UiO : Department of Informatics
University of Oslo

Exercises for IN1900

Cecilie Glittum, Eirill Strand Hauge, Simen Moe,
Ella Wolff Kristensen, Kristian Wold and Joakim Sundnes

September 6, 2022

Preface

This document contains a number of programming exercises made for the course IN1900. The chapter numbers and titles correspond mainly to the chapters of the book "Introduction to Scientific Programming in Python" by Joakim Sundnes. The book is a short and updated version of the more extensive book "A primer on Scientific Programming with Python" by Hans Petter Langtangen, and this exercise collection is meant to be a supplement to the exercises in that book. Most of the exercises are new, some are modified versions of exercises from the book of Langtangen, and some are pure copies of exercises from that book, but updated to Python 3. Most of the exercises are motivated by applications in science and applied mathematics.

The exercise collection was first used in the IN1900 course in 2018, but has been substantially updated and extended first in 2019 and then in 2020. If you find typos or have other comments or questions about the exercises, please send them to Joakim Sundnes: *sundnes@simula.no*.

Chapter 1

Introduction to Python

The first chapter of the book the book "Introduction to Scientific Programming in Python" does not contain much programming, but is inteded to get people started by writing and running a very small Python program. The exercises included here serve the same purpose. The programming itself is trivial, but if you are new to Python and programming these exercises are very to verify that you have installed the necessary tools and that they are working correctly.

Problem 1.1. Write a Hello, World! program
Write a Python program that prints the words "Hello, World!" to the screen. Save the program in a file `hello.py` and run it from the terminal, or from an IDE (such as Spyder) if you prefer. Verify that the expected output is printed in the terminal

Filename: `hello.py`

Problem 1.2. Check your Python version
Open a terminal window, for instance `Terminal` on MacOS or `Powershell` on Windows. Type the words

```
python --version
```

in the terminal. You should then see your Python version printed in the terminal, most likely Python 3.7.x, where x is some number. Next, simply type

```
python
```

in the same terminal window. This will open an *interactive Python session*. Your Python version is again displayed first, followed by a special prompt where you can type Python commands. Try to type a simple mathematical expression such as `2+2` and press return. The type the line `print("Hello")`! and press return. Verify that it works as expected, and look for typos in the line you wrote if you get an error message. Finally, write `quit()` or `exit()` to end the session.

Chapter 2

Computing with Formulas

The exercises in this chapter concern with programming of mathematical formulae, and correspond to Chapter 2 in the book by Sundnes and Chapter 1 in the book by Langtangen.

Problem 2.1. Throw a ball
When throwing a ball in the air, the position of the ball can be calculated using the acceleration of the ball. When neglecting air resistance, the acceleration will be the negative of the gravitational constant, $-g$. The height of the ball relative to its starting point is

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

where v_0 is the initial velocity of the ball and t is the time after the throw. The ball reaches its maximum height at time

$$t_{\max} = \frac{v_0}{g}.$$

Write a program computing the maximum height of the ball, that is $y(t_{\max})$, when $v_0 = 8.2\text{m/s}$ and $g = 9.81\text{m/s}^2$. Print the result.

Filename: `ball.py`

Problem 2.2. Growth of a bank deposit Let r be a bank's interest rate in percent per year. An initial amount P will then grow to

$$A = P(1 + r/100)^n$$

after n years. Write a program that computes how much money 1000 euros have grown to after three years with 5 percent interest rate, and prints the amount in the terminal.

Filename: `interest_rate.py`

Problem 2.3. Population growth
The growth of a population can often be described by a logistic function

$$N(t) = \frac{B}{1 + C\mathrm{e}^{-kt}},$$

where B is the carrying capacity of the species in the environment, i.e., the maximum size of the population that the environment can sustain indefinitely.

The constant k tells us something about how fast the population grows, while C is given by the initial conditions. Let us consider a bacterial colony where the carrying capacity is $B = 50000$, and $k = 0.2\text{h}^{-1}$ (where h is the unit of hours). The population is 5000 at $t = 0$. Use the initial condition to determine C and write a program that finds the number of bacteria in the colony after 24 hours.

Hint: To find C you insert $t = 0$ in the expression above, and solve the resulting equation for C. This gives a formula for calculating C as a function of N and B, and this formula can be used directly in your program.

Filename: `population.py`

Problem 2.4. Solve the quadratic equation
Given a quadratic equation

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

the two roots are

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

Make a program evaluating the roots of

$$6x^2 - 5x + 1 = 0.$$

Print out both roots with two decimals.

Filename: `find_roots.py`

Problem 2.5. Forces in the hydrogen atom
There are two kinds of forces acting between the proton and the electron in the hydrogen atom; Coulomb force and gravitational force. The Coulomb force can be expressed as

$$F_C = k_e \frac{e^2}{r^2},$$

where k_e is Coulomb's constant, e is the elementary charge, and r is the distance between the proton and the electron.

The gravitational force can be expressed as

$$F_G = G \frac{m_p m_e}{r^2},$$

where G is the gravitational constant, m_p is the mass of the proton, m_e is the mass of the electron, and r is the distance between the particles.

We can use these expressions for F_C and F_G to illustrate the difference in strength of these two forces, i.e., the electromagnetic force and gravitational force. Use the values $k_e = 9.0{\cdot}10^9$ Nm^2C^{-2}, $e = 1.6{\cdot}10^{-19}$ C, $G = 6.7{\cdot}10^{-11}$ $\text{Nkg}^{-2}\text{m}^2$, $m_p = 1.7{\cdot}10^{-27}$ kg and $m_e = 9.1{\cdot}10^{-31}$ kg. You can take the distance between the proton and electron to be approximately the Bohr radius $r = a_0 = 5.3 \cdot 10^{-11}$ m.

Make a program that computes both the Coulomb force and the gravitational force between the proton and the electron. Write out the forces in scientific notation with one decimal in units of Newton ($\text{N} = \text{kgm/s}^2$). Also print the ratio between the two forces.

Filename: `hydrogen.py`

Problem 2.6. Type in program text
Type the following program in your editor, save it to a file `formulas_shapes.py` and run the program. If thes program does not work, check that you have typed the code correctly.

```
from math import pi

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

area_para = h*b
print(f'The area of the parallelogram is {area_para:.3f}')
area_square = b**2
print(f'The area of the square is {area_square:g}')
area_circle = pi*r**2
print(f'The area of the circle is {area_circle:.3f}')
volume_cone = 1.0/3*pi*r**2*h
print('The volume of the cone is {volume_cone:.3f}')
```

Filename: `formulas_shapes.py`

Chapter 3

Loops and Lists

The exercises of this chapter are about loops and list, and correspond to Chapter 3 in the book by Sundnes and Chapter 2 in the book of Langtangen.

Problem 3.1. Multiply by five
Write a code printing out $5 \cdot 1$, $5 \cdot 2$, ..., $5 \cdot 10$, using either a `for` or a `while` loop.
Filename: `multiplication.py`

Problem 3.2. Multiplication table
Write a new code based on the one from Problem 3.1. This code should print the whole miltiplication table from $1 \cdot 1$ to $10 \cdot 10$.
Hint: You may want to consider using one loop inside another.
Filename: `mult_table.py`

Problem 3.3. Stirling's approximation
Stirling's approximation can be written $\ln(x!) \approx x \ln x - x$. This is a good approximation for large x. Write out a nicely formatted table of integer x values, the actual value of $\ln(x!)$, and Stirling's approximation to $\ln(x!)$.
Filename: `stirling.py`

Problem 3.4. Errors in summation
The following code is supposed to compute the sum $s = \sum_{k=1}^{M} \frac{1}{(2k)^2}$, for $M = 3$.

```
s = 0
M = 3

for i in range(M):
    s += 1/2*k**2

print(s)
```

The program has three errors and therefore does not work. Find the three errors and write a correct program. Put comments in your program to indicate what the mistakes were.

There are two basic ways to find errors in a program:

1. Read the program carefully line by line, and think about the consequences of each statement. Look for inconsistencies in the form of variables being used before they are defined, and perform the calculations in the statements by hand to see how variables change their value.

2. Run the program and check for errors. In a Python program there may be errors of two different types. The first is an error that Python itself will notice, and make the program stop with an error message. The message will indicate which line the error occurs, and alghough the error messages may be a little cryptic these errors are usually quite easy to find and correct. The second type are errors where the program seems to run and complete normally, but the result is not what we want. These errors are often harder to find, and a good method to find them is often to add `print` statements in the code to write intermediate values of variables to the screen, and compare these values with hand calculations.

Try the first method first (*code inspection*) and find as many errors as you can. Thereafter, try the second method, by for instance adding a print statement inside the for-loop to print the value of `k` and `s`.

Filename: `sum_for.py`

Problem 3.5. Sum as a `while` loop Write the (corrected) `for` loop from the previous exercise as a a `while` loop. Check that the two loops compute the same answer.

Filename: `sum_while.py`

Problem 3.6. Binomial coefficient
The binomial coefficient is indexed by two integers n and k and is written $\binom{n}{k}$. It is given by the formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}. \tag{3.1}$$

We can write this out, and get

$$\binom{n}{k} = \prod_{j=1}^{n-k} \frac{k+j}{j}. \tag{3.2}$$

Use Eq. (3.2) and a `for` loop to find the binomial coefficient for $n = 14$ and $k = 3$. Compute the same value using Eq. (3.1) and check that the results are correct.

Hint: The $\prod$ sign is a product sign. Thus $\prod_{j=1}^{n-k} \frac{k+j}{j} = \frac{k+1}{1}\frac{k+2}{2}\cdots\frac{k+(n-k)}{(n-k)}$. When checking the result you will need `math.factorial`.

Filename: `binomial.py`

Problem 3.7. Table showing population growth
Consider again the bacterial colony from Problem 2.3. We now want to study how the population grows with time, by calculating the number of individuals for $n + 1$ uniformly spaced t values throughout the interval $[0, 48]$. Set $n = 12$ and write a `for` loop which computes and stores t and N values in two lists `t` and `N`. Thereafter, traverse the two lists with a separate `for` loop and write out a nicely formatted table of t and N values.

Filename: `population_table.py`

Problem 3.8. Nested list

a) Compute two lists of t and N values as explained in Problem 3.7. Store the two lists in a new nested list `tN1` (havinglength two) where the element `tN1[0]` is the list containing t-values, and `tN[1]` is the list containing the N-values. Write out a table with t and N values in two columns by looping over the data in the `tN1` list. Each t and N value should be written in the table as integers.

b) Make a nested list `tN2` containing exactly the same data as in exercise **a)**, but with a different structure. The `tN2` list shall have the same length as the original t- and N-lists, and each element `tN2[i]` shall be a list of lenght two, which contains the i-th element of both the t-list and the N-list. Loop over the `tN2` list and write out the t and N values in the table as integers. The output should look exactly as in exercise **a)**, although the underlying structure of the lists is different.

Filename: `population_table2.py`

Problem 3.9. Calculate Cesaro mean
Let $(a_n)_{n=1}^{\infty}$ be a sequence of numbers, $s_k = \sum_{n=0}^{k} a_n = a_0 + \ldots, +a_k$, and

$$S_N = \frac{1}{N-1} \sum_{k=0}^{N-1} s_k.$$

Let $(a_n)_{n=1}^{\infty}$ be the sequence with $a_n = (-1)^n$. Calculate S_N for $N = 1, 2, 3, 4, 5, 10, 50$ and print the results in a table.

Filename: `cesaro_mean.py`

Problem 3.10. Catalan numbers
A number on the form

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!}$$

is called a *Catalan number*. Compute and print the first 10 Catalan numbers.

Filename: `catalan.py`

Problem 3.11. Molar Mass of Alkanes
Alkanes are saturated hydrocarbons with the chemical formula $\mathrm{C_nH_{2n+2}}$. If there are n Carbon atoms in the alkane, there will be $m = 2n + 2$ Hydrogen atoms. The molar mass of the hydrocarbon is $M_{\mathrm{C_nH_m}} = nM_{\mathrm{C}} + mM_{\mathrm{H}}$, where M_C is the molar mass of Carbon and M_H is the molar mass of Hydrogen.

Use a `for`-loop or a `while`-loop to compute and print out the molar mass of the alkanes with two through nine Carbon atoms ($n \in [2, 9]$). The output should specify the chemical formula of the alkane as well as the molar mass. An example on how the formatted output should look like for $n = 2$ is given below.

```
M(C2H6) =  30.069 g/mol
```

You can set the molar masses of the atoms to be $M_{\mathrm{C}} = 12.011$ g/mol and $M_{\mathrm{H}} = 1.0079$ g/mol

Hint: An output of a chemical formula like the one above can be constructed using a formatted string (f-string). If we have `n=2` and `m=2*n+2`, the first part of the output above is given by the f-string `:d:df'M(CnH6 =...'`.

Filename: `alkane.py`

Problem 3.12. Simulate a program by hand
Consider the following program for computing with interest rates:

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

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

b) Change the program to make use of the operator `+=` wherever possible.

c) Explain with words what type of mathematical problem that is solved by this program.

Filename: `interest_rate_loop.py`

Problem 3.13. Matrix elements

This exercise involves no programming.
The answers should be written in a text file called `matrix.txt`

Consider a two dimensional 3×3 matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}.$$

In Python, the matrix A can be represented as a nested list `A`, either as a list of rows or a list of columns. Find the indices `i, j` of the Python list `A` such that `A[i][j]` $= a_{11}$ and the indices `k, l` such that `A[k][l]` $= a_{32}$ for the two following cases:

a) When A is represented as a list of rows. This means that `A` contains three lists, where each list corresponds to a row in A.

b) When A is represented as a list of columns. This means that each element in `A` contains a list with the elements of a column in A.

Filename: `matrix.txt`

Chapter 4

Functions and Branching

The main topics of this chapter are functions and branching (if-tests), corresponding to Chapter 4 in the book by Sundnes and Chapter 3 in the book of Langtangen.

Problem 4.1. Implement a function for population growth
Consider again the function

$$N(t, k, B, C) = \frac{B}{1 + C\mathrm{e}^{-kt}}.$$

Implement N as a python function `population(t, k, B, C)` that returns the number of individuals in a population after a time t.

Use a `for` loop to write out a nicely formatted table of t and N values for the time interval $t \in [0, 48]$ using the parameter values from Problem 3.7.

