```
a = sqrt(4*p + c)
print 'a =', a
```

A complete program has an additional vertical line to the left:

```
C = 21
F = (9.0/5)*C + 32
print F
```

As a reader of this book, you may wonder if a code shown is a complete program you can try out or if it is just a part of a program (a snippet) so that you need to add surrounding statements (e.g., import statements) to try the code out yourself. The appearance of a vertical line to the left or not will then quickly tell you what type of code you see.

An interactive Python session is typeset as

```
>>> from math import *
>>> p = 1; c = -1.5
>>> a = sqrt(4*p + c)
```

Running a program, say `ball_yc.py`, in the terminal window, followed by some possible output is typeset as[27]

Terminal

```
ball_yc.py
At t=0.0417064 s and 0.977662 s, the height is 0.2 m.
```

Sometimes just the output from a program is shown, and this output appears as plain "computer text":

```
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
```

Files containing data are shown in a similar way in this book:

```
date    Oslo   London   Berlin   Paris   Rome   Helsinki
01.05   18     21.2     20.2     13.7    15.8   15
01.06   21     13.2     14.9     18      24     20
01.07   13     14       16       25      26.2   14.5
```

1.8 Exercises

What Does It Mean to Solve an Exercise?

The solution to most of the exercises in this book is a Python program. To produce the solution, you first need understand the problem and

[27] Recall from Chapter 1.5.3 that we just write the program name. A real execution demands prefixing the program name by `python` in a DOS/Unix terminal window, or by `run` if you run the program from an interactive IPython session. We refer to Appendix H.1 for more complete information on running Python programs in different ways.

what the program is supposed to do, and then you need to understand how to translate the problem description into a series of Python statements. Equally important is the verification (testing) of the program. A complete solution to a programming exercises therefore consists of two parts: the program text and a demonstration that the program works correctly. Some simple programs, like the ones in the first two exercises below, have so simple output that the verification can just be to run the program and record the output.

In cases where the correctness of the output is not obvious, it is necessary to provide information together with the output to "prove" that the result is correct. This can be a calculation done separately on a calculator, or one can apply the program to a special simple test with known results. The requirement is to provide evidence that the program works as intended.

The sample run of the program to check its correctness can be inserted at the end of the program as a triple-quoted string[28]. The contents of the string can be text from the run in the terminal window, cut and pasted to the program file by the aid of the mouse. Alternatively, one can run the program and direct the output to a file[29]:

Terminal
```
Terminal> python myprogram.py > result
```

Afterwards, use the editor to insert the file `result` inside the string.

As an example, suppose we are to write a program for converting Fahrenheit degrees to Celsius. The solution process can be divided into three steps:

1. Establish the mathematics to be implemented: solving (1.2) with respect to C gives the conversion formula
$$C = \frac{5}{9}(F - 32)\,.$$
2. Coding of the formula in Python: `C = (5.0/9)*(F - 32)`
3. Establish a test case: from the `c2f.py` program in Chapter 1.3.3 we know that $C = 21$ corresponds to $F = 69.8$. We can therefore, in our new program, set $F = 69.8$ and check that $C = 21$. The output from a run can be appended as a triple quoted string at the end of the program.

[28] Alternatively, the output lines can be inserted as comments, but using a multi-line string requires less typing. (Technically, a string object is created, but not assigned to any name or used for anything in the program – but for a human the text in the string contains useful information.)

[29] The redirection to files does not work if the program is run inside IPython. In a DOS terminal window you may also choose to redirect output to a file, because cut and paste between the DOS window and the program window does not work by default unless you right-click the top bar, choose Properties and tick off Quick Edit Mode.

An appropriate complete solution to the exercise is then

```
# Convert from Fahrenheit degrees to Celsius degrees:
F = 69.8
C = (5.0/9)*(F - 32)
print C

'''
Sample run:
python f2c.py
21.0
'''
```

Exercise 1.1. *Compute 1+1.*

The first exercise concerns some very basic mathematics and programming: assign the result of 1+1 to a variable and print the value of that variable. Name of program file: `1plus1.py`. ◇

Exercise 1.2. *Write a "Hello, World!" program.*

Almost all books about programming languages start with a very simple program that prints the text "Hello, World!" no the screen. Make such a program in Python. Name of program file: `hello_world.py`. ◇

Exercise 1.3. *Derive and compute a formula.*

Can a newborn baby in Norway expect to live for one billion (10^9) seconds? Name of program file: `seconds2years.py`. ◇

Exercise 1.4. *Convert from meters to British length units.*

Make a program where you set a length given in meters and then compute and write out the corresponding length measured in inches, in feet, in yards, and in miles. Use that one inch is 2.54 cm, one foot is 12 inches, one yard is 3 feet, and one British mile is 1760 yards. As a verification, a length of 640 meters corresponds to 25196.85 inches, 2099.74 feet, 699.91 yards, or 0.3977 miles. Name of program file: `length_conversion.py`. ◇

Exercise 1.5. *Compute the mass of various substances.*

The density of a substance is defined as $\varrho = m/V$, where m is the mass of a volume V. Compute and print out the mass of one liter of each of the following substances whose densities in g/cm^3 are found in the file `src/files/densities.dat`: iron, air, gasoline, ice, the human body, silver, and platinum: 21.4. Name of program file: `1liter.py`. ◇

Exercise 1.6. *Compute the growth of money in a bank.*

Let p be a bank's interest rate in percent per year. An initial amount A has then grown to

$$A\left(1 + \frac{p}{100}\right)^n$$

after n years. Make a program for computing how much money 1000 euros have grown to after three years with 5% interest rate. Name of program file: `interest_rate.py`. ◇

Exercise 1.7. *Find error(s) in a program.*

Suppose somebody has written a simple one-line program for computing sin(1):

```
x=1; print 'sin(%g)=%g' % (x, sin(x))
```

Type in this program and try to run it. What is the problem? ◇

Exercise 1.8. *Type in program text.*

Type the following program in your editor and execute it. If your program does not work, check that you have copied the code correctly.

```
from math import pi

h = 5.0   # height
b = 2.0   # base
r = 1.5   # radius

area_parallelogram = h*b
print 'The area of the parallelogram is %.3f' % area_parallelogram

area_square = b**2
print 'The area of the square is %g' % area_square

area_circle = pi*r**2
print 'The area of the circle is %.3f' % area_circle

volume_cone = 1.0/3*pi*r**2*h
print 'The volume of the cone is %.3f' % volume_cone
```

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

Exercise 1.9. *Type in programs and debug them.*

Type these short programs in your editor and execute them. When they do not work, identify and correct the erroneous statements.

(a)Does $\sin^2(x) + \cos^2(x) = 1$?

```
from math import sin, cos
x = pi/4
1_val = sin^2(x) + cos^2(x)
print 1_VAL
```

Name of program file: `sin2_plus_cos2.py`

(b)Work with the expressions for movement with constant acceleration:

```
v0 = 3 m/s
t = 1 s
a = 2 m/s**2
s = v0*t + 1/2 a*t**2
print s
```

Name of program file: `acceleration.py`

(c)Verify these equations:

$$(a + b)^2 = a^2 + 2ab + b^2$$

$$(a - b)^2 = a^2 - 2ab + b^2$$

```
a = 3,3   b = 5,3
a2 = a**2
b2 = b**2

eq1_sum = a2 + 2ab + b2
eq2_sum = a2 - 2ab + b2

eq1_pow = (a + b)**2
eq2_pow = (a - b)**2

print 'First equation:  %g = %g', % (eq1_sum, eq1_pow)
print 'Second equation: %h = %h', % (eq2_pow, eq2_pow)
```

Name of program file: `a_pm_b_sqr.py` ◇

Exercise 1.10. *Evaluate a Gaussian function.*

The bell-shaped Gaussian function,

$$f(x) = \frac{1}{\sqrt{2\pi}\, s} \exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right], \tag{1.6}$$

is one of the most widely used functions in science and technology[30]. The parameters m and s are real numbers, where s must be greater than zero. Make a program for evaluating this function when $m = 0$, $s = 2$, and $x = 1$. Verify the program's result by comparing with hand calculations on a calculator. Name of program file: `Gaussian1.py`. ◇

Exercise 1.11. *Compute the air resistance on a football.*

The drag force, due to air resistance, on an object can be expressed as

$$F_d = \frac{1}{2} C_D \varrho A V^2, \tag{1.7}$$

where ϱ is the density of the air, V is the velocity of the object, A is the cross-sectional area (normal to the velocity direction), and C_D is the drag coefficient, which depends heavily on the shape of the object and the roughness of the surface.

The gravity force on an object with mass m is $F_g = mg$, where $g = 9.81 \mathrm{m\,s}^{-2}$.

We can use the formulas for F_d and F_g to study the importance of air resistance versus gravity when kicking a football. The density of air is $\varrho = 1.2$ kg m^{-3}. We have $A = \pi a^2$ for any ball with radius a. For a football $a = 11$ cm. The mass of a football is 0.43 kg, C_D can be taken as 0.2.

Make a program that computes the drag force and the gravity force on a football. Write out the forces with one decimal in units of Newton

[30] The function is named after Carl Friedrich Gauss, 1777–1855, who was a German mathematician and scientist, now considered as one of the greatest scientists of all time. He contributed to many fields, including number theory, statistics, mathematical analysis, differential geometry, geodesy, electrostatics, astronomy, and optics. Gauss introduced the function (1.6) when he analyzed probabilities related to astronomical data.

(N = kg m/s^2). Also print the ratio of the drag force and the gravity force. Define C_D, ϱ, A, V, m, g, F_d, and F_g as variables, and put a comment with the corresponding unit. Use the program to calculate the forces on the ball for a hard kick, $V = 120$ km/h and for a soft kick, $V = 10$ km/h (it is easy to mix inconsistent units, so make sure you compute with V expressed in m/s). Name of program file: `kick.py`. ◇

Exercise 1.12. *How to cook the perfect egg.*

As an egg cooks, the proteins first denature and then coagulate. When the temperature exceeds a critical point, reactions begin and proceed faster as the temperature increases. In the egg white the proteins start to coagulate for temperatures above 63 C, while in the yolk the proteins start to coagulate for temperatures above 70 C. For a soft boiled egg, the white needs to have been heated long enough to coagulate at a temperature above 63 C, but the yolk should not be heated above 70 C. For a hard boiled egg, the center of the yolk should be allowed to reach 70 C.

The following formula expresses the time t it takes (in seconds) for the center of the yolk to reach the temperature T_y (in Celsius degrees):

$$t = \frac{M^{2/3}c\rho^{1/3}}{K\pi^2(4\pi/3)^{2/3}}\ln\left[0.76\frac{T_o - T_w}{T_y - T_w}\right]. \qquad (1.8)$$

Here, M, ρ, c, and K are properties of the egg: M is the mass, ρ is the density, c is the specific heat capacity, and K is thermal conductivity. Relevant values are $M = 47$ g for a small egg and $M = 67$ g for a large egg, $\rho = 1.038\ \mathrm{g\,cm^{-3}}$, $c = 3.7\ \mathrm{J\,g^{-1}\,K^{-1}}$, and $K = 5.4\cdot 10^{-3}\ \mathrm{W\,cm^{-1}\,K^{-1}}$. Furthermore, T_w is the temperature (in C degrees) of the boiling water, and T_o is the original temperature (in C degrees) of the egg before being put in the water. Implement the formula in a program, set $T_w = 100$ C and $T_y = 70$ C, and compute t for a large egg taken from the fridge ($T_o = 4$ C) and from room temperature ($T_o = 20$ C). Name of program file: `egg.py`. ◇

Exercise 1.13. *Derive the trajectory of a ball.*

The purpose of this exercise is to explain how Equation (1.5) for the trajectory of a ball arises from basic physics. There is no programming in this exercise, just physics and mathematics.

The motion of the ball is governed by Newton's second law:

$$F_x = ma_x \qquad (1.9)$$
$$F_y = ma_y \qquad (1.10)$$

where F_x and F_y are the sum of forces in the x and y directions, respectively, a_x and a_y are the accelerations of the ball in the x and y directions, and m is the mass of the ball. Let $(x(t), y(t))$ be the position of the ball, i.e., the horizontal and vertical coordinate of the ball at time t. There are well-known relations between acceleration, velocity, and position: the acceleration is the time derivative of the velocity, and the velocity is the time derivative of the position. Therefore we have that

$$a_x = \frac{d^2x}{dt^2}, \tag{1.11}$$

$$a_y = \frac{d^2y}{dt^2}. \tag{1.12}$$

If we assume that gravity is the only important force on the ball, $F_x = 0$ and $F_y = -mg$.

Integrate the two components of Newton's second law twice. Use the initial conditions on velocity and position,

$$\frac{d}{dt}x(0) = v_0 \cos\theta, \tag{1.13}$$

$$\frac{d}{dt}y(0) = v_0 \sin\theta, \tag{1.14}$$

$$x(0) = 0, \tag{1.15}$$

$$y(0) = y_0, \tag{1.16}$$

to determine the four integration constants. Write up the final expressions for $x(t)$ and $y(t)$. Show that if $\theta = \pi/2$, i.e., the motion is purely vertical, we get the formula (1.1) for the y position. Also show that if we eliminate t, we end up with the relation (1.5) between the x and y coordinates of the ball. You may read more about this type of motion in a physics book, e.g., [6]. ◇

Exercise 1.14. *Find errors in the coding of formulas.*

Some versions of our program for calculating the formula (1.2) are listed below. Determine which versions that will not work correctly and explain why in each case.

```
C = 21;    F =  9/5*C + 32;        print F
C = 21.0;  F =  (9/5)*C + 32;      print F
C = 21.0;  F =  9*C/5 + 32;        print F
C = 21.0;  F =  9.*(C/5.0) + 32;   print F
C = 21.0;  F =  9.0*C/5.0 + 32;    print F
C = 21;    F =  9*C/5 + 32;        print F
C = 21.0;  F =  (1/5)*9*C + 32;    print F
C = 21;    F =  (1./5)*9*C + 32;   print F
```

◇

Exercise 1.15. *Explain why a program does not work.*

Find out why the following program does not work:

```
C = A + B
A = 3
B = 2
print C
```

◇

Exercise 1.16. *Find errors in Python statements.*

Try the following statements in an interactive Python shell. Explain why some statements fail and correct the errors.

```
1a = 2
a1 = b
x = 2
y = X + 4  # is it 6?
from Math import tan
print tan(pi)
pi = "3.14159’
print tan(pi)
c = 4**3**2**3
_ = ((c-78564)/c + 32))
discount = 12%
AMOUNT = 120.-
amount = 120$
address = hpl@simula.no
and = duck
class = ’INF1100, gr 2"
continue_ = x > 0
rev = fox = True
Norwegian = [’a human language’]
true = fox is rev in Norwegian
```

Hint: It might be wise to test the values of the expressions on the right-hand side, and the validity of the variable names, seperately before you put the left- and right-hand sides together in statements. The last two statements work, but explaining why goes beyond what is treated in this chapter. ◇

Exercise 1.17. *Find errors in the coding of a formula.*

Given a quadratic equation,

$$ax^2 + bx + c = 0,$$

the two roots are

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}. \qquad (1.17)$$

What are the problems with the following program?

```
a = 2; b = 1; c = 2
from math import sqrt
q = sqrt(b*b - 4*a*c)
x1 = (-b + q)/2*a
x2 = (-b - q)/2*a
print x1, x2
```

Hint: Compute all terms in (1.17) with the aid of a calculator, and compare with the corresponding intermediate results computed in the program (you need to add some `print` statements to see the result of `q`, `-b+q`, and `2*a`). ◇

`math.tan`. Since `pydoc` is very fast, many prefer `pydoc` over web pages, but `pydoc` has often less information compared to the Python Library Reference.

There are also numerous books about Python. Beazley [1] is an excellent reference that improves and extends the information in the Python Library Reference. The "Learning Python" book [8] has been very popular for many years as an introduction to the language. There is a special web page `http://wiki.python.org/moin/PythonBooks` listing most Python books on the market. A comprehensive book on the use of Python for doing scientific research is [5].

Quick references, which list "all" Python functionality in compact tabular form, are very handy. We recommend in particular the one by Richard Gruet: `http://rgruet.free.fr/#QuickRef`.

The website `http://www.python.org/doc/` contains a list of useful Python introductions and reference manuals.

2.7 Exercises

Exercise 2.1. *Make a Fahrenheit–Celsius conversion table.*

Write a program that prints out a table with Fahrenheit degrees $0, 10, 20, \ldots, 100$ in the first column and the corresponding Celsius degrees in the second column. Hint: Modify the `c2f_table_while.py` program from Chapter 2.1.2. Name of program file: `f2c_table_while.py`. ⋄

Exercise 2.2. *Write an approximate Fahrenheit–Celsius conversion table.*

Many people use an approximate formula for quickly converting Fahrenheit (F) to Celsius (C) degrees:

$$C \approx \hat{C} = (F - 30)/2 \qquad (2.2)$$

Modify the program from Exercise 2.1 so that it prints three columns: F, C, and the approximate value $\hat{C}$. Name of program file: `f2c_approx_table.py`. ⋄

Exercise 2.3. *Generate odd numbers.*

Write a program that generates all odd numbers from 1 to `n`. Set `n` in the beginning of the program and use a `while` loop to compute the numbers. (Make sure that if `n` is an even number, the largest generated odd number is `n-1`.) Name of program file: `odd.py`. ⋄

Exercise 2.4. *Store odd numbers in a list.*

Modify the program from Exercise 2.3 to store the generated odd numbers in a list. Start with an empty list and use a `while` loop where you in each pass of the loop append a new element to the list. Finally, print the list elements to the screen. Name of program file: `odd_list1.py`. ◇

Exercise 2.5. *Generate odd numbers by a list comprehension.*
Solve Exercise 2.4 using a list comprehension (with `for` and `range`). Name of program file: `odd_list2.py`. ◇

Exercise 2.6. *Make a table of values from a formula.*
Write a program that prints a nicely formatted table of t and $y(t)$ values, where

$$y(t) = v_0 t - \frac{1}{2} g t^2 \,.$$

Use n uniformly spaced t values throughout the interval $[0, 2v_0/g]$. Set $v_0 = 1$ and $n = 11$. Name of program file: `ball_table1.py`. ◇

Exercise 2.7. *Store values from a formula in lists.*
Modify the program from Exercise 2.6 so that the t and y values are stored in two lists `t` and `y`. Thereafter, transverse the lists with a `for` loop and write out a nicely formatted table of t and y values (using either a `zip` or `range` construction). Set $v_0 = 10$ and $n = 81$. Name of program file: `ball_table2.py`. ◇

Exercise 2.8. *Work with a list.*
Set a variable `primes` to a list containing the numbers 2, 3, 5, 7, 11, and 13. Write out each list element in a `for` loop. Assign 17 to a variable `p` and add `p` to the end of the list. Print out the whole new list. Name of program file: `primes.py`. ◇

Exercise 2.9. *Simulate operations on lists by hand.*
You are given the following program:

```
a = [1, 3, 5, 7, 11]
b = [13, 17]
c = a + b
print c
b[0] = -1
d = [e+1 for e in a]
print d
d.append(b[0] + 1)
d.append(b[-1] + 1)
print d[-2:]
```

Go through each statement and explain what is printed by the program. ◇

Exercise 2.10. *Generate equally spaced coordinates.*
We want to generate x coordinates between 1 and 2 with spacing 0.01. The coordinates are given by the formula $x_i = 1 + ih$, where $h = 0.01$ and i runs over integers $0, 1, \ldots, 100$. Compute the x_i values and store them in a list (use a `for` loop, and append each new x_i value to a list, which is empty initially). Name of program file: `coor1.py`. ◇

Exercise 2.11. *Use a list comprehension to solve Exer. 2.10.*

The problem is the same as in Exercise 2.10, but now we want the x_i values to be stored in a list using a list comprehension construct (see Chapter 2.3.5). Name of program file: `coor2.py`. ◇

Exercise 2.12. *Compute a mathematical sum.*

The following code is supposed to compute the sum $s = \sum_{k=1}^{M} \frac{1}{k}$:

```
s = 0;  k = 1;  M = 100
while k < M:
    s += 1/k
print s
```

This program does not work correctly. What are the three errors? (If you try to run the program, nothing will happen on the screen. Type `Ctrl-C`, i.e., hold down the Control (`Ctrl`) key and then type the `c` key, to stop a program.) Write a correct program. Name of program file: `sum_while.py`.

There are two basic ways to find errors in a program: (i) read the program carefully and think about the consequences of each statement, and (ii) print out intermediate results and compare with hand calculations. First, try method (i) and find as many errors as you can. Then, try method (ii) for $M = 3$ and compare the evolution of `s` with your own hand calculations. ◇

Exercise 2.13. *Use a for loop in Exer. 2.12.*

Rewrite the corrected version of the program in Exercise 2.12 using a `for` loop over `k` values instead of a `while` loop. Name of program file: `sum_for.py`. ◇

Exercise 2.14. *Simulate a program by hand.*

Consider the following program for computing with interest rates:

```
initial_amount = 100
p = 5.5  # interest rate
amount = initial_amount
years = 0
while amount <= 1.5*initial_amount:
    amount = amount + p/100*amount
    years = years + 1
print years
```

Explain with words what type of mathematical problem that is solved by this program. Compare this computerized solution with the technique your high school math teacher would prefer.

Use a pocket calculator (or use an interactive Python shell as substitute) and work through the program by hand. Write down the value of `amount` and `years` in each pass of the loop.

Change the value of `p` to 5. Why will the loop now run forever? (See Exercise 2.12 for how to stop the program if you try to run it.) Make the program more robust against such errors.

Make use of the operator `+=` wherever possible in the program.

Insert the text for the answers to (a) and (b) in a multi-line string in the program file. Name of program file: `interest_rate_loop.py`. ◇

Exercise 2.15. *Explore the Python Library Reference.*

Suppose you want to compute the inverse sine function: $\sin^{-1} x$. The `math` module has a function for computing $\sin^{-1} x$, but what is the right name of this function? Read Chapter 2.6.3 and use the `math` entry in the index of the Python Library Reference to find out how to compute $\sin^{-1} x$. Make a program where you compute $\sin^{-1} x$ for n x values uniformly distributed between 0 and 1, and write out the results in a nicely formatted table. For each x value, check that the sine of $\sin^{-1} x$ equals x. Name of program file: `inverse_sine.py`. ◇

Exercise 2.16. *Index a nested lists.*

We define the following nested list:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

Index this list to extract 1) the letter `a`; 2) the list `['d', 'e', 'f']`; 3) the last element `h`; 4) the `d` element. Explain why `q[-1][-2]` has the value `g`. Name of program file: `index_nested_list.py`. ◇

Exercise 2.17. *Construct a double for loop over a nested list.*

Consider the list from Exercise 2.16. We can visit all elements of `q` using this nested `for` loop:

```
for i in q:
    for j in range(len(i)):
        print i[j]
```

What type of objects are `i` and `j`? Name of program file: `nested_list_iter.py`. ◇

Exercise 2.18. *Store data in lists in Exercise 2.2.*

Modify the program from Exercise 2.2 so that all the F, C, and $\hat{C}$ values are stored in separate lists `F`, `C`, and `C_approx`, respectively. Then make a nested list `conversion` so that `conversion[i]` holds a row in the table: `[F[i], C[i], C_approx[i]]`. Finally, let the program traverse the `conversion` list and write out the same table as in Exercise 2.2. Name of program file: `f2c_approx_lists.py`. ◇

Exercise 2.19. *Store data from Exer. 2.7 in a nested list.*

After having computed the two lists of t and y values in the program from Exercise 2.7, store the two lists in a new list `ty1`. Write out a table of t and y values by traversing the data in the `ty1` list. Thereafter, make a list `ty2` which holds each row in the table of t and y values (`ty1` is a list of table columns while `ty2` is a list of table rows, as explained in Chapter 2.4). Write out the table by traversing the `ty2` list. Name of program file: `ball_table3.py`. ◇

Exercise 2.20. *Convert nested list comprehensions to nested standard loops.*

Rewrite the generation of the nested list `q`,

```
q = [r**2 for r in [10**i for i in range(5)]]
```

by using standard `for` loops instead of list comprehensions. Name of program file: `listcomp2for.py`. ◇

Exercise 2.21. *Values of boolean expressions.*

Explain the outcome of each of the following boolean expressions:

```
C = 41
C == 40
C != 40 and C < 41
C != 40 or  C < 41
not C == 40
not C > 40
C <= 41
not False
True and False
False or True
False or False or False
True and True and False
False == 0
True == 0
True == 1
```

Note: It makes sense to compare `True` and `False` to the integers 0 and 1, but not other integers (e.g., `True == 12` is `False` although the *integer* 12 evaluates to `True` in a boolean context, as in `bool(12)` or `if 12`). ◇

Exercise 2.22. *Explore round-off errors from a large number of inverse operations.*

Maybe you have tried to hit the square root key on a calculator multiple times and then squared the number again an equal number of times. These set of inverse mathematical operations should of course bring you back to the starting value for the computations, but this does not always happen. To avoid tedious pressing of calculator keys we can let a computer automate the process. Here is an appropriate program:

```
from math import sqrt
for n in range(1, 60):
    r = 2.0
    for i in range(n):
        r = sqrt(r)
    for i in range(n):
        r = r**2
    print '%d times sqrt and **2: %.16f' % (n, r)
```

Explain with words what the program does. Then run the program. Round-off errors are here completely destroying the calculations when `n` is large enough! Investigate the case when we come back to 1 instead of 2 by fixing the `n` value and printing out `r` in both `for` loops over `i`. Can you now explain why we come back to 1 and not 2? Name of program file: `repeated_sqrt.py`. ◇

Exercise 2.23. *Explore what zero can be on a computer.*
Type in the following code and run it:

```
eps = 1.0
while 1.0 != 1.0 + eps:
    print '...............', eps
    eps = eps/2.0
print 'final eps:', eps
```

Explain with words what the code is doing, line by line. Then examine the output. How can it be that the "equation" $1 \neq 1 + \text{eps}$ is not true? Or in other words, that a number of approximately size 10^{-16} (the final `eps` value when the loop terminates) gives the same result as if `eps`[9] were zero? Name of program file: `machine_zero.py`.

If somebody shows you this interactive session

```
>>> 0.5 + 1.45E-22
0.5
```

and claims that Python cannot add numbers correctly, what is your answer? ◇

Exercise 2.24. *Compare two real numbers on a computer.*
Consider the following simple program inspired by Chapter 1.4.3:

```
a = 1/947.0*947
b = 1
if a != b:
    print 'Wrong result!'
```

Try to run this example!

One should never compare two floating-point objects directly using `==` or `!=`, because round-off errors quickly make two identical mathematical values different on a computer. A better result is to test if $|a - b|$ is sufficiently small, i.e., if a and b are "close enough" to be considered equal. Modify the test according to this idea.

Thereafter, read the documentation of the function `float_eq` from SciTools: `scitools.std.float_eq` (see page 78 for how to bring up the documentation of a module or a function in a module). Use this function to check whether two real numbers are equal within a tolerance. Name of program file: `compare_float.py`. ◇

Exercise 2.25. *Interpret a code.*
The function `time` in the module `time` returns the number of seconds since a particular date (called the Epoch, which is January 1,

[9] This nonzero `eps` value is called *machine epsilon* or *machine zero* and is an important parameter to know, especially when certain mathematical techniques are applied to control round-off errors.

1970, on many types of computers). Python programs can therefore use `time.time()` to mimic a stop watch. Another function, `time.sleep(n)` causes the program to "sleep" for `n` seconds and is handy for inserting a pause. Use this information to explain what the following code does:

```
import time
t0 = time.time()
while time.time() - t0 < 10:
    print '....I like while loops!'
    time.sleep(2)
print 'Oh, no - the loop is over.'
```

How many times is the `print` statement inside the loop executed? Now, copy the code segment and change the `<` sign in the loop condition to a `>` sign. Explain what happens now. Name of program: `time_while.py`. ◇

Exercise 2.26. *Explore problems with inaccurate indentation.*

Type in the following program in a file and check carefully that you have exactly the same spaces:

```
C = -60; dC = 2
while C <= 60:
    F = (9.0/5)*C + 32
      print C, F
C = C + dC
```

Run the program. What is the first problem? Correct that error. What is the next problem? What is the cause of that problem? (See Exercise 2.12 for how to stop a hanging program.)

The lesson learned from this exercise is that one has to be very careful with indentation in Python programs! Other computer languages usually enclose blocks belonging to loops in curly braces, parentheses, or BEGIN-END marks. Python's convention with using solely indentation contributes to visually attractive, easy-to-read code, at the cost of requiring a pedantic attitude to blanks from the programmer. ◇

Exercise 2.27. *Simulate nested loops by hand.*

Go through the code below by hand, statement by statement, and calculate the numbers that will be printed.

```
n = 3
for i in range(-1, n):
    if i != 0:
        print i

for i in range(1, 13, 2*n):
    for j in range(n):
        print i, j

for i in range(1, n+1):
    for j in range(i):
        if j:
            print i, j
```

```
for i in range(1, 13, 2*n):
    for j in range(0, i, 2):
        for k in range(2, j, 1):
            b = i > j > k
            if b:
                print i, j, k
```

You may use a debugger, see Appendix F.1, to see what happens when you step through the code. ◇

Exercise 2.28. *Explore punctuation in Python programs.*
Some of the following assignments work and some do not. Explain in each case why the assignment works/fails and, if it works, what kind of object `x` refers to and what the value is if we do a `print x`.

```
x = 1
x = 1.
x = 1;
x = 1!
x = 1?
x = 1:
x = 1,
```

Hint: Explore the statements in an interactive Python shell. ◇

Exercise 2.29. *Investigate a for loop over a changing list.*
Study the following interactive session and explain in detail what happens in each pass of the loop, and use this explanation to understand the output.

```
>>> numbers = range(10)
>>> print numbers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for n in numbers:
...     i = len(numbers)/2
...     del numbers[i]
...     print 'n=%d, del %d' % (n,i), numbers
...
n=0, del 5 [0, 1, 2, 3, 4, 6, 7, 8, 9]
n=1, del 4 [0, 1, 2, 3, 6, 7, 8, 9]
n=2, del 4 [0, 1, 2, 3, 7, 8, 9]
n=3, del 3 [0, 1, 2, 7, 8, 9]
n=8, del 3 [0, 1, 2, 8, 9]
```

The message in this exercise is to *never modify a list that is used in a* `for` *loop.* Modification is indeed technically possible, as we show above, but you really need to know what you are dingo – to avoid getting frustrated by strange program behavior. ◇

```
    sum1 = 0
    for i in range(1, n/2 + 1):
        sum1 += f(a + (2*i-1)*h)

    sum2 = 0
    for i in range(1, n/2):
        sum2 += f(a + 2*i*h)

    integral = (b-a)/(3*n)*(f(a) + f(b) + 4*sum1 + 2*sum2)
    return integral
```

The complete code is found in the file `Simpson.py`.

A very good exercise is to simulate the program flow by hand, starting with the call to the `application` function. A debugger might be a convenient tool for controlling that your thinking is correct, see Appendix F.1.

3.5 Exercises

Exercise 3.1. *Write a Fahrenheit–Celsius conversion function.*

The formula for converting Fahrenheit degrees to Celsius reads

$$C = \frac{5}{9}(F - 32)\,. \tag{3.7}$$

Write a function `C(F)` that implements this formula. To verify the implementation of `C(F)`, you can convert a Celsius temperature to Fahrenheit and then back to Celsius again using the `F(C)` function from Chapter 3.1.1 and the `C(F)` function implementing (3.7). That is, you can check that the boolean expression `c == C(F(c))` is `True` for any temperature `c` (you should, however, be careful with comparing real numbers with `==`, see Exercise 2.24). Name of program file: `f2c.py`. ◇

Exercise 3.2. *Write the program in Exer. 2.12 as a function.*

Define a Python function `sum_1_div_k(M)` that returns the sum s as defined in Exercise 2.12. Print out the result of calling `s(3)` and check that the answer is correct. Name of program: `sum_func.py`. ◇

Exercise 3.3. *Write a function for solving $ax^2 + bx + c = 0$.*

Given a quadratic equation $ax^2 + bx + c = 0$, write a function `roots(a, b, c)` that returns the two roots of the equation. The returned roots should be `float` objects when the roots are real, otherwise the roots should be `complex` objects. Construct two test cases with known solutions, one with real roots and the other with complex roots, to check that the function returns correct values and correct type of objects. Hint: Use techniques from Chapter 1.6.3. Name of program: `roots_quadratic.py`. ◇

Exercise 3.4. *Implement the sum function.*

The standard Python function called `sum` takes a list as argument and computes the sum of the elements in the list:

```
>>> sum([1,3,5,-5])
4
```

Implement your own version of `sum`. Name of program: `mysum.py`. ◇

Exercise 3.5. *Integrate a function by one trapezoid.*

An approximation to the integral of a function $f(x)$ over an interval $[a, b]$ can found by first approximating $f(x)$ by the straight line that goes through the end points $(a, f(a))$ and $(b, f(b))$, and then finding the area under the straight line (which is the area of a trapezoid). The resulting formula becomes

$$\int_a^b f(x)dx \approx \frac{b-a}{2}(f(a)+f(b))\,. \tag{3.8}$$

Write a function `trapezint1(f, a, b)` that returns this approximation to the integral. The argument `f` is a Python implementation `f(x)` of the mathematical function $f(x)$.

Using (3.8), compute the following integrals: $\int_0^{\ln 3} e^x dx$, $\int_0^{\pi} \cos x\, dx$, $\int_0^{\pi} \sin x\, dx$, and $\int_0^{\pi/2} \sin x\, dx$, In each case, write out the error, i.e., the difference between the exact integral and the approximation (3.8). Make rough sketches of the trapezoid for each integral in order to understand how the method behaves in the different cases. Name of program file: `trapezint1.py`. ◇

Exercise 3.6. *Integrate a function by two trapezoids.*

We can easily improve the formula (3.8) from Exercise 3.5 by approximating the area under the function $f(x)$ by two equal-sized trapezoids. Derive a formula for this approximation and implement it in a function `trapezint2(f, a, b)`. Run the examples from Exercise 3.5 and see how much better the new formula is. Make sketches of the two trapezoids in each case. Name of program file: `trapezint2.py`. ◇

Exercise 3.7. *Derive the general Trapezoidal integration rule.*

A further improvement of the approximate integration method from Exercise 3.6 is to divide the area under the $f(x)$ curve into n equal-sized trapezoids. Based on this idea, derive the following formula for approximating the integral:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{1}{2}h\left(f(x_{i-1})+f(x_i)\right), \tag{3.9}$$

where h is the width of the trapezoids, $h=(b-a)/n$, and $x_i = a + ih$, $i = 0, \ldots, n$, are the coordinates of the sides of the trapezoids. Figure 5.14b on page 255 visualizes the idea of the Trapezoidal rule.

Implement (3.9) in a Python function `trapezint(f, a, b, n)`. Run the examples from Exercise 3.5 with $n = 10$. Name of program file: `trapezint.py`.

Remark. Formula (3.9) is not the most common way of expressing the Trapezoidal integration rule. The reason is that $f(x_i)$ is evaluated twice, first in term i and then as $f(x_{i-1})$ in term $i+1$. The formula can be further developed to avoid unnecessary evaluations of $f(x_i)$, which results in the standard form

$$\int_a^b f(x)dx \approx \frac{1}{2}h(f(a)+f(b)) + h\sum_{i=1}^{n-1} f(x_i)\,. \qquad (3.10)$$

◇

Exercise 3.8. *Derive the general Midpoint integration rule.*

The idea of the Midpoint rule for integration is to divide the area under the curve $f(x)$ into n equal-sized rectangles (instead of trapezoids as in Exercise 3.7). The height of the rectangle is determined by the value of f at the midpoint of the rectangle. Figure 5.14a on page 255 illustrates the idea. Compute the area of each rectangle, sum them up, and arrive at the formula for the Midpoint rule:

$$\int_a^b f(x)dx \approx h\sum_{i=1}^{n} f(a+\frac{1}{2}ih), \qquad (3.11)$$

where $h = (b-a)/n$ is the width of each rectangle. Implement this formula in a Python function `midpointint(f, a, b, n)` and test the function on the Examples listed in Exercise 3.5. How do the errors in the Midpoint rule compare with those of the Trapezoidal rule (Exercise 3.7) for $n = 1$ and $n = 10$? Name of program file: `midpointint.py`.
◇

Exercise 3.9. *Compute the area of an arbitrary triangle.*

An arbitrary triangle can be described by the coordinates of its three vertices: (x_1, y_1), (x_2, y_2), (x_3, y_3), numbered in a counterclockwise direction. The area of the triangle is given by the formula

$$A = \frac{1}{2}\left|x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1\right|\,. \qquad (3.12)$$

Write a function `area(vertices)` that returns the area of a triangle whose vertices are specified by the argument `vertices`, which is a nested list of the vertex coordinates. For example, computing the area of the triangle with vertex coordinates $(0, 0)$, $(1, 0)$, and $(0, 2)$ is done by

```
triangle1 = area([[0,0], [1,0], [0,2]])
# or
v1 = (0,0);  v2 = (1,0);  v3 = (0,2);
vertices = [v1, v2, v3]
```

```
triangle1 = area(vertices)

print 'Area of triangle is %.2f' % triangle1
```

Recall from Chapter 2.4 that extracting a coordinate like `x2` for use in the formula (3.12) is done by `vertices[1][0]`.

Test the `area` function on a triangle with known area. Name of program file: `area_triangle.py`. ◇

Exercise 3.10. *Compute the length of a path.*

Some object is moving along a path in the plane. At n points of time we have recorded the corresponding (x, y) positions of the object: (x_0, y_0), (x_1, y_2), …, (x_{n-1}, y_{n-1}). The total length L of the path from (x_0, y_0) to (x_{n-1}, y_{n-1}) is the sum of all the individual line segments $((x_{i-1}, y_{i-1})$ to (x_i, y_i), $i = 1, \ldots, n-1)$:

$$L = \sum_{i=1}^{n-1} \sqrt{(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2}\,. \qquad (3.13)$$

Make a function `pathlength(x, y)` for computing L according to the formula. The arguments `x` and `y` hold all the $x_0, \ldots, x_{n-1}$ and $y_0, \ldots, y_{n-1}$ coordinates, respectively. Test the function on a triangular path with the four points $(1, 1)$, $(2, 1)$, $(1, 2)$, and $(1, 1)$. Name of program file: `pathlength.py`. ◇

Exercise 3.11. *Approximate π.*

The value of π equals the circumference of a circle with radius 1/2. Suppose we approximate the circumference by a polygon through $N+1$ points on the circle. The length of this polygon can be found using the `pathlength` function from Exercise 3.10. Compute $N+1$ points (x_i, y_i) along a circle with radius 1/2 according to the formulas

$$x_i = \frac{1}{2}\cos(2\pi i/N), \quad y_i = \frac{1}{2}\sin(2\pi i/N), \quad i = 0, \ldots, N\,.$$

Call the `pathlength` function and write out the error in the approximation of π for $N = 2^k$, $k = 2, 3, \ldots, 10$. Name of program file: `pi_approx.py`. ◇

Exercise 3.12. *Write various hello-world functions.*

Write three functions:

1. `hw1`, which takes no arguments and returns the string `'Hello, World!'`
2. `hw2`, which takes no arguments and returns nothing, but the string `'Hello, World!'` is printed in the terminal window
3. `hw3`, which takes two string arguments and prints these two arguments separated by a comma.

