Ch.6: Array computing and curve plotting

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Sep 18, 2022

0.1 Plan for week 38

Monday 19 september

- Short quiz
- Left from last week
  - Error handling with try-except
  - Making our own modules left as self-study
- Live programming of ex 4.5
- Intro to NumPy arrays and plotting
- Ex 5.9, 5.10, 5.11

Wednesday 21 september

- Short quiz
- Live programming of ex 5.7, 5.13, (4.6)
- Making movies and animations from plots

0.2 Repetition quiz (1)

What is printed when the following code is run?

```python
def f(x,y):
    return 2*x + y

x = 2
y = 3
print(f(1,2))
```

(Mid term exam 2018)

0.3 Repetition quiz (2)

What is printed when the following code is run?

```python
x = 4
y = 5
print(x > 4 and y > 4)
```

(Mid term 2019)

0.4 Repetition quiz (3)

The following code is in a file barometric.py:

```python
import sys
from math import exp

h = sys.argv[1]

p0 = 100
h0 = 8400
print(p0*exp(-h/h0))
```

When we run the code we get the following output. What is wrong?

```
python barometric.py 2469
Traceback (most recent call last):
  File "/Users/sundnes/Desktop/baro_test.py", line 8, in <module>
    print(p0*exp(-h/h0))
TypeError: bad operand type for unary -: 'str'
```

0.5 Recap from last week - user input

Alternative 1:

The function `input` makes the program stop and wait for user input:

```python
var = input('Please provide some input data:')
```

Simple and intuitive to use, slow and annoying in the long run

Alternative 2:

Use `sys.argv` to access *command line arguments*:

```
import sys
var = sys.argv[1]
```

Run the program from the terminal:

```
python myprog.py 2.05
```

Or in iPython/Spyder:

```
run myprog.py 2.05
```

0.6 Recap from last week - file read/write

Reading from a file:

```
with open('myfile.txt','r') as infile:
    l = infile.readline() #read a single line

    for line in infile:
        words = line.split()
        var = float(words[-1])  # etc
```

Write to a file:

```
data = [...]
with open('myfile.txt','w') as outfile:
    for myvar in data:
        outfile.write(myvar)
        outfile.write('\n')  #linebreak
```

0.7 Left from last week - error handling with try-except

- Rather than test *if something is wrong, recover from error, else do what we indended to do*, it is common in Python (and many other languages) to *try* to do what we indend to, and if it fails, we recover from the error
- This principle makes use of a `try-except` block

```
try:
    <statements we intend to do>
except:
    <statements for handling errors>
```

If something goes wrong in the `try` block, Python raises an *exception* and the execution jumps immediately to the `except` block.

0.8 The barometric program with try-except

Try to read h from the command-line, if it fails, tell the user, and abort execution:

```
import sys
try:
    h = float(sys.argv[1])
except:
    print('You failed to provide a command line arg.!')
    exit()

p0 = 100.0; h0 = 8400
print(p0 * exp(-h/h0))
```

Execution:

```
Terminal> python altitude_cml_except1.py
You failed to provide a command line arg.!

Terminal> python altitude_cml_except1.py 2469m
You failed to provide a command line arg.!
```

0.9 Improvement: test for specific exceptions

It is good programming style to test for specific exceptions:

```
try:
    h = float(sys.argv[1])
except IndexError:
    print 'You failed to provide a command-line arg.!'
```

If we have an index out of bounds in `sys.argv`, an `IndexError` exception is raised, and we jump to the `except` block.

If any other exception arises, Python aborts the execution:

```
Terminal> python altitude_cml_except1.py 2469m
Traceback (most recent call last):
  File "altitude.py", line 3, in <module>
    C = float(sys.argv[1])
ValueError: invalid literal for float(): 2469m
```

0.10 Improvement: test for `IndexError` and `ValueError` exceptions

```
import sys
try:
    h = float(sys.argv[1])
except IndexError:
    print('No command line argument for h!')
    sys.exit(1)  # abort execution
except ValueError:
    print(f'h must be a pure number, not {sys.argv[1]}')
    exit()

p0 = 100.0; h0 = 8400
print(p0 * exp(-h/h0))
```

Executions:

```
Terminal> python altitude.py
No command line argument for h!

Terminal> python altitude.py 2469m
The altitude must be a pure number, not "2469m"
```

