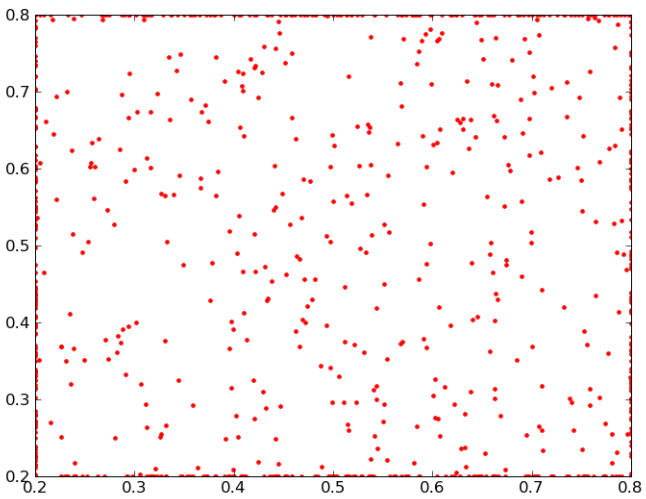
The ouptut is `[[2 4 6 8]]` which has no memory of the shape of A, but does contain the values. Note that the row and column indices start with zero, not one.

```
B=array([[6,6,6],[4,4,4],[2,2,2]])
i,j=where((A>3)*(B<5)>0)
```

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. In Matlab (or Scilab) we would say `find(A>4 and B<5)`. However, Python does not like this construction since `A>4` is not well defined. Are we to take all the truth values, and them all together and return a single true or false? Or something else? So Python does not permit such an operation to be part of a logical statement. However, we `can` do element by element operations. which is what we are doing here.

How do we use the `where` command? Exactly as in Scilab. Suppose we want to clip all 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()
```

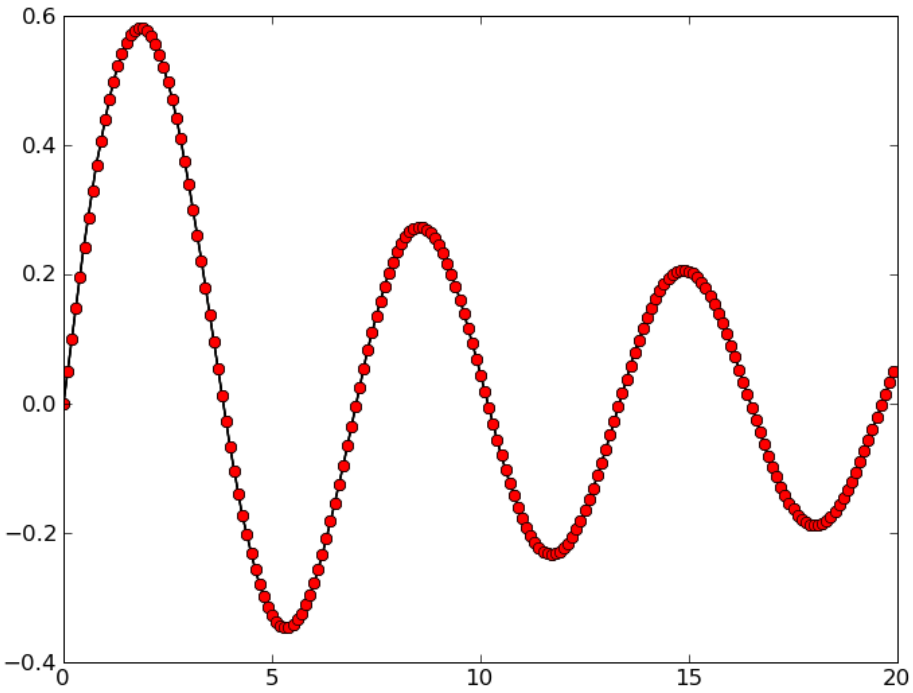


As can be seen, the points are all clipped to between 0.2 and 0.8.

2.7 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.

Certain things which are very intuitive in matlab like languages are very clumsy in Python. For instance, to construct a 200 by 2 matrix out of `x` and `y` we need to say

```
A=hstack([vstack(x),vstack(y)])
```

since the natural way for Python to think is to put everything in a row. `hstack` and `vstack` are directives for how to line up the vectors when they are combined. Python does not seem to distinguish between column and row vectors.

3 Python with inline C

The purpose of this code is to find the value of a function given its fourier coefficients. The coefficients are $1/(n^2+a^2)$ and it is a cosine series.

We first load the python packages.

10a ⟨* 10a⟩≡ 10b▷

