

This appendix is authored by Aslak Tveito

In this chapter we will discuss how to differentiate and integrate functions on a computer. To do that, we have to care about how to treat mathematical functions on a computer. Handling mathematical functions on computers is not entirely straightforward: A function $f(x)$ contains an infinite amount of information (function values at an infinite number of x values on an interval), while the computer can only store a finite¹ amount of data. Think about the $\cos x$ function. There are typically two ways we can work with this function on a computer. One way is to run an algorithm, like that in Exercise 2.38 on page 108, or we simply call `math.cos(x)` (which runs a similar type of algorithm), to compute an approximation to $\cos x$ for a given x , using a finite number of calculations. The other way is to store $\cos x$ values in a table for a finite number of x values² and use the table in a smart way to compute $\cos x$ values. This latter way, known as a *discrete* representation of a function, is in focus in the present chapter. With a discrete function representation, we can easily integrate and differentiate the function too. Read on to see how we can do that.

The folder `src/discalc` contains all the program example files referred to in this chapter.

A.1 Discrete Functions

Physical quantities, such as temperature, density, and velocity, are usually defined as continuous functions of space and time. However, as

¹ Allow yourself a moment or two to think about the terms “finite” and “infinite”; infinity is not an easy term, but it is not infinitely difficult. Or is it?

² Of course, we need to run an algorithm to populate the table with $\cos x$ numbers.

mentioned in above, discrete versions of the functions are more convenient on computers. We will illustrate the concept of discrete functions through some introductory examples. In fact, we used discrete functions in Chapter 4 to plot curves: We defined a finite set of coordinates \mathbf{x} and stored the corresponding function values $\mathbf{f}(\mathbf{x})$ in an array. A plotting program would then draw straight lines between the function values. A discrete representation of a continuous function is, from a programming point of view, nothing but storing a finite set of coordinates and function values in an array. Nevertheless, we will in this chapter be more formal and describe discrete functions by precise mathematical terms.

A.1.1 The Sine Function

Suppose we want to generate a plot of the sine function for values of x between 0 and π . To this end, we define a set of x -values and an associated set of values of the sine function. More precisely, we define $n + 1$ points by

$$x_i = ih \text{ for } i = 0, 1, \dots, n \quad (\text{A.1})$$

where $h = \pi/n$ and $n \geq 1$ is an integer. The associated function values are defined as

$$s_i = \sin(x_i) \text{ for } i = 0, 1, \dots, n. \quad (\text{A.2})$$

Mathematically, we have a sequence of coordinates $(x_i)_{i=0}^n$ and of function values $(s_i)_{i=0}^n$ (see the start of Chapter 5 for an explanation of the notation and the sequence concept). Often we “merge” the two sequences to one sequence of points: $(x_i, s_i)_{i=0}^n$. Sometimes we also use a shorter notation, just x_i , s_i , or (x_i, s_i) if the exact limits are not of importance. The set of coordinates $(x_i)_{i=0}^n$ constitutes a *mesh* or a *grid*. The individual coordinates x_i are known as *nodes* in the mesh (or grid). The discrete representation of the sine function on $[0, \pi]$ consists of the mesh and the corresponding sequence of function values $(s_i)_{i=0}^n$ at the nodes. The parameter n is often referred to as the *mesh resolution*.

In a program, we represent the mesh by a coordinate array, say \mathbf{x} , and the function values by another array, say \mathbf{s} . To plot the sine function we can simply write

```
from scitools.std import *

n = int(sys.argv[1])

x = linspace(0, pi, n+1)
s = sin(x)
plot(x, s, legend='sin(x), n=%d' % n, hardcopy='tmp.eps')
```

Figure A.1 shows the resulting plot for $n = 5, 10, 20$ and 100. As pointed out in Chapter 4, the curve looks smoother the more points

we use, and since $\sin(x)$ is a smooth function, the plots in Figures A.1a and A.1b do not look sufficiently good. However, we can with our eyes hardly distinguish the plot with 100 points from the one with 20 points, so 20 points seem sufficient in this example.

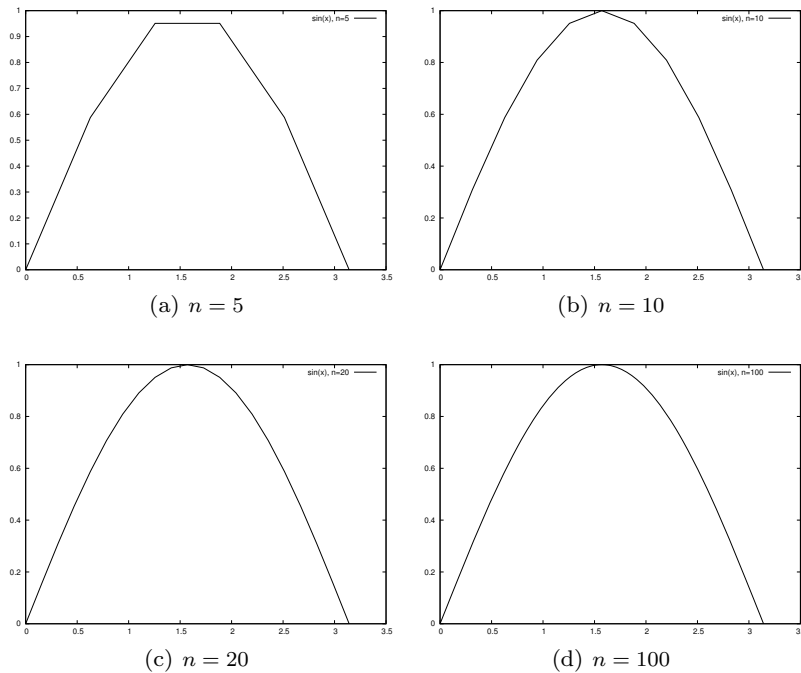


Fig. A.1 Plots of $\sin(x)$ with various n .

There are no tests on the validity of the input data (n) in the previous program. A program including these tests reads³:

```
#!/usr/bin/env python
from scitools.std import *

try:
    n = int(sys.argv[1])
except:
    print "usage: %s n" %sys.argv[0]
    sys.exit(1)

x = linspace(0, pi, n+1)
s = sin(x)
plot(x, s, legend='sin(x), n=%d' % n, hardcopy='tmp.eps')
```

Such tests are important parts of a good programming philosophy. However, for the programs displayed in this and the next chapter, we skip such tests in order to make the programs more compact and readable as part of the rest of the text and to enable focus on the mathematics in the programs. In the versions of these programs in the

³ For an explanation of the first line of this program, see Appendix E.1

files that can be downloaded you will, hopefully, always find a test on input data.

A.1.2 Interpolation

Suppose we have a discrete representation of the sine function: $(x_i, s_i)_{i=0}^n$. At the nodes we have the exact sine values s_i , but what about the points in between these nodes? Finding function values between the nodes is called *interpolation*, or we can say that we *interpolate* a discrete function.

A graphical interpolation procedure could be to look at one of the plots in Figure A.1 to find the function value corresponding to a point x between the nodes. Since the plot is a straight line from node value to node value, this means that a function value between two nodes is found from a straight line approximation⁴ to the underlying continuous function. We formulate this procedure precisely in terms of mathematics in the next paragraph.

Assume that we know that a given x^* lies in the interval from $x = x_k$ to x_{k+1} , where the integer k is given. In the interval $x_k \leq x < x_{k+1}$, we define the linear function that passes through (x_k, s_k) and (x_{k+1}, s_{k+1}) :

$$S_k(x) = s_k + \frac{s_{k+1} - s_k}{x_{k+1} - x_k}(x - x_k). \quad (\text{A.3})$$

That is, $S_k(x)$ coincides with $\sin(x)$ at x_k and x_{k+1} , and between these nodes, $S_k(x)$ is linear. We say that $S_k(x)$ interpolates the discrete function $(x_i, s_i)_{i=0}^n$ on the interval $[x_k, x_{k+1}]$.

A.1.3 Evaluating the Approximation

Given the values $(x_i, s_i)_{i=0}^n$ and the formula (A.3), we want to compute an approximation of the sine function for any x in the interval from $x = 0$ to $x = \pi$. In order to do that, we have to compute k for a given value of x . More precisely, for a given x we have to find k such that $x_k \leq x < x_{k+1}$. We can do that by defining

$$k = \lfloor x/h \rfloor$$

where the function $\lfloor z \rfloor$ denotes the largest integer that is smaller than z . In Python, $\lfloor z \rfloor$ is computed by `int(z)`. The program below takes x and n as input and computes the approximation of $\sin(x)$. The program

⁴ Strictly speaking, we also assume that the function to be interpolated is rather smooth. It is easy to see that if the function is very wild, i.e., the values of the function changes very rapidly, this procedure may fail even for very large values of n . Chapter 4.4.2 provides an example.

prints the approximation $S(x)$ and the exact⁵ value of $\sin(x)$ so we can look at the development of the error when n is increased.

```
from numpy import *
import sys

xp = eval(sys.argv[1])
n = int(sys.argv[2])

def S_k(k):
    return s[k] + \
        ((s[k+1] - s[k])/(x[k+1] - x[k]))*(xp - x[k])

h = pi/n
x = linspace(0, pi, n+1)
s = sin(x)
k = int(xp/h)

print 'Approximation of sin(%s):' % xp, S_k(k)
print 'Exact value of sin(%s):' % xp, sin(xp)
print 'Error in approximation:' % xp, sin(xp) - S_k(k)
```

To study the approximation, we put $x = \sqrt{2}$ and use the program `eval_sine.py` for $n = 5, 10$ and 20 .

Terminal

```
eval_sine.py 'sqrt(2)' 5
Approximation of sin(1.41421356237): 0.951056516295
Exact value of sin(1.41421356237): 0.987765945993
Error in approximation: 0.0367094296976
```

Terminal

```
eval_sine.py 'sqrt(2)' 10
Approximation of sin(1.41421356237): 0.975605666221
Exact value of sin(1.41421356237): 0.987765945993
Error in approximation: 0.0121602797718
```

Terminal

```
eval_sine.py 'sqrt(2)' 20
Approximation of sin(1.41421356237): 0.987727284363
Exact value of sin(1.41421356237): 0.987765945993
Error in approximation: 3.86616296923e-05
```

Note that the error is reduced as the n increases.

A.1.4 Generalization

In general, we can create a discrete version of a continuous function as follows. Suppose a continuous function $f(x)$ is defined on an interval

⁵ The value is not really exact – it is the value of $\sin(x)$ provided by the computer, `math.sin(x)`, and this value is calculated from an algorithm that only yields an approximation to $\sin(x)$. Exercise 2.38 provides an example of the type of algorithm in question.

ranging from $x = a$ to $x = b$, and let $n \geq 1$, be a given integer. Define the distance between nodes,

$$h = \frac{b - a}{n},$$

and the nodes

$$x_i = a + ih \text{ for } i = 0, 1, \dots, n. \quad (\text{A.4})$$

The discrete function values are given by

$$y_i = f(x_i) \text{ for } i = 0, 1, \dots, n. \quad (\text{A.5})$$

Now, $(x_i, y_i)_{i=0}^n$ is the discrete version of the continuous function $f(x)$. The program `discrete_func.py` takes f, a, b and n as input, computes the discrete version of f , and then applies the discrete version to make a plot of f .

```
def discrete_func(f, a, b, n):
    x = linspace(a, b, n+1)
    y = zeros(len(x))
    for i in xrange(len(x)):
        y[i] = func(x[i])
    return x, y

from scitools.std import *

f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
n = int(sys.argv[4])
f = StringFunction(f_formula)

x, y = discrete_func(f, a, b, n)
plot(x, y)
```

We can equally well make a vectorized version of the `discrete_func` function:

```
def discrete_func(f, a, b, n):
    x = linspace(a, b, n+1)
    y = f(x)
    return x, y
```

However, for the `StringFunction` tool to work properly in vectorized mode, we need to follow the recipe in Chapter 4.4.3:

```
f = StringFunction(f_formula)
f.vectorize(globals())
```

The corresponding vectorized program is found in the file `discrete_func_vec.py`.

A.2 Differentiation Becomes Finite Differences

You have heard about derivatives. Probably, the following formulas are well known to you:

$$\begin{aligned}\frac{d}{dx} \sin(x) &= \cos(x), \\ \frac{d}{dx} \ln(x) &= \frac{1}{x}, \\ \frac{d}{dx} x^m &= mx^{m-1},\end{aligned}$$

But why is differentiation so important? The reason is quite simple: The derivative is a mathematical expression of change. And change is, of course, essential in modeling various phenomena. If we know the state of a system, and we know the laws of change, then we can, in principle, compute the future of that system. Appendix B treats this topic in detail. Chapter 5 also computes the future of systems, based on modeling changes, but without using differentiation. In Appendix B you will see that reducing the step size in the difference equations in Chapter 5 results in derivatives instead of pure differences. However, differentiation of continuous functions is somewhat hard on a computer, so we often end up replacing the derivatives by differences. This idea is quite general, and every time we use a discrete representation of a function, differentiation becomes differences, or *finite differences* as we usually say.

The mathematical definition of differentiation reads

$$f'(x) = \lim_{\varepsilon \rightarrow 0} \frac{f(x + \varepsilon) - f(x)}{\varepsilon}.$$

You have probably seen this definition many times, but have you understood what it means and do you think the formula has a great practical value? Although the definition requires that we pass to the limit, we obtain quite good approximations of the derivative by using a fixed positive value of ε . More precisely, for a small $\varepsilon > 0$, we have

$$f'(x) \approx \frac{f(x + \varepsilon) - f(x)}{\varepsilon}.$$

The fraction on the right-hand side is a finite difference approximation to the derivative of f at the point x . Instead of using ε it is more common to introduce $h = \varepsilon$ in finite differences, i.e., we like to write

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}. \quad (\text{A.6})$$

A.2.1 Differentiating the Sine Function

In order to get a feeling for how good the approximation (A.6) to the derivative really is, we explore an example. Consider $f(x) = \sin(x)$ and the associated derivative $f'(x) = \cos(x)$. If we put $x = 1$, we have

$$f'(1) = \cos(1) \approx 0.540,$$

and by putting $h = 1/100$ in (A.6) we get

$$f'(1) \approx \frac{f(1 + 1/100) - f(1)}{1/100} = \frac{\sin(1.01) - \sin(1)}{0.01} \approx 0.536.$$

The program `forward_diff.py`, shown below, computes the derivative of $f(x)$ using the approximation (A.6), where x and h are input parameters.

```
def diff(f, x, h):
    return (f(x+h) - f(x))/float(h)

from math import *
import sys

x = eval(sys.argv[1])
h = eval(sys.argv[2])

approx_deriv = diff(sin, x, h)
exact = cos(x)
print 'The approximated value is: ', approx_deriv
print 'The correct value is: ', exact
print 'The error is: ', exact - approx_deriv
```

Running the program for $x = 1$ and $h = 1/1000$ gives

Terminal	
forward_diff.py 1 0.001	
The approximated value is:	0.53988148036
The correct value is:	0.540302305868
The error is:	0.000420825507813

A.2.2 Differences on a Mesh

Frequently, we will need finite difference approximations to a discrete function defined on a mesh. Suppose we have a discrete representation of the sine function: $(x_i, s_i)_{i=0}^n$, as introduced in Chapter A.1.1. We want to use (A.6) to compute approximations to the derivative of the sine function at the nodes in the mesh. Since we only have function values at the nodes, the h in (A.6) must be the difference between nodes, i.e., $h = x_{i+1} - x_i$. At node x_i we then have the following approximation of the derivative:

$$z_i = \frac{s_{i+1} - s_i}{h}, \quad (\text{A.7})$$

for $i = 0, 1, \dots, n-1$. Note that we have not defined an approximate derivative at the end point $x = x_n$. We cannot apply (A.7) directly since s_{n+1} is undefined (outside the mesh). However, the derivative of a function can also be defined as

$$f'(x) = \lim_{\varepsilon \rightarrow 0} \frac{f(x) - f(x - \varepsilon)}{\varepsilon},$$

which motivates the following approximation for a given $h > 0$,

$$f'(x) \approx \frac{f(x) - f(x - h)}{h}. \quad (\text{A.8})$$

This alternative approximation to the derivative is referred to as a *backward difference* formula, whereas the expression (A.6) is known as a *forward difference* formula. The names are natural: The forward formula goes forward, i.e., in the direction of increasing x and i to collect information about the change of the function, while the backward formula goes backwards, i.e., toward smaller x and i value to fetch function information.

At the end point we can apply the backward formula and thus define

$$z_n = \frac{s_n - s_{n-1}}{h}. \quad (\text{A.9})$$

We now have an approximation to the derivative at all the nodes. A plain specialized program for computing the derivative of the sine function on a mesh and comparing this discrete derivative with the exact derivative is displayed below (the name of the file is `diff_sine_plot1.py`).

```
from scitools.std import *

n = int(sys.argv[1])

h = pi/n
x = linspace(0, pi, n+1)
s = sin(x)
z = zeros(len(s))
for i in xrange(len(z)-1):
    z[i] = (s[i+1] - s[i])/h
# special formula for end point_
z[-1] = (s[-1] - s[-2])/h
plot(x, z)

xfine = linspace(0, pi, 1001) # for more accurate plot
exact = cos(xfine)
hold()
plot(xfine, exact)
legend('Approximate function', 'Correct function')
title('Approximate and discrete functions, n=%d' % n)
```

In Figure A.2 we see the resulting graphs for $n = 5, 10, 20$ and 100 . Again, we note that the error is reduced as n increases.

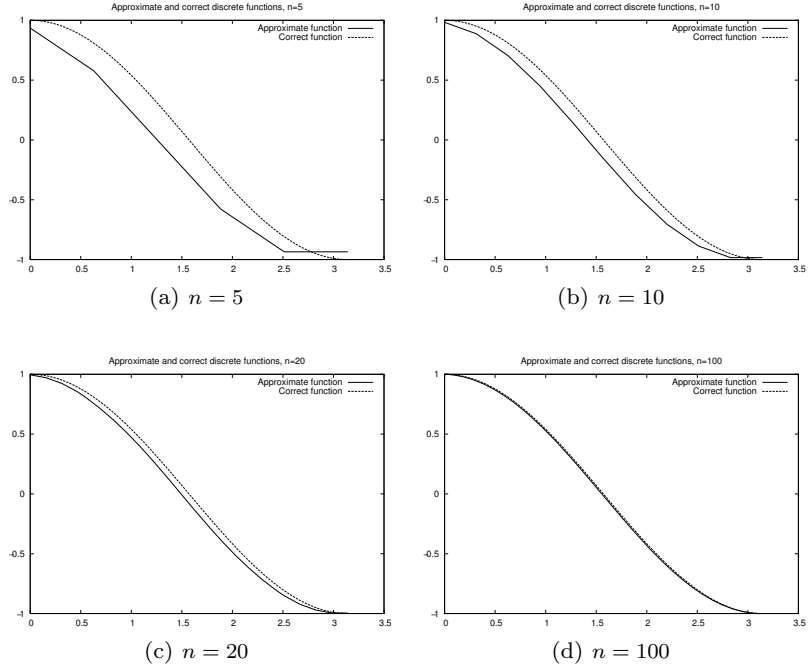


Fig. A.2 Plots for exact and approximate derivatives of $\sin(x)$ with varying values of the resolution n .

A.2.3 Generalization

The discrete version of a continuous function $f(x)$ defined on an interval $[a, b]$ is given by $(x_i, y_i)_{i=0}^n$ where

$$x_i = a + ih,$$

and

$$y_i = f(x_i)$$

for $i = 0, 1, \dots, n$. Here, $n \geq 1$ is a given integer, and the spacing between the nodes is given by

$$h = \frac{b - a}{n}.$$

A discrete approximation of the derivative of f is given by $(x_i, z_i)_{i=0}^n$ where

$$z_i = \frac{y_{i+1} - y_i}{h}$$

$i = 0, 1, \dots, n - 1$, and

$$z_n = \frac{y_n - y_{n-1}}{h}.$$

The collection $(x_i, z_i)_{i=0}^n$ is the discrete derivative of the discrete version $(x_i, f_i)_{i=0}^n$ of the continuous function $f(x)$. The program below, found in the file `diff_func.py`, takes f, a, b and n as input and computes the discrete derivative of f on the mesh implied by a, b , and h , and then a plot of f and the discrete derivative is made.

```
def diff(f, a, b, n):
    x = linspace(a, b, n+1)
    y = zeros(len(x))
    z = zeros(len(x))
    h = (b-a)/float(n)
    for i in xrange(len(x)):
        y[i] = func(x[i])
    for i in xrange(len(x)-1):
        z[i] = (y[i+1] - y[i])/h
    z[n] = (y[n] - y[n-1])/h
    return y, z

from scitools.std import *
f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
n = int(sys.argv[4])

f = StringFunction(f_formula)
y, z = diff(f, a, b, n)
plot(x, y, 'r-', x, z, 'b-',
      legend=('function', 'derivative'))
```

A.3 Integration Becomes Summation

Some functions can be integrated analytically. You may remember⁶ the following cases,

$$\begin{aligned}\int x^m dx &= \frac{1}{m+1} x^{m+1} \text{ for } m \neq -1, \\ \int \sin(x) dx &= -\cos(x), \\ \int \frac{x}{1+x^2} dx &= \frac{1}{2} \ln(x^2 + 1).\end{aligned}$$

These are examples of so-called indefinite integrals. If the function can be integrated analytically, it is straightforward to evaluate an associated definite integral. For example, we have⁷

⁶ Actually, we congratulate you if you remember the third one!

⁷ Recall, in general, that

$$[f(x)]_a^b = f(b) - f(a).$$

$$\begin{aligned}\int_0^1 x^m dx &= \left[\frac{1}{m+1} x^{m+1} \right]_0^1 = \frac{1}{m+1}, \\ \int_0^\pi \sin(x) dx &= [-\cos(x)]_0^\pi = 2, \\ \int_0^1 \frac{x}{1+x^2} dx &= \left[\frac{1}{2} \ln(x^2+1) \right]_0^1 = \frac{1}{2} \ln 2.\end{aligned}$$

But lots of functions cannot be integrated analytically and therefore definite integrals must be computed using some sort of numerical approximation. Above, we introduced the discrete version of a function, and we will now use this construction to compute an approximation of a definite integral.

A.3.1 Dividing into Subintervals

Let us start by considering the problem of computing the integral of $\sin(x)$ from $x = 0$ to $x = \pi$. This is not the most exciting or challenging mathematical problem you can think of, but it is good practice to start with a problem you know well when you want to learn a new method. In Chapter A.1.1 we introduce a discrete function $(x_i, s_i)_{i=0}^n$ where $h = \pi/n$, $s_i = \sin(x_i)$ and $x_i = ih$ for $i = 0, 1, \dots, n$. Furthermore, in the interval $x_k \leq x < x_{k+1}$, we defined the linear function

$$S_k(x) = s_k + \frac{s_{k+1} - s_k}{x_{k+1} - x_k}(x - x_k).$$

We want to compute an approximation of the integral of the function $\sin(x)$ from $x = 0$ to $x = \pi$. The integral

$$\int_0^\pi \sin(x) dx$$

can be divided into subintegrals defined on the intervals $x_k \leq x < x_{k+1}$, leading to the following sum of integrals:

$$\int_0^\pi \sin(x) dx = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} \sin(x) dx.$$

To get a feeling for this split of the integral, let us spell the sum out in the case of only four subintervals. Then $n = 4$, $h = \pi/4$,

$$\begin{aligned}
x_0 &= 0, \\
x_1 &= \pi/4, \\
x_2 &= \pi/2, \\
x_3 &= 3\pi/4 \\
x_4 &= \pi.
\end{aligned}$$

The interval from 0 to π is divided into four intervals of equal length, and we can divide the integral similarly,

$$\begin{aligned}
\int_0^\pi \sin(x)dx &= \int_{x_0}^{x_1} \sin(x)dx + \int_{x_1}^{x_2} \sin(x)dx + \\
&\quad \int_{x_2}^{x_3} \sin(x)dx + \int_{x_3}^{x_4} \sin(x)dx. \quad (\text{A.10})
\end{aligned}$$

So far we have changed nothing – the integral can be split in this way – with no approximation at all. But we have reduced the problem of approximating the integral

$$\int_0^\pi \sin(x)dx$$

down to approximating integrals on the subintervals, i.e. we need approximations of all the following integrals

$$\int_{x_0}^{x_1} \sin(x)dx, \int_{x_1}^{x_2} \sin(x)dx, \int_{x_2}^{x_3} \sin(x)dx, \int_{x_3}^{x_4} \sin(x)dx.$$

The idea is that the function to be integrated changes less over the subintervals than over the whole domain $[0, \pi]$ and it might be reasonable to approximate the sine by a straight line, $S_k(x)$, over each subinterval. The integration over a subinterval will then be very easy.

A.3.2 Integration on Subintervals

The task now is to approximate integrals on the form

$$\int_{x_k}^{x_{k+1}} \sin(x)dx.$$

Since

$$\sin(x) \approx S_k(x)$$

on the interval (x_k, x_{k+1}) , we have

$$\int_{x_k}^{x_{k+1}} \sin(x)dx \approx \int_{x_k}^{x_{k+1}} S_k(x)dx.$$

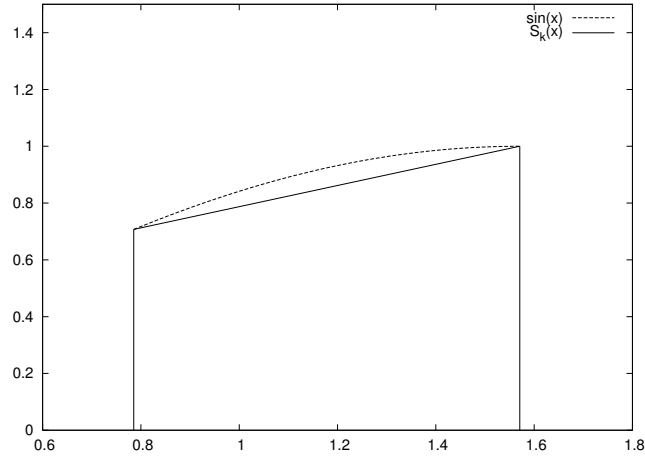


Fig. A.3 $S_k(x)$ and $\sin(x)$ on the interval (x_k, x_{k+1}) for $k = 1$ and $n = 4$.

In Figure A.3 we have graphed $S_k(x)$ and $\sin(x)$ on the interval (x_k, x_{k+1}) for $k = 1$ in the case of $n = 4$. We note that the integral of $S_1(x)$ on this interval equals the area of a trapezoid, and thus we have

$$\int_{x_1}^{x_2} S_1(x) dx = \frac{1}{2} (S_1(x_2) + S_1(x_1)) (x_2 - x_1),$$

so

$$\int_{x_1}^{x_2} S_1(x) dx = \frac{h}{2} (s_2 + s_1),$$

and in general we have

$$\begin{aligned} \int_{x_k}^{x_{k+1}} \sin(x) dx &\approx \frac{1}{2} (s_{k+1} + s_k) (x_{k+1} - x_k) \\ &= \frac{h}{2} (s_{k+1} + s_k). \end{aligned}$$

A.3.3 Adding the Subintervals

By adding the contributions from each subinterval, we get

$$\begin{aligned} \int_0^\pi \sin(x) dx &= \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} \sin(x) dx \\ &\approx \sum_{k=0}^{n-1} \frac{h}{2} (s_{k+1} + s_k), \end{aligned}$$

so

$$\int_0^\pi \sin(x) dx \approx \frac{h}{2} \sum_{k=0}^{n-1} (s_{k+1} + s_k). \quad (\text{A.11})$$

In the case of $n = 4$, we have

$$\begin{aligned}\int_0^\pi \sin(x)dx &\approx \frac{h}{2} [(s_1 + s_0) + (s_2 + s_1) + (s_3 + s_2) + (s_4 + s_3)] \\ &= \frac{h}{2} [s_0 + 2(s_1 + s_2 + s_3) + s_4] .\end{aligned}$$

One can show that (A.11) can be alternatively expressed as⁸

$$\int_0^\pi \sin(x)dx \approx \frac{h}{2} \left[s_0 + 2 \sum_{k=1}^{n-1} s_k + s_n \right] . \quad (\text{A.12})$$

This approximation formula is referred to as the Trapezoidal rule of numerical integration. Using the more general program `trapezoidal.py`, presented in the next section, on integrating $\int_0^\pi \sin(x)dx$ with $n = 5, 10, 20$ and 100 yields the numbers 1.5644, 1.8864, 1.9713, and 1.9998 respectively. These numbers are to be compared to the exact value 2. As usual, the approximation becomes better the more points (n) we use.

