# AMath 483/583 — Lecture 28 — June 1, 2011

Today:

- Python plus Fortran
- Comments on `quadtests.py` for project
- Linear vs. log-log plots
- Visualization

Friday:

- Animation: plots to movies
- Binary I/O
- Parallel IPython
- Reproducible research
- Course evaluations (please come!)

# Python scripting for Fortran codes

It's often convenient to run tests and plot results using Python

Final project includes `testquads.py`

```python
from pylab import *;    import os

def run_tests(nlist=None):
    # function body

if __name__=="__main__":
    dx,etrap,esimp = run_tests()
    plot_errors(dx,etrap,esimp)
```

The last bit is executed only if the code is run via one of:

```
$ python testquads.py
>>> execfile('testquads.py')
In[1] run testquads.py
```

Not executed if it is imported as a module.

# Python scripting for Fortran codes

```python
def run_tests(nlist=None):

    if nlist is None:
        # default: [10, 20, 40, ..., 10*2**14]:
        nlist = [10*2**j for j in range(15)]

    # Create arrays to accumalate dx and errors
    # Initialize to all ones.
    dx = ones(len(nlist))
    error_trap = ones(len(nlist))
    error_simpson = ones(len(nlist))
```

Uses list comprehension to define default `nlist`.

Arrays are initialized for accumulating results of tests.

# Python scripting for Fortran codes

Run a set of tests and read in results:

```python
for i in range(len(nlist)):
    n = nlist[i]
    infile = open('input.txt','w')
    # convert n to a string to write out,
    # followed by newline \n:
    infile.write("%s \n" % n)
    infile.close()

    print "Running code with n = ",n
    os.system('make run')

    outdata = loadtxt('output.txt')
    dx[i] = outdata[0]
    error_trap[i] = abs(outdata[1])
    error_simpson[i] = abs(outdata[2])
```

## Expected error

For smooth functions, the error in the Trapezoidal Rule will be $\mathcal{O}(\Delta x^2)$ as $\Delta x \to 0$:

$$E(\Delta x) \approx C_2 \Delta x^2 + C_3 \Delta x^3 + \cdots$$

for some constants $C_2$, $C_3$, etc. that depend on the integrand.

Note:

- This is only a statement about how error behaves for $\Delta x$ sufficiently small.

- This is a statement about how errors behave in exact arithmetic.

  On the computer, rounding errors will eventually dominate. Cannot expect error to go below rounding level.

# Plotting errors vs. $\Delta x$

Plotting the error on a linear scale is hard to interpret:
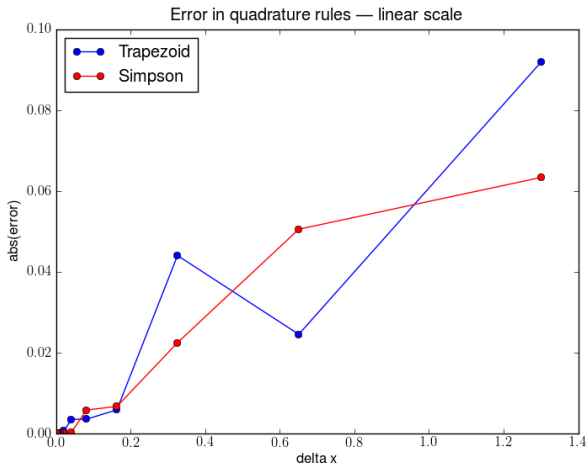


Behavior for small $\Delta x$ is impossible to see.

# Expected error in log-log scale

Trapezoidal Rule: $E(\Delta x) \approx C_2 \Delta x^2$

Taking logarithms:

$$\log(E) \approx \log C_2 + \log(\Delta x^2)$$
$$= \log C_2 + 2 \log(\Delta x)$$

So $\log(E)$ is a linear function of $\log(\Delta x)$ with slope 2.