```
from scipy import *
from matplotlib.pyplot import *
import time
import scipy.weave as weave
```

Define the parameter a and the limit of the summation N.

10b ⟨* 10a⟩+≡ ◁10a 10c▷

```
a=0.5
x=arange(0,3,.1)
# we sum till the Nth term is 1e-4 less than the first.
N=int(max(sqrt(9999)*a,10))
```

The series to be calculated is

$$f(x) = \sum_{k=0}^{N} \frac{\cos kx}{a^2+k^2}$$

We can either compute it directly (a bad idea unless you have a really excellent implementation for calculating $\cos kx$) or we can use Clenshaw's recurrence formula (NR p 176). This formula is based on the existence of a recurrence formula for the cosine function:

$$\cos(nx) = 2\cos x\cos((n-1)x) - \cos((n-2)x)$$

with $\cos(0x) = 1$ and $\cos(1x) = \cos x$. Clenshaw's algorithm says that the *function* can be calculated without ever calculating the cosines. In addition, it says that it is a stable calculation even if the recursion above is a "bad" one (you will understand this later when we get to function fitting.

Initialization: $y_{N+2} = y_{N+1} = 0$

Recursion: $y_k = \alpha y_{k+1} + \beta y_{k+2} + c_k$, where $\alpha = 2\cos x$, $\beta = -1$ and $k = N, N-1, \ldots, 1$

Final: $f(x) = \beta y_2 + y_1 \cos x + c_0$

As you can see, the cosine harmonics are never actually calculated. But the sum is computed through this single recursion.

We implement the recursion as a set of vector operations.

10c ⟨* 10a⟩+≡ ◁10b 10d▷

```
def clenshaw(N,c,x):
    y=zeros((N+3,len(x)))
    F1=cos(x)
    alpha=2*F1
    beta=-ones(x.shape)
    for k in range(N,0,-1):
        y[k,:]=alpha*y[k+1,:] + beta*y[k+2,:] + c[k]
    return(beta*y[2,:] + F1*y[1,:] + c[0])
```

We also do the calculation directly using the fourier sum.

10d ⟨* 10a⟩+≡ ◁10c 11a▷

```
def fourier(N,c,x):
    z=zeros(x.shape)
    for k in range(N+1):
        z += c[k]*cos(k*x)
    return(z)
```

The above functions are slow in execution since each iteartion of the loop does only a little work and Python spends most of its time in figuring out what we want to do. This immediately suggests that we should implement in C.

The beauty of Python is that there is a facility to implement C code "inline" just as most C compilers permit the insertion of assembly code "inline". Here is the code to speed up the Clenshaw algorithm.

11a ⟨* 10a⟩+≡ ◁10d 11b▷

```
def clenc(N,c,x):
    n=len(x)
    y=zeros((N+3,n))
    f=cos(x)
    code = """
    double alpha;
    for( int j=0 ; j<n ; j++ ){
        alpha = 2.0*f[j];
        for( int k=N ; k>0 ; k- ){
            Y2(k,j)=alpha*Y2(k+1,j)-Y2(k+2,j)+c[k];
        }
        f[j]=-Y2(2,j)+f[j]*Y2(1,j)+c[0];
    }
    """
    weave.inline(code,["y","c","f","N","n"],compiler="gcc")
    return(f)
```

The string `code` contains the fragment of code to be executed. It is basically C code, but has one special feature. The arrays defined in Python do not directly make sense in C. So the Python interpreter makes available, for every array that is passed to the code, ways of accessing the array as a vector and as an array. To access as a vector, just use the variable directly. But to use as an array, capitalize the name and then add the number of dimensions (here two). This is now a `macro` and and the array indices are passed as arguments. That is why `y[i,j]` becomes `Y2(i,j)` in the C code.

The C code is conditionally compiled (that is, it is compiled if the source has changed) and executed by the `weave.inline` statement. The first argument is the code itself. The second argument is a list of Python variables that are passed to the code. There are many more arguments, but here we only specify the compiler to be `gcc`.

How is information returned by the code? Well, we use a tricky feature of Python. The array `y` is a pointer to memory. That is passed "by value". But the data to which it points can be changed by C and those changes will persist. That is what is done here. The array `y` and the vector `f` are modified and their values are returned for Python to continue to work on them. As it turns out `y` could have been local to the C code itself, and I just put it in to show how array access is done.

We can also convert the direct algorithm by inlining C:

11b ⟨* 10a⟩+≡ ◁11a 12a▷