A.3.4 Generalization

An approximation of the integral

$$\int_a^b f(x)dx$$

can be computed using the discrete version of a continuous function $f(x)$ defined on an interval $[a, b]$. We recall that the discrete version of f is given by $(x_i, y_i)_{i=0}^n$ where

$$x_i = a + ih, \text{ and } y_i = f(x_i)$$

for $i = 0, 1, \dots, n$. Here, $n \geq 1$ is a given integer and $h = (b - a)/n$. The Trapezoidal rule can now be written as

$$\int_a^b f(x)dx \approx \frac{h}{2} \left[y_0 + 2 \sum_{k=1}^{n-1} y_k + y_n \right] .$$

The program `trapezoidal.py` implements the Trapezoidal rule for a general function f .

```
def trapezoidal(f, a, b, n):
    h = (b-a)/float(n)
    I = f(a) + f(b)
    for k in xrange(1, n, 1):
```

⁸ There are fewer arithmetic operations associated with (A.12) than with (A.11), so the former will lead to faster code.

```

        x = a + k*h
        I += 2*f(x)
    I *= h/2
    return I

from math import *
from scitools.StringFunction import StringFunction
import sys

def test(argv=sys.argv):
    f_formula = argv[1]
    a = eval(argv[2])
    b = eval(argv[3])
    n = int(argv[4])

    f = StringFunction(f_formula)
    I = trapezoidal(f, a, b, n)
    print 'Approximation of the integral: ', I

if __name__ == '__main__':
    test()

```

We have made the file as module such that you can easily import the `trapezoidal` function in another program. Let us do that: We make a table of how the approximation and the associated error of an integral are reduced as n is increased. For this purpose, we want to integrate $\int_{t_1}^{t_2} g(t)dt$, where

$$g(t) = -ae^{-at} \sin(\pi wt) + \pi we^{-at} \cos(\pi wt).$$

The exact integral $G(t) = \int g(t)dt$ equals

$$G(t) = e^{-at} \sin(\pi wt).$$

Here, a and w are real numbers that we set to $1/2$ and 1 , respectively, in the program. The integration limits are chosen as $t_1 = 0$ and $t_2 = 4$. The integral then equals zero. The program and its output appear below.

```

from trapezoidal import trapezoidal
from math import exp, sin, cos, pi

def g(t):
    return -a*exp(-a*t)*sin(pi*w*t) + pi*w*exp(-a*t)*cos(pi*w*t)

def G(t): # integral of g(t)
    return exp(-a*t)*sin(pi*w*t)

a = 0.5
w = 1.0
t1 = 0
t2 = 4
exact = G(t2) - G(t1)
for n in 2, 4, 8, 16, 32, 64, 128, 256, 512:
    approx = trapezoidal(g, t1, t2, n)
    print 'n=%3d approximation=%12.5e error=%12.5e' % \
        (n, approx, exact-approx)

```



```

n= 2 approximation= 5.87822e+00 error=-5.87822e+00
n= 4 approximation= 3.32652e-01 error=-3.32652e-01
n= 8 approximation= 6.15345e-02 error=-6.15345e-02
n= 16 approximation= 1.44376e-02 error=-1.44376e-02
n= 32 approximation= 3.55482e-03 error=-3.55482e-03
n= 64 approximation= 8.85362e-04 error=-8.85362e-04
n=128 approximation= 2.21132e-04 error=-2.21132e-04
n=256 approximation= 5.52701e-05 error=-5.52701e-05
n=512 approximation= 1.38167e-05 error=-1.38167e-05

```

We see that the error is reduced as we increase n . In fact, as n is doubled we realize that the error is roughly reduced by a factor of 4, at least when $n > 8$. This is an important property of the Trapezoidal rule, and checking that a program reproduces this property is an important check of the validity of the implementation.

A.4 Taylor Series

The single most important mathematical tool in computational science is the Taylor series. It is used to derive new methods and also for the analysis of the accuracy of approximations. We will use the series many times in this text. Right here, we just introduce it and present a few applications.

A.4.1 Approximating Functions Close to One Point

Suppose you know the value of a function f at some point x_0 , and you are interested in the value of f close to x . More precisely, suppose we know $f(x_0)$ and we want an approximation of $f(x_0 + h)$ where h is a small number. If the function is smooth and h is really small, our first approximation reads

$$f(x_0 + h) \approx f(x_0). \quad (\text{A.13})$$

That approximation is, of course, not very accurate. In order to derive a more accurate approximation, we have to know more about f at x_0 . Suppose that we know the value of $f(x_0)$ and $f'(x_0)$, then we can find a better approximation of $f(x_0 + h)$ by recalling that

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

Hence, we have

$$f(x_0 + h) \approx f(x_0) + hf'(x_0). \quad (\text{A.14})$$

A.4.2 Approximating the Exponential Function

Let us be a bit more specific and consider the case of

$$f(x) = e^x$$

around

$$x_0 = 0.$$

Since $f'(x) = e^x$, we have $f'(0) = 1$, and then it follows from (A.14) that

$$e^h \approx 1 + h.$$

The little program below (found in `taylor1.py`) prints e^h and $1 + h$ for a range of h values.

```
from math import exp
for h in 1, 0.5, 1/20.0, 1/100.0, 1/1000.0:
    print 'h=%8.6f exp(h)=%11.5e 1+h=%g' % (h, exp(h), 1+h)

h=1.000000 exp(h)=2.71828e+00 1+h=2
h=0.500000 exp(h)=1.64872e+00 1+h=1.5
h=0.050000 exp(h)=1.05127e+00 1+h=1.05
h=0.010000 exp(h)=1.01005e+00 1+h=1.01
h=0.001000 exp(h)=1.00100e+00 1+h=1.001
```

As expected, $1 + h$ is a good approximation to e^h the smaller h is.

A.4.3 More Accurate Expansions

The approximations given by (A.13) and (A.14) are referred to as Taylor series. You can read much more about Taylor series in any Calculus book. More specifically, (A.13) and (A.14) are known as the zeroth- and first-order Taylor series, respectively. The second-order Taylor series is given by

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0), \quad (\text{A.15})$$

the third-order series is given by

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0), \quad (\text{A.16})$$

and the fourth-order series reads

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0) + \frac{h^4}{24}f^{(4)}(x_0). \quad (\text{A.17})$$

In general, the n -th order Taylor series is given by

$$f(x_0 + h) \approx \sum_{k=0}^n \frac{h^k}{k!} f^{(k)}(x_0), \quad (\text{A.18})$$

where we recall that $f^{(k)}$ denotes the k -th derivative of f , and

$$k! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots (k-1) \cdot k$$

is the factorial (cf. Exercise 2.33). By again considering $f(x) = e^x$ and $x_0 = 0$, we have

$$f(x_0) = f'(x_0) = f''(x_0) = f'''(x_0) = f''''(x_0) = 1$$

which gives the following Taylor series:

$$\begin{array}{ll} e^h \approx 1, & \text{zeroth-order,} \\ e^h \approx 1 + h, & \text{first-order,} \\ e^h \approx 1 + h + \frac{1}{2}h^2, & \text{second-order,} \\ e^h \approx 1 + h + \frac{1}{2}h^2 + \frac{1}{6}h^3, & \text{third-order,} \\ e^h \approx 1 + h + \frac{1}{2}h^2 + \frac{1}{6}h^3 + \frac{1}{24}h^4, & \text{fourth-order.} \end{array}$$

The program below, called `taylor2.py`, prints the error of these approximations for a given value of h (note that we can easily build up a Taylor series in a list by adding a new term to the last computed term in the list).

```
from math import exp
import sys
h = float(sys.argv[1])

Taylor_series = []
Taylor_series.append(1)
Taylor_series.append(Taylor_series[-1] + h)
Taylor_series.append(Taylor_series[-1] + (1/2.0)*h**2)
Taylor_series.append(Taylor_series[-1] + (1/6.0)*h**3)
Taylor_series.append(Taylor_series[-1] + (1/24.0)*h**4)

print 'h =', h
for order in range(len(Taylor_series)):
    print 'order=%d, error=%g' % \
        (order, exp(h) - Taylor_series[order])
```

By running the program with $h = 0.2$, we have the following output:

```
h = 0.2
order=0, error=0.221403
order=1, error=0.0214028
order=2, error=0.00140276
order=3, error=6.94248e-05
order=4, error=2.75816e-06
```

We see how much the approximation is improved by adding more terms. For $h = 3$ all these approximations are useless:

```
h = 3.0
order=0, error=19.0855
order=1, error=16.0855
order=2, error=11.5855
order=3, error=7.08554
order=4, error=3.71054
```

However, by adding more terms we can get accurate results for any h . The method from Chapter 5.1.7 computes the Taylor series for e^x with n terms in general. Running the associated program `exp_Taylor_series_diffeq.py` for various values of h shows how much is gained by adding more terms to the Taylor series. For $h = 3$,

	$n + 1$	Taylor series	
$e^3 = 20.086$ and we have	2	4	For $h = 50$, $e^{50} =$
	4	13	
	8	19.846	
	16	20.086	
	$n + 1$	Taylor series	
$5.1847 \cdot 10^{21}$ and we have	2	51	Here, the evolution of
	4	$2.2134 \cdot 10^4$	
	8	$1.7960 \cdot 10^8$	
	16	$3.2964 \cdot 10^{13}$	
	32	$1.3928 \cdot 10^{19}$	
	64	$5.0196 \cdot 10^{21}$	
	128	$5.1847 \cdot 10^{21}$	

the series as more terms are added is quite dramatic (and impressive!).

A.4.4 Accuracy of the Approximation

Recall that the Taylor series is given by

$$f(x_0 + h) \approx \sum_{k=0}^n \frac{h^k}{k!} f^{(k)}(x_0). \quad (\text{A.19})$$

This can be rewritten as an equality by introducing an error term,

$$f(x_0 + h) = \sum_{k=0}^n \frac{h^k}{k!} f^{(k)}(x_0) + O(h^{n+1}). \quad (\text{A.20})$$

Let's look a bit closer at this for $f(x) = e^x$. In the case of $n = 1$, we have

$$e^h = 1 + h + O(h^2). \quad (\text{A.21})$$

This means that there is a constant c that does not depend on h such that

$$\left| e^h - (1 + h) \right| \leq ch^2, \quad (\text{A.22})$$

so the error is reduced quadratically in h . This means that if we compute the fraction

$$q_h^1 = \frac{|e^h - (1 + h)|}{h^2},$$

we expect it to be bounded as h is reduced. The program `taylor_err1.py` prints q_h^1 for $h = 1/10, 1/20, 1/100$ and $1/1000$.

```
from numpy import exp, abs

def q_h(h):
    return abs(exp(h) - (1+h))/h**2
```

```
print "  h      q_h"
for h in 0.1, 0.05, 0.01, 0.001:
    print "%5.3f %f" %(h, q_h(h))
```

We can run the program and watch the output:

Terminal

```
taylor_err1.py
  h      q_h
0.100 0.517092
0.050 0.508439
0.010 0.501671
0.001 0.500167
```

We observe that $q_h \approx 1/2$ and it is definitely bounded independent of h . We can now rewrite all the approximations of e^h defined above in term of equalities:

$$\begin{aligned}
 e^h &= 1 + O(h), && \text{zeroth-order,} \\
 e^h &= 1 + h + O(h^2), && \text{first-order,} \\
 e^h &= 1 + h + \frac{1}{2}h^2 + O(h^3), && \text{second-order,} \\
 e^h &= 1 + h + \frac{1}{2}h^2 + \frac{1}{6}h^3 + O(h^4), && \text{third-order,} \\
 e^h &= 1 + h + \frac{1}{2}h^2 + \frac{1}{6}h^3 + \frac{1}{24}h^4 + O(h^5), && \text{fourth-order.}
 \end{aligned}$$

The program `taylor_err2.py` prints

$$\begin{aligned}
 q_h^0 &= \frac{|e^h - 1|}{h}, \\
 q_h^1 &= \frac{|e^h - (1 + h)|}{h^2}, \\
 q_h^2 &= \frac{|e^h - (1 + h + \frac{h^2}{2})|}{h^3}, \\
 q_h^3 &= \frac{|e^h - (1 + h + \frac{h^2}{2} + \frac{h^3}{6})|}{h^4}, \\
 q_h^4 &= \frac{|e^h - (1 + h + \frac{h^2}{2} + \frac{h^3}{6} + \frac{h^4}{24})|}{h^5},
 \end{aligned}$$

for $h = 1/5, 1/10, 1/20$ and $1/100$.

```
from numpy import exp, abs

def q_0(h):
    return abs(exp(h) - 1) / h
def q_1(h):
    return abs(exp(h) - (1 + h)) / h**2
def q_2(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2)) / h**3
def q_3(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2 + \
                          (1/6.0)*h**3)) / h**4
def q_4(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2 + (1/6.0)*h**3 + \
```

```

(1/24.0)*h**4)) / h**5
hlist = [0.2, 0.1, 0.05, 0.01]
print "%-05s %-09s %-09s %-09s %-09s %-09s" \
      %("h", "q_0", "q_1", "q_2", "q_3", "q_4")
for h in hlist:
    print "%.02f %.04f %.04f %.04f %.04f %.04f" \
          %(h, q_0(h), q_1(h), q_2(h), q_3(h), q_4(h))

```

By using the program, we get the following table:

h	q_0	q_1	q_2	q_3	q_4
0.20	1.107014	0.535069	0.175345	0.043391	0.008619
0.10	1.051709	0.517092	0.170918	0.042514	0.008474
0.05	1.025422	0.508439	0.168771	0.042087	0.008403
0.01	1.005017	0.501671	0.167084	0.041750	0.008344

Again we observe that the error of the approximation behaves as indicated in (A.20).

A.4.5 Derivatives Revisited

We observed above that

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

By using the Taylor series, we can obtain this approximation directly, and also get an indication of the error of the approximation. From (A.20) it follows that

$$f(x+h) = f(x) + hf'(x) + O(h^2),$$

and thus

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h), \quad (\text{A.23})$$

so the error is proportional to h . We can investigate if this is the case through some computer experiments. Take $f(x) = \ln(x)$, so that $f'(x) = 1/x$. The program `diff_ln_err.py` prints h and

$$\frac{1}{h} \left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| \quad (\text{A.24})$$

at $x = 10$ for a range of h values.

```

def error(h):
    return (1.0/h)*abs(df(x) - (f(x+h)-f(x))/h)

from math import log as ln

def f(x):
    return ln(x)

def df(x):
    return 1.0/x

x = 10
hlist = []

```

```
for h in 0.2, 0.1, 0.05, 0.01, 0.001:
    print "%.4f" %4f " % (h, error(h))
```

From the output

```
0.2000  0.004934
0.1000  0.004967
0.0500  0.004983
0.0100  0.004997
0.0010  0.005000
```

we observe that the quantity in (A.24) is constant (≈ 0.5) independent of h , which indicates that the error is proportional to h .

A.4.6 More Accurate Difference Approximations

We can also use the Taylor series to derive more accurate approximations of the derivatives. From (A.20), we have

$$f(x+h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3). \quad (\text{A.25})$$

By using $-h$ instead of h , we get

$$f(x-h) \approx f(x) - hf'(x) + \frac{h^2}{2}f''(x) + O(h^3). \quad (\text{A.26})$$

By subtracting (A.26) from (A.25), we have

$$f(x+h) - f(x-h) = 2hf'(x) + O(h^3),$$

and consequently

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2). \quad (\text{A.27})$$

Note that the error is now $O(h^2)$ whereas the error term of (A.23) is $O(h)$. In order to see if the error is actually reduced, let us compare the following two approximations

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \text{ and } f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

by applying them to the discrete version of $\sin(x)$ on the interval $(0, \pi)$. As usual, we let $n \geq 1$ be a given integer, and define the mesh

$$x_i = ih \text{ for } i = 0, 1, \dots, n,$$

where $h = \pi/n$. At the nodes, we have the functional values

$$s_i = \sin(x_i) \text{ for } i = 0, 1, \dots, n,$$

and at the inner nodes we define the first (F) and second (S) order approximations of the derivatives given by

$$d_i^F = \frac{s_{i+1} - s_i}{h},$$

and

$$d_i^S = \frac{s_{i+1} - s_{i-1}}{2h},$$

respectively for $i = 1, 2, \dots, n-1$. These values should be compared to the exact derivative given by

$$d_i = \cos(x_i) \text{ for } i = 1, 2, \dots, n-1.$$

The following program, found in `diff_1st2nd_order.py`, plots the discrete functions $(x_i, d_i)_{i=1}^{n-1}$, $(x_i, d_i^F)_{i=1}^{n-1}$, and $(x_i, d_i^S)_{i=1}^{n-1}$ for a given n . Note that the first three functions in this program are completely general in that they can be used for any $f(x)$ on any mesh. The special case of $f(x) = \sin(x)$ and comparing first- and second-order formulas is implemented in the `example` function. This latter function is called in the test block of the file. That is, the file is a module and we can reuse the first three functions in other programs (in particular, we can use the third function in the next example).

```
def first_order(f, x, h):
    return (f(x+h) - f(x))/h

def second_order(f, x, h):
    return (f(x+h) - f(x-h))/(2*h)

def derivative_on_mesh(formula, f, a, b, n):
    """
    Differentiate f(x) at all internal points in a mesh
    on [a,b] with n+1 equally spaced points.
    The differentiation formula is given by formula(f, x, h).
    """
    h = (b-a)/float(n)
    x = linspace(a, b, n+1)
    df = zeros(len(x))
    for i in xrange(1, len(x)-1):
        df[i] = formula(f, x[i], h)
    # return x and values at internal points only
    return x[1:-1], df[1:-1]

def example(n):
    a = 0; b = pi;
    x, dF = derivative_on_mesh(first_order, sin, a, b, n)
    x, dS = derivative_on_mesh(second_order, sin, a, b, n)
    # accurate plot of the exact derivative at internal points:
    h = (b-a)/float(n)
    xfine = linspace(a+h, b-h, 1001)
    exact = cos(xfine)
    plot(x, dF, 'r-', x, dS, 'b-', xfine, exact, 'y-',
         legend=('First-order derivative',
                 'Second-order derivative',
                 'Correct function'),
         title='Approximate and correct discrete '\
               'functions, n=%d' % n)

# main program:
from scitools.std import *
try:
    n = int(sys.argv[1])
```



```
except:
    print "usage: %s n" %sys.argv[0]
    sys.exit(1)

example(n)
```

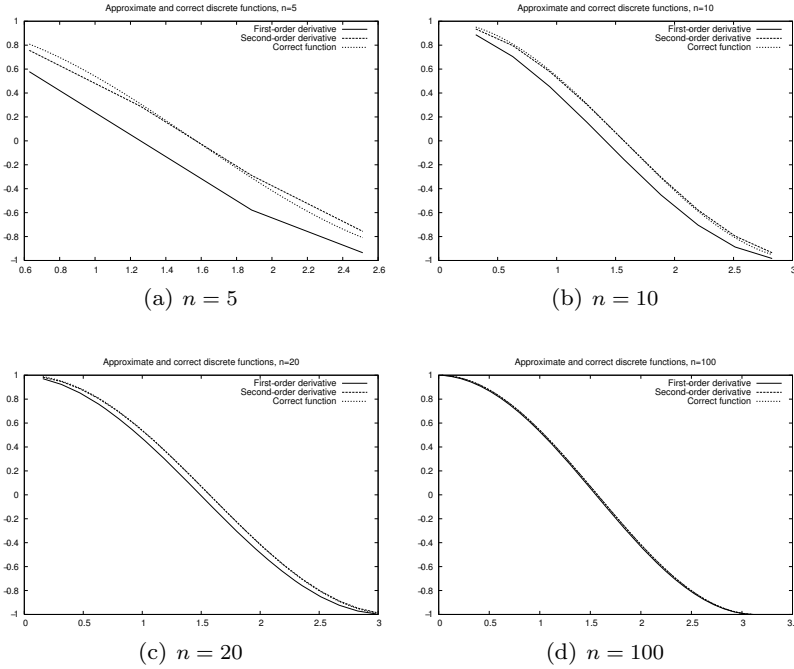


Fig. A.4 Plots of exact and approximate derivatives with various number of mesh points n .

The result of running the program with four different n values is presented in Figure A.4. Observe that d_i^S is a better approximation to d_i^F than d_i^F , and note that both approximations become very good as n is getting large.

A.4.7 Second-Order Derivatives

We have seen that the Taylor series can be used to derive approximations of the derivative. But what about higher order derivatives? Next we shall look at second order derivatives. From (A.20) we have

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0) + O(h^4),$$

and by using $-h$, we have

$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) - \frac{h^3}{6}f'''(x_0) + O(h^4)$$

By adding these equations, we have

$$f(x_0 + h) + f(x_0 - h) = 2f(x_0) + h^2 f''(x_0) + O(h^4),$$

and thus

$$f''(x_0) = \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} + O(h^2). \quad (\text{A.28})$$

For a discrete function $(x_i, y_i)_{i=0}^n$, $y_i = f(x_i)$, we can define the following approximation of the second derivative,

$$d_i = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}. \quad (\text{A.29})$$

We can make a function, found in the file `diff2nd.py`, that evaluates (A.29) on a mesh. As an example, we apply the function to

$$f(x) = \sin(e^x),$$

where the exact second-order derivative is given by

$$f''(x) = e^x \cos(e^x) - (\sin(e^x)) e^{2x}.$$

```
from diff_1st2nd_order import derivative_on_mesh
from scitools.std import *

def diff2nd(f, x, h):
    return (f(x+h) - 2*f(x) + f(x-h))/(h**2)

def example(n):
    a = 0; b = pi

    def f(x):
        return sin(exp(x))

    def exact_d2f(x):
        e_x = exp(x)
        return e_x*cos(e_x) - sin(e_x)*exp(2*x)

    x, d2f = derivative_on_mesh(diff2nd, f, a, b, n)
    h = (b-a)/float(n)
    xfine = linspace(a+h, b-h, 1001) # fine mesh for comparison
    exact = exact_d2f(xfine)
    plot(x, d2f, 'r-', xfine, exact, 'b-',
         legend=('Approximate derivative',
                 'Correct function'),
         title='Approximate and correct second order '\
               'derivatives, n=%d' % n,
         hardcopy='tmp.eps')

n = int(sys.argv[1])
example(n)
```

In Figure A.5 we compare the exact and the approximate derivatives for $n = 10, 20, 50$, and 100 . As usual, the error decreases when n becomes larger, but note here that the error is very large for small values of n .

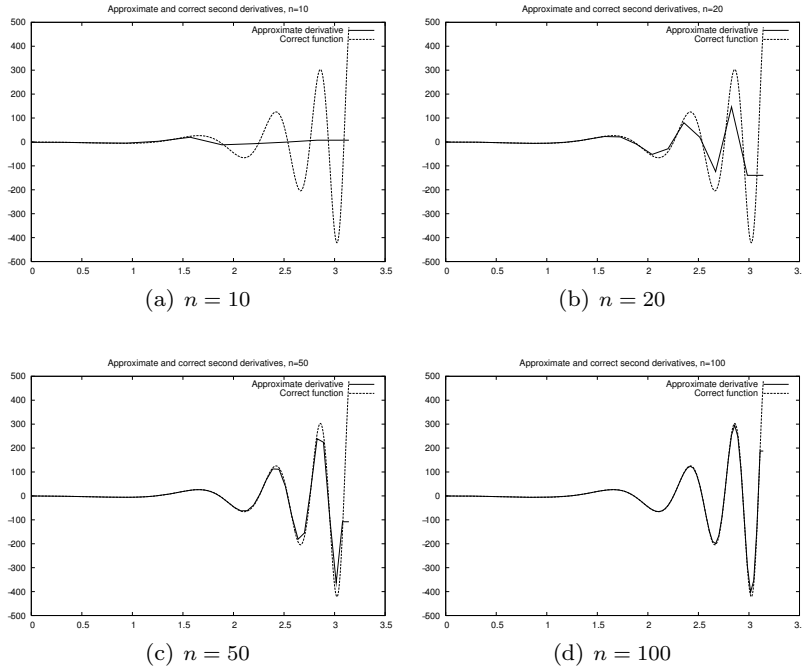


Fig. A.5 Plots of exact and approximate second-order derivatives with various mesh resolution n .

A.5 Exercises

Exercise A.1. Interpolate a discrete function.

In a Python function, represent the mathematical function

$$f(x) = \exp(-x^2) \cos(2\pi x)$$

on a mesh consisting of $q + 1$ equally spaced points on $[-1, 1]$, and return 1) the interpolated function value at $x = -0.45$ and 2) the error in the interpolated value. Call the function and write out the error for $q = 2, 4, 8, 16$. Name of program file: `interpolate_exp_cos.py` ◇

Exercise A.2. Study a function for different parameter values.

Develop a program that creates a plot of the function $f(x) = \sin(\frac{1}{x+\varepsilon})$ for x in the unit interval, where $\varepsilon > 0$ is a given input parameter. Use $n + 1$ nodes in the plot.

- Test the program using $n = 10$ and $\varepsilon = 1/5$.
- Refine the program such that it plots the function for two values of n ; say n and $n + 10$.
- How large do you have to choose n in order for the difference between these two functions to be less than 0.1? Hint: Each function gives an array. Create a `while` loop and use the `max` function of the arrays to retrieve the maximum value and compare these.

- (d) Let $\varepsilon = 1/10$, and repeat (c).
- (e) Let $\varepsilon = 1/20$, and repeat (c).
- (f) Try to find a formula for how large n needs to be for a given value of ε such that increasing n further does not change the plot so much that it is visible on the screen. Note that there is no exact answer to this question.

Name of program file: `plot_sin_eps.py`

◇

Exercise A.3. *Study a function and its derivative.*

Consider the function

$$f(x) = \sin\left(\frac{1}{x + \varepsilon}\right)$$

for x ranging from 0 to 1, and the derivative

$$f'(x) = \frac{-\cos\left(\frac{1}{x + \varepsilon}\right)}{(x + \varepsilon)^2}.$$

Here, ε is a given input parameter.

- (a) Develop a program that creates a plot of the derivative of $f = f(x)$ based on a finite difference approximation using n computational nodes. The program should also graph the exact derivative given by $f' = f'(x)$ above.
- (b) Test the program using $n = 10$ and $\varepsilon = 1/5$.
- (c) How large do you have to choose n in order for the difference between these two functions to be less than 0.1? Hint: Each function gives an array. Create a `while` loop and use the `max` function of the arrays to retrieve the maximum value and compare these.
- (d) Let $\varepsilon = 1/10$, and repeat (c).
- (e) Let $\varepsilon = 1/20$, and repeat (c).
- (f) Try determine experimentally how large n needs to be for a given value of ε such that increasing n further does not change the plot so much that you can view it on the screen. Note, again, that there is no exact solution to this problem.

Name of program file: `sin_deriv.py`

◇

Exercise A.4. *Use the Trapezoidal method.*

The purpose of this exercise is to test the program `trapezoidal.py`.

- (a) Let

$$\bar{a} = \int_0^1 e^{4x} dx = \frac{1}{4}e^4 - \frac{1}{4}.$$

Compute the integral using the program `trapezoidal.py` and, for a given n , let $a(n)$ denote the result. Try to find, experimentally, how large you have to choose n in order for

$$|\bar{a} - a(n)| \leq \varepsilon$$

where $\varepsilon = 1/100$.

(b) Repeat (a) with $\varepsilon = 1/1000$.

(c) Repeat (a) with $\varepsilon = 1/10000$.

(d) Try to figure out, in general, how large n has to be in order for

$$|\bar{a} - a(n)| \leq \varepsilon$$

for a given value of ε .

Name of program file: `trapezoidal_test_exp.py`

◇

Exercise A.5. *Compute a sequence of integrals.*

(a) Let

$$\bar{b}_k = \int_0^1 x^k dx = \frac{1}{k+1},$$

and let $b_k(n)$ denote the result of using the program `trapezoidal.py` to compute $\int_0^1 x^k dx$. For $k = 4, 6$ and 8 , try to figure out, by doing numerical experiments, how large n needs to be in order for $b_k(n)$ to satisfy

$$|\bar{b}_k - b_k(n)| \leq 0.0001.$$

Note that n will depend on k . Hint: Run the program for each k , look at the output, and calculate $|\bar{b}_k - b_k(n)|$ manually.

(b) Try to generalize the result in (a) to arbitrary $k \geq 2$.

(c) Generate a plot of x^k on the unit interval for $k = 2, 4, 6, 8$, and 10 , and try to figure out if the results obtained in (a) and (b) are reasonable taking into account that the program `trapezoidal.py` was developed using a piecewise linear approximation of the function.

Name of program file: `trapezoidal_test_power.py`

◇

Exercise A.6. *Use the Trapezoidal method.*

The purpose of this exercise is to compute an approximation of the integral⁹

$$I = \int_{-\infty}^{\infty} e^{-x^2} dx$$

using the Trapezoidal method.

(a) Plot the function e^{-x^2} for x ranging from -10 to 10 and use the plot to argue that

$$\int_{-\infty}^{\infty} e^{-x^2} dx = 2 \int_0^{\infty} e^{-x^2} dx.$$

(b) Let $T(n, L)$ be the approximation of the integral

⁹ You may consult your Calculus book to verify that the exact solution is $\sqrt{\pi}$.

$$2 \int_0^L e^{-x^2} dx$$

computed by the Trapezoidal method using n computational points. Develop a program that computes the value of T for a given n and L .

- (c) Extend the program developed in (b) to write out values of $T(n, L)$ in a table with rows corresponding to $n = 100, 200, \dots, 500$ and columns corresponding to $L = 2, 4, 6, 8, 10$.
- (d) Extend the program to also print a table of the errors in $T(n, L)$ for the same n and L values as in (c). The exact value of the integral is $\sqrt{\pi}$.

Comment. Numerical integration of integrals with finite limits requires a choice of n , while with infinite limits we also need to truncate the domain, i.e., choose L in the present example. The accuracy depends on both n and L . Name of program file: `integrate_exp.py` \diamond

Exercise A.7. Trigonometric integrals.

The purpose of this exercise is to demonstrate a property of trigonometric functions that you will meet in later courses. In this exercise, you may compute the integrals using the program `trapezoidal.py` with $n = 100$.

- (a) Consider the integrals

$$I_{p,q} = 2 \int_0^1 \sin(p\pi x) \sin(q\pi x) dx$$

and fill in values of the integral $I_{p,q}$ in a table with rows corresponding to $q = 0, 1, \dots, 4$ and columns corresponding to $p = 0, 1, \dots, 4$.

- (b) Repeat (a) for the integrals

$$I_{p,q} = 2 \int_0^1 \cos(p\pi x) \cos(q\pi x) dx.$$

- (c) Repeat (a) for the integrals

$$I_{p,q} = 2 \int_0^1 \cos(p\pi x) \sin(q\pi x) dx.$$

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

Exercise A.8. Plot functions and their derivatives.