# Expected error in log-log scale

Trapezoidal Rule: $E(\Delta x) \approx C_2 \Delta x^2$

Taking logarithms:

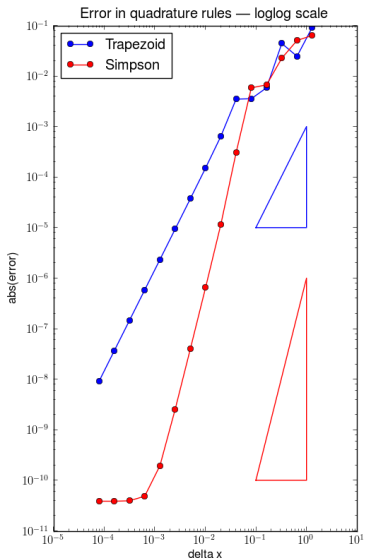$$\log(E) \approx \log C_2 + \log(\Delta x^2)$$
$$= \log C_2 + 2 \log(\Delta x)$$

So $\log(E)$ is a linear function of $\log(\Delta x)$ with slope 2.

Simpson's Rule: $E(\Delta x) \approx C_4 \Delta x^4$

$$\log(E) \approx \log C_4 + \log(\Delta x^4)$$
$$= \log C_4 + 4 \log(\Delta x)$$

So $\log(E)$ is a linear function of $\log(\Delta x)$ with slope 4.

# Plotting errors vs. $\Delta x$ on log-log scale

# Computer graphics / animation

3D Graphics is big business (gaming, animation).

Has driven development of high performance computing.

Example: The movie Avatar

240,000 computer generated frames (24 frames per second)

12 MB per frame, 288 MB per second,
    17.3 GB of data per minute of film.

Months of computing on 40,000 processors
    with 104 Terabytes of RAM.

[reference]

# GPUs — Graphical Processor Unit

Rendering graphics requires extensive processing.

For example, rotation of image requires:
  multiplying large number of vectors by rotation matrix.
  hidden line removal
  lighting/shading

Recent graphics boards have up to 512 cores.

Initially very specialized, not suitable for general programming.

GPGPU: General purpose GPU programming:

- CUDA, PyCUDA (nVidia)
- OpenCL

# Graphics and Visualization

Many tools are available for plotting numerical results.

Some open source Python options:

- matplotlib for 1d plots and
    2d plots (e.g. pseudocolor, contour, quiver)

- Mayavi for 3d plots (curves, surfaces, vector fields)

Mayavi is easiest to get going by installing the
    Enthought Python Distribution (EPD),
    which is available for many platforms.
    (Also includes NumPy, SciPy, matplotlib.)

# Surface plot using Mayavi

Plot of temperature from Homework 6... see
`$CLASSHG/codes/python/plotheat_mesh.py`

# Graphics and Visualization
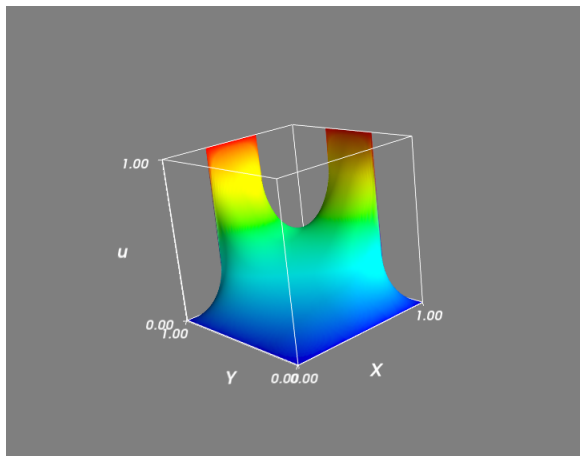
Open source packages developed by National Labs...

- VisIt
- ParaView

Harder to get going, but designed for large-scale 3d plots, distributed data, adaptive mesh refinement results, etc.:

Each have stand-alone GUI and also Python scripting capabilities.

Based on VTK (Visualization Tool Kit).

# Graphics and Visualization

Note: Cannot plot directly from Fortran.