Use the following main program to test the three functions:

```
print hw1()
hw2()
hw3('Hello ', 'World!')
```

Name of program: `hw_func.py`. ◇

Exercise 3.13. *Approximate a function by a sum of sines.*

We consider the piecewise constant function

$$f(t) = \begin{cases} 1, & 0 < t < T/2, \\ 0, & t = T/2, \\ -1, & T/2 < t < T \end{cases} \tag{3.14}$$

Sketch this function on a piece of paper. One can approximate $f(t)$ by the sum

$$S(t;n) = \frac{4}{\pi}\sum_{i=1}^{n} \frac{1}{2i-1}\sin\left(\frac{2(2i-1)\pi t}{T}\right). \tag{3.15}$$

It can be shown that $S(t;n) \rightarrow f(t)$ as $n \rightarrow \infty$.

Write a Python function `S(t, n, T)` for returning the value of $S(t;n)$. Also write a Python `f(t, T)` for computing $f(t)$. Write out tabular information showing how the error $f(t) - S(t;n)$ varies with n and t for the case where $1, 3, 5, 10, 30, 100$ and $t = \alpha T$, with $T = 2\pi$, and $\alpha = 0.01, 0.25, 0.49$. Use the table to comment on how the quality of the approximation depends on α and n. Name of program file: `sinesum1.py`.

Remark. A sum of sine and/or cosine functions, as in (3.15), is called a *Fourier series*. Approximating a function by a Fourier series is a very important technique in science and technology. Exercise 5.30 asks for visualization of how well $S(t;n)$ approximates $f(t)$ for some values of n. ◇

Exercise 3.14. *Implement a Gaussian function.*

Make a Python function `gauss(x, m=0, s=1)` for computing the Gaussian function

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

Call `gauss(x)` and print out the result for $x \in [-5, 5]$ (say for 11 uniformly spaced x values). Name of program file: `Gaussian2.py`. ◇

Exercise 3.15. *Make a function of the formula in Exer. 1.12.*

Implement the formula (1.8) from Exercise 1.12 in a Python function with three arguments: `egg(M, To=20, Ty=70)`. The parameters ρ, K, c, and T_w can be set as local (constant) variables inside the function. Let t be returned from the function. Compute t for a soft and hard boiled egg, of a small ($M = 47$ g) and large ($M = 67$ g) size, taken from the fridge ($T_o = 4$ C) and from a hot room ($T = 25$ C). Name of program file: `egg_func.py`. ◇

Exercise 3.16. *Write a function for numerical differentiation.*

The formula

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{3.16}$$

can be used to find an approximate derivative of a mathematical function $f(x)$ if h is small. Write a function `diff(f, x, h=1E-6)` that returns the approximation (3.16) of the derivative of a mathematical function represented by a Python function `f(x)`.

Apply (3.16) to differentiate $f(x) = e^x$ at $x = 0$, $f(x) = e^{-2x^2}$ at $x = 0$, $f(x) = \cos x$ at $x = 2\pi$, and $f(x) = \ln x$ at $x = 1$. Use $h = 0.01$. In each case, write out the error, i.e., the difference between the exact derivative and the result of (3.16). Name of program file: `diff_f.py`. ◇

Exercise 3.17. *Make an adaptive Trapezoidal integration rule.*

A problem with the Trapezoidal integration rule (3.9) is to decide how many trapezoids (n) to use in order to achieve a desired accuracy. Let E be the error in the Trapezoidal method, i.e., the difference between the exact integral and that produced by (3.9). We would like to prescribe a (small) tolerance ϵ and find an n such that $E \leq \epsilon$. This demands some expression for the error E involving the parameter n.

One can show that

$$E \leq \frac{1}{12}(b-a)h^2 \max_{x\in[a,b]} \left|f''(x)\right| . \tag{3.17}$$

The maximum of $|f''(x)|$ can be computed (approximately) by evaluating $f''(x)$ at a large number of points in $[a, b]$, taking the absolute value $|f''(x)|$, and finding the maximum value of these. You can use the `diff2` function from Chapter 3.1.9 to compute $f''(x)$.

With the computed estimate of $\max|f''(x)|$ we can find h from setting the worst case error in (3.17) (i.e., the right-hand side) equal to the desired tolerance:

$$\frac{1}{12}(b-a)h^2 \max_{x\in[a,b]} \left|f''(x)\right| = \epsilon$$

Solving with respect to h gives

$$h = \sqrt{12\epsilon}\left((b-a) \max_{x\in[a,b]} \left|f''(x)\right|\right)^{-1/2} . \tag{3.18}$$

With $n = (b-a)/h$ we have the n that corresponds to the desired accuracy ϵ.

Make a Python function `adaptive_trapezint(f, a, b, eps=1E-5)` for computing the integral $\int_a^b f(x)dx$ with an error less than or equal to `eps`. First compute n and then call `trapezint(f, a, b, n)` from Exercise 3.7. Name of program file: `adaptive_trapezint.py`.

A numerical method that applies an expression for the error to automatically compute a proper discretization parameter, like n (or h) here, is known as an *adaptive* numerical method. ◇

Exercise 3.18. *Compute a polynomial via a product.*

Given $n+1$ roots $r_0, r_1, \ldots, r_n$ of a polynomial $p(x)$ of degree $n+1$, $p(x)$ can be computed by

$$p(x) = \prod_{i=0}^{n}(x - r_i) = (x - r_0)(x - r_1)\cdots(x - r_{n-1})(x - r_n)\,. \qquad (3.19)$$

Store the roots $r_0, \ldots, r_n$ in a list and make a loop that computes the product in (3.19). Test the program on a polynomial with roots -1, 1, and 2. Name of program file: `polyprod.py`. ◇

Exercise 3.19. *Implement the factorial function.*

The factorial of n is written as $n!$ and defined as

$$n! = n(n-1)(n-2)\cdots 2\cdot 1, \qquad (3.20)$$

with the special cases

$$1! = 1, \quad 0! = 1\,. \qquad (3.21)$$

For example, $4! = 4\cdot 3\cdot 2\cdot 1 = 24$, and $2! = 2\cdot 1 = 2$. Write a function `fact(n)` that returns $n!$. Return 1 immediately if x is 1 or 0, otherwise use a loop to compute $n!$. Name of program file: `fact.py`.

Remark. You can import a ready-made factorial function by

```
>>> from math import factorial
>>> factorial(4)
24
```

◇

Exercise 3.20. *Compute velocity and acceleration from position data; one dimension.*

Let $x(t)$ be the position of an object moving along the x axis. The velocity $v(t)$ and acceleration $a(t)$ can be approximately computed by the formulas

$$v(t) \approx \frac{x(t+\Delta t) - x(t-\Delta t)}{2\Delta t}, \quad a(t) \approx \frac{x(t+\Delta t) - 2x(t) + x(t-\Delta t)}{\Delta t^2}, \qquad (3.22)$$

where Δt is a small time interval. As $\Delta t \to 0$, the above formulas approach the first and second derivative of $x(t)$, which coincide with the well-known definitions of velocity and acceleration.

Write a function `kinematics(x, t, dt=1E-6)` for computing x, v, and a time `t`, using the above formulas for v and a with Δt corresponding to `dt`. Let the function return x, v, and a. Test the function with the position function $x(t) = e^{-(t-4)^2}$ and the time point $t = 5$ (use $\Delta t = 10^{-5}$). Name of program: `kinematics1.py`. ◇

Exercise 3.21. *Compute velocity and acceleration from position data; two dimensions.*

An object moves a long a path in the xy plane such that at time t the object is located at the point $(x(t), y(t))$. The velocity vector in the plane, at time t, can be approximated as

$$v(t) \approx \left(\frac{x(t+\Delta t) - x(t-\Delta t)}{2\Delta t}, \frac{y(t+\Delta t) - y(t-\Delta t)}{2\Delta t} \right) . \tag{3.23}$$

The acceleration vector in the plane, at time t, can be approximated as

$$a(t) \approx \left(\frac{x(t+\Delta t) - 2x(t) + x(t-\Delta t)}{\Delta t^2}, \frac{y(t+\Delta t) - 2y(t) + y(t-\Delta t)}{\Delta t^2} \right) . \tag{3.24}$$

Here, Δt is a small time interval. As $\Delta t \to 0$, we have that $v(t) = (x'(t), y'(t))$ and $a(t) = (x''(t), y''(t))$.

Make a function `kinematics(x, y, t, dt=1E-6)` for computing the velocity and acceleration of the object according to the formulas above (`t` corresponds to t, and `dt` corresponds to Δt). The function should return three 2-tuples holding the position, the velocity, and the acceleration, all at time `t`. Test the function for the motion along a circle with radius R and absolute velocity $R\omega$: $x(t) = R\cos\omega t$ and $y(t) = R\sin\omega t$. Compute the velocity and acceleration for $t = 1$ using $R = 1$, $\omega = 2\pi$, and $\Delta t = 10^{-5}$. Name of program: `kinematics2.py`. ◇

Exercise 3.22. *Find the max and min values of a function.*

Write a function `maxmin(f, a, b, n=1000)` for finding the maximum and minimum values of a mathematical function `f(x)` in the interval between `a` and `b`. The following test program

```
from math import cos, pi
print maxmin(cos, -pi/2, 2*pi, 100001)
```

should write out `(1.0, -1.0)`.

The `maxmin` function can compute a set of `n` uniformly spaced coordinates between `a` and `b` stored in a list `x`, then compute `f` at the points in `x` and store the values in another list `y`. The Python functions `max(y)` and `min(y)` return the maximum and minimum values in the list `y`, respectively. Name of program file: `maxmin_f.py`. ◇

Exercise 3.23. *Find the max and min elements in a list.*

Given a list `a`, the `max` function in Python's standard library computes the largest element in `a`: `max(a)`. Similarly, `min(a)` returns the smallest element in `a`. The purpose of this exercise is to write your own `max` and `min` function. Use the following technique: Initialize a variable `max_elem` by the first element in the list, then visit all the remaining elements (`a[1:]`), compare each element to `max_elem`, and if greater, make `max_elem` refer to that element. Use a similar technique to compute the minimum element. Collect the two pieces of code in functions. Name of program file: `maxmin_list.py`. ◇

Exercise 3.24. *Implement the Heaviside function.*

The following "step" function is known as the Heaviside function and is widely used in mathematics:

$$H(x) = \begin{cases} 0, \, x < 0 \\ 1, \, x \geq 0 \end{cases} \tag{3.25}$$

Implement $H(x)$ in a Python function `H(x)`. Test your implementation for $x = -\frac{1}{2}, 0, 10$. Name of program file: `Heaviside.py`. ◇

Exercise 3.25. *Implement a smoothed Heaviside function.*

The Heaviside function (3.25) listed in Exercise 3.24 is discontinuous. It is in many numerical applications advantageous to work with a smooth version of the Heaviside function where the function itself and its first derivative are continuous. One such smoothed Heaviside function is

$$H_\epsilon(x) = \begin{cases} 0, & x < -\epsilon, \\ \frac{1}{2} + \frac{x}{2\epsilon} + \frac{1}{2\pi}\sin\left(\frac{\pi x}{\epsilon}\right), & -\epsilon \leq x \leq \epsilon \\ 1, & x > \epsilon \end{cases} \tag{3.26}$$

Implement $H_\epsilon(x)$ in a Python function `H_eps(x)`. Name of program file: `smoothed_Heaviside.py`. ◇

Exercise 3.26. *Implement an indicator function.*

In many applications there is need for an indicator function, which is 1 over some interval and 0 elsewhere. More precisely, we define

$$I(x; L, R) = \begin{cases} 1, \, x \in [L, R], \\ 0, \, \text{elsewhere} \end{cases} \tag{3.27}$$

Make two Python implementations of such an indicator function, one with a direct test if $x \in [L, R]$ and one which expresses the indicator function in terms of Heaviside functions (3.25):

$$I(x; L, R) = H(x - L)H(R - x)\,. \tag{3.28}$$

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

Exercise 3.27. *Implement a piecewise constant function.*

Piecewise constant functions have a lot of important applications when modeling physical phenomena by mathematics. A piecewise constant function can be defined as

$$f(x) = \begin{cases} v_0, x \in [x_0, x_1], \\ v_1, x \in [x_1, x_2], \\ \vdots \\ v_i \quad x \in [x_i, x_{i+1}], \\ \vdots \\ v_n \quad x \in [x_n, x_{n+1}] \end{cases} \tag{3.29}$$

That is, we have a union of non-overlapping intervals covering the domain $[x_0, x_{n+1}]$, and $f(x)$ is constant in each interval. One example is the function that is -1 on $[0, 1]$, 0 on $[1, 1.5]$, and 4 on $[1.5, 2]$, where we with the notation in (3.29) have $x_0 = 0, x_1 = 1, x_2 = 1.5, x_3 = 2$ and $v_0 = -1, v_1 = 0, v_3 = 4$.

Make a function `piecewise(x, data)` for evaluating a piecewise constant function as in (3.29) at the point `x`. The `data` object is a list of pairs (v_i, x_i) for $i = 0, \ldots, n$. For example, `data` is `[(0, -1), (1, 0), (1.5, 4)]` in the example listed above. Since x_{n+1} is not a part of the `data` object, we have no means for detecting whether `x` is to the right of the last interval $[x_n, x_{n+1}]$, i.e., we must assume that the user of the `piecewise` function sends in an $x \le x_{n+1}$. As always, make an additional function for testing that the `piecewise` function works well. Name of program file: `piecewise_constant1.py`. ◇

Exercise 3.28. *Apply indicator functions.*

Implement piecewise constant functions, as defined in Exercise 3.27, by observing that

$$f(x) = \sum_{i=0}^{n} v_i I(x; x_i, x_{i+1}), \tag{3.30}$$

where $I(x; x_i, x_{i+1})$ is the indicator function from Exercise 3.26. Name of program file: `piecewise_constant2.py`. ◇

Exercise 3.29. *Rewrite a mathematical function.*

We consider the $L(x; n)$ sum as defined in Chapter 3.1.4 and the corresponding function `L2(x, epsilon)` function from Chapter 3.1.6. The sum $L(x; n)$ can be written as

$$L(x; n) = \sum_{i=1}^{n} c_i, \quad c_i = \frac{1}{i}\left(\frac{x}{1+x}\right)^i .$$

Derive a relation between c_i and c_{i-1},

$$c_i = ac_{i-1},$$

where a is an expression involving i and x. This relation between c_i and c_{i-1} means that we can start with `term` as c_1, and then in each pass of the loop implementing the sum $\sum_i c_i$ we can compute the next term c_i in the sum as

```
term = a*term
```

Rewrite the L2 function to make use of this alternative computation. Compare the new version with the original one to verify the implementation. Name of program file: `L2_recursive.py`. ◇

Exercise 3.30. *Make a table for approximations of* $\cos x$*.*

The function $\cos(x)$ can be approximated by the sum

$$C(x; n) = \sum_{j=0}^{n} c_j, \tag{3.31}$$

where

$$c_j = -c_{j-1}\frac{x^2}{2j(2j-1)}, \quad j = 1, 2, \ldots, n,$$

and $c_0 = 1$. Make a Python function for computing $C(x; n)$. (Hint: Represent c_j by a variable `term`, make updates `term = -term*...` inside a `for` loop, and accumulate the `term` variable in a variable for the sum.)

Also make a function for writing out a table of the errors in the approximation $C(x; n)$ of $\cos(x)$ for some x and n values given as arguments to the function. Let the x values run downward in the rows and the n values to the right in the columns. For example, a table for $x = 4\pi, 6\pi, 8\pi, 10\pi$ and $n = 5, 25, 50, 100, 200$ can look like

```
     x          5          25          50         100         200
  12.5664   1.61e+04   1.87e-11    1.74e-12    1.74e-12    1.74e-12
  18.8496   1.22e+06   2.28e-02    7.12e-11    7.12e-11    7.12e-11
  25.1327   2.41e+07   6.58e+04   -4.87e-07   -4.87e-07   -4.87e-07
  31.4159   2.36e+08   6.52e+09    1.65e-04    1.65e-04    1.65e-04
```

Observe how the error increases with x and decreases with n. Name of program file: `cossum.py`. ◇

Exercise 3.31. *Use None in keyword arguments.*

Consider the functions `L(x, n)` and `L2(x, epsilon)` from Chapter 3.1.6, whose program code is found in the file `lnsum.py`. Let us make a more flexible function `L3` where we can either specify a tolerance `epsilon` or a number of terms `n` in the sum, and we can choose whether we want the sum to be returned or the sum and the number of terms. The latter set of return values is only meaningful with `epsilon` and not `n` is specified. The starting point for all this flexibility is to have some keyword arguments initialized to an "undefined" value that can be recognized:

```
def L3(x, n=None, epsilon=None, return_n=False):
```

You can test if `n` is given using the phrase[6]

[6] One can also apply `if n != None`, but the `is` operator is most common (it tests if `n` and `None` are identical objects, not just objects with equal contents).

```
    if n is not None:
```

A similar construction can be used for `epsilon`. Print error messages for incompatible settings when `n` *and* `epsilon` are `None` (none given) or `not None` (both given). Name of program file: `L3_flexible.py`. ◇

Exercise 3.32. *Write a sort function for a list of 4-tuples.*

Below is a list of the nearest stars and some of their properties. The list elements are 4-tuples containing the name of the star, the distance from the sun in light years, the apparent brightness, and the luminosity. The apparent brightness is how bright the stars look in our sky compared to the brightness of Sirius A. The luminosity, or the true brightness, is how bright the stars would look if all were at the same distance compared to the Sun. The list data are found in the file `stars.list`, which looks as follows:

```
data = [
('Alpha Centauri A',    4.3,  0.26,      1.56),
('Alpha Centauri B',    4.3,  0.077,     0.45),
('Alpha Centauri C',    4.2,  0.00001,   0.00006),
("Barnard's Star",      6.0,  0.00004,   0.0005),
('Wolf 359',            7.7,  0.000001,  0.00002),
('BD +36 degrees 2147', 8.2,  0.0003,    0.006),
('Luyten 726-8 A',      8.4,  0.000003,  0.00006),
('Luyten 726-8 B',      8.4,  0.000002,  0.00004),
('Sirius A',            8.6,  1.00,      23.6),
('Sirius B',            8.6,  0.001,     0.003),
('Ross 154',            9.4,  0.00002,   0.0005),
]
```

The purpose of this exercise is to sort this list with respect to distance, apparent brightness, and luminosity.

To sort a list `data`, one can call `sorted(data)`, which returns the sorted list (cf. Table 2.1). However, in the present case each element is a 4-tuple, and the default sorting of such 4-tuples result in a list with the stars appearing in alphabetic order. We need to sort with respect to the 2nd, 3rd, or 4th element of each 4-tuple. If a tailored sort mechanism is necessary, we can provide our own sort function as a second argument to `sorted`, as in `sorted(data, mysort)`. Such a tailored sort function `mysort` must take two arguments, say `a` and `b`, and returns -1 if `a` should become before `b` in the sorted sequence, 1 if `b` should become before `a`, and 0 if they are equal. In the present case, `a` and `b` are 4-tuples, so we need to make the comparison between the right elements in `a` and `b`. For example, to sort with respect to luminosity we write

```
def mysort(a, b):
    if a[3] < b[3]:
        return -1
    elif a[3] > b[3]:
        return 1
    else:
        return 0
```

Write the complete program which initializes the data and writes out three sorted tables: star name versus distance, star name versus apparent brightness, and star name versus luminosity. Name of program file: `sorted_stars_data.py`. ◇

Exercise 3.33. *Find prime numbers.*

The *Sieve of Eratosthenes* is an algorithm for finding all prime numbers less than or equal to a number N. Read about this algorithm on Wikipedia and implement it in a Python program. Name of program file: `find_primes.py`. ◇

Exercise 3.34. *Find pairs of characters.*

Write a function `count_pairs(dna, pair)` that returns the number of occurrences of a pair of characters (`pair`) in a DNA string (`dna`). For example, calling the function with `dna` as `'ACTGCTATCCATT'` and `pair` as `'AT'` will return 2. Name of program file: `count_pairs.py`. ◇

Exercise 3.35. *Count substrings.*

This is an extension of Exercise 3.34: count how many times a certain string appears in another string. For example, the function returns 2 when called with the DNA string `'ACGTTACGGAACG'` and the substring `'ACG'`. Hint: For each match of the first character of the substring in the main string, check if the next `n` characters in the main string matches the substring, where `n` is the length of the substring. Use slices like `s` to pick out a substring of `s`. Name of program file: `count_substr.py`. ◇

Exercise 3.36. *Explain why a program works.*

The following program is quite similar to the program in Exercise 1.15:

```
def add(A, B):
    C = A + B
    return C

A = 3
B = 2
print add(A, B)
```

Explain how and thereby why the above program works. ◇

Exercise 3.37. *Resolve a problem with a function.*

Consider the following interactive session:

```
>>> def f(x):
...     if 0 <= x <= 2:
...         return x**2
...     elif 2 < x <= 4:
...         return 4
...     elif x < 0:
...         return 0
...
>>> f(2)
4
>>> f(5)
>>> f(10)
```

Why do we not get any output when calling `f(5)` and `f(10)`? (Hint: Save the `f` value in a variable `r` and write `print r`.) ◇

Exercise 3.38. *Determine the types of some objects.*
Consider the following calls to the `makelist` function from page 92:

```
l1 = makelist(0, 100, 1)
l2 = makelist(0, 100, 1.0)
l3 = makelist(-1, 1, 0.1)
l4 = makelist(10, 20, 20)
l5 = makelist([1,2], [3,4], [5])
l6 = makelist((1,-1,1), ('myfile.dat', 'yourfile.dat'))
l7 = makelist('myfile.dat', 'yourfile.dat', 'herfile.dat')
```

Determine in each case what type of objects that become elements in the returned list and what the contents of `value` is after one pass in the loop.

Hint: Simulate the program by hand and check out in an interactive session what type of objects that result from the arithmetics. It is only necessary to simulate one pass of the loop to answer the questions. Some of the calls will lead to infinite loops if you really execute the `makelist` calls on a computer.

This exercise demonstrates that we can write a function and have in mind certain types of arguments, here typically `int` and `float` objects. However, the function can be used with other (originally unintended) arguments, such as lists and strings in the present case, leading to strange and irrelevant behavior (the problem here lies in the boolean expression `value <= stop` which is meaningless for some of the arguments).

◇

Exercise 3.39. *Explain the difference between if and elif.*
Consider the following code:

```
def where1(x, y):
    if x > 0:
        print 'quadrant I or IV'
    if y > 0:
        print 'quadrant I or II'

def where2(x, y):
    if x > 0:
        print 'quadrant I or IV'
    elif y > 0:
        print 'quadrant II'

for x, y in (-1, 1), (1, 1):
    where1(x,y)
    where2(x,y)
```

What is printed? ◇

Exercise 3.40. *Find an error in a program.*
Consider the following program for computing

$$f(x) = e^{rx}\sin(mx) + e^{sx}\sin(nx),$$

```
def f(x, m, n, r, s):
    return expsin(x, r, m) + expsin(x, s, n)

x = 2.5
print f(x, 0.1, 0.2, 1, 1)

from math import exp, sin

def expsin(x, p, q):
    return exp(p*x)*sin(q*x)
```

Running this code results in

```
NameError: global name 'expsin' is not defined
```

What is the problem? Simulate the program flow by hand or use the debugger to step from line to line. Correct the program. ◇

Exercise 3.41. *Find programming errors.*

What is wrong in the following code segments? Try first to find the errors in each case by visual inspection of the code. Thereafter, type in the code snippet and test it out in an interactive Python shell.

Case 1:

```
def f(x)
    return 1+x**2;
```

Case 2:

```
def f(x):
    term1 = 1
  term2 = x**2
  return term1 + term2
```

Case 3:

```
def f(x, a, b):
    return a + b*x

print f(1), f(2), f(3)
```

Case 4:

```
def f(x, w):
    from math import sin
    return sin(w*x)

f = 'f(x, w)'
w = 10
x = 0.1
print f(x, w)
```

Case 5:

```
from math import *

def log(message):
    print message

print 'The logarithm of 1 is', log(1)
```

Case 6:

```
import time

def print_CPU_time():
    print 'CPU time so far in the program:', time.clock()

print_CPU_time
```

◇

4.7 Exercises

Exercise 4.1. *Make an interactive program.*

Make a program that (i) asks the user for a temperature in Fahrenheit degrees and reads the number; (ii) computes the corresponding temperature in Celsius degrees; and (iii) prints out the temperature in the Celsius scale. Name of program file: `f2c_qa.py`. ◇

Exercise 4.2. *Read from the command line in Exer. 4.1.*

Modify the program from Exercise 4.1 such that the Fahrenheit temperature is read from the command line. Name of program file: `f2c_cml.py`. ◇

Exercise 4.3. *Use exceptions in Exer. 4.2.*

Extend the program from Exercise 4.2 with a `try-except` block to handle the potential error that the Fahrenheit temperature is missing on the command line. Name of program file: `f2c_cml_exc.py`. ◇

Exercise 4.4. *Read input from the keyboard.*

Make a program that asks for input from the user, applies `eval` to this input, and prints out the type of the resulting object and its value. Test the program by providing five types of input: an integer, a real number, a complex number, a list, and a tuple. Name of program file: `objects_qa.py`. ◇

Exercise 4.5. *Read input from the command line.*

Let a program store the result of applying the `eval` function to the first command-line argument. Print out the resulting object and its type. Run the program with different input: an integer, a real number, a list, and a tuple. (On Unix systems you need to surround the tuple expressions in quotes on the command line to avoid error message from the Unix shell.) Try the string `"this is a string"` as a command-line argument. Why does this string cause problems and what is the remedy? Name of program file: `objects_cml.py`. ◇

Exercise 4.6. *Prompt the user for input to a formula.*

Consider the simplest program for evaluating the formula $y(t) = v_0t - 0.5gt^2$:

```
v0 = 3; g = 9.81; t = 0.6
y = v0*t - 0.5*g*t**2
print y
```

Modify this code so that the program asks the user questions `t=?` and `v0=?`, and then gets `t` and `v0` from the user's input through the keyboard. Name of program file: `ball_qa.py`. ◇

Exercise 4.7. *Initialize a formula from command-line arguments.*

Modify the program listed in Exercise 4.6 such that `v0` and `t` are read from the command line. Name of program file: `ball_cml.py`. ◇

Exercise 4.8. *Make the program from Exer. 4.7 safer.*

The program from Exercise 4.7 reads input from the command line. Extend that program with exception handling such that missing command-line arguments are detected. In the `except IndexError` block, use the `raw_input` function to ask the user for missing input data. Name of program file: `ball_cml_qa.py`. ◇

Exercise 4.9. *Test more in the program from Exer. 4.7.*

Test if the `t` value read in the program from Exercise 4.7 lies between 0 and $2v_0/g$. If not, print a message and abort execution. Name of program file: `ball_cml_errorcheck.py`. ◇

Exercise 4.10. *Raise an exception in Exer. 4.9.*

Instead of printing an error message and aborting the program explicitly, raise a `ValueError` exception in the `if` test on legal `t` values in the program from Exercise 4.9. Include the legal interval for t in the exception message. Name of program file: `ball_cml_ValueError.py`. ◇

Exercise 4.11. *Compute the distance it takes to stop a car.*

A car driver, driving at velocity v_0, suddenly puts on the brake. What braking distance d is needed to stop the car? One can derive, using Newton's second law of motion or a corresponding energy equation, that

$$d = \frac{1}{2}\frac{v_0^2}{\mu g}\,. \tag{4.7}$$

Make a program for computing d in (4.7) when the initial car velocity v_0 and the friction coefficient μ are given on the command line. Run the program for two cases: $v_0 = 120$ and $v_0 = 50$ km/h, both with $\mu = 0.3$ (μ is dimensionless). (Remember to convert the velocity from km/h to m/s before inserting the value in the formula!) Name of program file: `stopping_length.py`. ◇

Exercise 4.12. *Look up calendar functionality.*

The purpose of this exercise is to make a program which takes a date, consisting of year (4 digits), month (2 digits), and day (1-31) on the command line and prints the corresponding name of the weekday (Monday, Tuesday, etc.). Python has a module `calendar`, which you must look up in the Python Library Reference (see Chapter 2.6.3), for calculating the weekday of a date. Name of program file: `weekday.py`. ◇

Exercise 4.13. *Use the StringFunction tool.*

Make the program `user_formula.py` from Chapter 4.1.3 shorter by using the convenient `StringFunction` tool from Chapter 4.1.4. Name of program file: `user_formula2.py`. ◇

Exercise 4.14. *Extend a program from Ch. 4.2.1.*

How can you modify the `add_cml.py` program from the end of Chapter 4.1.2 such that it accepts input like `sqrt(2)` and `sin(1.2)`? In this case the output should be

```
<type 'float'> + <type 'float'> becomes <type 'float'>
with value 2.34625264834
```

Hint: Mathematical functions like `sqrt` and `sin` must be defined in the program before using `eval`. Furthermore, Unix (`bash`) does not like the parentheses on the command line so you need to put quotes around the command-line arguments. Name of program file: `add_cml2.py`. ◇

Exercise 4.15. *Why we test for specific exception types.*

The simplest way of writing a `try-except` block is to test for *any* exception, for example,

```
try:
    C = float(sys.arg[1])
except:
    print 'C must be provided as command-line argument'
    sys.exit(1)
```

Write the above statements in a program and test the program. What is the problem?

The fact that a user can forget to supply a command-line argument when running the program was the original reason for using a `try` block. Find out what kind of exception that is relevant for this error and test for this specific exception and re-run the program. What is the problem now? Correct the program. Name of program file: `cml_exception.py`. ◇

Exercise 4.16. *Make a simple module.*

Make six conversion functions between temperatures in Celsius, Kelvin, and Fahrenheit: `C2F`, `F2C`, `C2K`, `K2C`, `F2K`, and `K2F`. Collect these functions in a module `convert_temp`. Import the module in an interactive Python shell and demonstrate some sample calls on temperature conversions. Name of program file: `convert_temp.py`. ◇

Exercise 4.17. *Make a useful main program for Exer. 4.16.*

Extend the module made in Exercise 4.16 with a main program in the test block. This main program should read the first command-line argument as a numerical value of a temperature and the second argument as a temperature scale: `C`, `K`, or `F`. Write out the temperature in the other two scales. For example, if `21.3 C` is given on the command line, the output should be `70.34 F 294.45 K`. Name of program file: `convert_temp2.py`. ◇

Exercise 4.18. *Make a module in Exer. 3.13.*

Collect the `f` and `S` functions in the program from Exercise 3.13 in a separate file such that this file becomes a module. Put the statements making the table (i.e., the main program from Exercise 3.13) in a separate function `table(n_values, alpha_values, T)`, and call this function only if the module file is run as a program (i.e., call `table` from a test block, see Chapter 4.5.2). Name of program file: `sinesum2.py`. ◇

Exercise 4.19. *Extend the module from Exer. 4.18.*
Extend the program from Exercise 4.18 such that T and a series of n and α values are read from the command line. The extended program should import the `table` function from the module `sinesum2` (and not copy any code from the module file). Name of program file: `sinesum3.py`. ◇

Exercise 4.20. *Use options and values in Exer. 4.19.*
Let the input to the program in Exercise 4.19 be option-value pairs of the type `-n`, `-alpha`, and `-T`, with sensible default values for these quantities set in the program. Apply the `argparse` module to read the command-line arguments. Name of program file: `sinesum4.py`. ◇

Exercise 4.21. *Check if mathematical identities hold on a computer.*
Because of round-off errors, it could happen that a mathematical rule like $(ab)^3 = a^3b^3$ does not hold (exactly) on a computer. The idea of this exercise is to check such identities for a large number of random numbers. We can make random numbers using the `random` module in Python:

```
import random
a = random.uniform(A, B)
b = random.uniform(A, B)
```

Here, `a` and `b` will be random numbers which are always larger than or equal to `A` and smaller than `B`.

Make a program that reads the number of tests to be performed from the command line. Set `A` and `B` to fixed values (say -100 and 100). Perform the test in a loop. Inside the loop, draw random numbers `a` and `b` and test if the two mathematical expressions `(a*b)**3` and `a**3*b**3` are equivalent. Count the number of failures of equivalence and write out the percentage of failures at the end of the program.

Duplicate the code segment outlined above to also compare the expressions a/b and $1/(b/a)$. Name of program file: `math_identities_failures.py`. ◇

Exercise 4.22. *Improve input to the program in Exer. 4.21.*
The purpose of this exercise is to extend the program from Exercise 4.21 to handle a large number of mathematical identities. Make a function `equal(expr1, expr2, A, B, n=500)` which tests if the mathematical expressions `expr1` and `expr2`, given as strings and involving numbers `a` and `b`, are exactly equal (`eval(expr1) == eval(expr2)`) for `n` random choices of numbers `a` and `b` in the interval between `A` and `B`. Return the percentage of failures. Make a module with the `equal` function

and a test block which feeds the `equal` function with arguments read from the command line. Run the module file as a program to test the two identities from Exercise 4.21. Also test the identities $e^{a+b} = e^a e^b$ and $\ln a^b = b \ln a$ (take a `from math import *` in the module file so that mathematical functions like `exp` and `log` are defined). Name of program file: `math_identities_failures_cml.py`. ◇

Exercise 4.23. *Apply the program from Exer. 4.22.*

Import the `equal` function from the module made in Exercise 4.22 and test the three identities from Exercise 4.21 in addition to the following identities:

- $a - b$ and $-(b - a)$
- a/b and $1/(b/a)$
- $(ab)^4$ and a^4b^4
- $(a + b)^2$ and $a^2 + 2ab + b^2$
- $(a + b)(a - b)$ and $a^2 - b^2$
- e^{a+b} and $e^a e^b$
- $\ln a^b$ and $b \ln a$
- $\ln ab$ and $\ln a + \ln b$
- ab and $e^{\ln a + \ln b}$
- $1/(1/a + 1/b)$ and $ab/(a + b)$
- $a(\sin^2 b + \cos^2 b)$ and a
- $\sinh(a + b)$ and $(e^a e^b - e^{-a} e^{-b})/2$
- $\tan(a + b)$ and $\sin(a + b)/\cos(a + b)$
- $\sin(a + b)$ and $\sin a \cos b + \sin b \cos a$

Store all the expressions in a list of 2-tuples, where each 2-tuple contains two mathematically equivalent expressions as strings which can be sent to the `eval` function. Make a nicely formatted table with a pair of equivalent expressions at each line followed by the failure rate. Try out `A=0` and `B=1` as well as `A=-1E+7` and `B=1E+7`. Does the failure rate seem to depend on the magnitude of the numbers a and b? Name of program file: `math_identities_failures_table.py`.

Remark. Exercise 4.21 can be solved by a simple program, but if you want to check 17 identities the present exercise demonstrates how important it is to be able to automate the process via the `equal` function and two nested loops over a list of equivalent expressions. ◇

Exercise 4.24. *Compute the binomial distribution.*

Consider an uncertain event where there are two outcomes only, typically success or failure. Flipping a coin is an example: The outcome is uncertain and of two types, either head (can be considered as success) or tail (failure). Throwing a die can be another example, if (e.g.) getting a six is considered success and all other outcomes represent failure. Let the probability of success be p and that of failure $1-p$. If we perform n experiments, where the outcome of each experiment does not depend on the outcome of previous experiments, the probability of getting success x times (and failure $n - x$ times) is given by

$$B(x, n, p) = \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x} \,. \tag{4.8}$$

This formula (4.8) is called the binomial distribution. The expression $x!$ is the factorial of x as defined in Exercise 3.19. Implement (4.8) in a function `binomial(x, n, p)`. Make a module containing this `binomial` function. Include a test block at the end of the module file. Name of program file: `binomial_distribution.py`. ◇

Exercise 4.25. *Apply the binomial distribution.*

Use the module from Exercise 4.24 to make a program for solving the problems below.

1. What is the probability of getting two heads when flipping a coin five times?
   This probability corresponds to $n = 5$ events, where the success of an event means getting head, which has probability $p = 1/2$, and we look for $x = 2$ successes.
2. What is the probability of getting four ones in a row when throwing a die?
   This probability corresponds to $n = 4$ events, success is getting one and has probability $p = 1/6$, and we look for $x = 4$ successful events.
3. Suppose cross country skiers typically experience one ski break in one out of 120 competitions. Hence, the probability of breaking a ski can be set to $p = 1/120$. What is the probability b that a skier will experience a ski break during five competitions in a world championship?
   This question is a bit more demanding than the other two. We are looking for the probability of 1, 2, 3, 4 or 5 ski breaks, so it is simpler to ask for the probability c of *not* breaking a ski, and then compute $b = 1 - c$. Define "success" as breaking a ski. We then look for $x = 0$ successes out of $n = 5$ trials, with $p = 1/120$ for each trial. Compute b.

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

Exercise 4.26. *Compute probabilities with the Poisson distribution.*

Suppose that over a period of t_m time units, a particular uncertain event happens (on average) νt_m times. The probability that there will be x such events in a time period t is approximately given by the formula

$$P(x, t, \nu) = \frac{(\nu t)^x}{x!} e^{-\nu t} \,. \tag{4.9}$$

This formula is known as the Poisson distribution[11]. An important assumption is that all events are independent of each other and that the probability of experiencing an event does not change significantly over time.

Implement (4.9) in a function `Poisson(x, t, nu)`, and make a program that reads x, t, and ν from the command line and writes out the probability $P(x, t, \nu)$. Use this program to solve the problems below.

1. Suppose you are waiting for a taxi in a certain street at night. On average, 5 taxis pass this street every hour at this time of the night. What is the probability of not getting a taxi after having waited 30 minutes?
   Since we have 5 events in a time period of $t_m = 1$ hour, $\nu t_m = \nu = 5$. The sought probability is then $P(0, 1/2, 5)$. Compute this number. What is the probability of having to wait two hours for a taxi?
   If 8 people need two taxis, that is the probability that two taxis arrive in a period of 20 minutes?
2. In a certain location, 10 earthquakes have been recorded during the last 50 years. What is the probability of experiencing exactly three earthquakes over a period of 10 years in this area? What is the probability that a visitor for one week does not experience any earthquake?
   With 10 events over 50 years we have $\nu t_m = \nu \cdot 50$ years $= 10$ events, which implies $\nu = 1/5$ event per year. The answer to the first question of having $x = 3$ events in a period of $t = 10$ years is given directly by (4.9). The second question asks for $x = 0$ events in a time period of 1 week, i.e., $t = 1/52$ years, so the answer is $P(0, 1/52, 1/5)$.
3. Suppose that you count the number of misprints in the first versions of the reports you write and that this number shows an average of six misprints per page. What is the probability that a reader of a first draft of one of your reports reads six pages without hitting a misprint?
   Assuming that the Poisson distribution can be applied to this problem, we have "time" t_m as 1 page and $\nu \cdot 1 = 6$, i.e., $\nu = 6$ events (misprints) per page. The probability of no events in a "period" of six pages is $P(0, 6, 6)$.

◇

[11] It can be shown that (4.9) arises from (4.8) when the probability p of experiencing the event in a small time interval t/n is $p = \nu t/n$ and we let $n \to \infty$.

The scaled temperature has only one “free” parameter b. That is, the shape of the graph is completely determined by b.

