Lab: More Python Programming and 1D Finite Differencing

Math/CS 471, Intro to Scientific Computing
September 27, 2018

Lab Preliminaries

- You are expected to finish this lab, because these skills will be required during future homeworks. However, you will *not* turn in a lab assignment for grading.
- This lab is intended to introduce you to more advanced Python programming concepts, and then have you implement two finite-differencing formulas, while measuring their convergence.

1 Python Preliminaries

First, make sure that you have a Python environment with Scipy and Matplotlib.

- If you already have such a Python environment (e.g., on your laptop), then skip to Section 2.
- If you don't, then just follow these instructions while on a lab machine. Note, that these instructions may not work on your home machine, because these instructions assume that `pip` has been installed and that you are in a Linux-style environment.

To install Scipy and Matplotlib on a lab machine, clone the repository

```
$ git clone https://lobogit.unm.edu/jbschroder/Fall_2018_Math471.git
```

Then, go into the `Fall_2018_Math471/labs/lab2` directory. You will need to update your local python environment. Let's see how to do that,

```
$ more pip_install
$ more python_alias
$
$ ./pip_install
$ source python_alias
```

Then make sure the python version is 3.7. *Note again that this is only applies to running on the lab machines. Python 2 can work fine on home machines or CARC machines.*

```
$ python
>>> exit()
$ ipython
```

2 Python Debugger

This first part of the lab will go over some more advanced Python features that you didn't see in the previous tutorial information.

1. PDB is the name of the Python debugger, and is remarkably useful when debugging longer programs. This capability lets you "jump" into your code, part way through it's execution, and then step through lines of your program, inspecting data, and discovering what's wrong.
2. Let's see how it works. Create a short Python program, in a file called `pdbtest.py`. We will "set a trace" at a certain line of code, which is Python-speak for setting a breakpoint in the code. This file should contain the following code.

Note: If you copy and paste the below text, remember that all characters might not paste correctly, e.g., the ' character.

```
1  from scipy import *
2  from matplotlib import pyplot
3  from pdb import set_trace
4
5  def fprime(f, h):
6      # Return an approximation to the first derivative
7      # set_trace()
8      fp = zeros(x.shape)
9      fp[:] = (f[1:] - f[:-1])/h
10     return fp
11
12 # set_trace()
13 x = linspace(0,2*pi,31)
14 h = 2*pi/30
15 f = sin(x)
16 fp = fprime(f,h)
17
18 pyplot.plot(x,cos(x))
19 pyplot.plot(x,fp)
20 pyplot.legend(['f-prime', 'f-prime approx.'])
21 pyplot.show()
```

3. Explanation: A breakpoint is a stopping place that halts the execution of your program at certain line. The debugger then lets you inspect all the internals (like variables) of your program at that breakpoint. So, if you set a breakpoint at line 100, then the debugger will halt at line 100, and you can print out the values of variables, interactively type in new lines of code, and even step forward one line at a time in your code. This last feature of stepping through code can be very valuable when trying to understand subtle bugs.

4. Try running this program. It will crash with an error at line 9 in the function. Instead of debugging this purely through deduction, comment in line 7 and set a trace.

5. Re-run the new program, and see that it brings you to a new "shell", the PDB shell, denoted by the prompt `(Pdb)`. This new shell is similar to the iPython prompt, except that this shell is present at line 7, where your code halted.

   Your basic PDB commands are

   - `(Pdb) n`
     which will execute the next line in your code. If the next line is a function call, then that function will execute and return.
   - `(Pdb) s`
     which will execute the next line in your code. But if the next line is a function call, the PDB shell will step inside of that function to the first line of code inside that function.
   - `(Pdb) p <expression>`
     which will print an expression (such as a variable) to the screen
   - `(Pdb) <expression>`
     which execute some expression, like `A = dot(B,C)`.
   - `(Pdb) q`
     which will quit the PDB debugger

   ⇒ PDB lets you step through your code, and figure out where the bugs are. So let's do that!

6. Type `n <enter>` to execute the next line of code (initialize fp to zeros)

7. Type `n <enter>` to execute the next line of code, but you get an error. Let's see why.

8. Type `p fp.shape <enter>` to see the size of the array fp

9. Type `p ((f[1:]  - f[:-1])/h).shape <enter>` to see the size of the array that we are trying to assign to fp. They are of a different size, hence the error.