Python Options:

- Compute and plot in Python (may be too slow)
- Compute in Fortran, write to disk, read into Python
- Use f2py to call Fortran subroutines from Python

# Using matplotlib

```
$ ipython -pylab
```

starts `ipython` in manner that interactive plots work.
This also automatically does...

```
from pylab import *
```

which puts all NumPy and matplotlib plotting routines in
namespace, so e.g.:

```
In [1]: x = linspace(0, 1, 101)
In [2]: plot(x, x**2, 'r-o')
```

To make it clear where things come from:

```
In [1]: import numpy as np
In [2]: from matplotlib import pyplot as plt
In [3]: x = np.linspace(0, 1, 101)
In [4]: plt.plot(x, x**2, 'r-o')
```

# Tips on plots for papers, publications

## Make lines thick enough, symbols large enough

These often fade out when reduced or copied

```
plt.plot(x, y, 'o-', linewidth=2, markersize=8)
```

Remember that doc strings can be see by `plt.plot?` in IPython.
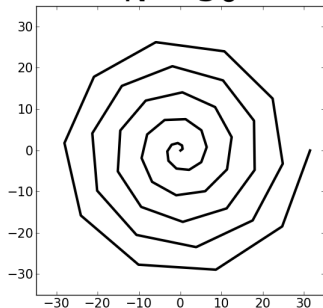
See also documentation and gallery.

## Make fonts large enough to read axes, title, legends, etc.

```
plt.xticks(fontsize=15)
plt.yticks(fontsize=15)
plt.title("Plot of temperature", fontsize=15)
plt.xlabel("x", fontsize=15)
```
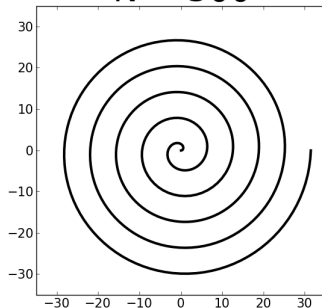
# 1d plots of spiral...

matplotlib plot command connects points with line segments...

# 1d plots of spiral...

```
for N in [50, 500]:
  theta = np.linspace(0., 10*np.pi, N)
  x = theta * np.cos(theta)
  y = theta * np.sin(theta)

  # x and y are NumPy arrays of length N
  plt.plot(x, y, 'k', linewidth=3)

  plt.axis('scaled')
  plt.xlim(-35,35);          plt.ylim(-35,35)
  plt.xticks(fontsize=15); plt.yticks(fontsize=15)

  plt.title('N = %s' % N, fontsize=20)
  plt.savefig('spiral%s.png' % N)
```
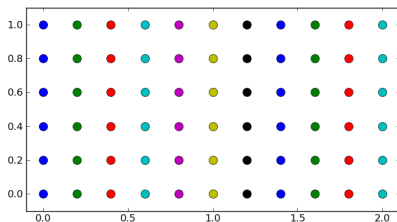
# 2d plots: defining grids in Python with `meshgrid`

```
m = 11;    n = 6
x = np.linspace(0, 2, m)
y = np.linspace(0, 1, n)

x2,y2 = np.meshgrid(x,y)
# These have shape (n,m) and                    !!!
#      (x2[i,j], y2[i,j]) is (x[j], y[i])       !!!

plt.plot(x2, y2, 'o', markersize=10)
# plots columns of x2 array vs. columns of y2
# (different color for each column)
```

# 2d plots: defining grids in Python

```
>>> x
array([ 1.,   2.,   3.,   4.,   5.])     # m=5

>>> y
array([ 10.,   20.,   30.])              # n=3

x2, y2 = np.meshgrid(x,y)

>>> x2
array([[ 1.,   2.,   3.,   4.,   5.],
       [ 1.,   2.,   3.,   4.,   5.],
       [ 1.,   2.,   3.,   4.,   5.]])

>>> y2
array([[ 10.,   10.,   10.,   10.,   10.],
       [ 20.,   20.,   20.,   20.,   20.],
       [ 30.,   30.,   30.,   30.,   30.]])

>>> (x[2], y[1])
(3.0, 20.0)

>>> (x2[1,2], y2[1,2])
(3.0, 20.0)
```
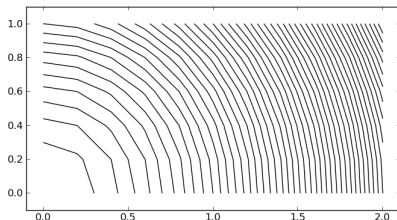
# 2d contour plots

```
x2,y2 = np.meshgrid(x,y)

# These have shape (n,m) = (6,11)          !!!
#         (x2[i,j], y2[i,j]) is (x[j], y[i])  !!!

# Function defined at grid points:
u = x2**2 + y2**2    # also has shape (6,11)!!!

levels = np.linspace(0,5,51)
plt.contour(x2, y2, u, levels, colors='k')
```