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

We can reuse the `animate` function also in the scaled case, but we need to make a new $T(z,t)$ function and, e.g., a main program where b can be read from the command line:

```
def T(z, t):
    return exp(-b*z)*cos(t - b*z)  # b is global

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

Running the program, found as the file `heatwave_scaled.py`, for different b values shows that b governs how deep the temperature variations on the surface $z = 0$ penetrate. A large b makes the temperature changes confined to a thin layer close to the surface (see Figure 5.12 for $b = 20$), while a small b leads to temperature variations also deep down in the ground (see Figure 5.13 for $b = 2$).

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

5.9 Exercises

Exercise 5.1. *Fill lists with function values.*

Define

$$h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \,. \tag{5.15}$$

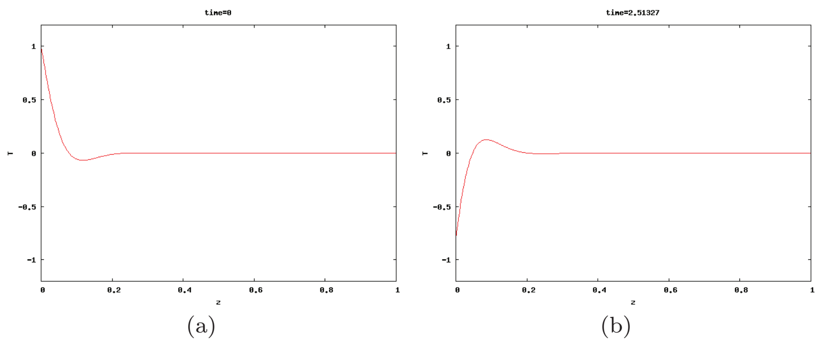

Fig. 5.12 Plot of the dimensionless temperature $T(z,t)$ in the ground for two different t values and $b = 20$.

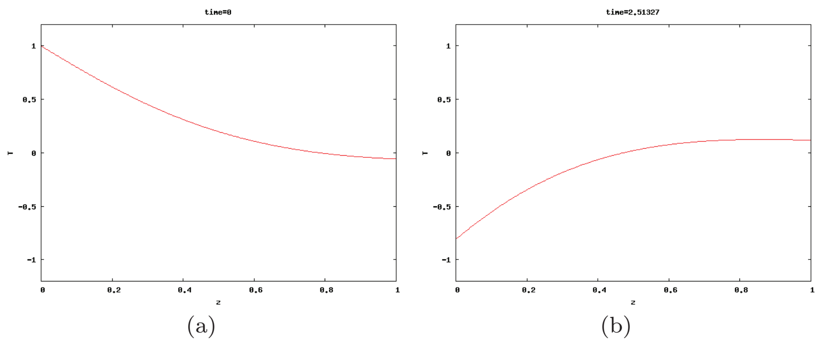

Fig. 5.13 Plot of the dimensionless temperature $T(z,t)$ in the ground for two different t values and $b = 2$.

Fill lists `xlist` and `hlist` with x and $h(x)$ values for 41 uniformly spaced x coordinates in $[-4, 4]$. Hint: You may adapt the example in Chapter 5.2.1. Name of program file: `fill_lists.py`. ◇

Exercise 5.2. *Fill arrays; loop version.*
The aim is to fill two arrays `x` and `y` with x and $h(x)$ values, respectively, where $h(x)$ is defined in (5.15). Let the x values be as in Exercise 5.1. Create empty arrays `x` and `y` arrays and compute each element in `x` and `y` with a `for` loop. Name of program file: `fill_arrays_loop.py`. ◇

Exercise 5.3. *Fill arrays; vectorized version.*
Vectorize the code in Exercise 5.2 by creating the x values using the `linspace` function from the `numpy` package and by evaluating $h(x)$ for an array argument. Name of program file: `fill_arrays_vectorized.py`. ◇

Exercise 5.4. *Plot a function.*
Make a plot of the function in Exercise 5.1 for $x \in [-4, 4]$. Name of program file: `plot_Gaussian.py`. ◇

Exercise 5.5. *Apply a function to a vector.*

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

Exercise 5.6. *Simulate by hand a vectorized expression.*

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

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

If `x` holds two elements, 0 and 2, and `t` holds the elements 1 and 1.5, calculate by hand (using a calculator) the `y` array. Thereafter, write a program that mimics the series of computations you did by hand (typically a sequence of operations of the kind we listed on page 192 – use explicit loops, but at the end you can use Numerical Python functionality to check the results). Name of program file: `simulate_vector_computing.py`. ◇

Exercise 5.7. *Demonstrate array slicing.*

Create an array `w` with values $0, 0.1, 0.2, \ldots, 3$. Write out `w[:]`, `w[:-2]`, `w[::5]`, `w[2:-2:6]`. Convince yourself in each case that you understand which elements of the array that are printed. Name of program file: `slicing.py`. ◇

Exercise 5.8. *Replace list operations by array computing.*

The data analysis problem in Chapter 2.6.2 is solved by list operations. Convert the list to a two-dimensional array and perform the three tasks using array operations. There should be no explicit loops in this Python program. Name of program file: `sun_data_vec.py`. ◇

Exercise 5.9. *Plot a formula.*

Make a plot of the function $y(t) = v_0 t - 0.5gt^2$ for $v_0 = 10$, $g = 9.81$, and $t \in [0, 2v_0/g]$. The label on the x axis should be 'time (s)' and the label on the y axis should be 'height (m)'. Name of program file: `plot_ball1.py`. ◇

Exercise 5.10. *Plot a formula for several parameters.*

Make a program that reads a set of v_0 values from the command line and plots the corresponding curves $y(t) = v_0 t - 0.5gt^2$ in the same figure (set $g = 9.81$). Let $t \in [0, 2v_0/g]$ for each curve, which implies that you need a different vector of t coordinates for each curve. Name of program file: `plot_ball2.py`. ◇

Exercise 5.11. *Specify the x and y axes in Exer. 5.10.*

Extend the program from Exercises 5.10 such that the minimum and maximum x and y values are computed, and use the extreme values to specify the extent of the x and y axes. Add some space above the highest curve. Name of program file: `plot_ball3.py`. ◇

Exercise 5.12. *Plot exact and inexact Fahrenheit–Celsius conversion formulas.*

Exercise 2.2 introduces a simple rule to quickly compute the Celsius temperature from the Fahrenheit degrees: $C = (F - 30)/2$. Compare this curve against the exact curve $C = (F-32)5/9$ in a plot. Let F vary between -20 and 120. Name of program file: `f2c_shortcut_plot.py`. ◇

Exercise 5.13. *Plot the trajectory of a ball.*

The formula for the trajectory of a ball is given in (1.5) on page 39. In a program, first read the input data y_0, θ, and v_0 from the command line. Then plot the trajectory $y = f(x)$ for $y \geq 0$. Name of program file: `plot_trajectory.py`. ◇

Exercise 5.14. *Implement Lagrange's interpolation formula.*

Imagine we have $n+1$ measurements of some quantity y that depends on x: $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$. We may think of y as a function of x and ask what y is at some arbitrary point x not coinciding with any of the points $x_0, \ldots, x_n$. This problem is known as *interpolation*. One way to solve this problem is to fit a continuous function that goes through all the $n+1$ points and then evaluate this function for any desired x. A candidate for such a function is the polynomial of degree n that goes through all the points. This polynomial can be written

$$p_L(x) = \sum_{k=0}^{n} y_k L_k(x), \tag{5.16}$$

where

$$L_k(x) = \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i}\,. \tag{5.17}$$

The $\prod$ notation corresponds to $\sum$, but the terms are multiplied. For example,

$$\prod_{i=0, i\neq k}^{n} x_i = x_0 x_1 \cdots x_{k-1} x_{k+1} \cdots x_n\,.$$

The polynomial $p_L(x)$ is known as Lagrange's interpolation formula, and the points $(x_0, y_0), \ldots, (x_n, y_n)$ are called interpolation points.

Make functions `p_L(x, xp, yp)` and `L_k(x, k, xp, yp)` that evaluate $p_L(x)$ and $L_k(x)$ by (5.16) and (5.17), respectively, at the point `x`. The arrays `xp` and `yp` contain the x and y coordinates of the $n+1$ interpolation points, respectively. That is, `xp` holds $x_0, \ldots, x_n$, and `yp` holds $y_0, \ldots, y_n$.

To verify the program, we observe that $L_k(x_k) = 1$ and that $L_k(x_i) = 0$ for $i \neq k$, implying that $p_L(x_k) = y_k$. That is, the polynomial p_L goes through all the points $(x_0, y_0), \ldots, (x_n, y_n)$. Write a function `verify(xp, yp)` that computes $|p_L(x_k) - y_k|$ at all the interpolation points (x_k, y_k) and checks that the value is approximately zero. Call `verify` with `xp` and `yp` corresponding to 5 equally spaced points along the curve $y = \sin(x)$ for $x \in [0, \pi]$. Thereafter, evaluate $p_L(x)$ for an x in the middle of two interpolation points and compare the value of $p_L(x)$ with the exact one, $\sin(x)$. Name of program file: `Lagrange_poly1.py`. ◇

Exercise 5.15. *Plot the polynomial in Exer. 5.14.*

Write a function `graph(f, n, xmin, xmax, resolution=1001)` for plotting $p_L(x)$ in Exercise 5.14, based on interpolation points taken from some mathematical function $f(x)$ represented by the argument `f`. The argument `n` denotes the number of interpolation points sampled from the $f(x)$ function, and `resolution` is the number of points between `xmin` and `xmax` used to plot $p_L(x)$. The x coordinates of the `n` interpolation points can be uniformly distributed between `xmin` and `xmax`. In the graph, the interpolation points $(x_0, y_0), \ldots, (x_n, y_n)$ should be marked by small circles. Test the `graph` function by choosing 5 points in $[0, \pi]$ and `f` as $\sin x$.

Make a module `Lagrange_poly2` containing the `p_L`, `L_k`, `verify`, and `graph` functions. The call to `verify` described in Exercise 5.14 and the call to `graph` described above should appear in the module's test block. Name of program file: `Lagrange_poly2.py`. ◇

Exercise 5.16. *Investigate the polynomial in Exer. 5.14.*

Unfortunately, the polynomial $p_L(x)$ defined and implemented in Exercise 5.14 can exhibit some undesired oscillatory behavior which we shall explore graphically in this exercise. Compute $n + 1$ interpolation points taken from the curve $f(x) = |x|$ for $x \in [-2, 2]$: $x_k = -2 + 4k/n$ and $y_k = |x_k|$, $k = 0, 1, \ldots, n$. Call the `graph` function from Exercise 5.15. for $n = 2, 4, 6, 10$. All the graphs of $p_L(x)$ should appear in the same plot for comparison. In addition, make a new figure with calls to `graph` for $n = 13$ and $n = 20$. All the calls to `graph` should appear in some separate program file which imports the `Lagrange_poly2` module made in Exercise 5.15.

The purpose of the $p_L(x)$ function is to compute (x, y) between some given (often measured) data points $(x_0, y_0), \ldots, (x_n, y_n)$. We see from the graphs that for a small number of interpolation points, $p_L(x)$ is quite close to the curve $y = |x|$ we used to generate the data points, but as n increases, $p_L(x)$ starts to oscillate, especially toward the end points (x_0, y_0) and (x_n, y_n). Much research has historically been focused on methods that do not result in such strange oscillations when fitting a polynomial to a set of points. Name of program file: `Lagrange_poly2b.py`. ◇

Exercise 5.17. *Plot a wave packet.*

The function

$$f(x, t) = e^{-(x-3t)^2} \sin\left(3\pi(x - t)\right) \tag{5.18}$$

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

Exercise 5.18. *Judge a plot.*

Assume you have the following program for plotting a parabola:

```
import numpy as np
x = np.linspace(0, 2, 20)
y = x*(2 - x)
import matplotlib.pyplot as plt
plt.plot(x, y)
plt.show()
```

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

Exercise 5.19. *Plot the viscosity of water.*

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

$$\mu(T) = A \cdot 10^{B/(T-C)}, \tag{5.19}$$

where $A = 2.414 \cdot 10^{-5}$ Pa s, $B = 247.8$ K, and $C = 140$ K. Plot $\mu(T)$ for T between 0 and 100 degrees Celsius. Label the x axis with 'temperature (C)' and the y axis with 'viscosity (Pa s)'. Note that T in the formula for μ must be in Kelvin. Name of program file: `water_viscosity.py`. ◇

Exercise 5.20. *Explore a complicated function graphically.*

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

$$c(\lambda) = \sqrt{\frac{g\lambda}{2\pi}\left(1 + s\frac{4\pi^2}{\rho g \lambda^2}\right)\tanh\left(\frac{2\pi h}{\lambda}\right)}\,. \tag{5.20}$$

Here, g is the acceleration of gravity (981 cm/s), s is the air-water surface tension ($7.9 \cdot 10^{-4}$ N/cm) , ρ is the density of water (can be taken as 1 kg/cm^3), and h is the water depth in cm. Let us fix h at 5000 cm. First make a plot of $c(\lambda)$ (in cm/s) for small λ (0.1 cm to 10 cm). Then make a plot $c(\lambda)$ for larger λ (1 m to 2 km, but converted to cm since this cm is the length scale in the values for ρ, s, and g). Name of program file: `water_wave_velocity.py`. ◇

Exercise 5.21. *Plot Taylor polynomial approximations to* $\sin x$.

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

$$\sin x \approx S(x; n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!}\,. \tag{5.21}$$

The expression $(2j+1)!$ is the factorial (see Exercise 3.19). The error in the approximation $S(x;n)$ decreases as n increases and in the limit we have that $\lim_{n\to\infty} S(x;n) = \sin x$. The purpose of this exercise is to visualize the quality of various approximations $S(x;n)$ as n increases.

The first part of the exercise is to write a Python function `S(x, n)` that computes $S(x;n)$. Use a straightforward approach where you compute each term as it stands in the formula, i.e., $(-1)^j x^{2j+1}$ divided by the factorial $(2j+1)!$. (We remark that Exercise A.18 outlines a much more efficient computation of the terms in the series.)

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

Exercise 5.22. *Animate a wave packet.*

Display an animation of the function $f(x,t)$ in Exercise 5.17 by plotting f as a function of x on $[-6,6]$ for a set of t values in $[-1,1]$. Also make an animated GIF file. A suitable resolution can be 1000 intervals (1001 points) along the x axis, 60 intervals (61 points) in time, and 6 frames per second in the animated GIF file. Use the recipe in Chapter 5.3.4 and remember to remove the family of old plot files in the beginning of the program. Name of program file: `plot_wavepacket_movie.py`. ◇

Exercise 5.23. *Animate a smoothed Heaviside function.*

Visualize the smoothed Heaviside function $H_\epsilon(x)$, defined in (3.26) on page 128, as an animation where ϵ starts at 2 and then goes to zero. Name of program file: `smoothed_Heaviside_movie.py`. ◇

Exercise 5.24. *Animate two-scale temperature variations.*

We consider temperature oscillations in the ground as addressed in Chapter 5.8.2. Now we want to visualize daily and annual variations. Let A_1 be the amplitude of annual variations and A_2 the amplitude of the day/night variations. Let also $P_1 = 365$ days and $P_2 = 24$ h be the periods of the annual and the daily oscillations. The temperature at time t and depth z is then given by[16]

$$T(z,t) = T_0 + A_1 e^{-a_1 z}\sin(\omega_1 t - a_1 z) + A_2 e^{-a_2 z}\sin(\omega_2 t - a_2 z), \quad (5.22)$$

where

[16] Here we assume that the temperature T equals the reference temperature T_0 at $t=0$, resulting in a sine variation rather than the cosine variation in (5.13).

$$\begin{aligned}\omega_1 &= 2\pi P_1,\\ \omega_2 &= 2\pi P_2,\\ a_1 &= \sqrt{\frac{\omega_1}{2k}},\\ a_2 &= \sqrt{\frac{\omega_2}{2k}}\,.\end{aligned}$$

Choose $k = 10^{-6}$ m^2/s, $A_1 = 15$ C, $A_2 = 7$ C, and the resolution Δt as $P_2/10$. Modify the `heatwave.py` program in order to animate this new temperature function. Name of program file: `heatwave2.py`. ◇

Exercise 5.25. *Improve the solution in Exer. 5.24.*

Watching the animation in Exercise 5.24 reveals that there are rapid oscillations in a small layer close to $z = 0$. The variations away from $z = 0$ are much smaller in time and space. It would therefore be wise to use more z coordinates close to $z = 0$ than for larger z values. Given a set $x_0 < x_1 < \cdots < x_n$ of uniformly spaced coordinates in $[a, b]$, we can compute new coordinates $\bar{x}_i$, stretched toward $x = a$, by the formula

$$\bar{x}_i = a + (b-a)\left(\frac{x_i - a}{b - a}\right)^s,$$

for some $s > 1$. Use this formula to stretch the z coordinates to the left. Experiment with $s \in [1.2, 3]$ and few points (say 15) and visualize the curve as a line with circles at the points so that you can easily see the distribution of points toward the left end. Run the animation with no circles and (say) 501 points when a suitable s has been found.

We say that the new z coordinates are *adapted* to the curve, meaning that we have distributed the z coordinates where we need them most, i.e., where there are rapid variations of the curve. Name of program file: `heatwave2a.py`. ◇

Exercise 5.26. *Animate a sequence of approximations to π.*

Exercise 3.11 outlines an idea for approximating π as the length of a polygon inside the circle. Wrap the code from Exercise 3.11 in a function `pi_approx(N)` which returns the approximation to π using a polygon with $N+1$ equally distributed points. The task of the present exercise is to visually display the polygons as a movie, where each frame shows the polygon with $N+1$ points together with the circle and a title reflecting the corresponding error in the approximate value of π. The whole movie arises from letting N run through $4, 5, 6, \ldots, K$, where K is some (large) prescribed value. Let there be a pause of 0.3 s between each frame in the movie. By playing the movie you will see how the polygons move closer and closer to the circle and how the approximation to π improves. Name of program file: `pi_polygon_movie.py`. ◇

Exercise 5.27. *Animate a planet's orbit.*

A planet's orbit around a star has the shape of an ellipse. The purpose of this exercise is to make an animation of the movement along the orbit. One should see a small disk, representing the planet, moving along an elliptic curve. An evolving solid line shows the development of the planet's orbit as the planet moves.

The points (x, y) along the ellipse are given by the expressions

$$x = a\cos(\omega t), \quad y = b\sin(\omega t),$$

where a is the semimajor axis of the ellipse, b is the semiminor axis, ω is an angular velocity of the planet around the star, and t denotes time. One complete orbit corresponds to $t \in [0, 2\pi/\omega]$. Let us discretize time into time points $t_k = k\Delta t$, where $\Delta t = 2\pi/(\omega n)$. Each frame in the movie corresponds to (x, y) points along the curve with t values $t_0, t_1, \ldots, t_i$, i representing the frame number ($i = 1, \ldots, n$). Let the plot title of each frame display the planet's instantaneous velocity magnitude. This magnitude is the length of the velocity vector

$$(\frac{dx}{dt}, \frac{dy}{dt}) = (-\omega a\sin(\omega t), \omega b\cos(\omega t)),$$

which becomes $\omega\sqrt{a^2\sin^2(\omega t) + b^2\cos^2(\omega t)}$.

Implement the visualization of the planet's orbit using the method above. Run the special case of a circle and verify that the magnitude of the velocity remains constant as the planet moves. Name of program file: `planet_orbit.py`. ◇

Exercise 5.28. *Animate the evolution of Taylor polynomials.*

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

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

For example, the Taylor polynomial of degree N for e^x equals $S(x; 0, N)$ with $f_k(x) = x^k/k!$. The purpose of the exercise is to make a movie of how $S(x; M, N)$ develops (and hopefully improves as an approximation) as we add terms in the sum. That is, the frames in the movie correspond to plots of $S(x; M, M)$, $S(x; M, M + 1)$, $S(x; M, M + 2)$, $\ldots$, $S(x; M, N)$.

Make a function

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

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

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

In the beginning of the `animate_series` function, it is necessary to remove all old plot files of the form `tmp_*.png`. This can be done by the `glob` module and the `os.remove` function as exemplified in Chapter 5.3.4.

Test the `animate_series` function in two cases:

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

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

Exercise 5.29. *Plot the velocity profile for pipeflow.*

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

$$v(r) = \left(\frac{\beta}{2\mu_0}\right)^{1/n} \frac{n}{n+1}\left(R^{1+1/n} - r^{1+1/n}\right), \tag{5.23}$$

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

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

Exercise 5.30. *Plot the functions from Exer. 3.13.*

Exercise 3.13 defines the approximation $S(t;n)$ to a function $f(t)$. Plot $S(t;1)$, $S(t;3)$, $S(t;20)$, $S(t;200)$, and the exact $f(t)$ function in the same plot. Use $T = 2\pi$. Name of program file: `sinesum1_plot.py`. ◇

Exercise 5.31. *Make a movie of the functions from Exer. 3.13.*

First perform Exercise 5.30. A natural next step is to animate the evolution of $S(t;n)$ as n increases. Create such an animation and observe how the discontinuity in $f(t)$ is poorly approximated by $S(t;n)$, even when n grows large (plot $f(t)$ in each frame). This is a well-known deficiency, called Gibb's phenomenon, when approximating discontinuous functions by sine or cosine (Fourier) series. Name of program file: `sinesum1_movie.py`. ◇

Exercise 5.32. *Plot functions from the command line.*

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

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

A specific example is

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

Hint: Make x coordinates from the second and third command-line arguments and then use `eval` (or `StringFunction` from Chapters 4.1.4 and 5.5.1) on the first first argument. Try to write as short program as possible (we leave it to Exercise 5.33 to test for valid input). Name of program file: `plotf_v1.py`. ◇

Exercise 5.33. *Improve the program from Exercise 5.32.*

Equip the program from Exercise 5.32 with tests on valid input on the command line. Also allow an optional fourth command-line argument for the number of points along the function curve. Set this number to 501 if it is not given. Name of program file: `plotf.py`. ◇

Exercise 5.34. *Demonstrate energy concepts from physics.*

The vertical position $y(t)$ of a ball thrown upward is given by $y(t) = v_0 t - 0.5gt^2$, where g is the acceleration of gravity and v_0 is the velocity at $t = 0$. Two important physical quantities in this context are the potential energy, obtained by doing work against gravity, and the kinetic energy, arising from motion. The potential energy is defined as $P = mgy$, where m is the mass of the ball. The kinetic energy is defined as $K = \frac{1}{2}mv^2$, where v is the velocity of the ball, related to y by $v(t) = y'(t)$. Plot $P(t)$ and $K(t)$ in the same plot, along with their sum $P+K$. Let $t \in [0, 2v_0/g]$. Read m and v_0 from the command line. Run the program with various choices of m and v_0 and observe that $P + K$ is always constant in this motion. In fact, it turns out that $P + K$ is constant for a large class of motions, and this is a very important result in physics. Name of program file: `energy_physics.py`. ◇

Exercise 5.35. *Plot a w-like function.*
Define mathematically a function that looks like the 'w' character. Plot this function. Name of program file: `plot_w.py`. ◇

Exercise 5.36. *Plot a piecewise constant function.*
Consider the piecewise constant function defined in Exercise 3.27. Make a Python function `plot_piecewise(data, xmax)` which draws a graph of the function, where `data` is the nested list explained in Exercise 3.27 and `xmax` is the maximum x coordinate. Use ideas from Chapter 5.4.1. Name of program file: `plot_piecewise_constant.py`. ◇

Exercise 5.37. *Vectorize a piecewise constant function.*
Consider the piecewise constant function defined in Exercise 3.27. Make vectorized version implementation. Name of program file: `piecewise_constant_vec.py`. ◇

Exercise 5.38. *Visualize approximations in the Midpoint integration rule.*
Consider the midpoint rule for integration from Exercise 3.8. Use Matplotlib to make an illustration of the midpoint rule as shown in Figure 5.14a. Look up the documentation of the Matplotlib function `fill_between` and use this function to create the filled areas between $f(x)$ and the approximating rectangles. The $f(x)$ function used in Figure 5.14 is

$$f(x) = x(12 - x) + \sin(\pi x), \quad x \in [0, 10]\,.$$

Note that the `fill_between` requires the two curves to have the same number of points. For accurate visualization of $f(x)$ you need quite many x coordinates, and the rectangular approximation to $f(x)$ must be drawn using the same set of x coordinates. Name of program file: `viz_midpoint.py`. ◇

Exercise 5.39. *Visualize approximations in the Trapezoidal integration rule.*
Redo Exercise 5.38 for the Trapezoidal rule from Exercise 3.7 to produce the graph shown in Figure 5.14b. Name of program file: `viz_trapezoidal.py`. ◇

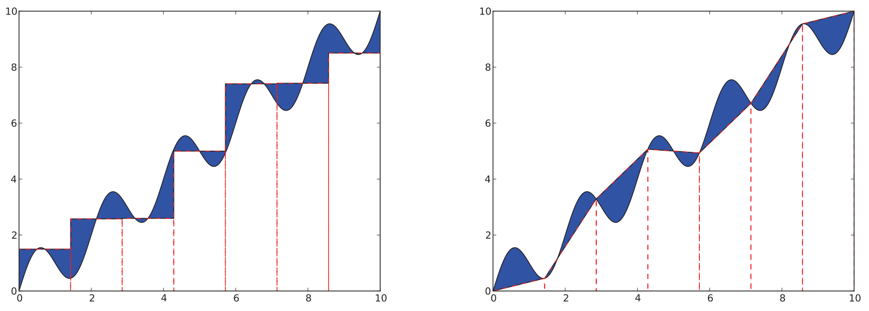

Fig. 5.14 Visualization of numerical integration rules: (a) the Midpoint rule; (b) the Trapezoidal rule. The filled areas illustrate the errors in the approximation of the area under the curve.

Exercise 5.40. *Experience overflow in a function.*

When an object (ball, car, airplane) moves through the air, there is a very, very thin layer of air close to the object's surface where the air velocity varies dramatically[17], from the same value as the velocity of the object at the object's surface to zero a few centimeters away. The change in velocity is quite abrupt and can be modeled by the functiion

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

where $x = 1$ is the object's surface, and $x = 0$ is some distance away where one cannot notice any wind velocity v because of the passing object ($v = 0$). The vind velocity coincides with the velocity of the object at $x = 1$, here set to $v = 1$. The parameter μ is very small and related to the viscosity of air. With a small value of μ, it becomes difficult to calculate $v(x)$ on a computer.

Make a function `v(x, mu=1E-6, exp=math.exp)` for calculating the formula for $v(x)$ using `exp` as a possibly user-given exponential function. Let the `v` function return the nominator and denominator in the formula as well as the fraction (result). Call the `v` function for various `x` values between 0 and 1 in a `for` loop, let `mu` be `1E-3`, and have an inner `for` loop over two different `exp` functions: `math.exp` and `numpy.exp`. The output will demonstrate how the denominator is subject to overflow and how difficult it is to calculate this function on a computer.

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

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

Exercise 5.41. *Experience less overflow in a function.*

In the program from Exercise 5.40, convert x and eps to a higher precision representation of real numbers, with the aid of the NumPy type float96:

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

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

Exercise 5.42. *Extend Exer. 5.5 to a rank 2 array.*

Let A be the two-dimensional array

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

Apply the function f from Exercise 5.5 to each element in A. Then calculate the result of the array expression $A**3 + A*e^A + 1$, and demonstrate that the end result of the two methods are the same. ◇

Exercise 5.43. *Explain why array computations fail.*

The following loop computes the array y from x:

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

However, the alternative loop

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

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

```
        grade  = course['grade']
        sum += grade2number[grade]*weight
        weights += weight
    avg = sum/float(weights)
    return number2grade[round(avg)]
```

The complete code is found in the file `students.py`. Running this program gives the following output of the average grades:

```
John Doe: B
Kari Nordmann: C
Jan Modaal: C
```

One feature of the `students.py` code is that the output of the names are sorted after the last name. How can we accomplish that? A straight `for name in data` loop will visit the keys in an unknown (random) order. To visit the keys in alphabetic order, we must use

```
for name in sorted(data):
```

This default sort will sort with respect to the first character in the `name` strings. We want a sort according to the last part of the name. A tailored sort function can then be written (see Exercise 3.32 for an introduction to tailored sort functions). In this function we extract the last word in the names and compare them:

```
def sort_names(name1, name2):
    last_name1 = name1.split()[-1]
    last_name2 = name2.split()[-2]
    if last_name1 < last_name2:
        return -1
    elif last_name1 > last_name2:
        return 1
    else:
        return 0
```

We can now pass on `sort_names` to the `sorted` function to get a sequence that is sorted with respect to the last word in the students' names:

```
for name in sorted(data, sort_names):
    print '%s: %s' % (name, average_grade(data, name))
```

6.8 Exercises

Exercise 6.1. *Read a two-column data file.*

The file `src/files/xy.dat` contains two columns of numbers, corresponding to x and y coordinates on a curve. The start of the file looks as this:

```
-1.0000      -0.0000
-0.9933      -0.0087
-0.9867      -0.0179
-0.9800      -0.0274
-0.9733      -0.0374
```

Make a program that reads the first column into a list `x` and the second column into a list `y`. Then convert the lists to arrays, and plot the curve. Print out the maximum and minimum y coordinates. (Hint: Read the file line by line, split each line into words, convert to `float`, and append to `x` and `y`.) Name of program file: `read_2columns.py` ◇

Exercise 6.2. *Read a data file.*

The files `density_water.dat` and `density_air.dat` files in the folder `src/files` contain data about the density of water and air (resp.) for different temperatures. The data files have some comment lines starting with `#` and some lines are blank. The rest of the lines contain density data: the temperature in the first column and the corresponding density in the second column. The goal of this exercise is to read the data in such a file and plot the density versus the temperature as distinct (small) circles for each data point. Let the program take the name of the data file as command-line argument. Apply the program to both files. Name of program file: `read_density_data.py` ◇

Exercise 6.3. *Simplify the implementation of Exer. 6.1.*

Look up the documentation of the `numpy.loadtxt` function and use it to load the file data in Exercise 6.1 into arrays. Name of program file: `read_2columns_loadtxt.py`. ◇

Exercise 6.4. *Fit a polynomial to data.*

The purpose of this exercise is to find a simple mathematical formula for the how the density of water or air depends on the temperature. First, load the density data from file as explained in Exercises 6.2 or 6.3. Then we want to experiment with NumPy utilities that can find a polynomial that approximate the density curve.

NumPy has a function `polyfit(x, y, deg)` for finding a “best fit” of a polynomial of degree `deg` to a set of data points given by the array arguments `x` and `y`. The `polyfit` function returns a list of the coefficients in the fitted polynomial, where the first element is the coefficient for the term with the highest degree, and the last element corresponds to the constant term. For example, given points in `x` and `y`, `polyfit(x, y, 1)` returns the coefficients `a, b` in a polynomial `a*x + b` that fits the data in the best way[17].

NumPy also has a utility `poly1d` which can take the tuple or list of coefficients calculated by, e.g., `polyfit` and return the polynomial as a Python function that can be evaluated. The following code snippet demonstrates the use of `polyfit` and `poly1d`:

[17] More precisely, a line $y = ax+b$ is a “best fit” to the data points (x_i, y_i), $i = 0, \ldots, n-1$ if a and b are chosen to make the sum of squared errors $R = \sum_{j=0}^{n-1}(y_j - (ax_j + b))^2$ as small as possible. This approach is known as *least squares approximation* to data and proves to be extremely useful throughout science and technology.

```
coeff = polyfit(x, y, deg)
p = poly1d(coeff)
print p  # prints the polynomial expression
y_fitted = p(x)
plot(x, y, 'r-', x, y_fitted, 'b-',
     legend=('data', 'fitted polynomial of degree %d' % deg'))
```

For the density–temperature relationship we want to plot the data from file and two polynomial approximations, corresponding to a 1st and 2nd degree polynomial. From a visual inspection of the plot, suggest simple mathematical formulas that relate the density of air to temperature and the density of water to temperature. Make three separate plots of the Name of program file: `fit_density_data.py` ◇

Exercise 6.5. *Read acceleration data and find velocities.*

A file `src/files/acc.dat` contains measurements $a_0, a_1, \ldots, a_{n-1}$ of the acceleration of an object moving along a straight line. The measurement a_k is taken at time point $t_k = k\Delta t$, where Δt is the time spacing between the measurements. The purpose of the exercise is to load the acceleration data into a program and compute the velocity $v(t)$ of the object at some time t.

In general, the acceleration $a(t)$ is related to the velocity $v(t)$ through $v'(t) = a(t)$. This means that

$$v(t) = v(0) + \int_0^t a(\tau)d\tau\,. \tag{6.1}$$

If $a(t)$ is only known at some discrete, equally spaced points in time, $a_0, \ldots, a_{n-1}$ (which is the case in this exercise), we must compute the integral (6.1) in numerically, for example by the Trapezoidal rule:

$$v(t_k) \approx \Delta t\left(\frac{1}{2}a_0 + \frac{1}{2}a_k + \sum_{i=1}^{k-1} a_i\right), \quad 1 \leq k \leq n-1\,. \tag{6.2}$$

We assume $v(0) = 0$ so that also $v_0 = 0$.

Read the values $a_0, \ldots, a_{n-1}$ from file into an array, plot the acceleration versus time, and use (6.2) to compute one $v(t_k)$ value, where Δt and $k \geq 1$ are specified on the command line. Name of program file: `acc2vel_v1.py`. ◇

Exercise 6.6. *Read acceleration data and plot velocities.*

The task in this exercise is the same as in Exercise 6.5, except that we now want to compute $v(t_k)$ for all time points $t_k = k\Delta t$ and plot the velocity versus time. Repeated use of (6.2) for all k values is very inefficient. A more efficient formula arises if we add the area of a new trapezoid to the previous integral (see also Appendix A.1.7):

$$v(t_k) = v(t_{k-1}) + \int\limits_{t_{k-1}}^{t_k} a(\tau)d\tau \approx v(t_{k-1}) + \Delta t\frac{1}{2}(a_{k-1} + a_k), \qquad (6.3)$$

for $k = 1, 2, \ldots, n-1$, while $v_0 = 0$. Use this formula to fill an array `v` with velocity values. Now only Δt is given on the command line, and the $a_0, \ldots, a_{n-1}$ values must be read from file as in Exercise 6.5. Name of program file: `acc2vel.py`. ◇

Exercise 6.7. *Find velocity from GPS coordinates.*

Imagine that a GPS device measures your position at every s seconds. The positions are stored as (x, y) coordinates in a file `src/files/pos.dat` with the an x and y number on each line, except for the first line which contains the value of s.

First, load s into a `float` variable and the x and y numbers into two arrays and draw a straight line between the points (i.e., plot the y coordinates versus the x coordinates).

The next task is to compute and plot the velocity of the movements. If $x(t)$ and $y(t)$ are the coordinates of the positions as a function of time, we have that the velocity in x direction is $v_x(t) = dx/dt$, and the velocity in y direction is $v_y = dy/dt$. Since x and y are only known for some discrete times, $t_k = ks$, $k = 0, \ldots, n-1$, we must use numerical differentiation. A simple (forward) formula is

$$v_x(t_k) \approx \frac{x(t_{k+1}) - x(t_k)}{s}, \quad v_y(t_k) \approx \frac{y(t_{k+1}) - y(t_k)}{s}, \quad k = 0, \ldots, n-2\,.$$

Compute arrays `vx` and `vy` with velocities based on the formulas above for $v_x(t_k)$ and $v_y(t_k)$, $k = 0, \ldots, n-2$. Plot `vx` versus time and `vy` versus time. Name of program file: `position2velocity.py`. ◇

Exercise 6.8. *Make a dictionary from a table.*

The file `src/files/constants.txt` contains a table of the values and the dimensions of some fundamental constants from physics. We want to load this table into a dictionary `constants`, where the keys are the names of the constants. For example, `constants['gravitational constant']` holds the value of the gravitational constant ($6.67259 \cdot 10^{-11}$) in Newton's law of gravitation. Make a function that that reads and interprets the text in the file, and thereafter returns the dictionary. Name of program file: `fundamental_constants.py`. ◇

Exercise 6.9. *Explore syntax differences: lists vs. dicts.*

Consider this code:

```
t1 = {}
t1[0] = -5
t1[1] = 10.5
```

Explain why the lines above work fine while the ones below do not:

```
t2 = []
t2[0] = -5
t2[1] = 10.5
```

What must be done in the last code snippet to make it work properly? Name of program file: `list_vs_dict.py`. ◇

Exercise 6.10. *Improve the program from Ch. 6.2.5.*
Consider the program `density.py` from Chapter 6.2.5. One problem we face when implementing this program is that the name of the substance can contain one or two words, and maybe more words in a more comprehensive table. The purpose of this exercise is to use string operations to shorten the code and make it more general. Implement the following two methods in separate functions in the same program, and control that they give the same result.

1. Let `substance` consist of all the words but the last, using the `join` method in string objects to combine the words.
2. Observe that all the densities start in the same column file and use substrings to divide `line` into two parts. (Hint: Remember to strip the first part such that, e.g., the density of ice is obtained as `densities['ice']` and not `densities['ice          ']`.)

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

Exercise 6.11. *Interpret output from a program.*
The program `src/funcif/lnsum.py` produces, among other things, this output:

```
epsilon: 1e-04, exact error: 8.18e-04, n=55
epsilon: 1e-06, exact error: 9.02e-06, n=97
epsilon: 1e-08, exact error: 8.70e-08, n=142
epsilon: 1e-10, exact error: 9.20e-10, n=187
epsilon: 1e-12, exact error: 9.31e-12, n=233
```

Redirect the output to a file (by `python lnsum.py > file`). Write a Python program that reads the file and extracts the numbers corresponding to `epsilon`, `exact error`, and `n`. Store the numbers in three arrays and plot `epsilon` and the `exact error` versus `n`. Use a logarithmic scale on the y axis (enabled by the `log='y'` keyword argument to the `plot` function). Name of program file: `read_error.py`. ◇

Exercise 6.12. *Make a dictionary.*
Based on the stars data in Exercise 3.32, make a dictionary where the keys contain the names of the stars and the values correspond to the luminosity. Name of program file: `stars_data_dict1.py`. ◇

Exercise 6.13. *Make a nested dictionary.*
Store the data about stars from Exercise 3.32 in a nested dictionary such that we can look up the distance, the apparent brightness, and the luminosity of a star with name `N` by

```
stars[N]['distance']
stars[N]['apparent brightness']
stars[N]['luminosity']
```

Name of program file: `stars_data_dict2.py`. ◇

Exercise 6.14. *Make a nested dictionary from a file.*

The file `src/files/human_evolution.txt` holds information about various human species and their hight, weight, and brain volume. Make a program that reads this file and stores the tabular data in a nested dictionary `humans`. The keys in `humans` correspond to the specie name (e.g., "homo erectus"), and the values are dictionaries with keys for "height", "weight", "brain volume", and "when" (the latter for when the specie lived). For example, `humans['homo neanderthalensis']['mass']` should equal `'55-70'`. Let the program write out the `humans` dictionary in a nice tabular form similar to that in the file. Name of program file: `humans.py`. ◇

Exercise 6.15. *Compute the area of a triangle.*

The purpose of this exercise is to write an `area` function as in Exercise 3.9, but now we assume that the vertices of the triangle is stored in a dictionary and not a list. The keys in the dictionary correspond to the vertex number (1, 2, or 3) while the values are 2-tuples with the x and y coordinates of the vertex. For example, in a triangle with vertices $(0,0)$, $(1,0)$, and $(0,2)$ the `vertices` argument becomes