```
def fourc(N,c,x):
    n=len(x)
    z=zeros(x.shape)
    code="""
    double xx;
    for( int j=0 ; j<n ; j++ ){
        xx=x[j];z[j]=0;
        for( int k=0 ; k<=N ; k++ )
            z[j] += c[k]*cos(k*xx);
    }
    """
    weave.inline(code,["z","c","x","N","n"],compiler="gcc")
    return(z)
```

Here we can see direct vector access and the code is intuitively obvious. Now we run these codes and see how much time they take and how accurate they are. We define M to be the number of times to run the code for getting speed information and create the coefficient vector $c_n = 1/\left(a^2+n^2\right)$

12a ⟨* 10a⟩+≡ ◁11b 12b▷

```
M=1000
nn=arange(N+2)
c=array(1/(nn*nn+a*a))
```

We now run the clenshaw routine and time it. To do this we use the `time.time()` function which returns time correct a millisecond.

12b ⟨* 10a⟩+≡ ◁12a 12c▷

```
t1=time.time()
for i in range(M):
    f=clenshaw(N,c,x)
t2=time.time()
py1=(t2-t1)/M
print "Time for clenshaw=%f" % py1
fmax=max(abs(f))
```

Now we run the clenshaw routine with inline C and time it.

12c ⟨* 10a⟩+≡ ◁12b 12d▷

```
t1=time.time()
for i in range(M):
    fc=clenc(N,c,x)
t2=time.time()
pyc=(t2-t1)/M
print "Time for clenshaw implemented in C=%f" % pyc,
print " (speedup=%f)" % (py1/pyc),
# print relative error with respect to the clenshaw in python.
print " rel err=%e" % (max(abs(f-fc))/fmax)
```

Now for the direct calculation in python.

12d ⟨* 10a⟩+≡ ◁12c 12e▷

```
t1=time.time()
for i in range(M):
    z=fourier(N,c,x)
t2=time.time()
f1=(t2-t1)/M
print "Time for fourier=%f" % f1,
print " rel err=%e" % (max(abs(f-z))/fmax)
```

The direct calculation with inline C:

12e ⟨* 10a⟩+≡ ◁12d 13a▷

```
t1=time.time()
for i in range(M):
    zz=fourc(N,c,x)
t2=time.time()
f2=(t2-t1)/M
print "Time for fourier in C=%f" % f2,
print " (speedup=%f)" % (f1/f2),
print " rel err=%e" % (max(abs(f-zz))/fmax)
```

Looking at the results, here is what we get for $a = 0.5$:

Program	Clenshaw in Python	Clenshaw in C	Fourier in Python	Fourier in C
Running Time	4.35 msec	0.11msec	3.22 msec	0.253 msec
Speed up	-	43.2	-	12.7
Relative Error	-	1.64×10^{-16}	8.22×10^{-16}	8.22×10^{-16}

A very strange result! The fourier calculation takes **less** time in Python than the Clenshaw. This is because python spends most of its time in figuring out what you want it to do, not in the numerical calculations themselves.

The *C* implementations are much faster than the pure Python ones, as expected. Note that in *C*, Clenshaw is much faster than the direct calculation. This despite a Pentium processor that has a fast algorithm to compute sines and cosines. On a general processor, it would be a no contest.

A note about the error. You can try different values of a and you will find that large a require more fourier terms and also that the error is larger. Why is this?

The error is primarily round off error. i.e., each operation has an error equal to half the LSB. These errors add up randomly and hence grow with the number of operations. The larger the a, the larger the number of terms in the sum, and hence the larger the error.

We finally plot all the curves and ofcourse they lie over each other just confirming the correctness of the different methods.

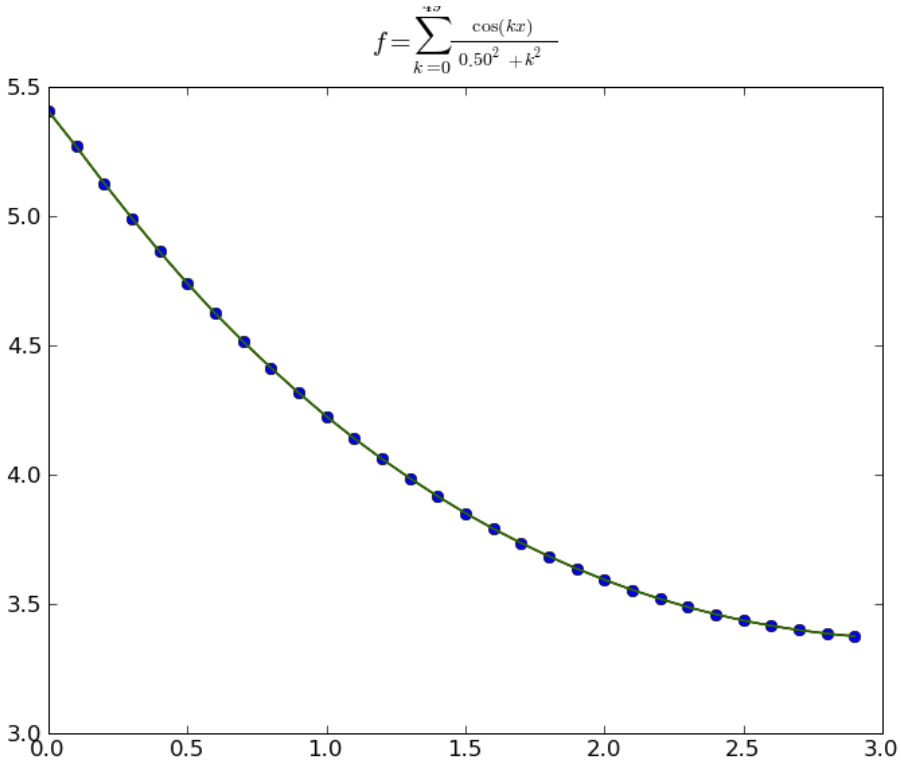
13a ⟨* 10a⟩+≡ ◁12e 13b▷