0.11 The programmer can raise exceptions

- Instead of just letting Python raise exceptions, we can raise our own and tailor the message to the problem at hand
- We provide two examples on this:
  - catching an exception, but raising a new one with an improved (tailored) error message
  - raising an exception because of wrong input data
- Basic syntax: `raise ExceptionType(message)`

0.12 Examples on re-raising exceptions with better messages

```python
def read_altitude():
    try:
        h = float(sys.argv[1])
    except IndexError:
        # re-raise, but with specific explanation:
        raise IndexError(
          'The altitude must be supplied on the command line.')
    except ValueError:
        # re-raise, but with specific explanation:
        raise ValueError(
         f'Altitude must be number, not "{sys.argv[1]}".')

    # h is read correctly as a number, but has a wrong value:
    if h < -430 or h > 13000:
        raise ValueError(f'The formula is not valid for h={h}')
    return h
```

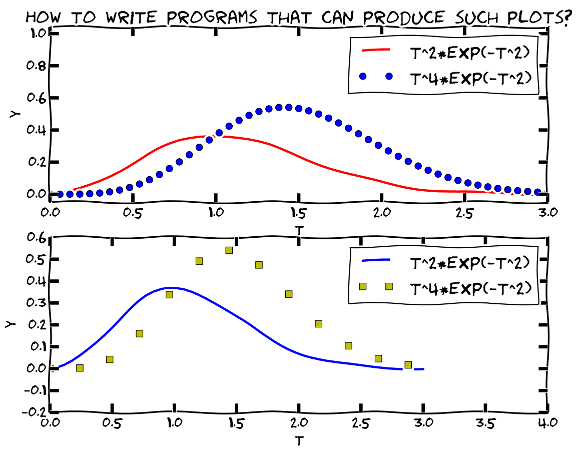
0.13 Calling the previous function and running the program

```
try:
    h = read_altitude()
except (IndexError, ValueError) as e:
    # print exception message and stop the program
    print(e)
    exit()
```

Executions:

```
Terminal> python altitude_cml_except2.py
The altitude must be supplied on the command line.

Terminal> python altitude_cml_except2.py 1000m
Altitude must be number, not 1000m.

Terminal> python altitude_cml_except2.py 20000
The formula is not valid for h=20000.

Terminal> python altitude_cml_except2.py 8848
34.8773231887747
```

0.14 Goal: learn to visualize functions



0.15 We need to learn about a new object: array

- Curves $y = f(x)$ are visualized by drawing straight lines between points along the curve
- Need to store the coordinates of the points along the curve in lists or *arrays* x and y
- Arrays ≈ lists, but computationally much more efficient

- To compute the `y` coordinates (in an array) we need to learn about *array computations* or *vectorization*
- Array computations are useful for much more than plotting curves!

0.16 The minimal need-to-know about vectors

- Vectors are known from high school mathematics, e.g., point (x, y) in the plane, point (x, y, z) in space
- In general, a vector v is an n-tuple of numbers: $v = (v_0, \ldots, v_{n-1})$
- Vectors can be represented by lists: v_i is stored as `v[i]`, but we shall use arrays instead

0.17 Arrays can have more than one index

Just as nested lists, arrays can have multiple indices: $A_{i,j}$, $A_{i,j,k}$
Example: table of numbers, one index for the row, one for the column

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix} \qquad A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

- The no of indices in an array is the *rank* or *number of dimensions*
- Vector = one-dimensional array, or rank 1 array
- In Python code, we use Numerical Python arrays instead of nested lists to represent mathematical arrays (because this is computationally more efficient)

0.18 Storing (x,y) points on a curve in lists

Collect points on a function curve $y = f(x)$ in lists:

```
def f(x):
    return x**3

n = 5                   # no of points
dx = 1.0/(n-1)          # x spacing in [0,1]

for i in range(n):
    x.append(i*dx)
    y.append(f(x))

#turn lists into NumPy arrays
import numpy as np        # module for arrays
x = np.array(xlist)      # turn list xlist into array
y = np.array(ylist)
```

0.19 Make arrays directly (instead of lists)

Or drop the lists and make NumPy arrays directly:

```
>>> n = 5                     # number of points
>>> x = np.linspace(0, 1, n) # n points in [0, 1]
>>> y = np.zeros(n)          # n zeros (float data type)
>>> for i in range(n):
...     y[i] = f(x[i])
...
```

0.20 Arrays are not as flexible as list, but computationally much more efficient

- List elements can be *any* Python objects
- Array elements can only be of *one object type*
- Arrays are very efficient to store in memory and compute with if the element type is `float`, `int`, or `complex`
- Rule: use arrays for sequences of numbers!