Filename: `pop_func.py`

Problem 4.2. Sum of integers
We consider the sum $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n$ of positive integers up to n. It can be shown that the sum is equal to $\frac{n(n+1)}{2}$.

a) Write a function `sumint(n)` that returns the sum of all positive integers up to n.

b) Write a function implementing $\frac{n(n+1)}{2}$.

c) Write test functions for both a) and b) testing for specific known values.

Filename: `sumint.py`

Problem 4.3. Implement the factorial

a) The factorial can be implemented by a so called recursive function call. Use a recursive function call to implement a function `myfactorial(n)` that returns $n!$.

b) Write a test function where you call the `myfactorial` function and check the value of the returned object for one value of n using `math.factorial`.

Filename: `factorial.py`

Problem 4.4. Compute the area of an arbitrary triangle An arbitrary triangle can be described by the coordinates of its three vertices: $(x1, y1), (x2, y2), (x3, y3)$, numbered in a counterclockwise direction. The area of the triangle is given by the formula

$$A = \frac{1}{2}|x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1|$$

Write a function `triangle_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. Copy and paste the following test function into your code, and call the test function to verify that your `triangle_area` function works:

```
def test_triangle_area():
    """
    Verify the area of a triangle with vertices
    (0,0), (1,0), and (0,2).
    """
    v1 = [0,0];  v2 = [1,0];  v3 = [0,2]
    vertices = [v1, v2, v3]
    expected = 1
    computed = triangle_area(vertices)
    tol = 1E-14
    success = abs(expected - computed) < tol
    msg = f"computed area={computed} != {expected}(expected)"
    assert success, msg
```

Filename: `triangle_area.py`

Problem 4.5. Half-wave rectifier
In a half-wave rectifier the positive part of a signal passes, while the negative part is blocked. Thus, for a signal passing through a half-wave rectifier, the negative values are set to zero. A sine signal that has passed through a half-wave rectifier will look as follows:

$$f(x) = \begin{cases} \sin x & \text{if } \sin x > 0 \\ 0 & \text{if } \sin x \leq 0. \end{cases}$$

Implement $f(x)$ as a Python function `f(x)` and make a test function for testing the implementation of `f(x)`. The test function should test for at least two values of x, one that gives $\sin x < 0$ and one where $\sin x > 0$.

Filename: `half_wave.py`

Problem 4.6. Primality checker
Recall that a prime number is a number greater than 1 that has exactly 2 divisors. Said differently, a number greater than one is a prime if it is divisible by only itself and one. A number that is not prime is called composite. Every number n can be written as a unique product of primes (e.g. $12 = 2 \cdot 2 \cdot 3$), this is called the prime factorization of n.

a) Make a function that takes a number n, and returns true if it's prime, and false if it's not. Use the program to find all prime numbers up to 100.

Hint: You will only need to check divisibility for numbers up to and including $\sqrt{(n)}$, because any greater divisor will imply that there is a divisor less than this.

b) Make a function that instead finds the prime factorization of the input number. It should print "prime" and return nothing if the number is prime, and both print and return the factorization if it's composite. Find the prime factorization of 5525612.

c) Make test functions for the two functions above where you check for small values of n.

d) Compare the runtime of the two functions with the number 33425626272. Is the difference big? If so, why do you think one is faster than the other? The following code returns the mean time it takes for your program to run once:

```
import timeit
timeit.timeit('your_func(args)', \
    'from __main__ import your_func',number=1)
```

Filename: `prime.py`

Problem 4.7. Eulers totient function
Two numbers n and m are called relatively prime if they have no common divisors except for 1. That is, no number greater than one should divide both numbers with no residue.

a) Make a function that takes two numbers and returns true if they're relatively prime and false if they're not.

b) Euler's totient function is defined as

$$\phi(d) = \#\{\text{Numbers less than d which are relatively prime to d}\}.$$

Implement Eulers totient function and print $\phi(d)$ for $d = 10, 50, 100, 200$.

c) Make a test function for both a) and b).
Filename: `euler.py`

Problem 4.8. Simple Statistical Functions
In this problem you will implement two statistical functions and test them by comparing the results with statistical functions from the `numpy` module. We will trust that the functions from the `numpy` module are correct, and will use them as benchmark values in the test functions. When you import the `numpy` module you should follow the convention of renaming it `np`, as shown below.

```
import numpy as np
```

a) The mean of a set of numbers $x_1, x_2, x_3, ..., x_N$ is defined as

$$\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i,$$

where N is the size of the set. Implement a function `mean(x_list)` that returns the mean value of a list of numbers.

b) Make a test function `test_mean()` that tests the function from a). Compare the returned value with the result from `numpy.mean`. (Such that `expected =np.mean(x_test_values)`).

c) The standard deviation of a set of numbers $x_1, x_2, x_3, ..., x_N$ is defined as

$$s_N = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \overline{x})^2}.$$

Implement a function `standard_deviation(x_list)` which returns the standard deviation of a list of numbers. The mean value of the list will be necessary to calculate the relative deviation. Obtain the mean value inside the `standard_deviation` function by calling the function you made in a).

d) Make a test function `test_standard_deviation()` that tests the function from c). Compare the returned value with the result from `numpy.std`. (Such that `expected =np.std(x_test_values)`).

You may use the list below as an example for your test functions.

```
x_test_values = [0.699, 0.703, 0.698, 0.688, 0.701]
```

Filename: `statistics.py`

Problem 4.9. Münchhausen Numbers

A Münchhausen number is a number such that the sum of every digit to the power of itself equals the original number. E.g. $1^1 = 1$ is a Münchhausen number, and $5^5 + 3^3 + 2^2 = 3156 \neq 532$, so 532 is not.

Make a function that checks if a number is Münchhausen. Find a Münchhausen number different from one.

Hint: There is only one such number different from 1 and also under one million

Filename: `m_numbers.py`

Chapter 5

User Input and Error Handling

The exercises of this chapter are about user input and error handling. They correspond to Chapter 5 in the book by Sundnes and Chapter 4 in the book of Langtangen.

Problem 5.1. Quadratic with user input
Consider the usual formula for computing solutions to the quadratic equation $ax^2 + bx + c = 0$ given by

$$x_{\pm} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Write a program that asks the user to input values for a, b, and c, using `input` (or `raw_input` if you are using Python 2). Compute the corresponding solutions and print them.

Filename: `quadratic_roots_input.py`

Problem 5.2. Quadratic with command line
Modify the program from 5.1 such that a, b and c are read from the command line.

Filename: `quadratic_roots_cml.py`

Problem 5.3. Quadratic with exceptions
Extend the program from 5.2 with exception handling such that missing command line arguments are detected. In the `except IndexError` block, instead of exiting the program you should use `input` to ask the user for the missing input data.

Filename: `quadratic_roots_error.py`

Problem 5.4. Quadratic with raising Error Consider the program from Problem 5.1. Not all inputs yield real solutions. Modify the program such that it raises a `ValueError` if the input values for a, b and c yield complex roots. (That is if $b^2 - 4ac < 0$). Provide a suitable error message. Run your program with different values of the coefficients to verify that it prints out real roots and that a `ValueError` is raised when the roots are complex. An example of values that provide complex roots could be $a = 1$, $b = 1$, $c = 1$, while $a = 1$, $b = 0$, $c = -1$ will give real roots.

Filename: `quadratic_roots_error2.py`

Problem 5.5. Estimating harmonic series
Let $f(x)$ be the function

$$f(x) = \sum_{n=1}^{\infty} \frac{x^n}{n} = x + \frac{x^2}{2} + \frac{x^3}{3} + \dots$$

Write a program that approximates $f(x)$ (that is, evaluates $f_N(x) = \sum_{n=1}^{N} \frac{x^n}{n}$) with values of x and N given as command line arguments. Run the program for $x = 0.9$, $x = 1$, and $N = 10000$. Print the results.

Remark. For $x = 1$ this is known as the *harmonic series*. Despite the low values for large N, the series does not converge, but diverges very slowly. Try to run the program for different values of N to see how big you can get the value of $f(1)$.

Filename: `harmonic.py`

Problem 5.6. Estimating harmonic series extended
Using the program from Problem 5.5, consider the following values for x and N in a text file

```
x: 0.9 1
N: 500 1000 10 100 50000 10000 5000
```

a) Write a function to read a file containing information in the above format that returns two lists containing the values of x and N.

b) Write a test function for a) that generates a file in the given format and checks that the values returned by the function is correct.

c) Use the program from Problem 5.5 to evaluate $f_N(x)$ for the different values of x and N. Create a function that writes the information to a file in a table format with the first column containing the values of N in increasing order, and the second and third the values of $f_N(x)$ at 0.9 and 1 respectively.

Filename: `harmonic_table.py`

Problem 5.7. Read isotope file
Isotopes of a chemical element in its ground state have the same number of protons but differ in the number of neutrons. The weight of isotopes of the same chemical element will therefore be different.

The molar mass, M, of a chemical element, can be calculated by summing over all its isotopes $M = \sum_i m_i w_i$, where m_i is the weight of the i-th isotope and w_i the corresponding natural abundance.

The file `Oxygen.txt`, which is given below, contains the information on Oxygen's isotopes (^{16}O, ^{17}O and ^{18}O).

```
Isotope    weight [g/mol]    Natural abundance
(16)O          15.99491          0.99759
(17)O          16.99913          0.00037
(18)O          17.99916          0.00204
```

Write a script in Python to read the file `Oxygen.txt` and extract the weights and the natural abundance of all the isotopes of Oxygen. Use these to calculate the molar mass of Oxygen. Print out the result with four decimals and provide the correct units.

Filename: `read_file_isotopes.py`

Problem 5.8. A result on prime numbers

A famous result concerning prime numbers states that the number of primes below a natural number n, denoted $\pi(n)$, is approximately given by

$$\pi(n) \approx \frac{n}{\log(n)}.$$

That is, the fraction $p(n) = \pi(n)/\frac{n}{\log(n)}$ tends to 1 as $n \to \infty$. The following table contains the exact values of $\pi(n)$ for some values of n.

```
n: 10**20 10**4 10**2 10**1 10**12 10**4 10**6 10**15
pi(n): 2220819602560918840 1229 25 4 37607912018 168
       78498 29844570422669
```

a) Write a function that reads the file given above and returns two tuples containing sorted values of n and $\pi(n)$. It is important that the correspondence in the orderings are correct, that is, the same as in the table above.

b) Write a test function that generates a file with the format above and tests that the returned values are correct. It should test that the order of the elements are in correspondence as in the file.

Hint: The `==` *operator on tuples will take the order into account. The same operator on lists will not.*

c) Create a function that writes the values of n and $p(n)$ to a file in a table format in increasing order with the values of n in the first column and the corresponding values of $p(n)$ in the second column.

Bonus problem There are better approximations to $\pi(n)$, for example the function

$$\mathrm{Li}(n) = \int_2^n \frac{1}{\log(t)} dt$$

Approximate the integral for different values of n and modify the program to write these into a third column.

Hint: Implement an algorithm for approximating the integral (e.g. the trapezoidal rule) and compute the difference as before.

Filename: `primes.py`

Problem 5.9. Conversion from other bases
Recall that a binary number is a sequence of zeros and ones which converted to the decimal system becomes $\sum_i 2^i$ where i is a term in the sequence containing a 1 (e.g. $100101 = 2^5 + 2^2 + 2^0 = 37$).

a) Write a function that takes a binary number and converts it to a decimal number. If the argument is not a binary number, a message should be printed and nothing returned.

Hint: Let the number in the argument be of type string to avoid problems with numbers starting with a zero.

b) Let the binary number from a) be taken as a command line argument. Use exceptions (IndexError) to handle missing input. Print the conversion of 100111101.

c) Extend the program with a function to also handle numbers written in base 3.

Hint: An example of a ternary number(a number in base 3) converted to a decimal number: $1201 = 1 \cdot 3^3 + 2 \cdot 3^2 + 0 \cdot 3^1 + 1 \cdot 3^0$.

Filename: `base_conversion.py`

Problem 5.10. Read temperatures from two files
We consider data sets from the Norwegian Meteorological Institute, containing daily temperatures of any month of any year at Blindern (Oslo). There is one file for each month, and each file typically looks like this:

```
Year: 1997. Month: April. Location: Blindern(Oslo)
9.0  12.3    15.8  13.4  11.0 16.2  13.3
12.9  14.0  14.1  12.0  17.3  15.5 15.4
...
```

The observations are given chronologically, and the temperatures are given in degrees Celsius. There are no empty lines in the bottom of the file. Two example files, `temp_oct_1945.txt` and `temp_oct_2014.txt`, can be downloaded from the course website. The files contain daily temperature observations from October 1945 and October 2014, respectively.

a) Write a function `extract_data(filename)` that reads any such file and returns a list of the temperatures from the given month. Write a program that uses the function to read the two given files and store the temperatures in lists `oct_1945` and `oct_2014`. Calculate the average, maximum and minimum value of the temperatures of both months, and print the results. You may use the `numpy.mean()`, `numpy.max()` and `numpy.min()` methods to do the calculations.

b) Write a function `write_formatting(filename, list1, list2)`, which writes the data in `list1` and `list2` as two nicely formatted columns to a file with name `filename`. You can assume that the two lists have equal length. Finally, call the function such that the file `temp_formatted.txt` is created, using the lists `oct_1945` and `oct_2014`.
Filenames: `temp_read_write.py`, `temp_formatted.txt`

Problem 5.11. Why we test for specific exception types
The simplest way of writing a `try-except` block is to test for any exception. For example, consider the simplel program

```
import sys

try:
    C = float(sys.arg[1])
except:
    print("Please provide C as a command-line argument")
    sys.exit(1)

print(f"Successfully read the number {C} from the command-line")
```

Write or copy these statements into a program file and run it from the terminal. Try to run the program without providing a command-line argument, by providing a number, and by providing some random text. 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 test the program again. What is the problem now? Correct the program.

Filename: `unnamed_exception.py`

Chapter 6

Array Computing and Curve Plotting

The topics of this chapter are array-computing and plotting. The exercises correspond to Chapter 6 in the book by Sundnes and Chapter 5 in the book of Langtangen.

Problem 6.1. Fill arrays; loop version
We study the function

$$f(x) = \ln(x).$$

We want to fill two arrays `x` and `y` with the values of x and $f(x)$, respectively. Use 101 uniformly spaced x values in the interval $[1, 10]$. First create arrays `x` and `y` with the correct length, containing all zeros. Then compute and fill in each element in `x` and `y` with a `for` loop.