```
plot(x,f,'bo')
plot(x,z,'k')
plot(x,fc,'r')
plot(x,zz,'g')
```

We now write out a title. Here Python really shines. Usually when we modify parameters, our computations are ok, since they depend on the parameters. But often our strings need manual tweaking, since we used string literals. But here, we have created a complicated math expression for the title. And in that expression the values of N and a are taken from the parameter values defined at the top of the code. Change that value and your title will still be correct. AND the math expression will reflect the new values! try it and see.

13b ⟨* 10a⟩+≡ ◁13a

```
title(r"$f=\sum_{k=0}^{%d} \frac{\cos(kx)}{%.2f^2+k^2}$" % (N,a))
show()
```



Note the structure of the string: `r'''...'''`. This means that the string should not be processed in any way by Python. Within the string is a TEXmath expression which is processed by a module in Python and formatted appropriately. It is within that expression that the cleverness I mentioned above happens. Look at the fragment `\sum_{k=0}^{%d}`. This is a template waiting for something to replace the `%d`. That something is *N* which is passed to the string as an argument.

Someone who has used templates in C++ will feel very comfortable with Python strings. They are just string templates.

In conclusion - C programming in Python is very simple. So what are the gotchas? There is only one. That is - what to do if something goes wrong? For example, I had first written the `clenc` function as

```
def clenc(N,c,x):
    n=len(x)
    y=zeros((N+3,n))
    f=cos(x)
    code = """
    double alpha;
    for( int j=0 ; j<n ; j++ ){
        alpha = 2.0*f[j];
        for( int k=N ; k>0 ; k-- ){
            y[k,j]=alpha*y[k+1,j]-y[k+2,j]+c[k];
        }
        f[j]=-y[2,j]+f[j]*y[1,j]+c[0];
    }
    """
    weave.inline(code,["y","c","f","N","n"],compiler="gcc")
    return(f)
```

As discussed above, `y` is only known as a pointer to memory. So `y[k,j]` makes no sense to the C compiler. The program objected. How to debug?

If you look at the output from the Python interpreter, you will see gcc error messages that actually tell quite a bit. But you can also get the name of the source code that was created from the inline C code. This is usually in your home directory in a hidden subdirectory called `~/.python25_compiled/`. So if it says there is an error on line 675 of this file, you can directly open that file and look at that line. The entire inline code is present as is, and you can see what the gcc objection is actually saying.

If you wish to edit and recompile that code directly, then you have to use the gcc command also mentioned by the Python interpreter. It is a complex one that includes header files etc. It is not easy, but it is not that difficult either.

4 Debugging Python

Python has a built in debugger. You can take advantage of it in several ways.

- You can edit the python source file in the `idle` editor. The editor starts the python window as well. In that window, click on `Debug->debugger`. The debugger is very slow if you turn on local or global display. This is because it has to evaluate and display all the variables at each step. So turn on the source display and turn off the variables. Then go to the `idle` edit window and press F5, which runs the module. You are now in the debugger. Ofcourse this does not allow you to debug the C code, only the python code.

- You can debug the python file in **ipython**. To do this, start ipython as

  ```
  ipython -pylab
  ```

  At the prompt, you can run codes with

  ```
  run abc.py arg1 arg2 arg3
  ```

  where `abc.py` is the source file and arguments to the source file are entered after the source file name. To debug the source file, use

  ```
  run -d abc.py arg1 arg2 arg3
  ```

  and type 'c<Enter>' at the prompt. This will start up the python debugger and place you at the beginning of the program. The interface is pretty much the same as gdb. So, for instance, to put a break point on line 18 to stop when $i = j$, we enter (at the pdb prompt):

  ```
  b 18,i==j
  ```

  Suppose we want to stop in a loop at line 21 whenever $i = j - 1$ and print out the value of a variable, `p1`.

