Programming exercises with applications in physics
IN1900

Jonas Gahr Sturtzel Lunde
jonassl@student.matnat.uio.no

Kristine Baluka Hein
krisbhei@student.matnat.uio.no

June 2017

With Python, life is much simpler.

Morten Hjorth-Jensen

Department of Physics

Preface

These exercises were created by request of the Department of Physics, as a supplement to the Python course IN1900. The idea was to create a set of problems more relevant to the field of physics, corresponding in difficulty to the original exercises provided by the course. In theory, no prior knowledge of physics is required, and the main focus of the exercises is to teach Python.

If you find a mistake or have other questions regarding the set, please direct them towards

Jonas Gahr Sturtzel Lunde `jonassl@student.matnat.uio.no`
Kristine Baluka Hein `krisbhei@student.matnat.uio.no`

Contents

Exercise 1.1 - Density

Different materials has different densities. The density is defined as mass divided by the volume of the object, often given in kg/m^3.

Materials	Polystyrene (low density)	Cork	Rhenium	Platinium
Density (in kg/m^3)	20	220	21020	21450

Table 1: Density of different materials

A cube weights 858 g and has volume 40 cm^3.

Write a program which find the density of the cube. Use your result from the program to determine which material the cube is made of. The cube is made of one from the given materials in the table. It may happen that your result will not be exactly equal to one of the densities in the table. If this happens, you should choose the material which is closest to the result your program gives.

Filename: `massdensity.py`

Exercise 1.2 - Calculate the solar mass

The solar mass can be calculated by using the relation:

$$M_{Sun} = \frac{4\pi^2 \cdot (1\,\mathrm{AU})^3}{G \cdot (1\,\mathrm{yr})^2} \tag{1}$$

In this task we will use approximate values for AU and G. The unit AU is an astronomical unit of length. Its value is defined to be the average distance between the Sun and Earth:

$$1\,\mathrm{AU} = 1.58 \times 10^{-5}\,\text{light years}$$

where $1\,\text{lightyear} = 9.5 \times 10^{12}\,\mathrm{km}$. The constant G is called the gravitational constant and has the following value:

$$G = 6.674 \times 10^{-11}\,\mathrm{m}^3\,\mathrm{kg}^{-1}\,\mathrm{s}^{-1}$$

Write a program which calculates the solar mass by using equation (1). The calculated result must then be presented in kilogram using `print`. Your program should give that $M_{Sun} \approx 2.01 \cdot 10^{30}\,\mathrm{kg}$.

Filename: `solarmass.py`

Exercise 1.3 - Half-life

Certain atomic nuclei are unstable, and will over time decay into other nuclei, through emission of radiation. We call such atomic nuclei *radioactive*. The process of radioactive decay is completely random, but for large collections of atoms, we can model how much remains of the original matter after a certain time.

From an original mass N_0 of a radioactive material, the remaining mass after a time t (in seconds) is given by the equation for radioactive decay:

$$N(t) = N_0 \mathrm{e}^{-t/\tau} \tag{2}$$

τ is the so-called 'mean lifetime' of the radioactive material, and represents the average lifespan of a single nucleus in the radioactive material. A larger value indicates a more stable nuclei, and it can vary from $10^{30}\,\mathrm{s}$ for very stable materials, to $10^{-20}\,\mathrm{s}$ for very unstable materials.

a)

Carbon-11 is an unstable carbon isotope, and has a mean lifetime of $\tau = 1760\,\text{s}$.

Make a program which calculates how much remains of an original mass $N_0 = 4.5\,\text{kg}$ of carbon-11 after 10 minutes.

Hint: You can check whether your program works as intended by setting t equal to 0, and a very large number, and see if your results are as expected.

b)

The mean lifetime makes for a pretty formula, but we are more often talking about the *half-life* of a radioactive material. The half-life represents the time it takes for the material to reduce to exactly half of its original mass. The relation between the mean lifetime and the half-life is given as

$$\tau = \frac{t_{\frac{1}{2}}}{\ln 2}$$

The half-life of carbon-11 is $t_{\frac{1}{2}} = 1220\,\text{s}$. Rewrite your program such that it first calculates the mean lifetime from the half-life, and then calculates the remaining mass, just like in exercise a. Check that you get the same results as in exercise a.

