Ch.5: Array computing and curve plotting

Joakim Sundnes[1,2]

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

Sep 13, 2019

The main goal for Chapter 5 is to learn to visualize mathematical functions and results of mathematical calculations. You have probably used a variety of plotting tools earlier, and we will now do much of the same thing in Python. The standard way of plotting a curve in Python, and many other programming languages, is to first compute a number of points lying on the curve, and then draw straight lines between the points. If we have enough points, the result looks like a smooth curve. For plotting mathematical functions this approach may seem a bit primitive, since there are plenty of other tools where we can simply type in a mathematical expression and get the curve plotted on teh screen. However, the approach of Python is also much more flexible, since we can plot data where there is no underlying mathematical function, for instance experimental data read from a file or results from a numerical experiment. To be able to plot functions in Python we need to learn two new tool boxes; `numpy` for storing *arrays* of data for efficient computations, and `matplotlib`, which is an extensive toolbox for plotting and visualization in Python.

1 Numpy for array computing

The standard way to plot a curve $y = f(x)$ is to draw straight lines between points along the curve, and for this purpose we need to store the coordinates of the points. We could use lists for this, for instance two lists `x` and `y`, and many of the plotting tools we will use actually work fine with lists. However, a data structure known as an *array* is much more efficient than the list, and also offers a number of other features and advantages. Computing with arrays is often referred to as *array computations* or *vectorized computations*, and these concepts are useful for much more than just plotting curves.

Arrays are generalizations of vectors. In high-school mathematics, vectors were introduced as line segments with a direction, represented by coordinates

(x, y) in the plane or (x, y, z) in space. This concept of vectors can be generalized to any number of dimensions, and we may view a vector v is a general n-tuple of numbers; $v = (v_0, \ldots, v_{n-1})$. In Python, we could use a list to represent such a vector, by storing component v_i as element `v[i]` in the list. However, vectors are so useful and common in scientific programming that a special datastructure has been created for them; the *Numpy array*. An array is much less flexible than a list, in that it has a fixed length (i.e. no `append`-method), and a single array can only hold one type of variables. But arrays are also much more efficient to use in computations, and since they are designed for use in computations they have a number of useful features that can make the code shorter and more readable.

For the purpose of plotting we will mostly use one-dimensional arrays, but an array can have multiple indices, similar to a nested list, For instance, a two-dimensional array $A_{i,j}$ can be viewed as a table of numbers, with one index for the row and one for the column

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

Such a two-dimensional case is similar to a matrix in linear algebra, but arrays do not follow the standard rules for mathematical operations on matrices. There are good reasons for this, which we will explain below. The number of indices in an array is often referred to as the *rank* or the *number of dimensions*.

Storing (x,y) points on a curve in lists and arrays. To make the array concept a bit more concrete, consider a typical task where we want to store the points on a function curve $y = f(x)$. All the plotting cases we will consider in this chapter will be based on this idea, so it makes sense to introduce it for a simple example. As we have seen in previous chapters, there are multiple ways to store such pairs of numbers, including nested lists containing (x, y) pairs. However, for the purpose of plotting the easiest is to create two lists or arrays, one holding the x-values and another holding the y-values. The two lists should have equal length, and we will always create them using the same recipe. First we create a list/array of n uniformly spaced x-values, which cover the interval where we want to plot the function. Then, we run through these points and compute the corresponding y-points, and store these in a separate list or array. We can start with lists, since we already know how to use them. The following interactive Python session illustrates how we can use list comprehensions to create first a list of 5 x-points on the interval $[0, 1]$, and then compute the corresponding points $y = f(x)$ for $f(x) = x^3$.

```
>>> def f(x):
...     return x**3
...
>>> n = 5                    # no of points
>>> dx = 1.0/(n-1)           # x spacing in [0,1]
>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]
```

Now that we have the two lists, they could be sent directly to a tool like `matplotlib` for plotting, but before we do this we will introduce NumPy arrays. As noted above, arrays are designed particularly for scientific computations, and have a number of features that make them convenient to use. If we continue the interactive session from above, the following lines turn the two lists into NumPy arrays:

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

It is worth noticing how we import numpy in the first line. As always, we could import with `from numpy import *`, but this is a bad habit since `numpy` and `math` contain many functions with the same name, and we will often use both modules in the same program. To ensure that we always know which module we are using, it is a good habit to import NumPy as we have done here. Using `import numpy as np` instead of simply `import numpy` saves us some typing in the rest of the code, and is also more or less an accepted standard in the field.

