Ch.5: Array computing and curve plotting

Joakim Sundnes[1,2] Hans Petter Langtangen[1,2]

Simula Research Laboratory[1]

University of Oslo, Dept. of Informatics[2]

Sep 27, 2017

Plan for week 39

Wednesday 27 september

- Live programming of ex 5.13, 5.29, 5.39
- Animations in matplotlib
- Making our own modules (from Chapter 4)

Friday 29 september

- Live programming of ex 5.39, A.1
- Programming of difference equations (Appendix A)

Quick recap 1: the plotting recipe

Plot the curve of $y(t) = t^2 e^{-t^2}$:

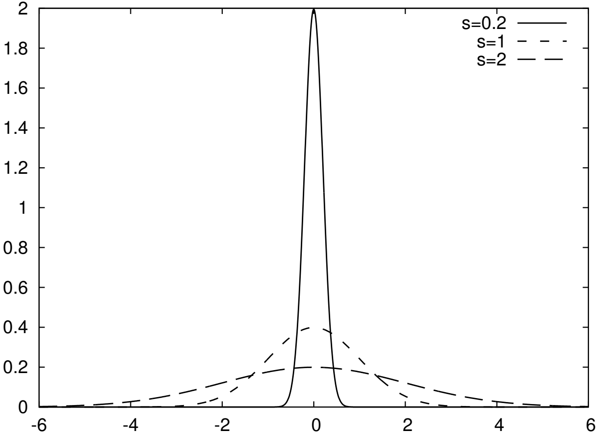
```
from matplotlib.pyplot import *
from numpy import *

# Make points along the curve
t = linspace(0, 3, 51)       # 50 intervals in [0, 3]
y = t**2*exp(-t**2)          # vectorized expression

xlabel('t')                           # label on the x axis
ylabel('y')                                 # label on the y axis
legend()                                  # mark the curve
title('My First Matplotlib Demo')
plot(t, y, label='t^2*exp(-t^2)'))

savefig('fig.pdf')                   # save figure as pdf
show()
```

Quick recap 2: minimal typing

Plotting code can be short. Here's a lazy version for plotting two curves in the same plot:

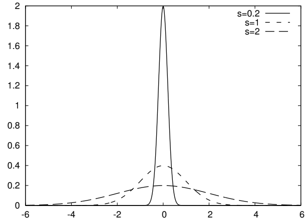
```
from matplotlib.pyplot import *
from numpy import *

t = linspace(0, 3, 51)
plot(t, t**2*exp(-t**2), t, t**4*exp(-t**2))
show()
```

Let's make a movie/animation



The Gaussian/bell function

$$f(x; m, s) = \frac{1}{\sqrt{2\pi}} \frac{1}{s} \exp\left[-\frac{1}{2} \left(\frac{x-m}{s} \right)^2 \right]$$



- m is the location of the peak
- s is a measure of the width of the function
- Make a movie (animation) of how $f(x; m, s)$ changes shape as s goes from 2 to 0.2

Movies are made from a (large) set of individual plots

- Goal: make a movie showing how $f(x)$ varies in shape as s decreases
- Idea: put many plots (for different s values) together (exactly as a cartoon movie)
- Very important: fix the y axis! Otherwise, the y axis always adapts to the peak of the function and the visual impression gets completely wrong

Three alternative recipes

1. Let the animation run *live*, without saving any files
   - Not possible to pause, slow down etc
2. Loop over all data values, plot and make a hardcopy (file) for each value, combine all hardcopies to a movie
   - Requires separate software (for instance *ImageMagick*) to see the animation
3. Use the 'Animate' function in 'matplotlib'
   - Plays the animation *live*
   - Relies on external software to save a movie file

Alt. 1: General idea

- Fix the axes!
- Use a 'for'-loop to loop over s-values
- Compute new y-values and update the plot for each run through the loop

Alt. 1: Complete code

```
from matplotlib.pyplot import *
from numpy import *

def f(x, m, s):
    return (1.0/(sqrt(2*pi)*s))*exp(-0.5*((x-m)/s)**2)

m = 0;  s_start = 2;  s_stop = 0.2
s_values = linspace(s_start, s_stop, 30)

x = linspace(m -3*s_start, m + 3*s_start, 1000)
# f is max for x=m (smaller s gives larger max value)
max_f = f(m, m, s_stop)

y = f(x,m,s_stop)
lines = plot(x,y)  #Returns a list of line objects!
axis([x[0], x[-1], -0.1, max_f])
xlabel('x')
ylabel('f')

for s in s_values:
    y = f(x, m, s)
    lines[0].set_ydata(y) #update plot data and redraw
    draw()
    pause(0.1)
```

Alt. 2: General idea

- Same 'for'-loop as alternative 1
- Use 'printf'-formatting to generate a unique file name for each plot
- Save file

Alt. 2: Complete code

```
from matplotlib.pyplot import *
from numpy import *

def f(x, m, s):
    return (1.0/(sqrt(2*pi)*s))*exp(-0.5*((x-m)/s)**2)

m = 0;  s_start = 2;  s_stop = 0.2
s_values = linspace(s_start, s_stop, 30)

x = linspace(m -3*s_start, m + 3*s_start, 1000)
max_f = f(m, m, s_stop)

y = f(x,m,s_stop)
lines = plot(x,y)
axis([x[0], x[-1], -0.1, max_f])

frame_counter = 0
for s in s_values:
    y = f(x, m, s)
    lines[0].set_ydata(y)
    draw()
    savefig('tmp_%04d.png' % frame_counter) #unique filename
    frame_counter += 1
```

How to combine plot files to a movie (video file)

We now have a lot of files:

`tmp_0000.png` `tmp_0001.png` `tmp_0002.png` ...