Filename: `half_life.py`

Exercise 1.4 - The velocity of an atom

The atoms within a material is structured such that they create a lattice. We will look at an atom which moves along the surface of a material. Since the atoms are aligned as a lattice, we could use a periodic model to find the velocity of the atom moving across the surface:

$$v(x) = \sqrt{v_0^2 + \frac{2F_0}{m}\left(\cos\left(\frac{x}{n}\right) - 1\right)}$$

where m is the mass of the atom, x is its position, v_0 is its initial velocity and n is a scaled distance between the atoms within the material. We set the force $F_0 = 1\,\text{N}$.
Find the velocity of the atom when $x = 1$, $v_0 = 2$, $n = 4$ and $m = 3$.

Filename: `velocity_of_atom.py`

Exercise 1.5 - Correct Einstein's mistakes

Special relativity is the area of physics dealing with incredibly large velocities. In special relativity, the momentum p of an object with velocity v (in m/s), and mass m (in kg) is given as

$$p = m \cdot v \cdot \gamma, \qquad \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

where $c \approx 300\,000\,000\,\text{m/s}$ is the speed of light. The program below attempts to calculate the momentum of an object with speed equal to 1/3 the speed of light, and a mass $m = 0.14\,\text{kg}$. The program has many errors, and doesn't work. Copy and run the program. Correct the errors, and make it work like intended.

```
from math import squareroot
c = 300 000 000  # m/s
v = 100 000 000  # m/s
m = 0,14  # kg

gamma = 1/squareroot(1-(v^2/c^2))
```

```
p = m*v*gamma

print p
```

Filename: `Einsteins_errors.py`

Exercise 1.6 - Rydberg's constant

Rydberg's constant R_∞ for a heavy atom is used in physics to calculate the wavelength to spectral lines[1] .

The constant has been found to have the following value:

$$R_\infty = \frac{m_e e^4}{8\varepsilon_0^2 h^3 c}$$

where

- $m_e = 9.109 \times 10^{-31}\,\mathrm{m}$ is the mass of an electron
- $e = 1.602 \times 10^{-19}\,\mathrm{C}$ is the charge of a proton (also called the *elementary charge*)
- $\varepsilon_0 = 8.854 \times 10^{-12}\,\mathrm{C\,V^{-1}\,m^{-1}}$ is the electrical constant
- $h = 6.626 \times 10^{-34}\,\mathrm{J\,s}$ is Planck's constant
- $c = 3 \times 10^{8}\,\mathrm{m\,/s}$ is the speed of light

These are physical constants which are widely used in physics. You will probably encounter these several times in other physics courses!

Write a program which assigns the values of the physical constants to variables, and use the variables to calculate the value of Rydberg's constant. The program shall then present the result by using `print`.

See if your program gives that

$$R_\infty = 10961656.2162 \quad (\text{in } \mathrm{m^{-1}})$$

Notice that this is only an approximate value since we have rounded the physical constants to three decimals.

Filename: `constant_Rydberg.py`

[1] The reason why we use ∞ in the constant's symbol is because we assume that the mass of the atomic nucleus if infinitely large compared to the mass of the electron.

Exercise 2.1 - Measure time

A ball is dropped straight down from a cliff with height h. The position of the ball after a time t can be expressed as:

$$y(t) = v_0 t - \frac{1}{2}at^2 + h$$

where a is the acceleration (in $\mathrm{m/s^2}$) and v_0 is the initial velocity of the ball (measured in m/s).

We wish to find for how long time t_1 is takes the ball to pass a certain height h_1. In other words, we wish to find the value of t_1 such that $y(t_1) = h_1$. The position of the ball is measured per Δt seconds.

Write a program which finds out how long time t_1 it takes before the ball reaches height h_1 by using a while loop.

Here, we let $h = 10\,\mathrm{m}$, $y_1 = 5\,\mathrm{m}$, $\Delta t = 0.01\,\mathrm{s}$, $v_0 = 0\,\mathrm{m/s}$ and $a = 9.81\,\mathrm{m/s^2}$.

Hint: We cannot use '`==`' in the while loop to find t_1 for when the ball pass h_1. This is because we are measuring the time in Δt seconds where we most likely will measure the position of the ball *after* it has reached h_1. The best thing we could do, is to increase the time by Δt as long as the height of the ball is *greater* than h_1. We can then set the time to be t_1 at once where the height of the ball is less or equal to h_1. So, our program will be inaccurate, but this is the best we can do, given the fact that we do not have infinitely many time points.

