Introduction to Tools - Python

August 7, 2016

1 Assignment

1. Revise your Python skills, especially scientific python (numpy, scipy & pylab)
2. We wish to interpolate $f(x) = sin(x)$ from a table of function values sampled at

   $$0, \pi/4, \pi/2, 3\pi/4, \pi, 5\pi/4, 3\pi/2, 7\pi/4, 2\pi$$

   Write a Lagrange Interpolation function with the following calling sequence

   ```
   float lintp(float *xx,float *yy,float x,int n)
   ```

   in C that will perform an n^{th} order interpolation at the point x. Use $n = 8$ and generate the interpolated values.
3. Run the program and evaluate the function on 100 uniformly spaced values between 0 and 2π. Write out the answers to a text file *output.txt* with each line containing

   ```
   x y
   ```
4. Write a python script that uses system via

   ```
   import os
   os.system("...")
   ```

   to run the C executable. It should then use loadtxt to read in *output.txt* and plot both the interpolated values as well as the exact function $sin(x)$ in a plot. Another plot should plot the error between the interpolated and exact function vs x.
5. Suppose we wanted only a cubic interpolation, even though we have 9 points. For each x we would need to locate the nearest set of 4 points and then do Lagrange interpolation on them. change the C main program, while keeping the function unchanged to achieve this. Plot the agreement and the error as in part 4.

2 Using Python with inline C

There are two ways in which to do this. The first (that I prefer) has been officially deprecated by the python folks. That is to use the *weave* package. This is built into python 2.7 but is missing in python 3.x

The other approach is to use *cython*, which is the official inline C approach.

Here I will show how to use weave through an example.

2.1 Weave

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.

2a ⟨* 2a⟩≡ 2b▷

```
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.

2b ⟨* 2a⟩+≡ ◁2a 2c▷

```
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

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

This can be done as a double sum:

```
for( i=0 ; i<M ; i++ ){

    z[i]=0.0;
    for( k=0 ; k<N ; k++ )
        z[i] += c[k]*cos(k*x[i]);

}
```

Python's vector operations eliminate one of the sums by calculating the series for the entire set of x values at one go. Here is the Python code:

2c ⟨* 2a⟩+≡ ◁2b 2d▷

```
def fourier(N,c,x):
    z=zeros(x.shape)  # create and initialize z
    for k in range(N+1):
        z += c[k]*cos(k*x) # update kth term for all z
    return(z)
```

This is as fast as Python can make the code. Now let us do some inline C coding.

2d ⟨* 2a⟩+≡ ◁2c 3a▷

```
def fourc(N,c,x):
    n=len(x)
    z=zeros(x.shape)
        # now define the C code in a string.
    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)
```

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 we only need to manipulate vectors. So we directly use the arrays. Later we will see how to handle arrays.

Weave is a module, and has a function *inline* that does the hard work. The list of variables passed to *inline* are the Python variables that are made available to C. Any other variables in the C code are local variables. In the above code, xx is a local variable.

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 vector `z` is modified and its values are returned for Python to continue to work on them.

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 $c_n = 1/\left(a^2+n^2\right)$

3a ⟨* 2a⟩+≡ ◁2d 3b▷

```
M=10000
nn=arange(N+2)
c=array(1/(nn*nn+a*a))
```

We run the calculation in python by calling *fourier*.

3b ⟨* 2a⟩+≡ ◁3a 3c▷

```
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,
```

Now we do the same with the inline version, *fourc*

3c ⟨* 2a⟩+≡ ◁3b

```
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),
```

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

Program	Fourier in Python	Fourier in C
Running Time	0.654 msec	0.094 msec
Speed up	-	6.95

The *C* implementations are much faster than the pure Python ones, despite the python code being vectorized.

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.

3 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.

4 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 print 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 fourier=0.000572  Time for fourier in C=0.000083  (speedup=6.907532)
         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    0.002    0.000    0.082    0.000 week0.py:38(fourc)
1000    0.569    0.001    0.571    0.001 week0.py:17(fourier)
   1    0.007    0.007    4.566    4.566 week0.py:1(<module>)
```

We can see that the least time was spent in routine `fourc` and the most time was in `fourier`. 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 the above in your machine.