Python for the working (or aspiring) mathematician

John Kerl

Department of Mathematics, University of Arizona
Software Interest Group

October 26, 2009

Abstract

- Why people use Python.
- The minimum you need to know to write and run simple Python programs, even if you haven't used such languages before.
- Numerics.
- Formatted I/O.
- The `pylab` module, which offers Matlab-like plotting functionality.

Why Python? What can one more language do for you?

- Python programs are fast to develop: no compilation phase. Python syntax is clean, clear, and intuitive: it looks as simple as pseudocode.
- Unlike monolithic programs such as Matlab, Mathematica, etc., it's easy to connect (pipe) Python scripts to other programs.
- Python has built-in support for text manipulation, flexible lists, and complex arithmetic.
- It works the same on Unix/Linux or Windows.
- It's good for doing math, as well as other quick-and-dirty programs. Python is a general-purpose language: which you can use to automate all sorts of miscellaneous tedious time-consuming tasks — times when you think *I am really sick and tired of this miscellaneous tedious time-consuming task which is keeping me from what I'd rather be doing* — grade calcuations, turning data into nicely formatting LaTeX output, making bulk changes to many files at once, making a nice index for LaTeX documents, etc. etc.

Reading proficiency

The department has a foreign language requirement for PhD students. Why? Well, not all important results are published in your native language!

By the same token, if you're doing compuational work of any sort, you may find that your collaborators' favorite programming languages aren't the same as yours.

If you're going to be doing numerical work, learn some Python, C, Fortran, Matlab, R.

Magic

There are many pre-existing software tools. In this presentation, I'll give examples of an ODE solver and a data formatter. Matlab et al. already do these things for you. Why should you listen to me reinventing the wheel?

- Python is freely available; Matlab is not.
- Pre-existing packages don't always have what you want — be ready to roll your own when necessary.
- Computers appear to be magical. When you learn to program, you learn to create your own wizardry and find out how things really work. Python is a good starter language.

Installation and on-line help

Installation:

- Unix/Linux: It's already there.
- Mac: It's there too — Python is included in Mac OS X.
- Windows: Download it from the web.

On-line help example (we'll talk about `re.sub` in a few slides):

```
[shell prompt]$ python
>>> import re
>>> help(re.sub)
>>> exit()
```

Running Python programs

Put the following text in a file, say `hello.py`:

```
print 'Hello, world!'
```

Then at the command prompt, type:

```
[shell prompt]$ python hello.py
```

You should see:

```
Hello, world!
```

More about running Python programs

For simplicity of presentation, I suggested typing `python hello.py`.

You can also just type `hello.py`, if you first do the following:

- On Unix/Linux: The first line of `hello.py` should be `#!/usr/bin/python`. Then, at the command line, `chmod u+x hello.py`. (The `.py` extension isn't necessary.)
- On Windows: You must use a `.py` extension. Also, set up an association to open such programs using `python`.
- I can give you more info about this topic but it's a technical detail.

Numerics by example

Here is a naive Euler (first-order) integrator for the Lorenz equations. (Feel free to implement higher-order techniques such as Runge-Kutta — it would simply occupy another slide or two.)

Derivation of Euler scheme:

$$\begin{aligned} x &= x(t) \\ x' &\approx \frac{x(t+h) - x(t)}{h} \\ x(t+h) &\approx x(t) + hx' \\ x_{k+1} &:= x_k + hx' \end{aligned}$$

Let's turn that into instructions for a computer to carry out, using the Lorenz equations.