Filename: `throw_ball_height.py`

Exercise 2.2 - Relativistic momentum

In classical physics, we define the momentum p of an object with mass m and velocity v as

$$p = m \cdot v$$

A satellite with mass $m = 1200\,\mathrm{kg}$ is trapped in the gravity of a black hole. It accelerates quickly from velocity $v = 0$ to $v = 0.9c$, where c is the speed of light, $c \approx 3 \times 10^8\,\mathrm{m/s}$.

a)

Write a program which prints a nicely formatted table to the terminal, containing the speed of the satellite in one column, and the momentum of the satellite in the other. Use time-intervals of $0.1c$ between $0c$ and $0.9c$.

Hint: Use scientific notation '%e' when printing the values, to avoid incredibly large floats. Alternatively, '%g', which picks the best notation for you. Try to limit the number of decimals to a reasonable number.

b)

In exercise 1.5, we saw how the momentum of an object is defined in special relativity, which deals with physics at very large velocities. We defined the momentum as

$$p_{rel} = m \cdot v \cdot \gamma, \qquad \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

This is the actual momentum of any object, but the classical version we used in exercise a) is a good approximation at 'small' velocities.

Expand your program such that it prints a table with three columns, the third one containing the momentum as defined in special relativity.

Filename: `relativistic_momentum.py`

Exercise 2.3 - Radioactive list

a)

In exercise 1.3 we studied the formula for radioactive decay, which tells us how much remains of a radioactive material with an original mass N_0, after a time t.

$$N(t) = N_0 \mathrm{e}^{-t/\tau}$$

Make a while loop which fills two lists: One with spaced time-points t, and one with values of $N(t)$ at these time-points. The loop should run until the remaining amount of material is below 50% of the original. Start in $t = 0\,\mathrm{s}$, and use time-steps of $60\,\mathrm{s}$. We are, as in exercise 1.3, looking at a mass $N_0 = 4.5\,\mathrm{kg}$ of carbon-11, which has a time constant $\tau = 1760\,\mathrm{s}$.

b)

You might have noticed that by aborting the loop when half of the material is gone, the last element in our time-list should be the *half-life* of the carbon-11, $t_{\frac{1}{2}}$. From exercise 1.3 we remember that the half life of a material is simply the time it takes for half the material to decay.

Test that this is true by printing and comparing the last element in your time-list to the half-life of carbon-11, defined as

$$t_{\frac{1}{2}} = \tau \ln 2$$

Remember that because your program uses time-steps of one minute, your measured half-life can have an error of up to 60 seconds.

c)

Combine the lists into a nested list `Nt`, such that every element in the list `Nt` is a pair of matching t and $N(t)$ values. For example, the first element `Nt[0]` of this list should be a list of `[0, 4.5]`.

Use the new nested list to write nicely a formatted table of corresponding t and $N(t)$ values to the terminal.

Filename: `radioactive_list.py`

Exercise 2.4 - Newton's law of universal gravitation

Newton's law of universal gravitation described how the gravitation acts as an attractive force between two objects:

$$F = G\frac{m_1 m_2}{r^2}$$

where m_1 and m_2 are the masses of the two objects which attracts each other and r is the distance between them.

The constant G is the gravitational constant which has the following value:

$$G = 6.674 \times 10^{-11}\,\mathrm{m^3\,kg^{-2}\,s^{-1}}$$

We will take a closer look at an object with mass $M = 3\,\mathrm{kg}$ which is influenced by N objects with mass $m_1, m_2, \ldots, m_N$. The i-th object has mass $m_i = \frac{i}{6} + 2\,\mathrm{kg}$ and distance $r_i = \sqrt{\left(\frac{i}{4}\right)^2 + 10}$ meters from the object with mass M.

The number of interacting objects is $N = 10$.

Write a program which calculates the total force $\sum_{i=1}^{N} F_i = F_1 + F_2 + ... + F_N$ which affects the object with mass M.

Filename: `newton_gravitation.py`

Exercise 2.5 - Trapped quantum particle

Quantum mechanics is the part of physics that deals with reality at very small scales. One of the rules in quantum mechanics is that, sometimes, particles are only allowed to have specific energies, and can never have an energy in between these allowed levels. The particle must therefore jump straight from one energy level to another.

