

Lists are introduced in Chapter 2 to store “tabular data” in a convenient way. An array is an object that is very similar to a list, but less flexible and computationally much more efficient. When using the computer to perform mathematical calculations, we often end up with a huge amount of numbers and associated arithmetic operations. Storing numbers in lists may in such contexts lead to slow programs, while arrays can make the programs run much faster. This may not be very important for the mathematical problems in this book, since most of the programs usually finish execution within a few seconds. Nevertheless, in more advanced applications of mathematics, especially the applications met in industry and science, computer programs may run for weeks and months. Any clever idea that reduces the execution time to days or hours is therefore paramount¹.

This chapter gives a brief introduction to arrays – how they are created and what they can be used for. Array computing usually ends up with a lot of numbers. It may be very hard to understand what these numbers mean by just looking at them. Since the human is a visual animal, a good way to understand numbers is to visualize them. In this chapter we concentrate on visualizing curves that reflect functions of one variable, e.g., curves of the form $y = f(x)$. A synonym for curve is graph, and the image of curves on the screen is often called a plot. We will use arrays to store the information about points along the curve. It is fair to say that array computing demands visualization and visualization demands arrays.

¹ Many may argue that programmers of mathematical software have traditionally paid too much attention to efficiency and smart program constructs. The resulting software often becomes very hard to maintain and extend. In this book we advocate a focus on clear, well-designed, and easy-to-understand programs that work correctly. Optimization for speed should always come as a second step in program development.

All program examples in this chapter can be found as files in the folder `src/plot`.

4.1 Vectors

This section gives a brief introduction to the vector concept, assuming that you have heard about vectors in the plane and maybe vectors in space before. This background will be valuable when we start to work with arrays and curve plotting.

4.1.1 The Vector Concept

Some mathematical quantities are associated with a set of numbers. One example is a point in the plane, where we need two coordinates (real numbers) to describe the point mathematically. Naming the two coordinates of a particular point as x and y , it is common to use the notation (x, y) for the point. That is, we group the numbers inside parentheses. Instead of symbols we might use the numbers directly: $(0, 0)$ and $(1.5, -2.35)$ are also examples of coordinates in the plane.

A point in three-dimensional space has three coordinates, which we may name x_1 , x_2 , and x_3 . The common notation groups the numbers inside parentheses: (x_1, x_2, x_3) . Alternatively, we may use the symbols x , y , and z , and write the point as (x, y, z) , or numbers can be used instead of symbols.

From high school you may have a memory of solving two equations with two unknowns. At the university you will soon meet problems that are formulated as n equations with n unknowns. The solution of such problems contains n numbers that we can collect inside parentheses and number from 1 to n : $(x_1, x_2, x_3, \dots, x_{n-1}, x_n)$.

Quantities such as (x, y) , (x, y, z) , or (x_1, \dots, x_n) are known as *vectors* in mathematics. A visual representation of a vector is an arrow that goes from the origin to a point. For example, the vector (x, y) is an arrow that goes from $(0, 0)$ to the point with coordinates (x, y) in the plane. Similarly, (x, y, z) is an arrow from $(0, 0, 0)$ to the point (x, y, z) in three-dimensional space.

Mathematicians found it convenient to introduce spaces with higher dimension than three, because when we have a solution of n equations collected in a vector (x_1, \dots, x_n) , we may think of this vector as a point in a space with dimension n , or equivalently, an arrow that goes from the origin $(0, \dots, 0)$ in n -dimensional space to the point (x_1, \dots, x_n) . Figure 4.1 illustrates a vector as an arrow, either starting at the origin, or at any other point. Two arrows/vectors that have the same direction and the same length are mathematically equivalent.

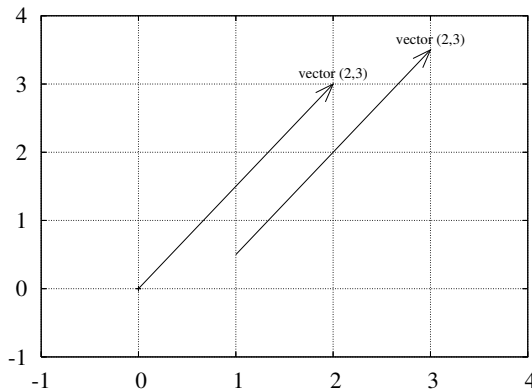


Fig. 4.1 A vector $(2,3)$ visualized in the standard way as an arrow from the origin to the point $(2,3)$, and mathematically equivalently, as an arrow from $(1, \frac{1}{2})$ (or any point (a,b)) to $(3, 3\frac{1}{2})$ (or $(a+2, b+3)$).

We say that (x_1, \dots, x_n) is an n -vector or a vector with n components. Each of the numbers x_1, x_2, \dots is a component or an element. We refer to the first component (or element), the second component (or element), and so forth.

A Python program may use a list or tuple to represent a vector:

```
v1 = [x, y]          # list of variables
v2 = (-1, 2)         # tuple of numbers
v3 = (x1, x2, x3)     # tuple of variables
from math import exp
v4 = [exp(-i*0.1) for i in range(150)]
```

While $v1$ and $v2$ are vectors in the plane and $v3$ is a vector in three-dimensional space, $v4$ is a vector in a 150-dimensional space, consisting of 150 values of the exponential function. Since Python lists and tuples have 0 as the first index, we may also in mathematics write the vector (x_1, x_2) as (x_0, x_1) . This is not at all common in mathematics, but makes the distance from a mathematical description of a problem to its solution in Python shorter.

It is impossible to visually demonstrate how a space with 150 dimensions looks like. Going from the plane to three-dimensional space gives a rough feeling of what it means to add a dimension, but if we forget about the idea of a visual perception of space, the mathematics is very simple: Going from a 4-dimensional vector to a 5-dimensional vector is just as easy as adding an element to a list of symbols or numbers.

4.1.2 Mathematical Operations on Vectors

Since vectors can be viewed as arrows having a length and a direction, vectors are extremely useful in geometry and physics. The velocity of a car has a magnitude and a direction, so has the acceleration, and

the position of a car is a point² which is also a vector. An edge of a triangle can be viewed as a line (arrow) with a direction and length.

In geometric and physical applications of vectors, mathematical operations on vectors are important. We shall exemplify some of the most important operations on vectors below. The goal is not to teach computations with vectors, but more to illustrate that such computations are defined by mathematical rules³. Given two vectors, (u_1, u_2) and (v_1, v_2) , we can add these vectors according to the rule:

$$(u_1, u_2) + (v_1, v_2) = (u_1 + v_1, u_2 + v_2). \quad (4.1)$$

We can also subtract two vectors using a similar rule:

$$(u_1, u_2) - (v_1, v_2) = (u_1 - v_1, u_2 - v_2). \quad (4.2)$$

A vector can be multiplied by a number. This number, called a below, is usually denoted as a *scalar*:

$$a \cdot (v_1, v_2) = (av_1, av_2). \quad (4.3)$$

The inner product, also called dot product, or scalar product, of two vectors is a number⁴:

$$(u_1, u_2) \cdot (v_1, v_2) = u_1v_1 + u_2v_2. \quad (4.4)$$

There is also a *cross product* defined for 2-vectors or 3-vectors, but we do not list the cross product formula here.

The length of a vector is defined by

$$\|(v_1, v_2)\| = \sqrt{(v_1, v_2) \cdot (v_1, v_2)} = \sqrt{v_1^2 + v_2^2}. \quad (4.5)$$

The same mathematical operations apply to n -dimensional vectors as well. Instead of counting indices from 1, as we usually do in mathematics, we now count from 0, as in Python. The addition and subtraction of two vectors with n components (or elements) read

$$(u_0, \dots, u_{n-1}) + (v_0, \dots, v_{n-1}) = (u_0 + v_0, \dots, u_{n-1} + v_{n-1}), \quad (4.6)$$

$$(u_0, \dots, u_{n-1}) - (v_0, \dots, v_{n-1}) = (u_0 - v_0, \dots, u_{n-1} - v_{n-1}). \quad (4.7)$$

² A car is of course not a mathematical point, but when studying the acceleration of a car, it suffices to view it as a point. In other occasions, e.g., when simulating a car crash on a computer, the car may be modeled by a large number (say 10^6) of connected points.

³ You might recall many of the formulas here from high school mathematics or physics. The really new thing in this chapter is that we show how rules for vectors in the plane and in space can easily be extended to vectors in n -dimensional space.

⁴ From high school mathematics and physics you might recall that the inner or dot product also can be expressed as the product of the lengths of the two vectors multiplied by the cosine of the angle between them. We will not make use of this formula.

Multiplication of a scalar a and a vector (v_0, \dots, v_{n-1}) equals

$$(av_0, \dots, av_{n-1}). \quad (4.8)$$

The inner or dot product of two n -vectors is defined as

$$(u_0, \dots, u_{n-1}) \cdot (v_0, \dots, v_{n-1}) = u_0v_0 + \dots + u_{n-1}v_{n-1} = \sum_{j=0}^{n-1} u_jv_j. \quad (4.9)$$

Finally, the length $\|v\|$ of an n -vector $v = (v_0, \dots, v_{n-1})$ is

$$\begin{aligned} \sqrt{(v_0, \dots, v_{n-1}) \cdot (v_0, \dots, v_{n-1})} &= (v_0^2 + v_1^2 + \dots + v_{n-1}^2)^{\frac{1}{2}} \\ &= \left(\sum_{j=0}^{n-1} v_j^2 \right)^{\frac{1}{2}}. \end{aligned} \quad (4.10)$$

4.1.3 Vector Arithmetics and Vector Functions

In addition to the operations on vectors in Chapter 4.1.2, which you might recall from high school mathematics, we can define other operations on vectors. This is very useful for speeding up programs. Unfortunately, the forthcoming vector operations are hardly treated in textbooks on mathematics, yet these operations play a significant role in mathematical software, especially in computing environment such as Matlab, Octave, Python, and R.

Applying a mathematical function of one variable, $f(x)$, to a vector is defined as a vector where f is applied to each element. Let $v = (v_0, \dots, v_{n-1})$ be a vector. Then

$$f(v) = (f(v_0), \dots, f(v_{n-1})).$$

For example, the sine of v is

$$\sin(v) = (\sin(v_0), \dots, \sin(v_{n-1})).$$

It follows that squaring a vector, or the more general operation of raising the vector to a power, can be defined as applying the operation to each element:

$$v^b = (v_0^b, \dots, v_{n-1}^b).$$

Another operation between two vectors that arises in computer programming of mathematics is the “asterix” multiplication, defined as

$$u * v = (u_0v_0, u_1v_1, \dots, u_{n-1}v_{n-1}). \quad (4.11)$$

Adding a scalar to a vector or array can be defined as adding the scalar to each component. If a is a scalar and v a vector, we have

$$a + v = (a + v_0, \dots, a + v_{n-1}).$$

A compound vector expression may look like

$$v^2 * \cos(v) * e^v + 2. \quad (4.12)$$

How do we calculate this expression? We use the normal rules of mathematics, working our way, term by term, from left to right, paying attention to the fact that powers are evaluated before multiplications and divisions, which are evaluated prior to addition and subtraction. First we calculate v^2 , which results in a vector we may call u . Then we calculate $\cos(v)$ and call the result p . Then we multiply $u * p$ to get a vector which we may call w . The next step is to evaluate e^v , call the result q , followed by the multiplication $w * q$, whose result is stored as r . Then we add $r + 2$ to get the final result. It might be more convenient to list these operations after each other:

1. $u = v^2$
2. $p = \cos(v)$
3. $w = u * p$
4. $q = e^v$
5. $r = w * q$
6. $s = r + 2$

Writing out the vectors u , w , p , q , and r in terms of a general vector $v = (v_0, \dots, v_{n-1})$ (do it!) shows that the result of the expression (4.12) is the vector

$$(v_0^2 \cos(v_0) e^{v_0} + 2, \dots, v_{n-1}^2 \cos(v_{n-1}) e^{v_{n-1}} + 2).$$

That is, component no. i in the result vector equals the number arising from applying the formula (4.12) to v_i , where the $*$ multiplication is ordinary multiplication between two numbers.

We can, alternatively, introduce the function

$$f(x) = x^2 \cos(x) e^x + 2$$

and use the result that $f(v)$ means applying f to each element in v . The result is the same as in the vector expression (4.12).

In Python programming it is important for speed (and convenience too) that we can apply functions of one variable, like $f(x)$, to vectors. What this means mathematically is something we have tried to explain in this subsection. Doing Exercises 4.4 and 4.5 may help to grasp the ideas of vector computing, and with more programming experience you will hopefully discover that vector computing is very useful. It is not necessary to have a thorough understanding of vector computing in order to proceed with the next sections.

Arrays are used to represent vectors in a program, but one can do more with arrays than with vectors. Until Chapter 4.6 it suffices to think of arrays as the same as vectors in a program.

4.2 Arrays in Python Programs

This section introduces array programming in Python, but first we create some lists and show how arrays differ from lists.

4.2.1 Using Lists for Collecting Function Data

Suppose we have a function $f(x)$ and want to evaluate this function at a number of x points x_0, x_1, \dots, x_{n-1} . We could collect the n pairs $(x_i, f(x_i))$ in a list, or we could collect all the x_i values, for $i = 0, \dots, n-1$, in a list and all the associated $f(x_i)$ values in another list. We learned how to create such lists in Chapter 2, but as a review, we present the relevant program statements in an interactive session:

```
>>> def f(x):  
...     return x**3          # sample function  
...  
>>> n = 5                    # no of points along the x axis  
>>> dx = 1.0/(n-1)          # spacing between x points in [0,1]  
>>> xlist = [i*dx for i in range(n)]  
>>> ylist = [f(x) for x in xlist]  
>>> pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

Here we have used list comprehensions for achieving compact code. Make sure that you understand what is going on in these list comprehensions (you are encouraged to write the same code using standard for loops and appending new list elements in each pass of the loops).

The list elements consist of objects of the same type: any element in `pairs` is a list of two `float` objects, while any element in `xlist` or `ylist` is a `float`. Lists are more flexible than that, because an element can be an object of any type, e.g.,

```
mylist = [2, 6.0, 'tmp.ps', [0,1]]
```

Here `mylist` holds an `int`, a `float`, a `string`, and a `list`. This combination of diverse object types makes up what is known as *heterogeneous* lists. We can also easily remove elements from a list or add new elements anywhere in the list. This flexibility of lists is in general convenient to have as a programmer, but in cases where the elements are of the same type and the number of elements is fixed, arrays can be used instead. The benefits of arrays are faster computations, less memory demands, and extensive support for mathematical operations on the

data. Because of greater efficiency and mathematical convenience, arrays will be used to a large extent in this book. The great use of arrays is also prominent in other programming environments such as Matlab, Octave, and R, for instance. Lists will be our choice instead of arrays when we need the flexibility of adding or removing elements or when the elements may be of different object types.

4.2.2 Basics of Numerical Python Arrays

An *array* object can be viewed as a variant of a list, but with the following assumptions and features:

- All elements must be of the same type, preferably integer, real, or complex numbers, for efficient numerical computing and storage.
- The number of elements must be known⁵ when the array is created.
- Arrays are not part of standard Python⁶ – one needs an additional package called *Numerical Python*, often abbreviated as NumPy. The Python name of the package, to be used in `import` statements, is `numpy`.
- With `numpy`, a wide range of mathematical operations can be done directly on complete arrays, thereby removing the need for loops over array elements. This is commonly called *vectorization* and may cause a dramatic speed-up of Python programs. Vectorization makes use of the vector computing concepts from Chapter 4.1.3.
- Arrays with one index are often called vectors. Arrays with two indices are used as an efficient data structure for tables, instead of lists of lists. Arrays can also have three or more indices.

The fundamental import statement to get access to Numerical Python array functionality reads

```
from numpy import *
```

To convert a list `r` to an array, we use the `array` function from `numpy`:

```
a = array(r)
```

To create a new array of length `n`, filled with zeros, we write

```
a = zeros(n)
```

The array elements are of a type that corresponds to Python's `float` type. A second argument to `zeros` can be used to specify other element types, e.g., `int`. Arrays with more than one index are treated in Chapter 4.6.

⁵ The number of elements can be changed, at a substantial computational cost.

⁶ Actually, there is an object type called `array` in standard Python, but this data type is not so efficient for mathematical computations.

Often one wants an array to have n elements with uniformly distributed values in an interval $[p, q]$. The `numpy` function `linspace` creates such arrays:

```
a = linspace(p, q, n)
```

We remark that there are a large number of functions and modules within `numpy`.

Array elements are accessed by square brackets as for lists: `a[i]`. Slices also work as for lists, for example, `a[1:-1]` picks out all elements except the first and the last, but contrary to lists, `a[1:-1]` is not a copy of the data in `a`. Hence,

```
b = a[1:-1]
b[2] = 0.1
```

will also change `a[3]` to 0.1. A slice `a[i:j:s]` picks out the elements starting with index `i` and stepping `s` indices at the time up to, but not including, `j`. Omitting `i` implies `i=0`, and omitting `j` implies `j=n` if `n` is the number of elements in the array. For example, `a[0:-1:2]` picks out every two elements up to, but not including, the last element, while `a[::4]` picks out every four elements in the whole array.

4.2.3 Computing Coordinates and Function Values

With these basic operations at hand, we can continue the session from the previous section and make arrays out of the lists `xlist`, `ylist`, and `pairs`:

```
>>> from numpy import *
>>> x2 = array(xlist)      # turn list xlist into array x2
>>> y2 = array(ylist)
>>> x2
array([ 0.   ,  0.25,  0.5  ,  0.75,  1.   ])
>>> y2
array([ 0.       ,  0.015625,  0.125   ,  0.421875,  1.       ])
```

Instead of first making a list and then converting the list to an array, we can compute the arrays directly. The equally spaced coordinates in `x2` are naturally computed by the `linspace` function. The `y2` array can be created by `zeros`, to ensure that `y2` has the right length⁷ `len(x2)`, and then we can run a `for` loop to fill in all elements in `y2` with `f` values:

```
>>> n = len(xlist)
>>> x2 = linspace(0, 1, n)
>>> y2 = zeros(n)
>>> for i in xrange(n):
```

⁷ This is referred to as *allocating* the array, and means that a part of the computer's memory is marked for being occupied by this array.