Filename: `fill_log_arrays_loop.py`

Problem 6.2. Fill arrays; vectorized version
Vectorize the code in Problem 6.1 by creating the x values using the `linspace` function from the `numpy` package and evaluating $f(x)$ with an array argument. Since the calculation should be vectorized, you should not use any form of loop in the code.

Filename: `fill_log_arrays_vectorized.py`

Problem 6.3. Plot the population growth
Again, we're considering a population undergoing logistic growth. The number of individuals in the population is given by

$$N(t, k, B, C) = \frac{B}{1 + C\mathrm{e}^{-kt}}.$$

Plot this function for $t \in [0, 48]$ with a carrying capacity $B = 50000$, $C = 9$ from the initial condition that we have 5000 individuals at $t = 0$ and a steepeness of $k = 0.2$.

Filename: `population_plot.py`

Problem 6.4. Oscillating spring

A rock of mass m is hung from a spring, and pulled down a length A. When released, the rock will oscillate up and down with a vertical position given by

$$y(t) = Ae^{-\gamma t} \cos\left(\sqrt{\frac{k}{m}}t\right),$$

Here, y is the vertical position of the rock, k is the spring constant, and γ is a friction coefficient representing air resistance. Set $k = 4$ kg s^{-2} and $\gamma = 0.15$ s^{-1}, $m = 9$ kg, and $A = -0.3$ m.

a) Create arrays `t_array` and `y_array` of size 101, both initially filled with zeros. Use a `for` loop to fill them with time values in the range from 0 to 25 seconds, and the corresponding $y(t)$ values.

b) Vectorize your program by using the NumPy's `linspace` function to generate the `t_array`, and send it into a function `y(t)` to generate the `y_array`. Your program should now be free of for loops.

c) Plot the position of the rock against time in the given time interval. Use the arrays from both exercise a) and b), and confirm that they give the same result. Put the correct units on both axes.

Filename: `oscillating_spring.py`

Problem 6.5. Plot Stirling's approximation
Stirling's approximation is

$$\ln(x!) \approx x \ln x - x.$$

a) Make two functions `stirling(x)` and `exact(x)`, returning Stirling's approximation and the exact value of $\ln(x!)$, respectively. Plot both the approximation and the exact curve in the same figure.

Hint: To implement a vectorized version of the `exact` *function, you can use scipy.special.gamma(x). This function is a "generalized factorial" which can find the "factorial" of float numbers. It works such that* $n! = \text{gamma}(n+1)$. *You can also just consider integer values and plot the value of* $\ln(x!)$ *for each integer* x *in the interval you're considering. Keep in mind that* `math.factorial` *is not vectorized.*

b) Use a `while` loop and find the minimal value of x for the relative error to be less than 0.1%.

Hint: Relative error is given as $(a - \tilde{a})/a$, *where* a *is the exact value and* $\tilde{a}$ *is the approximation. Also, do not start with* x *smaller than or equal to* 1, *why?*

Filename: `stirling_plot.py`

Problem 6.6. Plotting roots of a complex number
The n'th roots of a complex number $z = re^{i\theta}$ can be found by

$$\omega_k = \sqrt[n]{z} = \sqrt[n]{r}e^{\mathrm{i}\frac{\theta+2\pi\mathrm{k}}{\mathrm{n}}},$$

for $k = 0, 1, ..., n-1$. The roots can be rewritten to separate the real component x_k and the imaginary component y_k, such that $\omega_k = x_k + iy_k$. Through the relation between the exponential function and the sine functions we get

$$x_k = r^{\frac{1}{n}} \cos \frac{\theta + 2\pi k}{n}$$

and

$$y_k = r^{\frac{1}{n}} \sin \frac{\theta + 2\pi k}{n},$$

for $k = 0, 1, ..., n-1$.

a) Write a function that takes the angle θ, the radius r and the degree n of the roots as parameters. The function should calculate and return all of the n'th roots of a complex number $re^{i\theta}$, as two lists or arrays corresponding to the real part $x = x_0, x_1, ..., x_{n-1}$ and the complex part $y = y_0, y_1, ..., y_{n-1}$ of the roots. An example of a function call on the function you will write is given below.

```
x, y = roots(r, theta, n)
```

b) Consider the complex number $z = 10^{-4}e^{i2\pi}$. Use the function from a) to get all the roots of order $n = 6$, $n = 12$ and $n = 24$. Plot the roots as points, and plot all the three orders of roots in the same plot. Label the different orders of roots. And example of code for plotting the roots of order $n = 6$ is given below.

```
plt.plot(x_n_6, y_n_6, "o", label="n = 6")
```

Filename: `roots.py`

Problem 6.7. Fermi-Dirac distribution

The Fermi-Dirac distribution says something about the probability of an energy state being occupied by a particle, or more precisely a fermion, e.g. an electron. It is a function of energy and temperature given by

$$f(E, T) = \frac{1}{1 + e^{(E-\mu)/kT}}, \tag{6.1}$$

where E is energy, T is temperature, k is Boltzmann's constant and μ is the so-called chemical potential. Use $k = 8.6 \cdot 10^{-5}\text{eVK}^{-1}$ and $\mu = 4.74\text{eV}$ and make a program that visualizes the Fermi-Dirac distribution on the interval $E \in [0, 10]\text{eV}$ when $T = 0.1\text{K}$. (eV is a unit of energy, $1\text{eV} = 1.6 \cdot 10^{-19}\text{J}$.)

Filename: `Fermi_Dirac.py`

Problem 6.8. Animate the temperature dependence of the Fermi-Dirac distribution

Make an animation of the Fermi-Dirac distribution $f(E, T)$ from Problem 6.7 We're interested in studying how the distribution changes when we raise the temperature. Plot f as a function of E on $[0, 10]$ for a set of temperatures $T \in [0.1, 3 \cdot 10^4]$. Also make an animated GIF file. Remember to label your axes and include a legend to show the value of the temperature.

Hint: A suitable resolution can be 1000 *intervals (*1001 *points) along the* E *axis,* 60 *intervals (*61 *points) in temperature, and* 6 *frames per second in the*

animated GIF file. Use the recipe in Section 5.3.4 and remember to remove the family of old plot files in the beginning of the program.

Filename: `Fermi_Dirac_movie.py`

Problem 6.9. Bump functions

Consider the function

$$f(x) = \begin{cases} ke^{-\frac{1}{1-x^2}} & -1 < x < 1 \\ 0 & \textit{otherwise.} \end{cases}$$

a) Plot the function with $k = 1$ on the interval $-2 \leq x \leq 2$ by implementing a vectorized version in your program.

b) Animate the function on the same interval as above when k decreases from 1 to 0.

Filename: `bump.py`

Problem 6.10. Band structure of solids

Electrons in solids are waves. These waves have different wave lengths λ. Often, waves are characterised by their wave number $k = 2\pi/\lambda$, and the wave number is associated with the energy of the electron. The energies of electrons in solids have a band structure, i.e., there are different bands of energies separated by a band gap.

The file `bands.txt` contains k-values and corresponding energies for the three first bands of a solid. Have your program read the values for k and the energies and plot the energy bands as functions of k in the same figure. You will see that some energies can never be obtained by electrons in the solids. These areas of non-allowed energies are called the band gaps.

Filename: `band_structure.py`

Problem 6.11. Half-wave rectifier vectorized

In Problem 4.5, we implemented a function illustrating a sine signal after it had passed through a half-wave rectifier. Vectorize this function and plot $f(x)$ for $x \in [0, 10\pi]$.

Hint: The `numpy.where(condition, x1, x2)` function returns an array of the same length as `condition`, whose element number `i` equals `x1[i]` if `condition` is `True`, and `x2[i]` otherwise.

Filename: `half_wave_vec.py`

Problem 6.12. Singularity plot

In this problem we consider the function

$$f(r, \theta) = \left(e^{\frac{1}{r}\cos\theta} \cos\left(-\frac{1}{r}\sin\theta \right), -e^{\frac{1}{r}} \sin\left(-\frac{1}{r}\sin\theta \right) \right)$$

with $0.01 \leq r \leq 1$ and $0 \leq \theta \leq 2\pi$. Create arrays of r and θ values on the unit circle centered at the origin with n uniformly spaced values. Fix axes between -0.5 and 0.5 for x and y and visualize the function for $n = 10, 50, 100, 500$. You can use the following to generate the correct values for r and θ:

```
theta = np.linspace(0,2*np.pi,100)
r = np.linspace(0.01,1,100)
r, theta = np.meshgrid(r,theta)
```

Remark. If we had an ideal computer that could calculate every value in an interval and plot it, then the image we have plotted would touch every single value in the plane, except for at most one! In our program we have $0.01 < r < 1$. The remarkable thing is that the same is true if we replace the inequality with $0 < r < \epsilon$ for any $\epsilon > 0$. Not only that, but all those points are hit an infinite number of times!

Filename: `ess_sing.py`

Problem 6.13. Approximate $|x|$
For x in $[-\pi, \pi]$ the absolute value $|x|$ can be approximated by a sum

$$f(x, N) = \frac{\pi}{2} - \frac{4}{\pi} \sum_{k=1}^{N} \frac{\cos((2k-1)x)}{(2k-1)^2}.$$

As usual, the accuracy of the approximation increases with increasing N. Write a function `abs_approx(x,N)` that calculates the sum and of the sum and returns $f(x, N)$. Use the function to compute the approximation for $N = 1, 2, 3, 4$ and plot it against the exact function. Let the x-axis be $[-\pi, \pi]$ with a suitable y-axis.

Filename: `approx_abs.py`

Problem 6.14. Plotting graphs
A graph is a collection of lines and points in the plane such that each line connects two points. In this exercise we will create functions for plotting graphs on a set of points.

a) Make a function `plot_line(p1,p2)` that takes two points as arguments and plots the line between them. The two arguments should be lists or tuples specifying x- and y-coordinates, i.e., `p1 =(x1,y1)`. Demonstrate that the function works by plotting a vertical and a horizontal line.

b) A complete graph is a graph such that any two points has a line that connects them. Make a function `complete_graph(points)` that takes a list of points and plots the complete graph on those points. To verify that the function works, first choose the four corners of the square $((0,0),(1,0),(0,1),(1,1))$ and then the points $(1,0),(\alpha,\alpha),(0,1),(-\alpha,\alpha),(-1,0),(-\alpha,-\alpha),(0,-1),(\alpha,-\alpha)$, with $\alpha = \sqrt{2}/2$. The resulting complete graphs should look like the ones in Figure 6.1.

Hint: Modify the `plot_line` function from question a) so that it only calls `plot()` but not `show()`. The complete graph can then be drawn by looping over the points and calling `plot_line` for each pair, and finally calling `show()` after the loop.

Filename: `graph1.py`

Problem 6.15. Plotting graphs
Given a natural number n, make a function that plots the following graph:

- Two vertical rows with n points should be placed side by side.
- Each point on the left side should have a line to every point on the right side and vice versa.

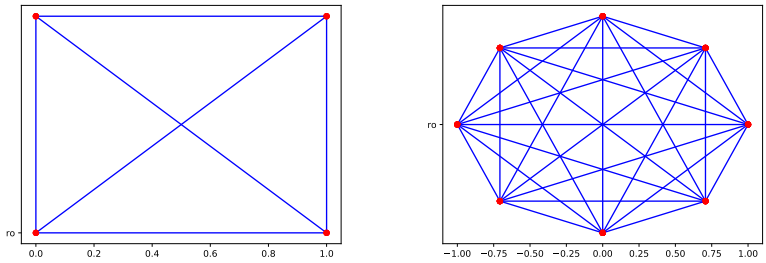

Figure 6.1: Examples of complete graphs from Problem 5.14 b). The left shows the graph on the corners of the unit square, the right is the graph for eight equally spaced points on a circle.

- No two points on the same side should be connected by a single line.

Filename: `graph2.py`

Problem 6.16. Inefficiency of primality checker
Consider the program from Problem 4.6. Use the `timeit` module and run the program to find the time it takes to find a factorization of an n digit number. Plot the time against the number of digits for the numbers in the file `prime_check.txt`. You can use the following code to time the function for different numbers:

```
str1 = "f(" + str(n) + ")"
str2= "g(" + str(n) + ")"
N = 100
time1 = timeit.timeit(str1, 'from __main__ import f',number=N)
time2 = timeit.timeit(str2, 'from __main__ import g',number=N)
```

Filename: `prime_ineff.py`

Problem 6.17. Animating a cycloid
One may create a curve by placing a circle on the x-axis, fixing a point on the circle, and then drawing the trace of the point as the circle is rolling. The resulting curve is called a cycloid. In mathematical language it is given as

$$r(\theta) = [R(\theta - \sin\theta), R(1 - \cos\theta)]$$

where R is the radius of the rotating circle and θ is the angle starting at 0 and increasing.

a) Animate the cycloid as a function of θ starting at 0, ending at 15. Draw a point at the end of the cycloid that varies with the animation.

Hint: A point can be added through a new plot using for example
`point, =axes.plot([],[],'o')`
and updating during the animation.

b) Add the rolling circle defining the cycloid to the plot. You may use that at a given time θ, the circle is given as $s(\theta) = (R \cdot \theta + \cos\theta, R + \sin\theta)$.

Filename: `cycloid.py`

Problem 6.18. Calibration curve

A tool in chemical analysis for measuring the concentration of a substance in a sample (e.g. blood or urine), is making a calibration curve. Solutions with known concentrations of a substance (standard solutions) are measured. The X-axis is the concentrations of the standard solutions, and the Y-axis is for the measured intensity of these solutions. The concentration of the substance in the sample can then be determined using an equation that best describes the calibration curve. This equation is determined using linear regression.

In Python, you can use the `numpy` module to obtain a linear regression. How to do this will be shown.

Assume that you have made five standard solutions with concentrations of 10, 20, 30, 40 and 50 mg/L of the substance you wish to test for. You have used equipment that detects this substance, and noted down the intensity for each of your standard samples. The code below shows how the linear regression can be performed.

```
# standard concentrations and height of their signals
I_stand = [9.19, 19.8, 27.0, 34.7, 44.9]
conc_stand = [10, 20, 30, 40, 50]

# linear regression
fit = np.poly1d(np.polyfit(conc_stand, I_stand, 1))
conc_curve = np.linspace(0, 60, 100)
signal_curve = fit(conc_curve)
```

a) Plot the calibration curve as a line, and plot the intensities of the calibration solutions as points on top of the calibration curve.