When a particle is trapped in a tiny box[2] of size L, quantum mechanics says that it is only allowed to have energies

$$E_n = \frac{n^2 h^2}{8mL^2}, \qquad n = 1, 2, 3 \ldots$$

where m is the particle's mass and h is Planck's constant, $h = 6.626 \times 10^{-34}\,\mathrm{J\,s}$.

Consider an electron with mass $9.11 \times 10^{-31}\,\mathrm{kg}$, trapped in a box of size $10^{-11}\,\mathrm{m}$. It starts at the lowest energy-level, E_1 (not E_0!), and jumps upwards, one step at a time, ending up at a much higher energy level, E_{30}. Each step from a level E_i to a level E_{i+1} will have required an energy

$$E_{i+1} - E_i = \frac{((i+1)^2 - i^2)h^2}{8mL^2}$$

Write a for loop which calculates the energy required for each step along the way, and saves them in a list. Sum also up the total energy the particle has used on its way upwards.

Filename: `quantum_trap.py`

[2] Also known as an infinitely deep square potential

Exercise 2.6 - Looping over radii of loops

To be able to drive through a loop, one need to have a minimal speed to not loose contact with the loop.

Suppose a stunt person must drive through a loop. For the stunt person to be in contact with the loop throughout the stunt, it is necessary that the person have a speed of at *least*:

$$v = \sqrt{gr}$$

Here is $g = 9.81\,\mathrm{m/s^2}$ and r the radius of the loop (in meters).

You have been given a program which use a while loop to iterate through a list of radii r of different loops. For every radius r the program iterates through,the least speed v is calculated and displayed:

```
from math import sqrt

g = 9.81
r = [2.7,3.43,5.62,7.1]
num_loops = len(r)
v = [0]*num_loops

i = 0
while i < num_loops:
    v[i] = sqrt(r[i]*g)      #in m/s
    v[i] = v[i]*3600/1000    #convert to km/h
    print "Least speed to complete the loop: %.2f km/h"%v[i]
    i += 1
```

The program is named `while_loop_over_loops.py` which you can find here: path-to-file.

Change the given program such that it uses a for loop instead. Also, change the program such that it does not print the value of v in the same loop v has been calculated, but rather prints the vs in a separate for loop.

Filename: `for_loop_over_loops.py`

Exercise 3.1 - Radioactive function

Implement the formula for radioactive decay from exercise 1.3 as a function, `N(N0, tau, t)`.

Write a test function that uses the values $N_0 = 4.5\,\text{kg}$, $\tau = 1760\,\text{s}$, $t = 600\,\text{s}$, and confirms that you get the same results (within a tolerance) as you did with the carbon-11 in exercise 1.3 or 2.3. (This was approximately 3.2 kg).

Don't set your tolerance smaller than 10^{-4}, because the answer you are comparing to is not entirely exact.

Filename: `radioactive_function.py`

Exercise 3.2 - Quantum function

In exercise 2.5 we saw that particles trapped in a very small box of length L are only allowed to have energies

$$E_n = \frac{n^2 h^2}{8mL^2}, \qquad n = 1, 2, 3 \ldots$$

with m being the particle's mass, and h being Planck's constant $h = 6.626 \times 10^{-34}\,\text{J s}$

Write a function `quantum_energy`, which takes two energy-levels[3], the length of the box, and the particle's mass as arguments, and returns the energy difference between the two energy levels.

Filename: `quantum_function.py`

Exercise 3.3 - Wooden block

In this exercise, you are supposed to create a program which finds out how far a wooden block with initial velocity v_0 m/s will slide across surfaces of different materials. The material of the surface will affect the frictional force acting on the wooden block.

The position of the block can be expressed as such:

$$x(t) = v_0 t - \frac{1}{2}\mu g t^2$$

where μ is a coefficient of friction and $g = 9.81\ \text{m/s}^2$.

One can find by some calculations at which time T the block will stop moving. The time T is found to be

$$T = \frac{v_0}{\mu g}$$

Define a function which takes a list of different coefficients of friction as parameter, calculates the position at time T, that is $x(T)$, and then stores the results in a list. The list of the calculated positions must then be returned by the function.