- (a) Use the program `diff_func.py` to plot approximations of the derivative for the following functions defined on the interval ranging from $x = 1/1000$ to $x = 1$:

$$f(x) = \ln \left(x + \frac{1}{100} \right),$$

$$g(x) = \cos(e^{10x}),$$

$$h(x) = x^x.$$

- (b) Extend the program such that both the discrete approximation and the correct (analytical) derivative can be plotted. The analytical derivative should be evaluated in the same computational points as the numerical approximation. Test the program by comparing the discrete and analytical derivative of x^3 .
- (c) Use the program developed in (b) to compare the analytical and discrete derivatives of the functions given in (a). How large do you have to choose n in each case in order for the plots to become indistinguishable on your screen. Note that the analytical derivatives are given by:

$$f'(x) = \frac{1}{x + \frac{1}{100}},$$

$$g'(x) = -10e^{10x} \sin(e^{10x})$$

$$h'(x) = (\ln x) x^x + x x^{x-1}$$

Name of program file: `diff_functions.py`

◇

Exercise A.9. Use the Trapezoidal method.

Develop an efficient program that creates a plot of the function

$$f(x) = \frac{1}{2} + \frac{1}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

for $x \in [0, 10]$. The integral should be approximated using the Trapezoidal method and use as few function evaluations of e^{-t^2} as possible.

Name of program file: `plot_integral.py`

◇

This appendix is authored by Aslak Tveito

Differential equations have proven to be an immensely successful instrument for modeling phenomena in science and technology. It is hardly an exaggeration to say that differential equations are used to define mathematical models in virtually all parts of the natural sciences. In this chapter, we will take the first steps towards learning how to deal with differential equations on a computer. This is a core issue in Computational Science and reaches far beyond what we can cover in this text. However, the ideas you will see here are reused in lots of advanced applications, so this chapter will hopefully provide useful introduction to a topic that you will probably encounter many times later.

We will show you how to build programs for solving differential equations. More precisely, we will show how a differential equation can be formulated in a discrete manner suitable for analysis on a computer, and how to implement programs to compute the discrete solutions. The simplest differential equations can be solved analytically in the sense that you can write down an explicit formula for the solutions. However, differential equations arising in practical applications are usually rather complicated and thus have to be solved numerically on a computer. Therefore we focus on implementing numerical methods to solve the equations. Chapters 7.4 and 9.4 describe more advanced implementation techniques aimed at making an easy-to-use toolbox for solving differential equations. Exercises in this appendix and the mentioned chapters aim at solving a variety of differential equations arising in various disciplines of science.

As with all the other chapters, the source code can be found in `src`, in this case in the subdirectory `ode` (the short form ODE is commonly

used as abbreviation for “Ordinary Differential Equations”, which is the type of differential equation that we primarily address in this chapter).

B.1 The Simplest Case

Consider the problem of solving the following equation

$$u'(t) = t^3. \quad (\text{B.1})$$

The solution can be computed directly by integrating (B.1), which gives

$$u(t) = \frac{1}{4}t^4 + C,$$

where C is an arbitrary constant. To obtain a unique solution, we need an extra condition to determine C . Specifying $u(t_1)$ for some time point t_1 represents a possible extra condition. It is common to view (B.1) as an equation for the function $u(t)$ for $t \in [0, T]$, and the extra condition is usually that the start value $u(0)$ is known. This is called the *initial condition*. Say

$$u(0) = 1. \quad (\text{B.2})$$

In general, the solution of the differential equation (B.1) subject to the initial condition B.2 is¹

$$\begin{aligned} u(t) &= u(0) + \int_0^t u'(\tau) d\tau, \\ &= 1 + \int_0^t \tau^3 d\tau \\ &= 1 + \frac{1}{4}t^4. \end{aligned}$$

Let us go back and check: Does $u(t) = 1 + \frac{1}{4}t^4$ really satisfy the two requirements listed in (B.1) and (B.2)? Obviously, $u(0) = 1$, and $u'(t) = t^3$, so the solution is correct.

More generally, we consider the equation

$$u'(t) = f(t) \quad (\text{B.3})$$

together with the initial condition

$$u(0) = u_0. \quad (\text{B.4})$$

Here we assume that $f(t)$ is a given function, and that u_0 is a given number. Then, by reasoning as above, we have

¹ If you are confused by the use of t and τ , don't get too upset; you see: "In mathematics you don't understand things. You just get used to them." –John von Neumann, mathematician, 1903-1957.

$$u(t) = u_0 + \int_0^T f(\tau) d\tau. \quad (\text{B.5})$$

By using the methods introduced in Appendix A, we can find a discrete version of u by approximating the integral. Generally, an approximation of the integral

$$\int_0^T f(\tau) d\tau$$

can be computed using the discrete version of a continuous function $f(\tau)$ defined on an interval $[0, t]$. The discrete version of f is given by $(\tau_i, y_i)_{i=0}^n$ where

$$\tau_i = ih, \text{ and } y_i = f(\tau_i)$$

for $i = 0, 1, \dots, n$. Here $n \geq 1$ is a given integer and $h = T/n$. The Trapezoidal rule can now be written as

$$\int_0^T f(\tau) d\tau \approx \frac{h}{2} \left[y_0 + 2 \sum_{k=1}^{n-1} y_k + y_n \right]. \quad (\text{B.6})$$

By using this approximation, we find that an approximate solution of (B.3)–(B.4) is given by

$$u(t) \approx u_0 + \frac{h}{2} \left[y_0 + 2 \sum_{k=1}^{n-1} y_k + y_n \right].$$

The program `integrate_ode.py` computes a numerical solution of (B.3)–(B.4), where the function f , the time t , the initial condition u_0 , and the number of time-steps n are inputs to the program.

```
def integrate(T, n, u0):
    h = T/float(n)
    t = linspace(0, T, n+1)
    I = f(t[0])
    for k in iseq(1, n-1, 1):
        I += 2*f(t[k])
    I += f(t[-1])
    I *= (h/2)
    I += u0
    return I

from scitools.std import *
f_formula = sys.argv[1]
T = eval(sys.argv[2])
u0 = eval(sys.argv[3])
n = int(sys.argv[4])

f = StringFunction(f_formula, independent_variables='t')
print "Numerical solution of u'(t)=t**3: ", integrate(T, n, u0)
```

We apply the program for computing the solution of

$$u'(t) = te^{t^2},$$

$$u(0) = 0,$$

at time $T = 2$ using $n = 10, 20, 50$ and 100 :

Terminal

```
integrate_ode.py 't*exp(t**2)' 2 0 10
scitools.easyviz backend is matplotlib
Numerical solution of u'(t)=t**3: 28.4066160877
```

Terminal

```
integrate_ode.py 't*exp(t**2)' 2 0 20
scitools.easyviz backend is matplotlib
Numerical solution of u'(t)=t**3: 27.2059977451
```

Terminal

```
integrate_ode.py 't*exp(t**2)' 2 0 50
scitools.easyviz backend is matplotlib
Numerical solution of u'(t)=t**3: 26.86441489
```

Terminal

```
integrate_ode.py 't*exp(t**2)' 2 0 100
scitools.easyviz backend is matplotlib
Numerical solution of u'(t)=t**3: 26.8154183399
```

The exact solution is given by $\frac{1}{2}e^{2^2} - \frac{1}{2} \approx 26.799$, so we see that the approximate solution becomes better as n is increased, as expected.

B.2 Exponential Growth

The example above was really not much of a differential equation, because the solution was obtained by straightforward integration. Equations of the form

$$u'(t) = f(t) \tag{B.7}$$

arise in situations where we can explicitly specify the derivative of the unknown function u . Usually, the derivative is specified in terms of the solution itself. Consider, for instance, population growth under idealized conditions as modeled in Chapter 5.1.4. We introduce the symbol v_i for the number of individuals at time τ_i (v_i corresponds to x_n in Chapter 5.1.4). The basic model for the evolution of v_i is (5.9):

$$v_i = (1 + r)v_{i-1}, \quad i = 1, 2, \dots, \text{ and } v_0 \text{ known.} \tag{B.8}$$

As mentioned in Chapter 5.1.4, r depends on the time difference $\Delta\tau = \tau_i - \tau_{i-1}$: the larger $\Delta\tau$ is, the larger r is. It is therefore natural to

introduce a growth rate α that is independent of $\Delta\tau$: $\alpha = r/\Delta\tau$. The number α is then fixed regardless of how long jumps in time we take in the difference equation for v_i . In fact, α equals the growth in percent, divided by 100, over a time interval of unit length.

The difference equation now reads

$$v_i = v_{i-1} + \alpha\Delta\tau v_{i-1}.$$

Rearranging this equation we get

$$\frac{v_i - v_{i-1}}{\Delta\tau} = \alpha v_{i-1}. \quad (\text{B.9})$$

Assume now that we shrink the time step $\Delta\tau$ to a small value. The left-hand side of (B.9) is then an approximation to the time-derivative of a function $v(\tau)$ expressing the number of individuals in the population at time τ . In the limit $\Delta\tau \rightarrow 0$, the left-hand side becomes the derivative exactly, and the equation reads

$$v'(\tau) = \alpha v(\tau). \quad (\text{B.10})$$

As for the underlying difference equation, we need a start value $v(0) = v_0$. We have seen that reducing the time step in a difference equation to zero, we get a differential equation.

Many like to scale an equation like (B.10) such that all variables are without physical dimensions and their maximum absolute value is typically of the order of unity. In the present model, this means that we introduce new dimensionless variables

$$u = \frac{v}{v_0}, \quad t = \frac{\tau}{\alpha}$$

and derive an equation for $u(t)$. Inserting $v = v_0 u$ and $\tau = \alpha t$ in (B.10) gives the prototype equation for population growth:

$$u'(t) = u(t) \quad (\text{B.11})$$

with the initial condition

$$u(0) = 1. \quad (\text{B.12})$$

When we have computed the dimensionless $u(t)$, we can find the function $v(\tau)$ as

$$v(\tau) = v_0 u(\tau/\alpha).$$

We shall consider practical applications of population growth equations later, but let's start by looking at the idealized case (B.11).

Analytical Solution. Our differential equation can be written in the form

$$\frac{du}{dt} = u,$$

which can be rewritten as

$$\frac{du}{u} = dt,$$

and then integration on both sides yields

$$\ln(u) = t + c,$$

where c is a constant that has to be determined by using the initial condition. Putting $t = 0$, we have

$$\ln(u(0)) = c,$$

hence

$$c = \ln(1) = 0,$$

and then

$$\ln(u) = t,$$

so we have the solution

$$u(t) = e^t. \quad (\text{B.13})$$

Let us now check that this function really solves (B.7, B.11). Obviously, $u(0) = e^0 = 1$, so (B.11) is fine. Furthermore

$$u'(t) = e^t = u(t),$$

thus (B.7) also holds.

Numerical Solution. We have seen that we can find a formula for the solution of the equation of exponential growth. So the problem is solved, and it is trivial to write a program to graph the solution. We will, however, go one step further and develop a numerical solution strategy for this problem. We don't really need such a method for this problem since the solution is available in terms of a formula, but as mentioned earlier, it is good practice to develop methods for problems where we know the solution; then we are more confident when we are confronted with more challenging problems.

Suppose we want to compute a numerical approximation of the solution of

$$u'(t) = u(t) \quad (\text{B.14})$$

equipped with the initial condition

$$u(0) = 1. \quad (\text{B.15})$$

We want to compute approximations from time $t = 0$ to time $t = 1$. Let $n \geq 1$ be a given integer, and define

$$\Delta t = 1/n. \quad (\text{B.16})$$

Furthermore, let u_k denote an approximation of $u(t_k)$ where

$$t_k = k\Delta t \quad (\text{B.17})$$

for $k = 0, 1, \dots, n$. The key step in developing a numerical method for this differential equation is to invoke the Taylor series as applied to the exact solution,

$$u(t_{k+1}) = u(t_k) + \Delta t u'(t_k) + O(\Delta t^2), \quad (\text{B.18})$$

which implies that

$$u'(t_k) \approx \frac{u(t_{k+1}) - u(t_k)}{\Delta t}. \quad (\text{B.19})$$

By using (B.14), we get

$$\frac{u(t_{k+1}) - u(t_k)}{\Delta t} \approx u(t_k). \quad (\text{B.20})$$

Recall now that $u(t_k)$ is the exact solution at time t_k , and that u_k is the approximate solution at the same point in time. We now want to determine u_k for all $k \geq 0$. Obviously, we start by defining

$$u_0 = u(0) = 1.$$

Since we want $u_k \approx u(t_k)$, we require that u_k satisfy the following equality

$$\frac{u_{k+1} - u_k}{\Delta t} = u_k \quad (\text{B.21})$$

motivated by (B.20). It follows that

$$u_{k+1} = (1 + \Delta t)u_k. \quad (\text{B.22})$$

Since u_0 is known, we can compute u_1, u_2 and so on by using the formula above. The formula is implemented² in the program `exp_growth.py`.

```
def compute_u(u0, T, n):
    """Solve u'(t)=u(t), u(0)=u0 for t in [0,T] with n steps."""
    u = u0
    dt = T/float(n)
    for k in range(0, n, 1):
        u = (1+dt)*u
    return u # u(T)
```

² Actually, we do not need the method and we do not need the program. It follows from (B.22) that

$$u_k = (1 + \Delta t)^k u_0$$

for $k = 0, 1, \dots, n$ which can be evaluated on a pocket calculator or even on your cellular phone. But again, we show examples where everything is as simple as possible (but not simpler!) in order to prepare your mind for more complex matters ahead.

```
import sys
n = int(sys.argv[1])

# special test case: u'(t)=u, u(0)=1, t in [0,1]
T = 1; u0 = 1
print 'u(1) =', compute_u(u0, T, n)
```

Observe that we do not store the u values: We just overwrite a float object u by its new value. This saves a lot of storage if n is large.

Running the program for $n = 5, 10, 20$ and 100 , we get the approximations 2.4883, 2.5937, 2.6533, and 2.7048. The exact solution at time $t = 1$ is given by $u(1) = e^1 \approx 2.7183$, so again the approximations become better as n is increased.

An alternative program, where we plot $u(t)$ and therefore store all the u_k and $t_k = k\Delta t$ values, is shown below.

```
def compute_u(u0, T, n):
    """Solve u'(t)=u(t), u(0)=u0 for t in [0,T] with n steps."""
    t = linspace(0, T, n+1)
    t[0] = 0
    u = zeros(n+1)
    u[0] = u0
    dt = T/float(n)
    for k in range(0, n, 1):
        u[k+1] = (1+dt)*u[k]
        t[k+1] = t[k] + dt
    return u, t

from scitools.std import *
n = int(sys.argv[1])

# special test case: u'(t)=u, u(0)=1, t in [0,1]
T = 1; u0 = 1
u, t = compute_u(u0, T, n)
plot(t, u)
tfine = linspace(0, T, 1001) # for accurate plot
v = exp(tfine)                # correct solution
hold('on')
plot(tfine, v)
legend(['Approximate solution', 'Correct function'])
title('Approximate and correct discrete functions, n=%d' % n)
hardcopy('tmp.eps')
```

Using the program for $n = 5, 10, 20$, and 100 , results in the plots in Figure B.1. The convergence towards the exponential function is evident from these plots.

B.3 Logistic Growth

Exponential growth can be modelled by the following equation

$$u'(t) = \alpha u(t)$$

where $\alpha > 0$ is a given constant. If the initial condition is given by

$$u(0) = u_0$$

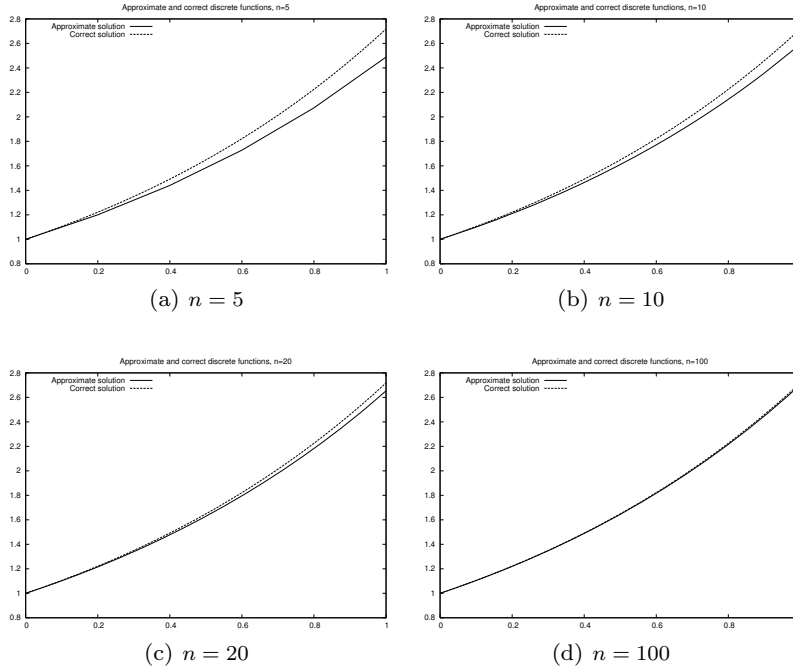


Fig. B.1 Plots of exact and approximate solutions of $u'(t) = u(t)$ with varying number of time steps in $[0, 1]$.

the solution is given by

$$u(t) = u_0 e^{\alpha t}.$$

Since $a > 0$, the solution becomes very large as t increases. For a short time, such growth of a population may be realistic, but over a longer time, the growth of a population is restricted due to limitations of the environment, as discussed in Chapter 5.1.5. Introducing a logistic growth term as in (5.12) we get the differential equation

$$u'(t) = \alpha u(t) \left(1 - \frac{u(t)}{R} \right), \quad (\text{B.23})$$

where α is the growth-rate, and R is the carrying capacity (which corresponds to M in Chapter 5.1.5). Note that R is typically very large, so if $u(0)$ is small, we have

$$\frac{u(t)}{R} \approx 0$$

for small values of t , and thus we have exponential growth for small t ;

$$u'(t) \approx \alpha u(t).$$

But as t increases, and u grows, the term $u(t)/R$ will become important and limit the growth.

A numerical scheme for the logistic equation (B.23) is given by

$$\frac{u_{k+1} - u_k}{\Delta t} = \alpha u_k \left(1 - \frac{u_k}{R}\right),$$

which we can solve with respect to the unknown u_{k+1} :

$$u_{k+1} = u_k + \Delta t \alpha u_k \left(1 - \frac{u_k}{R}\right). \quad (\text{B.24})$$

This is the form of the equation that is suited for implementation.

B.4 A General Ordinary Differential Equation

Let us briefly consider a general ordinary³ differential equations on the form

$$u'(t) = f(u(t)) \quad (\text{B.25})$$

subject to the initial condition

$$u(0) = u_0$$

where $f(u)$ is a given function and the initial state u_0 is given. Suppose we want to compute an approximate solution of (B.25) for t ranging from $t = 0$ to $t = T$ where $T > 0$ is given. As above we start by introducing the time step

$$\Delta t = T/n.$$

where $n \geq 1$ is a given integer, and we let u_k denote an approximation of the exact solution $u(t_k)$. Also as above, we replace the derivative of u with a finite difference and obtain the scheme

$$\frac{u_{k+1} - u_k}{\Delta t} = f(u_k).$$

The scheme can be rewritten in a form that is more suitable for computations;

³ Differential equations are divided into two groups: ordinary differential equations and partial differential equations. Ordinary differential equations contain derivatives with respect to one variable (usually t in our examples), whereas partial differential equations contain derivatives with respect to more than one variable, typically with respect to space and time. A typical ordinary differential equation is

$$u'(t) = u(t),$$

and a typical partial differential equation is

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2},$$

The latter is known as the heat or diffusion equation.

$$u_{k+1} = u_k + \Delta t f(u_k). \quad (\text{B.26})$$

We note that since u_0 is known, we can compute u_1, u_2 and so on. This scheme is commonly referred to as the Explicit Euler⁴ scheme, or the Forward Euler scheme, or the Forward Euler method. We will later derive the Implicit Euler scheme, also called the Backward Euler scheme (or method). That scheme is slightly harder to use but it has some nice properties that will be discussed later.

The Explicit Euler scheme is implemented in the function `Explicit_Euler` in the module `Euler`. The test block in the module file allows input from the command line: the formula for $f(u)$, the number of time steps n , the final time T , and the initial condition u_0 .

```
from numpy import linspace, zeros

def Explicit_Euler(f, u0, T, n):
    dt = T/float(n)
    t = linspace(0, T, n+1)
    u = zeros(n+1)
    u[0] = u0
    for k in range(n):
        u[k+1] = u[k] + dt*f(u[k])
    return u, t

if __name__ == '__main__':
    f_formula = sys.argv[1]
    n = int(sys.argv[2])
    T = eval(sys.argv[3])
    u0 = eval(sys.argv[4])

    f = StringFunction(f_formula, independent_variables='u')
    u, t = Explicit_Euler(f, u0, T, n)
    plot(t, u)
```

In Figure B.2 we see the results of the program as applied to the problem

$$u' = e^u$$

with $u_0 = 0$, $T = 1$. The numerical results are provided for $n = 5, 10, 20$ and 100. Convergence is not as obvious anymore, so let us also try the program for $n = 100, 200, 300$ and 400. The results are given in Figure B.3 and we see that the approximations seem to tend to a common limiting function.

B.5 A Simple Pendulum

So far we have considered scalar ordinary differential equations, i.e., equations with one single function $u(t)$ as unknown. Now we shall deal

⁴ Leonhard Paul Euler, 1707–1783. A pioneering Swiss mathematician and physicist who spent most of his life in Russia and Germany. Euler is one of the greatest scientists of all time, and made important contributions to calculus, mechanics, optics, and astronomy. He also introduced much of the modern terminology and notation in mathematics.

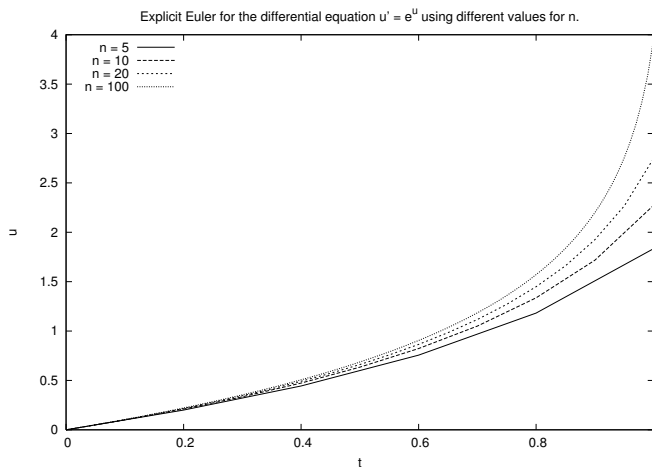


Fig. B.2 Explicit Euler for the differential equation $u' = e^u$ using different values for n .

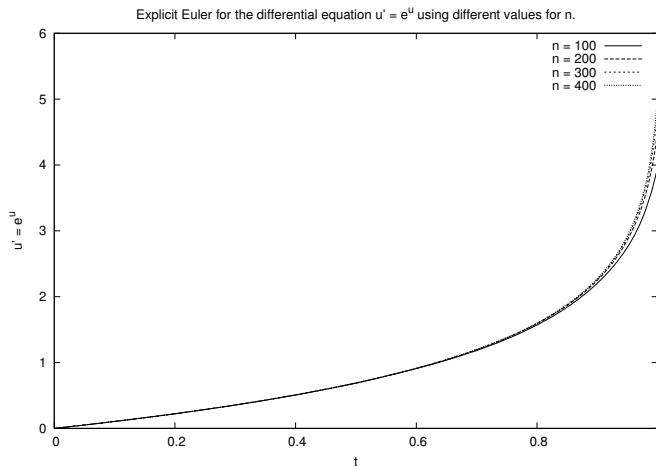


Fig. B.3 Explicit Euler for the differential equation $u' = e^u$ using different values for n .

with systems of ordinary differential equations, where in general n unknown functions are coupled in a system of n equations. Our introductory example will be a system of two equations having two unknown functions $u(t)$ and $v(t)$. The example concerns the motion of a pendulum, see Figure B.4. A sphere with mass m is attached to a massless rod of length L and oscillates back and forth due to gravity. Newton's second law of motion applied to this physical system gives rise the differential equation

$$\theta''(t) + \alpha \sin(\theta) = 0 \quad (\text{B.27})$$

where $\theta = \theta(t)$ is the angle the rod makes with the vertical, measured in radians, and $\alpha = g/L$ (g is the acceleration of gravity). The un-

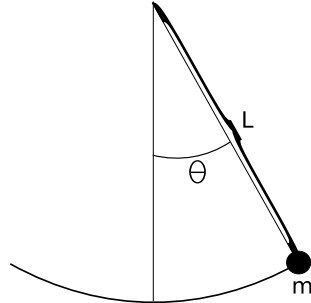


Fig. B.4 A pendulum with m = mass, L = length of massless rod and $\theta = \theta(t)$ = angle.

known function to solve for is θ , and knowing θ , we can quite easily compute the position of the sphere, its velocity, and its acceleration, as well as the tension force in the rod. Since the highest derivative in (B.27) is of second order, we refer to (B.27) as a *second-order differential equations*. Our previous examples in this chapter involved only first-order derivatives, and therefore they are known as *first-order differential equations*.

Equation (B.27) can be solved by the same numerical method as we use in Appendix C.1.2, because (B.27) is very similar to Equation C.8, which is the topic of Appendix C. The only difference is that C.8 has extra terms, which can be skipped, while the kS term in C.8 must be extended to $\alpha \sin(S)$ to make C.8 identical to (B.27). This extension is easily performed. However, here we shall not solve the second-order equation (B.27) as it stands. We shall instead rewrite it as a system of two first-order equations so that we can use numerical methods for first-order equations to solve it.

To transform a second-order equation to a system of two first-order equations, we introduce a new variable for the first-order derivative (the angular velocity of the sphere): $v(t) = \theta'(t)$. Using v and θ in (B.27) yields

$$v'(t) + \alpha \sin(\theta) = 0.$$

In addition, we have the relation

$$v = \theta'(t)$$

between v and θ . This means that (B.27) is equivalent to the following system of two coupled first-order differential equations:

$$\theta'(t) = v(t), \tag{B.28}$$

$$v'(t) = -\alpha \sin(\theta). \tag{B.29}$$

As for scalar differential equations, we need initial conditions, now two conditions because we have two unknown functions:

$$\begin{aligned}\theta(0) &= \theta_0, \\ v(0) &= v_0,\end{aligned}$$

Here we assume the initial angle θ_0 and the initial angular velocity v_0 to be given.

It is common to group the unknowns and the initial conditions in 2-vectors: $(\theta(t), v(t))$ and (θ_0, v_0) . One may then view (B.28)–(B.29) as a *vector equation*, whose first component equation is (B.28), and the second component equation is (B.29). In Python software, this vector notation makes solution methods for scalar equations (almost) immediately available for vector equations, i.e., systems of ordinary differential equations.

In order to derive a numerical method for the system (B.28)–(B.29), we proceed as we did above for one equation with one unknown function. Say we want to compute the solution from $t = 0$ to $t = T$ where $T > 0$ is given. Let $n \geq 1$ be a given integer and define the time step

$$\Delta t = T/n.$$

Furthermore, we let (θ_k, v_k) denote approximations of the exact solution $(\theta(t_k), v(t_k))$ for $k = 0, 1, \dots, n$. A Forward Euler type of method will now read

$$\frac{\theta_{k+1} - \theta_k}{\Delta t} = v_k, \quad (\text{B.30})$$

$$\frac{v_{k+1} - v_k}{\Delta t} = -\alpha \sin(\theta_k). \quad (\text{B.31})$$

This scheme can be rewritten in a form more suitable for implementation:

$$\theta_{k+1} = \theta_k + \Delta t v_k, \quad (\text{B.32})$$

$$v_{k+1} = v_k - \alpha \Delta t \sin(\theta_k). \quad (\text{B.33})$$

The next program, `pendulum.py`, implements this method in the function `pendulum`. The input parameters to the model, θ_0, v_0 , the final time T , and the number of time-steps n , must be given on the command line.

```
def pendulum(T, n, theta0, v0, alpha):
    """Return the motion (theta, v, t) of a pendulum."""
    dt = T/float(n)
    t = linspace(0, T, n+1)
    v = zeros(n+1)
    theta = zeros(n+1)
    v[0] = v0
    theta[0] = theta0
    for k in range(n):
        theta[k+1] = theta[k] + dt*v[k]
        v[k+1] = v[k] - alpha*dt*sin(theta[k+1])
    return theta, v, t
```

```

from scitools.std import *
n = int(sys.argv[1])
T = eval(sys.argv[2])
v0 = eval(sys.argv[3])
theta0 = eval(sys.argv[4])
alpha = eval(sys.argv[5])

theta, v, t = pendulum(T, n, theta0, v0)
plot(t, v, xlabel='t', ylabel='velocity')
figure()
plot(t, theta, xlabel='t', ylabel='velocity')

```

By running the program with the input data $\theta_0 = \pi/6$, $v_0 = 0$, $\alpha = 5$, $T = 10$ and $n = 1000$, we get the results shown in Figure B.5. The angle $\theta = \theta(t)$ is displayed in the left panel and the velocity is given in the right panel.

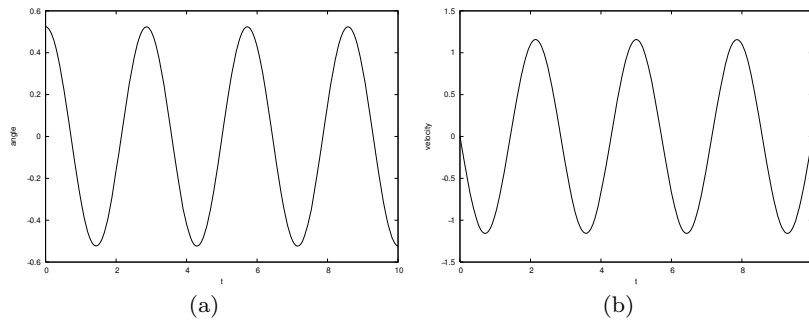


Fig. B.5 Motion of a pendulum: (a) the angle $\theta(t)$, and (b) the angular velocity θ' .