```
...     y2[i] = f(x2[i])
...
>>> y2
array([ 0.          ,  0.015625,  0.125      ,  0.421875,  1.          ])
```

Note that we here in the `for` loop have used `xrange` instead of `range`. The former is faster for long loops and is our preference over `range` when we have loops over (possibly long) arrays.

We used a list comprehension for computing the y , while we used a `for` loop for computing the array `y2`. List comprehensions do not work with arrays because the list comprehension creates a list, not an array. We can, of course, compute the y coordinates with a list comprehension and then turn the resulting list into an array:

```
>>> x2 = linspace(0, 1, n)
>>> y2 = array([f(xi) for xi in x2])
```

Nevertheless, there is a better way of computing `y2` as the next paragraph explains.

4.2.4 Vectorization

Loops over very long arrays may run slowly. A great advantage with arrays is that we can get rid of the loops and apply `f` directly to the whole array:

```
>>> y2 = f(x2)
>>> y2
array([ 0.          ,  0.015625,  0.125      ,  0.421875,  1.          ])
```

The magic that makes `f(x2)` work builds on the vector computing concepts from Chapter 4.1.3.

Instead of calling `f(x2)` we can equivalently write the function formula `x2**3` directly. As another example, a Python assignment like

```
r = sin(x)*cos(x)*exp(-x**2) + 2 + x**2
```

works perfectly for an array `x`. The resulting array is the same as if we apply the formula to each array entry:

```
r = zeros(len(x))
for i in xrange(len(x)):
    r[i] = sin(x[i])*cos(x[i])*exp(-x[i]**2) + 2 + x[i]**2
```

Replacing a loop like the one above by a vector/array expression (like `sin(x)*cos(x)*exp(-x**2) + 2 + x**2`) is what we call *vectorization*. The loop version is often referred to as *scalar code*. For example,

```
x = zeros(N); y = zeros(N)
dx = 2.0/(N-1) # spacing of x coordinates
for i in range(N):
    x[i] = -1 + dx*i
    y[i] = exp(-x[i])*x[i]
```

is scalar code, while the corresponding vectorized version reads

```
x = linspace(-1, 1, N)
y = exp(-x)*x
```

We remark that list comprehensions,

```
x = array([-1 + dx*i for i in range(N)])
y = array([exp(-xi)*xi for xi in x])
```

result in scalar code because we still have explicit, slow Python `for` loops. The requirement of vectorized code is that there are no explicit Python `for` loops. The loops that are required to compute each array element are performed in fast C or Fortran code in the `numpy` package.

Most Python functions intended for an scalar argument `x`, like

```
def f(x):
    return x**4*exp(-x)
```

automatically work for an array argument `x`:

```
x = linspace(-3, 3, 101)
y = f(x)
```

We say that the function `f` is vectorized. Not any Python function `f(x)` will be automatically vectorized, i.e., sending an array `x` to `f(x)` may lead to wrong results or an exception. Chapter 4.4.1 provides examples where we have to do special actions in order to vectorize functions.

Vectorization is very important for speeding up Python programs where we do heavy computations with arrays. Moreover, vectorization gives more compact code that is easier to read. Vectorization becomes particularly important for statistical simulations in Chapter 8.

4.3 Curve Plotting

Visualizing a function $f(x)$ is done by drawing the curve $y = f(x)$ in an xy coordinate system. When we use a computer to do this task, we say that we *plot* the curve. Technically, we plot a curve by drawing straight lines between n points on the curve. The more points we use, the smoother the curve appears.

Suppose we want to plot the function $f(x)$ for $a \leq x \leq b$. First we pick out n x coordinates in the interval $[a, b]$, say we name these x_0, x_1, \dots, x_{n-1} . Then we evaluate $y_i = f(x_i)$ for $i = 0, 1, \dots, n-1$.

The points (x_i, y_i) , $i = 0, 1, \dots, n-1$, now lie on the curve $y = f(x)$. Normally, we choose the x_i coordinates to be equally spaced, i.e.,

$$x_i = a + ih, \quad h = \frac{b-a}{n-1}.$$

If we store the x_i and y_i values in two arrays `x` and `y`, we can plot the curve by the command `plot(x,y)`.

Sometimes the names of the independent variable and the function differ from x and f , but the plotting procedure is the same. Our first example of curve plotting demonstrates this fact by involving a function of t .

4.3.1 The SciTools and Easyviz Packages

There are lots of plotting programs that we can use to create visual graphics with curves on the computer screen. As part of this book project, we have created a unified interface *Easyviz* to different plotting programs such that you can write one set of statements in your program regardless of which plotting program you actually use “behind the curtain” to create the graphics. The statements needed to plot a curve are very similar to those used in the Matlab and Octave computing environments.

Easyviz is part of a larger package called SciTools. This package contains many useful tools for mathematical computations in Python. SciTools builds heavily on Numerical Python. It also makes use of the comprehensive scientific computing environment SciPy. If you start your program with

```
from scitools.std import *
```

you will automatically perform import of many useful modules for numerical Python programming. Among Easyviz functions and other things, the import statement above performs imports from `numpy`, `scitools.numpyutils` (some convenience functions), `numpy.lib.scimath` (see Chapter 1.6.3), and `scipy` (if available). In addition, the statement imports `os`, `sys`, and the `StringFunction` tool (see Chapter 3.1.4). We refer to the paragraph “Importing Just Easyviz” on page 194 for a precise list of what is actually imported by a `from scitools.std import *`. The advantage with this particular import is that one line of code gives you a lot of the functionality you commonly need in this book. The downside is that this import statement is comprehensive and therefore takes some time (seconds) to execute, especially if `scipy` is available. If you find the waiting time annoying, you may instead use a minimal set of import statements as explained on page 194.

There are a couple of SciTools functions that you may find convenient to know about:

- `seq(a,b,h)` returns an array with equally spaced numbers starting with `a`, ending with `b`, and with a spacing of `h`.
- `iseq(a,b,h)` works as `seq(a,b,h)` except that `a`, `b`, and `h` are integers and the return array contains a set of integers. The advantage of `iseq` over `range` is that the upper limit `b` is included in the sequence of integers. When implementing mathematical algorithms where an index has a specified range, say $i = 1, \dots, n$, we think it is clearer to write `for i in iseq(1,n)` in the program instead of `for i in range(1,n+1)`.

The inverse trigonometric functions have different names in `math` and `numpy`, a fact that prevents an expression written for scalars, using `math` names, to be immediately valid for vectors. Therefore, the `from scitools.std import *` action also imports the names `asin`, `acos`, and `atan` for `numpy/scipy`'s `arcsin`, `arccos`, and `arctan` functions, to ease vectorization of mathematical expressions involving inverse trigonometric functions.

4.3.2 Plotting a Single Curve

Let us plot the curve $y = t^2 \exp(-t^2)$ for t values between 0 and 3. First we generate equally spaced coordinates for t , say 51 values (50 intervals). Then we compute the corresponding y values at these points, before we call the `plot(t,y)` command to make the curve plot. Here is the complete program:

```
from scitools.std import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 51)    # 51 points between 0 and 3
y = zeros(len(t))        # allocate y with float elements
for i in xrange(len(t)):
    y[i] = f(t[i])

plot(t, y)
```

The first line imports all of SciTools and Easyviz that can be handy to have when doing scientific computations. In this program we pre-allocate the `y` array and fill it with values, element by element, in a Python loop. Alternatively, we may operate on the whole `t` array at once, which yields faster and shorter code:

```
from scitools.std import *

def f(t):
    return t**2*exp(-t**2)

t = linspace(0, 3, 51)    # 51 points between 0 and 3
y = f(t)                  # compute all f values at once
plot(t, y)
```

The `f` function can also be skipped, if desired, so that we can write directly

```
y = t**2*exp(-t**2)
```

To include the plot in electronic documents, we need a hardcopy of the figure in PostScript, PNG, or another image format. The `hardcopy` command produces files with images in various formats:

```
hardcopy('tmp1.eps') # produce PostScript  
hardcopy('tmp1.png') # produce PNG
```

The filename extension determines the format: `.ps` or `.eps` for PostScript, and `.png` for PNG. Figure 4.2 displays the resulting plot.

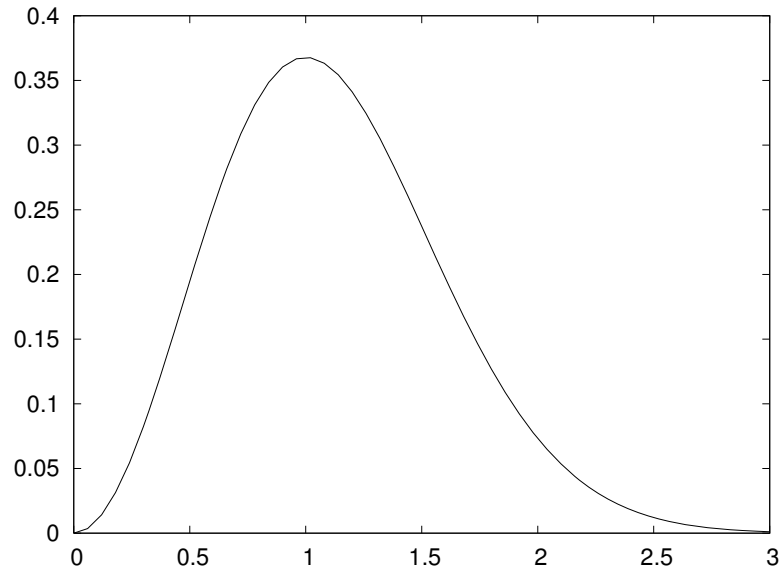


Fig. 4.2 A simple plot in PostScript format.

On some platforms, some backends may result in a plot that is shown in just a fraction of a second on the screen before the plot window disappears (using the Gnuplot backend on Windows machines or using the Matplotlib backend constitute two examples). To make the window stay on the screen, add

```
raw_input('Press Enter: ')
```

at the end of the program. The plot window is killed when the program terminates, and this statement postpones the termination until the user hits the Enter key.

4.3.3 Decorating the Plot

The x and y axis in curve plots should have labels, here t and y , respectively. Also, the curve should be identified with a label, or legend as it is often called. A title above the plot is also common. In addition, we may want to control the extent of the axes (although most plotting programs will automatically adjust the axes to the range of the data). All such things are easily added after the `plot` command:

```
xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)')
axis([0, 3, -0.05, 0.6]) # [tmin, tmax, ymin, ymax]
title('My First Easyviz Demo')
```

This syntax is inspired by Matlab to make the switch between Easyviz and Matlab almost trivial. Easyviz has also introduced a more "Pythonic" `plot` command where all the plot properties can be set at once:

```
plot(t, y,
      xlabel='t',
      ylabel='y',
      legend='t^2*exp(-t^2)',
      axis=[0, 3, -0.05, 0.6],
      title='My First Easyviz Demo',
      hardcopy='tmp1.eps',
      show=True)
```

With `show=False` one can avoid the plot window on the screen and just make the hardcopy. This feature is particularly useful if one generates a large number of plots in a loop.

Note that we in the curve legend write t square as `t^2` (LaTeX style) rather than `t**2` (program style). Whichever form you choose is up to you, but the LaTeX form sometimes looks better in some plotting programs (Gnuplot is one example). See Figure 4.3 for what the modified plot looks like and how `t^2` is typeset in Gnuplot.

4.3.4 Plotting Multiple Curves

A common plotting task is to compare two or more curves, which requires multiple curves to be drawn in the same plot. Suppose we want to plot the two functions $f_1(t) = t^2 \exp(-t^2)$ and $f_2(t) = t^4 \exp(-t^2)$. If we write two `plot` commands after each other, two separate plots will be made. To make the second `plot` command draw the curve in the first plot, we need to issue a `hold('on')` command. Alternatively, we can provide all data in a single `plot` command. A complete program illustrates the different approaches:

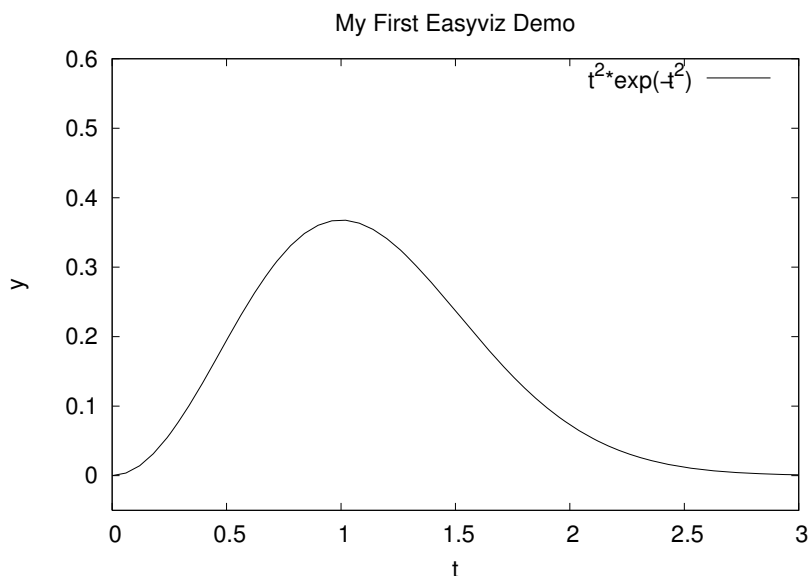


Fig. 4.3 A single curve with label, title, and axis adjusted.

```
from scitools.std import * # for curve plotting

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

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

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

# Matlab-style syntax:
plot(t, y1)
hold('on')
plot(t, y2)

xlabel('t')
ylabel('y')
legend('t^2*exp(-t^2)', 't^4*exp(-t^2)')
title('Plotting two curves in the same plot')
hardcopy('tmp2.eps')

# alternative:
plot(t, y1, t, y2, xlabel='t', ylabel='y',
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     hardcopy='tmp2.eps')
```

The sequence of the multiple legends is such that the first legend corresponds to the first curve, the second legend to the second curve, and so on. The visual result appears in Figure 4.4.

Doing a `hold('off')` makes the next plot command create a new plot.

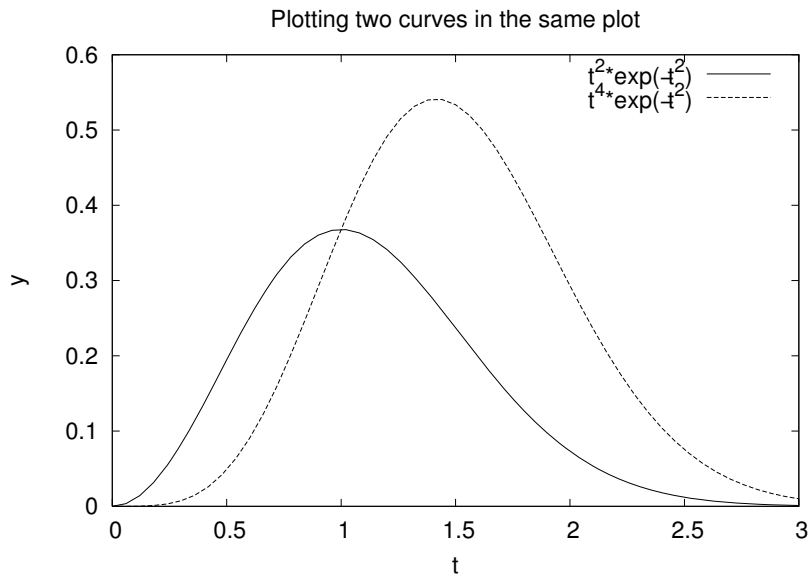


Fig. 4.4 Two curves in the same plot.

4.3.5 Controlling Line Styles

When plotting multiple curves in the same plot, the individual curves get distinct default line styles, depending on the program that is used to produce the curve (and the settings for this program). It might well happen that you get a green and a red curve (which is bad for a significant portion of the male population). Therefore, we often want to control the line style in detail. Say we want the first curve (t and y_1) to be drawn as a red solid line and the second curve (t and y_2) as blue circles at the discrete data points. The Matlab-inspired syntax for specifying line types applies a letter for the color and a symbol from the keyboard for the line type. For example, `r-` represents a red (`r`) line (`-`), while `bo` means blue (`b`) circles (`o`). The line style specification is added as an argument after the x and y coordinate arrays of the curve:

```
plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'bo')

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

The effect of controlling the line styles can be seen in Figure 4.5.