Let $v_0 = 5\,\text{m/s}$ and the list of coefficients of friction be `[0.62, 0.3, 0.45, 0.2]`. Call the function and write out every position along with corresponding coefficient of friction.

Filename: `block_frictions.py`

Exercise 3.4 - Hit target

We will now simulate a game based on a simple physical model. In this game a person is supposed to throw a ball at a wall with a target painted on. The person gets points according to where the ball hits at the wall.

[3] To specify: The function should take the index of the energy levels, n

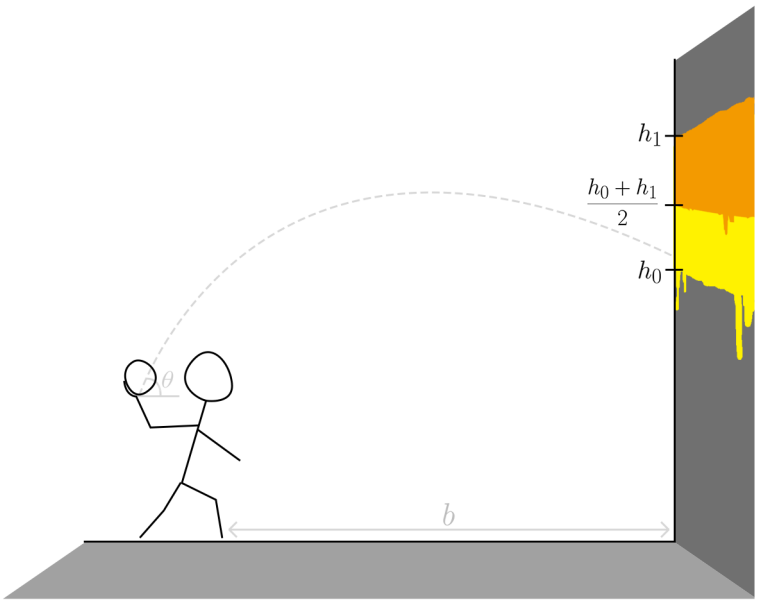

Figure 1: Illustration of the system which we will base our game simulation on.

The height of the ball can be modeled as:

$$y(t) = -\frac{1}{2}gt^2 + v_0 t \sin\theta$$

where v_0 is the speed the ball has been thrown with, θ is the angle at which the ball has been thrown from and $g = 9.81\,\mathrm{m/s^2}$.

a)

Write a function which returns the height of the ball at a given time t.

b)

One can find in our model that the ball will hit the wall at the time $T = \dfrac{b}{v_0 \cos\theta}$ where b is the distance between the person and the wall.

We must look at the value of $y(T)$ to be able to decide how many points the person will receive. The number of points must be calculated and returned from a function which you have to write.

The target is painted such that it covers the wall between height h_0 and height h_1 where $h_0 < h_1$. The points are given according to the following rules:

- The person gets 0 points if $y(T) < h_0$ or $y(T) > h_1$
- The person gets 1 point if $h_0 \leq y(T) < \frac{1}{2}(h_1 + h_0)$
- The person gets 2 points if $\frac{1}{2}(h_1 + h_0) \leq y(T) \leq h_1$

Write a program which prints in a for loop how many points the person gets using your newly written function if $h_0 = 3\,\mathrm{m}$, $h_1 = 3.5\,\mathrm{m}$, $\theta = \dfrac{\pi}{4}$, $b = 3.5\,\mathrm{m}$ for $v_0 = 15, 16, 19, 22\,\mathrm{m/s}$.

Filename: `hit_target.py`

Exercise 3.5 - Downward cliff throw

a)

We throw a rock straight down from a cliff, in a linear movement with constant acceleration. We ignore air resistance, and work only in one dimension. The velocity of the rock can be described either as a function of the distance it has traveled, x, or as a function of the time that has passed, t.

$$v(t) = v_0 + at \tag{3}$$

$$v(x) = \sqrt{v_0^2 + 2ax} \tag{4}$$

(The latter one is often written $v^2 - v_0^2 = 2ax$)

Where v_0 is the rock's initial velocity at $t = 0$, and a is its acceleration, which on earth is $a = 9.81\,\mathrm{m/s^2}$. We pick downwards as our positive direction, meaning that v_0, a and x are always positive values.