```
{1: (0,0), 2: (1,0), 3: (0,2)}
```

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

Exercise 6.16. *Compare data structures for polynomials.*

Write a code snippet that uses both a list and a dictionary to represent the polynomial $-\frac{1}{2} + 2x^{100}$. Print the list and the dictionary, and use them to evaluate the polynomial for $x = 1.05$ (you can apply the `poly1` and `poly2` functions from Chapter 6.2.3). Name of program file: `poly_repr.py`. ◇

Exercise 6.17. *Compute the derivative of a polynomial.*

A polynomial can be represented by a dictionary as explained in Chapter 6.2.3. Write a function `diff` for differentiating such a polynomial. The `diff` function takes the polynomial as a dictionary argument and returns the dictionary representation of the derivative. Recall the formula for differentiation of polynomials:

$$\frac{d}{dx}\sum_{j=0}^{n} c_j x^j = \sum_{j=1}^{n} j c_j x^{j-1}\,. \tag{6.4}$$

This means that the coefficient of the x^{j-1} term in the derivative equals j times the coefficient of x^j term of the original polynomial. With `p` as the polynomial dictionary and `dp` as the dictionary representing the derivative, we then have `dp[j-1] = j*p[j]` for `j` running over all keys in `p`, except when `j` equals 0.

Here is an example of the use of the function `diff`:

```
>>> p = {0: -3, 3: 2, 5: -1}    # -3 + 2*x**3 - x**5
>>> diff(p)                     # should be 6*x**2 - 5*x**4
{2: 6, 4: -5}
```

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

Exercise 6.18. *Generalize the program from Ch. 6.2.7.*

The program from Chapter 6.2.7 is specialized for three particular companies. Suppose you download n files from *finance.yahoo.com*, all with monthly stock price data for the *same* period of time. Also suppose you name these files `company.csv`, where `company` reflects the name of the company. Modify the program from Chapter 6.2.7 such that it reads a set of filenames from the command line and creates a plot that compares the evolution of the corresponding stock prices. Normalize all prices such that they initially start at a unit value. Name of program file: `stockprices3.py`. ◇

Exercise 6.19. *Write function data to file.*

We want to dump x and $f(x)$ values to a file, where the x values appear in the first column and the $f(x)$ values appear in the second. Choose n equally spaced x values in the interval $[a, b]$. Provide f, a, b, n, and the filename as input data on the command line. Use the `StringFunction` tool (see Chapters 4.1.4 and 5.5.1) to turn the textual expression for f into a Python function. (Note that the program from Exercise 6.1 can be used to read the file generated in the present exercise into arrays again for visualization of the curve $y = f(x)$.) Name of program files `write_cml_function.py`. ◇

Exercise 6.20. *Specify functions on the command line.*

Explain what the following two code snippets do and give an example of how they can be used. Snippet 1:

```
import sys
from scitools.StringFunction import StringFunction
parameters = {}
for prm in sys.argv[4:]:
    key, value = prm.split('=')
    parameters[key] = eval(value)
f = StringFunction(sys.argv[1], independent_variables=sys.argv[2],
                   **parameters)
var = float(sys.argv[3])
print f(var)
```

Snippet 2:

```
import sys
from scitools.StringFunction import StringFunction
f = eval('StringFunction(sys.argv[1], ' + \
         'independent_variables=sys.argv[2], %s)' % \
         (', '.join(sys.argv[4:])))
var = float(sys.argv[3])
print f(var)
```

Hint: Read about the `StringFunction` tool in Chapter 4.1.4 and about a variable number of keyword arguments in Appendix H.5. Name of program file: `cml_functions.py`. ◇

Exercise 6.21. *Interpret function specifications.*

To specify arbitrary functions $f(x_1, x_2, \ldots; p_1, p_2, \ldots)$ with independent variables $x_1, x_2, \ldots$ and a set of parameters $p_1, p_2, \ldots$, we allow the following syntax on the command line or in a file:

```
<expression> is function of <list1> with parameter <list2>
```

where `<expression>` denotes the function formula, `<list1>` is a comma-separated list of the independent variables, and `<list2>` is a comma-separated list of name=value parameters. The part `with parameters <list2>` is omitted if there are no parameters. The names of the independent variables and the parameters can be chosen freely as long as the names can be used as Python variables. Here are four different examples of what we can specify on the command line using this syntax:

```
sin(x) is a function of x
sin(a*y) is a function of y with parameter a=2
sin(a*x-phi) is a function of x with parameter a=3, phi=-pi
exp(-a*x)*cos(w*t) is a function of t with parameter a=1,w=pi,x=2
```

Create a Python function that takes such function specifications as input and returns an appropriate `StringFunction` object. This object must be created from the function expression and the list of independent variables and parameters. For example, the last function specification above leads to the following `StringFunction` creation:

```
f = StringFunction('exp(-a*x)*sin(k*x-w*t)',
                   independent_variables=['t'],
                   a=1, w=pi, x=2)
```

Hint: Use string operations to extract the various parts of the string. For example, the expression can be split out by calling `split('is a function')`. Typically, you need to extract `<expression>`, `<list1>`, and `<list2>`, and create a string like

```
StringFunction(<expression>, independent_variables=[<list1>],
               <list2>)
```

and sending it to `eval` to create the object. Name of program file: `text2func.py`. ◇

Exercise 6.22. *Compare average temperatures in cities.*

The tarfile `src/misc/city_temp.tar.gz` contains a set of files with temperature data for a large number of cities around the world. The files are in text format with four columns, containing the month number, the date, the year, and the temperature, respectively. Missing temperature observations are represented by the value -99. The mapping between the names of the text files and the names of the cities are defined in an HTML file `citylistWorld.htm`.

First, write a function that can read the `citylistWorld.htm` file and create a dictionary with mapping between city and filenames. Second, write a function that takes this dictionary and a city name as input, opens the corresponding text file, and loads the data into an appropriate data structure (dictionary of arrays and city name is a suggestion). Third, write a function that can take a number of such data structures and the corresponding city names to create a plot of the temperatures over a certain time period.

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

Exercise 6.23. *Try Word or OpenOffice to write a program.*

The purpose of this exercise is to tell you how hard it may be to write Python programs in the standard programs that most people use for writing text.

Type the following one-line program in either Microsoft Word or OpenOffice:

```
print "Hello, World!"
```

Both Word and OpenOffice are so "smart" that they automatically edit "print" to "Print" since a sentence should always start with a capital. This is just an example that word processors are made for writing documents, not computer programs.

Save the program as a `.doc` (Word) or `.odt` (OpenOffice) file. Now try to run this file as a Python program. You will get a message

```
SyntaxError: Non-ASCII character
```

Explain why you get this error.

Then save the program as a `.txt` file. Run this file as a Python program. It may work well if you wrote the program text in Microsoft Word, but with OpenOffice there may still be strange characters in the file. Use a text editor to view the exact contents of the file. Name of program file: `office.py`. ◇

Exercise 6.24. *Evaluate objects in a boolean context.*

Writing `if a:` or `while a:` in a program, where `a` is some object, requires evaluation of `a` in a boolean context. To see the value of an object `a` in a boolean context, one can call `bool(a)`. Try the following program to learn what values of what objects that are `True` or `False` in a boolean context:

```
objects = [
  '""',                 # empty string
  '"string"',           # non-empty string
  '[]',                 # empty list
  '[0]',                # list with one element
  '()',                 # empty tuple
  '(0,)',               # tuple with one element
  '{}',                 # empty dict
  '{0:0}',              # dict with one element
  '0',                  # int zero
  '0.0',                # float zero
  '0j',                 # complex zero
  '10',                 # int 10
  '10.',                # float 10
  '10j'                 # imaginary 10
  'zeros(0)',           # empty array
  'zeros(1)',           # array with one element (zero)
  'zeros(1)+10',        # array with one element (10)
  'zeros(2)',           # array with two elements (watch out!)
  ]
for element in objects:
    object = eval(element)
    print 'object = %s; if object: is %s' % \
        (element, bool(object))
```

Write down a rule for the family of Python objects that evaluate to `False` in a boolean context. ◇

Exercise 6.25. *Fit a polynomial to experimental data.*

Suppose we have measured the oscillation period T of a simple pendulum with a mass m at the end of a massless rod of length L. We have varied L and recorded the corresponding T value. The measurements are found in a file `src/files/pendulum.dat`, containing two columns. The first column contains L values and the second column has the corresponding T values.

Load the L and T values into two arrays. Plot L versus T using circles for the data points. We shall assume that L as a function of T is a polynomial. Use the NumPy utilities `polyfit` and `poly1d`, as explained in Exercise 6.4, and experiment with fitting polynomials of degree 1, 2, and 3. Visualize the polynomial curves together with the experimental data. Which polynomial fits the measured data best? Name of program file: `fit_pendulum_data.py`. ◇

Exercise 6.26. *Generate an HTML report with figures.*

The goal of this exercise is to let a program write a report in HTML format containing the solution to Exercise 5.22 on page 249. First, include the program from that Exercises, with additional explaining text if necessary. Program code can be placed inside `<pre>` and `</pre>` tags. Second, insert three plots of the $f(x,t)$ function for three different t values (find suitable t values that illustrate the displacement of the wave packet). Third, add an animated GIF file with the movie of $f(x,t)$. Insert headlines (`<h1>` tags) wherever appropriate. Name of program file: `wavepacket_report.py`. ◇

Exercise 6.27. *Extract information from a weather page.*

Find the Yahoo! page with the weather forecast for your favorite city. Study the HTML source and write a program that downloads the HTML page, extracts forecast information such as weather type, temperature, etc. Write out this information in a compact style on the screen. This exercise is a starter for the more useful Exercise 6.28 for comparing alternative forecasts in a compact fashion. Name of program file: `Yahoo_weather.py`. ◇

Exercise 6.28. *Compare alternative weather forecasts.*

For each of a collection of weather forecast sites, say

```
http://weather.yahoo.com
http://www.weather.com
http://www.weatherchannel.com
http://weather.cnn.com
http://yr.no
```

find the pages corresponding to your favorite location. Study the HTML sources and write a function for each HTML page that downloads the web page and extracts basic forecast information: date, weather type (name of symbol), and temperature. Write out a comparison of different forecasts on the screen. Name of program file: `weather_forecast_comparison1.py`. ◇

Exercise 6.29. *Improve the output in Exercise 6.28.*

Pure text on the screen was suggested as output in Exercise 6.28. A useful alternative is to construct an HTML page with compact forecast information, where also weather symbols (images) are displayed. Extend each of the functions in Exercise 6.28 to also extract the filename containing the weather symbol(s) and write the code for presenting the comparison in HTML. Name of program file: `weather_forecast_comparison2.py`. ◇

Exercise 6.30. *Allow different types for a function argument.*

Consider the family of `find_consensus_v*` functions from Chapter 6.6.2. The different versions work on different representations of the frequency matrix. Make a unified `find_consensus` function which accepts different data structures for the `frequency_matrix`. Test on the type of data structure and perform the necessary actions. Name of program file: `find_consensus.py`. ◇

Exercise 6.31. *Make a function more robust.*

Consider the function `get_base_counts(dna)` (from Chapter 6.6.3), which counts how many times `A`, `C`, `G`, and `T` appears in the string `dna`:

```
def get_base_counts(dna):
    counts = {'A': 0, 'T': 0, 'G': 0, 'C': 0}
    for base in dna:
        counts[base] += 1
    return counts
```

Unfortunately, this function crashes if other letters appear in `dna`. Write an enhanced function `get_base_counts2` which solves this problem. Test it on a string like `'ADLSTTLLD'`. Name of program file: `get_base_counts2.py`. ◇

Exercise 6.32. *Find proportion of bases inside/outside exons.*

Consider the lactase gene as described in Chapters 6.6.4 and 6.6.5. What is the proportion of base A inside and outside exons of the lactase gene? Write a function `get_exons`, which returns all the substrings of the exon regions concatenated, and a function `get_introns`, which returns all the substrings between the exon regions concatenated. The function `get_base_frequencies` from Chapter 6.6.3 can then be used to analyze the frequencies of bases A, C, G, and T in the two strings. Name of program file: `prop_A_exons.py`. ◇

The complete code of class `IntervalMath` is found in `IntervalMath.py`. Compared to the implementations shown above, the real implementation in the file employs some ingenious constructions and help methods to save typing and repeating code in the special methods for arithmetic operations.

7.8 Exercises

Exercise 7.1. *Make a function class.*

Make a class `F` that implements the function

$$f(x; a, w) = e^{-ax} \sin(wx)\,.$$

A `value(x)` method computes values of f, while a and w are class attributes. Test the class with the following main program:

```
from math import *
f = F(a=1.0, w=0.1)
print f.value(x=pi)
f.a = 2
print f.value(pi)
```

Name of program file: `F.py`. ◇

Exercise 7.2. *Extend the class from Ch. 7.2.1.*

Add an attribute `transactions` to the `Account` class from Chapter 7.2.1. The new attribute counts the number of transactions done in the `deposit` and `withdraw` methods. The total number of transactions should be printed in the `dump` method. Write a simple test program to demonstrate that `transaction` gets the right value after some calls to `deposit` and `withdraw`. Name of program file: `Account2.py`. ◇

Exercise 7.3. *Make classes for a rectangle and a triangle.*

The purpose of this exercise is to create classes like class `Circle` from Chapter 7.2.3 for representing other geometric figures: a rectangle with width W, height H, and lower left corner (x_0, y_0); and a general triangle specified by its three vertices (x_0, y_0), (x_1, y_1), and (x_2, y_2) as explained in Exercise 3.9. Provide three methods: `__init__` (to initialize the geometric data), `area`, and `circumference`. Name of program file: `geometric_shapes.py`. ◇

Exercise 7.4. *Make a class for straight lines.*

Make a class `Line` whose constructor takes two points `p1` and `p2` (2-tuples or 2-lists) as input. The line goes through these two points (see function `line` in Chapter 3.1.7 for the relevant formula of the line). A `value(x)` method computes a value on the line at the point `x`. Here is a demo in an interactive session:

```
>>> from Line import Line
>>> line = Line((0,-1), (2,4))
>>> print line.value(0.5), line.value(0), line.value(1)
0.25 -1.0 1.5
```

Name of program file: `Line.py`. ◇

Exercise 7.5. *Improve the constructor in Exer. 7.4.*

The constructor in class `Line` in Exercise 7.4 takes two points as arguments. Now we want to have more flexibility in the way we specify a straight line: we can give two points, a point and a slope, or a slope and the line's interception with the y axis. Hint: Let the constructor take two arguments `p1` and `p2` as before, and test with `isinstance` whether the arguments are `float` or `tuple`/`list` to determine what kind of data the user supplies:

```
if isinstance(p1, (tuple,list)) and isinstance(p2, (float,int)):
    # p1 is a point and p2 is slope
    self.a = p2
    self.b = p1[1] - p2*p1[0]
elif ...
```

Name of program file: `Line2.py`. ◇

Exercise 7.6. *Make a class for quadratic functions.*

Consider a quadratic function $f(x; a, b, c) = ax^2 + bx + c$. Make a class `Quadratic` for representing f, where a, b, and c are attributes, and the methods are

1. `value` for computing a value of f at a point x,
2. `table` for writing out a table of x and f values for n x values in the interval $[L, R]$,
3. `roots` for computing the two roots.

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

Exercise 7.7. *Make a class for linear springs.*

To elongate a spring a distance x, one needs to pull the spring with a force kx. The parameter k is known as the spring constant. The corresponding potential energy in the spring is $\frac{1}{2}kx^2$.

Make a class for springs. Let the constructor store k as a class attribute, and implement the methods `force(x)` and `energy(x)` for evaluating the force and the potential energy, respectively.

The following function prints a table of function values for an arbitrary mathematical function `f(x)`. Demonstrate that you can send the `force` and `energy` methods as the `f` argument to `table`.

```
def table(f, a, b, n, heading=''):
    """Write out f(x) for x in [a,b] with steps h=(b-a)/n."""
    print heading
    h = (b-a)/float(n)
    for i in range(n+1):
        x = a + i*h
        print 'function value = %10.4f at x = %g' % (f(x), x)
```

Name of program file: Spring.py. ◇

Exercise 7.8. *Wrap functions in a class.*

The purpose of this exercise is to offer the code from Exercises 5.14–5.16 as a class. We want to construct a class Lagrange which works like this:

```
import numpy as np
# Compute some interpolation points along y=sin(x)
xp = np.linspace(0, pi, 5)
yp = np.sin(xp)

# Lagrange's interpolation polynomial
p_L = LagrangeInterpolation(xp, yp)
x = 1.2
print 'p_L(%g)=g'  (x, p_L(x)),
print 'sin(%g)=g'  (x, sin(x))
p_L.plot()
```

The plot method visualizes $p_L(x)$ for x between the first and last interpolation point (xp[0] and xp[-1]). The class LagrangeInterpolation is easy to make if you import the Lagrange_poly2 module from Exercise 5.15 and make appropriate calls to the p_L and graph functions in this module. Also include a verify function outside the class and call it as described in Exercise 5.14, but now using the class instead of the p_L function directly. Use the class to repeat the investigations that show how the interpolating polynomial may oscillate as the number of interpolation points increases. Name of program file: Lagrange_poly3.py. ◇

Exercise 7.9. *Extend the constructor in Exer. 7.8.*

Instead of manually computing the interpolation points, we now want the possibility to pass on some Python function f(x) to the constructor in the class from Exercise 7.9. Typically, we would like to write this code:

```
from numpy import exp, sin, pi

def myfunction(x):
    return exp(-x/2.0)*sin(x)

p_L = LagrangeInterpolation(myfunction, x=[0, pi], n=11)
```

In this particular example, $n = 11$ uniformly distributed x points between 0 and π are computed, and the corresponding y values are obtained by calling myfunction. The previous types of calls, LangrangeInterpolation(xp, yp), must still be valid.

The constructor in class LagrangeInterpolation must now accept two different set of arguments (xp, yp vs. f, x, n). You can use the isinstance(a, t) function to test if object a is of type t. Declare the constructor with three arguments arg1, arg2, and arg3=None. Test if arg1 and arg2 are arrays (isinstance(arg1, numpy.ndarray)), and in that case, set `xp=arg1` and `yp=arg2`. On the other hand, if `arg1` is a function (`callable(arg1)` is `True`), `arg2` is a list or tuple (`isinstance(arg2, (list,tuple))`), and `arg3` is an integer, set `f=arg1`, `x=arg2`, and `n=arg3`. Name of program file: `Lagrange_poly4.py`. ◇

Exercise 7.10. *Deduce a class implementation.*

Write a class `Hello` that behaves as illustrated in the following session:

```
>>> a = Hello()
>>> print a('students')
Hello, students!
>>> print a
Hello, World!
```

Name of program file: `Hello.py`. ◇

Exercise 7.11. *Use special methods in Exer. 7.1.*

Modify the class from Exercise 7.1 such that the following code works:

```
f = F2(1.0, 0.1)
print f(pi)
f.a = 2
print f(pi)
print f
```

Name of program file: `F2.py`. ◇

Exercise 7.12. *Extend the class from Ch. 7.2.1.*

As alternatives to the `deposit` and `withdraw` methods in class `Account` class from Chapter 7.2.1, we could use `+=` for `deposit` and `-=` for `withdraw`. The special methods `__iadd__` and `__isub__` implement the `+=` and `-=` operators, respectively. For instance, `a -= p` implies a call to `a.__isub__(p)`. One important feature of `__iadd__` and `__isub__` is that they must return `self` to work properly, cf. the documentation of these methods in the Python Language Reference (not to be confused with the Python Library Reference).

Implement the `+=` and `-=` operators, a `__str__` method, and preferably a `__repr__` method. Provide, as always, some code to test that the new methods work as intended. Name of program file: `Account3.py`. ◇

Exercise 7.13. *Implement a class for numerical differentiation.*

A widely used formula for numerical differentiation of a function $f(x)$ takes the form

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}\,. \tag{7.7}$$

This formula usually gives more accurate derivatives than (7.1) because it applies a centered, rather than a one-sided, difference.

The goal of this exercise is to use the formula (7.7) to automatically differentiate a mathematical function $f(x)$ implemented as a Python function `f(x)`. More precisely, the following code should work:

```
def f(x):
    return 0.25*x**4

df = Central(f)  # make function-like object df
# df(x) computes the derivative of f(x) approximately:
for x in (1, 5, 10):
    df_value = df(x)  # approx value of derivative of f at point x
    exact = x**3      # exact value of derivative
    print "f'(%d)=%g  (error=%.2E)" % (x, df_value, exact-df_value)
```

Implement class `Central` and test that the code above works. Include an optional argument `h` to the constructor in class `Central` so that one can specify the value of h in the approximation (7.7). Apply class `Central` to produce a table of the derivatives and the associated approximation errors for $f(x) = \ln x$, $x = 10$, and $h = 0.5, 0.1, 10^{-3}, 10^{-5}, 10^{-7}, 10^{-9}, 10^{-11}$. Collect class `Central` and the two applications of the class in the same file, but organize the file as a module so that class `Central` can be imported in other files. Name of program file: `Central.py`. ◇

Exercise 7.14. *Verify a program.*

Consider this program file for computing a backward difference approximation to the derivative of a function `f(x)`:

```
from math import *

class Backward:
    def __init__(self, f, h=e-9):
        self.f, self.h = f, h
    def __call__(self, x):
        h, f = self.h, self.f
        return (f(x) - f(x-h))/h  # finite difference

dsin = Backward(sin)
e = dsin(0) - cos(0); print 'error:', e
dexp = Backward(exp, h=e-7)
e = dexp(0) - exp(0); print 'error:', e
```

The output becomes

```
error: -1.00023355634
error: 371.570909212
```

Is the approximation that bad, or are there bugs in the program? ◇

Exercise 7.15. *Test methods for numerical differentiation.*

Make a function `table(f, x, hlist, dfdx=None)` for writing out a nicely formatted table of the errors in the numerical derivative of a function `f(x)` at point `x` using the two formulas (7.1) and 7.7 and their implementations in classes `Derivative` (from Chapter 7.3.2), and `Central` (from Exercise 7.13). The first column in the table shows a list of h values (`hlist`), while the two next columns contain the corresponding errors arising from the two numerical approximations of the first derivative. The `dfdx` argument may hold a Python function that returns the exact derivative. Write out an additional column with the exact derivative if `dfdx` is given (i.e., not `None`).

Call `table` for each of the functions x^2, $\sin^6(\pi x)$, and $\tanh(10x)$, and the x values 0 and 0.25. Can you see from the errors in the tables which of the three approximations that seems to have the overall best performance in these examples? Plot the three functions on $[-1, 1]$ and try to understand the behavior of the various approximations from the plots. Name of program file: `Derivative_comparisons.py`. ◇

Exercise 7.16. *Modify a class for numerical differentiation.*

Make the two attributes `h` and `f` of class `Derivative` from Chapter 7.3.2 protected as explained in Chapter 7.2.1. That is, prefix `h` and `f` with an underscore to tell users that these attributes should not be accessed directly. Add two methods `get_precision()` and `set_precision(h)` for reading and changing `h`. Apply the modified class to make a table of the approximation error of the derivative of $f(x) = \ln x$ for $x = 1$ and $h = 2^{-k}$, $k = 1, 5, 9, 13 \ldots, 45$. Name of program file: `Derivative_protected.py`. ◇

Exercise 7.17. *Make a class for summation of series.*

Our task in this exercise is to calculate a sum $S(x) = \sum_{k=M}^{N} f_k(x)$, where $f_k(x)$ is a term in a sequence which is assumed to decrease in absolute value. In class `Sum`, for computing $S(x)$, the constructor requires the following three arguments: $f_k(x)$ as a function `f(k, x)`, M as an `int` object `M`, and N as an `int` object `N`. A `__call__` method computes and returns $S(x)$. The next term in the series, $f_{N+1}(x)$, should be computed and stored as an attribute `first_neglected_term`. Here is an example where we compute $S(x) = \sum_{k=0}^{N} (-x)^k$:

```
def term(k, x):  return (-x)**k

S = Sum(term, M=0, N=100)
x = 0.5
print S(x)
# Print the value of the first neglected term from last S(x) comp.
print S.first_neglected_term
```

Calculate by hand what the output of this test becomes, and use it to verify your implementation of class `Sum`.

Apply class `Sum` to compute the Taylor polynomial approximation for $\sin x$ at $x = \pi, 30\pi$ and $N = 5, 10, 20$. Compute the error and compare with the first neglected term $f_{N+1}(x)$. Present the result in nicely formatted tables. Repeat such calculations for the Taylor polynomial for e^{-x} at $x = 1, 3, 5$ and $N = 5, 10, 20$. Also demonstrate how class `Sum` can be used to calculate the sum (3.1) on page 94 (choose $x = 2, 5, 10$

and $N = 5, 10, 20$). Formulas for the Taylor polynomials can be looked up in Exercise 5.28. Name of program file: `Sum.py`. ◇

Exercise 7.18. *Apply the differentiation class from Ch. 7.3.2.*

Use class `Derivative` from page 362 to calculate the derivative of the function v on page 252 with respect to the parameter n, i.e., $\frac{dv}{dn}$. Choose $\beta/\mu_0 = 50$ and $r/R = 0.5$, and compare the result with the exact derivative. Hint: Make a class similar to `VelocityProfile` on page 350, but provide `r` as a parameter to the constructor, instead of `n`, and let `__call__` take `n` as parameter. Name of program file: `VelocityProfile_deriv.py`. ◇

Exercise 7.19. *Make a class for the Heaviside function.*

Use a class to implement the discontinuous Heaviside function (3.25) from Exercise 3.24 and the smoothed continuous version (3.26) from Exercise 3.25 such that the following code works:

```
H = Heaviside()         # original discontinous Heaviside function
print H(0.1)
H = Heaviside(eps=0.8)  # smoothed continuous Heaviside function
print H(0.1)
```

Name of program file: `Heaviside_class1.py`. ◇

Exercise 7.20. *Add vectorization to the class in Exer. 7.19.*

Ideally, class `Heaviside` from Exercise 7.19 should allow array arguments:

```
H = Heaviside()         # original discontinous Heaviside function
x = numpy.linspace(-1, 1, 11)
print H(x)
H = Heaviside(eps=0.8)  # smoothed Heaviside function
print H(x)
```

Use ideas from Chapter 5.5.2 to extend class `Heaviside` such that the code segment above works. Name of program file: `Heaviside_class2.py`. ◇

Exercise 7.21. *Equip the class in Exer. 7.19 with plotting.*

Another desired feature in class Heaviside from Exercises 7.19 and 7.20 is support for plotting the function in a given interval between `xmin` and `xmax`:

```
H = Heaviside()
x, y = H.plot(xmin=-4, xmax=4) # H(x-3)
from matplotlib.pyplot import plot
plot(x, y)

H = Heaviside(eps=1)
x, y = H.plot(xmin=-4, xmax=4)
plot(x, y)
```

Techniques from Chapter 5.4.1 must in the first case be used to return arrays `x` and `y` such that the discontinuity is exactly reproduced. In the continuous (smoothed) case, one needs to compute a sufficiently fine resolution (`x`) based on the `eps` parameter, e.g., $201/\epsilon$ points in the interval $[-\epsilon, \epsilon]$, with a coarser set of coordinates outside this interval where the smoothed Heaviside function is almost constant, 0 or 1. Extend class `Heaviside` with such a `plot` method. Name of program file: `Heaviside_class3.py`. ◇

Exercise 7.22. *Make a class for the indicator function.*

Consider the indicator function from Exercise 3.26. Make a class implementation of this this function, using the definition (3.28) in terms of Heaviside functions. Allow for an ϵ parameter in the calls to the Heaviside function, as in Exercise 7.19, such that we can easily choose between a discontinuous and a smoothed, continuous version of the indicator function:

```
I = Indicator(a, b)          # indicator function on [a,b]
print I(b+0.1), I((a+b)/2.0)
I = Indicator(0, 2, eps=1)   # smoothed indicator function on [0,2]
print I(0), I(1), I(1.9)
```

Note that if you build on the version of class `Heaviside` in Exercise 7.20, any `Indicator` instance will accept array arguments too. Name of program file: `Indicator.py`. ◇

Exercise 7.23. *Make a class for piecewise constant functions.*

The purpose of this exercise is to implement a piecewise constant function, as explained in Exercise 3.27, in a Python class. The following code demonstrates the minimum functionality of the class:

```
f = PiecewiseConstant([(0.4, 1), (0.2, 1.5), (0.1, 3)], xmax=4)
print f(1.5), f(1.75), f(4)

x = np.linspace(0, 4, 21)
print f(x)
```

Name of program file: `PiecewiseConstant1.py`. ◇

Exercise 7.24. *Extend the class in Exer. 7.23 with plot functionality.*

Add a `plot` method to class `PiecewiseConstant` from Exercise 7.23 such that we can easily plot the graph of the piecewise constant function. The `plot` method should return coordinate arrays `x` and `y` as explained in Exercise 7.21:

```
f = PiecewiseConstant([(0.4, 1), (0.2, 1.5), (0.1, 3)], xmax=4)
x, y = f.plot()
from matplotlib.pyplot import plot
plot(x, y)
```

Name of program file: `PiecewiseConstant2.py`. ◇

Exercise 7.25. *Make a module for piecewise constant functions.*

The purpose of this exercise is to pack the extended class `PiecewiseConstant` from Exercise 7.24 in a module, together with the most advanced versions of the classes Heaviside and Indicator from Exercises 7.19–7.22. Make a function _test() in the module which checks that all the functionality of the classes works as intended. Call _test() from the module's test block. Name of program file: PiecewiseConstant.py. ◇

Exercise 7.26. *Use classes for computing inverse functions.*

Appendix A.1.11 describes a method and implementation for computing the inverse function of a given function. The purpose of the present exercise is to improve the implementation in Appendix A.1.11 by introducing classes. This results in software that is more flexible with respect to the way we can specify the function to be inverted.

Implement the F and dFdx functions from Appendix A.1.11 as classes to avoid relying on global variables for h, xi, etc. Also introduce a class InverseFunction to run the complete algorithm and store the g array (from Appendix A.1.11) as an array attribute values. Here is a typical use of class InverseFunction:

```
>>> from InverseFunction import InverseFunction as I
>>> from scitools.std import *
>>> def f(x):
...     return log(x)
...
>>> x = linspace(1, 5, 101)
>>> f_inv = I(f, x)
>>> plot(x, f(x), x, f_inv.values)
```

Check, in the constructor, that f is monotonically increasing or decreasing over the set of coordinates (x). Errors may occur in the computations because Newton's method might divide by zero or diverge. Make sure sensible error messages are reported in those cases.

A __call__ method in class InverseFunction should evaluate the inverse function at an arbitrary point x. This is somewhat challenging since we only have the inverse function at discrete points along its curve. With aid of a function wrap2callable from scitools.std one can turn (x, y) points on a curve, stored in arrays x and y, into a (piecewise) *continuous* Python function q(x) by:

```
q = wrap2callable((x, y))
```

In a sense, the wrap2callable call draws lines between the discrete points to form the resulting continuous function. Use wrap2callable to make the __call__ method evaluate the inverse function at any point in the interval from x[0] to x[-1]. Name of program file: InverseFunction.py. ◇

Exercise 7.27. *Vectorize a class for numerical integration.*

Implement a vectorized version of the Trapezoidal rule in class Integral from Chapter 7.3.3. Use sum to compute the sum in the formula, and allow for either Python's built-in sum function or for the `sum` function from `numpy`. Apply the `time` module (see Appendix H.6.1) to measure the relative efficiency of the scalar version versus the two vectorized versions. Name of program file: `Integral_vec.py`. ◇

Exercise 7.28. *Speed up repeated integral calculations.*

The observant reader may have noticed that our `Integral` class from Chapter 7.3.3 is very inefficient if we want to tabulate or plot a function $F(x) = \int_a^x f(x)$ for several consecutive values of x, say $x_0 < x_1 < \cdots < x_n$. Requesting $F(x_k)$ will recompute the integral computed as part of $F(x_{k-1})$, and this is of course waste of computer work. Use the ideas from Appendix A.1.7 to modify the `__call__` method such that if `x` is an array, assumed to contain coordinates of increasing value: $x_0 < x_1 < \cdots < x_n$, the method returns an array with $F(x_0), F(x_1), \ldots, F(x_n)$. Name of program file: `Integral_eff.py`. ◇

Exercise 7.29. *Apply a polynomial class.*

The Taylor polynomial of degree N for the exponential function e^x is given by

$$p(x) = \sum_{k=0}^{N} \frac{x^k}{k!}\,.$$

Make a program that (i) imports class `Polynomial` from page 369, (ii) reads x and a series of N values from the command line, (iii) creates a `Polynomial` instance representing the Taylor polynomial, and (iv) prints the values of $p(x)$ for the given N values as well as the exact value e^x. Try the program out with $x = 0.5, 3, 10$ and $N = 2, 5, 10, 15, 25$. Name of program file: `Polynomial_exp.py`. ◇

Exercise 7.30. *Find a bug in a class for polynomials.*

Go through this alternative implementation of class `Polynomial` from page 369 and explain each line in detail:

```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        return sum([c*x**i for i, c in enumerate(self.coeff)])

    def __add__(self, other):
        maxlength = max(len(self), len(other))
        # Extend both lists with zeros to this maxlength
        self.coeff += [0]*(maxlength - len(self.coeff))
        other.coeff += [0]*(maxlength - len(other.coeff))
        result_coeff = self.coeff
        for i in range(maxlength):
            result_coeff[i] += other.coeff[i]
        return Polynomial(result_coeff)
```

The `enumerate` function, used in the `__call__` method, enables us to iterate over a list `somelist` with both list indices and list elements: `for index, element in enumerate(somelist)`. Write the code above in a file, and demonstrate that adding two polynomials does not work. Find the bug and correct it. Name of program file: `Polynomial_error.py`. ◇

Exercise 7.31. *Implement subtraction of polynomials.*

Implement the special method `__sub__` in class `Polynomial` from page 369. Name of program file: `Polynomial_sub.py`. ◇

Exercise 7.32. *Represent a polynomial by a NumPy array.*

Introduce a Numerical Python array for `self.coeff` in class `Polynomial` from page 369. Go through the class code and run the statements in the `_test` method in the `Polynomial.py` file to locate which statements that need to be modified because `self.coeff` is an array and not a list. Name of program file: `Polynomial_array1.py`. ◇

Exercise 7.33. *Vectorize a class for polynomials.*

Introducing an array instead of a list in class `Polynomial`, as suggested in Exercise 7.32, does not enhance the implementation. A real enhancement arise when the code is vectorized, i.e., when loops are replaced by operations on whole arrays.

First, vectorize the `__add__` method by adding the common parts of the coefficients arrays and then appending the rest of the longest array to the result (appending an array `a` to an array `b` is done by `concatenate(a, b)`).

Second, vectorize the `__call__` method by observing that evaluation of a polynomial, $\sum_{i=0}^{n-1} c_i x^i$, can be computed as the inner product of two arrays: $(c_0, \ldots, c_{n-1})$ and $(x^0, x^1, \ldots, x^{n-1})$. The latter array can be computed by `x**p`, where `p` is an array with powers $0, 1, \ldots, n-1$.

Third, the `differentiate` method can be vectorized by the statements

```
n = len(self.coeff)
self.coeff[:-1] = linspace(1, n-1, n-1)*self.coeff[1:]
self.coeff = self.coeff[:-1]
```

Show by hand calculations in a case where `n` is 3 that the vectorized statements produce the same result as the original `differentiate` method.

The `__mul__` method is more challenging to vectorize so you may leave this unaltered. Check that the vectorized versions of `__add__`, `__call__`, and `differentiate` work by comparing with the scalar code from Exercise 7.32 or the original, list-based `Polynomial` class. Name of program file: `Polynomial_array2.py`. ◇

Exercise 7.34. *Use a dict to hold polynomial coefficients; add.*

Use a dictionary for `self.coeff` in class `Polynomial` from page 369. The advantage with a dictionary is that only the nonzero coefficients need to be stored. Let `self.coeff[k]` hold the coefficient of the x^k term. Implement a constructor and the `__add__` method. Exemplify the implementation by adding $x - 3x^{100}$ and $x^{20} - x + 4x^{100}$. Name of program file: `Polynomial_dict1.py`. ◇

Exercise 7.35. *Use a dict to hold polynomial coefficients; mul.*

Extend the class in Exercise 7.34 with a `__mul__` method. First, study the algorithm in Chapter 7.3.7 for the `__mul__` method when the coefficients are stored in lists. Then modify the algorithm to work with dictionaries. Implement the algorithm and exemplify it by multiplying $x - 3x^{100}$ and $x^{20} - x + 4x^{100}$. Name of program file: `Polynomial_dict2.py`. ◇

Exercise 7.36. *Extend class Vec2D to work with lists/tuples.*

The `Vec2D` class from Chapter 7.4 supports addition and subtraction, but only addition and subtraction of two `Vec2D` objects. Sometimes we would like to add or subtract a point that is represented by a list or a tuple:

```
u = Vec2D(-2, 4)
v = u + (1,1.5)
w = [-3, 2] - v
```

That is, a list or a tuple must be allowed in the right or left operand. Use ideas from Chapters 7.5.3 and 7.5.4 to implement this extension. Name of program file: `Vec2D_lists.py`. ◇

Exercise 7.37. *Extend class Vec2D to 3D vectors.*

Extend the implementation of class `Vec2D` from Chapter 7.4 to a class `Vec3D` for vectors in three-dimensional space. Add a method `cross` for computing the cross product of two 3D vectors. Name of program file: `Vec3D.py`. ◇

Exercise 7.38. *Use NumPy arrays in class Vec2D.*

The internal code in class `Vec2D` from Chapter 7.4 can be valid for vectors in any space dimension if we represent the vector as a NumPy array in the class instead of separate variables `x` and `y` for the vector components. Make a new class `Vec` where you apply NumPy functionality in the methods. The constructor should be able to treat all the following ways of initializing a vector:

```
a = array([1, -1, 4], float)  # numpy array
v = Vec(a)
v = Vec([1, -1, 4])           # list
v = Vec((1, -1, 4))           # tuple
v = Vec(1, -1)                # coordinates
```

We will provide some helpful advice. In the constructor, use variable number of arguments as described in Appendix H.5. All arguments are then available as a tuple, and if there is only one element in the tuple, it should be an array, list, or tuple you can send through `asarray` to get a NumPy array. If there are many arguments, these are coordinates, and the tuple of arguments can be transformed by `array` to a NumPy array. Assume in all operations that the involved vectors have equal dimension (typically that `other` has the same dimension as `self`). Recall to return `Vec` objects from all arithmetic operations, not NumPy arrays, because the next operation with the vector will then not take place in `Vec` but in NumPy. If `self.v` is the attribute holding the vector as a NumPy array, the addition operator will typically be implemented as

```
class Vec:
    ...
    def __add__(self, other):
        return Vec(selv.v + other.v)
```

Name of program file: `Vec.py`. ◇

Exercise 7.39. *Use classes in the program from Ch. 6.7.2.*
Modify the `files/students.py` program described in Chapter 6.7.2 by making the values of the `data` dictionary instances of class `Student`. This class contains a student's name and a list of the courses. Each course is represented by an instance of class `Course`. This class contains the course name, the semester, the credit points, and the grade. Make `__str__` and/or `__repr__` write out the contents of the objects. Name of program file: `Student_Course.py`. ◇

Exercise 7.40. *Use a class in Exer. A.25.*
The purpose of this exercise is to make the program from Exercise A.25 on page 593 more flexible by creating a class that runs and archives all the experiments. Here is a sketch of the class:

```
class GrowthLogistic:
    def __init__(self, show_plot_on_screen=False):
        self.experiments = []
        self.show_plot_on_screen = show_plot_on_screen
        self.remove_plot_files()

    def run_one(self, y0, q, N):
        """Run one experiment."""
        # Compute y[n] in a loop...
        plotfile = 'tmp_y0_%g_q_%g_N_%d.png' % (y0, q, N)
        self.experiments.append({'y0': y0, 'q': q, 'N': N,
                                 'mean': mean(y[20:]),
                                 'y': y, 'plotfile': plotfile})
        # Make plot...