Lorenz equations

System of non-linear ODEs:

$$\begin{aligned} x' &= s(y - x) \\ y' &= rx - y - xz \\ z' &= xy - bz. \end{aligned}$$

Constants:

$$s = 10.0, \qquad r = 28.0, \qquad b = 2.6666667.$$

Initial conditions:

$$x_0 = 10, \qquad y_0 = 0, \qquad z_0 = 10.$$

Numerical script

```
tmax = 10.0; h = 0.002; N = int(tmax/h) # 10 seconds with 0.002-sec step
s    = 10.0; r = 28.0;  b = 2.666667    # Constants
t    = 0;    x = 10;    y = 0; z = 10   # Initial conditions
while t < tmax:
    print '%11.8f %11.8f %11.8f' % (x, y, z)
    xp = s * (y - x);       # Cheesy Euler integrator.
    yp = r * x - y - x * z
    zp = x * y - b * z
    x = x + h * xp; y = y + h * yp; z = z + h * zp; t += h
```

- Variable assignments, arithmetic expressions, etc. are about what you'd expect.
- Multiple statements on a single line are separated by semicolons.
- Control structures (for, if, while, etc.) are delimited using tab stops.
- The `print` syntax is a lot like in C or Perl.

Output of numerical script

```
[shell prompt]$ python lorenz.py
10.00000000  0.00000000 10.00000000
 9.80000000  0.36000000  9.94666666
 9.61120000  0.71312533  9.90067376
 9.43323851  1.05961157  9.86157814
 9.26576597  1.39970047  9.82897419
 9.10844466  1.73363801  9.80249158
 8.96094853  2.06167273  9.78179311
 ...         ...         ...
```

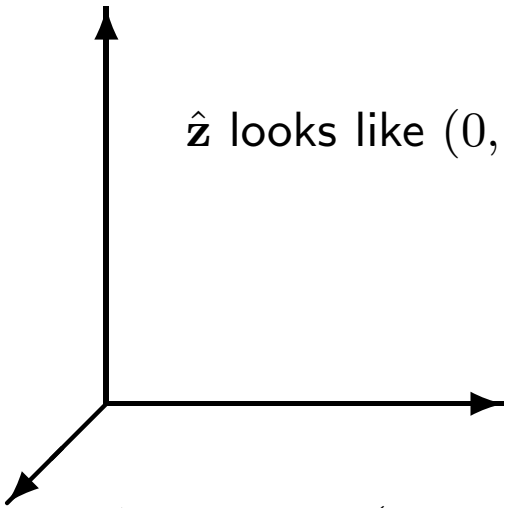
OK, so we're seeing the estimated x's, y's, and z's. But how to plot three variables on a two-dimensional screen?

Data manipulation by example

Answer: Projection from 3D to 2D is just a linear transformation. Write down a 2x3 matrix P whose columns are where you want $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$, and $\hat{\mathbf{z}}$ to go. Pre-multiply each data triple by that:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} (P\hat{\mathbf{x}})_0 & (P\hat{\mathbf{y}})_0 & (P\hat{\mathbf{z}})_0 \\ (P\hat{\mathbf{x}})_1 & (P\hat{\mathbf{y}})_1 & (P\hat{\mathbf{z}})_1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Then plot (u, v) pairs on the two-dimensional display.



$\hat{\mathbf{z}}$ looks like $(0, 1)$ in 2D

$\hat{\mathbf{y}}$ looks like $(1, 0)$ in 2D

$\hat{\mathbf{x}}$ looks like $(-0.25, -0.25)$ in 2D

$$P = \begin{bmatrix} -0.25 & 1 & 0 \\ -0.25 & 0 & 1 \end{bmatrix}$$

Outline of projection script

We'll make a separate (re-usable!) script that simply reads three-column data — from our Lorenz integrator *or anything else* — and writes two-column data, modified as described above:

- Read lines of input (triples of numbers, delimited by whitespace) one line at a time, splitting the line into three numbers x, y, and z.
- For each triple x, y, z, compute u and v.
- Write u and v.