The function created in the code above, `fit(x)`, is now a linear function on the form $f(x) = ax + b = y$, where x corresponds to the concentration and y will correspond to the intensity of the signal at the given concentration.

b) Determine a and b by using `fit(x)` and use these to implement an inverse function of $f(x)$ such that $g(y) = x$. Use your function to calculate the concentration of three different samples of the same unknown compound. The intensities of the compound of the samples are given below.

```
I_unknown = np.array([19.9, 20.1, 19.8])
```

Print out the mean value, with the uncertainty $(\overline{x} \pm s_N)$, of the samples of the unknown compound. You may use `np.mean` and `np.std` to find the mean value and the uncertainty.

Filename: `calibration.py`

Chapter 7

Dictionaries and Strings

This chapter contains exercises on programming with dictionaries and strings, corresponding to Chapter 7 in the book by Sundnes and Chapter 6 in the book by Langtangen.

Problem 7.1. A result on primes "dictionarized"
Consider the program from Problem 5.8. Since the entries correspond to each other, working with two seperate lists is cumbersome. We may avoid that using dictionaries. Modify the program such that the values are saved in a dictionary instead of a list. Let the values of n be keys with values $\pi(n)$.

Filename: `primes_dict.py`

Problem 7.2. Chemical elements in a dictionary
Consider the dictionary `elements_10` consisting of the 10 first chemical elements of the periodic table:

```
elements_10 = {1: '-', 2: 'Helium', 3: 'Lithium',
         4: 'Beryllium', 5: 'Boron', 6: 'Carbon',
         7: 'Nitrogen', 8: '-',
         9: 'Fluorine', 10: 'Neon'}
```

a) The chemical elements of number 1 (Hydrogen) and 8 (Oxygen) are missing. Copy `elements_10` into your file, and adjust the dictionary such that the keys 1 and 8 have their correct value. Use the technique as in this example:

```
dictionary[key] = 'value'
```

b) Copy the following code into your script, and run the file in your terminal. Find the difference between the two dictionaries that are printed, and explain why they are different from each other.

```
elements_10_copy = elements_10.copy()
elements_10_copy.update({11: 'Sodium'})
print(elements_10)
print('\n')
```

```
elements_11 = elements_10
elements_11.update({11: 'Sodium'})
print(elements_10)
```

Filename: `chemical_elements_dict.py`

Problem 7.3. Representation of polynomials
Let $f(x) = \sum_{i=0}^{n} a_i x^i$ and $g(x) = \sum_{j=0}^{m} b_m x^m$ be two polynomials. Recall that a polynomial can be expressed as a dictionary with keys equal to the degree of a term, and the corresponding coefficient as value (so $3x^2 + 1/2$ is represented by the dictionary $\{2 : 3, 0 : 1/2\}$).

You will be asked to implement three functions in this exercise. Check that each of your functions work as expected by creating two different polynomials represented as dictionaries, call your functions and print the returned values.

a) Create a function that takes two dictionaries (corresponding to two polynomials f and g) as arguments and returns a dictionary corresponding to the sum of the two.

b) Create a function as above that returns the dictionary corresponding two the product of two polynomials.
Hint: $fg = \sum_{k=0}^{n+m} c_k x^k$ *where* $c_k = \sum_{i+j=k} a_i b_j$

c) Add a function that evaluates a polynomial dictionary at a point.
Filename: `poly_dict.py`

Problem 7.4. Use string operations to create a pretty dictionary
The file `atm_moon.txt`, which can be downloaded from the course website, contains information about the composition of elements in the lunar atmosphere during nighttime. The values are given in particles per cubic centimetre.

Write a function that reads such a file, and returns a dictionary with the name of the elements as keys and the particle density as value. Transform all characters to their upper case equivalent. Strip off leading and trailing whitespaces in each of the string keys. Remove all the commas that mark every three digits, and then convert these values to float numbers.

For example, considering the information `' Neon 20 -40,000 '` extracted from the file, the dictionary element with key `'NEON 20'` should look like this:

```
'NEON 20': 40000
```

Filename: `atm_moon.py`

Problem 7.5. Interpret output from a program
The program `approx_derivative_sine.py` calculates an approximation to the derivative of $\sin\left(\frac{\pi}{3}\right)$ by the expression

$$\frac{\partial f}{\partial x} \approx \frac{f(x + \Delta x) - f(x)}{\Delta x},$$

for decreasing values of Δx. Direct the output of the program to a file (by `python approx_derivative_sine.py > filename`). Write a function that reads the file and returns three arrays consisting of numbers corresponding to

`delta_x`, `abs_error` and `n`. Plot `delta_x` and `abs_error` versus `n`. Use a logarithmic scale on the y axis. Explain why the absolute error increases after $n = 8$, i.e. after `delta_x` $= 10^{-8}$.

Hint: The function `semilogy` *is an alternative to* `plot` *and gives logarithmic scale on the y axis.*

Filename: `plot_round_off_error.py`

Problem 7.6. Saving information in a nested dictionary
The file below contains information about various people. The first column is the name, the second is the age, and the third is the gender.

```
John, 55, Male
Toney, 23, Male
Karin, 42, Female
Cathie, 29, Female
Rosalba, 12, Female
Nina, 50, Female
Burton, 16, Male
Joey, 90, Male
```

a) Write a function `read_person_data(filename)` that reads such a file and returns the information as a nested dictionary. For example the key 'John' has the dictionary {'Age': 55, 'Gender': 'Male'} as value. The keys in the main dictionary should be the name without the comma, i.e. "John", not "John, ".

b) Write a function `write_person_data(data_dict, filename)`, where the first argument is a nested dictionary like the one created in a), and the second argument is a file name. The function shall write the information in the dictionary to the specified file, in the format outlined above.

Filename: `people_dict.py`

Problem 7.7. Finding the frequency of words in a text

a) Write a function that reads the file *RandomWords.txt* and finds the frequency of words of length n. Save the information in a dictionary with the length as keys and the number of words of that length as values. You may assume that all words are separated by spaces and that only punctuation marks appear in the text.

Hint: For your program to be compatible with words of any length, it might be helpful to use defaultdict imported from collections. See page 339 in A Primer on Scientific Programming with Python*, by H. P. Langtangen. Use the function dict() on such an object to convert it to an ordinary dictionary*

b) Write a test function that generates a file of words and checks that the function returns the correct values.

Filename: `word_length.py`

Problem 7.8. The Euler's polyhedron formula
Let V, E, and F be the number of vertices, edges and faces in any polyhedron,

respectively. Then, Euler's polyhedron formula tells us that

$$V - E + F = 2.$$

In this exercise we shall check that the formula works for some given polyhedrons in a file with this setup:

```
Polyhedron: cube
vertices: 8 edges: 12 faces: 6

Polyhedron: pyramid
vertices: 5 edges: 8 faces: 5

...
```

a) Write a function that can read such a file, and returns a dictionary with the type of the polyhedron as a key, and a dictionary containing vertices, edges and faces as a value. The function shall strip of leading and trailing whitespaces in all strings. For example, the value of the key `'cube'` in the dictionary should look like this:

```
'cube': {'vertices': 8, 'edges': 12, 'faces': 6}
```

Note that you must convert the string numbers to integers, as for example 8, not '8'. Be aware of the fact that the value of each polyhedron in the dictionary is again a dictionary. Print the (nested) dictionary that is returned when reading the file `polyhedrons.txt`.

b) Write a test function `test_polyhedron_formula()` that checks that Euler's polyhedron formula works for the polyhedrons given in `polyhedrons.txt`. *Hint: Repeated indexing works for nested dictionaries as for nested lists. Below is an example of how to access the value of the key `'vertices'` inside the value of the key `'cube'`.*

```
cube_vertices = polyhedrons_dict['cube']['vertices']
```

Filename: `polyhedron_formula.py`

Problem 7.9. Compute digital roots

Given a number, say 5282, we can compute the sum of the digits. In this case $5 + 2 + 8 + 2 = 17$, and doing this again gives $1 + 7 = 8$. The one digit number we get by doing this is called the digital root of the number.

a) Make a function that calculates the digital root of a number.
Hint: Convert the number to a string in order to work with it.

b) Plot the digital root of numbers up to 500 with the digital root on the x-axis and the frequency of digital roots on the y-axis. Use `plt.scatter(x,y)` for the plot.

Filename: `dig_root.py`

Problem 7.10. Timezone converter

In the file `timezones.txt` you will find places and their timezone in GMT format.

a) Make a function that reads the file and saves the information in a dictionary.

b) Create a function that takes local Norwegian time (GMT +1) in the string format 'ddmmyy-hhmm', a place, and returns the local time at that place. Your program should display a message to the user if a place that is not saved in the dictionary is used. Do the following conversions:

- March 21st 2018 05.34 in Vancouver
- December 31th 2017 20.03 in Sydney
- January 1st 2018 00.15 in London

Filename: `timezones.py`

Chapter 8

Introduction to Classes

Thhe topic of this chapter is programming with classes and objects. These topics are important foundations for object-oriented programming (OOP), which is introduced later. The topics of the exercises are covered in Chapter 8 in the book by Sundnes and Chapter 7 in the book by Langtangen.

Problem 8.1. Saving information in a class
In this problem you will create a class that holds the same type of information as the dictionary in Problem 7.6.

a) Create a class `Person` where the constructor takes name, age, and gender as arguments, and stores them as attributes.

b) Add methods to the class for changing a persons name, age, and gender.

c) Add a method `__str__` that returns a string with all the information of that person. Create an instance of the class, using the information for 'John' in the table from Problem 7.6. Change the name and gender of John. Print the information of the instance before and after changing.

Filename: `class_people.py`

Problem 8.2. Right triangle class

a) Make a class `RightTriangle` that represents a right triangle. The constructor `__init__` should take two numbers a and b as arguments and store them as attributes. These are the lengths of the two legs (shortest sides) that define the right triangle. The constructor should also calculate and store the hypotenuse as an attribute in the class.

Remember that the hypotenuse c relates to the two short sides a and b by the Pythagorean Theorem:

$$a^2 + b^2 = c^2$$

b) Make an object `triangle1` of the class `RightTriangle` with both short sides equal to 1. Make another object `triangle2` with sides equal 3 and 4. Check that your implementation is correct by printing the hypotenuse of both objects. Use the usual `object.variable` convention to get the hypotenuse.

c) To make a robust program, we often want to code it such that it prevents being used in ways that does not make sense. In our case, a natural thing to prevent is making a triangle with sides having negative length, since length is strictly positive.

Make changes to the class' constructor such that a `ValueError` is raised if someone tries to make a triangle with sides of negative length. To test that your class raises the error correctly, you can test it with the following code:

```
def test_RightTriangle():
    success = False
    try:
        triangle3 = RightTriangle(1,-1)
    except ValueError:
        success = True
    assert success
```

d) Add a method `plot_triangle()` to your class which plots the triangle when you call it. The corner where the two shortest sides meet should be in origin. Side a should be along the x-axis, and side b along y. In order to plot, you might want to find out what the coordinates of each corner must be. You can use `plt.axis("equal")` to make the axes of the plot of equal length so that the triangle gets the right shape. Call the plotting method on the instance `triangle2` that you created in b).

Filename: `right_triangle.py`

Problem 8.3. Make a function class In this problem you will implement a `class F` which represents the function

$$f(x; n, m) = \sin(nx)\cos(mx).$$

Create the class and let n and m be parameters of the constructor, which must be stored as class attributes.

Since the class represents a function, it should be a callable class. An instance of a callable class can be called on like a function. The special method for creating a callable class is `__call__(self, args)`. Implement the special method `__call__(self, x)` such that it returns the value of the function evaluated at `x`.

Create two different instances `u` and `v` of `class F`. Choose values n and m for both the instances. Plot the two instances `u` and `v` evaluated in x against each other on the interval $x \in [0, 2\pi]$. In other words, plot `u(x_values)` on the x-axis and `v(x_values)` on the y-axis. Make sure to have enough points on the interval to ensure that the line is smooth.

Filename: `F.py`

Problem 8.4. Extending the `BankAccountP` class
Modify the class `BankAccountP` in the book to include a method `transfer` that transfers an amount between two accounts. The method should take an amount and the account you want to transfer to as arguments. Write a test function that checks that the methods `deposit`, `withdraw`, `transfer` and `get_balance` works properly.

Filename: `BankAccountP.py`

Problem 8.5. Approximating the square root of two
The square root of two can be represented by a so called *continued fraction* on the following form:

$$\begin{aligned}\sqrt{(}2) &= 1 + \frac{1}{1+\sqrt{(}2)} \\ &= 1 + \frac{1}{2+\frac{1}{1+\sqrt{(}2)}} \\ &= 1 + \frac{1}{2+\frac{1}{2+\frac{1}{1+\sqrt{(}2)}}}\end{aligned}$$

In this exercise we will exhibit two possibilities for approximating the number $\sqrt{(}2)$.

a) Make a class `Square` with a method `approx_frac` that takes an integer n, an initial value and returns the first n fractions as above with initial value x_0. This can be done by iterating the function

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

starting at x_0. For $n = 2$ and $x_0 = 1$ this gives

$$f(f(x_0)) = 1 + \frac{1}{2+\frac{1}{1+1}}.$$

b) Another way to approximate the square is by iterating the function $f(x) = \frac{1}{2}\left(x + \frac{2}{x}\right)$. Add a method `approx_iter` that takes a number x_0, an integer n, and returns the value of the function at x_0 iterated n times. For $n = 2$ we would have $f(f(x_0))$. From here on we assume for simplicity that $0.1 \leq x_0 \leq 2$.

c) Create a method that returns a nicely formatted table with the two approximations and their difference ϵ along with the exact value for $n = 1, 2, 5, 10$. Run the program, which approximation is best?

d) To visualize the approximation plot the exact value as a line in the plane and the two approximations as points (n, y_n), where y_n is the approximation. Use $n = 1, 2, 5, 10$.

Filename: `square_iteration.py`

Problem 8.6. Tangent lines on a quadratic curve
Consider a quadratic polynomial on the form $f(x) = x^2 + bx + c$. At a point x_0 the tangent line is given by $l(x) = (2x_0 + b)x + C$ where $C = f(x_0) - (2x_0 + b)x_0$.

a) Make a class `Quadratic` with a function $f(x)$ (a quadratic polynomial as above) as initial argument. Make a method that computes the tangent at a point and returns the function $l(x)$.