    def run_many(self, y0_list, q_list, N):
        """Run many experiments."""
        for q in q_list:
            for y0 in y0_list:
                self.run_one(y0, q, N)

    def remove_plot_files(self):
        """Remove plot files with names tmp_y0*.png."""
        import os, glob
        for plotfile in glob.glob('tmp_y0*.png'):
            os.remove(plotfile)

    def report(self, filename='tmp.html'):
        """
        Generate an HTML report with plots of all
        experiments generated so far.
        """
        # Open file and write HTML header...
        for e in self.experiments:
            html.write('<p><img src="%s">\n' % e['plotfile'])
        # Write HTML footer and close file...
```

Each time the `run_one` method is called, data about the current experiment is stored in the `experiments` list. Note that `experiments` contains a list of dictionaries. When desired, we can call the `report` method to collect all the plots made so far in an HTML report. A typical use of the class goes as follows:

```
N = 50
g = GrowthLogistic()
g.run_many(y0_list=[0.01, 0.3],
           q_list=[0.1, 1, 1.5, 1.8] + [2, 2.5, 3], N=N)
g.run_one(y0=0.01, q=3, N=1000)
g.report()
```

Make a complete implementation of class `GrowthLogistic` and test it with the small program above. The program file should be constructed as a module. Name of program file: `growth_logistic5.py`. ◇

Exercise 7.41. *Apply the class from Exer. 7.40 interactively.*

Class `GrowthLogistic` from Exercise 7.40 is very well suited for interactive exploration. Here is a possible sample session for illustration:

```
>>> from growth_logistic5 import GrowthLogistic
>>> g = GrowthLogistic(show_plot_on_screen=True)
>>> q = 3
>>> g.run_one(0.01, q, 100)
>>> y = g.experiments[-1]['y']
>>> max(y)
1.3326056469620293
>>> min(y)
0.0029091569028512065
```

Extend this session with an investigation of the oscillations in the solution y_n. For this purpose, make a function for computing the local maximum values y_n and the corresponding indices where these local maximum values occur. We can say that y_i is a local maximum value if

$$y_{i-1} < y_i > y_{i+1}\,.$$

Plot the sequence of local maximum values in a new plot. If $I_0, I_1, I_2, \ldots$ constitute the set of increasing indices corresponding to the local maximum values, we can define the periods of the oscillations as $I_1 - I_0$, $I_2 - I_1$, and so forth. Plot the length of the periods in a separate plot. Repeat this investigation for $q = 2.5$. Name of program file: `GrowthLogistic_interactive.py`. ◇

Exercise 7.42. *Find local and global extrema of a function.*

Extreme points of a function $f(x)$ are normally found by solving $f'(x) = 0$. A much simpler method is to evaluate $f(x)$ for a set of discrete points in the interval $[a, b]$ and look for local minima and maxima among these points. We work with n equally spaced points $a = x_0 < x_1 < \cdots < x_{n-1} = b$, $x_i = a + ih$, $h = (b - a)/(n - 1)$.

1. First we find all local extreme points in the interior of the domain. Local minima are recognized by
$$f(x_{i-1}) > f(x_i) < f(x_{i+1}), \quad i = 1, \ldots, n-2\,.$$
Similarly, at a local maximum point x_i we have
$$f(x_{i-1}) < f(x_i) > f(x_{i+1}), \quad i = 1, \ldots, n-2\,.$$
We let $P_{\min}$ be the set of x values for local minima and $F_{\min}$ the set of the corresponding $f(x)$ values at these minimum points. Two sets $P_{\max}$ and $F_{\max}$ are defined correspondingly, containing the maximum points and their values.
2. The boundary points $x = a$ and $x = b$ are for algorithmic simplicity also defined as local extreme points: $x = a$ is a local minimum if $f(a) < f(x_1)$, and a local maximum otherwise. Similarly, $x = b$ is a local minimum if $f(b) < f(x_{n-2})$, and a local maximum otherwise. The end points a and b and the corresponding function values must be added to the sets $P_{\min}, P_{\max}, F_{\min}, F_{\max}$.
3. The global maximum point is defined as the x value corresponding to the maximum value in $F_{\max}$. The global minimum point is the x value corresponding to the minimum value in $F_{\min}$.

Make a class `MinMax` with the following functionality:

- The constructor takes $f(x)$, a, b, and n as arguments, and calls a method `_find_extrema` to compute the local and global extreme points.
- The method `_find_extrema` implements the algorithm above for finding local and global extreme points, and stores the sets $P_{\min}, P_{\max}, F_{\min}, F_{\max}$ as list attributes in the (`self`) instance.
- The method `get_global_minimum` returns the global minimum point (x).
- The method `get_global_maximum` returns the global maximum point (x).
- The method `get_all_minima` returns a list or array of all minimum points.
- The method `get_all_maxima` returns a list or array of all maximum points.
- The method `__str__` returns a string where all the min/max points are listed, plus the global extreme points.

Here is a sample code using class `MinMax`:

```
def f(x):
    return x**2*exp(-0.2*x)*sin(2*pi*x)

m = MinMax(f, 0, 4, 5001)
print m
```

The output becomes

```
All minima: 0.8056, 1.7736, 2.7632, 3.7584, 0
All maxima: 0.3616, 1.284, 2.2672, 3.2608, 4
Global minimum: 3.7584
Global maximum: 3.2608
```

Make sure that the program also works for functions without local extrema, e.g., linear functions $f(x) = px + q$. Name of program file: `minmaxf.py`. ◇

Exercise 7.43. *Improve the accuracy in Exer. 7.42.*

The algorithm in Exercise 7.42 finds local extreme points x_i, but we know is that the true extreme point is in the interval (x_{i-1}, x_{i+1}). A more accurate algorithm may take this interval as a starting point and run a Bisection method (see Chapter 4.6.2) to find the extreme point $\bar{x}$ such that $f'(\bar{x}) = 0$. In class `MinMax`, add a method `_refine_extrema`, which goes through all the interior local minima and maxima and solves $f'(\bar{x}) = 0$. Compute $f'(x)$ using the `Derivative` class (Chapter 7.3.2 with $h \ll x_{i+1} - x_{i-1}$. Name of program file: `minmaxf2.py`. ◇

Exercise 7.44. *Find the optimal production for a company.*

The company PROD produces two different products, P_1 and P_2, based on three different raw materials, M_1, M_2 and M_3. The following table shows how much of each raw material M_i that is required to produce *a single unit* of each product P_j:

	P_1	P_2
M_1	2	1
M_2	5	3
M_3	0	4

For instance, to produce one unit of P_2 one needs 1 unit of M_1, 3 units of M_2 and 4 units of M_3. Furthermore, PROD has available 100, 80 and 150 units of material M_1, M_2 and M_3 respectively (for the time period considered). The revenue per produced unit of product P_1 is 150 NOK, and for one unit of P_2 it is 175 NOK. On the other hand the raw materials M_1, M_2 and M_3 cost 10, 17 and 25 NOK per unit, respectively. The question is: How much should PROD produce of each product? We here assume that PROD wants to maximize its net revenue (which is revenue minus costs).

a) Let x and y be the number of units produced of product P_1 and P_2, respectively. Explain why the total revenue $f(x, y)$ is given by

$$f(x, y) = 150x - (10 \cdot 2 + 17 \cdot 5)x + 175y - (10 \cdot 1 + 17 \cdot 3 + 25 \cdot 4)y$$

and simplify this expression. The function $f(x, y)$ is *linear* in x and y (check that you know what linearity means).
b) Explain why PROD's problem may be stated mathematically as follows:

$$\begin{aligned} &\text{maximize} \quad f(x, y) \\ &\text{subject to} \\ &\qquad 2x + y \leq 100 \\ &\qquad 5x + 3y \leq 80 \\ &\qquad 4y \leq 150 \\ &\qquad x \geq 0, y \geq 0. \end{aligned} \tag{7.8}$$

This is an example of a *linear optimization problem.*
c) The production (x, y) may be considered as a point in the plane. Illustrate geometrically the set T of all such points that satisfy the constraints in model (7.8). Every point in this set is called a *feasible point.* (Hint: For every inequality determine first the straight line obtained by replacing the inequality by equality. Then, find the points satisfying the inequality (a half-plane), and finally, intersect these half-planes.)
d) Make a program `optimization1.py` for drawing the straight lines defined by the inequalities. Each line can be written as $ax + by = c$. Let the program read each line from the command line as a list of the a, b, and c values. In the present case the command-line arguments will be

```
'[2,1,100]' '[5,3,80]' '[0,4,150]' '[1,0,0]' '[0,1,0]'
```

(Hint: Perform an `eval` on the elements of `sys.argv[1:]` to get a, b, and c for each line as a list in the program.)
e) Let α be a positive number and consider the *level set* of the function f, defined as the set

$$L_\alpha = \{(x, y) \in T : f(x, y) = \alpha\}.$$

This set consists of all feasible points having the same net revenue α. Extend the program with two new command-line arguments holding p and q for a function $f(x, y) = px + qy$. Use this information to compute the level set lines $y = \alpha/q - px/q$, and plot the level set lines for some different values of α (use the α value in the legend for each line).
f) Use what you saw in e) to solve the problem (7.8) geometrically. (Hint: How large can you choose α such that L_α is nonempty?) This solution is called an *optimal solution.*

Name of program file: `optimization1.py`. ◇

Exercise 7.45. *Extend the program from Exer. 7.44.*

Assume that we have other values on the revenues and costs than the actual numbers in Exercise 7.44. Explain why (7.8), with these new parameter values, still has an optimal solution lying in a corner point of T. Extend the program from Exercise 7.44 to calculate all the corner points of a region T in the plane determined by the linear inequalities like those listed in Exercise 7.44. Moreover, the program shall compute the maximum of a given linear function $f(x, y) = px + qy$ over T by calculating the function values in the corner points and finding the smallest function value. Name of program file: `optimization2.py`.

The example in Exercises 7.44 and 7.45 is from *linear optimization*, also called *linear programming*. Most universities and business schools have a good course in this important area of applied mathematics. ◇

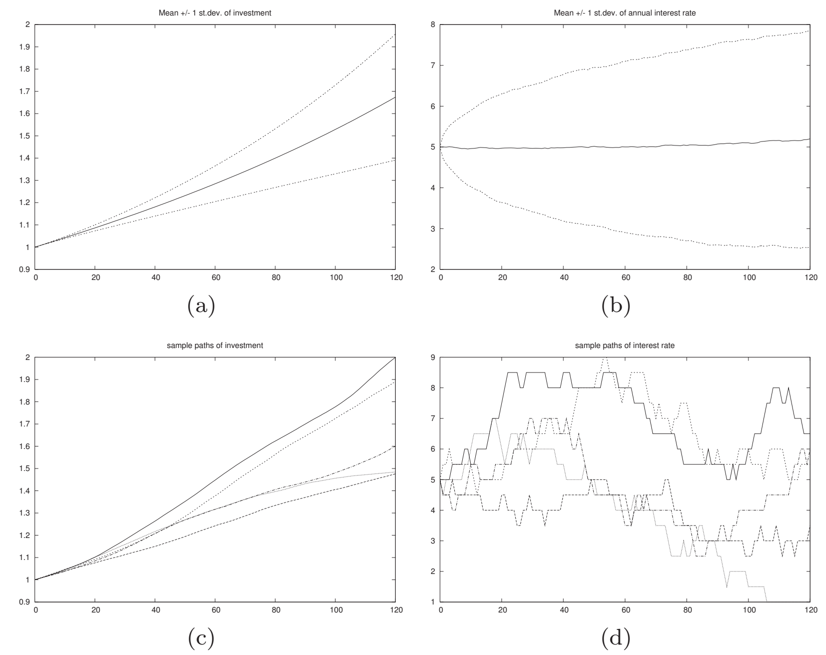

Fig. 8.8 Development of an investment with random jumps of the interest rate at random points of time: (a) mean value of investment ± one standard deviation; (b) mean value of the interest rate ± one standard deviation; (c) five paths of the investment development; (d) five paths of the interest rate development.

8.9 Exercises

Exercise 8.1. *Flip a coin N times.*

Make a program that simulates flipping a coin N times. Print out "tail" or "head" for each flip and let the program count the number of heads. (Hint: Use `r = random.random()` and define head as `r <= 0.5`, or draw an integer among $\{1, 2\}$ with `r = random.randint(1,2)` and define head when `r` is 1.) Name of program file: `flip_coin.py`. ◇

Exercise 8.2. *Compute a probability.*

What is the probability of getting a number between 0.5 and 0.6 when drawing uniformly distributed random numbers from the interval $[0, 1)$? To answer this question empirically, let a program draw N such random numbers using Python's standard `random` module, count how many of them, M, that fall in the interval $(0.5, 0.6)$, and compute the probability as M/N. Run the program with the four values $N = 10^i$ for $i = 1, 2, 3, 6$. Name of program file: `compute_prob.py`. ◇

Exercise 8.3. *Choose random colors.*

Suppose we have eight different colors. Make a program that chooses one of these colors at random and writes out the color. Hint: Use a list of color names and use the `choice` function in the `random` module to pick a list element. Name of program file: `choose_color.py`. ◇

Exercise 8.4. *Draw balls from a hat.*

Suppose there are 40 balls in a hat, of which 10 are red, 10 are blue, 10 are yellow, and 10 are purple. What is the probability of getting two blue and two purple balls when drawing 10 balls at random from the hat? Name of program file: `4balls_from10.py`. ◇

Exercise 8.5. *Computing probabilities of rolling dice.*

1. You throw a die. What is the probability of getting a 6?
2. You throw a die four times in a row. What is the probability of getting 6 all the times?
3. Suppose you have thrown the die three times with 6 coming up all times. What is the probability of getting a 6 in the fourth throw?
4. Suppose you have thrown the die 100 times and experienced a 6 in every throw. What do you think about the probability of getting a 6 in the next throw?

First try to solve the questions from a theoretical or common sense point of view. Thereafter, make functions for simulating cases 1, 2, and 3. Name of program file: `rolling_dice.py`. ◇

Exercise 8.6. *Estimate the probability in a dice game.*

Make a program for estimating the probability of getting at least one 6 when throwing n dice. Read n and the number of experiments from the command line. (To verify the program, you can compare the estimated probability with the exact result 11/36 when $n = 2$.) Name of program file: `one6_ndice.py`. ◇

Exercise 8.7. *Compute the probability of hands of cards.*

Use the `Deck.py` module (in `src/random`) and the `same_rank` and `same_suit` functions from the `cards` module to compute the following probabilities by Monte Carlo simulation:

- exactly two pairs among five cards,
- four or five cards of the same suit among five cards,
- four-of-a-kind among five cards.

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

Exercise 8.8. *Decide if a dice game is fair.*

Somebody suggests the following game. You pay 1 euro and are allowed to throw four dice. If the sum of the eyes on the dice is less than 9, you win r euros, otherwise you lose your investment. Should you play this game when $r = 10$? Answer the question by making a program that simulates the game. Read r and the number of experiments N from the command line Name of program file: `sum9_4dice.py`. ◇

Exercise 8.9. *Adjust the game in Exer. 8.8.*

It turns out that the game in Exercise 8.8 is not fair, since you lose money in the long run. The purpose of this exercise is to adjust the winning award so that the game becomes fair, i.e., that you neither lose nor win money in the long run.

Make a program that computes the probability p of getting a sum less than s when rolling n dice. Use the reasoning in Chapter 8.3.2 to find the award per game, r, that makes the game fair. Run the program from Exercise 8.8 with this r on the command line and verify that the game is now fair. Name of program file: `sum_s_ndice_fair.py`. ◇

Exercise 8.10. *Generalize the game from Chap. 8.3.2.*

Consider the game in Chapter 8.3.2. A generalization is to think as follows: you throw one die until the number of eyes is less than or equal to the previous throw. Let m be the number of throws in a game. Use Monte Carlo simulation to compute the probability of getting $m = 2, 3, 4, \ldots$. (For $m \geq 6$ the throws must be exactly $1, 2, 3, 4, 5, 6, 6, 6, \ldots$, and the probability of each is $1/6$, giving the total probability 6^{-m}.) If you pay 1 euro to play this game, what is the fair amount to get paid when win? Answer this question for each of the cases $m = 2, 3, 4, 5$. Use $N = 10^6$ experiments (this should suffice to estimate the probabilities for $m \leq 5$, and beyond that we have the analytical expression). Name of program file: `incr_eyes_m.py`. ◇

Exercise 8.11. *Compare two playing strategies.*

Suggest a player strategy for the game in Chapter 8.4.2. Remove the question in the `player_guess` function in the file `src/random/ndice2.py`, and implement the chosen strategy instead. Let the program play a large number of games, and record the number of times the computer wins. Which strategy is best in the long run: the computer's or yours? Name of program file: `simulate_strategies1.py`. ◇

Exercise 8.12. *Solve Exercise 8.11 with different no. of dice.*

Solve Exercise 8.11 for two other cases with 3 and 50 dice, respectively. Name of program file: `simulate_strategies2.py`. ◇

Exercise 8.13. *Extend Exercise 8.12.*

Extend the program from Exercise 8.12 such that the computer and the player can use a different number of dice. Let the computer choose a random number of dice between 2 and 20. Experiment to find out if there is a favorable number of dice for the player. Name of program file: `simulate_strategies3.py`. ◇

Exercise 8.14. *Investigate the winning chances of some games.*

An amusement park offers the following game. A hat contains 20 balls: 5 red, 5 yellow, 3 green, and 7 brown. At a cost of $2n$ euros you can draw $4 \leq n \leq 10$ balls at random from the hat (without putting them back). Before you are allowed to look at the drawn balls, you must choose one of the following options:

1. win 60 euros if you have drawn exactly three red balls

2. win $7 + 5\sqrt{n}$ euros if you have drawn at least three brown balls
3. win $n^3 - 26$ euros if you have drawn exactly one yellow ball and one brown ball
4. win 23 euros if you have drawn at least one ball of each color

For each of the $4n$ different types of games you can play, compute the net income (per play) and the probability of winning. Is there any of the games (i.e., any combinations of n and the options 1-4) where you will win money in the long run? Name of program file: `draw_balls.py`. ◇

Exercise 8.15. *Compute probabilities of throwing two dice.*

Make a computer program for throwing two dice a large number of times. Record the sum of the eyes each time and count how many times each of the possibilities for the sum (2, 3, ..., 12) appear. A dictionary with the sum as key and count as value is convenient here. Divide the counts by the total number of trials such that you get the frequency of each possible sum. Write out the frequencies and compare them with exact probabilities. (To find the exact probabilities, set up all the 6×6 possible outcomes of throwing two dice, and then count how many of them that has a sum s for $s = 2, 3, \ldots, 12$.) Name of program file: `freq_2dice.py`. ◇

Exercise 8.16. *Play with vectorized boolean expressions.*

Using the `numpy.random` module, make an array `r` containing N uniformly distributed random numbers between 0 and 1. Print out the arrays `r <= 0.5`, `r[r <= 0.5]`, `where(r <= 0.5, 1, 0)` and convince yourself that you understand what these arrays express. We want to compute how many of the elements in `r` that are less than or equal to 0.5. How can this be done in a vectorized way, i.e., without explicit loops in the program, but solely with operations on complete arrays? Name of program file: `bool_vec.py`. ◇

Exercise 8.17. *Vectorize the program from Exer. 8.1.*

Simulate flipping a coin N times and write out the number of tails. The code should be vectorized, i.e., there must be no loops in Python. Hint: Use ideas from Exercise 8.16. Name of program file: `flip_coin_vec.py`. ◇

Exercise 8.18. *Vectorize the code in Exer. 8.2.*

The purpose of this exercise is to speed up the code in Exercise 8.2 by vectorization. Hint: First draw an array `r` with a large number of random numbers in $[0, 1)$. The simplest way to count how many elements in `r` that lie between 0.5 and 0.6, is to first extract the elements larger than 0.5: `r1 = r[r>0.5]`, and then extract the elements in `r1` that are less than 0.6 and get the size of this array: `r1[r1<0.6].size`. Name of program file: `compute_prob_vec.py`.

Remark. An alternative and more complicated method is to use the `where` function. The condition (the first argument to `where`) is now a compound boolean expression `0.5 <= r <= 0.6`, but this cannot be used with NumPy arrays. Instead one must test for `0.5 <= r` *and* `r < = 0.6`. The needed boolean construction in the `where` call is `operator.and_(0.5 <= r, r <= 0.6)`. See the discussion of the this topic in Chapter 5.5.3. ◇

Exercise 8.19. *Throw dice and compute a small probability.*

Compute the probability of getting 6 eyes on all dice when rolling 7 dice. Since you need a large number of experiments in this case (see the first paragraph of Chapter 8.3), you can save quite some simulation time by using a vectorized implementation. Name of program file: `roll_7dice.py`. ◇

Exercise 8.20. *Difference equation for random numbers.*

Simple random number generators are based on simulating difference equations. Here is a typical set of two equations:

$$x_n = (ax_{n-1} + c) \bmod m, \tag{8.14}$$

$$y_n = x_n/m, \tag{8.15}$$

for $n = 1, 2, \ldots$. A seed x_0 must be given to start the sequence. The numbers $y_1, y_2, \ldots,$ represent the random numbers and $x_0, x_1, \ldots$ are "help" numbers. Although y_n is completely deterministic from (8.14)–(8.15), the sequence y_n *appears* random. The mathematical expression $p \bmod q$ is coded as `p % q` in Python.

Use $a = 8121$, $c = 28411$, and $m = 134456$. Solve the system (8.14)–(8.15) in a function that generates and returns N random numbers. Make a histogram to examine the distribution of the numbers (the y_n numbers are randomly distributed if the histogram is approximately flat). Name of program file: `diffeq_random.py`. ◇

Exercise 8.21. *Make a class for drawing balls from a hat.*

Consider the example about drawing colored balls from a hat in Chapter 8.3.3. It could be handy to have an object that acts as a hat:

```
# Make a hat with balls of 3 colors, each color appearing
# on 4 balls
hat = Hat(colors=('red', 'black', 'blue'), number_of_each_color=4)

# Draw 3 balls at random
balls = hat.draw(number_of_balls=3)
```

Realize such code with a class `Hat`. You can borrow useful code from the `balls_in_hat.py` program and ideas from Chapter 8.2.5. Use the `Hat` class to solve the probability problem from Exercise 8.4. Name of program file: `Hat.py`. ◇

Exercise 8.22. *Independent vs. dependent random numbers.*

Generate a sequence of N independent random variables with values 0 or 1 and print out this sequence without space between the numbers (i.e., as `001011010110111010`).

The next task is to generate random zeros and ones that are dependent. If the last generated number was 0, the probability of generating a new 0 is p and a new 1 is $1-p$. Conversely, if the last generated was 1, the probability of generating a new 1 is p and a new 0 is $1-p$. Since the new value depends on the last one, we say the variables are dependent. Implement this algorithm in a function returning an array of N zeros and ones. Print out this array in the condense format as described above.

Choose $N = 80$ and try the probabilities $p = 0.5$, $p = 0.8$ and $p = 0.9$. Can you by visual inspection of the output characterize the differences between sequences of independent and dependent random variables? Name of program file: `dependent_random_variables.py`. ◇

Exercise 8.23. *Compute the probability of flipping a coin.*

Modify the program from either Exercise 8.1 or 8.17 to incorporate the following extensions: look at a subset $N_1 \leq N$ of the experiments and compute probability of getting a head (M_1/N_1, where M_1 is the number of heads in N_1 experiments). Choose $N = 1000$ and print out the probability for $N_1 = 10, 100, 500, 1000$. (Generate just N numbers once in the program.) How do you think the accuracy of the computed probability vary with N_1? Is the output compatible with this expectation? Name of program file: `flip_coin_prob.py`. ◇

Exercise 8.24. *Extend Exer. 8.23.*

We address the same problem as in Exercise 8.23, but now we want to study the probability of getting a head, p, as a function of N_1, i.e., for $N_1 = 1, \ldots, N$. We also want to vectorize all operations in the code. A first try to compute the probability array for p is

```
import numpy as np
h = np.where(r <= 0.5, 1, 0)
p = np.zeros(N)
for i in range(N):
    p[i] = np.sum(h[:i+1])/float(i+1)
```

An array `q[i] = np.sum(h([:i]))` reflects a *cumulative sum* and can be efficiently generated by `np.cumsum`: `q = np.cumsum(h)`. Thereafter we can compute `p` by `q/I`, where `I[i]=i+1` and `I` can be computed by `np.arange(1,N+1)` or `r_[1:N+1]` (integers 1, 2, ..., up to but not including `N+1`). Implement both the loop over `i` and the vectorized version based on `cumsum` and check in the program that the resulting `p` array has the same elements (for this purpose you have to compare `float` elements and you can use the `float_eq` function from SciTools, see Exercise 2.24, or the `allclose` function in `numpy` (`float_eq` actually uses `allclose` for array arguments)). Plot `p` against `I` for the case where $N = 10000$. Annotate the axis and the plot with relevant text. Name of program file: `flip_coin_prob_developm.py`. ◇

Exercise 8.25. *Simulate the problems in Exer. 4.25.*

Exercise 4.25 describes some problems that can be solved exactly using the formula (4.8), but we can also simulate these problems and find approximate numbers for the probabilities. That is the task of this exercise.

Make a general function `simulate_binomial(p, n, x)` for running n experiments, where each experiment have two outcomes, with probabilities p and $1-p$. The n experiments constitute a "success" if the outcome with probability p occurs exactly x times. The `simulate_binomial` function must repeat the n experiments N times. If M is the number of "successes" in the N experiments, the probability estimate is M/N. Let the function return this probability estimate together with the error (the exact result is (4.8)). Simulate the three cases in Exercise 4.25 using this function. Name of program file: `simulate_binomial.py`. ◇

Exercise 8.26. *Simulate a poker game.*

Make a program for simulating the development of a poker (or simplified poker) game among n players. Use ideas from Chapter 8.2.4. Name of program file: `poker.py`. ◇

Exercise 8.27. *Write a non-vectorized version of a code.*

Read the file `birth_policy.py` containing the code from Chapter 8.3.5. To prove that you understand what is going on in this simulation, replace all the vectorized code by explicit loops over the random arrays. For such code it is natural to use Python's standard `random` module instead of `numpy.random`. However, to verify your alternative implementation it is important to have the same sequence of random numbers in the two programs. To this end, use `numpy.random`, but draw a single number at a time. Name of program file: `birth_policy2.py`. ◇

Exercise 8.28. *Estimate growth in a simulation model.*

The simulation model in Chapter 8.3.5 predicts the number of individuals from generation to generation. Make a simulation of the "one son" policy with 10 generations, a male portion of 0.51 among newborn babies, set the fertility to 0.92, and assume that 6% of the population will break the law and want 6 children in a family. These parameters implies a significant growth of the population. See if you can find a factor r such that the number of individuals in generation n fulfills the difference equation

$$x_n = (1 + r)x_{n-1}\,.$$

Hint: Compute r for two consecutive generations x_{n-1} and x_n ($r = x_n/x_{n-1} - 1$) and see if r is approximately constant as n increases. Name of program file: `estimate_growth.py`. ◇

Exercise 8.29. *Investigate guessing strategies for Ch. 8.4.1.*

In the game from Chapter 8.4.1 it is smart to use the feedback from the program to track an interval $[p, q]$ that must contain the secret number. Start with $p = 1$ and $q = 100$. If the user guesses at some number n, update p to $n + 1$ if n is less than the secret number (no need to care about numbers smaller than $n + 1$), or update q to $n - 1$ if n is larger than the secret number (no need to care about numbers larger than $n - 1$).

Are there any smart strategies to pick a new guess $s \in [p, q]$? To answer this question, investigate two possible strategies: s as the midpoint in the interval $[p, q]$, or s as a uniformly distributed random integer in $[p, q]$. Make a program that implements both strategies, i.e., the player is not prompted for a guess but the computer computes the guess based on the chosen strategy. Let the program run a large number of games and see if either of the strategies can be considered as superior in the long run. Name of program file: `strategies4guess.py`. ◇

Exercise 8.30. *Make a vectorized solution to Exer. 8.8.*

Vectorize the simulation program from Exercise 8.8 with the aid of the module `numpy.random` and the `numpy.sum` function. Name of program file: `sum9_4dice_vec.py`. ◇

Exercise 8.31. *Compute π by a Monte Carlo method.*

Use the method in Chapter 8.5.2 to compute π by computing the area of a circle. Choose G as the circle with its center at the origin and with unit radius, and choose B as the rectangle $[-1, 1] \times [-1, 1]$. A point (x, y) lies within G if $x^2 + y^2 < 1$. Compare the approximate π with `math.pi`. Name of program file: `MC_pi.py`. ◇

Exercise 8.32. *Implement a variant of Exer. 8.31.*

This exercise has the same purpose of computing π as in Exercise 8.31, but this time you should choose G as a circle with center at $(2, 1)$ and radius 4. Select an appropriate rectangle B. A point (x, y) lies within a circle with center at (x_c, y_c) and with radius R if $(x - x_c)^2 + (y - y_c)^2 < R^2$. Name of program file: `MC_pi2.py`. ◇

Exercise 8.33. *Compute π by a random sum.*

Let $x_0, \ldots, x_N$ be $N + 1$ uniformly distributed random numbers between 0 and 1. Explain why the random sum $S_N = (N + 1)^{-1} \sum_{i=0}^{N} 2(1 - x_i^2)^{-1/2}$ is an approximation to π. (Hint: Interpret the sum as Monte Carlo integration and compute the corresponding integral exactly by hand.) Make a program for plotting S_N versus N for $N = 10^k$, $k = 0, 1/2, 1, 3/2, 2, 5/2, \ldots, 6$. Write out the difference between S_{10^6} and `pi` from the `math` module. Name of program file: `MC_pi_plot.py`. ◇

Exercise 8.34. *1D random walk with drift.*

Modify the `walk1D.py` program such that the probability of going to the right is r and the probability of going to the left is $1 - r$ (draw numbers in $[0, 1)$ rather than integers in $\{1, 2\}$). Compute the average position of n_p particles after 100 steps, where n_p is read from the command line. Mathematically one can show that the average position approaches $rn_s - (1 - r)n_s$ as $n_p \to \infty$. Write out this exact result together with the computed mean position with a finite number of particles. Name of program file: `walk1D_drift.py`. ◇

Exercise 8.35. *1D random walk until a point is hit.*

Set `np=1` in the `walk1Dv.py` program and modify the program to measure how many steps it takes for one particle to reach a given point $x = x_p$. Give x_p on the command line. Report results for $x_p = 5, 50, 5000, 50000$. Name of program file: `walk1Dv_hit_point.py`. ◇

Exercise 8.36. *Simulate making a fortune from gaming.*

A man plays a game where the probability of winning is p and that of losing is consequently $1 - p$. When winning he earns 1 euro and when losing he loses 1 euro. Let x_i be the man's fortune from playing this game i number of times. The starting fortune is x_0. We assume that the man gets a necessary loan if $x_i < 0$ such that the gaming can continue. The target is a fortune F, meaning that the playing stops when $x = F$ is reached.

Explain why x_i is a 1D random walk. Modify one of the 1D random walk programs to simulate the average number of games it takes to reach the target fortune $x = F$. This average must be computed by running a large number of random walks that start at x_0 and reach F. Use $x_0 = 10$, $F = 100$, and $p = 0.49$ as example.

Suppose the average number of games to reach $x = F$ is proportional to $(F - x_0)^r$, where r is some exponent. Try to find r by experimenting with the program. The r value indicates how difficult it is to make a substantial fortune by playing this game. Note that the *expected* earning is negative when $p < 0.5$, but there is still a small probability for hitting $x = F$. Name of program file: `game_as_walk1D.py`. ◇

Exercise 8.37. *Make a class for 2D random walk.*

The purpose of this exercise is to reimplement the `walk2D.py` program from Chapter 8.7.1 with the aid of classes. Make a class `Particle` with the coordinates (x, y) and the time step number of a particle as attributes. A method `move` moves the particle in one of the four directions and updates the (x, y) coordinates. Another class, `Particles`, holds a list of `Particle` objects and a `plotstep` parameter (as in `walk2D.py`). A method `move` moves all the particles one step, a method `plot` can make a plot of all particles, while a method `moves` performs a loop over time steps and calls `move` and `plot` in each step.

Equip the `Particle` and `Particles` classes with print functionality such that one can print out all particles in a nice way by saying `print p` (for a `Particles` instance `p`) or `print self` (inside a method). Hint: In `__str__`, apply the `pformat` function from the `pprint` module to the list of particles, and make sure that `__repr__` just reuse `__str__` in both classes.

To verify the implementation, print the first three positions of four particles in the `walk2D.py` program and compare with the corresponding results produced by the class-based implementation (the seed of the random number generator must of course be fixed identically in the two programs). You can just perform `p.move()` and `print p` three times in a `verify` function to do this verification task.

Organize the complete code as a module such that the classes `Particle` and `Particles` can be reused in other programs. The test block should call a `run(N)` method to run the walk for `N` steps, where `N` is given on the command line.

Compare the efficiency of the class version against the vectorized version in `walk2Dv.py`, using the techniques of Appendix H.6.1. Name of program file: `walk2Dc.py`. ◇

Exercise 8.38. *Vectorize the class code from Exer. 8.37.*

The program developed in Exercise 8.37 cannot be vectorized as long as we base the implementation on class `Particle`. However, if we remove that class and focus on class `Particles`, the latter can employ arrays for holding the positions of all particles and vectorized updates of these positions in the `moves` method. Use ideas from the `walk2Dv.py` program to vectorize class `Particle`. Verify the code against `walk2Dv.py` as explained in Exercise 8.37, and measure the efficiency gain over the version with class `Particle`. Name of program file: `walk2Dcv.py`. ◇

Exercise 8.39. *2D random walk with walls; scalar version.*

Modify the `walk2D.py` or `walk2Dc.py` programs from Exercise 8.37 so that the walkers cannot walk outside a rectangular area $A = [x_L, x_H] \times [y_L, y_H]$. Do not move the particle if its new position is outside A. Name of program file: `walk2D_barrier.py`. ◇

Exercise 8.40. *2D random walk with walls; vectorized version.*

Modify the `walk2Dv.py` program so that the walkers cannot walk outside a rectangular area $A = [x_L, x_H] \times [y_L, y_H]$. Hint: First perform the moves of one direction. Then test if new positions are outside A. Such a test returns a boolean array that can be used as index in the position arrays to pick out the indices of the particles that have moved outside A and move them back to the relevant boundary of A. Name of program file: `walk2Dv_barrier.py`. ◇

Exercise 8.41. *Simulate the mixture of gas molecules.*

Suppose we have a box with a wall dividing the box into two equally sized parts. In one part we have a gas where the molecules are uniformly distributed in a random fashion. At $t = 0$ we remove the wall. The gas molecules will now move around and eventually fill the whole box.

This physical process can be simulated by a 2D random walk inside a fixed area A as introduced in Exercises 8.39 and 8.40 (in reality the motion is three-dimensional, but we only simulate the two-dimensional part of it since we already have programs for doing this). Use the program from either Exercises 8.39 or 8.40 to simulate the process for $A = [0, 1] \times [0, 1]$. Initially, place 10000 particles at uniformly distributed random positions in $[0, 1/2] \times [0, 1]$. Then start the random walk and visualize what happens. Simulate for a long time and make a hardcopy of the animation (an animated GIF file, for instance). Is the end result what you would expect? Name of program file: `disorder1.py`.

Molecules tend to move randomly because of collisions and forces between molecules. We do not model collisions between particles in the random walk, but the nature of this walk, with random movements, simulates the effect of collisions. Therefore, the random walk can be used to model molecular motion in many simple cases. In particular, the random walk can be used to investigate how a quite ordered system, where one gas fills one half of a box, evolves through time to a more disordered system. ◇

Exercise 8.42. *Solve a variant of Exer. 8.41.*

Solve Exercise 8.41 when the wall dividing the box is not completely removed, but instead we make a small hole in the wall initially. Name of program file: `disorder2.py`. ◇

Exercise 8.43. *Guess beer brands.*

You are presented n glasses of beer, each containing a different brand. You are informed that there are $m \geq n$ possible brands in total, and the names of all brands are given. For each glass, you can pay p euros to taste the beer, and if you guess the right brand, you get $q \geq p$ euros back. Suppose you have done this before and experienced that you typically manage to guess the right brand T times out of 100, so that your probability of guessing the right brand is $b = T/100$.

Make a function `simulate(m, n, p, q, b)` for simulating the beer tasting process. Let the function return the amount of money earned and how many correct guesses ($\leq n$) you made. Call `simulate` a large number of times and compute the average earnings and the probability of getting full score in the case $m = n = 4$, $p = 3$, $q = 6$, and $b = 1/m$ (i.e., four glasses with four brands, completely random guessing, and a payback of twice as much as the cost). How much more can you earn from this game if your ability to guess the right brand is better, say $b = 1/2$? Name of program file: `simulate_beer_tasting.py`. ◇

Exercise 8.44. *Simulate stock prices.*

A common mathematical model for the evolution of stock prices can be formulated as a difference equation

$$x_n = x_{n-1} + \Delta t \mu x_{n-1} + \sigma x_{n-1}\sqrt{\Delta t} r_{n-1}, \tag{8.16}$$

where x_n is the stock price at time t_n, Δt is the time interval between two time levels ($\Delta t = t_n - t_{n-1}$), μ is the growth rate of the stock price, σ is the volatility of the stock price, and $r_0, \ldots, r_{n-1}$ are normally distributed random numbers with mean zero and unit standard deviation. An initial stock price x_0 must be prescribed together with the input data μ, σ, and Δt.

We can make a remark that Equation (8.16) is a Forward Euler discretization of a stochastic differential equation for $x(t)$:

$$\frac{dx}{dt} = \mu x + \sigma N(t),$$

where $N(t)$ is a so-called white noise random time series signal. Such equations play a central role in modeling of stock prices.

Make R realizations of (8.16) for $n = 0, \ldots, N$ for $N = 5000$ steps over a time period of $T = 180$ days with a step size $\Delta t = T/N$. Name of program file: `stock_prices.py`. ◇

Exercise 8.45. *Compute with option prices in finance.*

In this exercise we are going to consider the pricing of so-called Asian options. An Asian option is a financial contract where the owner earns money when certain market conditions are satisfied.

The contract is specified by a *strike price* K and a maturity time T. It is written on the average price of the underlying stock, and if this average is bigger than the strike K, the owner of the option will earn the difference. If, on the other hand, the average becomes less, the owner receives nothing, and the option matures in the value zero. The average is calculated from the last trading price of the stock for each day.

From the theory of options in finance, the price of the Asian option will be the expected present value of the payoff. We assume the stock price dynamics given as,

$$S(t+1) = (1+r)S(t) + \sigma S(t)\epsilon(t), \tag{8.17}$$

where r is the interest-rate, and σ is the volatility of the stock price. The time t is supposed to be measured in days, $t = 0, 1, 2, \ldots$, while $\epsilon(t)$ are independent identically distributed normal random variables with mean zero and unit standard deviation. To find the option price, we must calculate the expectation

$$p = (1+r)^{-T}\mathrm{E}\left[\max\left(\frac{1}{T}\sum_{t=1}^{T} S(t) - K, 0\right)\right]. \tag{8.18}$$

The price is thus given as the expected discounted payoff. We will use Monte Carlo simulations to estimate the expectation. Typically, r and σ can be set to $r = 0.0002$ and $\sigma = 0.015$. Assume further $S(0) = 100$.

a) Make a function that simulates a path of $S(t)$, that is, the function computes $S(t)$ for $t = 1, \ldots, T$ for a given T based on the recursive definition in (8.17). The function should return the path as an array.
b) Create a function that finds the average of $S(t)$ from $t = 1$ to $t = T$. Make another function that calculates the price of the Asian option based on N simulated averages. You may choose $T = 100$ days and $K = 102$.
c) Plot the price p as a function of N. You may start with $N = 1000$.
d) Plot the error in the price estimation as a function N (assume that the p value corresponding to the largest N value is the "right" price). Try to fit a curve of the form $c/\sqrt{N}$ for some c to this error plot. The purpose is to show that the error is reduced as $1/\sqrt{N}$.

Name of program file: `option_price.py`.

If you wonder where the values for r and σ come from, you will find the explanation in the following. A reasonable level for the yearly interest-rate is around 5%, which corresponds to a daily rate $0.05/250 = 0.0002$. The number 250 is chosen because a stock exchange is on average open this amount of days for trading. The value for σ is calculated as the volatility of the stock price, corresponding to the standard deviation of the daily returns of the stock defined as $(S(t + 1) - S(t))/S(t)$. "Normally", the volatility is around 1.5% a day. Finally, there are theoretical reasons why we assume that the stock price dynamics is driven by r, meaning that we consider the *risk-neutral* dynamics of the stock price when pricing options. There is an exciting theory explaining the appearance of r in the dynamics of the stock price. If we want to simulate a stock price dynamics mimicing what we see in the market, r in Equation (8.17) must be substituted with μ, the expected return of the stock. Usually, μ is higher than r. $\diamond$

Exercise 8.46. *Compute velocity and acceleration.*

In a laboratory experiment waves are generated through the impact of a model slide into a wave tank. (The intention of the experiment is to model a future tsunami event in a fjord, generated by loose rocks that fall into the fjord.) At a certain location, the elevation of the surface, denoted by η, is measured at discrete points in time using an ultra-sound wave gauge. The result is a time series of vertical positions of the water surface elevations in meter: $\eta(t_0), \eta(t_1), \eta(t_2), \ldots, \eta(t_n)$. There are 300 observations per second, meaning that the time difference between two neighboring measurement values $\eta(t_i)$ and $\eta(t_{i+1})$ is $h = 1/300$ second.

Write a Python program that accomplishes the following tasks:

1. Read h from the command line.

2. Read the η values in the file `src/random/gauge.dat` into an array `eta`.
3. Plot `eta` versus the time values.
4. Compute the velocity v of the surface by the formula

$$v_i \approx \frac{\eta_{i+1} - \eta_{i-1}}{2h}, \quad i = 1, \ldots, n-1\,.$$

   Plot v versus time values in a separate plot.
5. Compute the acceleration a of the surface by the formula

$$a_i \approx \frac{\eta_{i+1} - 2\eta_i + \eta_{i-1}}{h^2}, \quad i = 1, \ldots, n-1\,.$$

   Plot a versus the time values in a separate plot.

Name of program file: `labstunami1.py`. ◇

Exercise 8.47. *Differentiate noisy signals.*

The purpose of this exercise is to look into numerical differentiation of time series signals that contain measurement errors. This insight might be helpful when analyzing the noise in real data from a laboratory experiment in Exercises 8.46 and 8.48.

1. Compute a signal

$$\bar{\eta}_i = A\sin(\frac{2\pi}{T}t_i), \quad t_i = i\frac{T}{40}, \ i = 0, \ldots, 200\,.$$

   Display $\bar{\eta}_i$ versus time t_i in a plot. Choose $A = 1$ and $T = 2\pi$. Store the $\bar{\eta}$ values in an array `etabar`.
2. Compute a signal with random noise E_i,

$$\eta_i = \bar{\eta}_i + E_i,$$

   E_i is drawn from the normal distribution with mean zero and standard deviation $\sigma = 0.04A$. Plot this η_i signal as circles in the same plot as $\bar{\eta}_i$. Store the E_i in an array `E` for later use.
3. Compute the first derivative of $\bar{\eta}_i$ by the formula

$$\frac{\bar{\eta}_{i+1} - \bar{\eta}_{i-1}}{2h}, \quad i = 1, \ldots, n-1,$$

   and store the values in an array `detabar`. Display the graph.
4. Compute the first derivative of the error term by the formula

$$\frac{E_{i+1} - E_{i-1}}{2h}, \quad i = 1, \ldots, n-1,$$

   and store the values in an array `dE`. Calculate the mean and the standard deviation of `dE`.
5. Plot `detabar` and `detabar + dE`. Use the result of the standard deviation calculations to explain the qualitative features of the graphs.

6. The second derivative of a time signal η_i can be computed by

$$\frac{\eta_{i+1} - 2\eta_i + \eta_{i-1}}{h^2}, \quad i = 1, \ldots, n-1\,.$$

Use this formula on the `etabar` data and save the result in `d2etabar`. Also apply the formula to the `E` data and save the result in `d2E`. Plot `d2etabar` and `d2etabar + d2E`. Compute the standard deviation of `d2E` and compare with the standard deviation of `dE` and `E`. Discuss the plot in light of these standard deviations.

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

Exercise 8.48. *Model the noise in the data in Exer. 8.47.*

We assume that the measured data can be modeled as a smooth time signal $\bar{\eta}(t)$ plus a random variation $E(t)$. Computing the velocity of $\eta = \bar{\eta} + E$ results in a smooth velocity from the $\bar{\eta}$ term and a noisy signal from the E term. We can estimate the level of noise in the first derivative of E as follows. The random numbers $E(t_i)$ are assumed to be independent and normally distributed with mean zero and standard deviation σ. It can then be shown that

$$\frac{E_{i+1} - E_{i-1}}{2h}$$

produces numbers that come from a normal distribution with mean zero and standard deviation $2^{-1/2}h^{-1}\sigma$. How much is the original noise, reflected by σ, magnified when we use this numerical approximation of the velocity?

The fraction

$$\frac{E_{i+1} - 2E_i + E_{i-1}}{h^2}$$

will also generate numbers from a normal distribution with mean zero, but this time with standard deviation $2h^{-2}\sigma$. Find out how much the noise is magnified in the computed acceleration signal.

The numbers in the `gauge.dat` file are given with 5 digits. This is no certain indication of the accuracy of the measurements, but as a test we may assume σ is of the order 10^{-4}. Check if the visual results for the velocity and acceleration are consistent with the standard deviation of the noise in these signals as modeled above. ◇

Exercise 8.49. *Reduce the noise in Exer. 8.47.*

If we have a noisy signal η_i, where $i = 0, \ldots, n$ counts time levels, the noise can be reduced by computing a new signal where the value at a point is a weighted average of the values at that point and the neighboring points at each side. More precisely, given the signal η_i, $i = 0, \ldots, n$, we compute a filtered (averaged) signal with values $\eta_i^{(1)}$ by the formula

$$\eta_i^{(1)} = \frac{1}{4}(\eta_{i+1} + 2\eta_i + \eta_{i-1}), \quad i = 1, \ldots, n-1,\ \eta_0^{(1)} = \eta_0,\ \eta_n^{(1)} = \eta_n\,. \tag{8.19}$$

Make a function `filter` that takes the η_i values in an array `eta` as input and returns the filtered $\eta_i^{(1)}$ values in an array. Let $\eta_i^{(k)}$ be the signal arising by applying the `filtered` function k times to the same signal. Make a plot with curves η_i and the filtered $\eta_i^{(k)}$ values for $k = 1, 10, 100$. Make similar plots for the velocity and acceleration where these are made from both the original η data and the filtered data. Discuss the results. Name of program file: `labstunami2.py`. ◇

Exercise 8.50. *Make a class for differentiating noisy data.*

Suppose you have some time series signal $y(t_k)$ for $k = 0, \ldots, n-1$, where $t_k = k\Delta t$ are time points. Differentiating such a signal can give very inaccurate results if the signal contains noise. Exercises 8.46–8.49 explore this topic, and Exercise 8.49 suggests to filter the signal. The purpose of the present exercise is to make a tool for differentiating noisy signals.

Make a class `DiffNoisySignal` where the constructor takes three arguments: the signal $y(t_k)$ (as an array), the order of the desired derivative (as an `int`, either 1 or 2), and the name of the signal (as a string). A method `filter(self, n)` runs the filter from Exercise 8.49 `n` times on the signal. The method `diff(self)` performs the differentiation and stores the differentiated signal as an attribute in the class. There should also be some plotting methods: `plot(self)` for plotting the current (original or filtered) signal, `plot_diff(self)` for plotting the differentiated signal, `animate_filter` for animating the effect of filtering (run `filter` once per frame in the movie), and `animate_diff` for animating the evolution of the derivative when `filter` and `diff` are called once each per frame.

Implement the class and test it on the noisy signal

$$y(t_k) = \cos(2\pi t_k) + 0.1 r_k, \quad t_k = k\Delta t,\ k = 0, \ldots, n-1,$$

with $\Delta t = 1/60$. The quantities r_k are random numbers in $[0, 1)$. Make animations with the `animate_filter` and `animate_diff` methods. Name of program file: `DiffNoisySignal.py`. ◇

Exercise 8.51. *Speed up Markov chain mutation.*

The functions `transition` and `mutate_via_markov_chain` from Chapter 8.3.4 were made for being easy to read and understand. Upon closer inspection, we realize that the `transition` function constructs the `interval_limits` every time a random transition is to be computed, and we want to run a large number of transitions. By (i) merging the two functions, (ii) pre-computing interval limits for each `from_base`, and (iii) adding a loop over `N` mutations, one can reduce the computation of interval limits to a minimum. Perform such an efficiency enhancement. Measure the CPU time of this new function versus the `mutate_via_markov_chain` function for 1 million mutations. Name of program file: `markov_chain_mutation2.py`. ◇

Terminal