0.21 We can work with entire arrays at once - instead of one element at a time

Compute the sine of an array:

```
from math import sin

for i in range(len(x)):
    y[i] = sin(x[i])
```

However, if `x` is array, `y` can be computed by

```
import numpy as np
y = np.sin(x)                 # x: array, y: array
```

The loop is now inside `np.sin` and implemented in very efficient C code.

Vectorization gives:

- shorter, more readable code, closer to the mathematics
- much faster code

0.22 A function `f(x)` written for a number `x` usually works for array `x` too

```
from numpy import sin, exp, linspace

def f(x):
    return x**3 + sin(x)*exp(-3*x)

x = 1.2                     # float object
y = f(x)                    # y is float

x = linspace(0, 3, 10001)   # 10000 intervals in [0,3]
y = f(x)                    # y is array
```

0.23 NOTE: `math` is for numbers and `numpy` for arrays

```
>>> import math, numpy
>>> x = numpy.linspace(0, 1, 11)
>>> math.sin(x[3])
0.2955202066613396
>>> math.sin(x)
...
TypeError: only length-1 arrays can be converted to Python scalars
>>> numpy.sin(x)
array([ 0.      ,  0.09983,  0.19866,  0.29552,  0.38941,
        0.47942,  0.56464,  0.64421,  0.71735,  0.78332,
        0.84147])
```

0.24 Very important application: vectorized code for computing points along a curve

$$f(x) = x^2 e^{-\frac{1}{2}x} \sin(x - \frac{1}{3}\pi), \quad x \in [0, 4\pi]$$

Vectorized computation of $n+1$ points along the curve.

```
import numpy as np

n = 100
x = np.linspace(0, 4*pi, n+1)
y = 2.5 + x**2*np.exp(-0.5*x)*np.sin(x-pi/3)
```

0.25 New term: vectorization

- *Scalar*: a number
- *Vector* or *array*: sequence of numbers (vector in mathematics)
- We speak about scalar computations (one number at a time) versus vectorized computations (operations on entire arrays, no Python loops)

- *Vectorized functions* can operate on arrays (vectors)
- *Vectorization* is the process of turning a non-vectorized algorithm with (Python) loops into a vectorized version without (Python) loops
- Mathematical functions in Python without `if` tests automatically work for both scalar and vector (array) arguments (i.e., no vectorization is needed by the programmer)

0.26 Small quiz:

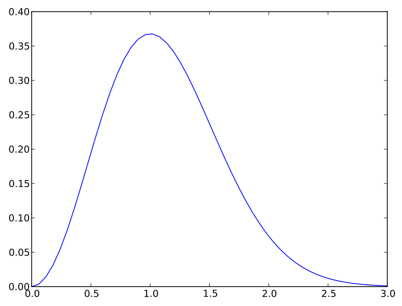
What is output from the following code? Why?

```
import numpy as np

l = [0,0.25,0.5,0.75,1]
a = np.array(l)

print(l*2)
print(a*2)
```

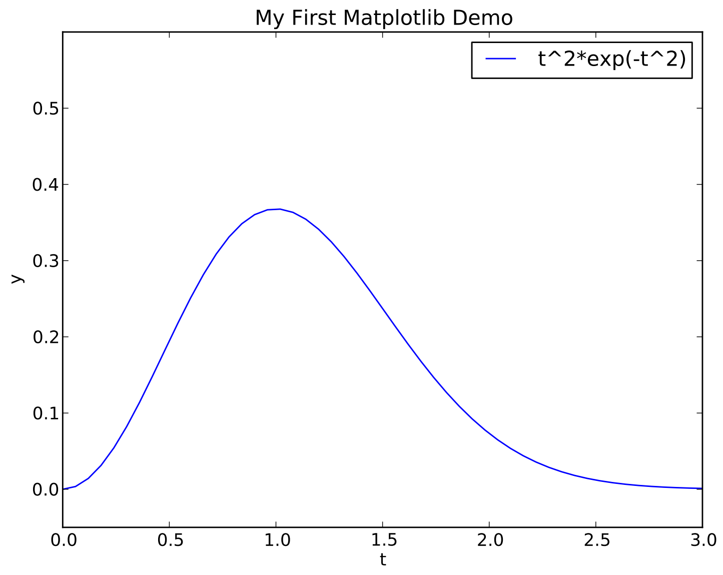
0.27 Plotting the curve of a function: the very basics