  ```
  b 21,i==j-1
  commands
  silent
  p p1
  end
  ```

  This only stops at line 21 when the condition is satisfied. It does not print the usual information about the break point number etc, but prints the value of `p1`.

- My preferred way to debug is to emacs. Emacs has a unified debugging interface for C, C++, Fortran, perl and python. To start the python debugger, you need to set up things in emacs.

  - First load a python file in emacs and make sure that your emacs version understand the python syntax (the window will say "Python" mode or "Python Abbrev Fill" mode or something like that.) If it does not, you have to get the corresponding lisp file to teach emacs the editing specialities of python. Usually python support is built in.
  - Emacs has a command called "pdb" which starts the python debugger on a source code. However, this expects the python debugger to be called "pdb" and to be found in the path. To make this happen, on my PC I have defined

```
#!/bin/sh
exec /usr/bin/python /usr/lib64/python2.6/pdb.py $*
```

The actual command will change depending on your python version. I have version 2.6 on a 64 bit processor so this is the command that works.

That is it. You can now do source debugging of python code as you wish. I have not yet figured out how to debug C under python as I do for C under Scilab. But I am sure it can be done.

- Using which ever method you wish, run week0b.py and place a breakpoint on the return statement of clenshaw(). Use commands to print out y.

5 Profiling

While debugging finds the errors in your code, the more challenging problem is to identify either subtle bugs or inefficient portions of your code. Debugging cannot easily find such parts of your code. What you need is to profile your code.

Write the python code in this assignment to a file, `week0.py`. We can execute the python code as

```
python week0.py
```

This will run the code and show the graphs (provided you had a `show()` command in your script) and messages on the console. It tells you that the C code ran faster than the python code.

Now we run the code under the profiler.

```
python -m cprofile week0.py > profile.log
```

This runs the script `week0.py` under the profiler and writes the output to `profile.log`. Note that the first few lines of the output will actually be the output of the script. Following that is the output of the profiler itself. This is too detailed. Let us look at the first few lines of it:

```
Time for clenshaw=0.000694
Time for clenshaw implemented in C=0.000022  (speedup=31.331461)  rel err=0.000000e
Time for fourier=0.000572  rel err=8.215998e-16
Time for fourier in C=0.000083  (speedup=6.907532)  rel err=8.215998e-16
         973789 function calls (955354 primitive calls) in 4.566 CPU seconds
   Ordered by: standard name
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    4.567    4.567 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 <string>:1(ArgInfo)
        1    0.000    0.000    0.000    0.000 <string>:1(ArgSpec)
        1    0.000    0.000    0.000    0.000 <string>:1(Arguments)
        1    0.000    0.000    0.000    0.000 <string>:1(Attribute)
        1    0.000    0.000    0.000    0.000 <string>:1(DBNotFoundError)
```

The first four lines are program output. Following that the profiler starts and gives statistics in a table. The important number is the “cumtime” which is the cumulative time spent by the function.

Let us parse this profile and look for those functions that correspond to our code `week0.py` and are ordered by cumulative time.

```
grep week0 profile.log|sort -n -k4,4
```

This extracts those lines containing the word `week0` in the profile output. It then sorts them in numerically ascending order (-n) and the sorting is done on field 4 (-k4,4). The result of this command is:

```
1000    0.005    0.000    0.021    0.000 week0.py:22(clenc)
1000    0.002    0.000    0.082    0.000 week0.py:38(fourc)
1000    0.569    0.001    0.571    0.001 week0.py:17(fourier)
1000    0.682    0.001    0.692    0.001 week0.py:9(clenshaw)
   1    0.007    0.007    4.566    4.566 week0.py:1(<module>)
```

We can see that the least time was spent in routine `clenc` and the most time was in `clenshaw`. Ofcourse we knew this already, but this is an invaluable tool when used for looking at which part of the code to optimize.

- Run the code under the profiler and see if the above in your machine.