```
demo_ReadInput.py GUI
{'a': -1, 'b': 10, 'filename': 'tmp.dat',
 'formula': 'x+1', 'n': 2}
```

The GUI is shown in Figure 9.13.

Fortunately, it is now quite obvious how to apply the `ReadInput` hierarchy of classes in your own programs to simplify input. Especially in applications with a large number of parameters one can initially define these in a dictionary and then automatically create quite comprehensive user interfaces where the user can specify only some subset of the parameters (if the default values for the rest of the parameters are suitable).

9.7 Exercises

Exercise 9.1. *Demonstrate the magic of inheritance.*

Consider class `Line` from Chapter 9.1.1 and a subclass `Parabola0` defined as

```
class Parabola0(Line):
    pass
```

That is, class `Parabola0` does not have any own code, but it inherits from class `Line`. Demonstrate in a program or interactive session, using methods from Chapter 7.5.5, that an instance of class `Parabola0` contains everything (i.e., all attributes and methods) that an instance of class `Line` contains. Name of program file: `dir_subclass.py`. ◇

Exercise 9.2. *Inherit from classes in Ch. 9.1.*

The task in this exercise is to make a class `Cubic` for cubic functions

$$c_3x^3 + c_2x^2 + c_1x + c_0$$

with a call operator and a `table` method as in classes `Line` and `Parabola` from Chapter 9.1. Implement class `Cubic` by inheriting from class `Parabola`, and call up functionality in class `Parabola` in the same way as class `Parabola` calls up functionality in class `Line`.

Make a similar class `Poly4` for 4-th degree polynomials

$$c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

by inheriting from class `Cubic`. Insert `print` statements in all the `__call__` to help following the program flow. Evaluate cubic and a 4-th degree polynomial at a point, and observe the printouts from all the superclasses. Name of program file: `Cubic_Poly4.py`. ◇

Exercise 9.3. *Inherit more from classes in Ch. 9.1.*

Implement a class for the function $f(x) = A\sin(wx) + ax^2 + bx + c$. The class should have a call operator for evaluating the function for some argument x, and a constructor that takes the function parameters `A`, `w`, `a`, `b`, and `c` as arguments. Also a `table` method as in classes `Line` and `Parabola` should be present. Implement the class by deriving it from class `Parabola` and call up functionality already implemented in class `Parabola` whenever possible. Name of program file: `sin_plus_quadratic.py`. ◇

Exercise 9.4. *Reverse the class hierarchy from Ch. 9.1.*

Let class `Polynomial` from Chapter 7.3.7 be a superclass and implement class `Parabola` as a subclass. The constructor in class `Parabola` should take the three coefficients in the parabola as separate arguments. Try to reuse as much code as possible from the superclass in the subclass. Implement class `Line` as a subclass specialization of class `Parabola`.

Which class design do you prefer – class `Line` as a subclass of `Parabola` and `Polynomial`, or `Line` as a superclass with extensions in subclasses? Name of program file: `Polynomial_hier.py`. ◇

Exercise 9.5. *Make circle a subclass of an ellipse.*

Chapter 7.2.3 presents class `Circle`. Make a similar class `Ellipse` for representing an ellipse. Then create a new class `Circle` that is a subclass of `Ellipse`. Name of program file: `Ellipse_Circle.py`. ◇

Exercise 9.6. *Make super- and subclass for a point.*

A point (x, y) in the plane can be represented by a class:

```
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __str__(self):
        return '(%g, %g)' % (self.x, self.y)
```

We can extend the `Point` class to also contain the representation of the point in polar coordinates. To this end, create a subclass `PolarPoint` whose constructor takes the polar representation of a point, (r, θ), as arguments. Store r and θ as attributes and call the superclass constructor with the corresponding x and y values (recall the relations $x = r\cos\theta$ and $y = r\sin\theta$ between Cartesian and polar coordinates). Add a `__str__` method in class `PolarPoint` which prints out r, θ, x, and y. Verify the implementation by initializing three points and printing these points. Name of program file: `PolarPoint.py`. ◇

Exercise 9.7. *Modify a function class by subclassing.*

Consider the `VelocityProfile` class from page 350 for computing the function $v(r; \beta, \mu_0, n, R)$ in formula (5.23) on page 252. Suppose we want to have v explicitly as a function of r and n (this is necessary if we want to illustrate how the velocity profile, the $v(r)$ curve, varies as n varies). We would then like to have a class `VelocityProfile2` that is initialized with β, μ_0, and R, and that takes r and n as arguments in the `__call__` method. Implement such a class by inheriting from class `VelocityProfile` and by calling the `__init__` and `value` methods in the superclass. It should be possible to try the class out with the following statements:

```
v = VelocityProfile2(beta=0.06, mu0=0.02, R=2)
# Evaluate v for various n values at r=0
for n in 0.1, 0.2, 1:
    print v(0, n)
```

Name of program file: `VelocityProfile2.py`. ◇

Exercise 9.8. *Explore the accuracy of difference formulas.*

The purpose of this exercise is to investigate the accuracy of the `Backward1`, `Forward1`, `Forward3`, `Central2`, `Central4`, `Central6` methods for the function[8]

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

To solve the exercise, modify the `src/oo/Diff2_examples.py` program which produces tables of errors of difference approximations as discussed at the end of Chapter 9.2.3. Test the approximation methods for $x = 0, 0.9$ and $\mu = 1, 0.01$. Plot the $v(x)$ function for the two μ values using 1001 points. Name of program file: `boundary_layer_derivative.py`. ◇

Exercise 9.9. *Implement a subclass.*

Make a subclass `Sine1` of class `FuncWithDerivatives` from Chapter 9.1.7 for the $\sin x$ function. Implement the function only, and rely on the inherited `df` and `ddf` methods for computing the derivatives. Make another subclass `Sine2` for $\sin x$ where you also implement the `df` and `ddf` methods using analytical expressions for the derivatives. Compare `Sine1` and `Sine2` for computing the first- and second-order derivatives of $\sin x$ at two x points. Name of program file: `Sine12.py`. ◇

Exercise 9.10. *Make classes for numerical differentiation.*

Carry out Exercise 7.13. Find the common code in the classes `Derivative`, `Backward`, and `Central`. Move this code to a superclass, and let the three mentioned classes be subclasses of this superclass. Compare the resulting code with the hierarchy shown in Chapter 9.2.1. Name of program file: `numdiff_classes.py`. ◇

Exercise 9.11. *Implement a new subclass for differentiation.*

A one-sided, three-point, second-order accurate formula for differentiating a function $f(x)$ has the form

[8] This function is discussed more in detail in Exercise 5.40.

$$f'(x) \approx \frac{f(x-2h) - 4f(x-h) + 3f(x)}{2h}\,. \tag{9.17}$$

Implement this formula in a subclass `Backward2` of class `Diff` from Chapter 9.2. Compare `Backward2` with `Backward1` for $g(t) = e^{-t}$ for $t = 0$ and $h = 2^{-k}$ for $k = 0, 1, \ldots, 14$ (write out the errors in $g'(t)$). Name of program file: `Backward2.py`. ◇

Exercise 9.12. *Understand if a class can be used recursively.*

Suppose you want to compute $f''(x)$ of some mathematical function $f(x)$, and that you apply some class from Chapter 9.2 twice, e.g.,

```
ddf = Central2(Central2(f))
```

Will this work? Hint: Follow the program flow, and find out what the resulting formula will be. Then see if this formula coincides with a formula you know for approximating $f''(x)$ (actually, to recover the well-known formula with an h parameter, you would use $h/2$ in the nested calls to `Central2`). ◇

Exercise 9.13. *Represent people by a class hierarchy.*

Classes are often used to model objects in the real world. We may represent the data about a person in a program by a class `Person`, containing the person's name, address, phone number, date of birth, and nationality. A method `__str__` may print the person's data. Implement such a class `Person`.

A worker is a person with a job. In a program, a worker is naturally represented as class `Worker` derived from class `Person`, because a worker *is* a person, i.e., we have an is-a relationship. Class `Worker` extends class `Person` with additional data, say name of company, company address, and job phone number. The print functionality must be modified accordingly. Implement this `Worker` class.

A scientist is a special kind of a worker. Class `Scientist` may therefore be derived from class `Worker`. Add data about the scientific discipline (physics, chemistry, mathematics, computer science, ...). One may also add the type of scientist: theoretical, experimental, or computational. The value of such a type attribute should not be restricted to just one category, since a scientist may be classified as, e.g., both experimental and computational (i.e., you can represent the value as a list or tuple). Implement class `Scientist`.

Researcher, postdoc, and professor are special cases of a scientist. One can either create classes for these job positions, or one may add an attribute (`position`) for this information in class `Scientist`. We adopt the former strategy. When, e.g., a researcher is represented by a class `Researcher`, no extra data or methods are needed. In Python we can create such an "empty" class by writing `pass` (the empty statement) as the class body:

```
class Researcher(Scientist):
    pass
```

Finally, make a demo program where you create and print instances of classes `Person`, `Worker`, `Scientist`, `Researcher`, `Postdoc`, and `Professor`. Print out the attribute contents of each instance (use the `dir` function).

Remark. An alternative design is to introduce a class `Teacher` as a special case of `Worker` and let `Professor` be both a `Teacher` and `Scientist`, which is natural. This implies that class `Professor` has two superclasses, `Teacher` and `Scientist`, or equivalently, class `Professor` inherits from two superclasses. This is known as *multiple inheritance* and technically achieved as follows in Python:

```
class Professor(Teacher, Scientist):
    pass
```

It is a continuous debate in computer science whether multiple inheritance is a good idea or not. One obvious problem[9] in the present example is that class `Professor` inherits two names, one via `Teacher` and one via `Scientist` (both these classes inherit from `Person`). Neither of the two widely used languages Java and C# allow multiple inheritance. Nor in this book will we pursue the idea of multiple inheritance further. Name of program file: `Person.py`. ◇

Exercise 9.14. *Add a new class in a class hierarchy.*

Add the Monte Carlo integration method from Chapter 8.5.1 as a subclass in the `Integrator` hierarchy explained in Chapter 9.3. Import the superclass `Integrator` from the `integrate` module in the file with the new integration class. Test the Monte Carlo integration class in a case with known analytical solution. Name of program file: `MCint_class.py`. ◇

Exercise 9.15. *Change the user interface of a class hierarchy.*

All the classes in the `Integrator` hierarchy from Chapter 9.3 take the integration limits a and b plus the number of integration points n as input to the constructor. The `integrate` method takes the function to integrate, $f(x)$, as parameter. Another possibility is to feed $f(x)$ to the constructor and let `integrate` take a, b, and n as parameters. Make this change to the `integrate.py` file with the `Integrator` hierarchy. Name of program file: `integrate2.py`. ◇

Exercise 9.16. *Compute convergence rates of numerical integration methods.*

Most numerical methods have a discretization parameter, call it n, such that if n increases (or decreases), the method performs better.

[9] It is usually not a technical problem, but more a conceptual problem when the world is modeled by objects in a program.

Often, the relation between the error in the numerical approximation (compared with the exact analytical result) can be written as

$$E = Cn^r,$$

where E is the error, and C and r are constants.

Suppose you have performed an experiment with a numerical method using discretization parameters $n_0, n_1, \ldots, n_N$. You have computed the corresponding errors $E_0, E_1, \ldots, E_N$ in a test problem with an analytical solution. One way to estimate r goes as follows. For two successive experiments we have

$$E_{i-1} = Cn_{i-1}^r$$

and

$$E_i = Cn_i^r\,.$$

Divide the first equation by the second to eliminate C, and then take the logarithm to solve for r:

$$r = \frac{\ln(E_{i-1}/E_i)}{\ln(n_{i-1}/n_i)}\,.$$

We can compute r for all pairs of two successive experiments. Usually, the "last r", corresponding to $i = N$ in the formula above, is the "best" r value[10]. Knowing r, we can compute C as $E_N n_N^{-r}$.

Having stored the n_i and E_i values in two lists `n` and `E`, the following code snippet computes r and C:

```
from scitools.convergencerate import convergence_rate
C, r = convergence_rate(n, E)
```

Construct a test problem for integration where you know the analytical result of the integral. Run different numerical methods (the Midpoint method, the Trapezoidal method, Simpson's method, Monte Carlo integration) with the number of evaluation points $n = 2^k + 1$ for $k = 2, \ldots, 11$, compute corresponding errors, and use the code snippet above to compute the r value for the different methods in questions. The higher the absolute error of r is, the faster the method converges to the exact result as n increases, and the better the method is. Which is the best and which is the worst method?

Let the program file import methods from the `integrate` module and the module with the Monte Carlo integration method from Exercise 9.14. Name of program file: `integrators_convergence.py`. ◇

[10] This guideline is good if the method converges and round-off errors do not influence the values of E_i. For very large/small n, the computation of r may be unreliable.

Exercise 9.17. *Add common functionality in a class hierarchy.*

Suppose you want to use classes in the `Integrator` hierarchy from Chapter 9.3 to calculate integrals of the form

$$F(x) = \int_a^x f(t)dt\,.$$

Such functions $F(x)$ can be efficiently computed by the method from Exercise 7.28. Implement this computation of $F(x)$ in an additional method in the superclass `Integrator`. Test that the implementation is correct for $f(x) = 2x - 3$ for all the implemented integration methods (the Midpoint, Trapezoidal and Gauss-Legendre methods, as well as Simpson's rule, integrate a linear function exactly). Name of program file: `integrate_efficient.py`. ◇

Exercise 9.18. *Make a class hierarchy for root finding.*

Given a general nonlinear equation $f(x) = 0$, we want to implement classes for solving such an equation, and organize the classes in a class hierarchy. Make classes for three methods: Newton's method (Appendix A.1.10), the Bisection method (Chapter 4.6.2), and the Secant method (Exercise A.14).

It is not obvious how such a hierarchy should be organized. One idea is to let the superclass store the $f(x)$ function and its derivative $f'(x)$ (if provided – if not, use a finite difference approximation for $f'(x)$). A method

```
def solve(start_values=[0], max_iter=100, tolerance=1E-6):
    ...
```

in the superclass can implement a general iteration loop. The `start_values` argument is a list of starting values for the algorithm in question: one point for Newton, two for Secant, and an interval $[a, b]$ containing a root for Bisection. Let `solve` define a list `self.x` holding all the computed approximations. The initial value of `self.x` is simply `start_values`. For the Bisection method, one can use the convention $a, b, c =$ `self.x[-3:]`, where $[a, b]$ represents the most recently computed interval and c is its midpoint. The `solve` method can return an approximate root x, the corresponding $f(x)$ value, a boolean indicator that is `True` if $|f(x)|$ is less than the `tolerance` parameter, and a list of all the approximations and their f values (i.e., a list of $(x, f(x))$ tuples).

Do Exercise A.15 using the new class hierarchy. Name of program file: `Rootfinders.py`. ◇

Exercise 9.19. *Make a calculus calculator class.*

Given a function $f(x)$ defined on a domain $[a, b]$, the purpose of many mathematical exercises is to sketch the function curve $y = f(x)$, compute the derivative $f'(x)$, find local and global extreme points, and compute the integral $\int_a^b f(x)dx$. Make a class `CalculusCalculator` which can perform all these actions for any function $f(x)$ using numerical differentiation and integration, and the method explained in Exercise 7.42 or 7.43 for finding extrema.

Here is an interactive session with the class where we analyze $f(x) = x^2 e^{-0.2x}\sin(2\pi x)$ on $[0, 6]$ with a grid (set of x coordinates) of 700 points:

```
>>> from CalculusCalculator import *
>>> def f(x):
...     return x**2*exp(-0.2*x)*sin(2*pi*x)
...
>>> c = CalculusCalculator(f, 0, 6, resolution=700)
>>> c.plot()               # plot f
>>> c.plot_derivative()    # plot f'
>>> c.extreme_points()

All minima: 0.8052, 1.7736, 2.7636, 3.7584, 4.7556, 5.754, 0
All maxima: 0.3624, 1.284, 2.2668, 3.2604, 4.2564, 5.2548, 6
Global minimum: 5.754
Global maximum: 5.2548

>>> c.integral
-1.7353776102348935
>>> c.df(2.51)    # c.df(x) is the derivative of f
-24.056988888465636
>>> c.set_differentiation_method(Central4)
>>> c.df(2.51)
-24.056988832723189
>>> c.set_integration_method(Simpson)  # more accurate integration
>>> c.integral
-1.7353857856973565
```

Design the class such that the above session can be carried out.

Hint: Use classes from the `Diff` and `Integrator` hierarchies (Chapters 9.2 and 9.3) for numerical differentiation and integration (with, e.g., `Central2` and `Trapezoidal` as default methods for differentiation and integration). The method `set_differentiation_method` takes a subclass name in the `Diff` hierarchy as argument, and makes an attribute `df` that holds a subclass instance for computing derivatives. With `set_integration_method` we can similarly set the integration method as a subclass name in the `Integrator` hierarchy, and then compute the integral $\int_a^b f(x)dx$ and store the value in the attribute `integral`. The `extreme_points` method performs a `print` on a `MinMax` instance, which is stored as an attribute in the calculator class. Name of program file: `CalculusCalculator.py`. ◇

Exercise 9.20. *Extend Exer. 9.19.*

Extend class `CalculusCalculator` from Exercise 9.19 to offer computations of inverse functions. A numerical way of computing inverse functions is explained in Appendix A.1.11. Exercise 7.26 suggests an improved implementation using classes. Use the `InverseFunction` implementation from Exercise 7.26 in class `CalculusCalculator`. Name of program file: `CalculusCalculator2.py`. ◇

Exercise 9.21. *Make line drawing of a person; program.*

A very simple sketch of a human being can be made of a circle for the head, two lines for the arms, one vertical line, a triangle, or a rectangle for the torso, and two lines for the legs. Make such a drawing in a program, utilizing appropriate classes in the `Shape` hierarchy. Name of program file: `draw_person.py`. ◇

Exercise 9.22. *Make line drawing of a person; class.*

Use the code from Exercise 9.21 to make a subclass of `Shape` that draws a person. Supply the following arguments to the constructor: the center point of the head and the radius R of the head. Let the arms and the torso be of length $4R$, and the legs of length $6R$. The angle between the legs can be fixed (say 30 degrees), while the angle of the arms relative to the torso can be an argument to the constructor with a suitable default value. Name of program file: `Person.py`. ◇

Exercise 9.23. *Animate a person with waving hands.*

Make a subclass of the class from Exercise 9.22 where the constructor can take an argument describing the angle between the arms and the torso. Use this new class to animate a person who waves her/his hands. Name of program file: `waving_person.py`. ◇

'logistic', performing an eval(seqtype)(N) is the same as if we had written logistic(N). This technique allows the user of the program to choose a function call inside the code. Without eval we would need to explicitly test on values:

```
if seqtype == 'logistic':
    x = logistic(N)
elif seqtype == 'oscillations':
    x = oscillations(N)
```

This is not much extra code to write in the present example, but if we have a large number of functions generating sequences, we can save a lot of boring if-else code by using the eval construction.

The next step, as a reader who have understood the problem and the implementation above, is to run the program for two cases: the oscillations sequence with $N = 40$ and the logistic sequence with $N = 100$. By altering the q parameter to lower values, you get other sounds, typically quite boring sounds for non-oscillating logistic growth ($q < 1$). You can also experiment with other transformations of the form (A.46), e.g., increasing the frequency variation from 200 to 400.

A.3 Exercises

Exercise A.1. *Determine the limit of a sequence.*

Given the sequence

$$a_n = \frac{7 + 1/n}{3 - 1/n^2},$$

make a program that computes and prints out a_n for $n = 1, 2, \ldots, N$. Read N from the command line. Does a_n approach a finite limit when $n \to \infty$? Name of program file: sequence_limit1.py. ◇

Exercise A.2. *Determine the limit of a sequence.*

Solve Exercise A.1 when the sequence of interest is given by

$$D_n = \frac{\sin(2^{-n})}{2^{-n}}\,.$$

Name of program file: sequence_limit2.py. ◇

Exercise A.3. *Experience convergence problems.*

Given the sequence

$$D_n = \frac{f(x+h) - f(x)}{h}, \quad h = 2^{-n} \tag{A.47}$$

make a function D(f, x, N) that takes a function $f(x)$, a value x, and the number N of terms in the sequence as arguments, and returns an array with the D_n values for $n = 0, 1, \ldots, N-1$. Make a call to the D function with $f(x) = \sin x$, $x = 0$, and $N = 80$. Plot the evolution of the computed D_n values, using small circles for the data points.

Make another call to `D` where $x = \pi$ and plot this sequence in a separate figure. What would be your expected limit? Why do the computations go wrong for large N? (Hint: Print out the numerator and denominator in D_n.) Name of program file: `sequence_limits3.py`. ◇

Exercise A.4. *Compute π via sequences.*
The following sequences all converge to π:

$$(a_n)_{n=1}^{\infty}, \quad a_n = 4\sum_{k=1}^{n} \frac{(-1)^{k+1}}{2k-1},$$

$$(b_n)_{n=1}^{\infty}, \quad b_n = \left(6\sum_{k=1}^{n} k^{-2}\right)^{1/2},$$

$$(c_n)_{n=1}^{\infty}, \quad c_n = \left(90\sum_{k=1}^{n} k^{-4}\right)^{1/4},$$

$$(d_n)_{n=1}^{\infty}, \quad d_n = \frac{6}{\sqrt{3}}\sum_{k=0}^{n} \frac{(-1)^{k}}{3^k(2k+1)},$$

$$(e_n)_{n=1}^{\infty}, \quad e_n = 16\sum_{k=0}^{n} \frac{(-1)^{k}}{5^{2k+1}(2k+1)} - 4\sum_{k=0}^{n} \frac{(-1)^{k}}{239^{2k+1}(2k+1)}.$$

Make a function for each sequence that returns an array with the elements in the sequence. Plot all the sequences, and find the one that converges fastest toward the limit π. Name of program file: `pi.py`. ◇

Exercise A.5. *Reduce memory usage of difference equations.*
Consider the program `growth_years.py` from Appendix A.1.1. Since x_n depends on x_{n-1} only, we do not need to store all the $N+1$ x_n values. We actually only need to store x_n and its previous value x_{n-1}. Modify the program to use two variables for x_n and not an array. Also avoid the `index_set` list and use an integer counter for n and a `while` instead. (Of course, without the arrays it is not possible to plot the development of x_n, so you have to remove the `plot` call.) Name of program file: `growth_years_efficient.py`. ◇

Exercise A.6. *Compute the development of a loan.*
Solve (A.16)–(A.17) for $n = 1, 2, \ldots, N$ in a Python function. Name of program file: `loan.py`. ◇

Exercise A.7. *Solve a system of difference equations.*
Solve (A.32)–(A.33) by generating the x_n and c_n sequences in a Python function. Let the function return the computed sequences as arrays. Plot the x_n sequence. Name of program file: `fortune_and_inflation1.py`. ◇

Exercise A.8. *Extend the model* (A.32)–(A.33).

In the model (A.32)–(A.33) the new fortune is the old one, plus the interest, minus the consumption. During year n, x_n is normally also reduced with t percent tax on the earnings $x_{n-1} - x_{n-2}$ in year $n-1$. Extend the model with an appropriate tax term, modify the program from Exercise A.7, and plot x_n with tax ($t = 28$) and without tax ($t = 0$). Name of program file: `fortune_and_inflation2.py`. ◇

Exercise A.9. *Experiment with the program from Exer. A.8.*

Suppose you expect to live for N years and can accept that the fortune x_n vanishes after N years. Experiment with the program from Exercise A.8 for how large the initial c_0 can be in this case. Choose some appropriate values for p, q, I, and t. Name of program file: `fortune_and_inflation3.py`. ◇

Exercise A.10. *Change index in a difference equation.*

A mathematically equivalent equation to (A.5) is

$$x_{i+1} = x_i + \frac{p}{100}x_i, \tag{A.48}$$

since the name of the index can be chosen arbitrarily. Suppose someone has made the following program for solving (A.48) by a slight editing of the program `growth1.py`:

```
from scitools.std import *
x0 = 100               # initial amount
p = 5                  # interest rate
N = 4                  # number of years
index_set = range(N+1)
x = zeros(len(index_set))

# Compute solution
x[0] = x0
for i in index_set[1:]:
    x[i+1] = x[i] + (p/100.0)*x[i]
print x
plot(index_set, x, 'ro', xlabel='years', ylabel='amount')
```

This program does not work. Make a correct version, but keep the difference equations in its present form with the indices `i+1` and `i`. Name of program file: `growth1_index_ip1.py`. ◇

Exercise A.11. *Construct time points from dates.*

A certain quantity p (which may be an interest rate) is piecewise constant and undergoes changes at some specific dates, e.g.,

$$p \text{ changes to } \begin{cases} 4.5 & \text{on Jan 4, 2009} \\ 4.75 & \text{on March 21, 2009} \\ 6.0 & \text{on April 1, 2009} \\ 5.0 & \text{on June 30, 2009} \\ 4.5 & \text{on Nov 1, 2009} \\ 2.0 & \text{on April 1, 2010} \end{cases} \tag{A.49}$$

Given a start date d_1 and an end date d_2, fill an array `p` with the right p values, where the array index counts days. Use the `datetime` module to compute the number of days between dates. Name of program file: `dates2days.py`. ◇

Exercise A.12. *Solve nonlinear equations by Newton's method.*

Import the `Newton` function from the `Newton.py` file from Appendix A.1.10 to solve the following nonlinear algebraic equations:

$$\sin x = 0, \tag{A.50}$$

$$x = \sin x, \tag{A.51}$$

$$x^5 = \sin x, \tag{A.52}$$

$$x^4 \sin x = 0, \tag{A.53}$$

$$x^4 = 0, \tag{A.54}$$

$$x^{10} = 0, \tag{A.55}$$

$$\tanh x = x^{10}\,. \tag{A.56}$$

Implement the $f(x)$ and $f'(x)$ functions, required by Newton's method, for each of the nonlinear equations. Collect the names of the $f(x)$ and $f'(x)$ in a list, and make a `for` loop over this list to call the `Newton` function for each equation. Read the starting point x_0 from the command line. Print out the evolution of the roots (based on the `info` list) for each equation. You will need to carefully plot the various $f(x)$ functions to understand how Newton's method will behave in each case for different starting values. Find a starting value x_0 value for each equation so that Newton's method will converge toward the root $x = 0$. Name of program file: `Newton_examples.py`. ◇

Exercise A.13. *Visualize the convergence of Newton's method.*

Let $x_0, x_1, \ldots, x_N$ be the sequence of roots generated by Newton's method applied to a nonlinear algebraic equation $f(x) = 0$ (cf. Appendix A.1.10). In this exercise, the purpose is to plot the sequences $(x_n)_{n=0}^N$ and $(|f(x_n)|)_{n=0}^N$. Make a general function `Newton_plot(f, x, dfdx, epsilon=1E-7)` for this purpose. The arguments `f` and `dfdx` are Python functions representing the $f(x)$ function in the equation and its derivative $f'(x)$, respectively. Newton's method is run until $|f(x_N)| \leq \epsilon$, and the ϵ value is stored in the `epsilon` argument. The `Newton_plot` function should make two separate plots of $(x_n)_{n=0}^N$ and $(|f(x_n)|)_{n=0}^N$ on the screen and also save these plots to PNG files. Because of the potentially wide scale of values that $|f(x_n)|$ may exhibit, it may be wise to use a logarithmic scale on the y axis. (Hint: You can save quite some coding by calling the improved `Newton` function from Appendix A.1.10, which is available in the `Newton` module in `src/diffeq/Newton.py`.)

Demonstrate the function on the equation $x^6 \sin \pi x = 0$, with $\epsilon = 10^{-13}$. Try different starting values for Newton's method: $x_0 =$

$-2.6, -1.2, 1.5, 1.7, 0.6$. Compare the results with the exact solutions $x = \ldots, -2 - 1, 0, 1, 2, \ldots$. Name of program file: `Newton2.py`. ◇

Exercise A.14. *Implement the Secant method.*

Newton's method (A.34) for solving $f(x) = 0$ requires the derivative of the function $f(x)$. Sometimes this is difficult or inconvenient. The derivative can be approximated using the last two approximations to the root, x_{n-2} and x_{n-1}:

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}} \,.$$

Using this approximation in (A.34) leads to the Secant method:

$$x_n = x_{n-1} - \frac{f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}, \quad x_0, x_1 \text{ given} \,. \tag{A.57}$$

Here $n = 2, 3, \ldots$. Make a program that applies the Secant method to solve $x^5 = \sin x$. Name of program file: `Secant.py`. ◇

Exercise A.15. *Test different methods for root finding.*

Make a program for solving $f(x) = 0$ by Newton's method (Appendix A.1.10), the Bisection method (Chapter 4.6.2), and the Secant method (Exercise A.14). For each method, the sequence of root approximations should be written out (nicely formatted) on the screen. Read $f(x)$, $f'(x)$, a, b, x_0, and x_1 from the command line. Newton's method starts with x_0, the Bisection method starts with the interval $[a, b]$, whereas the Secant method starts with x_0 and x_1.

Run the program for each of the equations listed in Exercise A.12. You should first plot the $f(x)$ functions as suggested in that exercise so you know how to choose x_0, x_1, a, and b in each case. Name of program file: `root_finder_examples.py`. ◇

Exercise A.16. *Make difference equations for the Midpoint rule.*

Use the ideas of Appendix A.1.7 to make a similar system of difference equations and corresponding implementation for the Midpoint integration rule from Exercise 3.8. Name of program file: `diffeq_midpoint.py`. ◇

Exercise A.17. *Compute the arc length of a curve.*

Sometimes one wants to measure the length of a curve $y = f(x)$ for $x \in [a, b]$. The arc length from $f(a)$ to some point $f(x)$ is denoted by $s(x)$ and defined through an integral

$$s(x) = \int_a^x \sqrt{1 + [f'(\xi)]^2} d\xi \,. \tag{A.58}$$

We can compute $s(x)$ via difference equations as explained in Appendix A.1.7. Make a Python function `arclength(f, a, b, n)` that returns an array `s` with $s(x)$ values for n uniformly spaced coordinates x

in $[a, b]$. Here `f(x)` is the Python implementation of the function that defines the curve we want to compute the arc length of. How can you test that the `arclength` function works correctly? Test the function on

$$f(x) = \int_{-2}^{x} = \frac{1}{\sqrt{2\pi}} e^{-4t^2} dt, \quad x \in [-2, 2] .$$

Compute $f(x)$ and plot it together with $s(x)$. Name of program file: `arclength.py`. ◇

Exercise A.18. *Find difference equations for computing* $\sin x$*.*

The purpose of this exercise is to derive and implement difference equations for computing a Taylor polynomial approximation to $\sin x$, using the same ideas as in (A.24)–(A.25) for a Taylor polynomial approximation to e^x in Appendix A.1.8.

The Taylor series for $\sin x$ is presented in Exercise 5.21, Equation (5.21) on page 248. To compute $S(x; n)$ efficiently, we try to compute a new term from the last computed term. Let $S(x; n) = \sum_{j=0}^{n} a_j$, where the expression for a term a_j follows from the formula (5.21). Derive the following relation between two consecutive terms in the series,

$$a_j = -\frac{x^2}{(2j + 1)2j} a_{j-1} . \tag{A.59}$$

Introduce $s_j = S(x; j - 1)$ and define $s_0 = 0$. We use s_j to accumulate terms in the sum. For the first term we have $a_0 = x$. Formulate a system of two difference equations for s_j and a_j in the spirit of (A.24)–(A.25). Implement this system in a function `sine_Taylor(x, n)`, which returns s_{n+1} and $|a_{n+1}|$. The latter is the first neglected term in the sum (since $s_{n+1} = \sum_{j=0}^{n} a_j$) and may act as a rough measure of the size of the error in the approximation.

Verify the implementation by computing the difference equations for $n = 2$ by hand (or in a separate program) and comparing with the output from the `sine_Taylor` function. Also make a table of s_n for various x and n values to verify that the accuracy of a Taylor polynomial improves as n increases and x decreases. Be aware of the fact that `sine_Taylor(x, n)` can give extremely inaccurate approximations to $\sin x$ if x is not sufficiently small and n sufficiently large. Name of program file: `sin_Taylor_series_diffeq.py`. ◇

Exercise A.19. *Find difference equations for computing* $\cos x$*.*

Carry out the steps in Exercise A.18, but do it for the Taylor series of $\cos x$ instead of $\sin x$ (look up the Taylor series for $\cos x$ in a mathematics textbook or search on the Internet). Name of program file: `cos_Taylor_series_diffeq.py`. ◇

Exercise A.20. *Make a guitar-like sound.*

Given start values $x_0, x_1, \ldots, x_p$, the following difference equation is known to create guitar-like sound:

$$x_n = \frac{1}{2}(x_{n-p} + x_{n-p-1}), \quad n = p+1, \ldots, N\,. \tag{A.60}$$

With a sampling rate r, the frequency of this sound is given by r/p. Make a program with a function `solve(x, p)` which returns the solution array `x` of (A.60). To initialize the array `x[0:p+1]` we look at two methods, which can be implemented in two alternative functions:

1. $x_0 = 1$, $x_1 = x_2 = \cdots = x_p = 0$
2. $x_0, \ldots, x_p$ are uniformly distributed random numbers in $[-1, 1]$

Import `max_amplitude`, `write`, and `play` from the `scitools.sound` module. Choose a sampling rate r and set $p = r/440$ to create a 440 Hz tone (A). Create an array `x1` of zeros with length $3r$ such that the tone will last for 3 seconds. Initialize `x1` according to method 1 above and solve (A.60). Multiply the `x1` array by `max_amplitude`. Repeat this process for an array `x2` of length $2r$, but use method 2 for the initial values and choose p such that the tone is 392 Hz (G). Concatenate `x1` and `x2`, call `write` and then `play` to play the sound. As you will experience, this sound is amazingly similar to the sound of a guitar string, first playing A for 3 seconds and then playing G for 2 seconds. (The method (A.60) is called the Karplus-Strong algorithm and was discovered in 1979 by a researcher, Kevin Karplus, and his student Alexander Strong, at Stanford University.) Name of program file: `guitar_sound.py`. ◇

Exercise A.21. *Damp the bass in a sound file.*

Given a sequence $x_0, \ldots, x_{N-1}$, the following *filter* transforms the sequence to a new sequence $y_0, \ldots, y_{N-1}$:

$$y_n = \begin{cases} x_n, & n = 0 \\ -\frac{1}{4}(x_{n-1} - 2x_n + x_{n+1}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.61}$$

If x_n represents sound, y_n is the same sound but with the bass damped. Load some sound file (e.g., the one from Exercise A.20) or call