Assume now that we want to plot the blue circles at every 4 points only. We can grab every 4 points out of the t array by using an appropriate slice: `t2 = t[:4]`. Note that the first colon means the range from the first to the last data point, while the second colon separates

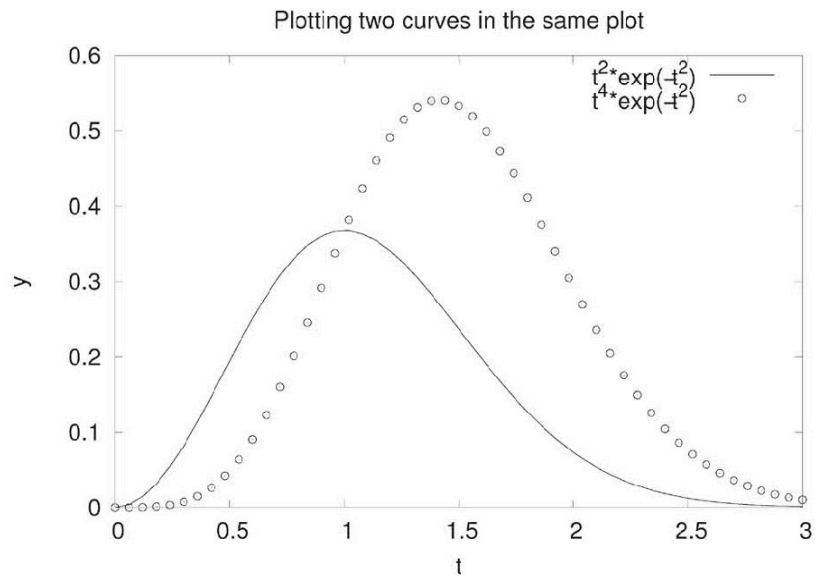


Fig. 4.5 Two curves in the same plot, with controlled line styles.

this range from the stride, i.e., how many points we should "jump over" when we pick out a set of values of the array.

```
from scitools.std import *

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

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

t = linspace(0, 3, 51)
y1 = f1(t)
t2 = t[::4]
y2 = f2(t2)

plot(t, y1, 'r-6', t2, y2, 'bo3',
     xlabel='t', ylabel='y',
     axis=[0, 4, -0.1, 0.6],
     legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
     title='Plotting two curves in the same plot',
     hardcopy='tmp2.eps')
```

In this plot we also adjust the size of the line and the circles by adding an integer: `r-6` means a red line with thickness 6 and `bo3` means red circles with size 5. The effect of the given line thickness and symbol size depends on the underlying plotting program. For the Gnuplot program one can view the effect in Figure 4.6.

The different available line colors include

- yellow: 'y'
- magenta: 'm'
- cyan: 'c'

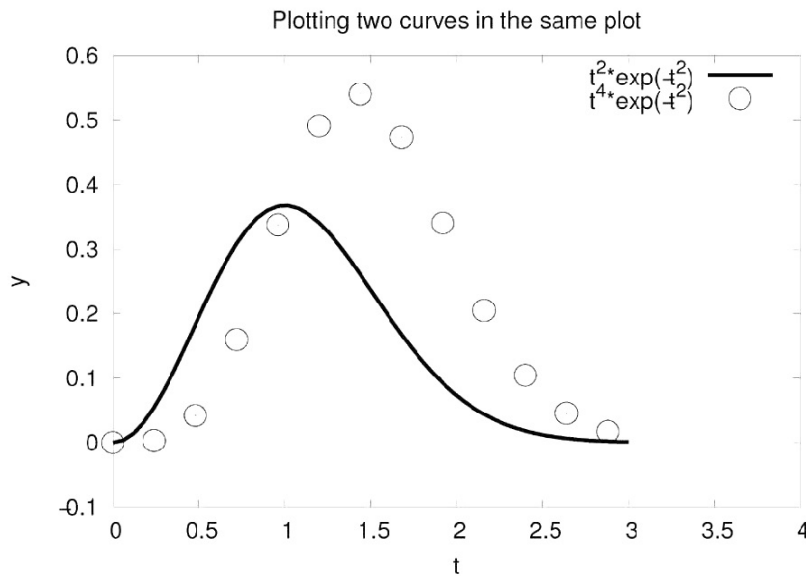


Fig. 4.6 Circles at every 4 points and extended line thickness (6) and circle size (3).

- red: 'r'
- green: 'g'
- blue: 'b'
- white: 'w'
- black: 'k'

The different available line types are

- solid line: '-'
- dashed line: '--'
- dotted line: ':'
- dash-dot line: '-.'

During programming, you can find all these details in the documentation of the `plot` function. Just type `help(plot)` in an interactive Python shell or invoke `pydoc` with `scitools.easyviz.plot`. This tutorial is available through `pydoc scitools.easyviz`.

We remark that in the Gnuplot program all the different line types are drawn as solid lines on the screen. The hardcopy chooses automatically different line types (solid, dashed, etc.) and not in accordance with the line type specification.

Lots of markers at data points are available:

- plus sign: '+'
- circle: 'o'
- asterisk: '*'
- point: '.'
- cross: 'x'

- square: 's'
- diamond: 'd'
- upward-pointing triangle: '^'
- downward-pointing triangle: 'v'
- right-pointing triangle: '>'
- left-pointing triangle: '<'
- five-point star (pentagram): 'p'
- six-point star (hexagram): 'h'
- no marker (default): None

Symbols and line styles may be combined, for instance as in 'kx-', which means a black solid line with black crosses at the data points.

Another Example. Let us extend the previous example with a third curve where the data points are slightly randomly distributed around the $f_2(t)$ curve:

```
from scitools.std import *

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

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

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

# pick out each 4 points and add random noise:
t3 = t[::4]          # slice, stride 4
random.seed(11)      # fix random sequence
noise = random.normal(loc=0, scale=0.02, size=len(t3))
y3 = y2[::4] + noise

plot(t, y1, 'r-')
hold('on')
plot(t, y2, 'ks-')    # black solid line with squares at data points
plot(t3, y3, 'bo')

legend('t^2*exp(-t^2)', 't^4*exp(-t^2)', 'data')
title('Simple Plot Demo')
axis([0, 3, -0.05, 0.6])
xlabel('t')
ylabel('y')
show()
hardcopy('tmp3.eps')
hardcopy('tmp3.png')
```

The plot is shown in Figure 4.7.

Minimalistic Typing. When exploring mathematics in the interactive Python shell, most of us are interested in the quickest possible commands. Here is an example of minimalistic syntax for comparing the two sample functions we have used in the previous examples:

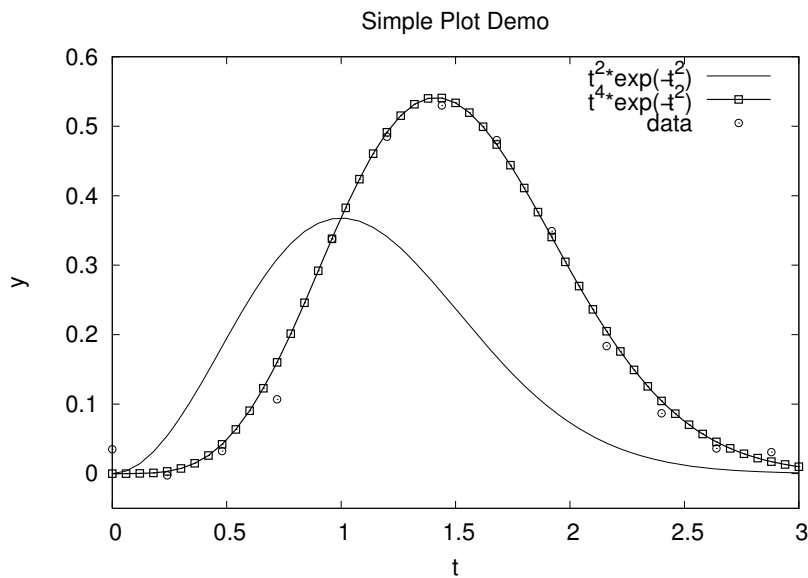


Fig. 4.7 A plot with three curves.

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

Text. A text can be placed at a point (x, y) using the call

```
text(x, y, 'Some text')
```

More Examples. The examples in this tutorial, as well as additional examples, can be found in the `examples` directory in the root directory of the SciTools source code tree.

4.3.6 Interactive Plotting Sessions

All the Easyviz commands can of course be issued in an interactive Python session. The only thing to comment is that the `plot` command returns a result:

```
>>> t = linspace(0, 3, 51)
>>> plot(t, t**2*exp(-t**2))
[<scitools.easyviz.common.Line object at 0xb5727f6c>]
```

Most users will just ignore this output line.

All Easyviz commands that produce a plot return an object reflecting the particular type of plot. The `plot` command returns a list of `Line` objects, one for each curve in the plot. These `Line` objects can be invoked to see, for instance, the value of different parameters in the plot:

```
>>> line, = plot(x, y, 'b')
>>> getp(line)
{'description': '',
 'dims': (4, 1, 1),
 'legend': '',
 'linecolor': 'b',
 'pointsize': 1.0,
 ...}
```

Such output is mostly of interest to advanced users.

4.3.7 Making Animations

A sequence of plots can be combined into an animation and stored in a movie file. First we need to generate a series of hardcopies, i.e., plots stored in files. Thereafter we must use a tool to combine the individual plot files into a movie file.

Example. The function $f(x; m, s) = (2\pi)^{-1/2} s^{-1} \exp \left[-\frac{1}{2} \left(\frac{x-m}{s} \right)^2 \right]$ is known as the Gaussian function or the probability density function of the normal (or Gaussian) distribution. This bell-shaped function is "wide" for large s and "peak-formed" for small s , see Figure 4.8. The function is symmetric around $x = m$ ($m = 0$ in the figure). Our goal is to make an animation where we see how this function evolves as s is decreased. In Python we implement the formula above as a function `f(x, m, s)`.

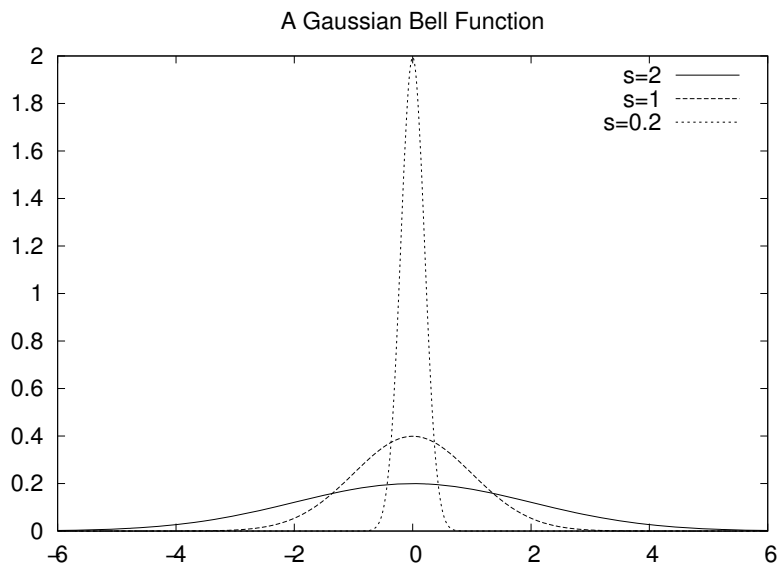


Fig. 4.8 Different shapes of a Gaussian function.

The animation is created by varying s in a loop and for each s issue a plot command. A moving curve is then visible on the screen. One can also make a movie file that can be played as any other computer movie using a standard movie player. To this end, each plot is saved to a file, and all the files are combined together using some suitable tool, which is reached through the `movie` function in Easyviz. All necessary steps will be apparent in the complete program below, but before diving into the code we need to comment upon a couple of issues with setting up the plot command for animations.

The underlying plotting program will normally adjust the axis to the maximum and minimum values of the curve if we do not specify the axis ranges explicitly. For an animation such automatic axis adjustment is misleading - the axis ranges must be fixed to avoid a jumping axis. The relevant values for the axis range is the minimum and maximum value of f . The minimum value is zero, while the maximum value appears for $x = m$ and increases with decreasing s . The range of the y axis must therefore be $[0, f(m; m, \min s)]$.

The function f is defined for all $-\infty < x < \infty$, but the function value is very small already $3s$ away from $x = m$. We may therefore limit the x coordinates to $[m - 3s, m + 3s]$.

Now we are ready to take a look at the complete code for animating how the Gaussian function evolves as the s parameter is decreased from 2 to 0.2:

```
from scitools.std import *
import time

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)

# show the movie on the screen
# and make hardcopies of frames simultaneously:
counter = 0
for s in s_values:
    y = f(x, m, s)
    plot(x, y, axis=[x[0], x[-1], -0.1, max_f],
         xlabel='x', ylabel='f', legend='s=%4.2f' % s,
         hardcopy='tmp%04d.png' % counter)
    counter += 1
    #time.sleep(0.2) # can insert a pause to control movie speed

# make movie file the simplest possible way:
movie('tmp*.png')
```

Note that the s values are decreasing (`linspace` handles this automatically if the start value is greater than the stop value). Also note

that we, simply because we think it is visually more attractive, let the y axis go from -0.1 although the f function is always greater than zero.

Remarks on Filenames. For each frame (plot) in the movie we store the plot in a file. The different files need different names and an easy way of referring to the set of files in right order. We therefore suggest to use filenames of the form `tmp0001.png`, `tmp0002.png`, `tmp0003.png`, etc. The `printf` format `04d` pads the integers with zeros such that 1 becomes 0001, 13 becomes 0013 and so on. The expression `tmp*.png` will now expand (by an alphabetic sort) to a list of all files in proper order. Without the padding with zeros, i.e., names of the form `tmp1.png`, `tmp2.png`, ..., `tmp12.png`, etc., the alphabetic order will give a wrong sequence of frames in the movie. For instance, `tmp12.png` will appear before `tmp2.png`.

Note that the names of plot files specified when making hardcopies must be consistent with the specification of names in the call to `movie`. Typically, one applies a Unix wildcard notation in the call to `movie`, say `plotfile*.eps`, where the asterisk will match any set of characters. When specifying hardcopies, we must then use a filename that is consistent with `plotfile*.eps`, that is, the filename must start with `plotfile` and end with `.eps`, but in between these two parts we are free to construct (e.g.) a frame number padded with zeros.

We recommend to always remove previously generated plot files before a new set of files is made. Otherwise, the movie may get old and new files mixed up. The following Python code removes all files of the form `tmp*.png`:

```
import glob, os
for filename in glob.glob('tmp*.png'):
    os.remove(filename)
```

These code lines should be inserted at the beginning of the code example above. Alternatively, one may store all plotfiles in a subfolder and later delete the subfolder. Here is a suitable code segment:

```
import shutil, os
subdir = 'temp' # subfolder for plot files
if os.path.isdir(subdir): # does the subfolder already exist?
    shutil.rmtree(subdir) # delete the whole folder
os.mkdir(subdir) # make new subfolder
os.chdir(subdir) # move to subfolder
# do all the plotting
# make movie
os.chdir(os.pardir) # optional: move up to parent folder
```

Movie Formats. Having a set of (e.g.) `tmp*.png` files, one can simply generate a movie by a `movie('tmp*.png')` call. The `movie` function generates a movie file called `movie.avi` (AVI format), `movie.mpeg` (MPEG format), or `movie.gif` (animated GIF format) in the current working

directory. The movie format depends on the encoders found on your machine.

You can get complete control of the movie format and the name of the movie file by supplying more arguments to the `movie` function. First, let us generate an animated GIF file called `tmpmovie.gif`:

```
movie('tmp_*.eps', encoder='convert', fps=2,  
      output_file='tmpmovie.gif')
```

The generation of animated GIF images applies the `convert` program from the ImageMagick suite. This program must of course be installed on the machine. The argument `fps` stands for frames per second so here the speed of the movie is slow in that there is a delay of half a second between each frame (image file). To view the animated GIF file, one can use the `animate` program (also from ImageMagick) and give the movie file as command-line argument. One can alternatively put the GIF file in a web page in an `IMG` tag such that a browser automatically displays the movie.

An MPEG movie can be generated by the call

```
movie('tmp_*.eps', encoder='ffmpeg', fps=4,  
      output_file='tmpmovie1.mpeg',
```

Alternatively, we may use the `ppmtompeg` encoder from the Netpbm suite of image manipulation tools:

```
movie('tmp_*.eps', encoder='ppmtompeg', fps=24,  
      output_file='tmpmovie2.mpeg',
```

The `ppmtompeg` supports only a few (high) frame rates.