B.6 A Model for the Spread of a Disease

Mathematical models are used intensively to analyze the spread of infectious diseases⁵. In the simplest case, we may consider a population, that is supposed to be constant, consisting of two groups; the susceptibles (S) who can catch the disease, and the infectives (I) who have the disease and are able to transmit it. A system of differential equations modelling the evolution of S and I is given by

$$\begin{aligned}
 S' &= -rSI, \\
 I' &= rSI - aI.
 \end{aligned}$$

Here r and a are given constants reflecting the characteristics of the epidemic. The initial conditions are given by

⁵ The interested reader may consult the excellent book [10] on Mathematical Biology by J.D. Murray for an introduction to such models.

$$\begin{aligned} S(0) &= S_0, \\ I(0) &= I_0, \end{aligned}$$

where the initial state (S_0, I_0) is assumed to be known.

Suppose we want to compute numerical solutions of this system from time $t = 0$ to $t = T$. Then, by reasoning as above, we introduce the time step

$$\Delta t = T/n$$

and the approximations (S_k, I_k) of the solution $(S(t_k), I(t_k))$. An explicit Forward Euler method for the system takes the following form,

$$\begin{aligned} \frac{S_{k+1} - S_k}{\Delta t} &= -rS_kI_k, \\ \frac{I_{k+1} - I_k}{\Delta t} &= rS_kI_k - aI_k, \end{aligned}$$

which can be rewritten on computational form

$$\begin{aligned} S_{k+1} &= S_k - \Delta t r S_k I_k, \\ I_{k+1} &= I_k + \Delta t (r S_k I_k - a I_k). \end{aligned}$$

This scheme is implemented in the program `exp_epidemic.py` where r, a, S_0, I_0, n and T are input data given on the command line. The function `epidemic` computes the solution (S, I) to the differential equation system. This pair of time-dependent functions is then plotted in two separate plots.

```
def epidemic(T, n, S0, I0, r, a):
    dt = T/float(n)
    t = linspace(0, T, n+1)
    S = zeros(n+1)
    I = zeros(n+1)
    S[0] = S0
    I[0] = I0
    for k in range(n):
        S[k+1] = S[k] - dt*r*S[k]*I[k]
        I[k+1] = I[k] + dt*(r*S[k]*I[k] - a*I[k])
    return S, I, t

from scitools.std import *
n = int(sys.argv[1])
T = eval(sys.argv[2])
S0 = eval(sys.argv[3])
I0 = eval(sys.argv[4])
r = eval(sys.argv[5])
a = eval(sys.argv[6])

plot(t, S, xlabel='t', ylabel='Susceptibles')
plot(t, I, xlabel='t', ylabel='Infectives')
```

We want to apply the program to a specific case where an influenza epidemic hit a British boarding school with a total of 763 boys⁶. The

⁶ The data are from Murray [10], and Murray found the data in the British Medical Journal, March 4, 1978.

epidemic lasted from 21st January to 4th February in 1978. We let $t = 0$ denote 21st of January and we define $T = 14$ days. We put $S_0 = 762$ and $I_0 = 1$ which means that one person was ill at $t = 0$. In the Figure B.6 we see the numerical results using $r = 2.18 \times 10^{-3}$, $a = 0.44$, $n = 1000$. Also, we have plotted actual the measurements, and we note that the simulations fit the real data quite well.

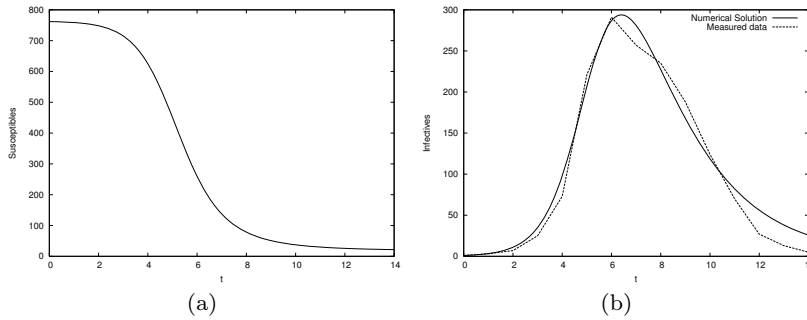


Fig. B.6 Graphs of (a) susceptibles and (b) infectives for an influenza in a British boarding school in 1978.

B.7 Exercises

Exercise B.1. *Solve a nonhomogeneous linear ODE.*

Solve the ODE problem

$$u' = 2u - 1, \quad u(0) = 2, \quad t \in [0, 6]$$

using the Forward Euler method. Choose $\Delta t = 0.25$. Plot the numerical solution together with the exact solution $u(t) = \frac{1}{2} + 2e^{2t}$. Name of program file: `nonhomogeneous_linear_ODE.py`. \diamond

Exercise B.2. *Solve a nonlinear ODE.*

Solve the ODE problem

$$u' = u^q, \quad u(0) = 1, \quad t \in [0, T]$$

using the Forward Euler method. The exact solution reads $u(t) = e^t$ for $q = 1$ and $u(t) = (t(1 - q) + 1)^{1/(1-q)}$ for $q > 1$ and $t(1 - q) + 1 > 0$. Read q , Δt , and T from the command line, solve the ODE, and plot the numerical and exact solution. Run the program for different cases: $q = 2$ and $q = 3$, with $\Delta t = 0.01$ and $\Delta t = 0.1$. Set $T = 6$ if $q = 1$ and $T = 1/(q - 1) - 0.1$ otherwise. Name of program file: `nonlinear_ODE.py`.

\diamond

Exercise B.3. *Solve an ODE for $y(x)$.*

We have given the following ODE problem:

$$\frac{dy}{dx} = \frac{1}{2(y-1)}, \quad y(0) = 1 + \sqrt{\epsilon}, \quad x \in [0, 4], \quad (\text{B.34})$$

where $\epsilon > 0$ is a small number. Formulate a Forward Euler method for this ODE problem and compute the solution for varying step size in x : $\Delta x = 1$, $\Delta x = 0.25$, $\Delta x = 0.01$. Plot the numerical solutions together with the exact solution $y(x) = 1 + \sqrt{x + \epsilon}$, using 1001 x coordinates for accurate resolution of the latter. Set ϵ to 10^{-3} . Study the numerical solution with $\Delta x = 1$, and use that insight to explain why this problem is hard to solve numerically. Name of program file: `yx_ODE.py`. \diamond

Exercise B.4. *Experience instability of an ODE.*

Consider the ODE problem

$$u' = \alpha u, \quad u(0) = u_0,$$

solved by the Forward Euler method. Show by repeatedly applying the scheme that

$$u_k = (1 + \alpha \Delta t)^k u_0.$$

We now turn to the case $\alpha < 0$. Show that the numerical solution will oscillate if $\Delta t > -1/\alpha$. Make a program for computing u_k , set $\alpha = -1$, and demonstrate oscillatory solutions for $\Delta t = 1.1, 1.5, 1.9$. Recall that the exact solution, $u(t) = e^{\alpha t}$, never oscillates.

What happens if $\Delta t > -2/\alpha$? Try it out in the program and explain then mathematically why not $u_k \rightarrow 0$ as $k \rightarrow \infty$. Name of program file: `unstable_ODE.py`. \diamond

Exercise B.5. *Solve an ODE for the arc length.*

Given a curve $y = f(x)$, the length of the curve from $x = x_0$ to some point x is given by the function $s(x)$, which fulfills the problem

$$\frac{ds}{dx} = \sqrt{1 + [f'(x)]^2}, \quad s(x_0) = 0. \quad (\text{B.35})$$

Since s does not enter the right-hand side, (B.35) can immediately be integrated from x_0 to x . However, we shall solve (B.35) as an ODE. Use the Forward Euler method and compute the length of a straight line (for verification) and a sine curve: $f(x) = \frac{1}{2}x + 1$, $x \in [0, 2]$; $f(x) = \sin(\pi x)$, $x \in [0, 2]$. Name of program file: `arclength_ODE.py`. \diamond

Exercise B.6. *Solve an ODE with time-varying growth.*

Consider the ODE for exponential growth,

$$u' = \alpha u, \quad u(0) = 1, \quad t \in [0, T].$$

Now we introduce a time-dependent α such that the growth decreases with time: $\alpha(t) = a - bt$. Solve the problem for $a = 1$, $b = 0.1$, and $T = 10$. Plot the solution and compare with the corresponding exponential growth using the mean value of $\alpha(t)$ as growth factor: $e^{(a-bT/2)t}$. Name of program file: `time_dep_growth.py`. \diamond

Exercise B.7. *Solve an ODE for emptying a tank.*

A cylindrical tank of radius R is filled with water to a height $h(t)$. By opening a valve of radius r at the bottom of the tank, water flows out, and $h(t)$ decreases with time. We can derive an ODE that governs the height function $h(t)$.

Mass conservation of water requires that the reduction in height balances the outflow. In a time interval Δt , the height is reduced by Δh , which corresponds to a water volume of $\pi R^2 \Delta h$. The water leaving the tank in the same interval of time equals $\pi r^2 v \Delta t$, where v is the outflow velocity. It can be shown (from what is known as Bernoulli's equation) that

$$v(t) = \sqrt{2gh(t) - h'(t)^2},$$

g being the acceleration of gravity [6, 11]. Letting $\Delta h > 0$ correspond to an increase in h , we have that the $-\pi R^2 \Delta h$ must balance $\pi r^2 v \Delta t$, which in the limit $\Delta t \rightarrow 0$ leads to the ODE

$$\frac{dh}{dt} = -\left(\frac{r}{R}\right)^2 \left(1 + \left(\frac{r}{R}\right)^4\right)^{-1/2} \sqrt{2gh}. \quad (\text{B.36})$$

A proper initial condition follows from the initial height of water, h_0 , in the tank: $h(0) = h_0$.

Solve (B.36) in a program using the Forward Euler scheme. Set $r = 1$ cm, $R = 20$ cm, $g = 9.81$ m/s², and $h_0 = 1$ m. Use a time step of 10 seconds. Plot the solution, and experiment to see what a proper time interval for the simulation is. Can you find an analytical solution of the problem to compare the numerical solution with? Name of program file: `tank_ODE.py`. \diamond

Exercise B.8. *Solve an ODE system for an electric circuit.*

An electric circuit with a resistor, a capacitor, an inductor, and a voltage source can be described by the ODE

$$L \frac{dI}{dt} + RI + \frac{Q}{C} = E(t), \quad (\text{B.37})$$

where LdI/dt is the voltage drop across the inductor, I is the current (measured in amperes, A), L is the inductance (measured in henrys, H), R is the resistance (measured in ohms, Ω), Q is the charge on the capacitor (measured in coulombs, C), C is the capacitance (measured in farads, F), $E(t)$ is the time-variable voltage source (measured in volts, V), and t is time (measured in seconds, s). There is a relation

between I and Q :

$$\frac{dQ}{dt} = I. \quad (\text{B.38})$$

Equations (B.37)–(B.38) is a system two ODEs. Solve these for $L = 1$ H, $E(t) = 2 \sin \omega t$ V, $\omega^2 = 3.5 \text{ s}^{-2}$, $C = 0.25$ C, $R = 0.2 \Omega$, $I(0) = 1$ A, and $Q(0) = 1C$. Use the Forward Euler scheme with $\Delta t = 2\pi/(60\omega)$. The solution will, after some time, oscillate with the same period as $E(t)$, a period of $2\pi/\omega$. Simulate 10 periods. (Actually, it turns out that the Forward Euler scheme overestimates the amplitudes of the oscillations. Exercise 9.33 compares the Forward Euler scheme with the more accurate 4th-order Runge-Kutta method.) Name of program file: `electric_circuit.py`. \diamond

The examples in the ordinary chapters of this book are quite compact and composed to convey programming constructs in a gentle pedagogical way. In this appendix the idea is to solve a more comprehensive real-world problem by programming. The problem solving process gets quite advanced because we bring together elements from physics, mathematics, and programming, in a way that a scientific programmer must master. Each individual element is quite forward in the sense that you have probably met the element already, either in high school physics or mathematics, or in this book. The challenge is to understand the problem, and analyze it by breaking it into a set of simpler elements. It is not necessary to understand this problem solving process in detail. As a computer programmer, all you need to understand is how you translate the given algorithm into a working program and how to test the program. We anticipate that this task should be doable without a thorough understanding of the physics and mathematics of the problem.

You can read the present appendix after the material in the first four chapters are digested. More specifically, you can read Appendices C.1 and C.2 after Chapter 3, while Appendix C.3 requires knowledge about curve plotting from Chapter 4.

Appendix C.1–C.2 can be read after the first three chapters of the book is digested. Appendix C.3 requires the plotting knowledge of Chapter 4.

All Python files associated with this appendix are found in `src/box_spring`.

C.1 About the Problem: Motion and Forces in Physics

C.1.1 The Physical Problem

We shall study a simple device which models oscillating systems. A box with mass m and height b is attached to a spring of length L as shown in Figure C.1. The end of the spring is attached to a plate which

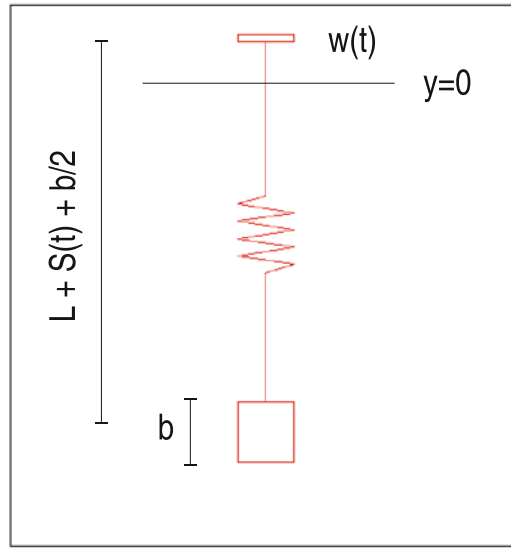


Fig. C.1 An oscillating system with a box attached to a spring.

we can move up and down with a displacement $w(t)$, where t denotes time. There are two ways the box can be set in motion: we can either stretch or compress the string initially by moving the box up or down, or we can move the plate. If $w = 0$ the box oscillates freely, otherwise we have what is called driven oscillations.

Why will such a system oscillate? When the box moves downward, the spring is stretched, which results in a force that tries to move the box upward. The more we stretch the spring, the bigger the force against the movement becomes. The box eventually stops and starts moving upward with an upward acceleration. At some point the spring is not stretched anymore and there is no spring force on the box, but because of inertia, the box continues its motion upward. This causes the spring to get compressed, causing a force from the spring on the box that acts downward, against the upward movement. The downward force increases in intensity and manages to stop the upward motion. The process repeats itself and results in an oscillatory motion of the box. Since the spring tries to restore the position of the box, we refer to the spring force as a *restoring force*.

You have probably experienced that oscillations in such springs tend to die out with time. There is always a *damping force* that works

against the motion. This damping force may be due to a not perfectly elastic string, and the force can be quite small, but we can also explicitly attach the spring to a damping mechanism to obtain a stronger, controllable damping of the oscillations (as one wants in a car or a mountain bike). We will assume that there is some damping force present in our system, and this can well be a damping mechanism although this is not explicitly included in Figure C.1.

Oscillating systems of the type depicted in Figure C.1 have a huge number of applications throughout science and technology. One simple example is the spring system in a car or bicycle, which you have probably experienced on a bumpy road (the bumps lead to a $w(t)$ function). When your washing machine jumps up and down, it acts as a highly damped oscillating system (and the $w(t)$ function is related to uneven distribution of the mass of the clothes). The pendulum in a wall clock is another oscillating system, not with a spring, but physically the system can (for small oscillations) be modeled as a box attached to a spring because gravity makes a spring-like force on a pendulum (in this case, $w(t) = 0$). Other examples on oscillating systems where this type of equation arise are briefly mentioned in Exercise 9.45. The bottom line is that understanding the dynamics of Figure C.1 is the starting point for understanding the behavior of a wide range of oscillating phenomena in nature and technical devices.

Goal of the Computations. Our aim is to compute the position of the box as a function of time. If we know the position, we can compute the velocity, the acceleration, the spring force, and the damping force. The mathematically difficult thing is to calculate the position – everything else is much easier¹.

We assume that the box moves in the vertical direction only, so we introduce $Y(t)$ as the vertical position of the center point of the box. We shall derive a mathematical equation that has $Y(t)$ as solution. This equation can be solved by an algorithm which can be implemented in a program. Our focus is on the implementation, since this is a book about programming, but for the reader interested in how computers play together with physics and mathematics in science and technology, we also outline how the equation and algorithm arise.

The Key Quantities. Let S be the stretch of the spring, where $S > 0$ means stretch and $S < 0$ implies compression. The length of the spring when it is unstretched is L , so at a given point of time t the actual length is $L + S(t)$. Given the position of the plate, $w(t)$, the length of

¹ More precisely, to compute the position we must solve a differential equation while the other quantities can be computed by differentiation and simple arithmetics. Solving differential equations is historically considered very difficult, but computers have simplified this task dramatically. Appendix B and Chapters 7.4 and 9.4 are devoted to this topic.

the spring, $L + S(t)$, and the height of the box, b , the position $Y(t)$ is then, according to Figure C.1,

$$Y(t) = w(t) - (L + S(t)) - \frac{b}{2}. \quad (\text{C.1})$$

You can think as follows: We first “go up” to the plate at $y = w(t)$, then down $L + S(t)$ along the spring and then down $b/2$ to the center of the box. While L , w , and b must be known as input data, $S(t)$ is unknown and will be output data from the program.

C.1.2 The Computational Algorithm

Let us now go straight to the programming target and present the recipe for computing $Y(t)$. The algorithm below actually computes $S(t)$, but at any point of time we can easily find $Y(t)$ from (C.1) if we know $S(t)$. The $S(t)$ function is computed at discrete points of time, $t = t_i = i\Delta t$, for $i = 0, 1, \dots, N$. We introduce the notation S_i for $S(t_i)$. The S_i values can be computed by the following algorithm.

1. Set initial stretch S_0 from input data
2. Compute S_1 by

$$S_{i+1} = \frac{1}{2m} (2mS_i - \Delta t^2 k S_i + m(w_{i+1} - 2w_i + w_{i-1}) + \Delta t^2 mg), \quad (\text{C.2})$$

with $i = 0$.

3. For $i = 1, 2, \dots, N - 1$, compute S_{i+1} by

$$S_{i+1} = (m + \gamma)^{-1} (2mS_i - mS_{i-1} + \gamma\Delta t S_{i-1} - \Delta t^2 k S_i + m(w_{i+1} - 2w_i + w_{i-1}) + \Delta t^2 mg). \quad (\text{C.3})$$

The parameter γ equals $\frac{1}{2}\beta\Delta t$. The input data to the algorithm are the mass of the box m , a coefficient k characterizing the spring, a coefficient β characterizing the amount of damping in the system, the acceleration of gravity g , the movement of the plate $w(t)$, the initial stretch of the spring S_0 , the number of time steps N , and the time Δt between each computation of S values. The smaller we choose Δt , the more accurate the computations become.

Now you have two options, either read the derivation of this algorithm in Appendix C.1.3–C.1.4 or jump right to implementation in Appendix C.2.

C.1.3 Derivation of the Mathematical Model

To derive the algorithm we need to make a mathematical model of the oscillating system. This model is based on physical laws. The most

important physical law for a moving body is Newton's second law of motion:

$$F = ma, \quad (\text{C.4})$$

where F is the sum of all forces on the body, m is the mass of the body, and a is the acceleration of the body. The body here is our box.

Let us first find all the forces on the box. Gravity acts downward with magnitude mg . We introduce $F_g = -mg$ as the gravity force, with a minus sign because a negative force acts downward, in negative y direction.

The spring force on the box acts upward if the spring is stretched, i.e., if $S > 0$ we have a positive spring force F_s . The size of the force is proportional to the amount of stretching, so we write² $F_s = kS$, where k is commonly known as the *spring constant*. We also assume that we have a damping force that is always directed toward the motion and proportional with the “velocity of the stretch”, $-dS/dt$. Naming the proportionality constant β , we can write the damping force as $F_d = \beta dS/dt$. Note that when $dS/dt > 0$, S increases in time and the box moves downward, the F_d force then acts upward, against the motion, and must be positive. This is the way we can check that the damping force expression has the right sign.

The sum of all forces is now

$$\begin{aligned} F &= F_g + F_s + F_d, \\ &= -mg + kS + \beta \frac{dS}{dt}. \end{aligned} \quad (\text{C.5})$$

We now know the left-hand side of (C.4), but S is unknown to us. The acceleration a on the right-hand side of (C.4) is also unknown. However, acceleration is related to movement and the S quantity, and through this relation we can eliminate a as a second unknown. From physics, it is known that the acceleration of a body is the second derivative in time of the position of the body, so in our case,

$$\begin{aligned} a &= \frac{d^2 Y}{dt^2}, \\ &= \frac{d^2 w}{dt^2} - \frac{d^2 S}{dt^2}, \end{aligned} \quad (\text{C.6})$$

(remember that L and b are constant).

Equation (C.4) now reads

$$-mg + kS + \beta \frac{dS}{dt} = m \left(\frac{d^2 w}{dt^2} - \frac{d^2 S}{dt^2} \right). \quad (\text{C.7})$$

² Spring forces are often written in the canonical form “ $F = -kx$ ”, where x is the stretch. The reason that we have no minus sign is that our stretch S is positive in the downward (negative) direction.

It is common to collect the unknown terms on the left-hand side and the known quantities on the right-hand side, and let higher-order derivatives appear before lower-order derivatives. With such a reordering of terms we get

$$m \frac{d^2 S}{dt^2} + \beta \frac{dS}{dt} + kS = m \frac{d^2 w}{dt^2} + mg. \quad (\text{C.8})$$

This is the equation governing our physical system. If we solve the equation for $S(t)$, we have the position of the box according to (C.1), the velocity v as

$$v(t) = \frac{dY}{dt} = \frac{dw}{dt} - \frac{dS}{dt}, \quad (\text{C.9})$$

the acceleration as (C.6), and the various forces can be easily obtained from the formulas in (C.5).

A key question is if we can solve (C.8). If $w = 0$, there is in fact a well-known solution which can be written

$$S(t) = \frac{m}{k}g + \begin{cases} e^{-\zeta t} \left(c_1 e^{t\sqrt{\beta^2-1}} + c_2 e^{-t\sqrt{\zeta^2-1}} \right), & \zeta > 1, \\ e^{-\zeta t} (c_1 + c_2 t), & \zeta = 1, \\ e^{-\zeta t} \left[c_1 \cos \left(\sqrt{1-\zeta^2} t \right) + c_2 \sin \left(\sqrt{1-\zeta^2} t \right) \right], & \zeta < 1. \end{cases} \quad (\text{C.10})$$

Here, ζ is a short form for $\beta/2$, and c_1 and c_2 are arbitrary constants. That is, the solution (C.10) is not unique.

To make the solution unique, we must determine c_1 and c_2 . This is done by specifying the state of the system at some point of time, say $t = 0$. In the present type of mathematical problem we must specify S and dS/dt . We allow the spring to be stretched an amount S_0 at $t = 0$. Moreover, we assume that there is no ongoing increase or decrease in the stretch at $t = 0$, which means that $dS/dt = 0$. In view of (C.9), this condition implies that the velocity of the box is that of the plate, and if the latter is at rest, the box is also at rest initially. The conditions at $t = 0$ are called *initial conditions*:

$$S(0) = S_0, \quad \frac{dS}{dt}(0) = 0. \quad (\text{C.11})$$

These two conditions provide two equations for the two unknown constants c_1 and c_2 . Without the initial conditions two things happen: (i) there are infinitely many solutions to the problem, and (ii) the computational algorithm in a program cannot start.

Also when $w \neq 0$ one can find solutions $S(t)$ of (C.8) in terms of mathematical expressions, but only for some very specific choices of $w(t)$ functions. With a program we can compute the solution $S(t)$ for any “reasonable” $w(t)$ by a quite simple method. The method gives only an approximate solution, but the approximation can usually be

made as good as desired. This powerful solution method is described below.

C.1.4 Derivation of the Algorithm

To solve (C.8) on a computer, we do two things:

1. We calculate the solution at some discrete time points $t = t_i = i\Delta t$, $i = 0, 1, 2, \dots, N$.
2. We replace the derivatives by finite differences, which are approximate expressions for the derivatives.

The first and second derivatives can be approximated by³

$$\frac{dS}{dt}(t_i) \approx \frac{S(t_{i+1}) - S(t_{i-1}))}{2\Delta t}, \quad (\text{C.12})$$

$$\frac{d^2S}{dt^2}(t_i) \approx \frac{S(t_{i+1}) - 2S(t_i) + S(t_{i-1}))}{\Delta t^2}. \quad (\text{C.13})$$

It is common to save some writing by introducing S_i as a short form for $S(t_i)$. The formulas then read

$$\frac{dS}{dt}(t_i) \approx \frac{S_{i+1} - S_{i-1}}{2\Delta t}, \quad (\text{C.14})$$

$$\frac{d^2S}{dt^2}(t_i) \approx \frac{S_{i+1} - 2S_i + S_{i-1}}{\Delta t^2}. \quad (\text{C.15})$$

Let (C.8) be valid at a point of time t_i :

$$m \frac{d^2S}{dt^2}(t_i) + \beta \frac{dS}{dt}(t_i) + kS(t_i) = m \frac{d^2w}{dt^2}(t_i) + mg. \quad (\text{C.16})$$

We now insert (C.14) and (C.15) in (C.16) (observe that we can approximate d^2w/dt^2 in the same way as we approximate d^2S/dt^2):

$$m \frac{S_{i+1} - 2S_i + S_{i-1}}{\Delta t^2} + \beta \frac{S_{i+1} - S_{i-1}}{2\Delta t} + kS_i = m \frac{w_{i+1} - 2w_i + w_{i-1}}{\Delta t^2} + mg. \quad (\text{C.17})$$

The computational algorithm starts with knowing S_0 , then S_1 is computed, then S_2 , and so on. Therefore, in (C.17) we can assume that S_i and S_{i-1} are already computed, and that S_{i+1} is the new unknown to calculate. Let us as usual put the unknown terms on the left-hand side (and multiply by Δt^2):

$$mS_{i+1} + \gamma S_{i+1} = 2mS_i - mS_{i-1} + \gamma S_{i-1} - \Delta t^2 kS_i + m(w_{i+1} - 2w_i + w_{i-1}) + \Delta t^2 mg, \quad (\text{C.18})$$

³ See Appendices A and B for derivations of such formulas.

where we have introduced the short form $\gamma = \frac{1}{2}\beta\Delta t$ to save space. Equation (C.18) can easily be solved for S_{i+1} :

$$S_{i+1} = (m - \gamma)^{-1} (2mS_i - mS_{i-1} + \gamma\Delta t S_{i-1} - \Delta t^2 kS_i + m(w_{i-1} - 2w_i + w_{i+1}) + \Delta t^2 mg) \quad (\text{C.19})$$

One fundamental problem arises when we try to start the computations. We know S_0 and want to apply (C.19) for $i = 0$ to calculate S_1 . However, (C.19) involves S_{i-1} , that is, S_{-1} , which is an unknown value at a point of time *before* we compute the motion. The initial conditions come to rescue here. Since $dS/dt = 0$ at $t = 0$ (or $i = 0$), we can approximate this condition as

$$\frac{S_1 - S_{-1}}{2\Delta t} = 0 \quad : \quad S_{-1} = S_1. \quad (\text{C.20})$$

Inserting this relation in (C.19) when $i = 0$ gives a special formula for S_1 (or S_{i+1} with $i = 0$, if we want):

$$S_{i+1} = \frac{1}{2m} (2mS_i - \Delta t^2 kS_i + m(w_{i+1} - 2w_i + w_{i-1}) + \Delta t^2 mg). \quad (\text{C.21})$$

Remember that $i = 0$ in this formula. The overall algorithm is summarized below:

1. Initialize S_0 from initial condition
2. Use (C.21) to compute S_{i+1} for $i = 0$
3. For $i = 0, 1, 2, \dots, N - 1$, use (C.19) to compute S_{i+1}

C.2 Program Development and Testing

C.2.1 Implementation

The aim now is to implement the algorithm on page 628 in a Python program. There are naturally two parts of the program, one where we read input data such as L , m , and $w(t)$, and one part where we run the computational algorithm. Let us write a function for each part.