Hint: You will need to extract the coefficients $b = f(1) - f(0) - 1$ *and* $c = f(0)$.

b) Create a method that plots the function along with its tangent at a point.

c) Make a method that animates the tangent line moving over the curve $f(x)$. Make the animation for uniformly distributed x values in the interval $-5 \leq x \leq 5$. Test the program with the function $f(x) = x^2$.

Filename: `quadratic_tangents.py`

Problem 8.7. Numerical approximations of the derivative
Let $f(x)$ be a function and $f'(x)$ its derivative. There are many ways to approximate the derivative, for instance:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h},$$
$$f'(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}.$$

a) Make a class `Diff` with a constructor that takes a function f as argument, and three methods `diff1`, `diff2`, and `diff3` that approximate the derivative using the above formulas.

b) Create an instance of the class `Diff` using $f(x) = \sin(2\pi x)$. Visualise the difference in accuracy of the three methods for computing the derivative by plotting the approximate derivatives in the same window as the exact derivative $f'(x) = 2\pi\cos(2\pi x)$. Let x be on the interval $x \in [-1, 1]$, and plot with $h = 0.9, 0.6, 0.3, 0.1$ to see how the accuracy improves.

Filename: `class_diff.py`

Problem 8.8. Visualizing functions
For a function $f(x)$ we can plot the graph of the function as points $(x, f(x))$. This results in a curve in the plane. Suppose we have a function

$$f(x, y) = (u(x, y), v(x, y)).$$

The graph of this function lives in four dimensions and is not easily visualized. On way to visualize these functions is to instead of looking at the graph we look at how f act on points. For example, how the grid lines in the plane look after f is applied.

a) First we consider a specific function $f(x, y) = (x^2 - y^2, 2xy)$. Write a program where you define the function f and make a figure with x and y axis from -2 to 2 where you plot a number of the grid lines in x and y direction in the same plot. You will need around 15 lines in each direction. Make a another plot side-by-side in the same figure of all points $(x^2 - y^2, 2xy)$ where x and y are the points in the first plot.

b) To make the construction more flexible, modify your program to be a class `Visualize` taking a function $f(x, y) = (u(x, y), v(x, y))$ as initial argument. It should contain a method `grid(self, n)` that generates two plots, one of grid lines, and one of the image as in a).

c) We used grid lines of the plane to see how the function f behaved. We could have used any curves in the plane. Extend the class with a method `circ` that instead of using points corresponding to grid lines, uses circles with expanding radii. Let the axes go from -5 to 5 and the radii be uniformly distributed between 0 and 10 (15 circles should be sufficient). Test with the function $f(x, y) = (x^2 - y^2 + x + 1, 2xy + y)$. The second plot should consist of circular like objects with a self-crossing.

d) Add a method `grid_anim` that shows an animation of the image of the functions

$$f_\epsilon(x, y) = [(1 - \epsilon)x + \epsilon u(x, y), (1 - \epsilon)y - \epsilon v(x, y)]$$

where ϵ varies from 0 to 1.

e) Using the functions

$$f(x, y) = (x^2 - y^2, 2xy) \quad \text{and} \quad g(x, y) = (x^2 - y^2 + x + 1, 2xy + y),$$

test the `grid` and `grid_anim` methods on f, and the `circ` method on g. Use 15 gridlines and 15 circles.

Remark. For the student familiar with complex numbers, this is exactly how one would visualize a complex function $f(z)$. In our case we can use this for any function $f(x, y)$, but we usually restrict ourself to look at functions corresponding to certain complex functions, namely the differentiable ones.

Filename: `plot_functions.py`

Problem 8.9. A class for coordinates

This exercise will focus on how to implement special methods. You will implement the class `Coords`, which represents coordinates in three dimensions.

Hint: the class for complex numbers described in "A Primer on Scientific Programming with Python", by H.P. Langtangen, is of similar nature to the class you should implement in this problem.

a) Create the class `Coords`. Start by implementing the special methods `__init__(self, args)` and `__str__(self)`. The constructor should take three parameters, x, y and z. The string representation of the class should be on the form (x, y, z).

The implementation should be such that the code below works.

```
sqrt3 = sqrt(3)
close = Coords(1/sqrt3, 1/sqrt3, 1/sqrt3)
far = Coords(3/sqrt3, 15/sqrt3, 21/sqrt3)

print(close)
print(far)
```

Output:

```
(0.58, 0.58, 0.58)
(1.73, 8.66, 12.12)
```

b) Implement the special methods `__len__(self)` and `__abs__(self)`. The length of coordinates should always be 3, as the coordinates will be in three dimensions. The absolute value should yield the Euclidean norm (or the physical length in space), which is given by

$$||(x, y, z)|| = \sqrt{x^2 + y^2 + z^2}.$$

The implementation should be such that the code below works.

```
print(f"The class represents coordinates in {len(close)} dimensions")
print(f"The distance from the centre to the point close is {abs(close)}")
print(f"The distance from the centre to the point far is {abs(far)}")
```

Output:

```
The class Coords represents coordinates in 3 dimensions

The distance from the centre to the point close is 1.00
The distance from the centre to the point far is 15.00
```

c) Implement the special methods `__add__(self, other)` and `__sub__(self, other)`. When adding or subtracting two objects of class `Coords`, a new object of class `Coords` should be returned.

The object returned when adding two coordinates should have the coordinates

$$x_{new} = x_{self} + x_{other}$$
$$y_{new} = y_{self} + y_{other}$$
$$z_{new} = z_{self} + z_{other},$$

and similarly should the method for subtracting should return an object of `Coords` with coordinates at

$$x_{new} = x_{self} - x_{other}$$
$$y_{new} = y_{self} - y_{other}$$
$$z_{new} = z_{self} - z_{other},$$

The implementation should be such that the code below works.

```
further = close + far
print(f"The coordinates further are at {further}")

distance = abs(far - close)
print(f"The distance from far to close is {distance}")

centre = further - further
print(f"The coordinates at the centre are {centre}")
```

Output:

```
The coordinates further are at (2.31, 9.24, 12.70)
The distance from far to close is 14.14
The coordinates at the centre are (0.00, 0.00, 0.00)
```

Filename: `Coords.py`

Chapter 9

Object-Oriented Programming

The topic of this chapter is object-oriented programming (OOP), which is covered in Chapter 9 of the book by Sundnes and the book by Langtangen.

Problem 9.1. Implement Newtons method

a) Make a subclass `Function` of the class `Diff` in problem 8.7 that takes a function f as an initial variable. It should contain a method such that the following code is compatible with your program and prints the value of f at 2.

```
def f(x):
    return x**2
func = Function(f)
print(f(2))
```

b) We would like the class to give estimated values for roots of f. That is, points such that $f(x) = 0$. To do this we implement Newton's formula. It is given recursively as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

where we give a starting point x_0. In some cases(not all) x_n will approach a root of f. Implement this in a method `approx_root` that takes a starting point and a bound $\epsilon < 1$ as arguments and approximates x_n such that $f(x_n) < \epsilon$.

Hint: Implement a simple convergence test. Check that $f(x_n) < 1$ after 100 iterations. If not terminate the loop and inform the user that there is no convergence for that starting point. It is still a possibility for convergence, but unlikely.

c) Test the program with the function $f(x) = x^2 - 1$ and starting value 5 with bounds 10^{-i}, for $i = 1, 2, 3, 4, 5, 6$. Print the approximated value for x, $f(x)$ and the bound in a table format. Try to run the program with starting value 0. What happens, can you see why?

Filename: `newton.py`

Problem 9.2. Implement Polynomials as a Class

a) Make a class `Quadratic` that implements second order polynomials. An object of `Quadratic` should be initialised with a list containing the coefficients. Add a `__call__(self,x)` method that evaluates the polynomial at a point `x`, and a `__str__(self)` method for printing the polynomial.
Create an instance of `Quadratic` with coefficients $(1, 3, 2)$. Print it and evaluate it at $x = 1$, $x = 2$ to make sure it works.

b) Make a subclass `Cubic` of the class `Quadratic` that implements third order polynomials. You should make use of inheritance to extend the class `Quadratic` you made in the previous exercise. Implement a method `derivative` for `Cubic`. The method should return an object of type Quadratic that corresponds to the derivative.
Create an instance of `Cubic` with coefficients $(1, 3, 2, 4)$. Print it and evaluate it at $x = 1$, $x = 2$. Also call `derivative` and print the result.

Filename: `polynomial.py`

Problem 9.3. Vectors

a) Make a class Vector2D that implements 2D-vectors(two components). Add `__add__` and `__sub__` methods so you can add and subtract vectors. Remember that vectors add and subtract element-wise, e.i.:

$$(1, 2) + (4, 5) = (5, 7)$$

b) For two vectors $\vec{a} = (a_1, a_2)$ and $\vec{b} = (b_1, b_2)$, the dot product is defined as

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2$$

Implement a method `dot` that calculates the dot product of two vectors.
Make two vectors $\vec{v} = (1, 2)$ and $\vec{w} = (-2, 5)$ from Vector2D. Print $\vec{v}$, $\vec{w}$, $\vec{v} + \vec{w}$, $\vec{v} - \vec{w}$ and $\vec{v} \cdot \vec{w}$

c) Make a subclass Vector3D of the class Vector2D, where Vector3D is extended with an additional coordinate. Make sure all the methods are updated to work with three coordinates. The dot product for 3D vectors extends as one would expect:

$$\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

Use inheritance to reuse old code as much as possible(especially for the dot product method).
Make two vectors $\vec{v} = (1, 2, 4)$ and $\vec{w} = (-2, 5, 1)$ from Vector3D. Print $\vec{v}$, $\vec{w}$, $\vec{v} + \vec{w}$, $\vec{v} - \vec{w}$ and $\vec{v} \cdot \vec{w}$

d) There is a common vector operation that is defined for 3D vectors, but not for 2D vectors. This is the cross product. It differs from the dot product in that the result is not a number, but a new vector:

$$\vec{c} = \vec{a} \times \vec{b}$$

where $\vec{c} = (c_1, c_2, c_3)$. The cross product is define as

$$\begin{aligned} c_1 &= a_2 b_3 - a_3 b_2 \\ c_2 &= a_3 b_1 - a_1 b_3 \\ c_3 &= a_1 b_2 - a_2 b_1 \end{aligned}$$

Implement a method `cross` for Vector3D only that calculates the cross product. Make two vectors $\vec{v} = (2, 0, 0)$, $\vec{w} = (0, 2, 0)$ from Vector3D. Print $v \times w$.
Filename: `vector.py`

Problem 9.4. Inheritance
In this exercise we will investigate how python handles inheritance by making some intuitive classes.

a) Begin by making a class `Mammal`. Add a method `info(self)` that returns a string stating something that is common to all mammals, for instance "I have hair on my body". Also add a method `identity_mammal(self)` that prints (not returns) "I am a mammal".

b) Make a subclass Primate of the class Mammal. Add a method `info(self)` that returns the same as `info(self)` for Mammal, in addition to something specific for Primates, for instance "I have a large brain". Use inheritance to include the general string from `Mammal`. Do not copy-paste it into the Primate class. Also add a method `identity_primate(self)` equivalent to `identity_mammal(self)`.

c) Now make two subclasses Human and Ape from Primate. Update the `info(self)` method in the same manner for both Human and Ape, and also add their respective identity methos.

Make an instance John of the class Human, and an object Julius of the class Ape. Try calling `info()`, `identity_mammal()`, `identity_primate()`, `identity_human()` and `identity_ape()` for both John and Julius. Does it work as you expect? Some of these calls are meant to cause an error.

d) Python is able to check if an object is of a particular class with the function `isinstance`. `isinstance(object_name, class_name)` returns `True` if the object `object_name` is of class `class_name`. An example could be `isinstance(John, Primate)`, which returns `True` if John is of the class Primate.

Use `isinstance` to check if John is Mammal, Primate, Human and Ape. Do the same for Julius.

Filename: `inheritance.py`

Appendix A

Sequences and Difference Equations

The exercises of this chapter correspond to Appendix A in the book by Langtangen, and cover programming of difference equations.

Problem A.1. Computing Bell numbers
Let $B_0 = B_1 = 1$, the n'th *Bell number* is defined recursively as

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k.$$

Make a function that returns the n first Bell numbers and print the first 10.
Filename: `bell.py`

Problem A.2. Solve a difference equation numerically
We study the difference equation

$$x_n = x_{n-1} + x_{n-2}.$$

Write a program that writes out the first 15 elements of the sequence for $x_0 = x_1 = 1$.
Filename: `fibonacci.py`

Problem A.3. The spreading of a disease
We want to study the spreading of a disease. Assume that people recover at a rate such that a ratio a of the people that are sick this week will still be sick next week. It takes two weeks from when you get infected until you become sick, and a person who is sick will on average infect b people each week, who then become sick two weeks later.

Let x_n be the number of sick people in week n. The number of sick people is then given by the following difference equation

$$x_n = ax_{n-1} + bx_{n-2}.$$

a) Write a function `disease_weeks(a, b, x0, x1, N)` that calculates the number of people that are ill with a given disease (defined by values a and b) and returns an array/list containing the number of sick people form the initial week to week N. (In other words, the function should return an array or a list with $x_0, x_1, ..., x_N$).

Let $x_0 = 100$, $x_1 = 150$ and $a = 0.25$. Use the function calculate the number of sick people in an array up to week $N = 15$ for both $b = 0.5$ and $b = 0.75$. Plot both scenarios in the same plot. Remember to use labels on the curves.

b) We do not need to store all the $N + 1$ values. Since x_n only depends on x_{n-1} and x_{n-2}, these are the only values we need to store. Write a function `disease_week_N(a, b, x0, x1, N)` that does not use any lists or arrays, and returns only the number of sick people in week N, x_N. Use this function to obtain the number of sick people in week $N = 15$ for both the cases you plotted in a). Print out the results, and remember to include information about which of the two cases you are printing. Verify that the result is the same as obtained using arrays.

Filename: `disease.py`

Problem A.4. Finding π with Newton's method

It is common knowledge that $\pi \approx 3.14$, but in this exercise you will use Newton's method and knowledge of the sine function to improve an approximation of π.

Newton's method can be written as a difference equation defined

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})},$$

and is used to find roots of $f(x)$, based on the function $f(x)$, the derivative of the function $f'(x)$ and an initial guess x_0.

Consider the function $f(x) = \sin(x)$. Finding the functions derivative $f'(x)$, should be trivial. This function has infinitely many roots, which we know are located at $x = k\pi$, where k is an integer. This exercise will focus on the root for $k = 1$, and use Newton's method to approximate π. For Newton's method to converge towards the correct root, the initial condition, x_0, needs to be set close to the value of π.