10. While still in the PDB shell, try out commands to assign fp correctly with the approximate derivative. For instance, try

    ```
    (Pdb) fp[1:] = (f[1:] - f[:-1])/h
    (Pdb) fp[0] = fp[1]
    (Pdb) p fp
    ```

    This a very powerful aspect of PDB–the ability to try out new lines of code in the middle of your program running.

11. These commands seem to work well, and provide an approximation to f' at each grid point. (Please make sure that you understand this.) So, replace `fp[:]  = (f[1:]  - f[:-1])/h` with the above two-line fix.

12. Comment back out the set trace command, and type `q` to quit PDB. Then, re-run `pdbtest.py`.

13. You'll notice that we've used finite-differencing to approximate the derivative of $\sin(x)$. You can increase the number of points (if you're curious) and see how the approximation improves.

Python Debugger 2

1. Here, we will use `s` to step into the troublesome function from above. You will need to use this aspect of PDB to debug more complicated codes.

2. Go back to the original buggy code, but instead comment in the set trace command in line 12. Leave the set trace command in line 7 commented out.

3. Run the code.

4. Type `n <enter>` multiple times to execute all the lines of your code. You'll notice that you cannot execute past line 16, which is the call to `fprime()`, because of the error.

5. Type `q <enter>` to quit PDB.

6. Start again by re-running `pdbtest.py`. But this time, after you get to the point where the next `n` will execute line 16 in the debugger, instead type `s <enter>` to *step* into `fprime`.

    Notice that the next line to be executed in PDB is always displayed right above the `(Pdb)` prompt.

7. Now you are inside of `fprime`, and you can type `n` as before and step through fprime and explore fixing the buggy line of code.

3 Finite Differencing

The last part of this lab concerns measuring the quadratic and quartic convergence of the finite difference stencils derived in class for the second derivative. These stencils are depicted in Figure 1, where the red point is the center point in the stencil.

1. Modify one of your previous scripts to compute the two different finite difference stencils. Use the fact that we know the exact second derivative of $\delta^2 \sin(x)/\delta x^2 = -\sin(x)$ to compute the error for each stencil for $n = 10, 11, ..., 1000$.

    The core of your code should look something like the below fragment.

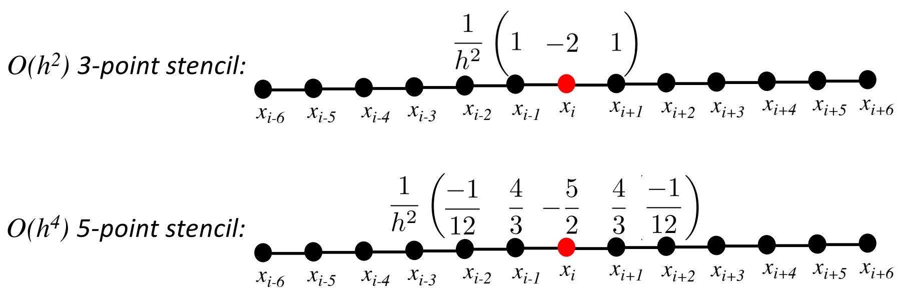
$O(h^2)$ 3-point stencil:
$\frac{1}{h^2}\begin{pmatrix}1 & -2 & 1\end{pmatrix}$
$O(h^4)$ 5-point stencil:
$\frac{1}{h^2}\begin{pmatrix}\frac{-1}{12} & \frac{4}{3} & -\frac{5}{2} & \frac{4}{3} & \frac{-1}{12}\end{pmatrix}$
x_{i-6} x_{i-5} x_{i-4} x_{i-3} x_{i-2} x_{i-1} x_i x_{i+1} x_{i+2} x_{i+3} x_{i+4} x_{i+5} x_{i+6}