Converting lists to arrays using the `array` function from NumPy is intuitive and flexible, but NumPy has a number of builtin functions that are much more convenient to use. Two of the most widely used ones are called `linspace` and `zeros`. The following interactive session is a list-free version of the example above, where we create the NumPy arrays directly using `linspace` and `zeros`.

```
>>> import numpy as np
>>> def f(x):
...     return x**3
...
>>> 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])
...
```

As illustrated here, we will usually call `linspace` with three arguments, with the general form `linspace(start,stop,n)`, which will create an array of length `n`, containing unfiformly distributed values on the interval from `start` to `stop`. If we leave out the third argument, as in `linspace(start,stop)`, a default value of `n=50` is used. The `start` and `stop` arguments must always be provided. An array of equally spaced x-values is needed nearly every time we plot something, so we will use `linspace` frequently. It is worth spending some time to get familiar with how it is used and what it returns. The second NumPy function above, `zeros(n)`, does exactly what we would expect, it creates an array of length `n` containing only zeros. We have seen earlier that a common way to create a list is to start with an empty list and fill it with values using a for-loop and the `append`-method. We will often use a similar approach to create an array, but since an array has fixed length and no `append`-method, we first create an array filled with zeros and then index into the array to fill it with the values we need. We often need to create an array of zeros, so remembering that the `zeros` function exists in NumPy is important.

As we saw in Chapter 2, lists in Python are extremely flexible, and can contain any Python object. Arrays are much more static, and we will typically use them for numbers, of type `float` or `int`. They can also be of other types, for instance boolean arrays (True/False), but a single array always contains a single object type. We have also seen that arrays are of fixed length, they don't have the convenient `append`-method, so why do we use arrays at all? One reason, which was mentioned above, is that arrays are more efficient to store in memory and to use in computations. The other reason is that arrays can make our code shorter and more readable, since we can perform operations on an entire array at once instead of using loops. Say for instance that we want to compute the sine of all elements in a list or array `x`. We know how to do this using a for-loop

```
from math import sin

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

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

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

In addition to being shorter and quicker to write, this code will run much faster than the code with the loop. Such computations are usually referred to as vectorized computations, since they work on the entire array (or vector) at once. Most of the standard functions we find in `math` have a corresponding function in `numpy` that will work for arrays. Under the hood these NumPy functions still contain a for-loop, since they need to traverse all the elements of the array, but this loop is written in very efficient C code and is therefore much faster than the Python loops.

A function `f(x)` which was written to work a for a single number `x`, will often work fine for an array too. If the function only uses basic mathematical operators (+,'-', etc.), we can pass it either a number or an array as argument and it will work just fine with no modifications. If the function uses more advanced operations that we need to import, we have to make sure to import these from `numpy` rather than `math`, since the functions in `math` only work with single numbers. The following example illustrates how it works.

```
from numpy import sin, exp, linspace

def g(x):
    return x**3+2*x**2-3

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, 101)     # 100 intervals in [0,3]
y = f(x)                    # y is array
z = g(x)                    # z is array
```

We see that, except for the initial import from NumPy, the two functions look exactly the same as if they were written to work on a single number. The result of the two function calls will be two arrays `y,z` of length 101, with each element being the function value computed for the corresponding value of `x`.

If we try to send an array with length > 1 to a function imported from `math`, we will get an error message:

```
>>> 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])
```

On the other hand, using NumPy functions on single numbers will work just fine. A natural question to ask is then why we ever need to import fram `math` at all. Why not use NumPy functions all the time, since they do the job both for arrays and numbers? The answer is that we can certainly do this, and in most cases it works fine, but the functions in `math` are more optimized for single numbers (scalars) and therefore faster. One will rarely notice the difference, but there may be certain application where this extra efficiency matters. There are also some functions in `math` (for instance `factorial`) which do not have a corresponding array version in NumPy.

Above we introduced the very important application of computing points a long a curve. We started out using lists and for-loops, but it is much easier to solve this task using NumPy. Say we want to compute points on the curve described by the function

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

for $x \in [0, 4 * \pi]$. The vectorized code may look as follows:

```
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)
```

This code is shorter and quicker to write than the one with lists and loops, most people find it easier to read since it is closer to the mathematics, and it runs much faster than the list version.

We have already mentioned the term *vectorized computations*, and later in the course (or elsewhere) you will probably at some point be asked to *vectorize* a function or a computation in a code. This usually means nothing more than to ensure that all mathematical functions are imported from `numpy` rather than `math`, and then to do all operations on entire arrays rather than looping over their

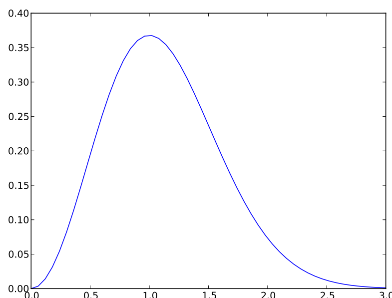
individual elements. The vectorized code should contain no for-loops written in Python. The example above involving the mathematical functions `g(x)` and `f(x)` provide a perfect example of vectorized functions, even though the actual functions look identical to the scalar versions. The only major exceptions to this simple recipe for vectorization are functions that include if-tests. For instance, in Chapter 3 we implemented piecewisely defined mathematical functions using if-tests. These functions will not work if the input argument is an array, because tests like `if x > 0` do not work for arrays. There are ways to solve this problem, which we will look into later in the chapter.

2 Plotting the curve of a function: the very basics

The motivation for introducing NumPy arrays was to plot mathematical functions, but so far we have not really plotted much. To start with a simple example, say we want to plot the curve $y(t) = t^2 e^{-t^2}$, for t ranging from 0 to 3. The code looks like this:

```
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.show()
```

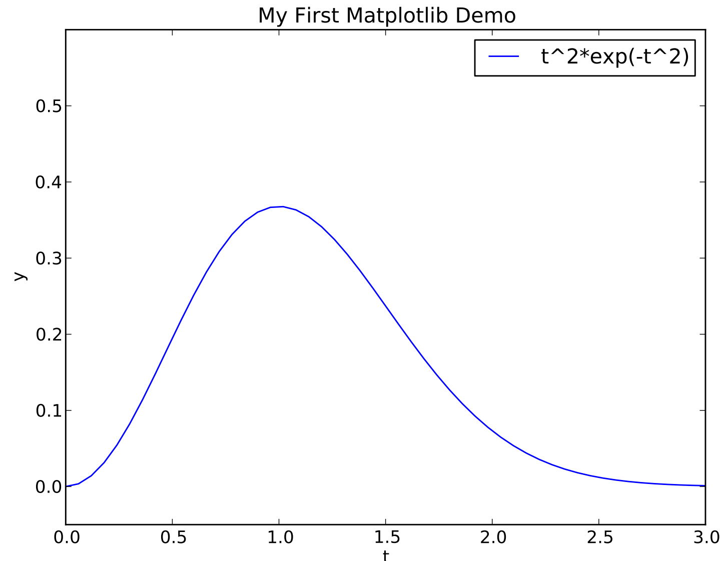
The first line imports the plotting tools from the `matplotlib` package, which is an extensive library of functions for scientific visualization. We will only use a small subset of the capabilities of `matplotlib`, mainly to plot curves and to create animations of curves that change over time. Next we import `numpy` for array-based computations, and then the two first lines of real code create arrays containing uniformly spaced t-values and corresponding values of y. The creation of the arrays follows the recipe outlined above, using `linspace` and then a vectorized calculation of the `y`-values. The last two lines will do the actual plotting; the call `plt.plot(x,y)` first creates the plot of the curve, and then `plt.show()` displays the plot on the screen. The reason for keeping these separate is that it makes it easy to plot multiple curves in a single plot, by calling `plot` multiple times followed by a single call to `show`. A common mistake is to forget the `plt.show()` call, and then the program will simply end without displaying anything on the screen.



The plot produced by the code above is very simple, and contains no title, axis labels, or other information. We can easily add such information the plot using tools from `matplotlib`:

```
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.savefig('fig.pdf')   # make PDF image for reports
plt.savefig('fig.png')   # make PNG image for web pages
plt.show()
```

Most of the lines in this code should be self-explanatory, with a couple of exceptions. The call to legend will create a legend for the plot, using the information provided in the `label` argument passed to `plt.plot`. This is very useful when plotting multiple curves in a single plot. The `axis` function will set the length of the horisontal and vertical axes. These are otherwise set automatically to default, which usually works fine, but in some cases the plot looks better if we set the axes manually. Later in this chapter we will create animations of curves, and then it will be essential to set the axes to fixed lengths. Finally, the two calls to `savefig` will save our plot in two different formats, automatically determined by the file name.



As noted above, we can plot multiple curves in a single plot. In that case Matplotlib will choose the color of each curve automatically, and this usually works well, but we can control the look of each curve if we want to. Say we want to plot the functions $t^2e^{-t^2}$ 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)') # red (r) line (-)
plt.plot(t, y2, 'bo', label = 't^4*exp(-t^2)')  # blue (b) circles (o)

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

From this example we can see that the options for changing the color and plotting style of curves are fairly intuitive, and can easily be explored by trial

and error. For a full overview of all the options we refer to the Matplotlib documentation.

Although the code example above was not too complex, we had to write an excess of 20 lines just to plot two simple functions on the screen. This level of programming is needed if we want to produce professional-looking plots, for instance for use in a presentation, a Master's thesis or a scientific report. However, if we just want a quick plot on the screen it can be done quicker. The following code lines will plot the same two curves as in the example above, using just three lines: that can be used in a scien

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

As always, the effort we put in depends on what the resulting plot will be used for, and in particular on whether we are just exploring some data on our own or plan on presenting it to others.

Example: Plot a function specified on the command line. Say we want to write a small program `plotf.py` that allows a user to speficy a mathematical function $f(x)$ as a command line argument, and then plots the curve $y = f(x)$. We may assume that the user should also specify the boundaries of the curve, i.e., the lower and upper limit for x. The general use of the program in the terminal should be

```
Terminal> python plotf.py expression xmin xmax
```

For instance, the command

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

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

We saw in the previous chapter how we could combine the `sys.argv` list of arguments with `exec` to build a Python function for a mathematical expression. The task at hand can be solved by building on this approach, and simply adding some statemments for plotting the function at the end. However, we can make an even simpler version, where we use `eval` to evaluate the expression directly, without even creating a Python function. The complete code may look like this:

```
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.show()
```

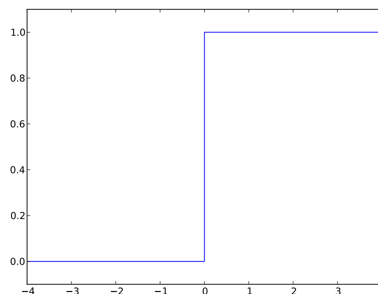
This small program will plot any formula provided as a command-line argument. Note that in this case we have a good reason to import Numpy with `from numpy import *`. We want the user to be able type a formula using standard mathematical terminology, such as `sin(x) + x**2` (rather than `np.sin(x) + x**2`). For this to work we need to import all mathematical functions from Numpy with no prefix.

Plotting discontinuous and piecewisely defined functions. Discontinuous functions, and functions defined in a piecewise manner, are common in science and engineering. We saw in Chapter 3 how these could be implemented in Python using if-tests, but as we briefly commented above this implementation gives rise to some challenges when using arrays and Numpy. To consider a concrete example, say we want to plot the Heaviside function, defined by

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

Following the ideas from Chapter 3, a Python implementation of this function may look like this

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



Now we want to plot the function using the simple approach introduced above. We would then simply create an array of values `x`, and pass this array to the function `H(x)` to compute the corresponding y-values:

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

However, if we try to run this code we get an error message, a `ValueError` error inside the function `H(x)`, coming from the `if x < 0` line. We can illustrate what goes wrong in an interactive Python session:

```
>>> x = 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()
```

We see here that the result of the statement `b = x < 0` is an array of boolean values, whereas if `b` was a single number the result would be a single boolean (True/False). Therefore, the statement `bool(b)`, or tests like `if b` or `if x < 0` do not make sense, since it is impossible to say whether an array of multiple True/False values is true or false.

There are several ways to fix this problem. One is to avoid the vectorization altogether, and go back to the traditional for-loop for computing the values:

```
import numpy as np
import matplotlib.pyplot as plt
n = 5
x = np.linspace(-5, 5, n+1)
y = np.zeros(n+1)

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

plt.plot(x,y)
plt.show()
```

A variation of the same approach is to alter the `H(x)` function itself, and put the for-loop inside it:

```
def H_loop(x):
    r = np.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)
```

We see that this latter approach ensures that we can call the function with an array argument `x`, but the downside to both version is that we need to write quite a lot of new code, and using a for-loop is much slower than using vectorized array computing.

An alternative approach is to use a builtin Numpy function named `vectorize`[1], which offers automatic vectorization of functions with if-tests. The line

[1] A fairly common misconception is to think that to vectorize a computation, or to make a vectorized version of a function, always involves using the function `numpy.vectorize`. This is not the case. In most cases we only need to make sure that we use array-ready functions such as `numpy.sin`, `numpy.exp`, etc. instead of the scalar version from `math`, and code all calculations so that they work on an entire array instead of stepping through the elements with a for-loop. The `vectorize`-function is usually only needed for functions containing if-tests.

```
Hv = np.vectorize(H)
```

creates a vectorized version `Hv(x)` of the function `H(x)`, which will work with an array argument. This approach is obviously better in the sense that the conversion is automatic so we need to write very little new code, but it is actually about as slow as the approaches using a for-loop.

A third approach is to write a new function, where the if-test is coded differently:

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

For this particular case the Numpy function `where` will evaluate the expression `x<0` for all elements in the array `x`, and return an array of the same length as `x`, with values 0.0 for all elements where `x<0` is True, and 1.0 for wherever `x<0` is False. More generally, a function with an if-test is converted to an array-ready vectorized version in the following way:

```
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
```

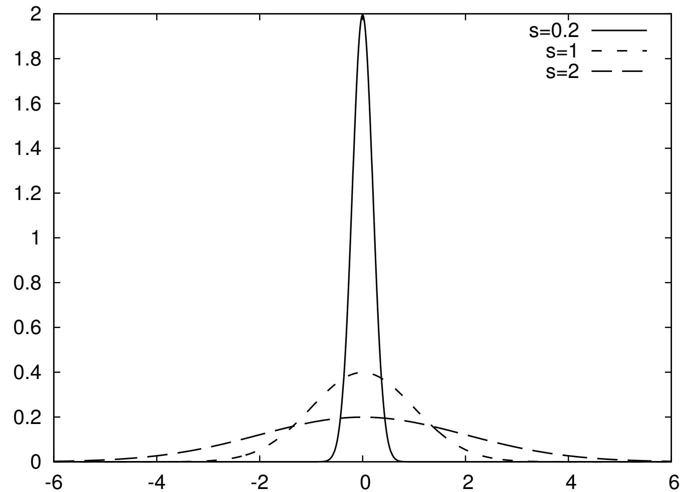
This conversion is of course not as automatic as using `vectorize`, and requires us to write some more code, but it is much more computationally efficient than the other versions. When working with long arrays this can sometimes be important.

2.1 How to make a movie/animation of a plot.

It is often useful to make animations or movies of plots, for instance if the plot represents some physical phenomenon that actually changes with time, or if we want to visualize the effect of changing parameters. Matplotlib has multiple tools for creating such plots, and we will explore some of them here. To start with a specific case, consider the well-known Gaussian bell function:

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

The parameter m is the location of the peak of the function, while s is a measure of the width of "bell". To illustrate the effect of these parameters, we want to 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. Movies of plots are created with the classical approach of cartoon movies (or really any movies), by creating a set of images and viewing them in rapid sequence. For our specific example, the typical approach is to write a for-loop to step through the s-values and either show the resulting plots directly or store them in individual files for later processing. Regardless of which approach we use, it is imporant to always fix the axes when making animations of plots. Otherwise, the y axis always adapts to the peak of the function and the visual impression gets completely wrong

More specifically, there are three ways to create a movie of the kind outlined above:

1. Let the animation run *live*, without saving any files. With this appraoch the plots are simply drawn on the screen as they are created, i.e. one plot is shown for each pass of the for-loop. The approach is simple, but has the drawback that we cannot pause the movie or change the speed.

2. Loop over all data values, plot and make a hardcopy (file) for each value, combine all hardcopies to a movie. This appraoch enables us to actually create a movie file, which can be played using standard movie player software. The drawback of this approach is that it requires separately installed software (for instance *ImageMagick*) to make create movie and see the animation.

3. Use a 'FuncAnimation' object from 'matplotlib'. This approach uses a more advanced feature of Matplotlib, and can be seen as a combination of the two approaches above. The animation is played *live*, but can also be stored in an actual movie file. The downside is that the creation of

the movie file still relies on externally installed software that needs to be integrated with Matplotlib.

Alternative one; running the movie "live" as the plots are created. This apprach is the simplest of the three, and requires very few tools that we have not already seen. We simply use a for-loop to loop over the s-values, and compute new y-values and update the plot for each iteration of the loop. However, there are a couple of technical details that we need to be aware of. In particular, the intuitive approach of simply including calls to `plot(x,y)` followed by `show()` inside the for-loop does not work. Calling `show()` will simply make the program stop after the first plot is made, and it will not run further until we close the plotting window. Also, recall that multiple calls to `plot` was used above to plot multiple curves in a single window, which is not what we want here. Instead, we need to create an object that represents the plot, and then update the y-values of this object for each pass through the loop. The complete code may look like this:

```
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)
```

Most of the lines in this code should be familiar, but there are a few items that are worth taking note of. First, we use the same `plot` function as earlier, but in a slightly different manner. In general, this function does two things; it creates a plot that shows up on the screen if we call `show()`, and it returns a special Matplotlib object which represents the plot (a `Line2D` object). In the examples above we did not need this object, so we did not care about it, but this time we store it in the variable `lines`. Note also that the function always returns a list of such objects. In this case, where we plot only one curve, it is simply a list of length one. To update the plot inside the for-loop, we call the

set_ydata method of this object, i.e. lines[0].set_ydata(y), every time we have computed a new y-array. After updating the data, we call the function draw() to draw the curve on the screen. The final line inside the for-loop is optional, and simply makes the program stop and wait for 0.1 seconds. If we remove this call the movie runs too fast to be visible at all, and we can obviously adjust the speed by changing the argument to the function. As a final on this code, remember the important message from above that we always need to fix the axes when creating movies. Otherwise Matplotlib will adjust the axes automatically for each plot, and the resulting movie will not really look like a movie. Here, we compute the maximum value that the function will obtain in the line max_f = f(m, m, s_stop) (based either on pror knowledge about the Gaussian function of by inspecting the mathematical expression). This value is then used to set the axes for all the plots that make up the movie.

Alternative two; save image files for later processing. This approach is nearly identical to the one above, but instead of showing the plots on the screen we save them to file using the savefig function from Matplotlib. To avoid having each new plot over-write the previous file, we include a counter variable and a formatted string to create a unique file name for each iteration of the for-loop. The complete code is nearly identical to the one above:

```
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
```

Running this program should create a number of image files, located in the directory that we run the program from. Converting these images into a movie requires some external software, for instance convert from the ImageMagick software suite to make animated gids, or for instance ffmpeg or avconv to make

MP4, Flash, or other movie formats. For instance, if we want to create an animated gif of the image files produced above, the following command will do the trick:

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

The resulting gif can be played using `animate` from ImageMagick or in a browser. Note that for this approach to work, one needs to be careful about the file names. The argument `tmp_*.png` passed to the convert function will simply replace `*` with any text, thereby sending all files with this pattern to `convert`. The files are sent in lexicographic (i.e. alphabetical) order, which is why we use the format specifier `04d` in the f-string above. It would be tempting so simply write `{frame_counter}` inside the f-string to create the unique file name, and not worry about the format specifier. However, we would then run into problems when creating the movie with `convert`, since for instance `tmp_10.png` comes before `tmp9.png` in the alphabetic ordering.

Alternative three; use builtin Matplotlib tools. The third approach is the most advanced and flexible, and relies on builtin Matplotlib tools instead of an explicit for-loop that we used above. Without the explicit for-loop the actual steps of creating the animation are more hidden, and the approach is therefore somewhat less intuitive. The essential steps are the following:

1. Make a function to update the plot. In our case this function should compute the new `y` array and call `set_ydata` as above to update the plot.
2. Make a list or array of the argument that changes (here s)
3. Pass the function and the list as arguments to create a `FuncAnimation` object

After creating this object, we can use various builtin methods to save the movie to a file, show it on the screen, etc. The complete code looks as follows

```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()
```

Most of the lines are identical to the examples above, but there are some key differences. We define a function `next_frame` which contains all the code that updates the plot for each frame. The argument to this function should be whatever argument that is changed for each frame (in our case s). After defining this function, it is used to create a `FuncAnimation` object in the next line:

```
ani = FuncAnimation(plt.gcf(), next_frame, frames=s_values, interval=100)
```

This function call will return an object of type `FuncAnimation` [2]. The first argument is simply the current figure object we are working (`gcf` being short for *get current figure*), the next is the function we just defined to update the frames, and the last is the interval between frames, in milliseconds. Numerous other optional arguments to the function can be used to tune the looks of the animation. We refer to the Matplotlib documentation for the details on this. After the object is created, we call the `save` method of the `FuncAnimation` class to create a movie file.[3] To display the animation on the screen, the usual `show()` call is sufficient.

2.2 More useful array operations

At the start of this chapter we introduced the most essential operations needed for using arrays in computations and for plotting, but Numpy arrays can do much more. Here we introduce a few additional useful operations that are convenient to know about when working with arrays. First, we often need to make an array with the same size as another array. This can be done in several ways, for instance using the `zeros` function introduced above:

```
import numpy as np
x = np.linspace(0,10,101)
a = zeros(x.shape, x.dtype)
```

by copying the `x` array:

```
a = x.copy()
```

or using the convenient funtion `zeros_liks`:

[2] Technically, what happens here is that we call the *constructor* of the class `FuncAnimation` to create an object of this class. We will cover classes and constructors in detail in Chapter 7, but for now it is sufficient to view this as a regular function call that returns an object of type `FuncAnimation`.

[3] This call relies on some external software to be installed and integrated with matplotlib, so it may not work on all platforms.

```
a = np.zeros_like(x)  # zeros and same size as x
```

If we write a function that takes either a list or an array as argument, but inside the function it needs to be an array, we can ensure it is converted using the function `asarray`:

```
a = asarray(a)
```

This statement will convert `a` to an array if needed (e.g., if `a` is a list or a single number), but do nothing if `a` is already an array.

The *list slicing* that we briefly introduced in Chapter 2 also works with arrays. Remember the syntax `a[f:t:i]`, where the slice `f:t:i` implies a set of indices (from, to, increment). We can also use any list or array of integers to index into another array:

```
>>> a = linspace(1, 8, 8)
>>> a
array([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.])
>>> a[[1,6,7]] = 10
>>> a
array([  1.,  10.,   3.,   4.,   5.,   6.,  10.,  10.])
>>> a[range(2,8,3)] = -2   # same as a[2:8:3] = -2
>>> a
array([  1.,  10.,  -2.,   4.,   5.,  -2.,  10.,  10.])
```

Finally, we can use an array of boolean expressions to pick out elements of an array, as demonstrated in this example:

```
>>> a < 0
[False, False, True, False, False, True, False, False]
>>> a[a < 0]           # pick out all negative elements
array([-2., -2.])