Set $x_0 = 3.14$ and calculate x_1 and x_2 following Newton's method. Print out x_0, x_1 and x_2, as well as the value of `numpy.pi` with 13 decimals. You should print them in a tidy way such that the values are easy to compare. If Newton's method was used correctly, your values for π should improve!

Filename: `finding_pi.py`

Problem A.5. Find difference equations for computing $\ln x$

The Taylor expansion of $\ln x$ is

$$\ln x = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(x-1)^n}{n},$$

for $x \in (0, 2]$.

We can define the sum as

$$\ln x \approx S(x; n) = \sum_{j=0}^{n} (-1)^{j+1} \frac{(x-1)^j}{j},$$

so that $S(x,n) = \sum_{j=1}^{n} a_j$ and

$$a_j = -\frac{(j-1)}{j}(x-1)a_{j-1}.$$

Introduce $s_j = S(x, j-1)$ and a_j as the two sequences to compute. We have the initial values $s_1 = 0$ and $a_1 = (x-1)$.

a) Find the set of difference equations for s_j and a_j.

Hint: You can find an example on how this is done for e^x in section A.1.8 in the book.

b) Implement the system of difference equations in a function `ln_Taylor(n, x)` which returns s_{n+1} and $|a_{n+1}|$. The term $|a_{n+1}|$ is the first neglected in the sum and may act as a rough estimate of the size of the error in the Taylor polynomial approximation.

c) Verify the implementation by computing the difference equations for $n = 3$ by hand and comparing with the output from the `ln_Taylor` function. Automate this comparison in a test function.

d) Check that the accuracy of the Taylor polynomial improves as n increaces and x is close to 1. What happens when $x > 2$?

Filename: `ln_Taylor_series_diffeq.py`

Problem A.6. Lotka-Volterra two species model

We have previously studied the logistic model for poulation growth, which describes the growth of a population in the abscence of predators. The Lotka-Volterra model describes interactions between two species in an ecosystem, a predator and a prey. We will in the following take the preys to be rabbits and the predators to be foxes. The number of rabbits and foxes in week n is denoted by R_n and F_n respectively, and the population is modelled by the equations

$$R_{n+1} = R_n + aR_n - cR_nF_n$$
$$F_{n+1} = F_n + ecR_nF_n - bF_n,$$

where a is the natural growth rate of rabbits in the absence of predation, b is the natural death rate of foxes in the absence of food (rabbits), c is the death rate per encounter of rabbits due to predation and e is the efficiency of turning predated rabbits into foxes.

Write a program that computes the number of rabbits and foxes up to $n = 500$. Use $a = 0.04$, $b = 0.1$, $c = 0.005$ and $e = 0.2$. In the begining we have $R_0 = 100$ and $F_0 = 20$. Plot how the number of individuals in the populations vary with time.

Filename: `Lotka_Volterra.py`

Problem A.7. Difference equations for computing arctan(x)

The purpose of this exercise is to implement difference equations for computing a Taylor polynomial approximation to $\arctan(x)$. For $x \in (-1, 1)$, we have

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

To compute $S(x;n)$ efficiently, we can write the sum as $S(x;n) = \sum_{j=0}^{n} a_j$, and derive a relation between two consecutive terms in the series:

$$a_j = -x^2 \frac{(2j-1)}{(2j+1)} a_{j-1}.$$

We introduce $s_j = S(x; j-1)$ and a_j as the two sequences to compute. We have $s_0 = 0$ and $a_0 = x$.

a) Implement the system of difference equations in a function `arctan_Taylor(x,n)`. The function shall not use any lists or arrays. (Since a_j only depends on a_{j-1}, and s_j only depends on s_{j-1} and a_{j-1}, these are the only values that we need to store.) The function shall return 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 Taylor polynomial approximation.

The function `arctan_Taylor(x, n)` will give extremely inaccurate approximations to $\arctan(x)$ for $x \notin (-1, 1)$. To find a good approximation to $\arctan(x)$ for all x, we can use the fact that

$$\begin{aligned} \arctan(x) =& \frac{\pi}{2} - \arctan\left(\frac{1}{x}\right) \\ \approx& \frac{\pi}{2} - S(x;n), \end{aligned}$$

for values of $x \geq 1$.

b) Implement the following piecewise function as a python function `arctan_Taylor_improved(x, n)`. Your function shall return both the approximation and the measure of the size of the error as in `a)`.

$$\arctan(x) \approx \begin{cases} \frac{\pi}{2} - S\left(\frac{1}{x}; n\right) & 1 \leq x \\ S\left(x; n\right) & -1 < x < 1 \\ -\frac{\pi}{2} - S\left(\frac{1}{x}; n\right) & -1 \geq x \end{cases}$$

Hint: Call the function `arctan_Taylor` inside the piecewise function.

c) Write a test function to verify the approximation from `b)`. Test for $x = [-4, -0.5, 0.7, 10]$ with $n = 10$, and with tolerance $= 10^{-5}$.

d) Make a table or plot of the approximation from `b)` for various x and n to illustrate that the accuracy improves as n increases.

In the table; include the x value and the order n, the approximation and the exact value, and the measure of the error.

In the plot; calculate for $x \in [-20, 20]$. Include also the exact function for comparison, and remember to label the graphs.

Filename: `arctan_Taylor_series_diffeq.py`

Appendix E

Programming of Differential Equations

The exercises of this chapter are about programming of solvers for ordinary differential equations, corresponding to Appendix E in the book by Langtangen. The topics of the exercises are also covered by the lecture notes *Solving Ordinary Differential Equations in Python* [1].

Problem E.1. Solve a simple ODE with function-based code
This exercise aims to solve the ODE problem $u - 5u' = 0, u(0) = 0.1$, for $t \in [0, 20]$.

a) Identify the mathematical function $f(u, t)$ in the generic ODE form $u' = f(u, t)$.

b) Implement the $f(u, t)$ function in a Python function.

c) Use the `ForwardEuler` function from Section 2.1 of the lecture notes *Solving Ordinary Differential Equations in Python* to compute a numerical solution of the ODE problem. Use a time step $\Delta t = 5$.

d) Plot the numerical solution and the exact solution $u(t) = 0.1e^{0.2t}$.

e) Perform simulations for smaller Δt values and demonstrate visually that the numerical solution approaches the exact solution.

Filename: `simple_ODE_func.py`

Problem E.2. Solve a simple ODE with class-based code
Solve the same ODE problem as in Exercise E.1, but use the `ForwardEuler` class described in Section 2.2 of the lecture notes *Solving Ordinary Differential Equations in Python*. Implement the right-hand side function f as a class too.

Filename: `simple_ODE_class.py`

[1] See: *https://sundnes.github.io/solving_odes_in_python/*

Problem E.3. Solve a simple ODE with the ODEsolver hierarchy
Solve the same ODE problem as in Exercise E.1, but use the `ForwardEuler` class in the `ODESolver` hierarchy from Section 2.4 of the lecture notes.

Filename: `simple_ODE_class_ODESolver.py`

Problem E.4. Decrease the length of the time steps
We have the following differential equation

$$\frac{dx}{dt} = \frac{\cos(6t)}{1+t+x}.$$

Use Forward Euler to solve this differential equation numerically. You should solve it on the interval $t \in [0, 10]$, with initial condition $x(0) = 0$, and for number of time steps $n = \{20, 30, 35, 40, 50, 100, 1000, 10000\}$. Plot all the solutions in the same plot.

Filename: `decrease_dt.py`

Problem E.5. Implement the explicit midpoint method
Implement a subclass `Midpoint` in the `ODESolver` hierarchy from Section 2.4 of the lecture notes *Solving Ordinary Differential Equations in Python*. The class should implement the explicit midpoint method described in Section 2.3 in the lecture notes, defined by

$$\begin{aligned} k_1 &= f(u_n, t_n), \\ k_2 &= f(u_n + \frac{\Delta t}{2} k_1, t_n + \frac{\Delta t}{2}), \\ u_{n+1} &= u_n + \Delta t\, k_2. \end{aligned}$$

Test your implementation by solving $u' = f(u,t) = \cos(t) - t\sin(t), u(0) = 0$ and plot the numerical solutions obtained from both the explicit midpoint method and Forward Euler together with the analytical solution $u(t) = t\cos(t)$. Use 20 time steps on the interval $t \in [0, 4\pi]$.

Filename: `Midpoint.py`

Problem E.6. Implement Heun's method as a function

a) Heun's method is defined by

$$\begin{aligned} k_1 &= f(u_n, t_n), \\ k_2 &= f(u_n + \Delta t k_1, t_n + \Delta t), \\ u_{n+1} &= u_n + \Delta t\,(k_1/2 + k_2/2). \end{aligned}$$

Implement Heun's method using a plain function `heuns_method` of the same type as the `ForwardEuler` function from Section 2.1 in the lecture notes *Solving Ordinary Differential Equations in Python*. Construct a test problem where you know the analytical solution.

b) Implement a test function `test_heuns_against_hand_calculations()` where you compare your own hand calculated results with those returned from the function. Compare the solution at one or two time steps.

c) Plot the difference between the numerical and analytical solution of your test problem. You should demonstrate that the numerical solution approaches the exact solution as Δt decreases.

Filename: `heuns_method_func.py`

Problem E.7. Solve an ODE describing cooling of coffee

Newton's law of cooling,

$$\frac{dT}{dt} = -h(T - T_s)$$

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 experimentally determined constant (heat transfer coefficient) describing how efficient the heat exchange with the surroundings is. In this exercise we shall model the cooling of freshly brewed coffee. First we must find a measure of h. Suppose we have measured T at $t_0 = 0$ and t_1. We can then use a Forward Euler approximation of $\frac{dT}{dt}$ 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

$$h = \frac{T(t_1) - T(0)}{t_1(T_s - T(0))}.$$

a) Make a `class Cooling` containing the parameters h and T_s as data attributes. Let these parameters be set in the constructor. Implement the right-hand side of the ODE in a `__call__(self, T, t)` method. We will use a class from the `ODESolver` hierarchy described in Section 2.4 of the lecture notes to solve the ODE.

Create a stand-alone function `estimate_h(t1, Ts, T0, T1)` to estimate the h parameter based on the initial temperature and one data point $(t_1, T(t_1))$. You can use this function to estimate a value for h prior to calling the constructor in the `Cooling` class.

b) Implement a test function `test_Cooling()` for testing that class `Cooling` works. The test function should verify that the `__call__` method returns the correct results for given values of the arguments `T` and `t`.

c) The temperature of freshly brewed coffee is 95° C at $t_0 = 0$ (when it is poured into your cup), and 92° C after 15 seconds, in a room with temperature T_s.

Solve the ODE numerically by a method of your choice from the `ODESolver` hierarchy, for $T_s = 20$ and $T_s = 25$. Plot the two solutions in the same plot. The time interval where you solve the equations should be chosen long enough for the solutions to be "almost flat" and close to the respective room temperature.

Filename: `coffee.py`

Problem E.8. Compare numerical methods for solving ODEs

a) Make two subclasses `Midpoint` and `Heun` in the `ODESolver` hierarchy described in Section 2.4 of the lecture notes *Solving Ordinarty Differential Equations in Python*. The classes should implement the explicit midpoint method, defined in Problem E.5, and Heun's method, defined in Problem E.6, respectively.

b) Test your implementation in the main block of your program, by solving the ODE $u'(t) = t\cos(t) - \sin(t), u(0) = 2$. Compute the numerical solution using Heun's method, the explicit midpoint method, and the 4th order Runge-Kutta method from the `ODESolver` hierarchy.

Make one plot for each method, where the numerical method is compared to the analytical solution $y(t) = t\sin(t) + 2\cos(t)$. You should solve and plot on the interval $t \in [0, 8\pi]$ using n number of points on the interval, for $n = \{20, 25, 50, 150\}$. To create multiple plots you can either use the `plt.subplot` function (assuming `mathplotlib.pyplot` is imported as `plt`) to get multiple plots in the same window, or you can use `plt.figure(1)`, `plt.figure(2)`, etc., to open multiple plot windows.

Remember to label/title each plot and to include a legend with the different values of n.

Filename: `compare_methods.py`

Problem E.9. Solving a system of ODEs; Lotka-Volterra model

In Chapter A we solved a system of difference equations known as the Lotka-Volterra model, which describes the dynamic interactions between two species. The Lotka-Volterra model is more commonly formulated as a system of ODEs, which for two species R and F is given by

$$\frac{dR}{dt} = aR - cRF,$$
$$\frac{dR}{dt} = ecRF - bF.$$

Here a is the natural growth rate of rabbits in the absence of predation, b is the natural death rate of foxes in the absence of food (rabbits), c is the death rate of rabbits due to predation and e is the efficiency of turning predated rabbits into foxes.

Solve this system using one of the solvers from the `ODESolver` hierarchy, up to time $T = 500$ weeks. Use $a = 0.04$ weeks^{-1}, $b = 0.1$ weeks^{-1}, $c = 0.005$ weeks^{-1}, and $e = 0.2$ (dimensionless). Use initial conditions $R_0 = 100$ and $F_0 = 20$, and select a suitable time step Δt.

Plot how the number of individuals in the populations vary with time.

Filename: `Lotka_Volterra_ODE.py`

Problem E.10. Solving a system of ODEs; motions of a spring

The `ODESolver` hierarchy is adapted to cope with both scalar ODEs and systems of ODEs. In this exercise we will solve a system of ODEs using `ODESolver`.

Any ordinary differential equation of n^{th} order can be written as a system of 1^{st} order equations.

We will see how a 2^{nd} order ODE can be used to study the motions of a spring. We have a block of mass m hanging from a spring. The block is pulled

downwards before it is released, causing the block to oscillate vertically. We will study the oscillation of the block in this exercise.

If you want to read the mathematical reasoning concerning this physical phenomena in detail, you may look up section 10.7 *Svingninger og resonans* in Kalkulus (Lindstrøm, 2016). This is however not necessary to solve the exercise.

a) Consider the following 2^{nd} order ODE

$$u'' + \frac{q}{m}u' + \frac{k}{m}u = 0,$$

which describes the motion of the spring. The initial condition is

$$u_0 = \left(1, \quad -\frac{q}{2m} + \sqrt{\frac{k}{m} - \left(\frac{q}{2m}\right)^2}\right).$$

The parameter k is a constant factor that describes the stiffness of the spring. The parameter q describes the friction, for example air resistance. When $q = 0$ there is no friction. In this entire exercise we will consider the case where $m = 1$ and $k = 2$.