```
x = scitools.sound.Nothing_Else_Matters(echo=True)
```

to get a sound sequence. Apply the filter (A.61) and play the resulting sound. Plot the first 300 values in the x_n and y_n signals to see graphically what the filter does with the signal. Name of program file: `damp_bass.py`. ◇

Exercise A.22. *Damp the treble in a sound file.*

Solve Exercise A.21 to get some experience with coding a filter and trying it out on a sound. The purpose of this exercise is to explore some other filters that reduce the treble instead of the bass. Smoothing the sound signal will in general damp the treble, and smoothing is typically obtained by letting the values in the new filtered sound sequence be an average of the neighboring values in the original sequence.

The simplest smoothing filter can apply a standard average of three neighboring values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{3}(x_{n-1} + x_n + x_{n+1}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.62}$$

Two other filters put less emphasis on the surrounding values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{4}(x_{n-1} + 2x_n + x_{n+1}), & 1 \le n \le N-2 \\ x_n, & n = N-1 \end{cases} \tag{A.63}$$

$$y_n = \begin{cases} x_n, & n = 0, 1 \\ \frac{1}{16}(x_{n-2} + 4x_{n-1} + 6x_n + 4x_{n+1} + x_{n+2}), & 2 \le n \le N-3 \\ x_n, & n = N-2, N-1 \end{cases} \tag{A.64}$$

Apply all these three filters to a sound file and listen to the result. Plot the first 300 values in the x_n and y_n signals for each of the three filters to see graphically what the filter does with the signal. Name of program file: `damp_treble.py`. ◇

Exercise A.23. *Demonstrate oscillatory solutions of* (A.13).

Modify the `growth_logistic.py` program from Appendix A.1.5 to solve the equation (A.13) on page 568. Read the input parameters y_0, q, and N from the command line.

Equation (A.13) has the solution $y_n = 1$ as $n \to \infty$. Demonstrate, by running the program, that this is the case when $y_0 = 0.3$, $q = 1$, and $N = 50$.

For larger q values, y_n does not approach a constant limit, but y_n oscillates instead around the limiting value. Such oscillations are sometimes observed in wildlife populations. Demonstrate oscillatory solutions when q is changed to 2 and 3.

It could happen that y_n stabilizes at a constant level for larger N. Demonstrate that this is not the case by running the program with $N = 1000$. Name of program file: `growth_logistic2.py`. ◇

Exercise A.24. *Improve the program from Exer. A.23.*

It is tedious to run a program like the one from Exercise A.23 repeatedly for a wide range of input parameters. A better approach is to let the computer do the manual work. Modify the program from Exercise A.23 such that the computation of y_n and the plot is made in a function. Let the title in the plot contain the parameters y_0 and q

(N is easily visible from the x axis). Also let the name of the plot file reflect the values of y_0, q, and N. Then make loops over y_0 and q to perform the following more comprehensive set of experiments:

- $y_0 = 0.01, 0.3$
- $q = 0.1, 1, 1.5, 1.8, 2, 2.5, 3$
- $N = 50$

How does the initial condition (the value y_0) seem to influence the solution?

The keyword argument `show=False` can be used in the `plot` call if you do not want all the plot windows to appear on the screen. Name of program file: `growth_logistic3.py`. ◇

Exercise A.25. *Generate an HTML report.*

Extend the program made in Exercise A.24 with a report containing all the plots. The report can be written in HTML and displayed by a web browser. The plots must then be generated in PNG format. The source of the HTML file will typically look as follows:

```
<html>
<body>
<p><img src="tmp_y0_0.01_q_0.1_N_50.png">
<p><img src="tmp_y0_0.01_q_1_N_50.png">
<p><img src="tmp_y0_0.01_q_1.5_N_50.png">
<p><img src="tmp_y0_0.01_q_1.8_N_50.png">
...
<p><img src="tmp_y0_0.01_q_3_N_1000.png">
</html>
</body>
```

Let the program write out the HTML text, either to the screen or to a file (cf. Chapter 6.5). When writing to the screen, redirect the output to a file,

Terminal
```
growth_logistic4.py > report.html
```

The file `report.html` can be loaded into a web browser.

You may let the function making the plots return the name of the plotfile such that this string can be inserted in the HTML file. Name of program file: `growth_logistic4.py`. ◇

Exercise A.26. *Simulate the price of wheat.*

The demand for wheat in year t is given by

$$D_t = ap_t + b,$$

where $a < 0$, > 0, and p_t is the price of wheat. Let the supply of wheat be

$$S_t = Ap_{t-1} + B + \ln(1 + p_{t-1}),$$

where A and B are given constants. We assume that the price p_t adjusts such that all the produced wheat is sold. That is, $D_t = S_t$.

For $A = 1$, $a = -3, b = 5, B = 0$, find from numerical computations, a stable price such that the production of wheat from year to year is constant. That is, find p such that $ap + b = Ap + B + \ln(1 + p)$.

Assume that in a very dry year the production of wheat is much less than planned. Given that price this year, p_0, is 4.5 and $D_t = S_t$, compute in a program how the prices $p_1, p_2, \ldots, p_N$ develop. This implies solving the difference equation

$$ap_t + b = Ap_{t-1} + B + \ln(1 + p_{t-1})\,.$$

From the p_t values, compute S_t and plot the points (p_t, S_t) for $t = 0, 1, 2, \ldots, N$. How do the prices move when $N \to \infty$? Name of program file: `wheat.py`. ◇

solvers. This feature makes it easy to switch between solvers to test a wide collection of numerical methods for a problem.

Odespy can be downloaded from . It is installed by the usual `python setup.py install` command.

E.3 Exercises

Exercise E.1. *Solve a simple ODE in two ways.*

The purpose of this exercise is to solve the ODE problem $u - 10u' = 0$, $u(0) = 0.2$, for $t \in [0, 20]$. Use both the `ForwardEuler` *function* from Appendix E.1.3 and the `ForwardEuler` *class* from Appendix E.1.7. Set $\Delta t = 1$. Check that the results produced by the two equivalent methods coincide. Name of program file: `simple_ODE.py`. ◇

Exercise E.2. *Use the ODESolver hierarchy to solve a simple ODE.*

Solve the ODE problem $u' = u/2$ with $u(0) = 1$, using a class in the `ODESolver` hierarchy. Choose $\Delta t = 0.5$ and find $u(t)$ for $t \in [0, 6]$. Write out the approximate final u_n value together with the exact value e^3. Repeat the calculations for $\Delta t = 0.001$. Name of program file: `ODESolver_demo.py`. ◇

Exercise E.3. *Solve an ODE for emptying a tank.*

A cylindrical tank of radius R is filled with water to a height h_0. By opening a valve of radius r at the bottom of the tank, water flows out, and the height of water, $h(t)$, 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]. Note that $\Delta h > 0$ implies an increase in h, which means that $-\pi R^2 \Delta h$ is the corresponding decrease in volume that must balance the outflow loss of volume $\pi r^2 v \Delta t$. Elimination of v and taking the limit $\Delta t \to 0$ lead to the ODE

$$\frac{dh}{dt} = -\left(\frac{r}{R}\right)^2 \left(1 - \left(\frac{r}{R}\right)^4\right)^{-1/2} \sqrt{2gh}\,.$$

For practical applications $r \ll R$ so that $1 - (r/R)^4 \approx 1$ is a reasonable approximation (friction is neglected in the derivation, and we are also going to solve the ODE by approximate methods). The final ODE then becomes

$$\frac{dh}{dt} = -\left(\frac{r}{R}\right)^2 \sqrt{2gh}\,. \tag{E.60}$$

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

Solve (E.60) by a numerical method of your choice in a program. Set $r = 1$ cm, $R = 20$ cm, $g = 9.81$ m/s^2, 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. Make sure to test for $h < 0$ so that you do not apply the square root function to negative numbers. Can you find an analytical solution of the problem to compare the numerical solution with? Name of program file: `tank_ODE.py`. ⋄

Exercise E.4. *Scale the logistic equation.*

Consider the logistic model (E.5):

$$u'(t) = \alpha u(t)\left(1 - \frac{u(t)}{R}\right), \quad u(0) = U_0\,.$$

This problem involves three input parameters: U_0, R, and α. Learning how u varies with U_0, R, and α requires much experimentation where we vary all three parameters and observe the solution. A much more effective approach is to *scale* the problem. By this technique the solution depends only on one parameter U_0/R in the present problem. This exercise tells how the scaling is done.

The idea of scaling is to introduce *dimensionless* versions of the independent and dependent variables:

$$v = \frac{u}{u_c}, \quad \tau = \frac{t}{t_c},$$

where u_c and t_c are characteristic sizes of u and t, respectively, such that the dimensionless variables v and τ are of approximately unit size. Since we know that $u \to R$ as $t \to \infty$, R can be taken as the characteristic size of u.

Insert $u = Rv$ and $t = t_c\tau$ in the governing ODE and choose $t_c = 1/\alpha$. Show that the ODE for the new function $v(\tau)$ becomes

$$\frac{dv}{d\tau} = v(1 - v), \quad v(0) = v_0\,. \tag{E.61}$$

We see that the three parameters U_0, R, and α have disappeared from the ODE problem, and only one parameter $v_0 = U_0/R$ is involved.

Show that if $v(\tau)$ is computed, one can recover $u(t)$ by

$$u(t) = Rv(\alpha t)\,. \tag{E.62}$$

Geometrically, the transformation from v to u is just a stretching of the two axis in the coordinate system.

Make a program `logistic_scaled.py` where you compute $v(\tau)$, given $v_0 = 0.05$, and then you use (E.62) to plot $u(t)$ for $R = 100, 500, 1000$ and $\alpha = 1$ in one figure, and $u(t)$ for $\alpha = 1, 5, 10$ and $R = 1000$ in another figure. Note how effectively you can generate $u(t)$ without needing to solve an ODE problem, and also note how varying R and α impacts the graph of $u(t)$. Name of program file: `logistic_scaled.py`. ◇

Exercise E.5. *Compute logistic growth with time-varying carrying capacity.*

Use classes `Problem2` and `AutoSolver` from Appendix E.2.8 to study logistic growth when the carrying capacity of the environment, R, changes periodically with time: $R = 500$ for $it_s \leq t < (i+1)t_s$ and $R = 200$ for $(i+1)t_s \leq t < (i+2)t_s$, with $i = 0, 2, 4, 6, \ldots$. Use the same data as in Appendix E.2.8, and find some relevant sizes of the period of variation, t_s, to experiment with. Name of program file: `seasonal_logistic_growth.py`. ◇

Exercise E.6. *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 solves the problem

$$\frac{ds}{dx} = \sqrt{1 + [f'(x)]^2}, \quad s(x_0) = 0\,. \tag{E.63}$$

Since s does not enter the right-hand side, (E.63) can immediately be integrated from x_0 to x (see Exercise A.17). However, we shall solve (E.63) as an ODE. Use the Forward Euler method and compute the length of a straight line (for easy verification) and a parabola: $f(x) = \frac{1}{2}x + 1$, $x \in [0, 2]$; $f(x) = x^2$, $x \in [0, 2]$. Name of program file: `arclength_ODE.py`. ◇

Exercise E.7. *Compute inverse functions by solving an ODE.*

The inverse function g of some function $f(x)$ takes the value $f(x)$ back to x again: $g(f(x)) = x$. The common technique to compute inverse functions is to set $y = f(x)$ and solve with respect to x. The formula on the right-hand side is then the desired inverse function $g(y)$. Exercise 7.26 makes use of such an approach, where $y - f(x) = 0$ is solved numerically with respect to x for different discrete values of y.

We can formulate a general procedure for computing inverse functions from an ODE problem. If we differentiate $y = f(x)$ with respect to y, we get $1 = f'(x)\frac{dx}{dy}$ by the chain rule. The inverse function we seek is $x(y)$, but this function then fulfills the ODE

$$x'(y) = \frac{1}{f'(x)}\,. \tag{E.64}$$

That y is the independent coordinate and x the function of y can be a somewhat confusing notation, so you might introduce u and t for x

and y:

$$u'(t) = \frac{1}{f'(u)}\,.$$

The initial condition is $x(0) = x_r$ where x_r solves the equation $f(x_r) = 0$ ($x(0)$ implies $y = 0$ and then from $y = f(x)$ it follows that $f(x(0)) = 0$).

Make a program that can use the described method to compute $x(y)$, given $f(x)$ and x_r. Use any numerical method of your choice for solving the ODE problem. Plot $f(x)$ as well as $x(y)$. Verify the implementation for $f(x) = 2x$. Test the method for $f(x) = \sqrt{x}$. Name of program file: `inverse_ODE.py`. ◇

Exercise E.8. *Generalize the implementation in Exer. E.7.*

The method for computing inverse functions described in Exercise E.7 is very general. The purpose now is to make a reusable utility, here called `Inverse`, for computing the inverse of some Python function `f(x)` on some interval `I=[a,b]`. The utility can be used as follows to calculate the inverse of $\sin x$ on $I = [0, \pi/2]$:

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

# Compute the inverse of f
inverse = Inverse(f, x0=0, I=[0, pi/2], resolution=100)
x, y = Inverse.compute()
from scitools.std import plot

plot(y, x, 'r-',
     x, f(x), 'b-',
     y, asin(y), 'go')
legend(['computed inverse', 'f(x)', 'exact inverse'])
```

Here, `x0` is the value of `x` at 0, or in general at the left point of the interval: `I[0]`. The parameter `resolution` tells how many equally sized intervals Δy we use in the numerical integration of the ODE. A default choice of 1000 can be used if it is not given by the user.

Make class `Inverse` and put it in a module. Include a function `_verify()` in the module which tests that class `Inverse` gives exact solution for the test problem $f(x) = 2x$. Name of program file: `Inverse1.py`. ◇

Exercise E.9. *Extend the implementation in Exer. E.8.*

Extend the module in Exercise E.8 such that the value of $x(0)$ (`x0` in class `Inverse`'s constructor) does not need to be provided by the user. To this end, class `Inverse` must solve $f(x) = 0$ and set `x0` equal to the root. You may use the Bisection method from Chapter 4.6.2, Newton's method from Appendix A.1.10, or the Secant method from Exercise A.14 to solve $f(x) = 0$. Class `Inverse` should figure out a suitable initial interval for the Bisection method or start values for the Newton or Secant methods. Computing $f(x)$ for x at many points and examining these may help in solving $f(x) = 0$ without any input from the user. Name of program file: `Inverse2.py`. ◇

Exercise E.10. *Compute inverse functions by interpolation.*

Instead of solving an ODE for computing the inverse function $g(y)$ of some function $f(x)$, as explained in Exercise E.7, one may use a simpler approach based on ideas from Appendix E.1.5. Say we compute discrete values of x and $f(x)$, stored in the arrays `x` and `y`. Doing a `plot(x, y)` shows $y = f(x)$ as a function of x, and doing `plot(y, x)` shows x as a function of y, i.e., we plot the inverse function $g(y)$. It is therefore trivial to plot the inverse function!

However, if we want the inverse function of $f(x)$ as some Python function `g(y)` that we can call for any `y`, we can use the tool `wrap2callable` from Appendix E.1.5 to turn the discrete inverse function, described by the arrays `y` (independent coordinate) and `x` (dependent coordinate), into a continuous function `g(y)`:

```
from scitools.std import wrap2callable
g = wrap2callable((y, x))

y = 0.5
print g(y)
```

The `g(y)` function applies linear interpolation in each interval between the points in the `y` array.

Implement this method in a program. Verify the implementation for $f(x) = 2x$ for $x \in [0, 4]$ and test it for $f(x) = \sin x$ for $x \in [0, \pi/2]$. Name of program file: `inverse_wrap2callable.py`. ◇

Exercise E.11. *Simulate a falling or rising body in a fluid.*

A body moving vertically through a fluid (liquid or gas) is subject to three different types of forces:

1. the gravity force $F_g = -mg$, where m is the mass of the body and g is the acceleration of gravity;
2. the drag force[2] $F_d = -\frac{1}{2}C_D\varrho A|v|v$ (see also Exercise 1.11), where C_D is a dimensionless drag coefficient depending on the body's shape, ϱ is the density of the fluid, A is the cross-sectional area (produced by a cutting plane $y =$ const through the thickest part of the body), and v is the velocity;
3. the uplift or buoyancy force ("Archimedes force") $F_b = \varrho gV$, where V is the volume of the body.

Newton's second law applied to the body says that the sum of these forces must equal the mass of the body times its acceleration a:

$$F_g + F_d + F_b = ma,$$

[2] Roughly speaking, the F_d formula is suitable for medium to high velocities, while for very small velocities, or very small bodies, F_d is proportional to the velocity, not the velocity squared, see [11].

which gives

$$-mg - \frac{1}{2}C_D\varrho A|v|v + \varrho gV = ma\,.$$

The unknowns here are v and a, i.e., we have two unknowns but only one equation. From kinematics in physics we know that the acceleration is the time derivative of the velocity: $a = dv/dt$. This is our second equation. We can easily eliminate a and get a single differential equation for v:

$$-mg - \frac{1}{2}C_D\varrho A|v|v + \varrho gV = m\frac{dv}{dt}\,.$$

A small rewrite of this equation is handy: We express m as $\varrho_b V$, where ϱ_b is the density of the body, and we isolate dv/dt on the left-hand side,

$$\frac{dv}{dt} = -g\left(1 - \frac{\varrho}{\varrho_b}\right) - \frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v\,. \tag{E.65}$$

This differential equation must be accompanied by an initial condition: $v(0) = V_0$.

Make a program for solving (E.65) numerically, using any numerical method of your choice. Implement the right-hand side of (E.65) in the `__call__` method of a class where the parameters g, ϱ, ϱ_b, C_D, A, and V are attributes.

To verify the program, assume a heavy body in air such that the F_b force can be neglected, and assume a small velocity such that the air resistance F_d can also be neglected. Setting $\varrho = 0$ removes both these terms from the equation. The motion then leads to the exact velocity $v(t) = y'(t) = v_0 - gt$. See how well the program reproduces this simple solution.

After the program is verified, we are ready to run two real examples and plot the evolution of v:

1. *Parachute jumper.* The point is to compute the motion of a parachute jumper in free fall before the parachute opens. We set the density of the human body as $\varrho_b = 1003$ kg/m^3 and the mass as $m = 80$ kg, implying $V = m/\varrho_b = 0.08$ m^3. We can base the cross-sectional area A on the height 1.8 m and a width of 50 cm, giving giving $A \approx \pi R^2 = 0.9$ m^2. The density of air decreases with height, and we here use the value 0.79 kg/m^3 which is relevant for about 5000 m height. The C_D coefficient can be set as 0.6. Start with $v_0 = 0$.
2. *Rising ball in water.* A ball with the size of a soccer ball is placed in deep water, and we seek to model its motion upwards. Contrary to the former example, where the buoyancy force F_b is very small, F_b is now the driving force, and the gravity force F_g is small. Set $A = \pi a^2$ with $a = 11$ cm, the mass of the ball is 0.43 kg, the density

of water is 1000 kg/m^3, and C_D is 0.2. Start with $v_0 = 0$ and see how the ball rises.

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

Exercise E.12. *Check the solution's limit in Exer. E.11.*

The solution of (E.65) often tends to a constant velocity, called the terminal velocity. This happens when the sum of the forces, i.e., the right-hand side in (E.65) vanishes. Compute the formula for the terminal velocity by hand. Solve the ODE using class `ODESolver` and call the `solve` method with a `terminate` function that terminates the computations when a constant velocity is reached, that is, when $|v(t_n) - v(t_{n-1})| \leq \epsilon$, where ϵ is a small number. Run a series of Δt values and make a graph of the terminal velocity as a function of Δt for the two cases in Exercise E.11. Indicate the exact terminal velocity in the plot by a horizontal line. Would you expect the accuracy of the computed terminal velocity to increase with decreasing Δt? Discuss! Name of program file: `body_in_fluid_termvel.py`. ◇

Exercise E.13. *Visualize the different forces in Exer. E.11.*

The purpose of this exercise is to plot the forces F_g, F_b, and F_d in the model from Exercise E.11 as functions of t. Seeing the relative importance of the forces as time develops gives an increased understanding of how the different forces contribute to change the velocity. Name of program file: `body_in_fluid_forces.py`. ◇

Exercise E.14. *Solve an ODE until constant solution.*

Newton's law of cooling,

$$\frac{dT}{dt} = -h(T - T_s) \tag{E.66}$$

can be used to see how the temperature T of an object changes because of heat exchange with the surroundings, which have a temperature T_s. The parameter h, with unit s^{-1} is an experimental constant (heat transfer coefficient) telling how efficient the heat exchange with the surroundings is. For example, (E.66) may model the cooling of a hot pizza taken out of the oven. The problem with applying (E.66), nevertheless, is that h must be measured. Suppose we have measured T at $t = 0$ and t_1. We can use a rough Forward Euler approximation of (E.66) with one time step of length t_1,

$$\frac{T(t_1) - T(0)}{t_1} = -h(T(0) - T_s),$$

to make the estimate

$$h = \frac{T(t_1) - T(0)}{t_1(T_s - T(0))} \,. \tag{E.67}$$

Suppose now you take a hot pizza out of the oven. The temperature of the pizza is 200 C at $t = 0$ and 180 C after 50 seconds, in a room with temperature 20 C. Find an estimate of h from the formula above.

Solve (E.66) to find the evolution of the temperature of the pizza. Use class `ForwardEuler` or `RungeKutta4`, and supply a `terminate` function to the `solve` method so that the simulation stops when T is sufficiently close to the final room temperature T_s. Plot the solution. Name of program file: `pizza_cooling1.py`. ◇

Exercise E.15. *Use classes in Exer. E.14.*

Solve Exercise E.14 with a class `Problem` containing the parameters h, T_s, $T(0)$, and Δt as attributes. A method `estimate_h` should take t_1 and $T(t_1)$ as arguments, compute h, and assign it to `self.h`. Also a method `__call__` for computing the right-hand side must be included. The `terminate` function can be a method in the class, a stand-alone function, or a lambda function. By using class `Problem`, we avoid having the physical parameters as global variables in the program and all the problem-specific data are packed into one object.

Write a function `solve(problem)` that takes a `Problem` object with name `problem` as argument, solves the ODE, and plots the solution. We now want to run experiments with different values of some parameters: $T_s = 15, 22, 30$ C and $T(0) = 250, 200$ C. Make a list of `Problem` objects and plot the solution for each problem in the same figure:

```
# Given h and dt
problems = [Problem(h, T_s, T_0, dt) \
            for T_s in 15, 22, 30 for T_0 in 250, 200]
for problem in problems:
    solve(problem)
    hold('on')
```

Name of program file: `pizza_cooling2.py`. ◇

Exercise E.16. *Scale away parameters in Exer. E.14.*

Use the scaling approach from Appendix E.2.8 to "scale away" the parameters in the ODE in Exercise E.14. That is, introduce a new unknown $u = (T - T_s)/(T(0) - T_s)$ and a new time scale $\tau = th$. Find the ODE and the initial condition that governs the $u(\tau)$ function. Make a program that computes $u(\tau)$ until $|u| < 0.001$. Store the discrete u and τ values in a file `u_tau.dat` if that file is not already present (you can use `os.path.isfile(f)` to test if a file with name `f` exists). Create a function `T(u, tau, h, T0, Ts)` that loads the u and τ data from the `u_tau.dat` file and returns two arrays with T and t values, corresponding to the computed arrays for u and τ. Plot T versus t. Give the parameters h, T_s, and $T(0)$ on the command line. Note that this program is supposed to solve the ODE once and then recover any $T(t)$ solution by a simple scaling of the single $u(\tau)$ solution. Name of program file: `pizza_cooling3.py`. ◇

Exercise E.17. *Use the 4th-order Runge-Kutta on* (C.34).

Investigate if the 4th-order Runge-Kutta method is better than the Forward Euler scheme for solving the challenging ODE problem (C.34) from Exercise C.3 on page 641. Name of program file: `yx_ODE2.py`. ◇

Exercise E.18. *Compare ODE methods.*

The equation $u' = -au$ is a relevant model for radioactive decay, where $u(t)$ is the fraction of particles that remains in the radioactive substance at time t. The parameter a is the inverse of the so-called mean lifetime of the substance. The initial condition is $u(0) = 1$.

Introduce a class `Decay` to hold information about the physical problem: the parameter a and a `__call__` method for computing the right-hand side $-au$ of the ODE. Initialize an instance of class `Decay` with $a = \ln(2)/5600$ 1/y (y means the unit years, and this value of a corresponds to the Carbon-14 radioactive isotope whose decay is used extensively in dating organic material that is tens of thousands of years old).

Solve (E.7) by both the Forward Euler and the 4-th order Runge-Kutta method, using the `ForwardEuler` and the `RungeKutta4` classes in the `ODESolver` hierarchy. Use a time step of 500 years, and simulate decay for $T = 20,000$ y (let the time unit be 1 y). Plot the two solutions. Write out the final $u(T)$ value and compare it with the exact value e^{-aT}. Name of program file: `radioactive_decay.py`. ◇

Exercise E.19. *Compare ODE methods.*

Consider the problem described in Exercise E.3 on page 707. Make a class `Problem` that holds problem-specific parameters: h_0, r, R, Δt, and T ($[0, T]$ being the time interval for simulation). Our aim is to solve this ODE problem using the `ForwardEuler`, `BackwardEuler`, and `RungeKutta4` classes in the `ODESolver` hierarchy. Read one or more Δt values from the command line, solve the problems for these Δt values, and plot the graphs in the same figure:

```
# Given h_0, r, R
problems = [Problem(h_0, r, R, float(dt)) for dt in sys.argv[1:]]
for problem in problems:
    for method in ForwardEuler, BackwardEuler, RungeKutta4:
        solve(problem, method)
        hold('on')
```

The `solve` function must use the `method` class and information in the `Problem` object to solve the ODE and make a curve plot of $h(t)$ (annotating the legend with the value of `dt` and `method`).

Try out different Δt values between 5 and 50 s. Comment upon the quality of the various methods to compute a correct limiting value of h as Δt is varied. (Hint: negative h values may appear when the problem is solved numerically, so set $h = 0$ if $h < 0$ before computing $\sqrt{h}$.) Name of program file: `tank_ODE_3methods.py`. ◇

Exercise E.20. *Solve two coupled ODEs for radioactive decay.*

Consider two radioactive substances A and B. The nuclei in substance A decay to form nuclei of type B with a mean lifetime τ_A, while substance B decay to form type A nuclei with a mean lifetime τ_B. Letting u_A and u_B be the fractions of the initial amount of material in substance A and B, respectively, the following system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$u_A' = u_B/\tau_B - u_A/\tau_A, \tag{E.68}$$
$$u_B' = u_A/\tau_A - u_B/\tau_B, \tag{E.69}$$

with $u_A(0) = u_B(0) = 1$. As in Exercise E.18, introduce a problem class, which holds the parameters τ_A and τ_B and offers a `__call__` method to compute the right-hand side vector of the ODE system, i.e., $(u_B/\tau_B - u_A/\tau_A, u_A/\tau_A - u_B/\tau_B)$. Solve for u_A and u_B using a subclass in the `ODESolver` hierarchy and the parameter choice $\tau_A = 8$ minutes, $\tau_B = 40$ minutes, and $\Delta t = 10$ seconds. Plot u_A and u_B against time measured in minutes. From the ODE system it follows that the ratio $u_A/u_B \rightarrow \tau_A/\tau_B$ as $t \rightarrow \infty$ (assuming $u_A' = u_B' = 0$ in the limit $t \rightarrow \infty$). Check that the solutions fulfills this requirement (this is a partial verification of the program). Name of program file: `radioactive_decay2.py`. ◇

Exercise E.21. *Code a 2nd-order Runge-Kutta method; function.*

Implement the 2nd-order Runge-Kutta method specified in formula (E.37). Use a plain function `RungeKutta2` of the type shown in Appendix E.1.2 for the Forward Euler method. Construct a test problem where you know the analytical solution, and plot the difference between the numerical and analytical solution. Name of program file: `RungeKutta2_func.py`. ◇

Exercise E.22. *Code a 2nd-order Runge-Kutta method; class.*

Make a new subclass `RungeKutta2` in the `ODESolver` hierarchy from Appendix E.2.5 for solving ordinary differential equations with the 2nd-order Runge-Kutta method specified in formula (E.37). Construct a test problem where you know the analytical solution, and plot the difference between the numerical and analytical solution as a function of time. Place the `RungeKutta2` class and the test problem in a separate module (where the superclass `ODESolver` is imported from the `ODESolver` module). Call the test problem from the test block in the module file. Name of program file: `RungeKutta2.py`. ◇

Exercise E.23. *Make a subclass for Heun's method.*

Implement the numerical method (E.35)–(E.36) in a subclass of `ODESolver`. Place the code in a separate file where the `ODESolver` class is imported. How can you verify that the implementation is correct? Name of program file: `Heun.py`. ◇

Exercise E.24. *Make a subclass for the Midpoint method.*

Implement the Midpoint method specified in formula (E.34) from page 686 in a subclass of `ODESolver`.

Compare in a plot the Midpoint method with the Forward Euler and 4th-order Runge-Kutta methods and the exact solution for the problem $u' = u$, $u(0) = 1$, with 10 steps between 0 and the end time $T = 5$. Name of program file: `Midpoint.py`. ◇

Exercise E.25. *Make a subclass for an Adams-Bashforth method.*

Implement the Adams-Bashforth method (E.45) on page 687 in a subclass of `ODESolver`. Use Heun's method (E.36) to compute u_1.

Compare in a plot the Adams-Bashforth method with the Forward Euler and 4th-order Runge-Kutta methods and the exact solution for the problem $u' = u$, $u(0) = 1$, with 10 steps between 0 and the end time $T = 5$. Name of program file: `AdamsBashforth3.py`. ◇

Exercise E.26. *Implement the iterated Midpoint method; function.*

Implement the numerical method (E.46)–(E.47) as a function

```
iterated_Midpoint_method(f, U0, T, n, N)
```

where `f` is a Python implementation of $f(u,t)$, `U0` is the initial condition $u(0) = U_0$, `T` is the final time of the simulation, `n` is the number of time steps, and `N` is the parameter N in the method (E.46). The `iterated_Midpoint_method` should return two arrays: $u_0, \ldots, u_n$ and $t_0, \ldots, t_n$. To verify the implementation, calculate by hand u_1 and u_2 when $N = 2$ for the ODE $u' = -2u$, $u(0) = 1$, with $\Delta t = 1/4$. Compare your hand calculations with the results of the program. Thereafter, run the program for the same ODE problem but with $\Delta t = 0.1$ and $T = 2$. Name of program file: `MidpointIter_func.py`. ◇

Exercise E.27. *Implement the iterated Midpoint method; class.*

The purpose of this exercise is to implement the numerical method (E.46)–(E.47) in a class like the `ForwardEuler` class from Appendix E.1.7. Create a module containing the class and a test function demonstrating the use:

```
def _application():
    def f(u, t):
        return -2*u

    solver = MidpointIter(f, N=4)
    solver.set_initial_condition(1)
    t_points = numpy.linspace(0, 1.5, 16)
    u, t = solver.solve(t_points)
    from scitools.std import plot
    plot(t, u)
```

Call the `_application` function from the test block in the module file. Also include a `_verify` function which compares the hand calculations of two time steps (see Exercise E.26) with the results produced by the class. Name of program file: `MidpointIter_class.py`. ◇

Exercise E.28. *Make a subclass for the iterated Midpoint method.*

Implement the numerical method (E.46)–(E.47) in a subclass of `ODESolver`. The code should reside in a separate file where the `ODESolver` class is imported. One can either fix N or introduce an ϵ and iterate until the change in $|v_q - v_{q-1}|$ is less than ϵ. Allow the constructor to take both N and ϵ as arguments. Compute a new v_q as long as $q \leq N$ or $|v_q - v_{q-1}| > \epsilon$. Let $N = 20$ and $\epsilon = 10^{-6}$ by default. Name of program file: `MidpointIter.py`. ◇

Exercise E.29. *Study convergence of numerical methods for ODEs.*

The approximation error when solving an ODE numerically is usually of the form $C\Delta t^r$, where C and r are constants that can be estimated from numerical experiments. The constant r, called the *convergence rate*, is of particular interest. Halving Δt halves the error if $r = 1$, but if $r = 3$, halving Δt reduces the error by a factor of 8.

Exercise 9.16 describes a method for estimating r from two consecutive experiments. Make a function

```
ODE_convergence(f, U0, u_e, method, dt=[])
```

that returns a series of estimated r values corresponding to a series of Δt values given as the `dt` list. The argument `f` is a Python implementation of $f(u,t)$ in the ODE $u' = f(u,t)$. The initial condition is $u(0) = U_0$, where U_0 is given as the `U0` argument, `u_e` is the exact solution $u_e(t)$ of the ODE, and `method` is the name of a class in the `ODESolver` hierarchy. The error between the exact solution u_e and the computed solution $u_0, u_1, \ldots, u_n$ can be defined as

$$e = \left(\Delta t \sum_{i=0}^{n} (u_e(t_i) - u_i)^2\right)^{1/2} .$$

Call the `ODE_convergence` function for some methods you have in the `ODESolver` hierarchy and answers to exercises, and print the estimated r values for each method. Use an ODE problem of your own choice. Name of program file: `ODE_convergence.py`. ◇

Exercise E.30. *Solve an ODE specified on the command line.*

To solve an ODE, we want to make a program `cmlodesolver.py` which accepts an ODE problem to be specified on the command line. The command-line arguments are `f u0 dt T`, where `f` is the right-hand side $f(u,t)$ specified as a string formula (to be converted to a `StringFunction` object), `u0` is the initial condition, `dt` is the time step, and `T` is the final time of the simulation. A fifth optional argument can be given to specify the class name of the numerical solution method (set any method of choice as default value). A curve plot of the solution versus time should be produced and stored in a file `plot.png`. Name of program file: `cmlodesolver.py`. ◇

Exercise E.31. *Find the body's position in Exer. E.11.*

In Exercise E.11 we compute the velocity $v(t)$. The position of the body, $y(t)$, is related to the velocity by $y'(t) = v(t)$. Extend the program from Exercise E.11 to solve the system

$$\frac{dy}{dt} = v,$$
$$\frac{dv}{dt} = -g\left(1 - \frac{\varrho}{\varrho_b}\right) - -\frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v\,.$$

Name of program file: `body_in_fluid2.py`. ◇

Exercise E.32. *Add the effect of air resistance on a ball.*

The differential equations governing the horizontal and vertical motion of a ball subject to gravity and air resistance read[3]

$$\frac{d^2x}{dt^2} = -\frac{3}{8}C_D\bar{\varrho}a^{-1}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\frac{dx}{dt}, \tag{E.70}$$

$$\frac{d^2y}{dt^2} = -g - \frac{3}{8}C_D\bar{\varrho}a^{-1}\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}\frac{dy}{dt}, \tag{E.71}$$

where (x, y) is the position of the ball (x is a horizontal measure and y is a vertical measure), g is the acceleration of gravity, $C_D = 0.2$ is a drag coefficient, $\bar{\varrho}$ is the ratio of the density of air and the ball, and a is the radius of the ball. The latter two quantities can be taken as 0.017 and 11 cm for a football.