We use some program to combine these files to a video file:

- `convert` for animated GIF format (if just a few plot files)
- `ffmpeg` (or `avconv`) for MP4, WebM, Ogg, and Flash formats

Make and play animated GIF file

Tool: `convert` from the ImageMagick software suite.
Unix command:

```
Terminal> convert -delay 20 tmp_*.png movie.gif
```

Delay: 30/100 s, i.e., 0.5 s between each frame.
Play animated GIF file with `animate` from ImageMagick:

```
Terminal> animate movie.gif
```

or open the file in a browser.

Alt. 3: General idea

- Make two functions:
  - One for initialization of plot
  - One that updates the plot for each frame
- Make a list or array of the argument that changes (here s)
- Pass both functions and the list as arguments to the function `AnimateFunc`

Alt. 3: Complete code

```
from numpy import *
from matplotlib.pyplot import *
from matplotlib.animation import FuncAnimation

def f(x, m, s):
    return (1.0/(sqrt(2*pi)*s))*exp(-0.5*((x-m)/s)**2)

m = 0;  s_start = 2;  s_stop = 0.2
s_values = np.linspace(s_start, s_stop, 30)
x = np.linspace(m -3*s_start, m + 3*s_start, 1000)
max_f = f(m, m, s_stop)
lines = plot([],[]) #empty plot to create the lines object

def init():
    axis([x[0], x[-1], -0.1, max_f])
    lines[0].set_xdata(x)
    return lines

def update(frame):
    y = f(x, m, frame)
    lines[0].set_ydata(y)
    return lines

ani = FuncAnimation(gcf(), update, frames=s_values,
                    init_func=init, blit=True)
ani.save('test.gif')
show()
```

Notes on making movies

- Making actual movie files require external software such as `ImageMagick` or `ffmpeg`
- The software may be tricky to install (simple recipes exist, but don't always work)
- For the animation assignments in this course, you do not have to make movie files. You either:
  - Use Alt 1 or Alt 3 to make the animation run *live*
  - Use Alt 2 to create a lot of image files
- If you can also make the movie files this is great, but it will not be required

Making your own modules

We have frequently used modules like `math` and `sys`:

```
from math import log
r = log(6)    # call log function in math module

import sys
x = eval(sys.argv[1])  # access list argv in sys module
```

Characteristics of modules:

- Collection of useful data and functions (later also classes)
- Functions in a module can be reused in many different programs
- If you have some general functions that can be handy in more than one program, make a module with these functions
- It's easy: just collect the functions you want in a file, and that's a module!

Case on making our own module

Here are formulas for computing with interest rates:

$$A = A_0 \left(1 + \frac{p}{360 \cdot 100}\right)^n, \tag{1}$$

$$A_0 = A \left(1 + \frac{p}{360 \cdot 100}\right)^{-n}, \tag{2}$$

$$n = \frac{\ln \frac{A}{A_0}}{\ln \left(1 + \frac{p}{360 \cdot 100}\right)}, \tag{3}$$

$$p = 360 \cdot 100 \left(\left(\frac{A}{A_0}\right)^{1/n} - 1\right). \tag{4}$$

A_0: initial amount, p: percentage, n: days, A: final amount

We want to make a module with these four functions.

First we make Python functions for the formuluas

```
from math import log as ln

def present_amount(A0, p, n):
    return A0*(1 + p/(360.0*100))**n

def initial_amount(A, p, n):
    return A*(1 + p/(360.0*100))**(-n)

def days(A0, A, p):
    return ln(A/A0)/ln(1 + p/(360.0*100))

def annual_rate(A0, A, n):
    return 360*100*((A/A0)**(1.0/n) - 1)
```

Then we can make the module file

- Collect the 4 functions in a file `interest.py`
- Now `interest.py` is actually a module `interest` (!)

Example on use:

```
# How long time does it take to double an amount of money?

from interest import days
A0 = 1; A = 2; p = 5
n = days(A0, 2, p)
years = n/365.0
print('Money has doubled after %.1f years' % years)
```

Adding a test block in a module file

- Module files can have an if test at the end containing a *test block* for testing or demonstrating the module
- The test block is not executed when the file is imported as a module in another program
- The test block is executed *only* when the file is run as a program

```
if __name__ == '__main__': # this test defineds the test block
    <block of statements>
```

Test blocks are often collected in functions

We can put the test in a real *test function*, and call it from the test block:

```
def test_all_functions():
    # Define compatible values
    A = 2.2133983053266699; A0 = 2.0; p = 5; n = 730
    # Given three of these, compute the remaining one
    # and compare with the correct value (in parenthesis)
    A_computed = present_amount(A0, p, n)
    A0_computed = initial_amount(A, p, n)
    n_computed = days(A0, A, p)
    p_computed = annual_rate(A0, A, n)
    def float_eq(a, b, tolerance=1E-12):
        """Return True if a == b within the tolerance."""
        return abs(a - b) < tolerance

    success = float_eq(A_computed,  A)  and \
              float_eq(A0_computed, A0) and \
              float_eq(p_computed,  p)  and \
              float_eq(n_computed,  n)
    assert success  # could add message here if desired

if __name__ == '__main__':
    test_all_functions()
```

How can Python find our new module?

- If the module is in the same folder as the main program, everything is simple and ok
- Home-made modules are normally collected in a common folder, say `/Users/hpl/lib/python/mymods`
- In that case Python must be notified that our module is in that folder

Technique 1: add folder to `PYTHONPATH` in `.bashrc`:

```
export PYTHONPATH=$PYTHONPATH:/Users/hpl/lib/python/mymods
```

Technique 2: add folder to `sys.path` in the program:

```python
sys.path.insert(0, '/Users/hpl/lib/python/mymods')
```

Technique 3: move the module file in a directory that Python already searches for libraries.