The next sample call to `movie` uses the `Mencoder` tool and specifies some additional arguments (video codec, video bitrate, and the quantization scale):

```
movie('tmp_*.eps', encoder='mencoder', fps=24,  
      output_file='tmpmovie.mpeg',  
      vcodec='mpeg2video', vbitrate=2400, qscale=4)
```

Playing movie files can be done by a lot of programs. Windows Media Player is a default choice on Windows machines. On Unix, a variety of tools can be used. For animated GIF files the `animate` program from the ImageMagick suite is suitable, or one can simply show the file in a web page with the HTML command ``. AVI and MPEG files can be played by, for example, the `myplayer`, `vlc`, or `totem` programs.

4.3.8 Advanced Easyviz Topics

The information in the previous sections aims at being sufficient for the daily work with plotting curves. Sometimes, however, one wants to

fine-control the plot or how Easyviz behaves. First, we explain how to set the backend. Second, we tell how to speed up the `WILL BE REPLACED BY ptex2tex` from `scitools.std import *` statement. Third, we show how to operate with the plotting program directly and using plotting program-specific advanced features. Fourth, we explain how the user can grab `Figure` and `Axis` objects that Easyviz produces "behind the curtain".

Controlling the Backend. The Easyviz backend can either be set in a config file (see Config File below), by importing a special backend in the program, or by adding a command-line option

```
--SCITTOOLS_easyviz_backend name
```

where `name` is the name of the backend: `gnuplot`, `vtk`, `matplotlib`, etc. Which backend you choose depends on what you have available on your computer system and what kind of plotting functionality you want.

An alternative method is to import a specific backend in a program. Instead of the `from scitools.std import *` statement one writes

```
from numpy import *
from scitools.easyviz.gnuplot_ import * # work with Gnuplot
# or
from scitools.easyviz.vtk_ import *      # work with VTK
```

Note the trailing underscore in the module names for the various backends.

Easyviz is a subpackage of SciTools, and the the SciTools configuration file, called `scitools.cfg` has a section `[easyviz]` where the backend in Easyviz can be set:

```
[easyviz]
backend = vtk
```

A `.scitools.cfg` file can be placed in the current working folder, thereby affecting plots made in this folder, or it can be located in the user's home folder, which will affect all plotting sessions for the user in question. There is also a common SciTools config file `scitools.cfg` for the whole site (located in the directory where the `scitools` package is installed).

The following program prints a list of the names of the available backends on your computer system:

```
from scitools.std import *
backends = available_backends()
print 'Available backends:', backends
```

There will be quite some output explaining the missing backends and what must be installed to use these backends.

Importing Just Easyviz. The `from scitools.std import *` statement imports many modules and packages::

```

from numpy import *
from scitools.numpyutils import * # some convenience functions
from numpy.lib.scimath import *
from scipy import *              # if scipy is installed
import sys, operator, math
from scitools.StringFunction import StringFunction
from glob import glob

```

The `scipy` import can take some time and lead to slow start-up of plot scripts. A more minimalistic import for curve plotting is

```

from scitools.easyviz import *
from numpy import *

```

Alternatively, one can edit the `scitools.cfg` configure file or add one's own `.scitools.cfg` file with redefinition of selected options, such as `load` in the `scipy` section. The user `.scitools.cfg` must be placed in the folder where the plotting script in action resides, or in the user's home folder. Instead of editing a configuration file, one can just add the command-line argument `--SCITOOLS_scipy_load no` to the curve plotting script (all sections/options in the configuration file can also be set by such command-line arguments).

Working with the Plotting Program Directly. Easyviz supports just the most common plotting commands, typically the commands you use "95 percent" of the time when exploring curves. Various plotting packages have lots of additional commands for different advanced features. When Easyviz does not have a command that supports a particular feature, one can grab the Python object that communicates with the underlying plotting program (known as "backend") and work with this object directly, using plotting program-specific command syntax. Let us illustrate this principle with an example where we add a text and an arrow in the plot, see Figure 4.9.

Easyviz does not support arrows at arbitrary places inside the plot, but Gnuplot does. If we use Gnuplot as backend, we may grab the Gnuplot object and issue Gnuplot commands to this object directly. Here is an example of the typical recipe, written after the core of the plot is made in the ordinary (plotting program-independent) way:

```

g = get_backend()
if backend == 'gnuplot':
    # g is a Gnuplot object, work with Gnuplot commands directly:
    g('set label "global maximum" at 0.1,0.5 font "Times,18"')
    g('set arrow from 0.5,0.48 to 0.98,0.37 linewidth 2')
    g.refresh()
    g.hardcopy('tmp2.eps') # make new hardcopy

```

We refer to the Gnuplot manual for the features of this package and the syntax of the commands. The idea is that you can quickly generate plots with Easyviz using standard commands that are independent of the underlying plotting package. However, when you need advanced

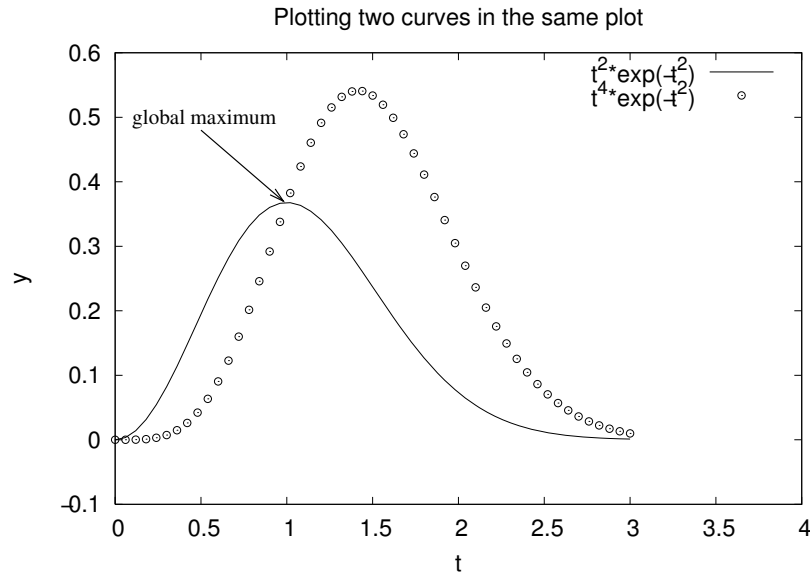


Fig. 4.9 Illustration of a text and an arrow using Gnuplot-specific commands.

features, you must add plotting package-specific code as shown above. This principle makes Easyviz a light-weight interface, but without limiting the available functionality of various plotting programs.

Working with Axis and Figure Objects. Easyviz supports the concept of Axis objects, as in Matlab. The Axis object represents a set of axes, with curves drawn in the associated coordinate system. A figure is the complete physical plot. One may have several axes in one figure, each axis representing a subplot. One may also have several figures, represented by different windows on the screen or separate hardcopies.

Users with Matlab experience may prefer to set axis labels, ranges, and the title using an Axis object instead of providing the information in separate commands or as part of a plot command. The `gca` (get current axis) command returns an Axis object, whose `set` method can be used to set axis properties:

```
plot(t, y1, 'r-', t, y2, 'bo',
      legend=('t^2*exp(-t^2)', 't^4*exp(-t^2)'),
      hardcopy='tmp2.eps')

ax = gca() # get current Axis object
ax.setp(xlabel='t', ylabel='y',
        axis=[0, 4, -0.1, 0.6],
        title='Plotting two curves in the same plot')
show() # show the plot again after ax.setp actions
```

The `figure()` call makes a new figure, i.e., a new window with curve plots. Figures are numbered as 1, 2, and so on. The command `figure(3)` sets the current figure object to figure number 3.

Suppose we want to plot our y_1 and y_2 data in two separate windows. We need in this case to work with two `Figure` objects:

```
plot(t, y1, 'r-', xlabel='t', ylabel='y',
     axis=[0, 4, -0.1, 0.6])

figure() # new figure

plot(t, y2, 'bo', xlabel='t', ylabel='y')
```

We may now go back to the first figure (with the y_1 data) and set a title and legends in this plot, show the plot, and make a PostScript version of the plot:

```
figure(1) # go back to first figure
title('One curve')
legend('t^2*exp(-t^2)')
show()
hardcopy('tmp2_1.eps')
```

We can also adjust figure 2:

```
figure(2) # go to second figure
title('Another curve')
hardcopy('tmp2_2.eps')
show()
```

The current `Figure` object is reached by `gcf` (get current figure), and the `dump` method dumps the internal parameters in the `Figure` object:

```
fig = gcf(); print fig.dump()
```

These parameters may be of interest for troubleshooting when Easyviz does not produce what you expect.

Let us then make a third figure with two plots, or more precisely, two axes: one with y_1 data and one with y_2 data. Easyviz has a command `subplot(r,c,a)` for creating r rows and c columns and set the current axis to axis number a . In the present case `subplot(2,1,1)` sets the current axis to the first set of axis in a "table" with two rows and one column. Here is the code for this third figure:

```
figure() # new, third figure
# plot y1 and y2 as two axis in the same figure:
subplot(2, 1, 1)
plot(t, y1, xlabel='t', ylabel='y')
subplot(2, 1, 2)
plot(t, y2, xlabel='t', ylabel='y')
title('A figure with two plots')
show()
hardcopy('tmp2_3.eps')
```

If we need to place an axis at an arbitrary position in the figure, we must use the command

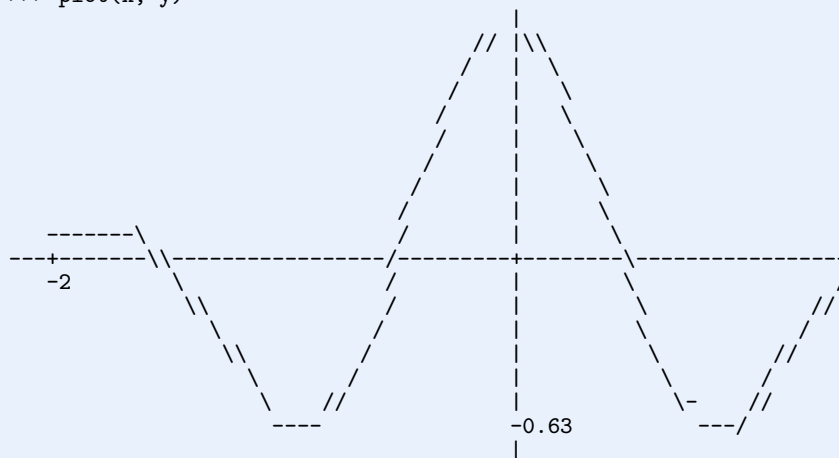
```
ax = axes(viewport=[left, bottom, width, height])
```

The four parameters `left`, `bottom`, `width`, `height` are location values between 0 and 1 ((0,0) is the lower-left corner and (1,1) is the upper-right corner). However, this might be a bit different in the different backends (see the documentation for the backend in question).

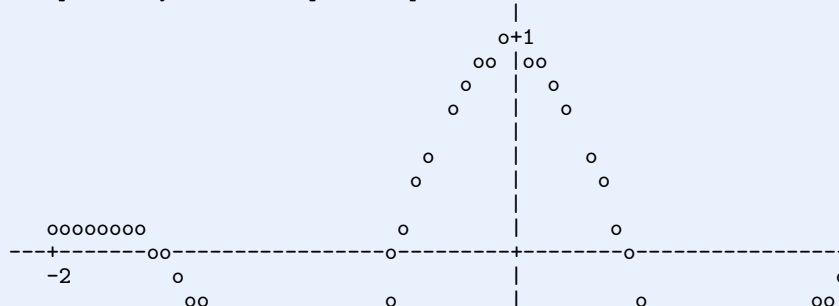
4.3.9 Curves in Pure Text

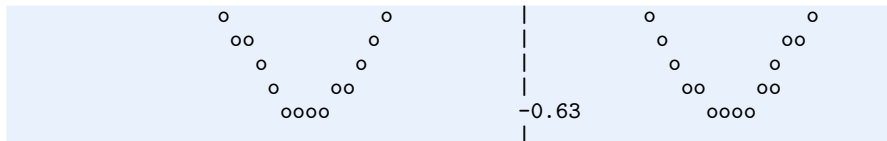
Sometimes it can be desirable to show a graph in pure ASCII text, e.g., as part of a trial run of a program included in the program itself (cf. the introduction to Chapter 1.8), or a graph can be needed in a doc string. For such purposes we have slightly extended a module by Imri Goldberg (`aplotter.py`) and included it as a module in SciTools. Running `pydoc` on `scitools.aplotter` describes the capabilities of this type of primitive plotting. Here we just give an example of what it can do:

```
>>> from scitools.aplotter import plot
>>> from numpy import linspace, exp, cos, pi
>>> x = linspace(-2, 2, 81)
>>> y = exp(-0.5*x**2)*cos(pi*x)
>>> plot(x, y)
```



```
>>> # plot circles at data points only:
>>> plot(x, y, dot='o', plot_slope=False)
```





```
>>> p = plot(x, y, output=str)    # store plot in a string:
>>> print p
```

(The last 13 characters of the output lines are here removed to make the lines fit the maximum textwidth of this book.)

4.4 Plotting Difficulties

The previous examples on plotting functions demonstrate how easy it is to make graphs. Nevertheless, the shown techniques might easily fail to plot some functions correctly unless we are careful. Next we address two types of difficult functions: piecewisely defined functions and rapidly varying functions.

4.4.1 Piecewisely Defined Functions

A piecewisely defined function has different function definitions in different intervals along the x axis. The resulting function, made up of pieces, may have discontinuities in the function value or in derivatives. We have to be very careful when plotting such functions, as the next two examples will show. The problem is that the plotting mechanism draws straight lines between coordinates on the function's curve, and these straight lines may not yield a satisfactory visualization of the function. The first example has a discontinuity in the function itself at one point, while the other example has a discontinuity in the derivative at three points.

Example: The Heaviside Function. Let us plot the Heaviside function defined in (2.18) on page 108. The most natural way to proceed is first to define the function as

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

The standard plotting procedure where we define a coordinate array x and do a

```
y = H(x)
plot(x, y)
```

fails with this $H(x)$ function. The test $x < 0$ results in an array where each element is `True` or `False` depending on whether $x[i] < 0$ or not.

A `ValueError` exception is raised when we use this resulting array in an if test:

```
>>> 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()
```

The suggestion of using the `any` or `all` methods do not help because this is not what we are interested in:

```
>>> b.any() # True if any element in b is True
True
>>> b.all() # True if all elements in b are True
False
```

We want to take actions element by element depending on whether `x[i] < 0` or not.

There are three ways to find a remedy to our problems with the if `x < 0` test: (i) we can write an explicit loop for computing the elements, (ii) we can use a tool for automatically vectorize `H(x)`, or (iii) we can manually vectorize the `H(x)` function. All three methods will be illustrated next.

Loop. The following function works well for arrays if we insert a simple loop over the array elements (such that `H(x)` operates on scalars only):

```
def H_loop(x):
    r = zeros(len(x))
    for i in xrange(len(x)):
        r[i] = H(x[i])
    return r

x = linspace(-5, 5, 6)
y = H_loop(x)
```

This `H_loop` version of `H` is sufficient for plotting the Heaviside function. The next paragraph explains other ways of making versions of `H(x)` that work for array arguments.

Automatic Vectorization. Numerical Python contains a method for automatically vectorizing a Python function that works with scalars (pure numbers) as arguments:

```
from numpy import vectorize
H_vec = vectorize(H)
```

The `H_vec(x)` function will now work with vector/array arguments `x`. Unfortunately, such automatically vectorized functions are often as slow as the explicit loop shown above.

Manual Vectorization. (Note: This topic is considered advanced and at another level than the rest of the book.) To allow array arguments in our Heaviside function *and* get the increased speed that one associates with vectorization, we have to rewrite the H function completely. The mathematics must now be expressed by functions from the Numerical Python library. In general, this type of rewrite is non-trivial and requires knowledge of and experience with the library. Fortunately, functions of the form

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

can in general be vectorized quite simply as

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

The `where` function returns an array of the same length as `condition`, and element no. `i` equals `x1[i]` if `condition[i]` is `True`, and `x2[i]` otherwise. With Python loops we can express this principle as

```
r = zeros(len(condition)) # array returned from where(...)
for i in xrange(condition):
    r[i] = x1[i] if condition[i] else x2[i]
```

The `x1` and `x2` variables can be pure numbers or arrays of the same length as `x`.

In our case we can use the `where` function as follows:

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

Plotting the Heaviside Function. Since the Heaviside function consists of two flat lines, one may think that we do not need many points along the x axis to describe the curve. Let us try only five points:

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

However, so few x points are not able to describe the jump from 0 to 1 at $x = 0$, as shown by the solid line in Figure 4.10a. Using more points, say 50 between -10 and 10 ,

```
x2 = linspace(-10, 10, 50)
plot(x, Hv(x), 'r', x2, Hv(x2), 'b',
     legend=('5 points', '50 points'),
     axis=[x[0], x[-1], -0.1, 1.1])
```

makes the curve look better, as you can see from the dotted line in Figure 4.10a. However, the step is still not vertical. This time the point $x = 0$ was not among the coordinates so the step goes from $x = -0.2$ to $x = 0.2$. More points will improve the situation. Nevertheless, the best is to draw two flat lines directly: from $(-10, 0)$ to $(0, 0)$, then to $(0, 1)$ and then to $(10, 1)$:

```
plot([-10, 0, 0, 10], [0, 0, 1, 1],
     axis=[x[0], x[-1], -0.1, 1.1])
```

The result is shown in Figure 4.10b.