Figure 1: Two sample finite difference stencils for $\delta^2 u/\delta x^2$

Note: If you copy and paste the below text, remember that all characters might not paste correctly, e.g., the ’ character.

```
errorp = zeros((1000-10,))
errorp2 = zeros((1000-10,))

for n in range(10, 1000):
    x = linspace(0,2*pi,n)
    h = 2*pi/(n-1)
    f = sin(x)
    fp = zeros( (x.shape[0]-2,) )
    fp2 = zeros( (x.shape[0]-4,) )
    for k in range(1,n-1):
       fp[k-1] = # insert second order accurate stencil
    for k in range(2,n-2):
       fp2[k-2] = # insert fourth order accurate stencil

    fpexact = # exact expression for the second derivative

    # Note that we did not compute the finite difference stencil at boundary points
    errorp[n-10] = norm(fpexact[1:-1] - fp)
    errorp2[n-10] = norm(fpexact[2:-2] - fp2)
```

Note that we are using the 2-norm to compute the size of the error. This function is found in `scipy.linalg.norm`.

2. Plot this error in a log-log plot, along with a reference curve for quadratic and quartic convergence. These reference curves will be plotted with something like the below code.

Note: If you copy and paste the below text, remember that all characters might not paste correctly, e.g., the ’ character.

```
n = arange(10,1000)
pyplot.loglog(n, 1.0/n**2)
pyplot.loglog(n, 1.0/n**4)
pyplot.legend(['FD Plot 1', 'FD Plot 2', 'Quadratic Ref', 'Quartic Ref'])
```

3. Do your numeric results match the reference curves? If not, then there’s a bug. If you don’t understand anything here, please ask!

   Your plot should look like Figure 2.

4. If you run into problems with your code, try using the python debugger.

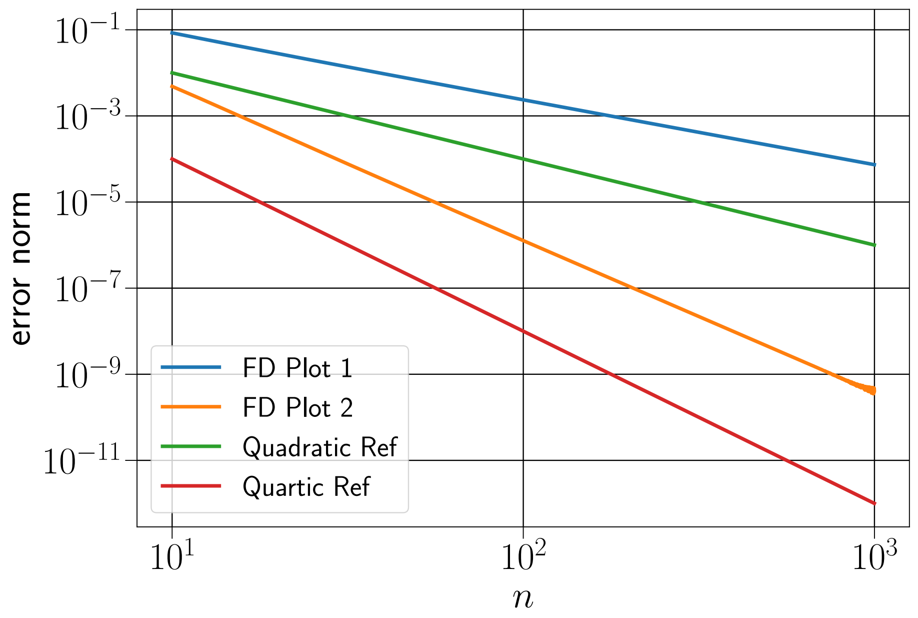

Figure 2: Numerical error for the two different finite-difference stencils.

4 If you finish early...

1. Replace the above loop computation of the derivative with array formulas.

   Note: If you copy and paste the below text, remember that all characters might not paste correctly, e.g., the ' character.

```
x = linspace(0,2*pi,301)
h = 2*pi/300
f = sin(x)
fp = (1/h**2)*(f[:-2] - 2*f[1:-1] + f[2:])
fp2 = ...
```

   Can you time this way of computing numerical derivatives, and compare it to the loop-based approach used above to compute numerical derivatives? Which is faster?

   You can use `time.time()` to get accurate timings in Python.

2. Try computing the second derivative at the boundary points using either the biased stencils derived in class, or by using the exact function values at the boundary (also discussed in class). See Sept. 25, 2018 lecture notes.

3. Feel free to work on your homework, or to help other students.