The set of input data to the program consists of the mathematical symbols

- m (the mass of the box)
- b (the height of the box)
- L (the length of the unstretched spring)
- β (coefficient for the damping force)
- k (coefficient for the spring force)
- Δt (the time step between each S_i calculation)
- N (the number of computed time steps)
- S_0 (the initial stretch of the spring)

- $w(t)$ (the vertical displacement of the plate)
- g (acceleration of gravity)

We make a function `init_prms` for initializing these input parameters from option-value pairs on the command line. That is, the user provides pairs like `-m 2` and `-dt 0.1` (for Δt). The `getopt` module from Chapter 3.2.4 can be used for this purpose. We supply default values for all parameters as arguments to the `init_prms` function. The function returns all these parameters with the changes that the user has specified on the command line. The w parameter is given as a string expression (called `w_formula` below), and the `StringFunction` tool from Chapter 3.1.4 can be used to turn the formula into a working Python function. An algorithmic sketch of the tasks in the `init_prms` function can be expressed by some pseudo Python code:

```
def init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N):
    import getopt, sys
    try:
        options, args = getopt.getopt(
            sys.argv[1:], '',
            <list of legal command-line options>)
    except getopt.GetoptError, e:
        <handle error in command-line options>

    for option, value in options:
        if option in ('--m', '--mass'):
            m = float(value)
        elif option in ('--b', '--boxheight'):
            b = float(value)
        elif ... # treat all other parameters

    from scitools.StringFunction import StringFunction
    w = StringFunction(w_formula, independent_variables='t')
    return m, b, L, k, beta, S0, dt, g, w, N
```

With such a sketch as a start, we can complete the indicated code and arrive at a working function for specifying input parameters to the mathematical model:

```
def init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N):
    import getopt, sys
    try:
        options, args = getopt.getopt(
            sys.argv[1:], '',
            ['m=', 'mass=',
            'b=', 'boxheight=',
            'L=', 'spring-length=',
            'k=', 'spring-stiffness=',
            'beta=', 'spring-damping=',
            'S0=', 'initial-position=',
            'dt=', 'timestep=',
            'g=', 'gravity=',
            'w=',
            'N='])
    except getopt.GetoptError, e:
        print 'Error in command-line option:\n', e
        sys.exit(1)

    for option, value in options:
        if option in ('--m', '--mass'):
```

```

        m = float(value)
    elif option in ('--b', '--boxheight'):
        b = float(value)
    elif option in ('--L', '--spring-length'):
        L = float(value)
    elif option in ('--k', '--spring-stiffness'):
        k = float(value)
    elif option in ('--beta', '--spring-damping'):
        beta = float(value)
    elif option in ('--S0', '--initial-position'):
        S0 = float(value)
    elif option in ('--dt', '--timestep'):
        dt = float(value)
    elif option in ('--g', '--gravity'):
        g = float(value)
    elif option in ('--w',):
        w_formula = value # string
    elif option == '--N':
        N = int(value)

from scitools.StringFunction import StringFunction
w = StringFunction(w_formula, independent_variables='t')
return m, b, L, k, beta, S0, dt, g, w, N

```

You may wonder why we specify g (gravity) since this is a known constant, but it is useful to turn off the gravity force to test the program. Just imagine the oscillations take place in the horizontal direction – the mathematical model is the same, but $F_g = 0$, which we can obtain in our program by setting the input parameter g to zero.

The computational algorithm is quite easy to implement, as there is a quite direct translation of the mathematical algorithm in Appendix C.1.2 to valid Python code. The S_i values can be stored in a list or array with indices going from 0 to N . To allow readers to follow the code here without yet having digested Chapter 4, we use a plain list. The function for computing S_i reads

```

def solve(m, k, beta, S0, dt, g, w, N):
    S = [0.0]*(N+1) # output list
    gamma = beta*dt/2.0 # short form
    t = 0
    S[0] = S0
    # special formula for first time step:
    i = 0
    S[i+1] = (1/(2.0*m))*(2*m*S[i] - dt**2*k*S[i] +
        m*(w(t+dt) - 2*w(t) + w(t-dt)) + dt**2*m*g)
    t = dt

    for i in range(1,N):
        S[i+1] = (1/(m + gamma))*(2*m*S[i] - m*S[i-1] +
            gamma*dt*S[i-1] - dt**2*k*S[i] +
            m*(w(t+dt) - 2*w(t) + w(t-dt))
            + dt**2*m*g)

        t += dt
    return S

```

The primary challenge in coding the algorithm is to set the index t and the time t right. Recall that in the updating formula for $S[i+1]$ at time $t+dt$, the time on the right-hand side shall be the time at time

step i , so the $t+=dt$ update must come after $S[i+1]$ is computed. The same is important in the special formula for the first time step as well.

A main program will typically first set some default values of the 10 input parameters, then call `init_prms` to let the user adjust the default values, and then call `solve` to compute the S_i values:

```
# default values:
from math import pi
m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 1;
dt = 2*pi/40; g = 9.81; w_formula = '0'; N = 80;

m, b, L, k, beta, S0, dt, g, w, N = \
    init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N)
S = solve(m, k, beta, S0, dt, g, w, N)
```

So, what shall we do with the solution S ? We can write out the values of this list, but the numbers do not give an immediate feeling for how the box moves. It will be better to graphically illustrate the $S(t)$ function, or even better, the $Y(t)$ function. This is straightforward with the techniques from Chapter 4 and is treated in Appendix C.3. In Chapter 9.5, we develop a drawing tool for drawing figures like Figure C.1. By drawing the box, string, and plate at every time level we compute S_i , we can use this tool to make a moving figure that illustrates the dynamics of the oscillations. Already now you can play around with a program doing that (`box_spring_figure_anim.py`).

C.2.2 Callback Functionality

It would be nice to make some graphics of the system while the computations take place, not only after the S list is ready. The user must then put some relevant statements in between the statements in the algorithm. However, such modifications will depend on what type of analysis the user wants to do. It is a bad idea to mix user-specific statements with general statements in a general algorithm. We therefore let the user provide a function that the algorithm can call after each S_i value is computed. This is commonly called a *callback* function (because a general function calls back to the user's program to do a user-specific task). To this callback function we send three key quantities: the S list, the point of time (t), and the time step number ($i + 1$), so that the user's code gets access to these important data.

If we just want to print the solution to the screen, the callback function can be as simple as

```
def print_S(S, t, step):
    print 't=%.2f S[%d]=%+g' % (t, step, S[step])
```

In the `solve` function we take the callback function as a keyword argument `user_action`. The default value can be an empty function, which we can define separately:

```
def empty_func(S, time, time_step_no):
    return None

def solve(m, k, beta, S0, dt, g, w, N,
         user_action=empty_func):
    ...
```

However, it is quicker to just use a lambda function (see Chapter 2.2.11):

```
def solve(m, k, beta, S0, dt, g, w, N,
         user_action=lambda S, time, time_step_no: None):
```

The new `solve` function has a call to `user_action` each time a new `S` value has been computed:

```
def solve(m, k, beta, S0, dt, g, w, N,
         user_action=lambda S, time, time_step_no: None):
    """Calculate N steps forward. Return list S."""
    S = [0.0]*(N+1) # output list
    gamma = beta*dt/2.0 # short form
    t = 0
    S[0] = S0
    user_action(S, t, 0)
    # special formula for first time step:
    i = 0
    S[i+1] = (1/(2.0*m))*(2*m*S[i] - dt**2*k*S[i] +
                        m*(w(t+dt) - 2*w(t) + w(t-dt))) + dt**2*m*g)
    t = dt
    user_action(S, t, i+1)

    # time loop:
    for i in range(1,N):
        S[i+1] = (1/(m + gamma))*(2*m*S[i] - m*S[i-1] +
                                gamma*dt*S[i-1] - dt**2*k*S[i] +
                                m*(w(t+dt) - 2*w(t) + w(t-dt))
                                + dt**2*m*g)

        t += dt
        user_action(S, t, i+1)

    return S
```

The two last arguments to `user_action` must be carefully set: these should be time value and index for the most recently computed `S` value.

C.2.3 Making a Module

The `init_prms` and `solve` functions can now be combined with many different types of main programs and `user_action` functions. It is therefore preferable to have the general `init_prms` and `solve` functions in a module `box_spring` and import these functions in more user-specific programs. Making a module out of `init_prms` and `solve` is, according to Chapter 3.5, quite trivial as we just need to put the functions in a file `box_spring.py`.

It is always a good habit to include a test block in module files. To make the test block small, we place the statements in a separate

function `_test` and just call `_test` in the test block. The initial underscore in the name `_test` prevents this function from being imported by a `from box_spring import *` statement. Our test here simply prints solution at each time level. The following code snippet is then added to the module file to include a test block:

```
def _test():
    def print_S(S, t, step):
        print 't=%.2f S[%d]=%+g' % (t, step, S[step])

    # default values:
    from math import pi
    m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 1;
    dt = 2*pi/40; g = 9.81; w_formula = '0'; N = 80;

    m, b, L, k, beta, S0, dt, g, w, N = \
        init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N)
    S = solve(m, k, beta, S0, dt, g, w, N,
              user_action=print_S)

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

C.2.4 Verification

To check that the program works correctly, we need a series of problems where the solution is known. These test cases must be specified by someone with a good physical and mathematical understanding of the problem being solved. We already have a solution formula (C.10) that we can compare the computations with, and more tests can be made in the case $w \neq 0$ as well.

However, before we even think of checking the program against the formula (C.10), we should perform some much simpler tests. The simplest test is to see what happens if we do nothing with the system. This solution is of course not very exciting – the box is at rest, but it is in fact exciting to see if our program reproduces the boring solution. Many bugs in the program can be found this way! So, let us run the program `box_spring.py` with `-S0 0` as the only command-line argument. The output reads

```
t=0.00 S[0]=+0
t=0.16 S[1]=+0.121026
t=0.31 S[2]=+0.481118
t=0.47 S[3]=+1.07139
t=0.63 S[4]=+1.87728
t=0.79 S[5]=+2.8789
t=0.94 S[6]=+4.05154
t=1.10 S[7]=+5.36626
...
```

Something happens! All $S[1]$, $S[2]$, and so forth should be zero. What is the error?

There are two directions to follow now: we can either visualize the solution to understand more of what the computed $S(t)$ function looks

like (perhaps this explains what is wrong), or we can dive into the algorithm and compute $S[1]$ by hand to understand why it does not become zero. Let us follow both paths.

First we print out all terms on the right-hand side of the statement that computes $S[1]$. All terms except the last one ($\Delta t^2 mg$) are zero. The gravity term causes the spring to be stretched downward, which causes oscillations. We can see this from the governing equation (C.8) too: If there is no motion, $S(t) = 0$, the derivatives are zero (and $w = 0$ is default in the program), and then we are left with

$$kS = mg \quad : \quad S = \frac{m}{k}g. \quad (\text{C.22})$$

This result means that if the box is at rest, the spring is stretched (which is reasonable!). Either we have to start with $S(0) = \frac{m}{k}g$ in the equilibrium position, or we have to turn off the gravity force by setting $-g \ 0$ on the command line. Setting either $-S_0 \ 0 \ -g \ 0$ or $-S_0 \ 9.81$ shows that the whole S list contains either zeros or 9.81 values (recall that $m = k = 1$ so $S_0 = g$). This constant solution is correct, and the coding looks promising.

We can also plot the solution using the program `box_spring_plot`:

```
box_spring_plot.py --S0 0 --N 200
```

Terminal

Figure C.2 shows the function $Y(t)$ for this case where the initial stretch is zero, but gravity is causing a motion. With some mathematical analysis of this problem we can establish that the solution is correct. We have that $m = k = 1$ and $w = \beta = 0$, which implies that the governing equation is

$$\frac{d^2S}{dt^2} + S = g, \quad S(0) = 0, \quad dS/dt(0) = 0.$$

Without the g term this equation is simple enough to be solved by basic techniques you can find in most introductory books on differential equations. Let us therefore get rid of the g term by a little trick: we introduce a new variable $T = S - g$, and by inserting $S = T + g$ in the equation, the g is gone:

$$\frac{d^2T}{dt^2} + T = 0, \quad T(0) = -g, \quad \frac{dT}{dt}(0) = 0. \quad (\text{C.23})$$

This equation is of a very well-known type and the solution reads $T(t) = -g \cos t$, which means that $S(t) = g(1 - \cos t)$ and

$$Y(t) = -L - g(1 - \cos t) - \frac{b}{2}.$$

With $L = 10$, $g \approx 10$, and $b = 2$ we get oscillations around $y \approx 21$ with a period of 2π and a start value $Y(0) = -L - b/2 = 11$. A rough visual inspection of the plot shows that this looks right. A more thorough analysis would be to make a test of the numerical values in a new callback function (the program is found in `box_spring_test1.py`):

```
from box_spring import init_prms, solve
from math import cos

def exact_S_solution(t):
    return g*(1 - cos(t))

def check_S(S, t, step):
    error = exact_S_solution(t) - S[step]
    print 't=%.2f S[%d]=%+g error=%g' % (t, step, S[step], error)

# fixed values for a test:
from math import pi
m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 0
dt = 2*pi/40; g = 9.81; N = 200

def w(t):
    return 0

S = solve(m, k, beta, S0, dt, g, w, N, user_action=check_S)
```

The output from this program shows increasing errors with time, up as large values as 0.3. The difficulty is to judge whether this is the error one must expect because the program computes an approximate solution, or if this error points to a bug in the program – or a wrong mathematical formula.

From these sessions on program testing you will probably realize that verification of mathematical software is challenging. In particular, the design of the test problems and the interpretation of the numerical output require quite some experience with the interplay between physics (or another application discipline), mathematics, and programming.

C.3 Visualization

The purpose of this section is to add graphics to the oscillating system application developed in Appendix C.2. Recall that the function `solve` solves the problem and returns a list `S` with indices from 0 to `N`. Our aim is to plot this list and various physical quantities computed from it.

C.3.1 Simultaneous Computation and Plotting

The `solve` function makes a call back to the user's code through a callback function (the `user_action` argument to `solve`) at each time level. The callback function has three arguments: `S`, the time, and the

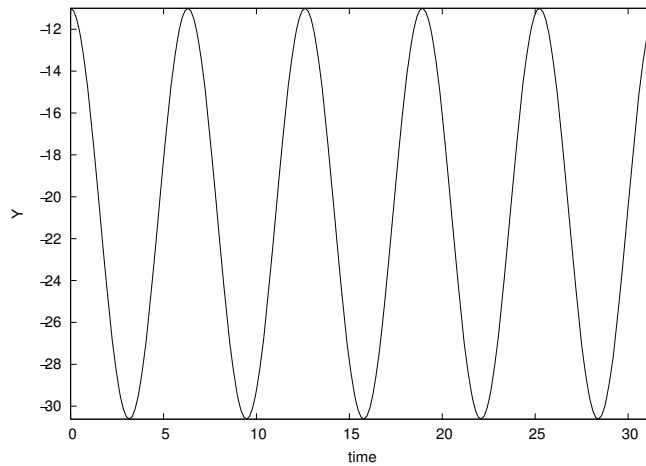


Fig. C.2 Positions $Y(t)$ of an oscillating box with $m = k = 1$, $w = \beta = 0$, $g = 9.81$, $L = 10$, and $b = 2$.

current time step number. Now we want the callback function to plot the position $Y(t)$ of the box during the computations. In principle this is easy, but S is longer than we want to plot, because S is allocated for the whole time simulation while the `user_action` function is called at time levels where only the indices in S up to the current time level have been computed (the rest of the elements in S are zero). We must therefore use a sublist of S , from time zero and up to the current time. The callback function we send to `solve` as the `user_action` argument can then be written like this:

```
def plot_S(S, t, step):
    if step == 0:          # nothing to plot yet
        return None

    tcoor = linspace(0, t, step+1)
    S = array(S[:len(tcoor)])
    Y = w(tcoor) - L - S - b/2.
    plot(tcoor, Y)
```

Note that L , dt , b , and w must be global variables in the user's main program.

The major problem with the `plot_S` function shown is that the `w(tcoor)` evaluation does not work. The reason is that `w` is a `StringFunction` object, and according to Chapter 4.4.3, `StringFunction` objects do not work with array arguments unless we call their `vectorize` function once. We therefore need to do a

```
w.vectorize(globals())
```

before calling `solve` (which calls `plot_S` repeatedly). Here is the main program with this important statement:

```

from box_spring import init_prms, solve
from scitools.std import *

# default values:
m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 1;
dt = 2*pi/40; g = 9.81; w_formula = '0'; N = 200;

m, b, L, k, beta, S0, dt, g, w, N = \
    init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N)

w.vectorize(globals())

S = solve(m, k, beta, S0, dt, g, w, N, user_action=plot_S)

```

Now the `plot_S` function works fine. You can try the program out by running

Terminal

```
box_spring_plot_v1.py
```

Fixing Axes. Both the t and the y axes adapt to the solution array in every plot. The adaptation of the y is okay since it is difficult to predict the future minimum and maximum values of the solution, and hence it is most natural to just adapt the y axis to the computed Y points so far in the simulation. However, the t axis should be fixed throughout the simulation, and this is easy since we know the start and end times. The relevant plot call now becomes⁴

```

plot(tcoor, Y,
     axis=[0, N*dt, min(Y), max(Y)],
     xlabel='time', ylabel='Y')

```

At the end of the simulation it can be nice to make a hardcopy of the last plot command performed in the `plot_S` function. We then just call

```
hardcopy('tmp_Y.eps')
```

after the `solve` function is finished.

In the beginning of the simulation it is convenient to skip plotting for a number of steps until there are some interesting points to visualize and to use for computing the axis extent. We also suggest to apply the recipe at the end of Chapter 4.4.3 to vectorize `w`. More precisely, we use `w.vectorize` in general, but turn to NumPy's `vectorize` feature only if the string formula contains an inline `if-else` test (to avoid requiring users to use `where` to vectorize the string expressions). One reason for paying attention to `if-else` tests in the w formula is that sudden movements of the plate are of interest, and this gives rise to step functions and strings like `'1 if t>0 else 0'`. A main program with all these features is listed next.

⁴ Note that the final time is $T = N\Delta t$.

```

from box_spring import init_prms, solve
from scitools.std import *

def plot_S(S, t, step):
    first_plot_step = 10          # skip the first steps
    if step < first_plot_step:
        return

    tcoor = linspace(0, t, step+1) # t = dt*step
    S = array(S[:len(tcoor)])
    Y = w(tcoor) - L - S - b/2.0    # (w, L, b are global vars.)

    plot(tcoor, Y,
         axis=[0, N*dt, min(Y), max(Y)],
         xlabel='time', ylabel='Y')

# default values:
m = 1; b = 2; L = 10; k = 1; beta = 0; S0 = 1
dt = 2*pi/40; g = 9.81; w_formula = '0'; N = 200

m, b, L, k, beta, S0, dt, g, w, N = \
    init_prms(m, b, L, k, beta, S0, dt, g, w_formula, N)

# vectorize the StringFunction w:
w_formula = str(w) # keep this to see if w=0 later
if 'else' in w_formula:
    w = vectorize(w)          # general vectorization
else:
    w.vectorize(globals())    # more efficient (when no if)

S = solve(m, k, beta, S0, dt, g, w, N, user_action=plot_S)

# first make a hardcopy of the the last plot of Y:
hardcopy('tmp_Y.eps')

```

C.3.2 Some Applications

What if we suddenly, right after $t = 0$, move the plate upward from $y = 0$ to $y = 1$? This will set the system in motion, and the task is to find out what the motion looks like.

There is no initial stretch in the spring, so the initial condition becomes $S_0 = 0$. We turn off gravity for simplicity and try a $w = 1$ function since the plate has the position $y = w = 1$ for $t > 0$:

Terminal

```
box_spring_plot.py --w '1' --S 0 --g 0
```

Nothing happens. The reason is that we specify $w(t) = 1$, but in the equation only d^2w/dt^2 has an effect and this quantity is zero. What we need to specify is a step function: $w = 0$ for $t \leq 0$ and $w = 1$ for $t > 0$. In Python such a function can be specified as a string expression `'1 if t>0 else 0'`. With a step function we obtain the right initial jump of the plate:

Terminal

```
box_spring_plot.py --w '1 if t > 0 else 0' \
  --S0 0 --g 0 --N 1000 --beta 0.1
```

Figure C.3 displays the solution. We see that the damping parameter has the effect of reducing the amplitude of $Y(t)$, and the reduction looks exponential, which is in accordance with the exact solution (C.10) (although this formula is not valid in the present case because $w \neq 0$ – but one gets the same exponential reduction even in this case). The box is initially located in $Y = 0 - (10 + 0) - 2/2 = -11$. During the first time step we get a stretch $S = 0.5$ and the plate jumps up to $y = 1$ so the box jumps to $Y = 1 - (10 + 0.5) - 2/2 = -10.5$. In Figure C.3b we see that the box starts correctly out and jumps upwards, as expected.

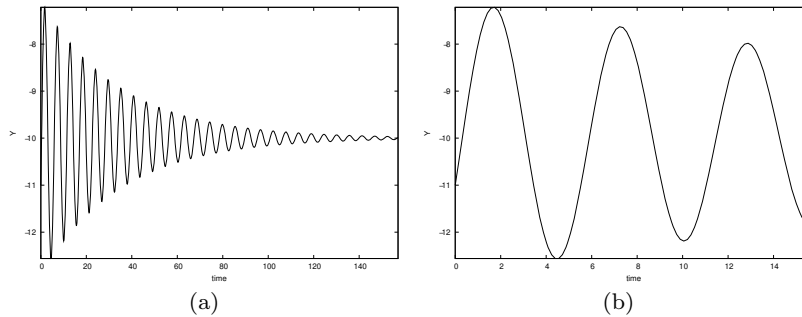


Fig. C.3 Plot of the position of an oscillating box where the end point of the spring ($w(t)$) is given a sudden movement at $t = 0$. Other parameters are $m = k = 1$, $\beta = 0.1$, $g = 0$, $S_0 = 0$. (a) 1000 time steps; (b) 100 steps for magnifying the first oscillation cycles.

More exciting motions of the box can be obtained by moving the plate back and forth in time, see for instance Figure C.4 on page 645.

C.3.3 Remark on Choosing Δt

If you run the `box_spring_plot.py` program with a large `-dt` argument (for Δt), strange things may happen. Try `-dt 2 -N 20` as command-line arguments and observe that Y jumps up and down in a saw tooth fashion so we clearly have too large time steps. Then try `-dt 2.1 -N 20` and observe that Y takes on very large values (10^5). This highly non-physical result points to an error in the program. However, the problem is not in the program, but in the numerical method used to solve (C.8). This method becomes unstable and hence useless if Δt is larger than a critical value. We shall not dig further into such problems, but just notice that mathematical models on a computer must be used with care, and that a serious user of simulation programs must understand how the mathematical methods work in detail and what their limitations are.

C.3.4 Comparing Several Quantities in Subplots

So far we have plotted Y , but there are other interesting quantities to look at, e.g., S , w , the spring force, and the damping force. The spring force and S are proportional, so only one of these is necessary to plot. Also, the damping force is relevant only if $\beta \neq 0$, and w is only relevant if the string formula is different from the default value '0'.

All the mentioned additional plots can be placed in the same figure for comparison. To this end, we apply the `subfigure` command in Easyviz and create a row of individual plots. How many plots we have depends on the values of `str(w)` and `beta`. The relevant code snippet for creating the additional plots is given below and appears after the part of the main program shown above.

```
# make plots of several additional interesting quantities:
tcoor = linspace(0, tstop, N+1)
S = array(S)

plots = 2          # number of rows of plots
if beta != 0:
    plots += 1
if w_formula != '0':
    plots += 1

# position Y(t):
plot_row = 1
subplot(plots, 1, plot_row)
Y = w(tcoor) - L - S - b/2.0
plot(tcoor, Y, xlabel='time', ylabel='Y')

# spring force (and S):
plot_row += 1
subplot(plots, 1, plot_row)
Fs = k*S
plot(tcoor, Fs, xlabel='time', ylabel='spring force')

if beta != 0:
    plot_row += 1
    subplot(plots, 1, plot_row)
    Fd = beta*diff(S) # diff is in numpy
    # len(diff(S)) = len(S)-1 so we use tcoor[:-1]:
    plot(tcoor[:-1], Fd, xlabel='time', ylabel='damping force')

if w_formula != '0':
    plot_row += 1
    subplot(plots, 1, plot_row)
    w_array = w(tcoor)
    plot(tcoor, w_array, xlabel='time', ylabel='w(t)')

# save this multi-axis plot in a file:
hardcopy('tmp.eps')
```

Figure C.4 displays what the resulting plot looks like for a test case with an oscillating plate (w). The command for this run is

Terminal

```
box_spring_plot.py --S0 0 --w '2*(cos(8*t)-1)' \
--N 600 --dt 0.05236
```

The rapid oscillations of the plate require us to use a smaller Δt and more steps (larger N).

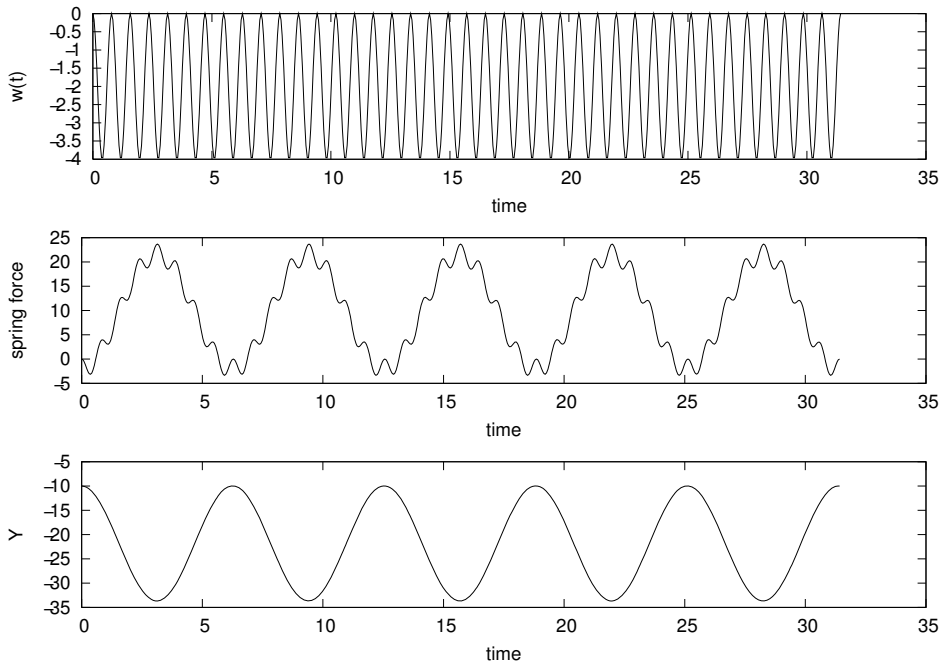


Fig. C.4 Plot of the plate position $w(t)$, the spring force (proportional to $S(t)$), and the position $Y(t)$ for a test problem where $w(t) = 2(\cos(8t) - 1)$, $\beta = g = 0$, $m = k = 1$, $S_0 = 0$, $\Delta t = 0.5236$, and $N = 600$.

C.3.5 Comparing Approximate and Exact Solutions

To illustrate multiple curves in the same plot and animations we turn to a slightly different program. The task now is to visually investigate how the accuracy of the computations depends on the Δt parameter. The smaller Δt is, the more accurate the solution S is. To look into this topic, we need a test problem with known solution. Setting $m = k = 1$ and $w = 0 = \beta = 0$ implies the exact solution $S(t) = g(1 - \cos t)$ (see Appendix C.2.4). The `box_spring_test1.py` program from Appendix C.2.4 can easily be extended to plot the calculated solution together with the exact solution. We drop the `user_action` callback function and just make the plot after having the complete solution S returned from the `solve` function:

```
tcoor = linspace(0, N*dt, len(S))
exact = exact_S_solution(tcoor)
plot(tcoor, S, 'r', tcoor, exact, 'b',
     xlabel='time', ylabel='S',
     legend=('computed S(t)', 'exact S(t)'),
     hardcopy='tmp_S.eps')
```


The two curves tend to lie upon each other, so to get some more insight into the details of the error, we plot the error itself, in a separate plot window:

```
figure()      # new plot window
S = array(S)  # turn list into NumPy array for computations
error = exact - S
plot(tcoor, error, xlabel='time', ylabel='error',
     hardcopy='tmp_error.eps')
```

The error increases in time as the plot in Figure C.5a clearly shows.

C.3.6 Evolution of the Error as Δt Decreases

Finally, we want to investigate how the error curve evolves as the time step Δt decreases. In a loop we halve Δt in each pass, solve the problem, compute the error, and plot the error curve. From the finite difference formulas involved in the computational algorithm, we can expect that the error is of order Δt^2 . That is, if Δt is halved, the error should be reduced by $1/4$.

The resulting plot of error curves is not very informative because the error reduces too quickly (by several orders of magnitude). A better plot is obtained by taking the logarithm of the error. Since an error curve may contain positive and negative elements, we take the absolute value of the error before taking the logarithm. We also note that S_0 is always correct, so it is necessary to leave out the initial value of the error array to avoid the logarithm of zero.

The ideas of the previous two paragraphs can be summarized in a Python code snippet:

```
figure()      # new plot window
dt = 2*pi/10
tstop = 8*pi  # 4 periods
N = int(tstop/dt)
for i in range(6):
    dt /= 2.0
    N *= 2
    S = solve(m, k, beta, S0, dt, g, w, N)
    S = array(S)
    tcoor = linspace(0, tstop, len(S))
    exact = exact_S_solution(tcoor)
    abserror = abs(exact - S)
    # drop abserror[0] since it is always zero and causes
    # problems for the log function:
    logerror = log10(abserror[1:])
    plot(tcoor[1:], logerror, 'r', xlabel='time',
         ylabel='log10(abs(error))')
    hold('on')
hardcopy('tmp_errors.eps')
```

The resulting plot is shown in Figure C.5b.

Visually, it seems to be a constant distance between the curves in Figure C.5b. Let d denote this difference and let E_i be the absolute

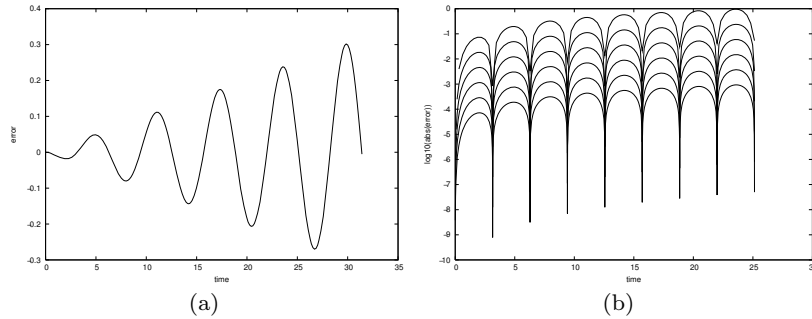


Fig. C.5 Error plots for a test problem involving an oscillating system: (a) the error as a function of time; (b) the logarithm of the absolute value of the error as a function of time, where Δt is reduced by one half from one curve to the next one below.

error curve associated with Δt in the i -th pass in the loop. What we plot is $\log_{10} E_i$. The difference between two curves is then $D_{i+1} = \log_{10} E_i - \log_{10} E_{i+1} = \log_{10}(E_i/E_{i+1})$. If this difference is roughly 0.5 as we see from Figure C.5b, we have

$$\log_{10} \frac{E_i}{E_{i+1}} = d = 0.5 \quad : \quad E_{i+1} = \frac{1}{3.16} E_i.$$

That is, the error is reduced, but not by the theoretically expected factor 4. Let us investigate this topic in more detail by plotting D_{i+1} .

We make a loop as in the last code snippet, but store the `logerror` array from the previous pass in the loop (E_i) in a variable `logerror_prev` such that we can compute the difference D_{i+1} as