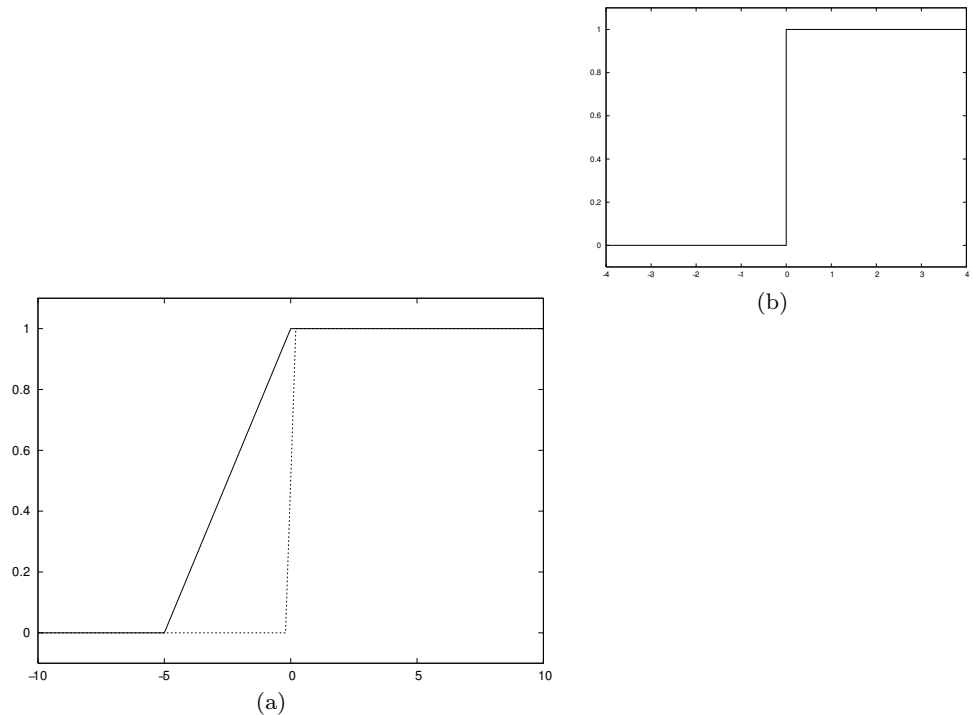


Fig. 4.10 Plot of the Heaviside function: (a) using equally spaced x points (5 and 50); (b) using a “double point” at $x = 0$.

Some will argue that the plot of $H(x)$ should not contain the vertical line from $(0, 0)$ to $(0, 1)$, but only two horizontal lines. To make such a plot, we must draw two distinct curves, one for each horizontal line:

```
plot([-10,0], [0,0], 'r-', [0,10], [1,1], 'r-',
     axis=[x[0], x[-1], -0.1, 1.1])
```

Observe that we must specify the same line style for both lines (curves), otherwise they would by default get different color on the screen and different line type in a hardcopy. We remark, however, that discontinuous functions like $H(x)$ are often visualized with vertical lines at the jumps, as we do in Figure 4.10b.

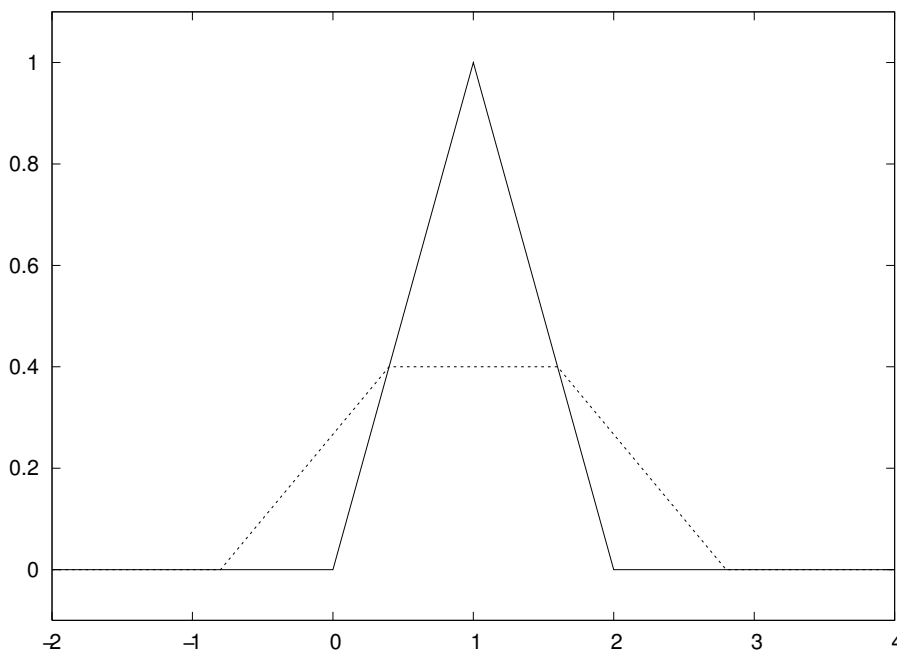


Fig. 4.11 Plot of a “hat” function. The solid line shows the exact function, while the dashed line arises from using inappropriate points along the x axis.

Example: A “Hat” Function. Let us plot the “hat” function $N(x)$, defined by (2.5) on page 89. The corresponding Python implementation $N(x)$ shown right after (2.5) does not work with array arguments x because the boolean expressions, like $x < 0$, are arrays and they cannot yield a single `True` or `False` value for the `if` tests. The simplest solution is to use `vectorize`, as explained for the Heaviside function above⁸:

```
N_vec = vectorize(N)
```

A manual rewrite, yielding a faster vectorized function, is more demanding than for the Heaviside function because we now have multiple branches in the `if` test. One attempt may be⁹

```
def Nv(x):
    r = where(x < 0, 0.0, x)
    r = where(0 <= x < 1, x, r)
    r = where(1 <= x < 2, 2-x, r)
    r = where(x >= 2, 0.0, r)
    return r
```

However, the condition like `0 <= x < 1`, which is equivalent to `0 <= x` and `x < 1`, does not work because the `and` operator does not work with array arguments. All operators in Python (`+`, `-`, `and`, `or`, etc.)

⁸ It is important that $N(x)$ return `float` and not `int` values, otherwise the vectorized version will produce `int` values and hence be incorrect.

⁹ This is again advanced material.

are available as pure functions in a module `operator` (`operator.add`, `operator.sub`, `operator.and_`, `operator.or_`¹⁰, etc.). A working `Nv` function must apply `operator.and_` instead:

```
def Nv(x):
    r = where(x < 0, 0.0, x)
    import operator
    condition = operator.and_(0 <= x, x < 1)
    r = where(condition, x, r)
    condition = operator.and_(1 <= x, x < 2)
    r = where(condition, 2-x, r)
    r = where(x >= 2, 0.0, r)
    return r
```

A second, alternative rewrite is to use boolean expressions in indices:

```
def Nv(x):
    r = x.copy() # avoid modifying x in-place
    r[x < 0.0] = 0.0
    condition = operator.and_(0 <= x, x < 1)
    r[condition] = x[condition]
    condition = operator.and_(1 <= x, x < 2)
    r[condition] = 2-x[condition]
    r[x >= 2] = 0.0
    return r
```

Now to the computation of coordinate arrays for the plotting. We may use an explicit loop over all array elements, or the `N_vec` function, or the `Nv` function. An approach without thinking about vectorization too much could be

```
x = linspace(-2, 4, 6)
plot(x, N_vec(x), 'r', axis=[x[0], x[-1], -0.1, 1.1])
```

This results in the dashed line in Figure 4.11. What is the problem? The problem lies in the computation of the `x` vector, which does not contain the points $x = 1$ and $x = 2$ where the function makes significant changes. The result is that the “hat” is “flattened”. Making an `x` vector with all critical points in the function definitions, $x = 0, 1, 2$, provides the necessary remedy, either

```
x = linspace(-2, 4, 7)
```

or the simple

```
x = [-2, 0, 1, 2, 4]
```

Any of these `x` alternatives and a `plot(x, N_vec(x))` will result in the solid line in Figure 4.11, which is the correct visualization of the $N(x)$ function.

¹⁰ Recall that `and` and `or` are reserved keywords, see page 10, so a module or program cannot have variables or functions with these names. To circumvent this problem, the convention is to add a trailing underscore to the name.

4.4.2 Rapidly Varying Functions

Let us now visualize the function $f(x) = \sin(1/x)$, using 10 and 1000 points:

```
def f(x):
    return sin(1.0/x)

x1 = linspace(-1, 1, 10)
x2 = linspace(-1, 1, 1000)
plot(x1, f(x1), label='%d points' % len(x))
plot(x2, f(x2), label='%d points' % len(x))
```

The two plots are shown in Figure 4.12. Using only 10 points gives a completely wrong picture of this function, because the function oscillates faster and faster as we approach the origin. With 1000 points we get an impression of these oscillations, but the accuracy of the plot in the vicinity of the origin is still poor. A plot with 100000 points has better accuracy, in principle, but the extremely fast oscillations near the origin just drown in black ink (you can try it out yourself).

Another problem with the $f(x) = \sin(1/x)$ function is that it is easy to define an x vector that contains $x = 0$, such that we get division by zero. Mathematically, the $f(x)$ function has a singularity at $x = 0$: it is difficult to define $f(0)$, so one should exclude this point from the function definition and work with a domain $x \in [-1, -\epsilon] \cup [\epsilon, 1]$, with ϵ chosen small.

The lesson learned from these examples is clear: You must investigate the function to be visualized and make sure that you use an appropriate set of x coordinates along the curve. A relevant first step is to double the number of x coordinates and check if this changes the plot. If not, you probably have an adequate set of x coordinates.

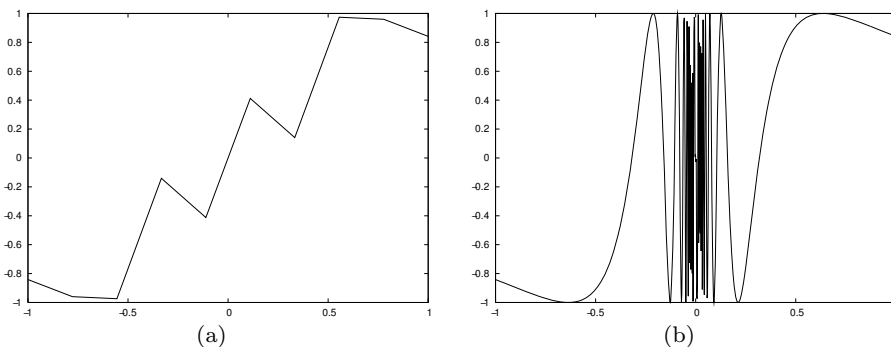


Fig. 4.12 Plot of the function $\sin(1/x)$ with (a) 10 points and (b) 1000 points.

4.4.3 Vectorizing StringFunction Objects

The `StringFunction` object described in Chapter 3.1.4 does unfortunately not work with array arguments unless we explicitly tell the object to do so. The recipe is very simple. Say `f` is some `StringFunction` object. To allow array arguments we must first call `f.vectorize(globals())` once:

```
f = StringFunction(formula)
x = linspace(0, 1, 30)

# f(x) will in general not work

from numpy import *
f.vectorize(globals())
# now f works with array arguments:
values = f(x)
```

It is important that you import everything from `numpy` or `scitools.std` *before* you call `f.vectorize`. We suggest to take the `f.vectorize` call as a magic recipe¹¹.

Even after calling `f.vectorize(globals())` a `StringFunction` object may face problems with vectorization. One example is a piecewise constant function as specified by a string expression `'1 if x > 2 else 0'`. One remedy is to use the vectorized version of an if test: `'where(x > 2, 1, 0)'`. For an average user of the program this construct is not at all obvious so a more user-friendly solution is to apply `vectorize` from `numpy`:

```
f = vectorize(f) # vectorize a StringFunction f
```

The line above is therefore the most general (but also the slowest) way of vectorizing a `StringFunction` object. After that call, `f` is no more a `StringFunction` object, but `f` behaves as a (vectorized) function. The `vectorize` tool from `numpy` can be used to allow any Python function taking scalar arguments to also accept array arguments.

To get better speed, one can use `vectorize(f)` only in the case the formula in `f` contains an inline if test (e.g., recognized by the string `'else '` inside the formula). Otherwise, we use `f.vectorize`. The formula in `f` is obtained by `str(f)` so we can test

¹¹ Some readers still want to know what the problem is. Inside the `StringFunction` module we need to have access to mathematical functions for expressions like `sin(x)*exp(x)` to be evaluated. These mathematical functions are by default taken from the `math` module and hence they do not work with array arguments. If the user, in the main program, has imported mathematical functions that work with array arguments, these functions are registered in a dictionary returned from `globals()`. By the `f.vectorize` call we supply the `StringFunction` module with the user's global namespace so that the evaluation of the string expression can make use of mathematical functions for arrays.

```
if ' else ' in str(f):
    f = vectorize(f)
else:
    f.vectorize(globals())
```

4.5 More on Numerical Python Arrays

This section lists some more advanced but useful operations with Numerical Python arrays.

4.5.1 Copying Arrays

Let `x` be an array. The statement `a = x` makes `a` refer to the same array as `x`. Changing `a` will then also affect `x`:

```
>>> x = array([1, 2, 3.5])
>>> a = x
>>> a[-1] = 3 # this changes x[-1] too!
>>> x
array([ 1.,  2.,  3.])
```

Changing `a` without changing `x` requires `a` to be a copy of `x`:

```
>>> a = x.copy()
>>> a[-1] = 9
>>> a
array([ 1.,  2.,  9.])
>>> x
array([ 1.,  2.,  3.])
```

4.5.2 In-Place Arithmetics

Let `a` and `b` be two arrays of the same shape. The expression `a += b` means `a = a + b`, but this is not the complete story. In the statement `a = a + b`, the sum `a + b` is first computed, yielding a new array, and then the name `a` is bound to this new array. The old array `a` is lost unless there are other names assigned to this array. In the statement `a += b`, elements of `b` are added directly into the elements of `a` (in memory). There is no hidden intermediate array as in `a = a + b`. This implies that `a += b` is more efficient than `a = a + b` since Python avoids making an extra array. We say that the operators `+=`, `*=`, and so on, perform *in-place* arithmetics in arrays.

Consider the compound array expression

```
a = (3*x**4 + 2*x + 4)/(x + 1)
```

The computation actually goes as follows with seven hidden arrays for storing intermediate results:

1. `r1 = x**4`
2. `r2 = 3*r1`
3. `r3 = 2*x`
4. `r4 = r2 + r3`
5. `r5 = r4 + 4`
6. `r6 = x + 1`
7. `r7 = r5/r6`
8. `a = r7`

With in-place arithmetics we can get away with creating three new arrays, at a cost of a significantly less readable code:

```
a = x.copy()
a **= 4
a *= 3
a += 2*x
a += 4
a /= x + 1
```

The three extra arrays in this series of statement arise from copying `x`, and computing the right-hand sides `2*x` and `x+1`.

Quite often in computational science and engineering, a huge number of arithmetics is performed on very large arrays, and then saving memory and array allocation time by doing in-place arithmetics is important.

The mix of assignment and in-place arithmetics makes it easy to make unintended changes of more than one array. For example, this code changes `x`:

```
a = x
a += y
```

since `a` refers to the same array as `x` and the change of `a` is done in-place.

4.5.3 Allocating Arrays

We have already seen that the `zeros` function is handy for making a new array `a` of a given size. Very often the size and the type of array elements have to match another existing array `x`. We can then either copy the original array, e.g.,

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

and fill elements in `a` with the right new values, or we can say


```
a = zeros(x.shape, x.dtype)
```

The attribute `x.dtype` holds the array element type (`dtype` for data type), and as mentioned before, `x.shape` is a tuple with the array dimensions.

Sometimes we may want to ensure that an object is an array, and if not, turn it into an array. The `asarray` function is useful in such cases:

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

Nothing is copied if `a` already is an array, but if `a` is a list or tuple, a new array with a copy of the data is created.

4.5.4 Generalized Indexing

Chapter 4.2.2 shows how slices can be used to extract and manipulate subarrays. The slice `f:t:i` corresponds to the index set `f`, `f+i`, `f+2*i`, ... up to, but not including, `t`. Such an index set can be given explicitly too: `a[range(f,t,i)]`. That is, the integer list from `range` can be used as a set of indices. In fact, any integer list or integer array can be used as index:

```
>>> 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.])
```

We can also use boolean arrays to generate an index set. The indices in the set will correspond to the indices for which the boolean array has `True` values. This functionality allows expressions like `a[x<m]`. Here are two examples, continuing the previous interactive session:

```
>>> a[a < 0] # pick out the negative elements of a
array([-2., -2.])
>>> a[a < 0] = a.max()
>>> a
array([ 1., 10., 10.,  4.,  5., 10., 10., 10.])
>>> # replace elements where a is 10 by the first
>>> # elements from another array/list:
>>> a[a == 10] = [10, 20, 30, 40, 50, 60, 70]
>>> a
array([ 1., 10., 20.,  4.,  5., 30., 40., 50.])
```

Generalized indexing using integer arrays or lists is important for vectorized initialization of array elements.