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

```
import matplotlib.pyplot as plt  # import and plotting
import numpy as np

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

plt.plot(t, y)             # make plot on the screen
plt.savefig('fig.pdf')     # make PDF image for reports
plt.savefig('fig.png')     # make PNG image for web pages
plt.show()
```



0.28 A plot should have labels on axis and a title



0.29 The code that makes the last plot

```
import matplotlib.pyplot as plt
import numpy as np

def f(t):
    return t**2*np.exp(-t**2)
```

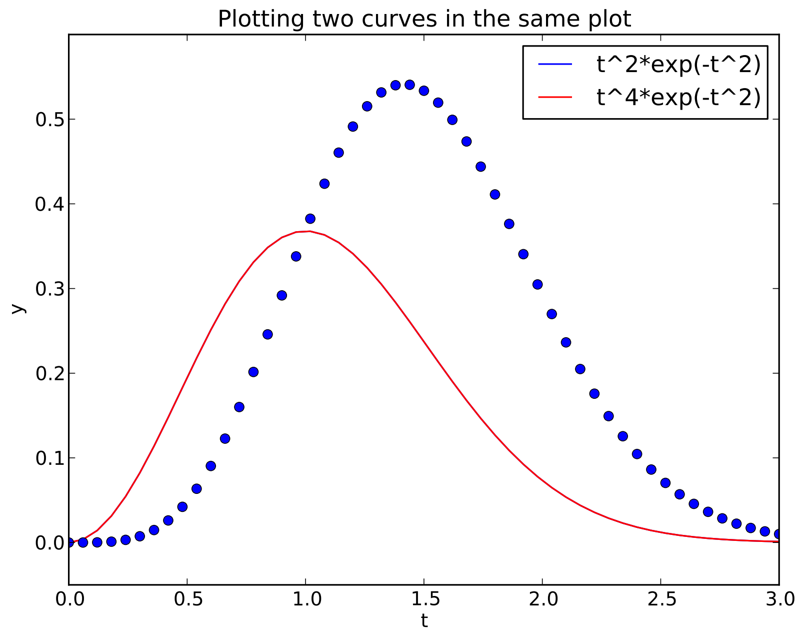
```
t = np.linspace(0, 3, 51)    # t coordinates
y = f(t)                     # corresponding y values

plt.plot(t, y,label="t^2*exp(-t^2)")

plt.xlabel('t')               # label on the x axis
plt.ylabel('y')               # label on the y axix
plt.legend()                  # mark the curve
plt.axis([0, 3, -0.05, 0.6])  # [tmin, tmax, ymin, ymax]
plt.title('My First Matplotlib Demo')
plt.show()
```

0.30 Plotting several curves in one plot

Plot $t^2e^{-t^2}$ and $t^4e^{-t^2}$ in the same plot:

```
import matplotlib.pyplot as plt
import numpy as np

def f1(t):
    return t**2*np.exp(-t**2)

def f2(t):
    return t**2*f1(t)

t = np.linspace(0, 3, 51)
y1 = f1(t)
y2 = f2(t)

plt.plot(t, y1,  'r-', label = 't^2*exp(-t^2)')
plt.plot(t, y2, 'bo', label = 't^4*exp(-t^2)')

plt.xlabel('t')
plt.ylabel('y')
plt.legend()
plt.title('Plotting two curves in the same plot')
plt.savefig('tmp2.png')
plt.show()
```

0.31 The resulting plot with two curves



0.32 Controlling line styles

When plotting multiple curves in the same plot, the different lines (normally) look different. We can control the line type and color, if desired:

```
plot(t, y1, 'r-')    # red (r) line (-)
plot(t, y2, 'bo')    # blue (b) circles (o)

# or
plot(t, y1, 'r-', t, y2, 'bo')
```

Documentation of colors and line styles, see the online Matplotlib documentation or

```
Unix> pydoc matplotlib.pyplot
```

0.33 Quick plotting with minimal typing

A lazy pro would do this:

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

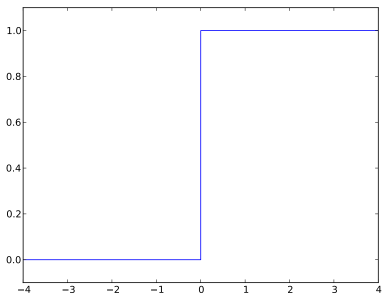
0.34 Example: plot a discontinuous function

The Heaviside function is frequently used in science and engineering:

$$H(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

Python implementation:

```
def H(x):
    if x < 0:
        return 0
    else:
        return 1
```



0.35 Plotting the Heaviside function: first try

Standard approach:

```
x = np.linspace(-10, 10, 5)  # few points (simple curve)
y = H(x)
plt.plot(x, y)
```

First problem: `ValueError` error in `H(x)` from `if x < 0`
Let us debug in an interactive shell:

```
>>> x = np.linspace(-10,10,5)
>>> x
array([-10.,  -5.,   0.,   5.,  10.])
>>> b = x < 0
>>> b
array([ True,  True, False, False, False], dtype=bool)
>>> bool(b)  # evaluate b in a boolean context
...
ValueError: The truth value of an array with more than
one element is ambiguous. Use a.any() or a.all()
```

0.36 `if x < 0` does not work if `x` is array

Remedy 1: use a loop over `x` values.

```
def H_loop(x):
    r = zeros(len(x))  # or r = x.copy()
    for i in range(len(x)):
        r[i] = H(x[i])
    return r

n = 5
x = np.linspace(-5, 5, n+1)
y = H_loop(x)

#or loop over x and call the original function
y = np.zeros_like(x)
for i in range(len(x)):
    y[i] = H(x[i])
```

Downside: much to write, slow code if `n` is large

0.37 `if x < 0` does not work if `x` is array

Remedy 2: use `numpy.vectorize`.

```
# Automatic vectorization of function H
Hv = np.vectorize(H)
# Hv(x) works with array x
```

Downside: The resulting function is as slow as Remedy 1

0.38 `if x < 0` does not work if `x` is array

Remedy 3: code the `if` test differently.

```
def Hv(x):
    return np.where(x < 0, 0.0, 1.0)
```

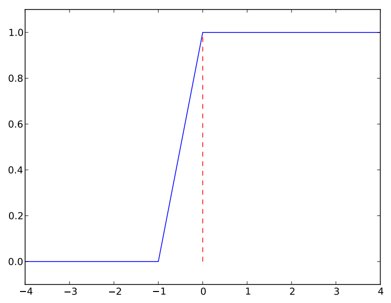
More generally:

```
def f(x):
    if condition:
        x = <expression1>
    else:
        x = <expression2>
    return x

def f_vectorized(x):
    x1 = <expression1>
    x2 = <expression2>
    r = np.where(condition, x1, x2)
    return r
```

0.39 Back to plotting the Heaviside function

With a vectorized `Hv(x)` function we can plot in the standard way

```
x = linspace(-10, 10, 5)    # linspace(-10, 10, 50)
y = Hv(x)
plot(x, y, axis=[x[0], x[-1], -0.1, 1.1])
```



0.40 How to make the function look discontinuous in the plot?

We could use a lot of x points to make the curve look steeper, but it does still not really look like a discontinuous function.

Question. How can we make the plot look like a proper discontinuous function?

0.41 Example: Plot function given on the command line

Task: plot function given on the command line.

```
Terminal> python plotf.py expression xmin xmax
Terminal> python plotf.py "exp(-0.2*x)*sin(2*pi*x)" 0 4*pi
```

Should plot $e^{-0.2x}\sin(2\pi x)$, $x \in [0, 4\pi]$. `plotf.py` should work for "any" mathematical expression.