Write two functions, `velocity1` and `velocity2`, based on the two equations above, which both returns the velocity of the rock from their three variables.

b)

Write a test function `test_velocity()`, which confirms that both functions returns the same velocity at a given time t with the same initial velocity v_0. You can find the distance the ball has traveled at a given time from the formula

$$x(t) = v_0 t + \frac{1}{2}at^2$$

c)

Expand `velocity1` from exercise a), such that t also can be a list of time-points, in which case the function will return a list of velocities corresponding to those time-points. Note that the functions should still be capable of taking in t as a single number, in which case the functions should also only return a single number.

Hint: Use an if/else block to check the type of t, and handle the two cases separately. Remember that the 'single number' might be either a float or integer.

Filename: `cliff_throw.py`

Exercise 3.6 - Another wooden block

We want to create a program which finds out how far a wooden block with initial velocity $v_0\,\mathrm{m/s}$ will slide across surfaces of different materials. The material of the surface will affect the frictional force acting on the wooden block.

The velocity and the distance of the wooden block at a time t can be expressed as:

$$v(t) = v_0 - \mu g t$$

$$x(t) = v_0 t - \frac{1}{2}\mu g t^2$$

where μ is a coefficient of friction and v_0 is the velocity of the block at time $t = 0$.

a)

Define a function which returns the distance $x(T)$ and takes t, v_0 and a coefficient of friction μ as parameter.

b)

Define a function which takes in a list of different coefficients of friction and returns a list of how far the block has moved for each of coefficient of friction.

To do so, the function must also take Δt as parameter, which is the time step we will use to find the velocity of the block.

The function must then use a while loop to test for which time t the velocity $v(t)$ becomes less or equal to 0. Use the time t which has been found to calculate the distance $x(t)$. The distance must then be stored in a list. Let the function return the list when all of the distances has been calculated and stored.

Let $v_0 = 5\,\text{m/s}$, $\Delta t = 0.0001\,\text{s}$ and the list of coefficients of friction be `[0.62, 0.3, 0.45, 0.2]`.

c)

Define a test function which tests if the list from b) are equal to the analytical distance $x\left(\frac{v_0}{\mu g}\right)$ at time $t = \frac{v_0}{\mu g}$ (this is the calculated time each block will use) for every coefficient of friction μ. A suitable threshold to use when testing for equality between the calculated and analytical values is 10^{-7} in this task.

Call the test function to test if you have gotten results which are more or less equal to the analytical distance for every μ.

Filename: `block_frictions2.py`

Exercise 4.1 - Heisenberg's uncertainty relation

Heisenberg proved in 1927 that we cannot exactly know the velocity of an particle and its position *at the same time*. This means that if we know precisely the velocity of a particle, we will not be able to have a precise measurement of the position of the particle, and vice versa.
This can be written mathematically as:

$$\Delta x \Delta p \geq \frac{h}{4\pi}$$

where Δx is the uncertainty (a measurement of how precise a measurement is) of the position of the particle and Δp is the uncertainty of the momentum of the particle[4].

We will use that Vi bruker at $h \approx 6.626 \times 10^{-34}\,\mathrm{J\,s}$.

a)

Write a program which takes Δx and Δp as arguments on the command-line. The program must then test the arguments in an `try-except`-block and display an appropriate message if the user has not provided the program enough arguments or the given Δx and Δp cannot be converted to floats.

b)

The program must then call a test function which you have implemented. The test function must receive Δx and Δp as parameters and test whether the uncertainty principle holds for the given uncertainties. An appropriate message has to be displayed if the principle does not hold. Let the function use `assert` to test the relation.

Test your program where

$\Delta x_1 = 3.101\,65 \times 10^{-9}\,\mathrm{m}$, $\Delta p_1 = 1.7 \times 10^{-26}\,\mathrm{kgm\,/s}$

and

$\Delta x_2 = 5.2 \times 10^{-32}\,\mathrm{m}$, $\Delta p_2 = 1 \times 10^{-3}\,\mathrm{kgm\,/s}$.

The uncertainties Δx_1 and Δp_1 does not violate the principle. However, the uncertainties Δx_2 and Δp_2 will violate with the principle (and your program should therefore display an error message for this case).

Hint: A low value such as 1.7×10^{-26} can be represented on the command-line as `1.7e-26`.