```
logerror_diff = logerror_prev - logerror
```

There are two problems to be aware of now in this array subtraction: (i) the `logerror_prev` array is not defined before the second pass in the loop (when `i` is one or greater), and (ii) `logerror_prev` and `logerror` have different lengths since `logerror` has twice as many time intervals as `logerror_prev`. Numerical Python does not know how to compute this difference unless the arrays have the same length. We therefore need to use every two elements in `logerror`:

```
logerror_diff = logerror_prev - logerror[::2]
```

An additional problem now arises because the set of time coordinates, `tcoor`, in the current pass of the loop also has twice as many intervals so we need to plot `logerror_diff` against `tcoor[::2]`.

The complete code snippet for plotting differences between the logarithm of the absolute value of the errors now becomes

```
figure()
dt = 2*pi/10
tstop = 8*pi # 4 periods
```

```

N = int(tstop/dt)
for i in range(6):
    dt /= 2.0
    N *= 2
    S = solve(m, k, beta, S0, dt, g, w, N)
    S = array(S)
    tcoor = linspace(0, tstop, len(S))
    exact = exact_S_solution(tcoor)
    aberror = abs(exact - S)
    logerror = log10(aberror[1:])
    if i > 0:
        logerror_diff = logerror_prev - logerror[:2]
        plot(tcoor[1::2], logerror_diff, 'r', xlabel='time',
              ylabel='difference in log10(abs(error))')
        hold('on')
        meandiff = mean(logerror_diff)
        print 'average log10(abs(error)) difference:', meandiff
        logerror_prev = logerror
hardcopy('tmp_errors_diff.eps')

```

Figure C.6 shows the result. We clearly see that the differences between

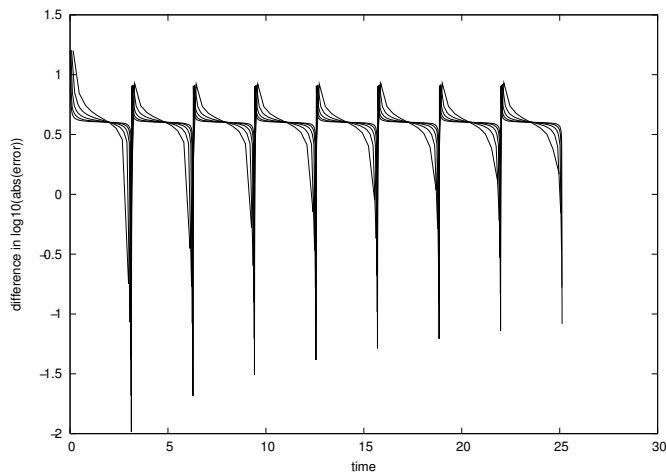


Fig. C.6 Differences between the curves in Figure C.5b.

the curves in Figure C.5b are almost the same even if Δt is reduced by several orders of magnitude.

In the loop we also print out the average value of the difference curves in Figure C.6:

```

average log10(abs(error)) difference: 0.558702094666
average log10(abs(error)) difference: 0.56541814902
average log10(abs(error)) difference: 0.576489014172
average log10(abs(error)) difference: 0.585704362507
average log10(abs(error)) difference: 0.592109360025

```

These values are “quite constant”. Let us use 0.57 as an representative value and see what it implies. Roughly speaking, we can then say that

$$\log_{10} E_i - \log_{10} E_{i+1} = 0.57.$$

Collecting the two first terms and applying the exponential function 10^x on both sides we get that

$$E_{i+1} = \frac{1}{3.7} E_i.$$

This error reduction when Δt is decreased is not quite as good as we would theoretically expect ($1/4$), but it is close. The purpose of this brief analysis is primarily to show how errors can be explored by plotting, and how we can take advantage of array computing to produce various quantities of interest in a problem. A more thorough investigation of how the error depends on Δt would use time integrals of the error instead of the complete error curves.

Again we mention that the complete problem analyzed in this appendix is challenging to understand because of its mix of physics, mathematics, and programming. In real life, however, problem solving in science and industry involve multi-disciplinary projects where people with different competence work together. As a scientific programmer you must then be able to fully understand what to program and how to verify the results. This is a requirement in the current summarizing example too. You have to accept that your programming problem is buried in a lot of physical and mathematical details.

Having said this, we expect that most readers of this book also gain a background in physics and mathematics so that the present summarizing example can be understood in complete detail, at least at some later stage.

C.4 Exercises

Exercise C.1. *Use a w function with a step.*

Set up a problem with the `box_spring_plot.py` program where the initial stretch in the spring is 1 and there is no gravity force. Between $t = 20$ and $t = 30$ we move the plate suddenly from 0 to 2 and back again:

$$w(t) = \begin{cases} 2, & 20 < t < 30, \\ 0, & \text{otherwise} \end{cases}$$

Run this problem and view the solution. ◇

Exercise C.2. *Make a callback function in Exercise C.1.*

Doing Exercise C.1 shows that the Y position increases significantly in magnitude when the “jump” the plate upward and back again at $t = 20$ and $t = 30$, respectively. Make a program where you import from the `box_spring` module and provide a callback function that checks if $Y < 9$ and then aborts the program. Name of program file: `box_spring_Ycrit.py`. ◇

Exercise C.3. *Improve input to the simulation program.*

The oscillating system in Appendix C.1 has an equilibrium position $S = mg/k$, see (C.22) on page 638. A natural case is to let the box start at rest in this position and move the plate to induce oscillations. We must then prescribe $S_0 = mg/k$ on the command line, but the numerical value depends on the values of m and g that we might also give in the command line. However, it is possible to specify `-S0 m*g/k` on the command line if we in the `init_prms` function first let `S0` be a string in the `elif` test and then, after the `for` loop, execute `S0 = eval(S0)`. At that point, `m` and `k` are read from the command line so that `eval` will work on `'m*g/k'`, or any other expression involving data from the command. Implement this idea.

A first test problem is to start from rest in the equilibrium position $S(0) = mg/k$ and give the plate a sudden upward change in position from $y = 0$ to $y = 1$. That is,

$$w(t) = \begin{cases} 0, & t \leq 0, \\ 1, & t > 0 \end{cases}$$

You should get oscillations around the displaced equilibrium position $Y = w - L - S_0 = -9 - 2g$. Name of program file: `box_spring2.py`. \diamond

D.1 Using a Debugger

A debugger is a program that can help you to find out what is going on in a computer program. You can stop the execution at any prescribed line number, print out variables, continue execution, stop again, execute statements one by one, and repeat such actions until you track down abnormal behavior and find bugs.

Here we shall use the debugger to demonstrate the program flow of the `ball_table.py` code from Chapter 2.4.2. You are strongly encouraged to carry out the steps below on your computer to get a glimpse of what a debugger can do.

1. Go to the folder `src/basic` associated with Chapter 2.
2. Start IPython:

Terminal

```
Unix/DOS> ipython
```

3. Run the program `ball_table.py` with the debugger on `(-d)`:

```
In [1]: run -d ball_table.py
```

Instead of running the program as usual, we now enter the debugger and get a prompt

```
ipdb>
```

After this prompt we can issue various debugger commands. The most important ones will be described as we go along.

4. Type `continue` or just `c` to go to the first statement in the file. Now you can see a printout of where we are in the program:

```
1---> 1 g = 9.81;  v0 = 5
      2 dt = 0.05
      3
```

Each program line is numbered and the arrow points to the next line to be executed. This is called the *current line*.

5. Type `step` or just `s` to execute the current line, which here initializes `g` and `v0`. Afterwards, we can print out their values by just writing the variable names at the `ipdb>` prompt:

```
ipdb>g
Out[1]: 9.8100000000000005
ipdb>v0
Out[1]: 5
```

We can explicitly demonstrate that `dt` is not yet initialized as a variable:

```
ipdb>dt
*** NameError: name 'dt' is not defined
```

6. Type `list` or just `l` to get a listing of the program lines around the current line:

```
ipdb>list
1      1 g = 9.81;  v0 = 5
----> 2 dt = 0.05
      3
      4 def y(t):
      5     return v0*t - 0.5*g*t**2
      6
      7 def table():
      8     data = []  # store [t, y] pairs in a nested list
      9     t = 0
     10     while y(t) >= 0:
     11         data.append([t, y(t)])
```

To see the lines between line number 11 and 30, type `list 11,30`. Writing `help list` results in a short description of the `list` command.

7. Let us set a *break point* at the line with the call `data = table()`. From the listing we see that this line has number 15. A break point means that the program execution will halt at this point to let us examine variables and perform step-wise execution with the `step` (`s`) command. Write

```
ipdb>break 15
```

to set the break point at line 15. To run the program up to this point, type `continue` or `c`. Alternatively, we could have issued some `step` commands to reach line 15.

8. Type repeated `step` (`s`) commands and see that we jump to the `table` function. Continue with step commands and observe that when we reach the `while` loop, we jump to the `y` function. After the `return` statement in the `y` function the debugger writes out the return value.

More step commands show how we jump between the `table` and `y` functions every time the `y(t)` expression appears in the loop (either in the `append` call or in the loop condition).

Inside the `y` or `table` function we may examine variables by just typing their names. One can, for instance, monitor how the `data` list develops. Inside the `y` function, `data` does not exist (since `data` is a local variable in the `table` not visible outside this function).

Stepwise execution with the `s` command is tedious. We may set a break point at line 11 when there are more elements in the `data` list, say when it has more than four elements:

```
ipdb>break 11, len(data)>4
```

Continue execution with the `continue` or `c` command and observe that the program stops at line 11 as soon as the condition on the length is fulfilled. Writing out the `data` list shows that it now has five elements.

9. The `next` or `n` command executes the current line, in the same way as the `step` or `s` command, but contrary to the latter, the `n` command does not enter functions being called. To demonstrate this point, type `n` a few times. You will experience that the statements involving `y(t)` calls are executed without stopping inside the `y` function. That is, `n` commands lead to stops inside the `table` function only. This is a quick way to examine how the `while` loop is executed.

10. Typing `c` continues execution until the next break point, but there are no more break points so the execution is continued until the end or until a Python error occurs. The latter action takes place in our program:

```

      8     data = [] # store [t, y] pairs in a nested list
      9     t = 0
----> 10     while y(t) >= 0:
      11         data.append([t, y(t)])
      12         t += dt

<type 'exceptions.TypeError': 'list' object is not callable>
```

We can now check what `y` is by typing its name, and we quickly realize that `y` is a list, not a function anymore.

At this point, I hope you realize that a debugger is a very handy tool for monitoring the program flow, checking variables, and thereby understanding why errors occur, as we have demonstrated in the step-wise exploration above.

D.2 How to Debug

Most programmers will claim that writing code consumes a small portion of the time it takes to develop a program – the major portion of

the work concerns testing the program and finding errors¹. Newcomers to programming often panic when their program runs for the first time and aborts with a seemingly cryptic error message. How do you approach the art of debugging? This appendix summarizes some important working habits in this respect. Some of the tips are useful for problem solving in general, not only when writing and testing Python programs.

D.2.1 A Recipe for Program Writing and Debugging

1. Make sure that you *understand the problem* the program is supposed to solve. We can make a general claim: If you do not understand the problem and the solution method, you will never be able to make a correct program². It may be necessary to read a problem description or exercise many times and study relevant background material.
2. *Work out some examples* on input and output of the program. Such examples are important for controlling the understanding of the purpose of the program, and for verifying the implementation.
3. *Decide on a user interface*, i.e., how you want to get data into the program (command-line input, file input, questions and answers, etc.).
4. *Sketch rough algorithms* for various parts of the program. Some programmers prefer to do this on a piece of paper, others prefer to start directly in Python and write Python-like code with comments to sketch the program (this is easily developed into real Python code later).
5. *Look up information* on how to program different parts of the problem. Few programmers can write the whole program without consulting manuals, books, and the Internet. You need to know and understand the basic constructs in a language and some fundamental problem solving techniques, but technical details can be looked up.

The more program examples you have studied (in this book, for instance), the easier it is to adapt ideas from an existing example to solve a new problem³. Remember that exercises in this book are often closely linked to examples in the text.

¹ “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.” –Brian W. Kernighan, computer scientist, 1942-.

² This is not entirely true. Sometimes students with limited understanding of the problem are able to grab a similar program and guess at a few modifications – and get a program that works. But this technique is based on luck and not on understanding. The famous Norwegian computer scientist Kristen Nygaard (1926-2002) phrased it precisely: “Programming is understanding”.

³ “The secret to creativity is knowing how to hide your sources.” –Albert Einstein, physicist, 1879-1955.

6. *Write the program.* Be extremely careful with what you write. In particular, compare all mathematical statements and algorithms with the original mathematical expressions on paper.

In longer programs, do not wait until the program is complete before you start testing it – test parts while you write.

7. *Run the program.*

If the program aborts with an error message from Python, these messages are fortunately quite precise and helpful. First, locate the line number where the error occurs and read the statement, then carefully read the error message. The most common errors (exceptions) are listed below.

SyntaxError: Illegal Python code.

```
File "somefile.py", line 5
x = . 5
    ^
SyntaxError: invalid syntax
```

Often the error is precisely indicated, as above, but sometimes you have to search for the error on the previous line.

NameError: A name (variable, function, module) is not defined.

```
File "somefile.py", line 20, in <module>
    table(10)
File "somefile.py", line 16, in table
    value, next, error = L(x, n)
File "somefile.py", line 8, in L
    exact_error = log(1+x) - value_of_sum
NameError: global name 'value_of_sum' is not defined
```

Look at the last of the lines starting with `File` to see where in the program the error occurs. The most common reasons for a `NameError` are

- a misspelled name,
- a variable that is not initialized,
- a function that you have forgotten to define,
- a module that is not imported.

TypeError: An object of wrong type is used in an operation.

```
File "somefile.py", line 17, in table
    value, next, error = L(x, n)
File "somefile.py", line 7, in L
    first_neglected_term = (1.0/(n+1))*(x/(1.0+x))**(n+1)
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Print out objects and their types (here: `print x, type(x), n, type(n)`), and you will most likely get a surprise. The reason for a `TypeError` is often far away from the line where the `TypeError` occurs.

ValueError: An object has an illegal value.

```
File "somefile.py", line 8, in L
    y = sqrt(x)
ValueError: math domain error
```

Print out the value of objects that can be involved in the error (here: `print x`).

IndexError: An index in a list, tuple, string, or array is too large.

```
File "somefile.py", line 21
    n = sys.argv[i+1]
IndexError: list index out of range
```

Print out the length of the list, and the index if it involves a variable (here: `print len(sys.argv), i`).

8. *Verify the implementation.* Assume now that we have a program that runs without error messages from Python. Before judging the results of the program, set precisely up a test case where you know the exact solution⁴. Insert `print` statements for all key results in the program so that you can easily compare calculations in the program with those done by hand.

If your program produces wrong answers, start to *examine intermediate results*. Also remember that your hand calculations may be wrong!

9. If you need a lot of `print` statements in the program, you may *use a debugger* as explained in Appendix D.1.

Some may think that this list is very comprehensive. However, it just contains the items that you should always address when developing programs. Never forget that computer programming is a difficult task⁵!

D.2.2 Application of the Recipe

Let us illustrate the points above in a specific programming problem.

Problem. Implement the Midpoint rule for numerical integration. The Midpoint rule for approximating an integral $\int_a^b f(x)dx$ reads

$$I = h \sum_{i=1}^n f(a + (i - \frac{1}{2})h), \quad h = \frac{b - a}{n}. \quad (\text{D.1})$$

Solution. We just follow the individual steps in the recipe.

1. *Understand the problem.* In this problem we must understand how to program the formula (D.1). Observe that we do not need to understand how the formula is derived, because we do not apply the

⁴ This is in general quite difficult. In complicated mathematical problems it is an art to construct good test problems and procedures for providing evidence that the program works.

⁵ “Program writing is substantially more demanding than book writing.” “Why is it so? I think the main reason is that a larger attention span is needed when working on a large computer program than when doing other intellectual tasks.” –Donald Knuth [4, p. 18], computer scientist, 1938–.

derivation in the program⁶. What is important, is to notice that the formula is an *approximation* of an integral. If we try to integrate a function $f(x)$, we will probably not get an exact answer. Whether we have an approximation error or a programming error is always difficult to judge. We will meet this difficulty below.

2. *Work out examples.* As a test case we choose to integrate

$$f(x) = \sin^{-1}(x). \quad (\text{D.2})$$

between 0 and π . From a table of integrals we find that this integral equals

$$\left[x \sin^{-1}(x) + \sqrt{1-x^2} \right]_0^\pi. \quad (\text{D.3})$$

The formula (D.1) gives an approximation to this integral, so the program will (most likely) print out a result different from (D.3). It would therefore be very helpful to construct a calculation where there are no approximation errors. Numerical integration rules usually integrate some polynomial of low order exactly. For the Midpoint rule it is obvious, if you understand the derivation of this rule, that a constant function will be integrated exactly. We therefore also introduce a test problem where we integrate $g(x) = 1$ from 0 to 10. The answer should be exactly 10.

Input and output: The input to the calculations is the function to integrate, the integration limits a and b , and the n parameter (number of intervals) in the formula (D.1). The output from the calculations is the approximation to the integral.

3. *User interface.* We decide to program the two functions $f(x)$ and $g(x)$ directly in the program. We also specify the corresponding integration limits a and b in the program, but we read a common n for both integrals from the command line. Note that this is not a flexible user interface, but it suffices as a start for creating a working program. A much better user interface is to read f , a , b , and n from the command line, which will be done later in a more complete solution to the present problem.

4. *Algorithm.* Like most mathematical programming problems, also this one has a generic part and an application part. The generic part is the formula (D.1), which is applicable to an arbitrary function $f(x)$. The implementation should reflect that we can specify any Python function $\mathbf{f}(\mathbf{x})$ and get it integrated. This principle calls for calculating (D.1) in a Python function where the input to the computation (f , a , b , n) are arguments. The function heading can look as `integrate(f, a, b, n)`, and the value of (D.1) is returned.

⁶ You often need to understand the background for and the derivation of a mathematical formula in order to work out sensible test problems for verification. Sometimes this must be done by experts on the particular problem at hand.

The test part of the program consists of defining the test functions $f(x)$ and $g(x)$ and writing out the calculated approximations to the corresponding integrals.

A first rough sketch of the program can then be

```
def integrate(f, a, b, n):
    # compute integral, store in I
    return I

def f(x):
    ...

def g(x):
    ...

# test/application part:
n = sys.argv[1]
I = integrate(g, 0, 10, n)
print "Integral of g equals %g" % I
I = integrate(f, 0, pi, n)
# calculate and print out the exact integral of f
```

The next step is to make a detailed implementation of the `integrate` function. Inside this function we need to compute the sum (D.1). In general, sums are computed by a `for` loop over the summation index, and inside the loop we calculate a term in the sum and add it to an accumulation variable. Here is the algorithm:

<pre>s = 0 for i from 1 to n: s = s + f(a + (i - 1/2)h) I = sh</pre>
--

5. *Look up information.* Our test function $f(x) = \sin^{-1}(x)$ must be evaluated in the program. How can we do this? We know that many common mathematical functions are offered by the `math` module. It is therefore natural to check if this module has an inverse sine function. The best place to look for Python modules is the Python Library Reference (see Chapter 2.4.3). We go to the index of this manual, find the “math” entry, and click on it. Alternatively, you can write `pydoc math` on the command line. Browsing the manual for the `math` module shows that there is an inverse sine function, with the name `asin`.

In this simple problem, where we use very basic constructs from the first three chapters of this book, there is hardly any need for looking at similar examples. Nevertheless, if you are uncertain about programming a mathematical sum, you may look at examples from, e.g., Chapter 2.2.4.

6. *Write the program.* Here is our first attempt to write the program. You can find the whole code in the file `appendix/integrate_v1.py`.

```
def integrate(f, a, b, n):
    s = 0
    for i in range(1, n):
        s += f(a + i*h)
    return s

def f(x):
    return asin(x)

def g(x):
    return 1

# test/application part:
n = sys.argv[1]
I = integrate(g, 0, 10, n)
print "Integral of g equals %g" % I
I = integrate(f, 0, pi, n)
I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
print "Integral of f equals %g (exact value is %g)' % \
      (I, I_exact)
```

7. *Run the program.* We try a first execution from IPython

```
In [1]: run integrate_v1.py
```

Unfortunately, the program aborts with an error:

```
File "integrate_v1.py", line 8
    return asin(x)
    ^
```

IndentationError: expected an indented block

We go to line 8 and look at that line and the surrounding code:

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

Python expects that the return line is indented, because the function body must always be indented. By the way, we realize that there is a similar error in the `g(x)` function as well. We correct these errors:

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

def g(x):
    return 1
```

Running the program again makes Python respond with

```
File "integrate_v1.py", line 24
    (I, I_exact)
    ^
```

SyntaxError: EOL while scanning single-quoted string

There is nothing wrong with line 24, but line 24 is a part of the statement starting on line 23:

```
print "Integral of f equals %g (exact value is %g)' % \
      (I, I_exact)
```

A `SyntaxError` implies that we have written illegal Python code. Inspecting line 23 reveals that the string to be printed starts with a

double quote, but ends with a single quote. We must be consistent and use the same enclosing quotes in a string. Correcting the statement,

```
print "Integral of f equals %g (exact value is %g)" % \
      (I, I_exact)
```

and rerunning the program yields the output

```
Traceback (most recent call last):
  File "integrate_v1.py", line 18, in <module>
    n = sys.argv[1]
NameError: name 'sys' is not defined
```

Obviously, we need to import `sys` before using it. We add `import sys` and run again:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 19, in <module>
    n = sys.argv[1]
IndexError: list index out of range
```

This is a very common error: We index the list `sys.argv` out of range because we have not provided enough command-line arguments. Let us use $n = 10$ in the test and provide that number on the command line:

```
In [5]: run integrate_v1.py 10
```

We still have problems:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 20, in <module>
    I = integrate(g, 0, 10, n)
  File "integrate_v1.py", line 7, in integrate
    for i in range(1, n):
TypeError: range() integer end argument expected, got str.
```

It is the final File line that counts (the previous ones describe the nested functions calls up to the point where the error occurred). The error message for line 7 is very precise: The end argument to `range`, `n`, should be an integer, but it is a string. We need to convert the string `sys.argv[1]` to `int` before sending it to the `integrate` function:

```
n = int(sys.argv[1])
```

After a new edit-and-run cycle we have other error messages waiting:

```
Traceback (most recent call last):
  File "integrate_v1.py", line 20, in <module>
    I = integrate(g, 0, 10, n)
  File "integrate_v1.py", line 8, in integrate
    s += f(a + i*h)
NameError: global name 'h' is not defined
```

The `h` variable is used without being assigned a value. From the formula (D.1) we see that $h = (b - a)/n$, so we insert this assignment at the top of the `integrate` function:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    ...
```

A new run results in a new error:

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 23, in <module>
    I = integrate(f, 0, pi, n)
NameError: name 'pi' is not defined
```

Looking carefully at all output, we see that the program managed to call the `integrate` function with `g` as input and write out the integral. However, in the call to `integrate` with `f` as argument, we get a `NameError`, saying that `pi` is undefined. When we wrote the program we took it for granted that `pi` was π , but we need to import `pi` from `math` to get this variable defined, before we call `integrate`:

```
from math import pi
I = integrate(f, 0, pi, n)
```

The output of a new run is now

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 24, in <module>
    I = integrate(f, 0, pi, n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 13, in f
    return asin(x)
NameError: global name 'asin' is not defined
```

A similar error occurred: `asin` is not defined as a function, and we need to import it from `math`. We can either do a

```
from math import pi, asin
```

or just do the rough

```
from math import *
```

to avoid any further errors with undefined names from the `math` module (we will get one for the `sqrt` function later, so we simply use the last “import all” kind of statement).

There are still more errors:

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 24, in <module>
    I = integrate(f, 0, pi, n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 13, in f
    return asin(x)
ValueError: math domain error
```

Now the error concerns a wrong `x` value in the `f` function. Let us print out `x`:


```
def f(x):
    print x
    return asin(x)
```

The output becomes

```
Integral of g equals 9
0.314159265359
0.628318530718
0.942477796077
1.25663706144
Traceback (most recent call last):
  File "integrate_v1.py", line 25, in <module>
    I = integrate(f, 0, pi, n)
  File "integrate_v1.py", line 9, in integrate
    s += f(a + i*h)
  File "integrate_v1.py", line 14, in f
    return asin(x)
ValueError: math domain error
```

We see that all the `asin(x)` computations are successful up to and including $x = 0.942477796077$, but for $x = 1.25663706144$ we get an error. A “math domain error” may point to a wrong x value for $\sin^{-1}(x)$ (recall that the domain of a function specifies the legal x values for that function).

To proceed, we need to think about the mathematics of our problem: Since $\sin(x)$ is always between -1 and 1 , the inverse sine function cannot take x values outside the interval $[-1, 1]$. The problem is that we try to integrate $\sin^{-1}(x)$ from 0 to π , but only integration limits within $[-1, 1]$ make sense (unless we allow for complex-valued trigonometric functions). Our test problem is hence wrong from a mathematical point of view. We need to adjust the limits, say 0 to 1 instead of 0 to π . The corresponding program modification reads

```
I = integrate(f, 0, 1, n)
```

We run again and get

```
Integral of g equals 9
0
0
0
0
0
0
0
0
0
0
Traceback (most recent call last):
  File "integrate_v1.py", line 26, in <module>
    I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
ValueError: math domain error
```

It is easy to go directly to the `ValueError` now, but one should always examine the output from top to bottom. If there is strange output before Python reports an error, there may be an error indicated by our `print` statements which causes Python to abort the program. This is not the case in the present example, but it is a good habit to start at the top of the output anyway. We see that all our `print x` statements inside the `f` function say that x is zero. This must be wrong – the idea

of the integration rule is to pick n different points in the integration interval $[0, 1]$.

Our $f(x)$ function is called from the `integrate` function. The argument to `f`, `a + i*h`, is seemingly always 0. Why? We print out the argument and the values of the variables that make up the argument:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    s = 0
    for i in range(1, n):
        print a, i, h, a+i*h
        s += f(a + i*h)
    return s
```

Running the program shows that `h` is zero and therefore `a+i*h` is zero.

Why is `h` zero? We need a new `print` statement in the computation of `h`:

```
def integrate(f, a, b, n):
    h = (b-a)/n
    print b, a, n, h
    ...
```

The output shows that `a`, `b`, and `n` are correct. Now we have encountered an error that we often discuss in this book: integer division (see Chapter 1.3.1). The formula $(1 - 0)/10 = 1/10$ is zero according to integer division. The reason is that `a` and `b` are specified as 0 and 1 in the call to `integrate`, and 0 and 1 imply `int` objects. Then `b-a` becomes an `int`, and `n` is an `int`, causing an `int/int` division. We must ensure that `b-a` is `float` to get the right mathematical division in the computation of `h`:

```
def integrate(f, a, b, n):
    h = float(b-a)/n
    ...
```

Thinking that the problem with wrong x values in the inverse sine function is resolved, we may remove all the `print` statements in the program, and run again.

The output now reads

```
Integral of g equals 9
Traceback (most recent call last):
  File "integrate_v1.py", line 25, in <module>
    I_exact = pi*asin(pi) - sqrt(1 - pi**2) - 1
ValueError: math domain error
```

That is, we are back to the `ValueError` we have seen before. The reason is that `asin(pi)` does not make sense, and the argument to `sqrt` is negative. The error is simply that we forgot to adjust the upper integration limit in the computation of the exact result. This is another very common error. The correct line is

```
I_exact = 1*asin(1) - sqrt(1 - 1**2) - 1
```

We could avoid the error by introducing variables for the integration limits, and a function for $\int f(x)dx$ would make the code cleaner:

```
a = 0; b = 1
def int_f_exact(x):
    return x*asin(x) - sqrt(1 - x**2)
I_exact = int_f_exact(b) - int_f_exact(a)
```

Although this is more work than what we initially aimed at, it usually saves time in the debugging phase to do things this proper way.

Eventually, the program seems to work! The output is just the result of our two print statements:

```
Integral of g equals 9
Integral of f equals 5.0073 (exact value is 0.570796)
```

8. *Verify the results.* Now it is time to check if the numerical results are correct. We start with the simple integral of 1 from 0 to 10: The answer should be 10, not 9. Recall that for this particular choice of integration function, there is no approximation error involved (but there could be a small round-off error). Hence, there must be a programming error.

To proceed, we need to calculate some intermediate mathematical results by hand and compare these with the corresponding statements in the program. We choose a very simple test problem with $n = 2$ and $h = (10 - 0)/2 = 5$. The formula (D.1) becomes

$$I = 5 \cdot (1 + 1) = 10.$$

Running the program with $n = 2$ gives

```
Integral of g equals 1
```

We insert some print statements inside the `integrate` function:

```
def integrate(f, a, b, n):
    h = float(b-a)/n
    s = 0
    for i in range(1, n):
        print 'i=%d, a+i*h=%g' % (i, a+i*h)
        s += f(a + i*h)
    return s
```

Here is the output:

```
i=1, a+i*h=5
Integral of g equals 1
i=1, a+i*h=0.5
Integral of f equals 0.523599 (exact value is 0.570796)
```

There was only one pass in the `i` loop in `integrate`. According to the formula, there should be n passes, i.e., two in this test case. The limits of `i` must be wrong. The limits are produced by the call `range(1,n)`. We recall that such a call results in integers going from 1 up to n , but

not including `n`. We need to include `n` as value of `i`, so the right call to `range` is `range(1,n+1)`.