4.5.5 Testing for the Array Type

Inside an interactive Python shell you can easily check an object's type using the `type` function (see Chapter 1.5.2). In case of a Numerical Python array, the type name is `ndarray`:

```
>>> a = linspace(-1, 1, 3)
>>> a
array([-1.,  0.,  1.])
>>> type(a)
<type 'numpy.ndarray'>
```

Sometimes you need to test if a variable is an `ndarray` or a `float` or `int`. The `isinstance` function was made for this purpose:

```
>>> isinstance(a, ndarray)
True
>>> type(a) == ndarray
True
>>> isinstance(a, (float,int)) # float or int?
False
```

A typical use of `isinstance` is shown next.

Example: Vectorizing a Constant Function. Suppose we have a constant function,

```
def f(x):
    return 2
```

This function accepts an array argument `x`, but will return a `float` while a vectorized version of the function should return an array of the same shape as `x` where each element has the value 2. The vectorized version can be realized as

```
def fv(x):
    return zeros(x.shape, x.dtype) + 2
```

The optimal vectorized function would be one that works for both a scalar and an array argument. We must then test on the argument type:

```
def f(x):
    if isinstance(x, (float, int)):
        return 2
    else: # assume array
        return zeros(x.shape, x.dtype) + 2
```

A more foolproof solution is to also test for an array and raise an exception if `x` is neither a scalar nor an array:

```
def f(x):
    if isinstance(x, (float, int)):
        return 2
    elif isinstance(x, ndarray):
        return zeros(x.shape, x.dtype) + 2
```

```

else:
    raise TypeError\
        ('x must be int, float or ndarray, not %s' % type(x))

```

4.5.6 Equally Spaced Numbers

We have used the `linspace` function heavily so far in this chapter, but there are some related, useful functions that also produce a sequence of uniformly spaced numbers. In `numpy` we have the `arange` function, where `arange(start, stop, inc)` creates an array with the numbers `start`, `start+inc`, `start+2*inc`, ..., `stop-inc`. Note that the upper limit `stop` is not included in the set of numbers (i.e., `arange` behaves as `range` and `xrange`):

```

>>> arange(-1, 1, 0.5)
array([-1. , -0.5,  0. ,  0.5])

```

Because of round-off errors the upper limit can be included in the array. You can try out

```

for i in range(1,500):
    a = arange(0, 1, 1.0/i)
    print i, a[-1]

```

Now and then, the last element `a[-1]` equals 1, which is wrong behavior! We therefore recommend to stay away from `arange`. A substitute for `arange` is the function `seq` from `SciTools`: `seq(start, stop, inc)` generates real numbers starting with `start`, ending with `stop`, and with increments of `inc`.

```

>>> from scitools.std import *
>>> seq(-1, 1, 0.5)
array([-1. , -0.5,  0. ,  0.5,  1. ])

```

For integers, a similar function, called `iseq`, is available. With `iseq(start, stop, inc)` we get the integers `start`, `start+inc`, `start+2*inc`, and so on up to `stop`. Unlike `range(start, stop, inc)` and `xrange(start, stop, inc)`, the upper limit `stop` is part of the sequence of numbers. This feature makes `iseq` more appropriate than `range` and `xrange` in many implementations of mathematical algorithms where there is an index whose limits are specified in the algorithm, because with `iseq` we get a one-to-one correspondence between the algorithm and the Python code. Here is an example: a vector x of length n , compute

$$a_i = f(x_{i+1}) - f(x_{i-1}) \text{ for } i = 1, \dots, n-2.$$

A Python implementation with `iseq` reads

```
for i in iseq(1, n-2):
    a[i] = f(x[i+1]) - f(x[i-1])
```

while with `range` we must write

```
for i in range(1, n-1):
    a[i] = f(x[i+1]) - f(x[i-1])
```

Direct correspondence between the mathematics and the code is very important and makes it much easier to find bugs by comparing the code and the mathematical description, line by line.

4.5.7 Compact Syntax for Array Generation

There is a special compact syntax `r_[f:t:s]` for the `linspace` and `arange` functions:

```
>>> a = r_[-5:5:11j] # same as linspace(-5, 5, 11)
>>> print a
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

Here, `11j` means 11 coordinates (between -5 and 5, including the upper limit 5). That is, the number of elements in the array is given with the imaginary number syntax.

The `arange` equivalent reads

```
>>> a = r_[-5:5:1.0]
>>> print a
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.]
```

With 1 as step instead of 1.0 (`r_[-5:5:1]`) the elements in `a` become integers.

4.5.8 Shape Manipulation

The `shape` attribute in array objects holds the shape, i.e., the size of each dimension. A function `size` returns the total number of elements in an array. Here are a few equivalent ways of changing the shape of an array:

```
>>> a = linspace(-1, 1, 6)
>>> a.shape
(6,)
>>> a.size
6
>>> a.shape = (2, 3)
>>> a.shape
(2, 3)
>>> a.size # total no of elements
6
>>> a.shape = (a.size,) # reset shape
>>> a = a.reshape(3, 2) # alternative
>>> len(a) # no of rows
3
```

Note that `len(a)` always returns the length of the first dimension of an array.

4.6 Higher-Dimensional Arrays

4.6.1 Matrices and Arrays

Vectors appeared when mathematicians needed to calculate with a list of numbers. When they needed a table (or a list of lists in Python terminology), they invented the concept of *matrix* (singular) and *matrices* (plural). A table of numbers has the numbers ordered into rows and columns. One example is

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

This table with three rows and four columns is called a 3×4 matrix¹². If the symbol A is associated with this matrix, $A_{i,j}$ denotes the number in row number i and column number j . Counting rows and columns from 0, we have, for instance, $A_{0,0} = 0$ and $A_{2,3} = -2$. We can write a general $m \times n$ matrix A as

$$\begin{bmatrix} A_{0,0} & \cdots & A_{0,n-1} \\ \vdots & \ddots & \vdots \\ A_{m-1,0} & \cdots & A_{m-1,n-1} \end{bmatrix}$$

Matrices can be added and subtracted. They can also be multiplied by a scalar (a number), and there is a concept of “length”. The formulas are quite similar to those presented for vectors, but the exact form is not important here.

We can generalize the concept of table and matrix to *array*, which holds quantities with in general d indices. Equivalently we say that the array has rank d . For $d = 3$, an array A has elements with three indices: $A_{p,q,r}$. If p goes from 0 to $n_p - 1$, q from 0 to $n_q - 1$, and r from 0 to $n_r - 1$, the A array has $n_p \times n_q \times n_r$ elements in total. We may speak about the *shape* of the array, which is a d -vector holding the number of elements in each “array direction”, i.e., the number of elements for each index. For the mentioned A array, the shape is (n_p, n_q, n_r) .

The special case of $d = 1$ is a vector, and $d = 2$ corresponds to a matrix. When we program we may skip thinking about vectors and matrices (if you are not so familiar with these concepts from a mathematical point of view) and instead just work with arrays. The number

¹² Mathematicians don’t like this sentence, but it suffices for our purposes.

of indices corresponds to what is convenient in the programming problem we try to solve.

4.6.2 Two-Dimensional Numerical Python Arrays

Recall the nested list from Chapter 2.1.7, where `[C, F]` pairs are elements in a list `table`. The construction of `table` goes as follows:

```
>>> 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]]
```

Note that the `table` list is a nested list. This nested list can be turned into an array,

```
>>> table2 = array(table)
>>> print table2
[[-30. -22.]
 [-20.  -4.]
 [-10.  14.]]
>>> type(table2)
<type 'numpy.ndarray'>
```

We say that `table2` is a *two-dimensional* array, or an array of rank 2.

The `table` list and the `table2` array are stored very differently in memory. The `table` variable refers to a list object containing three elements. Each of these elements is a reference to a separate list object with two elements, where each element refers to a separate `float` object. The `table2` variable is a reference to a single array object that again refers to a consecutive sequence of bytes in memory where the six floating-point numbers are stored. The data associated with `table2` are found in one “chunk” in the computer’s memory, while the data associated with `table` are scattered around in memory. On today’s machines, it is much more expensive to find data in memory than to compute with the data. Arrays make the data fetching more efficient, and this is major reason for using arrays. However, this efficiency gain is only present for very large arrays, not for a 3×2 array.

Indexing a nested list is done in two steps, first the outer list is indexed, giving access to an element that is another list, and then this latter list is indexed:

```
>>> table[1][0]      # table[1] is [-20,4], whose index 0 holds -20
-20
```

This syntax works for two-dimensional arrays too:

```
>>> table2[1][0]
-20.0
```

but there is another syntax which is more common for arrays:

```
>>> table2[1,0]
-20.0
```

A two-dimensional array reflects a table and has a certain number of “rows” and “columns”. We refer to “rows” as the *first dimension* of the array and “columns” as the *second dimension*. These two dimensions are available as `table2.shape`:

```
>>> table2.shape
(3, 2)
```

Here, 3 is the number of “rows” and 2 is the number of “columns”.

A loop over all the elements in a two-dimensional array is usually expressed as two *nested* for loops, one for each index:

```
>>> for i in range(table2.shape[0]):
...     for j in range(table2.shape[1]):
...         print 'table2[%d,%d] = %g' % (i, j, table2[i,j])
...
table2[0,0] = -30
table2[0,1] = -22
table2[1,0] = -20
table2[1,1] = -4
table2[2,0] = -10
table2[2,1] = 14
```

An alternative (but less efficient) way of visiting each element in an array with any number of dimensions makes use of a single for loop:

```
>>> for index_tuple, value in ndenumerate(table2):
...     print 'index %s has value %g' % \
...         (index_tuple, table2[index_tuple])
...
index (0,0) has value -30
index (0,1) has value -22
index (1,0) has value -20
index (1,1) has value -4
index (2,0) has value -10
index (2,1) has value 14
```

In the same way as we can extract sublists of lists, we can extract subarrays of arrays using slices.

```
table2[0:table2.shape[0], 1] # 2nd column (index 1)
array([-22., -4., 14.])

>>> table2[0:, 1]           # same
array([-22., -4., 14.])

>>> table2[:, 1]            # same
array([-22., -4., 14.])
```

To illustrate array slicing further, we create a bigger array:

```
>>> t = linspace(1, 30, 30).reshape(5, 6)
>>> t
array([[ 1.,  2.,  3.,  4.,  5.,  6.],
       [ 7.,  8.,  9., 10., 11., 12.],
       [13., 14., 15., 16., 17., 18.]])
```

```

      [ 19., 20., 21., 22., 23., 24.],
      [ 25., 26., 27., 28., 29., 30.]]

>>> t[1:-1:2, 2:]
array([[ 9., 10., 11., 12.],
       [ 21., 22., 23., 24.]])

```

To understand the slice, look at the original `t` array and pick out the two rows corresponding to the first slice `1:-1:2`,

```

[ 7., 8., 9., 10., 11., 12.]
[ 19., 20., 21., 22., 23., 24.]

```

Among the rows, pick the columns corresponding to the second slice `2:`,

```

[ 9., 10., 11., 12.]
[ 21., 22., 23., 24.]

```

Another example is

```

>>> t[: -2, :-1:2]
array([[ 1., 3., 5.],
       [ 7., 9., 11.],
       [ 13., 15., 17.]])

```

4.6.3 Array Computing

The operations on vectors in Chapter 4.1.3 can quite straightforwardly be extended to arrays of any dimension. Consider the definition of applying a function $f(v)$ to a vector v : we apply the function to each element v_i in v . For a two-dimensional array A with elements $A_{i,j}$, $i = 0, \dots, m$, $j = 0, \dots, n$, the same definition yields

$$f(A) = (f(A_{0,0}), \dots, f(A_{m-1,0}), f(A_{1,0}), \dots, f(A_{m-1,n-1})).$$

For an array B with any rank, $f(B)$ means applying f to each array entry.

The asterisk operation from Chapter 4.1.3 is also naturally extended to arrays: $A * B$ means multiplying an element in A by the corresponding element in B , i.e., element (i, j) in $A * B$ is $A_{i,j} B_{i,j}$. This definition naturally extends to arrays of any rank, provided the two arrays have the same shape.

Adding a scalar to an array implies adding the scalar to each element in the array. Compound expressions involving arrays, e.g., $\exp(-A * 2) * A + 1$, work as for vectors. One can in fact just imagine that all the array elements are stored after each other in a long vector¹³, and the array operations can then easily be defined in terms of the vector operations from Chapter 4.1.3.

¹³ This is the way the array elements are stored in the computer's memory.

Remark. Readers with knowledge of matrix computations may ask how an expression like A^2 interfere with $A**2$. In matrix computing, A^2 is a matrix-matrix product, which is very different from squaring each element in A as $A**2$ or $A*A$ implies. Fortunately, the matrix computing operations look different from the array computing operations in mathematical typesetting. In a program, however, $A*A$ and $A**2$ are identical computations, but the first one could lead to a confusion with a matrix-matrix product AA . With Numerical Python the matrix-matrix product is obtained by `dot(A, A)`. The matrix-vector product Ax , where x is a vector, is computed by `dot(A, x)`.

4.6.4 Two-Dimensional Arrays and Functions of Two Variables

Given a function of two variables, say

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

we can plot this function by writing

```
x = y = linspace(-5, 5, 21) # coordinates in x and y direction
xv, yv = ndgrid(x, y)
z = f(xv, yv)
mesh(xv, yv, z)
```

There are two new things here: (i) the call to `ndgrid`, which is necessary to transform one-dimensional coordinate arrays in the x and y direction into arrays valid for evaluating `f` over a two-dimensional grid; and (ii) the plot function whose name now is `mesh`, which is one out of many plot functions for two-dimensional functions. Another plot type you can try out is

```
surf(xv, yv, z)
```

More material on visualizing $f(x, y)$ functions is found in the section "Visualizing Scalar Fields" in the Easyviz tutorial. This tutorial can be reached through the command `pydoc scitools.easyviz` in a terminal window.

4.6.5 Matrix Objects

This section only makes sense if you are familiar with basic linear algebra and the matrix concept. The arrays created so far have been of type `ndarray`. NumPy also has a matrix type called `matrix` or `mat` for one- and two-dimensional arrays. One-dimensional arrays are then extended with one extra dimension such that they become matrices, i.e., either a row vector or a column vector:

```
>>> x1 = array([1, 2, 3], float)
>>> x2 = matrix(x)           # or mat(x)
>>> x2                        # row vector
matrix([[ 1.,  2.,  3.]])
>>> x3 = mat(x).transpose()   # column vector
>>> x3
matrix([[ 1.],
        [ 2.],
        [ 3.]])

>>> type(x3)
<class 'numpy.core.defmatrix.matrix'>
>>> isinstance(x3, matrix)
True
```

A special feature of `matrix` objects is that the multiplication operator represents the matrix-matrix, vector-matrix, or matrix-vector product as we know from linear algebra:

```
>>> A = eye(3)                # identity matrix
>>> A
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> A = mat(A)
>>> A
matrix([[ 1.,  0.,  0.],
        [ 0.,  1.,  0.],
        [ 0.,  0.,  1.]])
>>> y2 = x2*A                  # vector-matrix product
>>> y2
matrix([[ 1.,  2.,  3.]])
>>> y3 = A*x3                  # matrix-vector product
>>> y3
matrix([[ 1.],
        [ 2.],
        [ 3.]])
```

One should note here that the multiplication operator between standard `ndarray` objects is quite different, as the next interactive session demonstrates.