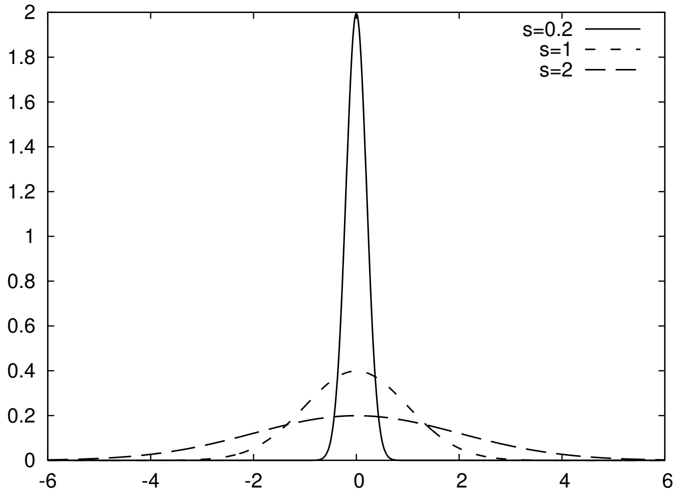
0.42 Solution

Complete program:

```
from numpy import *
import matplotlib.pyplot as plt
import sys

formula = sys.argv[1]
xmin = eval(sys.argv[2])
```

```
xmax = eval(sys.argv[3])

x = linspace(xmin, xmax, 101)
y = eval(formula)
plt.plot(x, y)
plt.title(formula)
plt.show()
```

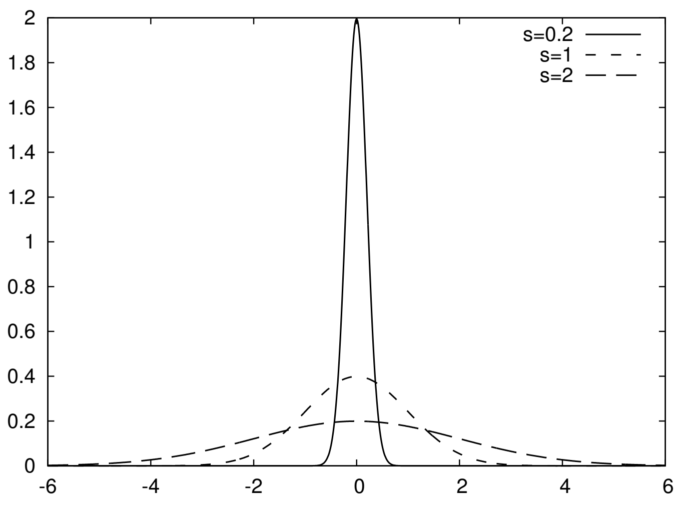
0.43 Let's make a movie/animation



0.44 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



0.45 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

0.46 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 a 'FuncAnimation' object from 'matplotlib'
   - Plays the animation *live*
   - Relies on external software to save a movie file

0.47 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

0.48 Alt. 1: Complete code

```python
import matplotlib.pyplot as plt
import numpy as np

def f(x, m, s):
    return (1.0/(np.sqrt(2*np.pi)*s))*np.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)
# 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 = plt.plot(x,y)  #Returns a list of line objects!

plt.axis([x[0], x[-1], -0.1, max_f])
plt.xlabel('x')
plt.ylabel('f')

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

0.49 Alt. 2: General idea

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

0.50 Alt. 2: Complete code

```
import matplotlib.pyplot as plt
import numpy as np

def f(x, m, s):
    return (1.0/(np.sqrt(2*np.pi)*s))*np.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)
# 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 = plt.plot(x,y)

plt.axis([x[0], x[-1], -0.1, max_f])
plt.xlabel('x')
plt.ylabel('f')

frame_counter = 0
for s in s_values:
    y = f(x, m, s)
    lines[0].set_ydata(y) #update plot data and redraw
    plt.draw()
    plt.savefig(f'tmp_{frame_counter:04d}.png') #unique filename
    frame_counter += 1
```

0.51 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

0.52 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.

0.53 Alt. 3: General idea

- Make a function to update the plot:
  - Updates the plot by calculating values and calling `set_ydata`
  - (Optional function to initialize the plot)
- Make a list or array of the argument that changes (here s)
- Pass the function and the list as arguments to create a `FuncAnimation` object
- Use functions in that object to animate, save a movie file etc.

0.54 Alt. 3: Complete code

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

def f(x, m, s):
    return (1.0/(np.sqrt(2*np.pi)*s))*np.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(-3*s_start,3*s_start, 1000)

max_f = f(m,m,s_stop)

plt.axis([x[0],x[-1],0,max_f])
plt.xlabel('x')
plt.ylabel('y')

y = f(x,m,s_start)
lines = plt.plot(x,y) #initial plot to create the lines object

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

ani = FuncAnimation(plt.gcf(), next_frame, frames=s_values, interval=100)
ani.save('movie.mp4',fps=20)
plt.show()
```

0.55 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