We make this correction and rerun the program. The output is now

```
i=1, a+i*h=5
i=2, a+i*h=10
Integral of g equals 2
i=1, a+i*h=0.5
i=2, a+i*h=1
Integral of f equals 2.0944 (exact value is 0.570796)
```

The integral of 1 is still not correct. We need more intermediate results!

In our quick hand calculation we knew that $g(x) = 1$ so all the $f(a + (i - \frac{1}{2})h)$ evaluations were rapidly replaced by ones. Let us now compute all the x coordinates $a + (i - \frac{1}{2})h$ that are used in the formula:

$$i = 1 : a + (i - \frac{1}{2})h = 2.5, \quad i = 2 : a + (i - \frac{1}{2})h = 7.5.$$

Looking at the output from the program, we see that the argument to `g` has a different value – and fortunately we realize that the formula we have coded is wrong. It should be `a+(i-0.5)*h`.

We correct this error and run the program:

```
i=1, a+(i-0.5)*h=2.5
i=2, a+(i-0.5)*h=7.5
Integral of g equals 2
...
```

Still the integral is wrong⁷.

Now we read the code more carefully and compare expressions with those in the mathematical formula. We should, of course, have done this already when writing the program, but it is easy to get excited when writing code and hurry for the end. This ongoing story of debugging probably shows that reading the code carefully can save much debugging time⁸. We clearly add up all the f evaluations correctly, but then this sum must be multiplied by h , and we forgot that in the code. The `return` statement in `integrate` must therefore be modified to

```
return s*h
```

Eventually, the output is

```
Integral of g equals 10
Integral of f equals 0.568484 (exact value is 0.570796)
```

and we have managed to integrate a constant function in our program! Even the second integral looks promising!

To judge the result of integrating the inverse sine function, we need to run several increasing n values and see that the approximation gets

⁷ At this point you may give up programming, but the more skills you pick up in debugging, the more fun it is to hunt for errors! Debugging is like reading an exciting criminal novel: the detective follows different ideas and tracks, but never gives up before the culprit is caught.

⁸ Actually, being extremely careful with what you write, and comparing all formulas with the mathematics, may be the best way to get more spare time when taking a programming course!

better. For $n = 2, 10, 100, 1000$ we get 0.550371, 0.568484, 0.570714, 0.570794, to be compared to the exact⁹ value 0.570796. The decreasing error provides evidence for a correct program, but it is not a strong proof. We should try out more functions. In particular, linear functions are integrated exactly by the Midpoint rule. We can also measure the speed of the decrease of the error and check that the speed is consistent with the properties of the Midpoint rule, but this is a mathematically more advanced topic.

The very important lesson learned from these debugging sessions is that you should start with a simple test problem where all formulas can be computed by hand. If you start out with $n = 100$ and try to integrate the inverse sine function, you will have a much harder job with tracking down all the errors.

9. *Use a debugger.* Another lesson learned from these sessions is that we needed many `print` statements to see intermediate results. It is an open question if it would be more efficient to run a debugger and stop the code at relevant lines. In an edit-and-run cycle of the type we met here, we frequently need to examine many numerical results, correct something, and look at all the intermediate results again. Plain `print` statements are often better suited for this massive output than the pure manual operation of a debugger, unless one writes a program to automate the interaction with the debugger.

The correct code for the implementation of the Midpoint rule is found in `integrate_v2.py`. Some readers might be frightened by all the energy it took to debug this code, but this is just the nature of programming. The experience of developing programs that finally work is very awarding¹⁰.

Refining the User Interface. We briefly mentioned that the chosen user interface, where the user can only specify n , is not particularly user friendly. We should allow f , a , b , and n to be specified on the command line. Since f is a function and the command line can only provide strings to the program, we may use the `StringFunction` object from `scitools.std` to convert a string expression for the function to be integrated to an ordinary Python function (see Chapter 3.1.4). The other parameters should be easy to retrieve from the command line if Chapter 3.2 is understood. As suggested in Chapter 3.3, we enclose the input statements in a `try-except` block, here with a specific exception type `IndexError` (because an index in `sys.argv` out of bounds is the only type of error we expect to handle):

⁹ This is not the mathematically exact value, because it involves computations of $\sin^{-1}(x)$, which is only approximately calculated by the `asin` function in the `math` module. However, the approximation error is very small ($\sim 10^{-16}$).

¹⁰ “People only become computer programmers if they’re obsessive about details, crave power over machines, and can bear to be told day after day exactly how stupid they are.” –Gregory J. E. Rawlins, computer scientist. Quote from the book “Slaves of the Machine: The Quickening of Computer Technology”, MIT Press, 1997.

```

try:
    f_formula = sys.argv[1]
    a = eval(sys.argv[2])
    b = eval(sys.argv[3])
    n = int(sys.argv[4])
except IndexError:
    print 'Usage: %s f-formula a b n' % sys.argv[0]
    sys.exit(1)

```

Note that the use of `eval` allows us to specify `a` and `b` as `pi` or `exp(5)` or another mathematical expression.

With the input above we can perform the general task of the program:

```

from scitools.std import StringFunction
f = StringFunction(f_formula)
I = integrate(f, a, b, n)
print I

```

Instead of having these test statements as a main program we follow the good habits of Chapter 3.5 and make a module with (i) the `integrate` function, (ii) a `verify` function for testing the `integrate` function's ability to exactly integrate linear functions, and (iii) a `main` function for reading data from the command line and calling `integrate` for the user's problem at hand. Any module should also have a test block, and doc strings for the module itself and all functions.

The `verify` function performs a loop over some specified `n` values and checks that the Midpoint rule integrates a linear function exactly¹¹. In the test block we can either run the `verify` function or the `main` function.

The final solution to the problem of implementing the Midpoint rule for numerical integration is now the following complete module file `integrate.py`:

```

"""Module for integrating functions by the Midpoint rule."""
from math import *
import sys

def integrate(f, a, b, n):
    """Return the integral of f from a to b with n intervals."""
    h = float(b-a)/n
    s = 0
    for i in range(1, n+1):
        s += f(a + (i-0.5)*h)
    return s*h

def verify():
    """Check that linear functions are integrated exactly."""

    def g(x):
        return p*x + q    # general linear function

    def int_g_exact(x):    # integral of g(x)
        return 0.5*p*x**2 + q*x

```

¹¹ We must be prepared for round-off errors, so “exactly” means errors less than (say) 10^{-14} .

```

a = -1.2; b = 2.8      # "arbitrary" integration limits
p = -2;   q = 10
passed = True          # True if all tests below are passed
for n in 1, 10, 100:
    I = integrate(g, a, b, n)
    I_exact = int_g_exact(b) - int_g_exact(a)
    error = abs(I_exact - I)
    if error > 1E-14:
        print 'Error=%g for n=%d' % (error, n)
        passed = False
if passed: print 'All tests are passed.'

def main():
    """
    Read f-formula, a, b, n from the command line.
    Print the result of integrate(f, a, b, n).
    """
    try:
        f_formula = sys.argv[1]
        a = eval(sys.argv[2])
        b = eval(sys.argv[3])
        n = int(sys.argv[4])
    except IndexError:
        print 'Usage: %s f-formula a b n' % sys.argv[0]
        sys.exit(1)

    from scitools.std import StringFunction
    f = StringFunction(f_formula)
    I = integrate(f, a, b, n)
    print I

if __name__ == '__main__':
    if sys.argv[1] == 'verify':
        verify()
    else:
        # compute the integral specified on the command line:
        main()

```

Here is a short demo computing $\int_0^{2\pi} (\cos(x) + \sin(x))dx$:

Terminal

```

integrate.py 'cos(x)+sin(x)' 0 2*pi 10
-3.48786849801e-16

```

E.1 Different Ways of Running Python Programs

Python programs are compiled and interpreted by another program called `python`. To run a Python program, you need to tell the operating system that your program is to be interpreted by the `python` program. This section explains various ways of doing this.

E.1.1 Executing Python Programs in IPython

The simplest and most flexible way of executing a Python program is to run it inside IPython. See Chapter 1.5.3 for a quick introduction to IPython. You start IPython either by the command `ipython` in a terminal window, or by double-clicking the IPython program icon (on Windows). Then, inside IPython, you can run a program `prog.py` by

```
In [1]: run prog.py arg1 arg2
```

where `arg1` and `arg2` are command-line arguments.

This method of running Python programs works the same way on all platforms. One additional advantage of running programs under IPython is that you can automatically enter the Python debugger if an exception is raised (see Appendix D.1. Although we advocate running Python programs under IPython in this book, you can also run them directly under specific operating systems. This is explained next for Unix, Windows, and Mac OS X.

E.1.2 Executing Python Programs on Unix

There are two ways of executing a Python program `prog.py` on Unix. The first explicitly tells which Python interpreter to use:

Terminal

```
Unix> python prog.py arg1 arg2
```

Here, `arg1` and `arg2` are command-line arguments.

There may be many Python interpreters on your computer system, usually corresponding to different versions of Python or different sets of additional packages and modules. The Python interpreter (`python`) used in the command above is the first program with the name `python` appearing in the folders listed in your `PATH` environment variable. A specific `python` interpreter, say in `/home/hpl/local/bin`, can easily be used to run a program `prog.py` in the current working folder by specifying the interpreter's complete filepath:

Terminal

```
Unix> /home/hpl/bin/python prog.py arg1 arg2
```

The other way of executing Python programs on Unix consists of just writing the name of the file:

Terminal

```
Unix> ./prog.py arg1 arg2
```

The leading `./` is needed to tell that the program is located in the current folder. You can also just write

Terminal

```
Unix> prog.py arg1 arg2
```

but then you need to have the `dot`¹ in the `PATH` variable, and this is not recommended of security reasons.

In the two latter commands there is no information on which Python interpreter to use. This information must be provided in the first line of the program, normally as

```
#!/usr/bin/env python
```

This looks like a comment line, and behaves indeed as a comment line when we run the program as `python prog.py`. However, when we run the program as `./prog.py`, the first line beginning with `#!` tells the operating system to use the program specified in the rest of the first line to interpret the program. In this example, we use the first `python` program encountered in the folders in your `PATH` variable. Alternatively, a specific `python` program can be specified as

¹ The dot acts as the name of the current folder (usually known as the current working directory). A double dot is the name of the parent folder.

```
#!/home/hpl/special/tricks/python
```

It is a good habit to always include such a first line (also called shebang line) in all Python programs and modules, but we have not done that in this book.

E.1.3 Executing Python Programs on Windows

In a DOS window you can always run a Python program by

```
DOS> python prog.py arg1 arg2
```

if `prog.py` is the name of the program, and `arg1` and `arg2` are command-line arguments. The extension `.py` can be dropped:

```
DOS> python prog arg1 arg2
```

If there are several Python installations on your system, a particular installation can be specified:

```
DOS> E:\hpl\myprogs\Python2.5.3\python prog arg1 arg2
```

Files with a certain extension can on Windows be associated with a file type, and a file type can be associated with a particular program to handle the file. For example, it is natural to associate the extension `.py` with Python programs. The corresponding program needed to interpret `.py` files is then `python.exe`. When we write just the name of the Python program file, as in

```
DOS> prog arg1 arg2
```

the file is always interpreted by the specified `python.exe` program. The details of getting `.py` files to be interpreted by `python.exe` go as follows:

```
DOS> assoc .py=PyProg
DOS> ftype PyProg=python.exe "%1" %*
```

Depending on your Python installation, such file extension bindings may already be done. You can check this with

```
DOS> assoc | find "py"
```

To see the programs associated with a file type, write `ftype name` where `name` is the name of the file type as specified by the `assoc` command. Writing `help ftype` and `help assoc` prints out more information about these commands along with examples.

One can also run Python programs by writing just the basename of the program file, i.e., `prog.py` instead of `prog.py`, if the file extension is registered in the `PATHEXT` environment variable.

Double-Clicking Python Files. The usual way of running programs on Windows is to double click on the file icon. This does not work well with Python programs without a graphical user interface. When you double click on the icon for a file `prog.py`, a DOS window is opened, `prog.py` is interpreted by some `python.exe` program, and when the program terminates, the DOS window is closed. There is usually too little time for the user to observe the output in this short-lived DOS window.

One can always insert a final statement that pauses the program by waiting for input from the user:

```
raw_input('Type CR:')
```

or

```
sys.stdout.write('Type CR:'); sys.stdin.readline()
```

The program will “hang” until the user presses the Return key. During this pause the DOS window is visible and you can watch the output from previous statements in the program.

The downside of including a final input statement is that you must always hit Return before the program terminates. This is inconvenient if the program is moved to a Unix-type machine. One possibility is to let this final input statement be active only when the program is run on Windows:

```
if sys.platform[:3] == 'win':  
    raw_input('Type CR:')
```

Python programs that have a graphical user interface can be double-clicked in the usual way if the file extension is `.pyw`.

Gnuplot Plots on Windows. Programs that call `plot` to visualize a graph with the aid of Gnuplot suffer from the same problem as described above: the plot window disappears quickly. Again, the recipe is to insert a `raw_input` call at the end of the program.

E.1.4 Executing Python Programs on Macintosh

Since a variant of Unix is used as core in the Mac OS X operating system, you can always launch a Unix terminal and use the techniques from Appendix E.1.2 to run Python programs.

E.1.5 Making a Complete Stand-Alone Executable

Python programs need a Python interpreter and usually a set of modules to be installed on the computer system. Sometimes this is inconvenient, for instance when you want to give your program to somebody who does not necessarily have Python or the required set of modules installed.

Fortunately, there are tools that can create a stand-alone executable program out of a Python program. This stand-alone executable can be run on every computer that has the same type of operating system and the same chip type. Such a stand-alone executable is a bundling of the Python interpreter and the required modules, along with your program, in a single file. Details of producing this single file are given in the book [9].

E.2 Integer and Float Division

Many languages, including C, C++, Fortran, and classical Python, interpret the division operator in two ways:

1. *Integer division*: If both operands `a` and `b` are integers, the result `a/b` is the *floor* of the mathematical result `a/b`. This yields the largest integer that `b` can be multiplied with such that the product is less than or equal to `a`. Or phrased simpler: The result of `a/b` is an integer which is “rounded down”. As an example, `5/2` becomes 2.
2. *Float division*: If one of the operands is a floating-point number or a complex number, `a/b` returns the mathematical result of the division.

Accidental integer division in places where mathematical division is needed, constitutes a very common source of errors in numerical programs.

It is often argued that in a statically typed language, where each variable is declared with a fixed type, the programmer always knows the type of the operands involved in a division expression. Therefore the programmer can determine whether an expression has the right form or not (the programmer can still oversee such errors). In a dynamically typed language, such as Python, variables can hold objects of any type. If `a` or `b` is provided by the user of the program, one can never know if both types end up as integer and `a/b` will imply integer division.

The only safe solution is to have two different operands for integer division and mathematical division. Python is currently moving in this direction. By default, `a/b` still has its original double meaning, depending on the types of operands. A new operator `//` is introduced for explicitly employing integer division. To force `a/b` to mean standard mathematical float division, one can write

```
from __future__ import division
```

This import statement must be present in every module file or script where the `/` operator always shall imply float division. Alternatively, one can run a Python program `someprogram.py` from the command line with the argument `-Qnew` to the Python interpreter:

```
Unix/DOS> python -Qnew someprogram.py
```

The future Python 3.0 is suggested to abandon integer division interpretation of `a/b`, i.e., `a/b` will always mean the relevant float division, depending on the operands (float division for `int` and `float` operands, and complex division if one of the operands is a `complex`).

Running a Python program with the `-Qwarnall` argument, say

```
Unix/DOS> python -Qwarnall someprogram.py
```

will print out a warning every time an integer division expression is encountered.

There are currently alternative ways out of the integer division problem:

1. If the operands involve an integer with fixed value, such as in `a/2`, the integer can be written as a floating-point number, as in `a/2.0` or `a/2.`, to enforce mathematical division regardless of whether `a` is integer, float, or complex.
2. If both operands are variables, as in `a/b`, the only safe way out of the problem is to write `1.0*a/b`. Note that `float(a)/b` or `a/float(b)` will work correctly from a mathematical viewpoint if `a` and `b` are of integer or floating-point type, but not if the argument to `float` is complex.

E.3 Visualizing a Program with Lumpy

Lumpy is a nice tool for graphically displaying the relations between the variables in a program. Consider the following program (inspired by Chapter 2.1.9), where we extract a sublist and modify the original list:

```
10 = [1, 4, 3]
11 = 10
12 = 11[:-1]
11[0] = 100
```

The point is that the change in 11 is reflected in 10, but not in 12, because sublists are created by taking a copy of the original list, while 11 and 10 refer to the same object. Lumpy can visually display the variables and how they relate, and thereby making it obvious that 10 and 11 refer to the same object and that 12 is a different object. To use Lumpy, some extra statements must be inserted in the program:

```
from scitools.Lumpy import Lumpy
lumpy = Lumpy()
lumpy.make_reference()
10 = [1, 4, 3]
11 = 10
12 = 11[:-1]
11[0] = 100
lumpy.object_diagram()
```

By running this program a graphical window is shown on the screen with the variables in the program, see Figure E.1a. The variables have lines to the object they point to, and inside the objects we can see the contents, i.e., the list elements in this case.

We can add some lines to the program above and make a new, additional drawing:

```
lumpy = Lumpy()
lumpy.make_reference()
n1 = 21.5
n2 = 21
l3 = [11, 12, [n1, n2]]
s1 = 'some string'
lumpy.object_diagram()
```

Figure E.1b shows the second object diagram with the additional variables.

We recommend to actively use Lumpy to make graphical illustrations of programs, especially if you search for an error and you are not 100% sure of how all variables related to each other.

E.4 Doing Operating System Tasks in Python

Python has extensive support for operating system tasks, such as file and folder management. The great advantage of doing operating system tasks in Python and not directly in the operating system is that the Python code works uniformly on Unix/Linux, Windows, and Mac (there are exceptions, but they are few). Below we list some useful operations that can be done inside a Python program or in an interactive session.

Make a folder:

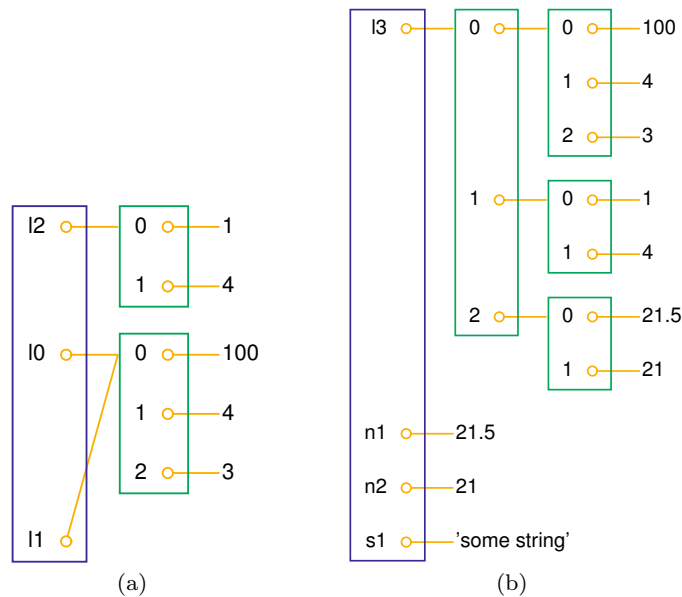


Fig. E.1 Output from Lumpy: (a) program with three lists; (b) extended program with another list, two floats, and a string.

```
import os
os.mkdir(foldername)
```

Recall that Python applies the term *directory* instead of *folder*. Ordinary files are created by the `open` and `close` functions in Python.

Make intermediate folders: Suppose you want to make a subfolder under your home folder:

```
$HOME/python/project1/temp
```

but the intermediate folders `python` and `project1` do not exist. This requires each new folder to be made separately by `os.mkdir`, or you can make all folders at once with `os.makedirs`:

```
foldername = os.path.join(os.environ['HOME'], 'python',
                           'project1', 'temp')
os.makedirs(foldername)
```

With `os.environ[var]` we can get the value of any environment variable `var` as a string.

Move to a folder:

```
origfolder = os.getcwd() # get name of current folder
os.chdir(foldername)     # move ("change directory")
...
os.chdir(origfolder)     # move back
```

Rename a file or folder:

```
os.rename(oldname, newname)
```

List files (using Unix shell wildcard notation):

```
import glob
filelist1 = glob.glob('*.py')
filelist2 = glob.glob('*[1-4]*.dat')
```

List all files and folders in a folder:

```
filelist1 = os.listdir(foldername)
filelist1 = os.listdir(os.curdir) # current folder (directory)
```

Check if a file or folder exists:

```
if os.path.isfile(filename):
    f = open(filename)
    ...

if os.path.isdir(foldername):
    filelist = os.listdir(foldername)
    ...
```

Remove files:

```
import glob
filelist = glob.glob('tmp_*.eps')
for filename in filelist:
    os.remove(filename)
```

Remove a folder and all its subfolders:

```
import shutil
shutil.rmtree(foldername)
```

It goes without saying that this command may be dangerous!

Copy a file to another file or folder:

```
shutil.copy(sourcefile, destination)
```

Copy a folder and all its subfolders:

```
shutil.copytree(sourcefolder, destination)
```

Run any operating system command:

```
cmd = 'c2f.py 21' # command to be run
failure = os.system(cmd)
if failure:
    print 'Execution of "%s" failed!\n' % cmd
```



```

        sys.exit(1)

# record output from the command:
from subprocess import Popen, PIPE
p = Popen(cmd, shell=True, stdout=PIPE)
output, errors = p.communicate()
# output contains text sent to standard output
# errors contains text sent to standard error

# process output:
for line in output.splitlines():
    # process line

# simpler recording of output on Linux/Unix:
import commands
failure, output = commands.getstatusoutput(cmd)
if failure:
    print 'Execution of "%s" failed!\n' % cmd, output
    sys.exit(1)

```

The constructions above are mainly used for running stand-alone programs. Any file or folder listing or manipulation should be done by the functionality in `os` or other modules.

Split file or folder name:

```

>>> fname = os.path.join(os.environ['HOME'], 'data', 'file1.dat')
>>> foldername, basename = os.path.split(fname)
>>> foldername
'/home/hpl/data'
>>> basename
'file1.dat'
>>> outfile = basename[:-4] + '.out'
>>> outfile
'file1.out'

```

E.5 Variable Number of Function Arguments

Arguments to Python functions are of four types:

1. positional arguments, where each argument has a name,
2. keyword arguments, where each argument has a name and a default value,
3. a variable number of positional arguments, where each argument has no name, but just a location in a list,
4. a variable number of keyword arguments, where each argument is a (name, default value) pair in a dictionary.

The corresponding general function definition can be sketched as

```
def f(pos1, pos2, key1=val1, key2=val2, *args, **kwargs):
```

Here, `pos1` and `pos2` are positional arguments, `key1` and `key2` are keyword arguments, `args` is a tuple holding a variable number of positional

arguments, and `kwargs` is a dictionary holding a variable number of keyword arguments. This appendix describes how to program with the `args` and `kwargs` variables and why these are handy in many situations.

E.5.1 Variable Number of Positional Arguments

Let us start by making a function that takes an arbitrary number of arguments and computes their sum:

```
>>> def add(*args):
...     print 'args:', args
...     return sum(args)
...
>>> add(1)
args: (1,)
1
>>> add(1,5,10)
args: (1, 5, 10)
16
```

We observe that `args` is a tuple and that all the arguments we provide in a call to `add` are stored in `args`.

Combination of ordinary positional arguments and a variable number of arguments is allowed, but the `*args` argument must appear after the ordinary positional arguments, e.g.,

```
def f(pos1, pos2, pos3, *args):
```

In each call to `f` we must provide at least three arguments. If more arguments are supplied in the call, these are collected in the `args` tuple inside the `f` function.

Example. Chapter 7.1.1 describes functions with parameters, e.g., $y(t; v_0) = v_0 t - \frac{1}{2}gt^2$, or the more general case $f(x; p_1, \dots, p_n)$. The Python implementation of such functions can take both the independent variable and the parameters as arguments: `y(t, v0)` and `f(x, p1, p2, ..., pn)`. Suppose that we have a general library routine that operates on functions of one variable. Relevant operations can be numerical differentiation, integration, or root finding. A simple example is a numerical differentiation function

```
def diff(f, x, h):
    return (f(x+h) - f(x))/h
```

This `diff` function cannot be used with functions `f` that take more than one argument, e.g., passing an `y(t, v0)` function as `f` leads to the exception

```
TypeError: y() takes exactly 2 arguments (1 given)
```

Chapter 7.1.1 provides a solution to this problem where `y` becomes a class instance. Here we can describe an alternative solution that allows our `y(t, v0)` function to be used as is.

The idea is that we pass additional arguments for the parameters in the `f` function *through* the `diff` function. That is, we view the `f` function as `f(x, *f_prms)`. Our `diff` routine can then be written as

```
def diff(f, x, h, *f_prms):
    print 'x:', x, 'h:', h, 'f_prms:', f_prms
    return (f(x+h, *f_prms) - f(x, *f_prms))/h
```

Before explaining this function in detail, we “prove” that it works in an example:

```
def y(t, v0):
    g = 9.81; return v0*t - 0.5*g*t**2

dydt = diff(y, 0.1, 1E-9, 3) # t=0.1, h=1E-9, v0=3
```

The output from the call to `diff` becomes

```
x: 0.1 h: 1e-09 f_prms: (3,)
```

The point is that the `v0` parameter, which we want to pass on to our `y` function, is now stored in `f_prms`. Inside the `diff` function, calling

```
f(x, *f_prms)
```

is the same as if we had written

```
f(x, f_prms[0], f_prms[1], ...)
```

That is, `*f_prms` in a call takes all the values in the tuple `*f_prms` and places them after each other as positional arguments. In the present example with the `y` function, `f(x, *f_prms)` implies `f(x, f_prms[0])`, which for the current set of argument values in our example becomes a call `y(0.1, 3)`.

For a function with many parameters,

```
def G(x, t, A, a, w):
    return A*exp(-a*t)*sin(w*x)
```

the output from

```
dGdx = diff(G, 0.5, 1E-9, 0, 1, 0.6, 100)
```

becomes

```
x: 0.5 h: 1e-09 f_prms: (0, 1, 1.5, 100)
```

We pass here the arguments `t`, `A`, `a`, and `w`, in that sequence, as the last four arguments to `diff`, and all the values are stored in the `f_prms` tuple.

The `diff` function also works for a plain function `f` with one argument:

```
from math import sin
mycos = diff(sin, 0, 1E-9)
```

In this case, `*f_prms` becomes an empty tuple, and a call like `f(x, *f_prms)` is just `f(x)`.

The use of a variable set of arguments for sending problem-specific parameters “through” a general library function, as we have demonstrated here with the `diff` function, is perhaps the most frequent use of `*args`-type arguments.

E.5.2 Variable Number of Keyword Arguments

A simple test function

```
>>> def test(**kwargs):
...     print kwargs
```

exemplifies that `kwargs` is a dictionary inside the `test` function, and that we can pass any set of keyword arguments to `test`, e.g.,

```
>>> test(a=1, q=9, method='Newton')
{'a': 1, 'q': 9, 'method': 'Newton'}
```

We can combine an arbitrary set of positional and keyword arguments, provided all the keyword arguments appear at the end of the call:

```
>>> def test(*args, **kwargs):
...     print args, kwargs
...
>>> test(1,3,5,4,a=1,b=2)
(1, 3, 5, 4) {'a': 1, 'b': 2}
```

From the output we understand that all the arguments in the call where we provide a name and a value are treated as keyword arguments and hence placed in `kwargs`, while all the remaining arguments are positional and placed in `args`.

Example. We may extend the example in Appendix E.5.1 to make use of a variable number of keyword arguments instead of a variable number of positional arguments. Suppose all functions with parameters in addition to an independent variable take the parameters as keyword arguments. For example,

```
def y(t, v0=1):
    g = 9.81; return v0*t - 0.5*g*t**2
```

In the `diff` function we transfer the parameters in the `f` function as a set of keyword arguments `**f_prms`:

```
def diff(f, x, h=1E-10, **f_prms):
    print 'x:', x, 'h:', h, 'f_prms:', f_prms
    return (f(x+h, **f_prms) - f(x, **f_prms))/h
```

In general, the `**f_prms` argument in a call

```
f(x, **f_prms)
```

implies that all the key-value pairs in `**f_prms` are provided as keyword arguments:

```
f(x, key1=f_prms[key1], key2=f_prms[key2], ...)
```

In our special case with the `y` function and the call

```
dydt = diff(y, 0.1, h=1E-9, v0=3)
```

`f(x, **f_prms)` becomes `y(0.1, v0=3)`. The output from `diff` is now

```
x: 0.1 h: 1e-09 f_prms: {'v0': 3}
```

showing explicitly that our `v0=3` in the call to `diff` is placed in the `f_prms` dictionary.