Subroutines

Here's the computation part. I'll use this as an example of subroutines. Assume we've already read in a triple of x, y, and z. We'll call this routine to find what u and v are.

```
Pxhat = [-0.25, -0.25]; Pyhat = [1, 0]; Pzhat = [0, 1]

def project_3D_to_2D_triple(x, y, z, Pxhat, Pyhat, Pzhat):
    u = Pxhat[0] * x + Pyhat[0] * y + Pzhat[0] * z
    v = Pxhat[1] * x + Pyhat[1] * y + Pzhat[1] * z
    return [u, v]
```

- Subroutines start with the `def` keyword, and they return values (scalars, lists, etc.) using the `return` keyword.
- You can have a list of scalars on the left-hand side of an assignment. E.g. the caller can do

  ```
  [u,v] = project_3D_to_2D_triple(x, y, z, Pxhat, Pyhat, Pzhat)
  ```
- Arrays are indexed (from zero up, math folks!) using square brackets.

Data I/O

We need a routine to read files like this:

```
10.00000000  0.00000000 10.00000000
 9.80000000  0.36000000  9.94666666
 9.61120000  0.71312533  9.90067376
 9.26576597  1.39970047  9.82897419
 ...
```

columnwise into Python arrays such as

```
xs = [10.00000000, 9.80000000, 9.61120000, 9.26576597, ...]
ys = [ 0.00000000, 0.36000000, 0.71312533, 1.39970047, ...]
zs = [10.00000000, 9.94666666, 9.90067376, 9.82897419, ...]
```

and, vice versa, a routine to write such arrays to files.

I wrote a Python module called tabutil.py which does these things. For the sake of time, I won't show the details today. Our script can simply import the module, and call its subroutines.

Projection script

```
import tabutil

def project_3D_to_2D_triple(x, y, z, Pxhat, Pyhat, Pzhat):
    u = Pxhat[0] * x + Pyhat[0] * y + Pzhat[0] * z
    v = Pxhat[1] * x + Pyhat[1] * y + Pzhat[1] * z
    return [u, v]

Pxhat = [-0.25, -0.25]; Pyhat = [1, 0]; Pzhat = [0, 1]
[xs, ys, zs] = tabutil.float_columns_from_file('xyzdata.txt')
for i in range(0, len(xs)):
    x = xs[i]; y = ys[i]; z = zs[i]
    [u, v] = project_3D_to_2D_triple(x, y, z, Pxhat, Pyhat, Pzhat)
    print '%11.7f %11.7f' % (u, v)
```

Output of numerical and projection scripts

```
[shell prompt]$ python lorenz.py | python projdown.py
 -2.5000000   7.5000000
 -2.0900000   7.4966667
 -1.6896747   7.4978738
 -1.2986981   7.5032685
 -0.9167410   7.5125327
 ...          ...
```

Or:

```
[shell prompt]$ python lorenz.py > xyz.txt
[shell prompt]$ python projdown.py < xyz.txt > uv.txt
```

Now one would like to see a picture!

Plotter script

The pylab module intentionally mimics Matlab. The basic piece to make a plot is

```
import pylab
... compute the us and vs arrays as above ...
pylab.figure()
pylab.plot(us, vs)
pylab.show()
```

Inside a computational script, you can simply call these subroutines. Alternatively, we can create a separate, reusable script which can plot *any* two-column data file. Here is my script plot2.py:

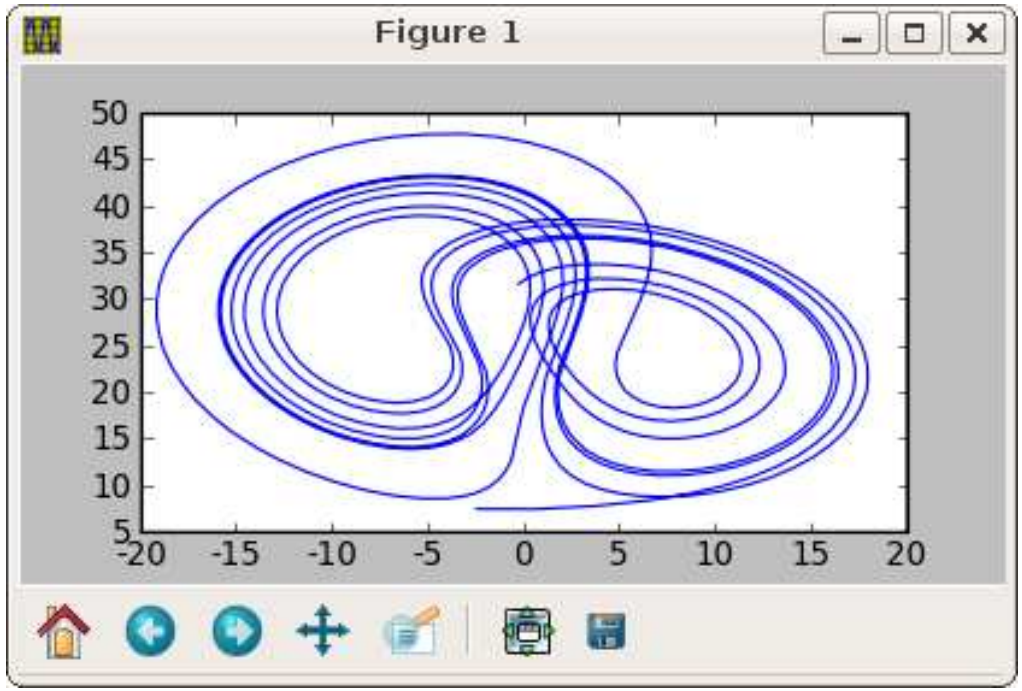
```
#!/usr/bin/python -Wall
import sys, tabutil, pylab
input_file_name = '-' # Read from standard input
if len(sys.argv) == 2: input_file_name = sys.argv[1] # Read from file
[us, vs] = tabutil.float_columns_from_file(input_file_name)
pylab.figure()
pylab.plot(us,vs)
pylab.show()
```

Output of plotter script

Now we can pipe all three of our programs together:

```
python lorenz.py | python projdown.py | python plot2.py
```

On the screen, we see the following:



Output of plotter script

More plot options:

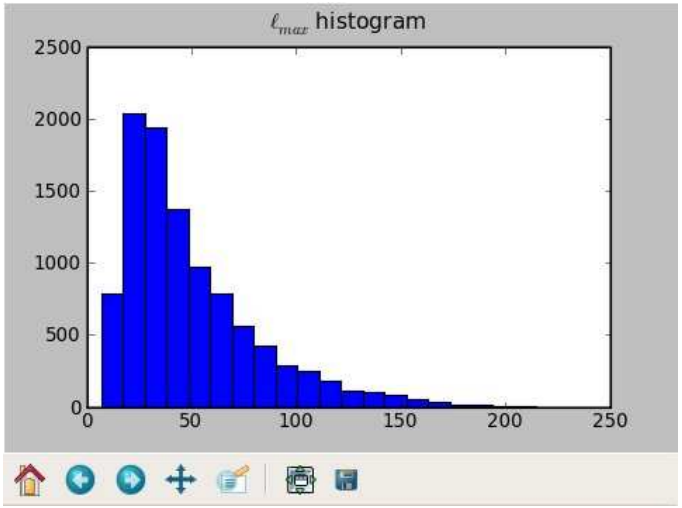
```
pylab.figure(1, figsize=(8, 3)) # Specify figure width/height in inches
pylab.plot(xs,ys,'ro-', markersize=8, label='legend label here')
   # 'ro-' means large red dots, with lines connecting the dots.
pylab.errorbar(xs, ys, y_errors)
pylab.bar(xs, ys, width=x[1]-x[0])
pylab.xlabel('Horizontal axis label'); pylab.ylabel('Vertical axis label')
pylab.title('Title text')
pylab.legend() # Make the above legend labels appear
if I_want_an_EPS_file:
    pylab.savefig('myfile.eps', dpi=150) # To disk
else:
    pylab.show()                         # To screen
```