Let the initial condition be $x = y = 0$ (start position in the origin) and

$$dx/dt = v_0\cos\theta, \quad dy/dt = v_0\sin\theta,$$

where v_0 is the magnitude of the initial velocity and θ is the angle the velocity makes with the horizontal. For a hard football kick we can set $v_0 = 120$ km/h and take θ as 30 degrees.

Express the two second-order equations above as a system of four first-order equations with four initial conditions. Implement the right-hand side in a problem class where the physical parameters C_D, $\bar{\varrho}$, a, v_0, and θ are stored along with the initial conditions. You may also want to add a `terminate` method in this class for checking when the ball hits the ground and then terminate the solution process.

Solve the ODE system for $C_D = 0$ (no air resistance) and $C_D = 0.2$, and plot y as a function of x in both cases to illustrate the effect of air resistance. Use the 4-th order Runge-Kutta method. Make sure you express all units in kg, m, s, and radians. Name of program file: `kick2D.py`. ◇

[3] The equations arise by combining the models in Exercises 1.11 and 1.13.

Exercise E.33. *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

$$L\frac{dI}{dt} + RI + \frac{Q}{C} = E(t), \tag{E.72}$$

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:

$$\frac{dQ}{dt} = I\,. \tag{E.73}$$

Equations (E.72)–(E.73) is a system two ODEs. Solve these for $L = 1$ H, $E(t) = 2\sin\omega t$ V, $\omega^2 = 3.5$ s^{-2}, $C = 0.25$ C, $R = 0.2$ Ω, $I(0) = 1$ 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 E.34 compares the Forward Euler scheme with the more accurate 4th-order Runge-Kutta method.) Name of program file: `electric_circuit.py`. ◇

Exercise E.34. *Compare methods for solving* (E.72)–(E.73).

Consider the system of ODEs in Exercise E.33 for simulating an electric circuit. The purpose now is to compare the Forward Euler scheme with the 4-th order Runge-Kutta method. Make a class `Circuit` for storing the physical parameters of the problem (L, R, C, $E(t)$) as well as the initial conditions ($I(0)$, $Q(0)$). Class `Circuit` should also define the right-hand side of the ODE through a `__call__` method. Create two solver instances, one from the `ForwardEuler` class and one from the `RungeKutta4` class. Solve the ODE system using both methods. Plot the two $I(t)$ solutions for comparison. As you will see, the Forward Euler scheme overestimates the amplitudes significantly, compared with the more accurate 4th-order Runge-Kutta method. Name of program file: `electric_circuit2.py`. ◇

Exercise E.35. *Simulate the spreading of a disease.*

We shall in this exercise model epidemiological diseases such as measles or swine flu. Suppose we have three categories of people: susceptibles (S) who can get the disease, infected (I) who has developed the disease and who can infect susceptibles, and recovered (R) who have recovered from the disease and become immune. Let $S(t)$, $I(t)$, and $R(t)$ be the number of people in category S, I, and R, respectively.

We have that $S + I + R = N$, where N is the size of the population, assumed constant here for simplicity.

When people mix in the population there are SI possible pairs of susceptibles and infected, and a certain fraction βSI per time interval does meet with the result that the infected "successfully" infects the susceptible. During a time interval Δt, $\beta SI\Delta t$ get infected and move from the S to the I category:

$$S(t + \Delta t) = S(t) - \beta SI\Delta t\,.$$

We divide by Δt and let $\Delta \to 0$ to get the differential equation

$$S'(t) = -\beta SI\,. \tag{E.74}$$

A fraction νI of the infected will per time unit recover from the disease. In a time Δt, $\nu I\Delta t$ recover and move from the I to the R category. The quantity $1/\nu$ typically reflects the duration of the disease. In the same time interval, $\beta SI\Delta t$ come from the S to the I category. The accounting for the I category therefore becomes

$$I(t + \Delta t) = I(t) + \beta SI\Delta t - \nu I\Delta t,$$

which in the limit $\Delta t \to \infty$ becomes the differential equation

$$I'(t) = \beta SI - \nu I\,. \tag{E.75}$$

Finally, the R category gets contributions from the I category:

$$R(t + \Delta t) = R(t) + \nu I\Delta t\,.$$

The corresponding ODE for R reads

$$R'(t) = \nu I\,. \tag{E.76}$$

In case the recovered do not become immune, we do not need the recovered category, since the recovered go directly out of the I category to the S category again. This gives a contribution νI to the equation for S and we end up with the S–I system (C.32)–(C.33) from Appendix C.5.

The system (E.74)–(E.76) is known as a SIR model in epidemiology (which is the name of the scientific field studying the spreading of epidemic diseases).

Solve the equations in the SIR model by any numerical method of your choice. Show from the equations that $S' + I' + R' = 0$, which means that $S + I + R$ must be constant. This can be used to test the program. Let $S(0) = 1500$, $I(0) = I_0$, and $R(0) = 0$. Set $\nu = 0.1$, $I_0 = 1$, $\Delta t = 0.5$, and $t \in [0, 60]$. Time t here counts days. Make a plot corresponding to $\beta = 0.0005$. Certain precautions, like staying inside, will reduce β. Try $\beta = 0.0001$ and watch the effect on $S(t)$. Name of program file: `SIR.py`. ◇

Exercise E.36. *Make a more flexible code in Exer. E.35.*

The parameters ν and β in the SIR model in Exercise E.35 can be constants or functions of time. Now we shall make an implementation of the $f(u,t)$ function specifying the ODE system such that ν and β can be given as either a constant or a Python function. Introduce a class for $f(u,t)$, with the following code sketch:

```
class Problem:
    def __init__(self, nu, beta, S0, I0, R0, T):
        """
        nu, beta: parameters in the ODE system
        S0, I0, R0: initial values
        T: simulation for t in [0,T]
        """
        if isinstance(nu, (float,int)):  # number?
            self.nu = lambda t: nu       # wrap as function
        elif callable(nu):
            self.nu = nu

        # same for beta and self.beta
        ...

        # store the other parameters

    def __call__(self, u, t):
        """Right-hand side function of the ODE system."""
        S, I, R = u
        return [-self.beta(t)*S*I,    # S equation
                ...,                  # I equation
                -self.nu(t)*I]        # R equation

# Example:
problem = Problem(beta=lambda t: 0.0005 if t <= 12 else 0.0001,
                  nu=0.1, S0=1500, I0=1)
solver = ODESolver.ForwardEuler(problem.f)
```

Write the complete code for class `Problem` based on the sketch of ideas above. The ν parameter is usually not varying with time as $1/\nu$ is a characteristic size of the period a person is sick, but introduction of new medicine during the disease might change the picture such that time dependence becomes relevant.

We can also make a class `Solver` for solving the problem (see Appendix E.2.8 for similar examples):

```
class Solver:
    def __init__(self, problem, dt):
        self.problem, self.dt = problem, dt

    def solve(self, method=ODESolver.RungeKutta4):
        self.solver = method(problem)
        ic = [self.problem.S0, self.problem.I0, self.problem.R0]
        self.solver.set_initial_condition(ic)
        n = int(round(self.problem.T/float(self.dt)))
        t = np.linspace(0, self.problem.T, n+1)
        u, self.t = self.solver.solve(t)
        self.S, self.I, self.R = u[:,0], u[:,1], u[:,2]
    def plot(self):
        # plot S(t), I(t), and R(t)
```

After the breakout of a disease, authorities often start campaigns for decreasing the spreading of the disease. Suppose a massive campaign telling people to wash their hands more frequently is launched, with the effect that β is significantly reduced after a some days. For the specific case simulated in Exercise E.35, let

$$\beta(t) = \begin{cases} 0.0005, & 0 \leq t \leq 12, \\ 0.0001, & t > 12 \end{cases}$$

Simulate this scenario with the `Problem` and `Solver` classes. Report the maximum number of infected people and compare it to the case where $\beta(t) = 0.0005$. Name of program file: `SIR_class.py`. ◇

Exercise E.37. *Introduce vaccination in Exer. E.35.*

We shall now extend the SIR model in Exercise E.35 with a vaccination program. If a fraction p of the susceptibles per time unit is being vaccinated, and we say that the vaccination is 100% effective, $pS\Delta t$ individuals will be removed from the S category in a time interval Δt. We place the vaccinated people in a new category V. The equations for S and V becomes

$$S' = -\beta SI - pS, \tag{E.77}$$

$$V' = pS\,. \tag{E.78}$$

The equations for I and R are not affected. The initial condition for V can be taken as $V(0) = 0$. The resulting model is named SIRV.

Make a program for computing $S(t)$, $I(t)$, $R(t)$, and $V(t)$. Try the same parameters as in Exercise E.35 in combination with $p = 0.1$. Watch the effect of vaccination on the maximum number of infected. Name of program file: `SIRV.py`. ◇

Exercise E.38. *Introduce a vaccination campaign in Exer. E.37.*

Let the vaccination campaign in Exercise E.37 start 6 days after the outbreak of the disease and let it last for 10 days,

$$p(t) = \begin{cases} 0.1, & 6 \leq t \leq 15, \\ 0, & \text{otherwise} \end{cases}$$

Plot the corresponding solutions $S(t)$, $I(t)$, $R(t)$, and $V(t)$. (It is clearly advantageous to have the SIRV model implemented as an extension to the classes in Exercise E.36.) Name of program file: `SIRV_varying_p.py`. ◇

Exercise E.39. *Find optimal vaccination period in Exer. E.38.*

Let the vaccination campaign in Exercise E.38 last for V_T days:

$$p(t) = \begin{cases} 0.1, & 6 \leq t \leq 6 + V_T, \\ 0, & \text{otherwise} \end{cases}$$

Compute the maximum number of infected people, $\max_t I(t)$, as a function of $V_T \in [0, 31]$. Plot this function. Determine the optimal V_T, i.e., the smallest vaccination period V_T such that increasing V_T has negligible effect on the maximum number of infected people. Name of program file: `SIRV_optimal_duration.py`. ◇

Exercise E.40. *Simulate human–zombie interaction.*

Suppose the human population is attacked by zombies. This is quite a common happening in movies, and the "zombification" of humans acts much like the spreading of a disease. Let us make a differential equation model, inspired by the SIR model from Exercise E.35, to simulate how humans and zombies interact.

We introduce four categories of individuals:

1. S: susceptible humans who can become zombies.
2. I: infected humans, being bitten by zombies.
3. Z: zombies.
4. R: removed individuals, either conquered zombies or dead humans.

The corresponding functions counting how many individuals we have in each category are named $S(t)$, $I(t)$, $Z(t)$, and $R(t)$, respectively.

The type of zombies considered here is inspired by the standard for modern zombies set by the classic movie *The Night of the Living Dead*, by George A. Romero from 1968. Only a small extension of the SIR model is necessary to model the effect of human–zombie interaction mathematically. A fraction of the human susceptibles is getting bitten by zombies and moves to the infected category. A fraction of the infected is then turned into zombies. On the other hand, humans can conquer zombies.

Now we shall precisely set up all the dynamic features of the human-zombie populations we aim to model. Changes in the S category are due to three effects:

1. Susceptibles are infected by zombies, modeled by a term $-\Delta t\beta SZ$, similar to the S-I interaction in the SIR model.
2. Susceptibles die naturally or get killed and therefore enter the removed category. If the probability that one susceptible dies during a unit time interval is δ_S, the total expected number of deaths in a time interval Δt becomes $\Delta t\delta_S S$.
3. We also allow new humans to enter the area with zombies, as this effect may be necessary to successfully run a war on zombies. The number of new individuals in the S category arriving per time unit is denoted by Σ, giving an increase in $S(t)$ by $\Delta t\Sigma$ during a time Δt.

We could also add newborns to the S category, but we simply skip this effect since it will not be significant over time scales of a few days.

The balance of the S category is then

$$S' = \Sigma - \beta SZ - \delta_S S,$$

in the limit $\Delta t \to 0$.

The infected category gets a contribution $\Delta t \beta SZ$ from the S category, but loses individuals to the Z and R category. That is, some infected are turned into zombies, while others die. Movies reveal that infected may commit suicide or that others (susceptibles) may kill them. Let δ_I be the probability of being killed in a unit time interval. During time Δt, a total of $\delta_I \Delta t I$ will die and hence be transferred to the removed category. The probability that a single infected is turned into a zombie during a unit time interval is denoted by ρ, so that a total of $\Delta t \rho I$ individuals are lost from the I to the Z category in time Δt. The accounting in the I category becomes

$$I' = \beta SZ - \rho I - \delta_I I\,.$$

The zombie category gains $-\Delta t \rho I$ individuals from the I category. We disregard the effect that any removed individual can turn into a zombie again, as we consider that effect as pure magic beyond reasonable behavior, at least according to what is observed in the Romero movie tradition. A fundamental feature in zombie movies is that humans can conquer zombies. Here we consider zombie killing in a "man-to-man" human–zombie fight. This interaction resembles the nature of zombification (or the susceptible-infective interaction in the SIR model) and can be modeled by a loss $-\alpha SZ$ for some parameter α with an interpretation similar to that of β. The equation for Z then becomes

$$Z' = \rho I - \alpha SZ\,.$$

The accounting in the R category consists of a gain δS of natural deaths from the S category, a gain δI from the I category, and a gain αSZ from defeated zombies:

$$R' = \delta_S S + \delta_I I + \alpha SZ\,.$$

The complete SIZR model for human–zombie interaction can be summarized as

$$S' = \Sigma - \beta SZ - \delta_S S, \tag{E.79}$$
$$I' = \beta SZ - \rho I - \delta_I I, \tag{E.80}$$
$$Z' = \rho I - \alpha SZ, \tag{E.81}$$
$$R' = \delta_S S + \delta_I I + \alpha SZ\,. \tag{E.82}$$

The interpretations of the parameters are as follows:

- Σ: the number of new humans brought into the zombified area per unit time.
- β: the probability that a theoretically possible human-zombie pair actually meets physically, during a unit time interval, with the result that the human is infected.
- δ_S: the probability that a susceptible human is killed or dies, in a unit time interval.
- δ_I: the probability that an infected human is killed or dies, in a unit time interval.
- ρ: the probability that an infected human is turned into a zombie, during a unit time interval.
- α: the probability that, during a unit time interval, a theoretically possible human-zombie pair fights and the human kills the zombie.

Note that probabilities per unit time do not necessarily lie in the interval $[0, 1]$. The real probability, lying between 0 and 1, arises after multiplication by the time interval of interest.

Implement the SIZR model with a `Problem` and `Solver` class as explained in Exercise E.36, allowing parameters to vary in time. The time variation is essential to make a realistic model that can mimic what happens in movies. It becomes necessary (cf. Exercise E.41) to work with piecewise constant functions in time, for instance from `scitools.std` (or Exercise 3.27):

```
from scitools.std import PiecewiseConstant

# Define f(t) as 1.5 in [0,3], 0.1 in [3,4] and 1 in [4,7]
f = PiecewiseConstant(domain=[0, 7],
                      data=[(0, 1.5), (3, 0.1), (4, 1)])
```

Test the implementation with the following data: $\beta = 0.0012$, $\alpha = 0.0016$, $\delta_I = 0.014$, $\Sigma = 2$, $\rho = 1$, $S(0) = 10$, $Z(0) = 100$, $I(0)$, $R(0) = 0$, and simulation time $T = 24$ hours. All other parameters can be set to zero. These values are estimated from the hysterical phase of the movie *The Night of the Living Dead*. The time unit is hours. Plot the S, I, Z, and R quantities. Name of program file: `SIZR.py`. ◇

Exercise E.41. *Simulate an entire zombie movie.*

The movie *The Night of the Living Dead* has three phases:

1. The initial phase, lasting for (say) 4 hours, where two humans meet one zombie and of the humans get infected. A rough (and uncertain) estimation of parameters in this phase, taking into account dynamics not shown in the movie, yet necessary to establish a more realistic evolution of the S and Z categories later in the movie, is $\Sigma = 20$, $\beta = 0.03$, $\rho = 1$, $S(0) = 60$, and $Z(0) = 1$. All other parameters are taken as zero when not specified.

2. The hysterical phase, when the zombie treat is evident. This phase lasts for 24 hours, and relevant parameters can be taken as $\beta = 0.0012$, $\alpha = 0.0016$, $\delta_I = 0.014$, $\Sigma = 2$, $\rho = 1$.
3. The counter attack by humans, estimated to last for 5 hours, with parameters $\alpha = 0.006$, $\beta = 0$ (humans no longer get infected), $\delta_S = 0.0067$, $\rho = 1$.

Use the program from Exercise E.40 to simulate all three phases of the movie. Name of program file: `Night_of_the_Living_Dead.py`. ◇

Exercise E.42. *Simulate a war on zombies.*

A war on zombies can be implemented through large-scale effective attacks. A possible model is to increase α in the SIZR model from Exercise E.40 by some amount $\omega(t)$, where $\omega(t)$ varies in time to model strong attacks at some distinct points of time $T_1 < T_2 < \cdots < T_m$. Around these t values we want ω to have a large value, while in between the attacks ω is small. One possible mathematical function with this behavior is a sum of Gaussian functions:

$$\omega(t) = a\sum_{i=0}^{m} \exp\left(\frac{1}{2}\left(\frac{t-T_i}{\sigma}\right)^2\right), \tag{E.83}$$

where a measures the strength of the attacks (the maximum value of $\omega(t)$) and σ measures the length of the attacks, which should be much less than the time between the points of attack: typically, 4σ measures the length of an attack, and we must have $4\sigma \ll T_i - T_{i-1}$. We should choose a significantly larger than α to make the attacks in the war on zombies much stronger than the "man-to-man" killing of zombies.

Modify the model and the implementation from Exercise E.40 to include a war on zombies. As a demonstration, we start out with 50 humans and 3 zombies, and $\beta = 0.0625$ as estimated from *The Night of The Living Dead* movie. These values lead to a rapid zombification. We assume there are some small resistances against zombies from the humans: $\alpha = 0.2\beta$. However, the humans implement three strong attacks, $a = 50\alpha$, at 5, 10, and 18 hours after zombification starts. The attacks last for about 2 hours ($\sigma = 0.5$). Add this effect to the simulation in Exercise E.41 and see if such attacks are sufficient to save mankind in this particular case. Use $\rho = 1$ as before and set non-specified parameters to zero. Name of program file: `war_on_zombies.py`. ◇

Exercise E.43. *Explore predator-prey population interactions.*

Suppose we have two species in an environment: a predator and a prey. How will the two populations interact and change with time? A system of ordinary differential equations can give insight into this question. Let $x(t)$ and $y(t)$ be the size of the prey and and the predator populations, respectively. In the absence of a predator, the population of the prey will follow the ODE derived in Appendix C.2:

$$\frac{dx}{dt} = rx,$$

with $r > 0$, assuming there are enough resources for exponential growth. Similarly, in the absence of prey, the predator population will just experience a death rate $m > 0$:

$$\frac{dy}{dt} = -my\,.$$

In the presence of the predator, the prey population will experience a reduction in the growth proportional to xy. The number of interactions (meetings) between x and y numbers of animals is xy, and in a certain fraction of these interactions the predator eats the prey. The predator population will correspondingly experience a growth in the population because of the xy interactions with the prey population. The adjusted growth of both populations can now be expressed as

$$\frac{dx}{dt} = rx - axy, \tag{E.84}$$

$$\frac{dy}{dt} = -my + bxy, \tag{E.85}$$

for positive constants r, m, a, and b. Solve this system and plot $x(t)$ and $y(t)$ for $r = m = 1$, $a = 0.3$, $b = 0.2$, $x(0) = 1$, and $y(0) = 1$, $t \in [0, 20]$. Try to explain the dynamics of the population growth you observe. Experiment with other values of a and b. Name of program file: `predator_prey.py`. ◇

Exercise E.44. *Formulate a 2nd-order ODE as a system.*

In this and subsequent exercises we shall deal with the following second-order ordinary differential equation with two initial conditions:

$$m\ddot{u} + f(\dot{u}) + s(u) = F(t), \quad t > 0, \quad u(0) = U_0,\ \dot{u}(0) = V_0\,. \tag{E.86}$$

The notation $\dot{u}$ and $\ddot{u}$ means $u'(t)$ and $u''(t)$, respectively. Write (E.86) as a system of two first-order differential equations. Also set up the initial condition for this system.

Physical Applications. Equation (E.86) has a wide range of applications throughout science and engineering. A primary application is damped spring systems in, e.g., cars and bicycles: u is the vertical displacement of the spring system attached to a wheel; $\dot{u}$ is then the corresponding velocity; $F(t)$ resembles a bumpy road; $s(u)$ represents the force from the spring; and $f(\dot{u})$ models the damping force (friction) in the spring system. For this particular application f and s will normally be linear functions of their arguments: $f(\dot{u}) = \beta\dot{u}$ and $s(u) = ku$, where k is a spring constant and β some parameter describing viscous damping.

Equation (E.86) can also be used to describe the motions of a moored ship or oil platform in waves: the moorings act as a nonlinear spring $s(u)$; $F(t)$ represents environmental excitation from waves, wind, and current; $f(\dot{u})$ models damping of the motion; and u is the one-dimensional displacement of the ship or platform.

Oscillations of a pendulum can be described by (E.86): u is the angle the pendulum makes with the vertical; $s(u) = (mg/L)\sin(u)$, where L is the length of the pendulum, m is the mass, and g is the acceleration of gravity; $f(\dot{u}) = \beta|\dot{u}|\dot{u}$ models air resistance (with β being some suitable constant, see Exercises 1.11 and E.48); and $F(t)$ might be some motion of the top point of the pendulum.

Another application is electric circuits with $u(t)$ as the charge, $m = L$ as the inductance, $f(\dot{u}) = R\dot{u}$ as the voltage drop across a resistor R, $s(u) = u/C$ as the voltage drop across a capacitor C, and $F(t)$ as an electromotive force (supplied by a battery or generator).

Furthermore, Equation (E.86) can act as a simplified model of many other oscillating systems: aircraft wings, lasers, loudspeakers, microphones, tuning forks, guitar strings, ultrasound imaging, voice, tides, the El Niño phenomenon, climate changes – to mention some.

We remark that (E.86) is a possibly nonlinear generalization of Equation (D.8) explained in Appendix D.1.3. The case in Appendix D corresponds to the special choice of $f(\dot{u})$ proportional to the velocity $\dot{u}$, $s(u)$ proportional to the displacement u, and $F(t)$ as the acceleration $\ddot{w}$ of the plate and the action of the gravity force. ◇

Exercise E.45. *Solve the system in Exer. E.44 in a special case.*

Make a function

```
def rhs(u, t):
    ...
```

for returning a list with two elements with the two right-hand side expressions in the first-order differential equation system from Exercise E.44. As usual, the `u` argument is an array or list with the two solution components `u[0]` and `u[1]` at some time `t`. Inside `rhs`, assume that you have access to three global Python functions `friction(dudt)`, `spring(u)`, and `external(t)` for evaluating $f(\dot{u})$, $s(u)$, and $F(t)$, respectively.

Test the `rhs` function in combination with the functions $f(\dot{u}) = 0$, $F(t) = 0$, $s(u) = u$, and the choice $m = 1$. The differential equation then reads $\ddot{u} + u = 0$. With initial conditions $u(0) = 1$ and $\dot{u}(0) = 0$, one can show that the solution is given by $u(t) = \cos(t)$. Apply three numerical methods: the 4th-order Runge-Kutta method and the Forward Euler method from the `ODESolver` module developed in Appendix E.2.5, as well as the 2nd-order Runge-Kutta method developed in Exercise E.22. Use a time step $\Delta t = \pi/20$.

Plot $u(t)$ and $\dot{u}(t)$ versus t together with the exact solutions. Also make a plot of $\dot{u}$ versus u (`plot(u[:,0], u[:,1])` if `u` is the array returned from the solver's `solve` method). In the latter case, the exact plot should be a circle[4], but the ForwardEuler method results in a spiral. Investigate how the spiral develops as Δt is reduced.

The kinetic energy K of the motion is given by $\frac{1}{2}m\dot{u}^2$, and the potential energy P (stored in the spring) is given by the work done by the spring force: $P = \int_0^u s(v)dv = \frac{1}{2}u^2$. Make a plot with K and P as functions of time for both the 4th-order Runge-Kutta method and the Forward Euler method, for the same physical problem described above. In this test case, the sum of the kinetic and potential energy should be constant. Compute this constant analytically and plot it together with the sum $K + P$ as calculated by the 4th-order Runge-Kutta method and the Forward Euler method.

Name of program file: `oscillator_v1.py`. ◇

Exercise E.46. *Make a tool for analyzing oscillatory solutions.*

The solution $u(t)$ of the equation (E.86) often exhibits an oscillatory behavior (for the test problem in Exercise E.45 we have that $u(t) = \cos t$). It is then of interest to find the wavelength of the oscillations. The purpose of this exercise is to find and visualize the distance between peaks in a numerical representation of a continuous function.

Given an array $(y_0, \ldots, y_{n-1})$ representing a function $y(t)$ sampled at various points $t_0, \ldots, t_{n-1}$, a local maximum of $y(t)$ occurs at $t = t_k$ if $y_{k-1} < y_k > y_{k+1}$. Similarly, a local minimum of $y(t)$ occurs at $t = t_k$ if $y_{k-1} > y_k < y_{k+1}$. By iterating over the $y_1, \ldots, y_{n-2}$ values and making the two tests, one can collect local maxima and minima as (t_k, y_k) pairs. Make a function `minmax(t, y)` which returns two lists, `minima` and `maxima`, where each list holds pairs (2-tuples) of t and y values of local minima or maxima. Ensure that the t value increases from one pair to the next. The arguments `t` and `y` in `minmax` hold the coordinates $t_0, \ldots, t_{n-1}$ and $y_0, \ldots, y_{n-1}$, respectively.

Make another function `wavelength(peaks)` which takes a list `peaks` of 2-tuples with t and y values for local minima or maxima as argument and returns an array of distances between consecutive t values, i.e., the distances between the peaks. These distances reflect the local wavelength of the computed y function. More precisely, the first element in the returned array is `peaks[1][0]-peaks[0][0]`, the next element is `peaks[2][0]-peaks[1][0]`, and so forth.

Test the `minmax` and `wavelength` functions on y values generated by $y = e^{t/4}\cos(2t)$ and $y = e^{-t/4}\cos(t^2/5)$ for $t \in [0, 4\pi]$. Plot the $y(t)$ curve in each case, and mark the local minima and maxima computed by `minmax` with circles and boxes, respectively. Make a separate plot with the array returned from the `wavelength` function (just plot the

[4] The points on the curve are $(\cos t, \sin t)$, which all lie on a circle as t is varied.

array against its indices - the point is to see if the wavelength varies or not). Plot only the wavelengths corresponding to maxima.

Make a module with the `minmax` and `wavelength` function, and let the test block perform the tests specified above. Name of program file: `wavelength.py`. ◇

Exercise E.47. *Enhance the code from Exer. E.45.*

The user-chosen functions f, s, and F in Exercise E.45 must be coded with particular names. It is then difficult to have several functions for $s(u)$ and experiment with these. A much more flexible code arises if we adopt the ideas of a problem and a solver class as explained in Appendix E.2.8. Specifically, we shall here make use of class `Problem3` in Appendix E.2.8 to store information about $f(\dot{u})$, $s(u)$, $F(t)$, $u(0)$, $\dot{u}(0)$, m, T, and the exact solution (if available). The solver class can store parameters related to the numerical quality of the solution, i.e., Δt and the name of the solver class in the `ODESolver` hierarchy. In addition we will make a visualizer class for producing plots of various kinds.

We want all parameters to be set on the command line, but also have sensible default values. As in Appendix E.2.8, the `argparse` module is used to read data from the command line. Class `Problem` can be sketched as follows:

```
class Problem:
    def define_command_line_arguments(self, parser):
        """Add arguments to parser (argparse.ArgumentParser)."""

        parser.add_argument(
            '--friction', type=func_dudt, default='0',
            help='friction function f(dudt)',
            metavar='<function expression>')
        parser.add_argument(
            '--spring', type=func_u, default='u',
            help='spring function s(u)',
            metavar='<function expression>')
        parser.add_argument(
            '--external', type=func_t, default='0',
            help='external force function F(t)',
            metavar='<function expression>')
        parser.add_argument(
            '--u_exact', type=func_t_vec, default='0',
            help='exact solution u(t) (0 or None: now known)',
            metavar='<function expression>')
        parser.add_argument(
            '--m', type=evalcmlarg, default=1.0, help='mass',
            type=float, metavar='mass')
        ...
        return parser

    def set(self, args):
        """Initialize parameters from the command line."""
        self.friction = args.friction
        self.spring = args.spring
        self.m = args.m
        ...

    def __call__(self, u, t):
        """Define the right-hand side in the ODE system."""
        m, f, s, F = \
           self.m, self.friction, self.spring, self.external
        ...
```

Several functions are specified as the `type` argument to `parser.add_argument` for turning strings into proper objects, in particular `StringFunction` objects with different independent variables:

```
def evalcmlarg(text):
    return eval(text)

def func_dudt(text):
    return StringFunction(text, independent_variable='dudt')

def func_u(text):
    return StringFunction(text, independent_variable='u')

def func_t(text):
    return StringFunction(text, independent_variable='t')

def func_t_vec(text):
    if text == 'None' or text == '0':
        return None
    else:
        f = StringFunction(text, independent_variable='t')
        f.vectorize(globals())
        return f
```

The use of `evalcmlarg` is essential: this function runs the strings from the command line through `eval`, which means that we can use mathematical formulas like `-T '4*pi'`.

Class `Solver` is relatively much shorter than class `Problem`:

```
class Solver:
    def __init__(self, problem):
        self.problem = problem

    def define_command_line_arguments(self, parser):
        """Add arguments to parser (argparse.ArgumentParser)."""
        # add --dt and --method
        ...
        return parser

    def set(self, args):
        self.dt = args.dt
        self.n = int(round(self.problem.T/self.dt))
        self.solver = eval(args.method)

    def solve(self):
        self.solver = self.method(self.problem)
        ic = [self.problem.initial_u, self.problem.initial_dudt]
        self.solver.set_initial_condition(ic)
        time_points = linspace(0, self.problem.T, self.n+1)
        self.u, self.t = self.solver.solve(time_points)
```

The `Visualizer` class holds references to a `Problem` and `Solver` instance and creates plots. The user can specify plots in an interactive dialog in the terminal window. Inside a loop, the user is repeatedly asked to specify a plot until the user responds with `quit`. The specification of a plot can be one of the words `u`, `dudt`, `dudt-u`, `K`, and `wavelength` which means a plot of $u(t)$ versus t, $\dot{u}(t)$ versus t, $\dot{u}$ versus u, K $(=\frac{1}{2}m\dot{u}^2$, kinetic energy) versus t, and u's wavelength versus its indices, respectively. The wavelength can be computed from the local maxima of u as explained in Exercise E.46.

A sketch of class `Visualizer` is given next:

```
class Visualizer:
    def __init__(self, problem, solver):
        self.problem = problem
        self.solver = solver

    def define_command_line_arguments(self, parser):
        parser.add_argument(
            '--plot', type=str, default='u dudt',
            help='specification of types of plots',
            metavar='<space separated list of plot types>')
        return parser

    def set(self, args):
        plot_spec = args.plot
        self.plots = plot_spec.split()

    def visualize(self):
        t = self.solver.t   # short form
        u, dudt = self.solver.u[:,0], self.solver.u[:,1]

        # Tag all plots with numerical and physical input values
        title = 'solver=%s, dt=%g, m=%g' % \
               (self.solver.method, self.solver.dt, self.problem.m)
        # Can easily get the formula for friction, spring and force
        # if these are string formulas.
        if isinstance(self.problem.friction, StringFunction):
            title += ' f=%s' % str(self.problem.friction)
        if isinstance(self.problem.spring, StringFunction):
            title += ' s=%s' % str(self.problem.spring)
        if isinstance(self.problem.external, StringFunction):
            title += ' F=%s' % str(self.problem.external)

        plot_type = ''
        while plot_type != 'quit':
            plot_type = raw_input('Specify a plot: ')
            figure()
            if plot_type == 'u':
                # Plot u vs t
                if self.problem.u_exact is not None:
                    hold('on')
                    # Plot self.problem.u_exact vs t
                show()
                savefig('tmp_u.eps')
            elif plot_type == 'dudt':
            ...
```

Make a complete implementation of the three proposed classes. Also make a `main` function that (i) creates a problem, solver, and visualizer, (ii) calls the functions to define command-line arguments in each of them, (iii) reads the command line, (iv) passes on the `args` from the reading of the command line to the problem, solver, and visualizer classes, (v) calls the solver, and (vi) calls the visualizer's `visualize` method to create plots. Collect the classes and functions in a module `oscillator`, which has a call to `main` in the test block.

The first task from Exercises E.45 can now be run as

Terminal

```
oscillator.py --method ForwardEuler --u_exact "cos(t)" \
              --dt "pi/20" --T "5*pi"
```

The other tasks from Exercises E.45 can be tested similarly.

Explore some of the possibilities of specifying several functions on the command line:

Terminal

```
oscillator.py --method RungeKutta4 --friction "0.1*dudt" \
              --external "sin(0.5*t)" --dt "pi/80" \
              --T "40*pi" --m 10

oscillator.py --method RungeKutta4 --friction "0.8*dudt" \
              --external "sin(0.5*t)" --dt "pi/80" \
              --T "120*pi" --m 50
```

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

Exercise E.48. *Allow flexible choice of functions in Exer. E.47.*

Some typical choices of $f(\dot{u})$, $s(u)$, and $F(t)$ in (E.86) are listed below:

1. Linear friction force (low velocities): $f(\dot{u}) = 6\pi\mu R\dot{u}$ (Stokes drag), where R is the radius of a spherical approximation to the body's geometry, and μ is the viscosity of the surrounding fluid.
2. Quadratic friction force (high velocities): $f(\dot{u}) = \frac{1}{2}C_D\varrho A|\dot{u}|\dot{u}$, see Exercise 1.11 for explanation of symbols.
3. Linear spring force: $s(u) = ku$, where k is a spring constant.
4. Sinusoidal spring force: $s(u) = k\sin u$, where k is a constant.
5. Cubic spring force: $s(u) = k(u - \frac{1}{6}u^3)$, where k is a spring constant.
6. Sinusoidal external force: $F(t) = F_0 + A\sin\omega t$, where F_0 is the mean value of the force, A is the amplitude, and ω is the frequency.
7. "Bump" force: $F(t) = H(t - t_1)(1 - H(t - t_2))F_0$, where $H(t)$ is the Heaviside function from Exercise 3.24, t_1 and t_2 are two given time points, and F_0 is the size of the force. This $F(t)$ is zero for $t < t_1$ and $t > t_2$, and F_0 for $t \in [t_1, t_2]$.
8. Random force 1: $F(t) = F_0 + A \cdot U(t; B)$, where F_0 and A are constants, and $U(t; B)$ denotes a function whose value at time t is random and uniformly distributed in the interval $[-B, B]$.
9. Random force 2: $F(t) = F_0 + A \cdot N(t; \mu, \sigma)$, where F_0 and A are constants, and $N(t; \mu, \sigma)$ denotes a function whose value at time t is random, Gaussian distributed number with mean μ and standard deviation σ.

Make a module `functions` where each of the choices above are implemented as a class with a `__call__` special method. Also add a class `Zero` for a function whose value is always zero. It is natural that the parameters in a function are set as arguments to the constructor. The different classes for spring functions can all have a common base class holding the k parameter as attribute. Name of program file: `functions.py`. ◇

Exercise E.49. *Use the modules from Exer. E.47 and E.48.*

The purpose of this exercise is to demonstrate the use of the classes from Exercise E.48 to solve problems described by (E.86).

With a lot of models for $f(\dot{u})$, $s(u)$, and $F(t)$ available as classes in `functions.py`, the initialization of `self.friction`, `self.spring`, etc., from the command line does not work, because we assume simple string formulas on the command line. Now we want to write things like `-spring 'LinearSpring(1.0)'`. There is a quite simple remedy: replace all the special conversion functions to `StringFunction` objects by `evalcmlarg` in the `type` specifications in the `parser.add_argument` calls. If a `from functions import *` is also performed in the `oscillator.py` file, a simple `eval` will turn strings like `'LinearSpring(1.0)'` into living objects.

However, we shall here follow a simpler approach, namely dropping initializing parameters on the command line and instead set them directly in the code. Here is an example:

```
problem = Problem()
problem.m = 1.0
k = 1.2
problem.spring = CubicSpring(k)
problem.friction = Zero()
problem.T = 8*pi/sqrt(k)
...
```

This is the simplest way of making use of the objects in the `functions` module.

Note that the `set` method in classes `Solver` and `Visualizer` is unaffected by the new objects from the `functions` module, so flexible initialization via command-line arguments works as before for `-dt`, `-method`, and `plot`. One may also dare to call the `set` method in the problem object to set parameters like `m`, `initial_u`, etc., or one can choose the safer approach of not calling `set` but initialize all attributes explicitly in the user's code.

Make a new file say `oscillator_test.py` where you import class `Problem`, `Sover`, and `Visualizer`, plus all classes from the `functions` module. Provide a `main1` function for solving the following problem: $m = 1$, $u(0) = 1$, $\dot{u}(0) = 0$, no friction (use class `Zero`), no external forcing (class `Zero`), a linear spring $s(u) = u$, $\Delta t = \pi/20$, $T = 8\pi$, and exact $u(t) = \cos(t)$. Use the Forward Euler method.

Then make another function `main2` for the case with $m = 5$, $u(0) = 1$, $\dot{u}(0) = 0$, linear friction $f(\dot{u}) = 0.1\dot{u}$, $s(u) = u$, $F(t) = \sin(\frac{1}{2}t)$, $\Delta t = \pi/80$, $T = 60\pi$, and no knowledge of an exact solution. Use the 4-th order Runge-Kutta method.

Let a test block use the first command-line argument to indicate a call to `main1` or `main2`. Name of program file: `oscillator_test.py`. ◇

Exercise E.50. *Model the economy of fishing.*

A population of fish is governed by the differential equation

$$\frac{dx}{dt} = \frac{1}{10}x\left(1 - \frac{x}{100}\right) - h, \quad x(0) = 500, \tag{E.87}$$

where $x(t)$ is the size of the population at time t and h is the harvest.

a) Assume $h = 0$. Find an exact solution for $x(t)$. For which value of t is $\frac{dx}{dt}$ largest? For which value of t is $\frac{1}{x}\frac{dx}{dt}$ largest?
b) Solve the differential equation (E.87) by the Forward Euler method. Plot the numerical and exact solution in the same plot.
c) Suppose the harvest h depends on the fishers' efforts, E, in the following way: $h = qxE$, with q as a constant. Set $q = 0.1$ and assume E is constant. Show the effect of E on $x(t)$ by plotting several curves, corresponding to different E values, in the same figure.
d) The fishers' total revenue is given by $\pi = ph - \frac{c}{2}E^2$, where p is a constant. In the literature about the economy of fisheries, one is often interested in how a fishery will develop in the case the harvest is not regulated. Then new fishers will appear as long as there is money to earn ($\pi > 0$). It can (for simplicity) be reasonable to model the dependence of E on π as

$$\frac{dE}{dt} = \gamma\pi, \tag{E.88}$$

where γ is a constant. Solve the system of differential equations for $x(t)$ and $E(t)$ by the 4th-order Runge-Kutta method, and plot the curve with points $(x(t), E(t))$ in the two cases $\gamma = 1/2$ and $\gamma \to \infty$. Choose $c = 0.3$, $p = 10$, $E(0) = 0.5$, and $T = 1$.

Name of program file: `fishery.py`. ◇