```
>>> A*x1                       # no matrix-array product!
Traceback (most recent call last):
...
ValueError: matrices are not aligned

>>> # try array*array product:
>>> A = (zeros(9) + 1).reshape(3,3)
>>> A
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> A*x1                        # [A[0,:]*x1, A[1,:]*x1, A[2,:]*x1]
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
>>> B = A + 1
>>> A*B                          # element-wise product
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
>>> A = mat(A); B = mat(B)
```

```
>>> A*B                                # matrix-matrix product
matrix([[ 6.,  6.,  6.],
        [ 6.,  6.,  6.],
        [ 6.,  6.,  6.]])
```

Readers who are familiar with Matlab, or intend to use Python and Matlab together, should seriously think about programming with `matrix` objects instead of `ndarray` objects, because the `matrix` type behaves quite similar to matrices and vectors in Matlab. Nevertheless, `matrix` cannot be used for arrays of larger dimension than two.

4.7 Summary

4.7.1 Chapter Topics

This chapter has introduced computing with arrays and plotting curve data stored in arrays. The Numerical Python package contains lots of functions for array computing, including the ones listed in Table 4.1. The syntax for plotting curves makes use of Easyviz commands, which are very similar to those of Matlab, but a wide range of plotting packages can be used as “engines” for producing the graphics. Easyviz is a subpackage of the SciTools package, which is the key collection of software accompanying the present book.

Array Computing. When we apply a Python function $f(x)$ to a Numerical Python array x , the result is the same as if we apply f to each element in x separately. However, when f contains `if` statements, these are in general invalid if an array x enters the boolean expression. We then have to rewrite the function, often by applying the `where` function from Numerical Python.

Plotting Curves. A typical Easyviz command for plotting some curves with control of curve legends, axis, plot title, and also making a file with the plot, can be illustrated by

```
from scitools.std import * # get all we need for plotting

plot(t1, y1, 'r', # curve 1, red line
     t2, y2, 'b', # curve 2, blue line
     t3, y3, 'o', # curve 3, circles at data points
     axis=[t1[0], t1[-1], -1.1, 1.1],
     legend=('model 1', 'model 2', 'measurements'),
     xlabel='time', ylabel='force',
     hardcopy='forceplot_%04d.png' % counter)
```

Making Movies. Each frame in a movie must be a hardcopy of a plot, i.e., a plotfile. These plotfiles should have names containing a counter padded with leading zeros. One example may be the name

Table 4.1 Summary of important functionality for Numerical Python arrays.

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

specified as the `hardcopy` argument in the `plot` command above: `forceplot_0001.eps`, `forceplot_0002.eps`, etc., if `counter` runs from 1. Having the plotfiles with names on this form, we can make a movie file `movie.gif` with two frames per second by

```
movie('forceplot_*.png', encoder='convert',
      output_file='movie.gif', fps=2)
```

The resulting movie, in the animated GIF format, can be shown in a web page or displayed by the `animate` program.

Other movie formats can be produced by using other encoders, e.g., `ppmtompeg` and `ffmpeg` for the MPEG format, or `mencoder` for the AVI format. There are lots of options to the `movie` function, which you can see by writing `pydoc scitools.easyviz.movie` (see page 98 for how to run such a command).

4.7.2 Summarizing Example: Animating a Function

Problem. In this chapter's summarizing example we shall visualize how the temperature varies downward in the earth as the surface temperature oscillates between high day and low night values. One question may be: What is the temperature change 10 m down in the ground if

the surface temperature varies between 2 C in the night and 15 C in the day?

Let the z axis point downwards, towards the center of the earth, and let $z = 0$ correspond to the earth's surface. The temperature at some depth z in the ground at time t is denoted by $T(z, t)$. If the surface temperature has a periodic variation around some mean value T_0 , according to

$$T(0, t) = T_0 + A \cos(\omega t),$$

one can find, from a mathematical model for heat conduction, that the temperature at an arbitrary depth is

$$T(z, t) = T_0 + Ae^{-az} \cos(\omega t - az), \quad a = \sqrt{\frac{\omega}{2k}}. \quad (4.13)$$

The parameter k reflects the ground's ability to conduct heat (k is called the *heat conduction coefficient*).

The task is to make an animation of how the temperature profile in the ground, i.e., T as a function of z , varies in time. Let ω correspond to a time period of 24 hours. The mean temperature T_0 at the surface can be taken as 10 C, and the maximum variation A can be set to 10 C. The heat conduction coefficient k equals 1 mm²/s (which is 10⁻⁶ m²/s in proper SI units).

Solution. To animate $T(z, t)$ in time, we need to make a loop over points in time, and in each pass in the loop we must make a hardcopy of the plot of T as a function of z . The files with the hardcopies can then be combined to a movie. The algorithm becomes

```
for  $t_i = i\Delta t$ ,  $i = 0, 1, 2 \dots, n$ :
    plot the curve  $y(z) = T(z, t_i)$ 
    make hardcopy (store the plot in a file)
combine all the plot files into a movie
```

It can be wise to make a general `animate` function where we just feed in some $f(x, t)$ function and make all the plot files. If `animate` has arguments for setting the labels on the axis and the extent of the y axis, we can easily use `animate` also for a function $T(z, t)$ (we just use z as the name for the x axis and T as the name for the y axis in the plot). Recall that it is important to fix the extent of the y axis in a plot when we make animations, otherwise most plotting programs will automatically fit the extent of the axis to the current data, and the tickmarks on the y axis will jump up and down during the movie. The result is a wrong visual impression of the function.

The names of the plot files must have a common stem appended with some frame number, and the frame number should have a fixed number of digits, such as 0001, 0002, etc. (if not, the sequence of the plot files

will not be correct when we specify the collection of files with an asterisk for the frame numbers, e.g., as in `tmp*.png`). We therefore include an argument to `animate` for setting the name stem of the plot files. By default, the stem is `tmp_`, resulting in the filenames `tmp_0000.png`, `tmp_0001.png`, `tmp_0002.png`, and so forth. Other convenient arguments for the `animate` function are the initial time in the plot, the time lag Δt between the plot frames, and the coordinates along the x axis. The `animate` function then takes the form

```
def animate(tmax, dt, x, function, ymin, ymax, t0=0,
            xlabel='x', ylabel='y', hardcopy_stem='tmp_'):
    t = t0
    counter = 0
    while t <= tmax:
        y = function(x, t)
        plot(x, y,
             axis=[x[0], x[-1], ymin, ymax],
             title='time=%g' % t,
             xlabel=xlabel, ylabel=ylabel,
             hardcopy=hardcopy_stem + '%04d.png' % counter)
        t += dt
        counter += 1
```

The $T(z, t)$ function is easy to implement, but we need to decide whether the parameters A , ω , T_0 , and k shall be arguments to the Python implementation of $T(z, t)$ or if they shall be global variables. Since the `animate` function expects that the function to be plotted has only two arguments, we must implement $T(z, t)$ as `T(z, t)` in Python and let the other parameters be global variables (Chapters 7.1.1 and 7.1.2 explain this problem in more detail and present a better implementation). The `T(z, t)` implementation then reads

```
def T(z, t):
    # T0, A, k, and omega are global variables
    a = sqrt(omega/(2*k))
    return T0 + A*exp(-a*z)*cos(omega*t - a*z)
```

Suppose we plot $T(z, t)$ at n points for $z \in [0, D]$. We make such plots for $t \in [0, t_{\max}]$ with a time lag Δt between the them. The frames in the movie are now made by

```
# set T0, A, k, omega, D, n, tmax, dt
z = linspace(0, D, n)
animate(tmax, dt, z, T, T0-A, T0+A, 0, 'z', 'T')
```

We have here set the extent of the y axis in the plot as $[T_0 - A, T_0 + A]$, which is in accordance with the $T(z, t)$ function.

The call to `animate` above creates a set of files with names of the form `tmp_*.png`. The animation is then created by a call

```
movie('tmp_*.png', encoder='convert', fps=2,
      output_file='tmp_heatwave.gif')
```

It now remains to assign proper values to all the global variables in the program: n , D , T_0 , A , ω , dt , t_{\max} , and k . The oscillation

period is 24 hours, and ω is related to the period P of the cosine function by $\omega = 2\pi/P$ (realize that $\cos(t2\pi/P)$ has period P). We then express $P = 24 \text{ h} = 24 \cdot 60 \cdot 60 \text{ s}$ and compute $\omega = 2\pi/P$. The total simulation time can be 3 periods, i.e., $t_{\max} = 3P$. The $T(z, t)$ function decreases exponentially with the depth z so there is no point in having the maximum depth D larger than the depth where T is approximately zero, say 0.001. We have that $e^{-aD} = 0.001$ when $D = -a^{-1} \ln 0.001$, so we can use this estimate in the program. The proper initialization of all parameters can then be expressed as follows¹⁴:

```
k = 1E-6      # heat conduction coefficient (in m*m/s)
P = 24*60*60.# oscillation period of 24 h (in seconds)
omega = 2*pi/P
dt = P/24     # time lag: 1 h
tmax = 3*P    # 3 day/night simulation
T0 = 10       # mean surface temperature in Celsius
A = 10        # amplitude of the temperature variations in Celsius
a = sqrt(omega/(2*k))
D = -(1/a)*log(0.001) # max depth
n = 501       # no of points in the z direction
```

We encourage you to run the program `heatwave.py` to see the movie. The hardcopy of the movie is in the file `tmp_heatwave.gif`. Figure 4.13 displays two snapshots in time of the $T(z, t)$ function.

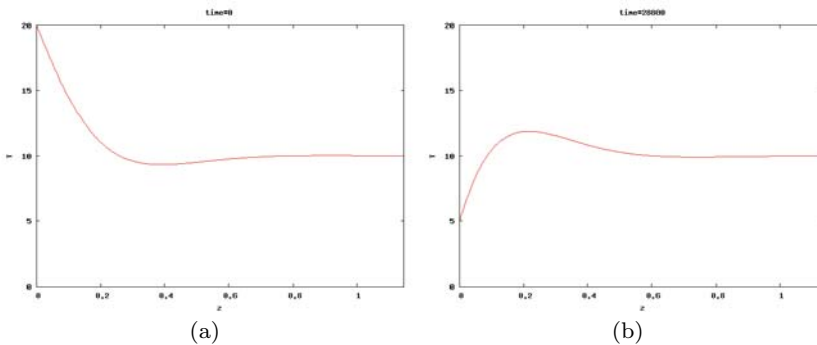


Fig. 4.13 Plot of the temperature $T(z, t)$ in the ground for two different t values.

Scaling. In this example, as in many other scientific problems, it was easier to write the code than to assign proper physical values to the input parameters in the program. To learn about the physical process, here how heat propagates from the surface and down in the ground, it is often advantageous to scale the variables in the problem so that we work with dimensionless variables. Through the scaling procedure we normally end up with much fewer physical parameters which must be assigned values. Let us show how we can take advantage of scaling the present problem.

¹⁴ Note that it is very important to use consistent units. Here we express all units in terms of meter, second, and Kelvin or Celsius.

Consider a variable x in a problem with some dimension. The idea of scaling is to introduce a new variable $\bar{x} = x/x_c$, where x_c is a *characteristic size* of x . Since x and x_c have the same dimension, the dimension cancels in \bar{x} such that \bar{x} is dimensionless. Choosing x_c to be the expected maximum value of x , ensures that $\bar{x} \leq 1$, which is usually considered a good idea. That is, we try to have all dimensionless variables varying between zero and one. For example, we can introduce a dimensionless z coordinate: $\bar{z} = z/D$, and now $\bar{z} \in [0, 1]$. Doing a proper scaling of a problem is challenging so for now it is sufficient to just follow the steps below - and not worry why we choose a certain scaling.

In the present problem we introduce these dimensionless variables:

$$\begin{aligned}\bar{z} &= z/D \\ \bar{T} &= \frac{T - T_0}{A} \\ \bar{t} &= \omega t\end{aligned}$$

We now insert $z = \bar{z}D$ and $t = \bar{t}/\omega$ in the expression for $T(z, t)$ and get

$$T = T_0 + Ae^{-b\bar{z}} \cos(\bar{t} - b\bar{z}), \quad b = aD$$

or

$$\bar{T}(\bar{z}, \bar{t}) = \frac{T - T_0}{A} = e^{-b\bar{z}} \cos(\bar{t} - b\bar{z}).$$

We see that \bar{T} depends on only *one* dimensionless parameter b in addition to the independent dimensionless variables \bar{z} and \bar{t} . It is common practice at this stage of the scaling to just drop the bars and write

$$T(z, t) = e^{-bz} \cos(t - bz). \quad (4.14)$$

This function is much simpler to plot than the one with lots of physical parameters, because now we know that T varies between -1 and 1 , t varies between 0 and 2π for one period, and z varies between 0 and 1 . The scaled temperature has only one “free” parameter b . That is, the shape of the graph is completely determined by b .

In our previous movie example, we used specific values for D , ω , and k , which then implies a certain $b = D\sqrt{\omega/(2k)}$ (≈ 6.9). However, we can now run different b values and see the effect on the heat propagation. Different b values will in our problems imply different periods of the surface temperature variation and/or different heat conduction values in the ground’s composition of rocks. Note that doubling ω and k leaves the same b – it is only the fraction ω/k that influences the value of b .

In a main program we can read b from the command line and make the movie:


```

b = float(sys.argv[1])
n = 401
z = linspace(0, 1, n)
animate(3*2*pi, 0.05*2*pi, z, T, -1.2, 1.2, 0, 'z', 'T')
movie('tmp_*.png', encoder='convert', fps=2,
      output_file='tmp_heatwave.gif')

```

Running the program, found as the file `heatwave_scaled.py`, for different b values shows that b governs how deep the temperature variations on the surface $z = 0$ penetrate. A large b makes the temperature changes confined to a thin layer close to the surface (see Figure 4.14 for $b = 20$), while a small b leads to temperature variations also deep down in the ground (see Figure 4.15 for $b = 2$).

We can understand the results from a physical perspective. Think of increasing ω , which means reducing the oscillation period so we get a more rapid temperature variation. To preserve the value of b we must increase k by the same factor. Since a large k means that heat quickly spreads down in the ground, and a small k implies the opposite, we see that more rapid variations at the surface requires a larger k to more quickly conduct the variations down in the ground. Similarly, slow temperature variations on the surface can penetrate deep in the ground even if the ground's ability to conduct (k) is low.

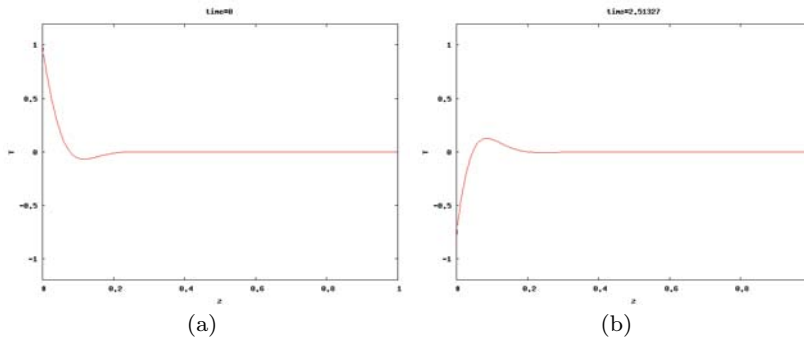


Fig. 4.14 Plot of the dimensionless temperature $T(z, t)$ in the ground for two different t values and $b = 20$.

4.8 Exercises

Exercise 4.1. *Fill lists with function values.*

A function with many applications in science is defined as

$$h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}. \quad (4.15)$$

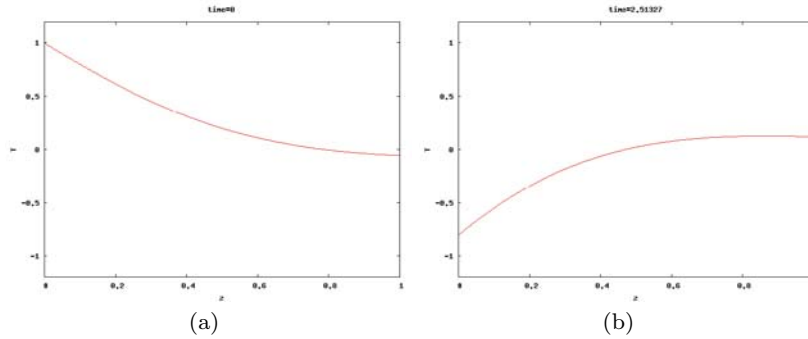


Fig. 4.15 Plot of the dimensionless temperature $T(z, t)$ in the ground for two different t values and $b = 2$.

Fill lists `xlist` and `hlist` with x and $h(x)$ values for uniformly spaced x coordinates in $[-4, 4]$. You may adapt the example in Chapter 4.2.1. Name of program file: `fill_lists.py`. ◇

Exercise 4.2. *Fill arrays; loop version.*

The aim is to fill two arrays `x` and `h` with x and $h(x)$ values, where $h(x)$ is defined in (4.15). Let the x values be uniformly spaced in $[-4, 4]$. Create two arrays of zeros and fill both arrays with values inside a loop. Name of program file: `fill_arrays_loop.py`. ◇

Exercise 4.3. *Fill arrays; vectorized version.*

Vectorize the code in Exercise 4.2 by creating the x values using the `linspace` function and by evaluating $h(x)$ for an array argument. Name of program file: `fill_arrays_vectorized.py`. ◇

Exercise 4.4. *Apply a function to a vector.*

Given a vector $v = (2, 3, -1)$ and a function $f(x) = x^3 + xe^x + 1$, apply f to each element in v . Then calculate $f(v)$ as $v^3 + v * e^v + 1$ using the vector computing rules. Show that the two results are equal. ◇

Exercise 4.5. *Simulate by hand a vectorized expression.*

Suppose `x` and `t` are two arrays of the same length, entering a vectorized expression