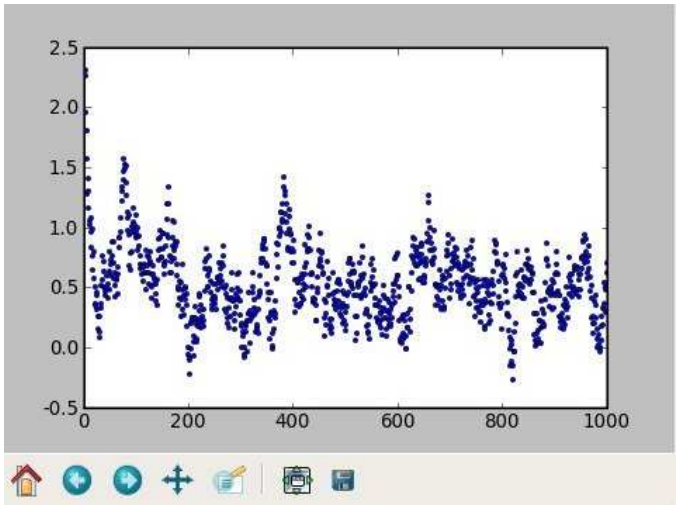
For more information, do the following:

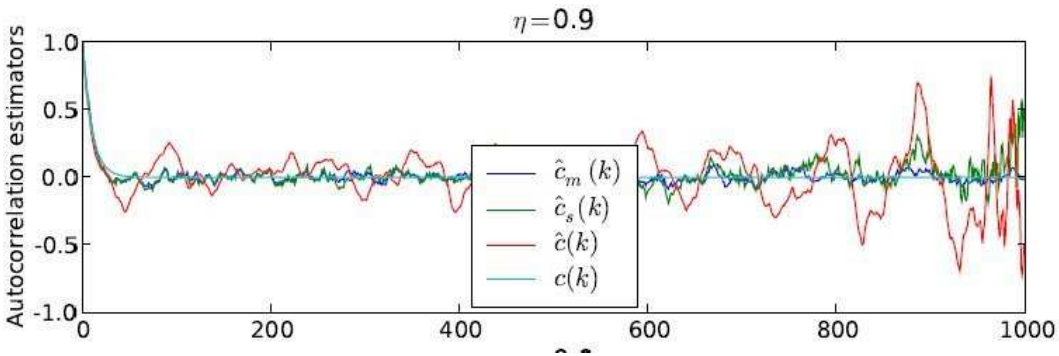
```
[shell prompt]$ python
>>> import pylab
>>> help(pylab.plot)
```

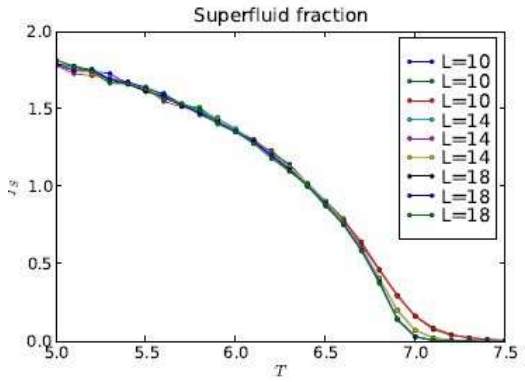
Sample plot outputs

I am using `pylab` exclusively for presentation graphics in my dissertation. Here are some examples of `pylab` plots, made using options such as those on the previous slide:









Downloading the sample scripts

Code used in this presentation is available in the directory

`http://math.arizona.edu/~kerl/doc/python-talk/`

These include:

`hello.py`
`lorenz.py`
`plot2.py`
`projdown.py`
`tabutil.py`

Also, `all-in-one.py` skips the file I/O and the pipes: it does the numerical integration, projection from 3D down to 2D, and the plotting all in one script.

Modules and references

There are a bajillion modules out there. In particular, you might web-search for `Scipy` and `Numpy`. These include numerical linear algebra (e.g. LAPACK/BLAS), database interfaces, etc. etc.

For more information:

- *Learning Python*, by Mark Lutz, published by O'Reilly.
- `www.python.org`
- `docs.python.org/tutorial`
- `http://www.aero.iitb.ac.in/~prabhu/tmp/python_cep07/course_handouts/science.pdf` (Whew!)

Thanks for attending!