Rewrite the equation to a system of ODEs by hand. Then create a class `ProblemSpring` that takes the parameters m, k, and q as instance attributes in the constructor. The vector u_0 should also be defined in the constructor, as it depends on m, k and q. Create a special method `__call__(self, u, t)` that returns the vector $\left[\frac{d}{dt}u(t), \frac{d}{dt}u'(t)\right]$ that you calculated by hand.

b) Create a new class `SolverSpring` that takes `problem`, T (time stop) and n (time steps) as parameters in the constructor. The `problem` attribute is supposed to be an instance of the `ProblemSpring` class. Write a method `solve(self, method)` that solves our system of ODEs. This is an example of how it can be done:

```
def solve(self, method=RungeKutta4):
        self.solver = method(self.problem)
        self.solver.set_initial_condition(self.problem.U0)
        time_points = linspace(0, self.T, self.n+1)
        U, self.t = self.solver.solve(time_points)
        self.u, self.u_der= U[:,0], U[:,1]
```

You can choose whether you want to use `RungeKutta4` or another method from `ODESolver`. Notice how we extract the inital condition `U0` from the `problem` instance of `class ProblemSpring`. Also note that the array `U` contains the values of the approximations to both $u(t)$ and $u'(t)$. Make sure you understand how to extract the two columns vectors `self.u` and `self.u_der`.

c) Write a method `exact(self)` that returns the analytical solution to our differential equation. The analytic solution is given by:

$$u(t) = \mathrm{e}^{-\frac{qt}{2m}}\left(\cos\left(t\sqrt{\frac{k}{m} - \left(\frac{q}{2m}\right)^2}\right) + \sin\left(t\sqrt{\frac{k}{m} - \left(\frac{q}{2m}\right)^2}\right)\right)$$

d) Write a method `plot(self)` which plots the exact solution of $u(t)$ together with your approximations to $u(t)$ and $u'(t)$.

e) In the main block, create two problem instances of `ProblemSpring` that represents the two cases where $q = 0$ and $q = 0.3\sqrt{km}$, respectively.

Also, create a solver instance of `SolverSpring` for each of the two problem instances you just created. Let $T = 30$ and $n = 1000$.

Call the `solve` and the `plot` methods for both of your solver instances.

f) You will now create two test functions to ensure your implementation of the problem class and solver class is correct.

Make one test function where you compare the computed solution with the analytical solution of $u(t)$ for some given parameters, and let the test pass if the maximum error is less than some given tolerance.

Make another test function where you compare the computed derivative $u'(t)$ with the exact $u'(t)$ for the case where $m = 1$, $k = 2$ and $q = 0$. In this case the analytical solution is

$$u'(t) = \sqrt{2}\left(\cos\left(t\sqrt{2}\right) - \sin\left(t\sqrt{(2)}\right)\right).$$

Let the test pass if the maximum error is less than some given tolerance.

Filename: `spring_diffeq.py`

Problem E.11. Modelling war between nations

We consider the interaction between two nations C1 and C2 and a system of equations for modelling a conflict between these. Assuming that each nation is determined to defend itself against a possible attack, let $x(t)$ and $y(t)$ denote the armaments of the first and second nation respectively. The change $x'(t)$ depends on the armaments of $y(t)$. We assume that it's proportional to it, say $ky(t)$ for some positive constant k. It also depends on the relationship of the two. Assuming anger leads to increased armaments, let g measure the relationship between them, positive numbers meaning anger towards the other nation and 0 means neutral, and negative numbers meaning disarmament. The cost of having an army will restrain $x(t)$, represented by a term $-\alpha x$ for some positive constant α. Similar setup for $y(t)$ yields a system of differential equations:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = ky(t) - \alpha x(t) + g, \quad \frac{\mathrm{d}y}{\mathrm{d}t} = lx(t) - \beta y(t) + h. \tag{E.1}$$

In the case where $x'(t) = y'(t) = 0$ we have reached a stable point where neither nation is increasing armies. We interpret such a fixed point as peace. In the case were $x(t)$ and $y(t)$ diverges we have an arms race, and we interpret this as war.

a) Make a function that solves the system (E.1) with a numerical method of your own choice (you may use ODESolver to do this) and a function that plots the solution curves of $x(t)$ and $y(t)$ for given initial values. Your program should not solve beyond the point where either x or y is zero. We want to allow the value zero, so have your program check whether x and y are larger than a very small negative number. If you use ODESolver this can be done by defining a terminate function to send with the solve method. Until otherwise specified, we let t be the time measured in years.

Filename: `C_model.py`

b) Modify your program to instead consist of two classes. The first class `ProblemConflict` should contain the following:

- An init method saving all information relevant to the problem (parameters etc)
- A call method such that the class can be called as a function. It should take an array $[x, y]$ of specific values of x and y at time t, the time t, and return the right hand side of the ODE system.

The second class `Solver` should consist of the following:

- An init method that takes a problem instance on the form above, and a step length dt.
- A method that solves the problem, with the same restrictions as in Problem a). It should solve any problem on the same form as `ProblemConflict` that is given by two differential equations.
- A method that plots the solutions as in Problem a).
- A method that saves an image of the plot in `.png` format. When this is called, no plot should be visible to the user.

Use the parameter values $\alpha = \beta = 0.2$, $g = h = 0$, $x_0 = 10000$, $y_0 = 20000$. Run the program once with $k = l = 0.2$, and once for $k = l = 0.3$ plotting the first 10 years. What is the interpreted difference between these two?

Hint: You may need to convert step length to the number of time points to use. This can be done as

```
n =int(round("Last time step"/ "Step length"))+1
```

Filename: `C_model_class.py`

c) Let us consider the parameter values $k = l = 0.9$, $\alpha = \beta = 0.2$, and $g = h = 0$. One can argue that these give rough estimates for the arms race between 1909 to 1914 between the alliances of France and Russia, and Germany and Austria-Hungary. Assuming stability prior to this and negligible armies, we assume $x_0 \approx 0$ and $y_0 \approx 0$ (This does not mean that neither nation had armies, but that they were much smaller prior to the arms race). Solve the problem when x_0 and y_0 are zero versus when they are small positive numbers. Plot the next 5 years of both in the same figure. What happens?

Filename: `C_model_c.py`

d) So far we have seen a model intended to describe a conflict situation prior to war. The preceding model doesn't describe what happens during a war, but similar equations can.

First of all, we will work with two types of warfare. The conventional one, what we know as regular warfare, and guerrilla warfare, where groups of combatants use military tactics such as ambush, raids, hit-and-run, among others.

We first consider two conventional armies engaging. Let $x(t)$ and $y(t)$ denote the respective forces (the number of soldiers) and t denote the time measured in days. The rate of change of $x(t)$ is affected by combat loss, operational loss

(non-combat related. e.g. disease, accidents), and reinforcements. Combat loss should be proportional to the size of the opponent, represented by a term $-\alpha y(t)$, $\alpha > 0$. The operational losses should depend only on $x(t)$, represented by a term $-kx(t)$, $k > 0$. The reinforcements are represented by a function $f(t)$. In short-term warfare, the operational losses are negligible, and we will assume it to be zero. We get equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\alpha y(t) + f(t), \quad \frac{\mathrm{d}y}{\mathrm{d}t} = -\beta x + g(t).$$

For a conventional-guerrilla combat, $y(t)$ representing the guerrilla army, we assume that the combat losses also depend on the size of its own army. As guerrilla armies often use strategies of surprise and hidden attacks, it is safe to assume that bigger losses are experienced when the army is larger. Let $-\beta x(t)y(t)$ denote the combat loss of a guerrilla army. By the same arguments as above, we get equations

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\alpha y(t) + f(t), \quad \frac{\mathrm{d}y}{\mathrm{d}t} = -\beta x(t)y(t) + g(t).$$

Make two classes `ProblemCCWar` and `ProblemGCWar` on the same form as `ProblemConflict` representing the new problems. Note that f and g are now functions. To handle the case of the provided f and/or g not being functions, you may need the commands `callable(f)` which checks if f a callable, and `isinstance(f, (float, int))` that checks if f is a float or int, in order to convert a constant to a constant function.

Filename: `CW_model.py`

e) The battle of Iwo Jima is a famous battle during World War II. It was fought on an island just outside of Japan. America invaded the island on February 19, 1945, and the fight lasted for 36 days. The Japanese army consisted of around 21500 soldiers, while the Americans had a number above 50000 by the 36th day.

During the war, the Japanese had no reinforcements. The Americans started with no soldiers, but landed 54000 soldiers the first day, 6000 the third, 13000 the sixth, and none for the remaining. The reinforcements is therefore given as

$$f(t) = \begin{cases} 54000 & 0 \leq t < 1 \\ 0 & 1 \leq t < 2 \\ 6000 & 2 \leq t < 3 \\ 0 & 3 \leq t < 5 \\ 13000 & 5 \leq t < 6 \\ 0 & t \geq 6 \end{cases}$$

It can be shown that good estimates for the parameter values are $\alpha = 0.0544$ and $\beta = 0.0106$. The exact values on a day to day basis is given in the file `casualties.txt`. Plot the modeled American army vs the exact numbers, and $y(t)$ vs $x(t)$. Both plots should have the x-axis corresponding to the first $T = 36$ days.

Filename: `iwo_jima.py`

f) Find the least number of soldiers Japan would need (according to the model) to have won the fight. You may round to the nearest hundred. *Hint: Check which army decreases to zero first. You might want to extend the variable T for this.*

Filename: `least_number.py`

g) Suppose the Japanese army was interpreted as a guerrilla army. Find a value for β such that the fight is close. Is it likely that the outcome would be different if America met a large guerrilla army?

Filename: `guerrilla.py`

Problem E.12. Simulate the spreading of a disease

In this exercise we will model epidemiological diseases by implementing the SIRD model. Suppose we have four categories of people: susceptible (S) who are healthy but may contract the disease, infected (I) who have developed the disease and can infect the susceptible population, recovered (R) who have recovered from the disease and become immune, and the deceased (D) who did not survive the disease. Let $S(t)$, $I(t)$, $R(t)$ and $D(t)$ be the number of people in category S, I, R and D, respectively at time t. We have that $S(t) + I(t) + R(t) + D(t) = N$, where N is the size of the initial population. For simplicity, we will assume that the populations otherwise remains constant.

Normal interaction between infected and susceptible members of the population causes a fraction of the susceptible to contract the disease. The fraction of the susceptible population that becomes infected will depend on the likelihood of encountering an infected individual as well as how contagious the disease is. This will be proportional to the number of infected members of the population.

During a time interval Δt starting at time t, the fraction of the susceptible population which contracts the disease is $\alpha I(t)\Delta t$. The number of people that moves from the S to the I category is given by

$$S(t + \Delta t) = S(t) - \alpha S(t)I(t)\Delta t.$$

Divide by Δt and let $\Delta \to 0$ to get the differential equation:

$$S'(t) = -\alpha S(t)I(t). \tag{E.2}$$

Per time unit a fraction β of the infected will recover from the disease, and a fraction γ of the infected will die as a result of the disease. In a time Δt, $\beta I(t)\Delta t$ people of the infected population will recover and move from the I to the R category, and $\gamma I(t)\Delta t$ dies and move from the I to the D category. In the same time interval, $\alpha S(t)I(t)\Delta t$ from the S category will be infected and moved to the I category. The accounting for the I category therefore becomes

$$I(t + \Delta t) = I(t) + \alpha S(t)I(t)\Delta t - \beta I(t)\Delta t - \gamma I(t)\Delta t,$$

which in the limit $\Delta t \to 0$ becomes the differential equation

$$I'(t) = \alpha S(t)I(t) - \beta I(t) - \gamma I(t). \tag{E.3}$$

The R category gets contributions from the I category:

$$R(t + \Delta t) = R(t) + \beta I(t)\Delta t,$$

and the corresponding ODE for R reads

$$R'(t) = \beta I(t). \tag{E.4}$$

Finally, the D category gets contributions from the I category as well:

$$D(t + \Delta t) = D(t) + \gamma I(t)\Delta t,$$

and the corresponding ODE for D reads

$$D'(t) = \gamma I(t). \tag{E.5}$$

The system (E.2)-(E.5) is what we will call a SIRD model.

a) Make three separate functions: (i) `SIRD(u,t)` which defines the right hand side of the ODE system of the SIRD model. The parameters α, β, γ can be defined as local variables inside the function. (ii) A function `solve_SIRD` for solving the system of differential equations in the SIRD model by a numerical method of your choice from the `ODESolver` class hierarchy. (iii) A function `plot_SIRD(u,t)` for visualising $S(t)$, $I(t)$, $R(t)$ and $D(t)$ in the same plot. Make sure to use labels in the plot.

We will use the functions from a) to study the spreading of one of the most devastating pandemics in human history, the Black Death. The Black Death, also called the Plague, was evidently spread to Norway in 1349 after a ship from England arrived in Bjørgvin (today Bergen), carrying the disease.

There lived approximately 7000 people in Bjørgvin in 1349. Let's say the ship crew consisted of 30 men, which were all infected with the Plague. For simplicity we only consider human-to-human transmission of the disease (we do not consider the rats and fleas). We are interested in how the disease developed in Bjørgvin the first 8 weeks after the ship arrived.

We assume that the Plague was 90% deadly and that death usually occurred four days after being infected.

b) Adding the equations shows that $S'(t)+I'(t)+R'(t)+D'(t) = 0$, which means that $S(t)+I(t)+R(t)+D(t)$ must be constant. Perform a test at each time step by checking that $S(t)+I(t)+R(t)+D(t)$ equals $S(0)+I(0)+R(0)+D(0)$ within some small tolerance. Since `ODESolver` is used to solve the ODE system, the test should be implemented as a user-specified `terminate(u, t, k)` function that is called by the `solve` function at every time step. The `terminate` function should simply print an error message and return `True` for if $S + I + R + D$ is not constant.

c) Visualise first how the disease develops when $\alpha = 6.5 \cdot 10^{-5}$ and print out the number of deceased after 63 days.

Certain precautions, like staying inside, will reduce α. Try $\alpha = 5.5 \cdot 10^{-5}$ and compare the plot with the plot where $\alpha = 6.5 \cdot 10^{-5}$. Comment how the α influences $S(t)$.