```
y = cos(sin(x)) + exp(1/t)
```

If `x` holds two elements, 0 and 2, and `t` holds the elements 1 and 1.5, calculate by hand (using a calculator) the `y` array. Thereafter, write a program that mimics the series of computations you did by hand (typically a sequence of operations of the kind we listed on page 174 – use explicit loops, but at the end you can use Numerical Python functionality to check the results). Name of program file: `simulate_vector_computing.py`. ◇

Exercise 4.6. *Demonstrate array slicing.*

Create an array `w` with values $0, 0.1, 0.2, \dots, 2$ using `linspace`. Write out `w[:]`, `w[:-2]`, `w[:5]`, `w[2:-2:6]`. Convince yourself in each case that you understand which elements of the array that are printed. Name of program file: `slicing.py`. \diamond

Exercise 4.7. *Plot the formula (1.1).*

Make a plot of the function $y(t)$ in (1.1) on page 1 for $v_0 = 10$ and $t \in [0, 2v_0/g]$. The label on the x axis should be 'time'.

If you use Easyviz with the Gnuplot backend on Windows machines, you need a `raw_input()` call at the end of the program such that the program halts until Gnuplot is finished with the plot (simply press Return to finish the program). See also Appendix E.1.3.

Name of program file: `plot_ball11.py`. \diamond

Exercise 4.8. *Plot the formula (1.1) for several v_0 values.*

Make a program that reads a set of v_0 values from the command line and plots the curve (1.1) for the different v_0 values (in the same figure). Let the t coordinates go from 0 to $2v_0/g$ for each curve, which implies that you need a different vector of t coordinates for each curve. Name of program file: `plot_ball12.py`. \diamond

Exercise 4.9. *Plot exact and inexact Fahrenheit–Celsius formulas.*

Exercise 2.20 introduces a simple rule to quickly compute the Celsius temperature from the Fahrenheit degrees: $C = (F - 30)/2$. Compare this curve against the exact curve $C = (F - 32)5/9$ in a plot. Let F vary between -20 and 120 . Name of program file: `f2c_shortcut_plot.py`. \diamond

Exercise 4.10. *Plot the trajectory of a ball.*

The formula for the trajectory of a ball is given in (1.5) on page 38. In a program, first read the input data y_0 , θ , and v_0 from the command line. Then compute where the ball hits the ground, i.e., the value x_g for which $f(x_g) = 0$. Plot the trajectory $y = f(x)$ for $x \in [0, x_g]$, using the same scale on the x and y axes such that we get a visually correct view of the trajectory. Name of program file: `plot_trajectory.py`. \diamond

Exercise 4.11. *Plot a wave packet.*

The function

$$f(x, t) = e^{-(x-3t)^2} \sin(3\pi(x-t)) \quad (4.16)$$

describes for a fixed value of t a wave localized in space. Make a program that visualizes this function as a function of x on the interval $[-4, 4]$ when $t = 0$. Name of program file: `plot_wavepacket.py`. \diamond

Exercise 4.12. *Use pyreport in Exer. 4.11.*

Use `pyreport` (see Chapter 1.8) in Exercise 4.11 to generate a nice report in HTML and PDF format. To get the plot inserted in the

report, you must call `show()` after the `plot` instruction. You also need to use the version of `pyreport` that comes with SciTools (that version is automatically installed when you install SciTools). Name of program file: `pyreport_wavepacket.py`. ◇

Exercise 4.13. *Judge a plot.*

Assume you have the following program for plotting a parabola:

```
x = linspace(0, 2, 20)
y = x*(2 - x)
plot(x, y)
```

Then you switch to the function $\cos(18\pi x)$ by altering the computation of `y` to `y = cos(18*pi*x)`. Judge the resulting plot. Is it correct? Display the $\cos(18\pi x)$ function with 1000 points in the same plot. Name of program file: `judge_plot.py`. ◇

Exercise 4.14. *Plot the viscosity of water.*

The viscosity of water, μ , varies with the temperature T (in Kelvin) according to

$$\mu(T) = A \cdot 10^{B/(T-C)}, \quad (4.17)$$

where $A = 2.414 \cdot 10^{-5}$ Ns/m², $B = 247.8$ K, and $C = 140$ K. Plot $\mu(T)$ for T between 0 and 100 degrees Celsius. ◇

Exercise 4.15. *Explore a function graphically.*

The wave speed of water surface waves, c , depends on the length of the wave, λ . The following formula relates c to λ :

$$c(\lambda) = \sqrt{\frac{g\lambda}{2\pi} \left(1 + s \frac{4\pi^2}{\rho g \lambda^2} \right) \tanh \left(\frac{2\pi h}{\lambda} \right)}, \quad (4.18)$$

where g is the acceleration of gravity, s is the surface tension between water and air ($7.9 \cdot 10^{-4}$ N/cm), ρ is the density of water (can be taken as 1 kg/cm³), and h is the water depth. Let us fix h at 50 m. First make a plot of $c(\lambda)$ for small λ (1 mm to 10 cm). Then make a plot $c(\lambda)$ for larger λ (1 m to 2 km). Name of program file: `water_wave_velocity.py`. ◇

Exercise 4.16. *Plot Taylor polynomial approximations to $\sin x$.*

The sine function can be approximated by a polynomial according to the following formula:

$$\sin x \approx S(x; n) = \sum_{j=0}^n (-1)^j \frac{x^{2j+1}}{(2j+1)!}. \quad (4.19)$$

The expression $(2j+1)!$ is the factorial (see Exercise 2.33). The error in the approximation $S(x; n)$ decreases as n increases and in the limit

we have that $\sin x = \lim_{n \rightarrow \infty} S(x; n)$. The purpose of this exercise is to visualize the quality of various approximations $S(x; n)$ as n increases.

The first part of the exercise is to write a Python function `S(x, n)` that computes $S(x; n)$. Use a straightforward approach where you compute each term as it stands in the formula, i.e., $(-1)^j x^{2j+1}$ divided by the factorial $(2j+1)!$. (We remark that Exercise 5.16 outlines a much more efficient computation of the terms in the series.)

The next part of the exercise is to plot $\sin x$ on $[0, 4\pi]$ together with the approximations $S(x; 1)$, $S(x; 2)$, $S(x; 3)$, $S(x; 6)$, and $S(x; 12)$. Name of program file: `plot_Taylor_sin.py`. \diamond

Exercise 4.17. *Animate a wave packet.*

Display an animation of the function $f(x, t)$ in Exercise 4.11 by plotting f as a function of x on $[-6, 6]$ for a set of t values in $[-1, 1]$. Also make an animated GIF file. A suitable resolution can be 1000 intervals (1001 points) along the x axis, 60 intervals (61 points) in time, and 6 frames per second in the animated GIF file. Use the recipe in Chapter 4.3.7 and remember to remove the family of old plot files in the beginning of the program.

You will see that $f(x, t)$ models waves that are moving to the right (when x is a space coordinate and t is time). The velocities of the individual waves and the packet are different, demonstrating an important case in physics when the phase velocity of waves is different from the group velocity. This effect is visible for water surface waves, particularly those generated by a boat. Name of program file: `plot_wavepacket_movie.py`. \diamond

Exercise 4.18. *Animate the evolution of Taylor polynomials.*

A general series approximation (to a function) can be written as

$$S(x; M, N) = \sum_{k=M}^N f_k(x).$$

For example, the Taylor polynomial for e^x equals $S(x)$ with $f_k(x) = x^k/k!$. The purpose of the exercise is to make a movie of how $S(x)$ develops (and hopefully improves as an approximation) as we add terms in the sum. That is, the frames in the movie correspond to plots of $S(x; M, M+1)$, $S(x; M, M+2)$, \dots , $S(x; M, N)$.

Make a function

```
animate_series(fk, M, N, xmin, xmax, ymin, ymax, n, exact)
```

for creating such animations. The argument `fk` holds a Python function implementing the term $f_k(x)$ in the sum, `M` and `N` are the summation limits, the next arguments are the minimum and maximum x and y values in the plot, `n` is the number of x points in the curves to be plotted, and `exact` holds the function that $S(x)$ aims at approximating.

The `animate_series` function must accumulate the $f_k(x)$ in a variable s , and for each k value, s is plotted against x together with a curve reflecting the exact function. Each plot must be saved in a file, say with names `tmp_0000.png`, `tmp_0001.png`, and so on (these filenames can be generated by `tmp_%04d.png`, using an appropriate counter). Use the `movie` function to combine all the plot files into a movie in a desired movie format.

In the beginning of the `animate_series` it is necessary to remove all old plot files of the form `tmp_*.png`. This can be done by the `glob` module and the `os.remove` function as exemplified in Chapter 4.3.7 and in Appendix E.4 (page 677).

Test the `animate_series` function in the two cases:

1. The Taylor series for $\sin x$, where $f_k(x) = (-1)^k x^{2k+1}/(2k+1)!$, and $x \in [0, 13\pi]$, $M = 0$, $N = 40$, $y \in [-2, 2]$.
2. The Taylor series for e^{-x} , where $f_k(x) = (-x)^k/k!$, and $x \in [0, 15]$, $M = 0$, $N = 30$, $y \in [-0.5, 1.4]$.

Name of program file: `animate_Taylor_series.py`. ◇

Exercise 4.19. *Plot the velocity profile for pipeflow.*

A fluid that flows through a (very long) pipe has zero velocity on the pipe wall and a maximum velocity along the centerline of the pipe. The velocity v varies through the pipe cross section according to the following formula:

$$v(r) = \left(\frac{\beta}{2\mu_0} \right)^{1/n} \frac{n}{n+1} \left(R^{1+1/n} - r^{1+1/n} \right), \quad (4.20)$$

where R is the radius of the pipe, β is the pressure gradient (the force that drives the flow through the pipe), μ_0 is a viscosity coefficient (small for air, larger for water and even larger for toothpaste), n is a real (!) number reflecting the viscous properties of the fluid ($n = 1$ for water and air, $n < 1$ for many modern plastic materials), and r is a radial coordinate that measures the distance from the centerline ($r = 0$ is the centerline, $r = R$ is the pipe wall).

Make a function that evaluates $v(r)$. Plot $v(r)$ as a function of $r \in [0, R]$, with $R = 1$, $\beta = 0.02$, $\mu = 0.02$, and $n = 0.1$. Thereafter, make an animation of how the $v(r)$ curves varies as n goes from 1 and down to 0.01. Because the maximum value of $v(r)$ decreases rapidly as n decreases, each curve can be normalized by its $v(0)$ value such that the maximum value is always unity. Name of program file: `plot_velocity_pipeflow.py`. ◇

Exercise 4.20. *Plot the approximate function from Exer. 1.13.*

First make a Python function `S(t, n)` that evaluates $S(t; n)$ defined in Exercise 1.13. Plot $S(t; 1)$, $S(t; 3)$, $S(t; 20)$, $S(t; 200)$, and the exact $f(t)$ function in the same plot.

The resulting plot shows how a step-like function can be approximated by a sum of sine functions of increasing frequency. Representation of functions as a sum of sines and cosines is an important mathematical concept, known as Fourier series, and has a wide range of applications. Name of program file: `plot_compare_func_sum.py`. ◇

Exercise 4.21. *Plot functions from the command line.*

For quickly get a plot a function $f(x)$ for $x \in [x_{\min}, x_{\max}]$ it could be nice to have a program that takes the minimum amount of information from the command line and produces a plot on the screen and a hardcopy `tmp.eps`. The usage of the program goes as follows:

```
plotf.py "f(x)" xmin xmax
```

A specific example is

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

Hint: Make x coordinates from the 2nd and 3rd command-line arguments and then use `eval` (or `StringFunction` from Chapters 3.1.4 and 4.4.3) on the first 1st argument. Try to write as short program as possible (we leave it to Exercise 4.22 to test for valid input). Name of program file: `plotf_v1.py`. ◇

Exercise 4.22. *Improve the program from Exercise 4.21.*

Equip the program from Exercise 4.21 with tests on valid input on the command line. Also allow an optional 4th command-line argument for the number of points along the function curve. Set this number to 501 if it is not given. Name of program file: `plotf.py`. ◇

Exercise 4.23. *Demonstrate energy concepts from physics.*

The vertical position $y(t)$ of a ball thrown upward is described by (1.1) on page 1. Two important physical quantities in this context are the potential energy, obtained by from doing work against gravity, and the kinetic energy, arising from motion. The potential energy is defined as $P = mgy$, where m is the mass of the ball. The kinetic energy is defined as $K = \frac{1}{2}mv^2$, where v is the velocity of the ball, related to y by $v(t) = y'(t)$. Plot $P(t)$ and $K(t)$ in the same plot, along with their sum $P + K$. Let $t \in \frac{2v_0}{g}$. Read m and v_0 from the command line. Run the program with various choices of m and v_0 and observe that $P + K$ is always constant in this motion. In fact, it turns out that $P + K$ is constant for a large class of motions, and this is a very important result in physics. Name of program file: `energy_physics.py`. ◇

Exercise 4.24. *Plot a w-like function.*

Define mathematically a function that looks like the 'w' character. Plot this function. Name of program file: `plot_w.py`. ◇

Exercise 4.25. *Plot a smoothed “hat” function.*

The “hat” function $N(x)$ defined by (2.5) on page 89 has a discontinuity in the derivative at $x = 1$. Suppose we want to “round” this function such that it looks smooth around $x = 1$. To this end, replace the straight lines in the vicinity of $x = 1$ by a (small) cubic curve

$$y = a(x - 1)^3 + b(x - 1)^2 + c(x - 1) + d,$$

for $x \in [1 - \epsilon, 1 + \epsilon]$, where a , b , c , and d are parameters that must be adjusted in order for the cubic curve to match the value and the derivative of the function $N(x)$. The new rounded functions has the specification

$$\tilde{N}(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 - \epsilon \\ a_1(x - 1)^3 + b(x - 1) + c(x - 1) + d_1, & 1 - \epsilon \leq x < 1, \\ a_2(x - 1)^3 + b(x - 1) + c(x - 1) + d_2, & 1 \leq x < 1 + \epsilon, \\ 2 - x, & 1 + \epsilon \leq x < 2 \\ 0, & x \geq 2 \end{cases} \quad (4.21)$$

with $a_1 = \frac{1}{3}\epsilon^{-2}$, $a_2 = -a_1$, $d_1 = 1 - \epsilon + a_1\epsilon^3$, $d_2 = 1 - \epsilon - a_2\epsilon^3$, and $b = c = 0$. Plot this function. (Hint: Be careful with the choice of x coordinates!) Name of program file: `plot_hat.py`. \diamond

Exercise 4.26. *Experience overflow in a function.*

When object (ball, car, airplane) moves through the air, there is a very, very thin layer of air close to the object’s surface where the air velocity varies dramatically¹⁵, from the same value as the velocity of the object at the object’s surface to zero a few centimeters away. The change in velocity is quite abrupt and can modeled by the function

$$v(x) = \frac{1 - e^{x/\mu}}{1 - e^{1/\mu}},$$

where $x = 1$ is the object’s surface and $x = 0$ is some distance “far” away where one cannot notice any wind velocity v because of the passing object ($v = 0$). The wind velocity coincides with the velocity of the object at $x = 1$, here set to $v = 1$. The parameter μ reflects the viscous behavior of air, which is very small, so typically $\mu = 10^{-6}$. With this relevant physical value of μ , it quickly becomes difficult to calculate $v(x)$ on a computer.

Make a function `v(x, mu=1E-6, exp=math.exp)` for calculating the formula of v using `exp` as a possibly user-given exponential function. Let the `v` function return the nominator and denominator in the formula as well as the fraction (result). Call the `v` function for various x

¹⁵ This layer is called a *boundary layer*. The physics in the boundary layer is very important for air resistance and cooling/heating of objects.

values between 0 and 1 in a `for` loop, let `mu` be `1E-3`, and have an inner `for` loop over two different `exp` functions: `math.exp` and `numpy.exp`. The output will demonstrate how the denominator is subject to overflow and how difficult it is to calculate this function on a computer.

Also plot $v(x)$ for $\mu = 1, 0.01, 0.001$ on $[0, 1]$ using 10,000 points to see what the function looks like. Name of program file: `boundary_layer_func1.py`. \diamond

Exercise 4.27. *Experience less overflow in a function.*

In the program from Exercise 4.26, convert `x` and `eps` to a higher precision representation of real numbers, with the aid of the NumPy type `float96`:

```
import numpy
x = numpy.float96(x); mu = numpy.float96(e)
```

Call the `v` function with these type of variables observe how much “better” results we get with `float96` compared the standard `float` value (which is `float64` – the number reflects the number of bits in the machine’s representation of a real number). Also call the `v` function with `x` and `mu` as `float32` variables and report how the function now behaves. Name of program file: `boundary_layer_func2.py`. \diamond

Exercise 4.28. *Extend Exer. 4.4 to a rank 2 array.*

Let A be a two-dimensional array with two indices:

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

Apply the function f from Exercise 4.4 to each element in A . Thereafter, calculate the result of the array expression $A * 3 + A * e^A + 1$ and demonstrate that the end result of the two methods are the same. \diamond

Exercise 4.29. *Explain why array computations fail.*

The following loop computes the array `y` from `x`:

```
>>> x = linspace(0, 1, 3)
>>> y = zeros(len(x))
>>> for i in range(len(x)):
...     y[i] = x[i] + 4
```

However, the alternative loop

```
>>> for xi, yi in zip(x, y):
...     yi = xi + 5
```

leaves `y` unchanged. Why? Explain in detail what happens in each pass of this loop and write down the contents of `xi`, `yi`, `x`, and `y` as the loop progresses. \diamond