>>> a[a < 0] = a.max() # if a[i]<10, set a[i]=10
>>> a
array([  1.,  10.,  10.,   4.,   5.,  10.,  10.,  10.])
```

These indexing methods can often be quite useful, since for efficiency reasons we often want to avoid for-loops to loop over arrays elements. Many operations that are naturally implemented as for-loops can be replaced by some creative array slicing and indexing, and the efficiency improvements may be substantial.

2.3 Two-dimensional arrays

Just as lists, arrays can have more than one index. Two-dimensional arrays are particularly relevant, since these are natural representaions of for instance a table of numbers. For instance, to represent a set of numbers like

$$\begin{bmatrix} 0 & 12 & -1 & 5 \\ -1 & -1 & -1 & 0 \\ 11 & 5 & 5 & -2 \end{bmatrix}$$

(called a *matrix* by mathematicians) it is natural to use a two-dimensional array $A_{i,j}$ with one index for the rows and one for the columns:

$$A = \left[\begin{array}{ccc} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{array} \right]$$

In Python code, two-dimensional arrays are not much different from the one-dimensional version, except for an extra index. Making, filling, and modifying a two-dimensional array is done in much the same way, as illustrated by this example:

```
A = zeros((3,4))    # 3x4 table of numbers
A[0,0] = -1
A[1,0] =  1
A[2,0] = 10
A[0,1] = -5
# #if FORMAT != 'ipynb'
...
# #endif
A[2,3] = -100

# can also write (as for nested lists)
A[2][3] = -100
```

We can also create a nested list, as we did in Chapter 2, and convert it to an array:

```
>>> Cdegrees = [-30 + i*10 for i in range(3)]
>>> Fdegrees = [9./5*C + 32 for C in Cdegrees]
>>> table = [[C, F] for C, F in zip(Cdegrees, Fdegrees)]
>>> print table
[[-30, -22.0], [-20, -4.0], [-10, 14.0]]
>>> table2 = array(table)
>>> print table2
[[-30. -22.]
 [-20.  -4.]
 [-10.  14.]]
```

Summary of useful array functionality.

Construction	Meaning
`array(ld)`	copy list data `ld` to a `numpy` array
`asarray(d)`	make array of data `d` (no data copy if already array)
`zeros(n)`	make a `float` vector/array of length `n`, with zeros
`zeros(n, int)`	make an `int` vector/array of length `n` with zeros
`zeros((m,n))`	make a two-dimensional `float` array with shape (`m`,‘n‘)
`zeros_like(x)`	make array of same shape and element type as `x`
`linspace(a,b,m)`	uniform sequence of `m` numbers in $[a, b]$
`a.shape`	tuple containing `a`’s shape
`a.size`	total no of elements in `a`
`len(a)`	length of a one-dim. array `a` (same as `a.shape[0]`)
`a.dtype`	the type of elements in `a`
`a.reshape(3,2)`	return `a` reshaped as 3×2 array
`a[i]`	vector indexing
`a[i,j]`	two-dim. array indexing
`a[1:k]`	slice: reference data with indices `1`,...,‘k-1‘
`a[1:8:3]`	slice: reference data with indices `1`, `4`,...,‘7‘
`b = a.copy()`	copy an array
`sin(a), exp(a), ...`	`numpy` functions applicable to arrays
`c = concatenate((a, b))`	`c` contains `a` with `b` appended
`c = where(cond, a1, a2)`	`c[i] = a1[i]` if `cond[i]`, else `c[i] = a2[i]`
`isinstance(a, ndarray)`	is `True` if `a` is an array