Note: Contour lines should be circular (but coarse grid!)

# 2d pcolor (pseudocolor) plots

```
x2,y2 = np.meshgrid(x,y)

# These have shape (n,m) = (6,11)

# Function defined at grid points:
u = x2**2 + y2**2   # also has shape (6,11)

plt.pcolor(x2, y2, u)
plt.colorbar(orientation='horizontal')
```
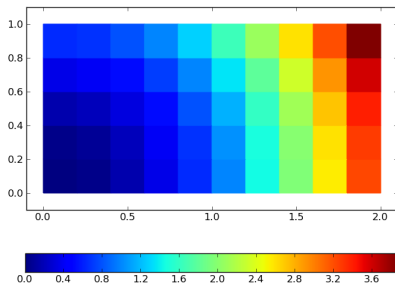
# 2d plots: using grids in Python

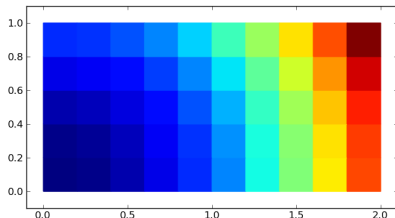Note: `x2,  y2` values are corners of cells on $m \times n$ grid.
  Colors are constant in $(n-1) \times (m-1)$ array of cells.

  Last row and column of `u` are ignored.

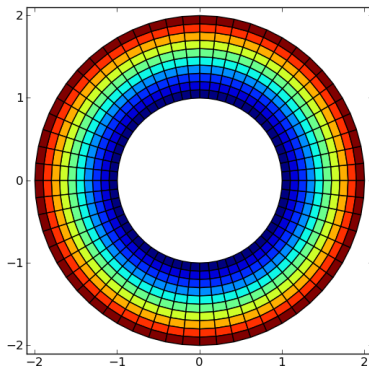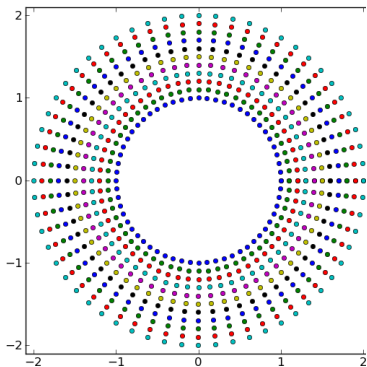For pcolor plots, better to define `u` at cell centers:

```
xmid = 0.5*(x[:-1] + x[1:])       # midpoints in x
ymid = 0.5*(y[:-1] + y[1:])       # midpoints in y
x2m,y2m = np.meshgrid(xmid,ymid)  # shape (m-1,n-1)

# Define a function of (x,y) at midpoints:
umid = x2m**2 + y2m**2
plt.pcolor(x2, y2, umid)          # original (x2,y2)
```

# Mapped grids — polar coordinates in annulus

Polar coordinate grid for $r \le r \le 2, \ 0 \le \theta \le 2\pi$

# Mapped grids — polar coordinates in annulus

```
m = 6;    n = 41

r = np.linspace(1, 2, m)
theta = np.linspace(0, 2*np.pi, n)
R,Theta = np.meshgrid(r,theta)

X = R * np.cos(Theta)
Y = R * np.sin(Theta)

U = X**2 + Y**2
plt.pcolor(X, Y, U, edgecolors='k')
```