The `G` function from Appendix E.5.1 can also have its parameters as keyword arguments:

```
def G(x, t=0, A=1, a=1, w=1):
    return A*exp(-a*t)*sin(w*x)
```

We can now make the call

```
dGdx = diff(G, 0.5, h=1E-9, t=0, A=1, w=100, a=1.5)
```

and view the output from `diff`,

```
x: 0.5 h: 1e-09 f_prms: {'A': 1, 'a': 1.5, 't': 0, 'w': 100}
```

to see that all the parameters get stored in `f_prms`. The `h` parameter can be placed anywhere in the collection of keyword arguments, e.g.,

```
dGdx = diff(G, 0.5, t=0, A=1, w=100, a=1.5, h=1E-9)
```

We can allow the `f` function of one variable and a set of parameters to have the general form `f(x, *f_args, **f_kwargs)`. That is, the parameters can either be positional or keyword arguments. The `diff` function must take the arguments `*f_args` and `**f_kwargs` and transfer these to `f`:

```
def diff(f, x, h=1E-10, *f_args, **f_kwargs):
    print f_args, f_kwargs
    return (f(x+h, *f_args, **f_kwargs) -
            f(x, *f_args, **f_kwargs))/h
```

This `diff` function gives the writer of an `f` function full freedom to choose positional and/or keyword arguments for the parameters. Here is an example of the `G` function where we let the `t` parameter be positional and the other parameters be keyword arguments:

```
def G(x, t, A=1, a=1, w=1):
    return A*exp(-a*t)*sin(w*x)
```

A call

```
dGdx = diff(G, 0.5, 1E-9, 0, A=1, w=100, a=1.5)
```

gives the output

```
(0,) {'A': 1, 'a': 1.5, 'w': 100}
```

showing that `t` is put in `f_args` and transferred as positional argument to `G`, while `A`, `a`, and `w` are put in `f_kwargs` and transferred as keyword arguments. We remark that in the last call to `diff`, `h` and `t` *must* be treated as positional arguments, i.e., we cannot write `h=1E-9` and `t=0` unless *all* arguments in the call are on the `name=value` form.

In the case we use both `*f_args` and `**f_kwargs` arguments in `f` and there is no need for these arguments, `*f_args` becomes an empty tuple and `**f_kwargs` becomes an empty dictionary. The example

```
mycos = diff(sin, 0)
```

shows that the tuple and dictionary are indeed empty since `diff` just prints out

```
() {}
```

Therefore, a variable set of positional and keyword arguments can be incorporated in a general library function such as `diff` without any disadvantage, just the benefit that `diff` works with different types `f` functions: parameters as global variables, parameters as additional positional arguments, parameters as additional keyword arguments, or parameters as instance variables (Chapter 7.1.2).

The program `varargs1.py` in the `appendix` folder implements the examples in this appendix.

E.6 Evaluating Program Efficiency

E.6.1 Making Time Measurements

Time is not just “time” on a computer. The *elapsed time* or *wall clock time* is the same time as you can measure on a watch or wall clock, while *CPU time* is the amount of time the program keeps the central processing unit busy. The *system time* is the time spent on operating

system tasks like I/O. The concept *user time* is the difference between the CPU and system times. If your computer is occupied by many concurrent processes, the CPU time of your program might be very different from the elapsed time.

The time Module. Python has a `time` module with some useful functions for measuring the elapsed time and the CPU time:

```
import time
e0 = time.time()      # elapsed time since the epoch
c0 = time.clock()     # total CPU time spent in the program so far
<do tasks...>
elapsed_time = time.time() - e0
cpu_time = time.clock() - c0
```

The term *epoch* means initial time (`time.time()` would return 0), which is 00:00:00 January 1, 1970. The `time` module also has numerous functions for nice formatting of dates and time, and the more recent `datetime` module has more functionality and an improved interface. Although the timing has a finer resolution than seconds, one should construct test cases that last some seconds to obtain reliable results.

The timeit Module. To measure the efficiency of a certain set of statements or an expression, the code should be run a large number of times so the overall CPU-time is of order seconds. The `timeit` module has functionality for running a code segment repeatedly. Below is an illustration of `timeit` for comparing the efficiency `sin(1.2)` versus `math.sin(1.2)`:

```
>>> import timeit
>>> t = timeit.Timer('sin(1.2)', setup='from math import sin')
>>> t.timeit(10000000)    # run 'sin(1.2)' 10000000 times
11.830688953399658
>>> t = timeit.Timer('math.sin(1.2)', setup='import math')
>>> t.timeit(10000000)
16.234833955764771
```

The first argument to the `Timer` constructor is a string containing the code to execute repeatedly, while the second argument is the necessary code for initialization. From this simple test we see that `math.sin(1.2)` runs almost 40 percent slower than `sin(1.2)`!

If you want to time a function, say `f`, defined in the same program as where you have the `timeit` call, the setup procedure must import `f` and perhaps other variables from the program, as exemplified in

```
t = timeit.Timer('f(a,b)', setup='from __main__ import f, a, b')
```

Here, `f`, `a`, and `b` are names initialized in the main program. Another example is found in `src/random/smart_power.py`.

Hardware Information. Along with CPU-time measurements it is often convenient to print out information about the hardware on which the

experiment was done. Python has a module `platform` with information on the current hardware. The function `scitools.misc.hardware_info` applies the `platform` module to extract relevant hardware information. A sample call is

```
>>> import scitools.misc, pprint
>>> pprint.pprint(scitools.misc.hardware_info())
{'cpuinfo':
  {'CPU speed': '1196.170 Hz',
   'CPU type': 'Mobile Intel(R) Pentium(R) III CPU - M 1200MHz',
   'cache size': '512 KB',
   'vendor ID': 'GenuineIntel'},
 'identifier': 'Linux-2.6.12-i686-with-debian-testing-unstable',
 'python build': ('r25:409', 'Feb 27 2007 19:35:40'),
 'python version': '2.5.0',
 'uname': ('Linux',
           'ubuntu',
           '2.6.12',
           '#1 Fri Nov 25 10:58:24 CET 2005',
           'i686',
           '')}
```

E.6.2 Profiling Python Programs

A profiler computes the time spent in the various functions of a program. From the timings a ranked list of the most time-consuming functions can be created. This is an indispensable tool for detecting bottlenecks in the code, and you should always perform a profiling before spending time on code optimization. The golden rule is to first write an easy-to-understand program, then verify it, then profile it, and then think about optimization².

Python comes with two profilers implemented in the `profile` and `hotshot` modules, respectively. The Python Library Reference has a good introduction to profiling in Python (Chapter 10: “The Python Profiler”). The results produced by the two alternative modules are normally processed by a special statistics utility `pstats` developed for analyzing profiling results. The usage of the `profile`, `hotshot`, and `pstats` modules is straightforward, but somewhat tedious so SciTools comes with a command `scitools profiler` that allows you to profile any program (say) `m.py` by writing

Terminal

```
Unix/DOS> scitools profiler m.py c1 c2 c3
```

Here, `c1`, `c2`, and `c3` are command-line arguments to `m.py`.

We refer to the Python Library Reference for detailed information on how to interpret the output. A sample output might read

² “Premature optimization is the root of all evil.” –Donald Knuth, computer scientist, 1938–.

1082 function calls (728 primitive calls) in 17.890 CPU seconds

Ordered by: internal time

List reduced from 210 to 20 due to restriction <20>

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
5	5.850	1.170	5.850	1.170	m.py:43(loop1)
1	2.590	2.590	2.590	2.590	m.py:26(empty)
5	2.510	0.502	2.510	0.502	m.py:32(myfunc2)
5	2.490	0.498	2.490	0.498	m.py:37(init)
1	2.190	2.190	2.190	2.190	m.py:13(run1)
6	0.050	0.008	17.720	2.953	funcs.py:126(timer)
...					

In this test, `loop1` is the most expensive function, using 5.85 seconds, which is to be compared with 2.59 seconds for the next most time-consuming function, `empty`. The `tottime` entry is the total time spent in a specific function, while `cumtime` reflects the total time spent in the function and all the functions it calls.

The CPU time of a Python program typically increases with a factor of about five when run under the administration of the `profile` module. Nevertheless, the relative CPU time among the functions are probably not much affected by the profiler overhead.

References

- [1] D. Beazley. *Python Essential Reference*. SAMS, 2nd edition, 2001.
- [2] J. E. Grayson. *Python and Tkinter Programming*. Manning, 2000.
- [3] D. Harms and K. McDonald. *The Quick Python Book*. Manning, 1999.
- [4] D. E. Knuth. Theory and practice. *EATCS Bull.*, 27:14–21, 1985.
- [5] H. P. Langtangen. *Python Scripting for Computational Science*, volume 3 of *Texts in Computational Science and Engineering*. Springer, third edition, 2009.
- [6] L. S. Lerner. *Physics for Scientists and Engineers*. Jones and Barlett, 1996.
- [7] M. Lutz. *Programming Python*. O'Reilly, second edition, 2001.
- [8] M. Lutz and D. Ascher. *Learning Python*. O'Reilly, 1999.
- [9] A. Martelli. *Python in a Nutshell*. O'Reilly, 2003.
- [10] J. D. Murray. *Mathematical Biology I: An Introduction*. Springer, 3rd edition, 2007.
- [11] F. M. White. *Fluid Mechanics*. McGraw-Hill, 2nd edition, 1986.

Index

- `**kwargs`, 681
- `*=`, 54
- `*args`, 679
- `+=`, 54
- `-=`, 54
- `/=`, 54
- `\n`, 13

- allocate, 177
- animate, 455
- API, 384
- `aplotter` (from `scitools`), 198
- `append` (list), 57
- application, 14
- application programming interface, 384
- `arange` (from `numpy`), 211, 212
- `array` (from `numpy`), 176
- `array` (datatype), 176
- array computing, 176
- array shape, 212, 213
- array slicing, 177
- `asarray` (from `numpy`), 209
- attribute (class), 342
- average, 422

- base class, 480
- `bin` (histogram), 420
- binomial distribution, 166
- bits, 275
- blank lines in files, 294
- blanks, 17

- body of a function, 72
- boolean expressions, 55
- `break`, 272, 307
- bytes, 275

- callable function, 357
- callable objects, 357
- callback function, 635
- check an object's type, 28, 210, 385, 483
- check file/folder existence (in Python), 677
- class hierarchy, 480
- class relationship
 - derived class, 480
 - has-a, 485
 - inheritance, 480
 - is-a, 485
 - subclass, 480
 - superclass, 480
- closure, 497
- `cmath` module, 33
- command-line arguments, 127
- `commands` module, 677
- comments, 10
- comparing
 - floating-point numbers, 113
 - objects, 113
 - real numbers, 113
- complex numbers, 31
- `concatenate` (from `numpy`), 255
- console (terminal) window, 4

- constructor (class), 341
- convert program, 454
- copy files (in Python), 677
- copy folders (in Python), 677
- CPU time measurements, 683
- cumulative sum, 468
- curve plotting, 179

- datetime module, 238, 684
- debugger tour, 651
- del, 57
- delete files (in Python), 192, 412, 677
- delete folders (in Python), 677
- derived class, 480
- dictionary, 278
 - nested, 284
- dictionary functionality, 318
- difference equations, 236
 - nonlinear, 252
- differential equations, 372, 508, 605
- dir function, 388
- directory, 1, 4, 676
- doc strings, 83
- dtype, 209
- duck typing, 386
- dynamic binding, 490
- dynamic typing, 386

- Easyviz, 180
- editor, 3
- efficiency, 683
- efficiency measure, 447
- elapsed time, 683
- enumerate function, 63
- environment variables, 676
- eval function, 121, 491
- event loop, 141
- except, 133
- Exception, 138
- exceptions, 133
- execute programs (from Python), 677
- execute Python program, 7, 29, 669
- expression, 16

- factorial (from scitools), 106
- factory function, 492

- find (string method), 292
- first-order ODEs, 617
- float_eq, 114
- Fourier series, 47
- function arguments
 - keyword, 81
 - named, 81
 - positional, 81
- function body, 72
- function header, 72
- function inside function, 515
- functional programming, 497

- Gaussian function, 44
- getopt module, 150
- glob.glob function, 412, 677
- global, 74
- globals function, 73, 206, 640
- grid, 574

- has-a class relationship, 485
- Heaviside function, 108
- heterogeneous lists, 175
- histogram (normalized), 420

- Idle, 4
- immutable objects, 280, 367
- in-place array arithmetics, 207
- IndexError, 134, 135
- information hiding, 384
- initial condition, 236, 606, 630
- input (data), 17
- insert (list), 57
- instance (class), 342
- integer random numbers, 424
- interactive sessions
 - IPython, 29
 - session recording (logging), 45
 - standard Python shell, 26
- interval arithmetics, 393
- IPython, 29
- is, 80
- is-a class relationship, 485
- isdigit (string method), 294
- iseq, 211
- iseq (from scitools), 211

- `isinstance` function, 92, 210, 385, 483
- `isspace` (string method), 294
- `join` (string method), 295
- keys (dictionaries), 279
- keyword arguments, 81, 678
- lambda functions, 87, 90
- least squares approximation, 324
- `len` (list), 57
- line break, 13
- `linspace` (from `numpy`), 177, 212
- list comprehension, 63, 65
- list files (in Python), 677
- list functionality, 92
- list, nested, 64
- lists, 56
- logical expressions, 55
- loops, 52
- `lower` (string method), 294
- `lstrip` (string method), 295
- Mac OS X, 18
- main program, 86
- make a folder (in Python), 675
- making graphs, 179
- making movie, 454
- `math` module, 23
- mean, 422
- `mean` (from `numpy`), 423
- measure time in programs, 447
- mesh, 574
- method (class), 58
- method (in class), 342
- `mod` function, 423
- module folders, 149
- modules, 141
- Monte Carlo integration, 443
- Monte Carlo simulation, 433
- move to a folder (in Python), 676
- multiple inheritance, 550
- mutable objects, 280, 367
- named arguments, 81
- `NameError`, 135
- namespace, 350
- nested dictionaries, 284
- nested lists, 64
- nested loops, 69
- newline character (line break), 13
- Newton's method, 247, 359
- `None`, 80
- nonlinear difference equations, 252
- normally distributed random numbers, 423
- `not`, 55
- Numerical Python, 176
- `NumPy`, 176
- `numpy`, 176
- `numpy.lib.scimath` module, 33
- object-based programming, 479
- object-oriented programming, 479
- objects, 20
- operating system (OS), 18
- optimization of Python code, 685
- option-value pairs (command line), 130
- `optparse` module, 150
- ordinary differential equations, 614
- `os` module, 675
- `os.chdir` function, 676
- `os.listdir` function, 677
- `os.makedirs` function, 676
- `os.mkdir` function, 675
- `os.pardir`, 149
- `os.path.isdir` function, 677
- `os.path.isfile` function, 677
- `os.path.join` function, 149, 676
- `os.path.split` function, 678
- `os.remove` function, 192, 412, 677
- `os.rename` function, 676
- `os.system` function, 677
- oscillating systems, 521, 615, 626, 632
- output (data), 17
- overloading (of methods), 502
- parent class, 480
- `pass`, 383
- `plot` (from `scitools`), 181
- plotting, 179
- Poisson distribution, 166
- polymorphism, 502
- positional arguments, 81, 678

- `pprint.pformat`, 66
- `pprint.pprint`, 65
- `pprint2` (from `scitools`), 66
- pretty print, 65
- private attributes (class), 384
- probability, 432
- `profiler.py`, 685
- profiling, 685
- protected attributes (class), 367, 384
- pydoc program, 98
- pyreport program, 42, 227
- `r_`, 212
- `raise`, 137
- `random` (from `numpy`), 421
- random module, 418
- random numbers, 417
 - histogram, 420
 - integers, 424
 - integration, 443
 - Monte Carlo simulation, 433
 - normal distribution, 423
 - random walk, 448
 - statistics, 422
 - uniform distribution, 419
 - vectorization, 421
- random walk, 448
- `range` function, 61
- `raw_input` function, 120
- refactoring, 157, 375
- regular expressions, 308
- remove files (in Python), 192, 412, 677
- remove folders (in Python), 677
- rename file/folder (in Python), 676
- `replace` (string method), 293
- resolution (mesh), 574
- `round` function, 28
- round-off errors, 25
- rounding float to integer, 28
- `rstrip` (string method), 295
- run programs (from Python), 677
- run Python program, 7, 29, 669
- scalar (math. quantity), 172
- scalar code, 178
- scalar differential equations, 615
- scaling, 243
- `scitools.pprint2` module, 66
- `scitools.pprint2.pprint`, 66
- `scitools.std`, 180
- search for module files, 149
- Secant method, 264
- second-order ODEs, 521, 617, 630
- seed, 418
- `seq` (from `scitools`), 211
- sequence (data type), 91
- sequence (mathematical), 235
- shape (of an array), 212, 213
- `shutil.copy` function, 677
- `shutil.copypath` function, 677
- `shutil.rmtree` function, 677
- slicing, 66, 292
- source code, 14
- special methods (in classes), 356
- `split` (string method), 293
- split filename, 678
- spread of a disease (model), 619
- standard deviation, 422
- standard error, 311
- standard input, 310
- standard output, 310
- statements, 15
- static class attributes, 390
- static class methods, 390
- static class variables, 390
- static typing, 386
- `std` (from `numpy`), 423
- `str2obj` (from `scitools`), 150
- string, 11
 - case change, 294
 - joining list elements, 295
 - searching, 292
 - splitting, 293
 - stripping leading/trailing blanks, 295
 - substitution, 293
 - substrings, 292
 - testing for number, 294
- string slicing, 292
- `StringFunction` (from `scitools`), 126
- `strip` (string method), 295
- strong typing, 386

- subarrays, 177
- subclass, 480
- sublist, 66
- subprocess module, 677
- substitution (in text), 293
- substrings, 292
- superclass, 480
- syntax, 16
- `SyntaxError`, 136
- `sys` module, 127
- `sys.argv`, 127
- `sys.exit` function, 133
- `sys.path`, 149
- `sys.stderr`, 311
- `sys.stdin`, 310
- `sys.stdout`, 310
- system time, 683
- systems of differential equations, 615
- terminal window, 4
- test block (in module files), 144
- time
 - CPU, 683
 - elapsed, 683
 - system, 683
 - user, 683
- `time` module, 93, 115, 447, 449, 684
- `timeit` module, 684
- timing utilities, 683
- triple-quoted strings, 13
- `try`, 133
- tuples, 70
- `type` function, 28, 210
- type conversion, 28
- `TypeError`, 136
- UML class diagram, 341
- uniformly distributed random numbers, 419
- Unix, 18
- `upper` (string method), 294
- user (of a program), 17
- user time, 683
- user-defined datatype (class), 342
- using a debugger, 651
- `_v1` (version numbering), 19
- `ValueError`, 134, 135
- `var` (from `numpy`), 423
- variable no. of function arguments, 679
- variance, 422
- vector computing, 171
- vectorization, 176, 178
- vectorized drawing of random numbers, 421
- vectors, 170
- weak typing, 386
- whitespace, 17, 294
- widgets, 140
- Windows, 18
- `wrap2callable` function, 404
- wrapper code, 361
- `xrange` function, 111, 178, 426
- `ZeroDivisionError`, 136
- `zeros` (from `numpy`), 176
- `zip` function, 63

Editorial Policy

§1. Textbooks on topics in the field of computational science and engineering will be considered. They should be written for courses in CSE education. Both graduate and undergraduate textbooks will be published in TCSE. Multidisciplinary topics and multidisciplinary teams of authors are especially welcome.

§2. Format: Only works in English will be considered. They should be submitted in camera-ready form according to Springer-Verlag's specifications.

Electronic material can be included if appropriate. Please contact the publisher.

Technical instructions and/or \TeX macros are available via

<http://www.springer.com/authors/book+authors?SGWID=0-154102-12-417900-0>

§3. Those considering a book which might be suitable for the series are strongly advised to contact the publisher or the series editors at an early stage.

General Remarks

TCSE books are printed by photo-offset from the master-copy delivered in camera-ready form by the authors. For this purpose Springer-Verlag provides technical instructions for the preparation of manuscripts. See also *Editorial Policy*.

Careful preparation of manuscripts will help keep production time short and ensure a satisfactory appearance of the finished book.

The following terms and conditions hold:

Regarding free copies and royalties, the standard terms for Springer mathematics monographs and textbooks hold. Please write to martin.peters@springer.com for details.

Authors are entitled to purchase further copies of their book and other Springer books for their personal use, at a discount of 33,3% directly from Springer-Verlag.

Series Editors

Timothy J. Barth
NASA Ames Research Center
NAS Division
Moffett Field, CA 94035, USA
e-mail: barth@nas.nasa.gov

Michael Griebel
Institut für Numerische Simulation
der Universität Bonn
Wagelerstr. 6
53115 Bonn, Germany
e-mail: griebel@ins.uni-bonn.de

David E. Keyes
Department of Applied Physics
and Applied Mathematics
Columbia University
200 S. W. Mudd Building
500 W. 120th Street
New York, NY 10027, USA
e-mail: david.keyes@columbia.edu

Risto M. Nieminen
Laboratory of Physics
Helsinki University of Technology
02150 Espoo, Finland
e-mail: rni@fyslab.hut.fi

Dirk Roose
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven-Heverlee, Belgium
e-mail: dirk.roose@cs.kuleuven.ac.be

Tamar Schlick
Department of Chemistry
Courant Institute of Mathematical
Sciences
New York University
and Howard Hughes Medical Institute
251 Mercer Street
New York, NY 10012, USA
e-mail: schlick@nyu.edu

Editor at Springer: Martin Peters
Springer-Verlag, Mathematics Editorial IV
Tiergartenstrasse 17
D-69121 Heidelberg, Germany
Tel.: +49 (6221) 487-8185
Fax: +49 (6221) 487-8355
e-mail: martin.peters@springer.com

Texts in Computational Science and Engineering

1. H.P. Langtangen, *Computational Partial Differential Equations*. Numerical Methods and Diffpack Programming. 2nd Edition
2. A. Quarteroni, F. Saleri, *Scientific Computing with MATLAB and Octave*. 2nd Edition
3. H.P. Langtangen, *Python Scripting for Computational Science*. 3rd Edition
4. H. Gardner, G. Manduchi, *Design Patterns for e-Science*.
5. M. Griebel, S. Knapek, G. Zumbusch, *Numerical Simulation in Molecular Dynamics*.
6. H.P. Langtangen, *A Primer on Scientific Programming with Python*.

For further information on these books please have a look at our mathematics catalogue at the following URL:
www.springer.com/series/5151

Monographs in Computational Science and Engineering

1. J. Sundnes, G.T. Lines, X. Cai, B.F. Nielsen, K.-A. Mardal, A. Tveito, *Computing the Electrical Activity in the Heart*.

For further information on these books please have a look at our mathematics catalogue at the following URL:
www.springer.com/series/7417

Lecture Notes in Computational Science and Engineering

1. D. Funaro, *Spectral Elements for Transport-Dominated Equations*.
2. H.P. Langtangen, *Computational Partial Differential Equations*. Numerical Methods and Diffpack Programming.
3. W. Hackbusch, G. Wittum (eds.), *Multigrid Methods V*.
4. P. Deuffhard, J. Hermans, B. Leimkuhler, A.E. Mark, S. Reich, R.D. Skeel (eds.), *Computational Molecular Dynamics: Challenges, Methods, Ideas*.
5. D. Kröner, M. Ohlberger, C. Rohde (eds.), *An Introduction to Recent Developments in Theory and Numerics for Conservation Laws*.

6. S. Turek, *Efficient Solvers for Incompressible Flow Problems*. An Algorithmic and Computational Approach.
7. R. von Schwerin, *Multi Body System SIMulation*. Numerical Methods, Algorithms, and Software.
8. H.-J. Bungartz, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing*.
9. T.J. Barth, H. Deconinck (eds.), *High-Order Methods for Computational Physics*.
10. H.P. Langtangen, A.M. Bruaset, E. Quak (eds.), *Advances in Software Tools for Scientific Computing*.
11. B. Cockburn, G.E. Karniadakis, C.-W. Shu (eds.), *Discontinuous Galerkin Methods*. Theory, Computation and Applications.
12. U. van Rienen, *Numerical Methods in Computational Electrodynamics*. Linear Systems in Practical Applications.
13. B. Engquist, L. Johnsson, M. Hammill, F. Short (eds.), *Simulation and Visualization on the Grid*.
14. E. Dick, K. Rienslagh, J. Vierendeels (eds.), *Multigrid Methods VI*.
15. A. Frommer, T. Lippert, B. Medeke, K. Schilling (eds.), *Numerical Challenges in Lattice Quantum Chromodynamics*.
16. J. Lang, *Adaptive Multilevel Solution of Nonlinear Parabolic PDE Systems*. Theory, Algorithm, and Applications.
17. B.I. Wohlmuth, *Discretization Methods and Iterative Solvers Based on Domain Decomposition*.
18. U. van Rienen, M. Günther, D. Hecht (eds.), *Scientific Computing in Electrical Engineering*.
19. I. Babuška, P.G. Ciarlet, T. Miyoshi (eds.), *Mathematical Modeling and Numerical Simulation in Continuum Mechanics*.
20. T.J. Barth, T. Chan, R. Haimes (eds.), *Multiscale and Multiresolution Methods*. Theory and Applications.
21. M. Breuer, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing*.
22. K. Urban, *Wavelets in Numerical Simulation*. Problem Adapted Construction and Applications.
23. L.F. Pavarino, A. Toselli (eds.), *Recent Developments in Domain Decomposition Methods*.
24. T. Schlick, H.H. Gan (eds.), *Computational Methods for Macromolecules: Challenges and Applications*.
25. T.J. Barth, H. Deconinck (eds.), *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*.
26. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations*.
27. S. Müller, *Adaptive Multiscale Schemes for Conservation Laws*.
28. C. Carstensen, S. Funken, W. Hackbusch, R.H.W. Hoppe, P. Monk (eds.), *Computational Electromagnetics*.
29. M.A. Schweitzer, *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*.
30. T. Biegler, O. Ghattas, M. Heinkenschloss, B. van Bloemen Waanders (eds.), *Large-Scale PDE-Constrained Optimization*.
31. M. Ainsworth, P. Davies, D. Duncan, P. Martin, B. Rynne (eds.), *Topics in Computational Wave Propagation*. Direct and Inverse Problems.
32. H. Emmerich, B. Nestler, M. Schreckenberg (eds.), *Interface and Transport Dynamics*. Computational Modelling.
33. H.P. Langtangen, A. Tveito (eds.), *Advanced Topics in Computational Partial Differential Equations*. Numerical Methods and Diffpack Programming.

34. V. John, *Large Eddy Simulation of Turbulent Incompressible Flows*. Analytical and Numerical Results for a Class of LES Models.
35. E. Bänsch (ed.), *Challenges in Scientific Computing – CISC 2002*.
36. B.N. Khoromskij, G. Wittum, *Numerical Solution of Elliptic Differential Equations by Reduction to the Interface*.
37. A. Iske, *Multiresolution Methods in Scattered Data Modelling*.
38. S.-I. Niculescu, K. Gu (eds.), *Advances in Time-Delay Systems*.
39. S. Attinger, P. Koumoutsakos (eds.), *Multiscale Modelling and Simulation*.
40. R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Wildlund, J. Xu (eds.), *Domain Decomposition Methods in Science and Engineering*.
41. T. Plewa, T. Linde, V.G. Weirs (eds.), *Adaptive Mesh Refinement – Theory and Applications*.
42. A. Schmidt, K.G. Siebert, *Design of Adaptive Finite Element Software*. The Finite Element Toolbox ALBERTA.
43. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations II*.
44. B. Engquist, P. Lötstedt, O. Runborg (eds.), *Multiscale Methods in Science and Engineering*.
45. P. Benner, V. Mehrmann, D.C. Sorensen (eds.), *Dimension Reduction of Large-Scale Systems*.
46. D. Kressner, *Numerical Methods for General and Structured Eigenvalue Problems*.
47. A. Boriçi, A. Frommer, B. Joó, A. Kennedy, B. Pendleton (eds.), *QCD and Numerical Analysis III*.
48. F. Graziani (ed.), *Computational Methods in Transport*.
49. B. Leimkuhler, C. Chipot, R. Elber, A. Laaksonen, A. Mark, T. Schlick, C. Schütte, R. Skeel (eds.), *New Algorithms for Macromolecular Simulation*.
50. M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.), *Automatic Differentiation: Applications, Theory, and Implementations*.
51. A.M. Bruaset, A. Tveito (eds.), *Numerical Solution of Partial Differential Equations on Parallel Computers*.
52. K.H. Hoffmann, A. Meyer (eds.), *Parallel Algorithms and Cluster Computing*.
53. H.-J. Bungartz, M. Schäfer (eds.), *Fluid–Structure Interaction*.
54. J. Behrens, *Adaptive Atmospheric Modeling*.
55. O. Widlund, D. Keyes (eds.), *Domain Decomposition Methods in Science and Engineering XVI*.
56. S. Kassinos, C. Langer, G. Iaccarino, P. Moin (eds.), *Complex Effects in Large Eddy Simulations*.
57. M. Griebel, M. A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations III*.
58. A.N. Gorban, B. Kégl, D.C. Wunsch, A. Zinovyev (eds.), *Principal Manifolds for Data Visualization and Dimension Reduction*.
59. H. Ammari (ed.), *Modeling and Computations in Electromagnetics: A Volume Dedicated to Jean-Claude Nédélec*.
60. U. Langer, M. Discacciati, D. Keyes, O. Widlund, W. Zulehner (eds.), *Domain Decomposition Methods in Science and Engineering XVII*.
61. T. Mathew, *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations*.
62. F. Graziani (ed.), *Computational Methods in Transport: Verification and Validation*.

63. M. Bebendorf, *Hierarchical Matrices. A Means to Efficiently Solve Elliptic Boundary Value Problems*.
64. C.H. Bischof, H.M. Bücker, P. Hovland, U. Naumann, J. Utke (eds.), *Advances in Automatic Differentiation*.
65. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations IV*.
66. B. Engquist, P. Lötstedt, O. Runborg (eds.), *Multiscale Modeling and Simulation in Science*.
67. I.H. Tuncer, Ü. Gülcät, D.R. Emerson, K. Matsuno (eds.), *Parallel Computational Fluid Dynamics*.
68. S. Yip, T. Diaz de la Rubia (eds.), *Scientific Modeling and Simulations*.
69. A. Hegarty, N. Kopteva, E. O’Riordan, M. Stynes (eds.), *BAIL 2008 – Boundary and Interior Layers*.
70. M. Bercovier, M.J. Gander, R. Kornhuber, O. Widlund (eds.), *Domain Decomposition Methods in Science and Engineering XVIII*.

For further information on these books please have a look at our mathematics catalogue at the following URL:
www.springer.com/series/3527