Use the constants $\beta = 0.1/4$ and $\gamma = 0.9/4$ to describe the Plague in Bjørgvin. The initial conditions would be $S(0) = 7000$, $I(0) = 30$, $R(0) = 0$, $D(0) = 0$, $\Delta t = 1$, and $t \in [0, 63]$. Time t here is given in days.

As there do not exist any exact data from the condition in Norway during the Black Death, the parameters above are all fictional. However, there is a broad consensus that the disease killed more than half of Norway's population. Read more about the disease here: https://snl.no/svartedauden and https://sml.snl.no/svartedauden.

Filename: `bjorgvin.py`

Problem E.13. Introduce classes in the SIRD model
Implement the SIRD model from exercise E.12 in a module called `SIRD.py`. First we will create a class `Region` which can represent any region given the specific initial conditions. Then we will create a problem class `ProblemSIRD` and a solver class `SolverSIRD` to solve the SIRD system of differential equations for a given region.

a) Create a class `Region` which has three methods, a constructor, a method `set_SIRD_values(self, u, t)` and a method for plotting the SIRD values `plot(self, x_label)`.

The constructor should take in the name of the region and the initial conditions $S(0)$, $I(0)$, $R(0)$ and $D(0)$. All five parameters should be stores as attributes in the class. You will also need an attribute `self.population` which is the total population of the region at t_0.

The method `set_SIRD_values(self, u, t)` should take out the SIRD values from `u` and store S, I, R, D and t as attributes of the class.

The method `plot(self, x_label)` should plot S, I, R and D in the same plot. The string for the `plt.xlabel` should be given as a parameter (as the units of time may vary), while the `plt.ylabel` should always be set to e.g "Population". Set the title of the plot to be the name of the region. Specify color and label for all the different SIRD categories (an example could be `plt.plot(self.t, self.S, label='Susceptible', color='Blue')`). Do not include `plt.legend()` or `plt.show()` in the code. This is because we may later want to add labels to the graphs, and because will use this method to plot several subplots.

In a main block in the bottom of the file, create an instance of class `Region` called `bjorgvin` using the parameters found in Problem E.12.

b) Create the class `ProblemSIRD`. The constructor should take in the parameters α, β, γ and a region, which must be an instance of the class `Region`. The parameter α in the SIRD model can be constant or function of time. The implementation of `ProblemSIRD` should be such that α can be given as either a constant or a Python function. The constructor should therefore look like this:

```
def __init__(self, region, alpha, beta, gamma):
    if isinstance(alpha, (float, int)): # number?
        self.alpha = lambda t: alpha    # wrap as function
    elif callable(alpha):
        self.alpha = alpha
    # Store the other parameters
    self.set_initial_condition()        # method call
```

Write the method `set_initial_condition(self)` which stores a list `self.initial_condition` containing the initial values of $S(0)$, $I(0)$, $R(0)$, $D(0)$

(in this particular order). The initial values should be extracted from the class attribute `region`.

Write a method `get_population(self)` which simply returns the value of the population of the region, which is stored in the class attribute `region`.

Write a method `solution(self, u, t)` which simply calls the method `set_SIRD_values(u, t)` of the class attribute `region`.

Finally, write a special method `__call__(self, u, t)` which represents the right-hand side function of our SIRD system of ODEs. This method will return a list of the calculated values of $S'(t), I'(t), R'(t)$ and $D'(t)$, in that order.

In the main block, create an instance of class `ProblemSIRD` called `problem` using the parameters found in Problem E.12 and the `Region bjorgvin`.

c) Now we will create a class `SolverSIRD`. The class constructor should take the parameters `problem` (which must be an instance of class `ProblemSIRD`), T (final time) and Δt, and store them as attributes. The constructor should also store an attribute called `total_population`, which is obtained by calling the `get_population` method of `problem`.

Implement a method `terminate` which at each time step t checks that $S(t)+I(t)+R(t)+D(t)$ equals `total_population` within some small tolerance. Simply print an error message and return `True` for termination if the total population is not constant. As the `ODESolver` class hierarchy will be used to solve the ODE system, this method will be called by the `solve` method in `ODESolver` at every time step.

Write a method `solve(self, method)` that solves the SIRD system of ODEs by a method of your choice from the `ODESolver` hierarchy. Use the following sketch for this method:

```
def solve(self, method=RungeKutta4):
    solver = method(self.problem)
    solver.set_initial_condition(...)
    t = np.linspace(...)
    # Remember to "activate" terminate in the solve call:
    u, t = solver.solve(t, self.terminate)
    # set the values of S, I, R, D, and t via
    # Problem class to the Region class:
    self.problem.solution(u, t)
```

In the main block, create an instance of class `SolverSIRD` called `solver` using the parameters found in Problem E.12 and the `ProblemSIRD problem`. Plot the results by calling the `plot(x_label)` method of the `Region bjorgvin`. Label the graphs by calling `plt.legend()` before you call `plt.show()`.

Filename: `SIRD.py`

Problem E.14. The SIRD model across regions

The problem class from exercise E.13 will only be able to model the spreading of a disease within one region. In this exercise we will improve our program with subclasses of `ProblemSIRD` and `Region` that permits people in one region to get infected by people from another region. The likelihood of transmission of disease across regions will depend on the distance between the regions.

We now denote the categories to specify which region they belong, such that e.g. $S_i(t)$ would be the number of susceptible in the i-th region at time t.

Let $S_i'(t)$, $I_i'(t)$, $R_i'(t)$ and $D_i'(t)$ belong to the i-th region. In this new, interacting SIRD model the expressions of $R_i'(t)$ and $D_i'(t)$ are unchanged from the SIRD model explained in Problem E.12, such that

$$R_i'(t) = \beta I_i(t)$$
$$D_i'(t) = \gamma I_i(t).$$

The expression of $I_i'(t)$ must consider the possibility of members of the population in the i-th region contracting the disease due to contact with another region. The expression of the i-th region is given by

$$S_i' = \sum_{j=1}^{M} -\alpha I_j e^{-d_{ij}} S_i,$$

where M is the number of regions and d_{ij} is the distance between the i-th and the j-th region. Note that the distance from a region to itself, d_{ii}, is always zero, which leaves the expression unchanged from the previous SIRD model.

The derivative for the infected category is then

$$I_i'(t) = -\beta I_i(t) - \gamma I_i(t) - S_i'(t).$$

a) Create a subclass of `Region` called `RegionInteraction`. In addition to the parameters in `Region`, the `RegionInteraction` needs two parameters `latitude` (ϕ) and `longitude` (λ). The constructor should store the two values and convert them from degrees to radii, which is done by multiplying by $\frac{\pi}{180^\circ}$. Use the super class to store the rest of the parameters as attributes.

Create a method `distance(self, other)` which calculates the distance between the `self` region (i) and another region (j). The distance between two regions is given by the arc length d between the regions:

$$d_{ij} = R_{Earth} \Delta\sigma_{ij},$$

where the radius of the Earth is $R_{Earth} = 64$ given in units of 10^4 m and $\Delta\sigma$ is given by

$$\Delta\sigma_{ij} = \arccos\big(\sin\phi_i \sin\phi_j + \cos\phi_i \cos\phi_j \cos\left(|\lambda_i - \lambda_j|\right)\big).$$

Warning: Roundoff error may cause problems in the arccos function, since the arguments may become slightly > 1 when a city is compared with itself, and this makes the function return a NaN (Not a Number) value. To avoid this problem, add an if-test inside the distance function to ensure that the argument to arccos is between 0 and 1.

b) Create a subclass `ProblemInteraction` of class `ProblemSIRD`. This class should take in a list of regions that must be instances of `RegionInteraction`. In adition to all the same parameters as its super class, the constructor of `ProblemInteraction` should take in and store `region_name`. Send all other parameters to the constructor in the super class.

The method `get_population(self)` should store the total population of all the regions combined as `self.total_population`.

The method `set_initial_condition(self)` must create a (not nested) list `self.initial_condition` with the initial values from all the regions. Loop over all the regions in the list `region` to get the list on the form

$$[S_1(0), I_1(0), R_1(0), D_1(0), S_2(0), I_2(0), R_2(0), D_2(0), ..., D_M(0)].$$

The special method `__call__(self, u, t)` should return a list with the derivatives at time t, in the same order as the list `self.initial_condition`. Below is a sketch of the implementation could look like:

```
def __call__(self, u, t):
    n = len(self.region)
    # create a nested: SIRD_list[i] = [S_i, I_i, R_i, D_i]:
    SIRD_list = [u[i:i+4] for i in range(0, len(u), 4)]
    # crate a list containing all the I(t)-values:
    I_list = ...
    derivative = []
    for i in range(n):
        S, I, R, D = SIRD_list[i]
        dS = 0
        for j in range(n):
            I_other = I_list[j]
            dS += ...
        # calculate dI, dR and dD
        # put the values in the end of derivative
    return derivative
```

The method `solution` must provide the SIRD lists to all the regions. The example below shows how to create a nested list where each element in the list contains the SIRD lists for a certain region. You do not have to use this code, but the result should be the same.

```
def solution(self, u, t):
        n = len(t)
        n_reg = len(self.region)
        self.t = t
        self.S = np.zeros(n)
        self.I = ...
        SIRD_list = [u[:, i:i+4] for i in range(0, n_reg*4, 4)]
        for part, SIRD in zip(self.region, SIRD_list):
            part.set_SIRD_values(SIRD, t)
            self.S += ...
```

The attributes `self.S`, `self.I`, `self.R` and `self.D` should be the total values for all the regions combined, i.e., each SIRD-category summed over all regions.

Create a new method `plot(self, x_label)`. the method should create the same kind of plot as class `Region`'s method `plot(self, x_label)`, as explained in Problem E.13. The method in `ProblemInteraction` should plot the SIRD values for all the regions combined, and the title of the plot should be `self.region_name`.

Filename: `SIRD_interaction.py`

Problem E.15. Simulate the spreading of the Plague in Norway
In this exercise we will use the classes `ProblemInteraction`, `SolverSIRD` and `RegionInteraction` from Problem E.14 to simulate the spread of the Black Death in Norway.

We assume that the disease first broke out in Bjørgvin in 1349, and that this was the only case of the Plague in country at that time. We assume that the disease ravaged for two years (104 weeks).

We divide Norway into five regions: Vestlandet, Sørlandet, Trøndelag, Østafjells and Nord-Norge. We assume that there lived about 370000 people in Norway; 90 000 in Vestlandet, 65 000 in Sørlandet, 80 000 in Østlandet, 70 000 in Trøndelag and 65 000 in Nord-Norge.

The positions of the regions are given by the latitude ang longitude of certain city in each region, which are given in the table below:

Region	city	latitude	longitude
Vestlandet	Bjørgvin:	60°	5.3°
Sørlandet	Øyslebø:	58°	7.6°
Østlandet	Brandbu:	60°	11°
Trøndelag	Steinkjer:	64°	11°
Nord-Norge	Bardufoss:	69°	19°

a) Create a function for simulating the Plague in Norway. The function must contain one instance of class `RegionInteraction` for each region. Let $S(0)$ be the population in each region. Except for $I(0) = 30$ in Vestlandet, let all the other initial conditions be set to zero. Create a list of the five `RegionInteraction` instances.

The function will plot one subplot of the disease progress of each region, and also one subplot for the total progress for all regions combined. The function could look like this:

```
def plague_Norway(alpha, beta, gamma, num_Weeks, dt):
    # create all five instances of regionInteraction
    # create a list containing all five regions
    # create problem, instance of ProblemInteraction
    # create solver, instance of SolverSIRD & call method solve
    plt.figure(figsize=(.., ..)) # set figsize
    index = 1
    # for each part in problem's attribute region:
        plt.subplot(2, 3, index)
        # Call plot method from current part
        index += 1
    plt.subplot(2, 3, index)
    # Call plot method from problem
    plt.legend()
    plt.show()
    # print the total percentage of deceased after two years
```

Hint: Write the percent sign twice (%%) to get % in a string.

Call the function using the parameters $\alpha = 7 \cdot 10^{-6}$, $\beta = 0.1/4$ and $\gamma = 0.9/4$. The unit of time is weeks. Solve for 104 weeks and set $\Delta t = 1/7$ such that one time step represent one day.

b) Until now we have assumed that α is constant. α is the parameter which describes the possibility that one susceptible meets an infected during the time interval Δt with the result that the infected "successfully" infect the susceptible. In reality, this α may probably not be constant. In times of bad weather, more people would stay at home and α would be lower. Other factors could also decrease alpha, like better hygiene and wearing masks over a certain period of time. On the other hand; factors like nice weather, festivities and other reasons for people to gather would increase α. As the Plague ravaged in Norway so long ago it is hard to reproduce an accurate approximation of α. Let us therefore assume that the weather and other factors made α look like this piecewise function:

$$\alpha(t) = \begin{cases} 3 \cdot 10^{-5} & 0 \leq t \leq 2 \\ 6 \cdot 10^{-6} & 4 < t \leq 19 \\ 6 \cdot 10^{-6} & 24 < t \leq 41 \\ 7 \cdot 10^{-6} & 49 < t \leq 75 \\ 1 \cdot 10^{-6} & \text{else} \end{cases}$$

Implement $\alpha(t)$ as a piecewise function `alpha(t)`. Call `plague_Norway` using the piecewise function for $\alpha(t)$. Keep the rest of the values from part a).

You may notice that by this model, the Plague barely reached Nord-Norge at all. In fact, we aren't even sure today whether the Plague ravaged in the Northern part of Norway or not.

c) In the Norwegian legends, Pesta was the literal personification of the Plague. Pesta was depicted as an old woman who wandered the countryside, carrying either a rake or a broom. Legend has it that if she passed by your home while carrying a rake, someone in the household would be infected by the plague while the rest would be spared. However, if she was carrying a broom then there were not much hope for anyone.

Implement a function `pesta(t)` where you generate a random number between 0 and 20. You can use the `randint` function imported from the `random` module like this:

```
from random import randint
number = randint(0,20)
```

If the generated number is 13, Pesta is at your door carrying a broom. The function should return ten times the value of `alpha(t)` from part b). If the generated number is 4, then Pesta has brought her rake. The function should return five times the value of `alpha(t)`. Otherwise, Pesta is not around today, and your function should return 0.4 times the value of `alpha(t)`.

Call `plague_Norway` using the `pesta(t)` function for the parameter α. Keep the rest of the values from part a). In which region was Pesta most present?

Filename: `plague.py`