Filename: `uncertainty_Heisenberg.py`

Exercise 4.2 - Particle accelerator

An electric field of strength E will apply a force $F = qE$ onto a particle with electric charge q.
In one dimension, with an initial velocity v_0, the particle's velocity and position will be given as

$$x(t) = v_0 t + 0.5\frac{qE}{m}t^2$$

and

$$v(t) = v_0 + \frac{qE}{m}t$$

[4]The momentum is defined by the velocity of the particle. If we know the momentum and the mass of the particle, then we will know the velocity of it. More precisely, the momentum p is defined as $p = mv$ where m is the mass of the particle and v the velocity of the particle

a)

Consider an electron, which has a mass $m \approx 9.1 \times 10^{-31}\,\text{kg}$ and electric charge $q \approx -1.6 \times 10^{-19}\,\text{C}$, caught in an electric field of strength $E = 0.02\,\text{N/C}$. [5]

Make a program that asks the user for values for v_0 and t, and prints the position and velocity of the electron.

Test your program by checking the electrons position and velocity at $t = 15\,\text{s}$ with $v_0 = 220\,\text{m/s}$.

b)

Rewrite your program to take v_0 and t, as well as q and m from the command line. Use a try/except block to initialize the variables, in case the user provides too few, or they cannot be converted to floats. In that case, ask the user for the parameters as input, like in exercise b).

Protons have mass $m \approx 1.67 \times 10^{-27}\,\text{kg}$ and electric charge $q \approx 1.6 \times 10^{-19}\,\text{C}$. Neutrons have virtually the same mass as protons, and no electric charge.

Check the position and velocity of these two particles with the same parameters as in exercise a).

Filename: `particle_accelerator.py`

Exercise 4.3 - Relativistic user input

In exercise 2.2 we compared the classical and relativistic momentum of an object with mass m and velocity v.

$$p_{clas} = m \cdot v$$
$$p_{rel} = m \cdot v \cdot \gamma, \qquad \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

a)

Write a program which asks the user for the velocity and mass of an object, and calculates the momentum of the object. Write your program such that it uses the classical formula if the given velocity is small enough that the classical formula is a good approximation to the relativistic one. Otherwise, use the relativistic formula. Use your results from exercise 2.2 to decide when the classical formula is a 'good enough' approximation.

If you haven't done 2.2, you may use the classical formula for velocities lower than $v = 1/3c \approx 10^8\,\text{m/s}$.

b)

Rewrite your program such that m and v are taken as arguments in the terminal, rather than as keyboard input.

c)

You shall now expand your program from b), such that it thoroughly checks if the input you get is both physically valid, and won't break your program. Your program should include the following functionality:

- A Try/Except block for initiating the terminal arguments. The possible errors these can give are a `IndexError` if the number of terminal arguments is less than three (the program itself, m, and v.), and a `ValueError` if one of the inputs cannot be converted to a float.

[5] We consider all movement to be one dimensional, and all forces and velocities to have the same positive direction. This means that an electric field with a positive field strength will accelerate the electron in the negative direction, due to its negative charge.

- Raise a `ValueError` if the given mass is not positive.
- Raise a `ValueError` if the absolute value of the velocitiy is larger than the speed of light.

Include an explanatory text to the user for each of the errors.

Filename: `momentum_input.py`

Exercise 4.4 - How large friction?

Suppose you have several books placed at an inclined slope with angle θ.

A book will precisely begin to slide when the friction force f is:

$$f = \mu_s mg \cos\theta$$

where m is the mass of the book (in kg), μ_s the static coefficient of friction and $g = 9.81\,\mathrm{m/s^2}$. The static coefficient of friction varies depending on which material the surface is made of.

We are given a collection of data, `slide_books.dat`, of the books. The file consists data of which mass m each book has, different angles θ for the inclined slope and static coefficients of friction.

a)

Write a program which opens `slide_books.dat` and reads the values for m, θ and μ_s. Make sure that your program is written such that it can read and store arbitrary number of values for m, θ and μ_s.

b)

Extend your program from a) and use the stored values to calculate how large the friction must be for each book to slide for every angle θ and every static coefficient of friction μ_s.

Make sure that your program converts the angles θ to radians when calculating the friction force f! To do the converting, use the following relation to convert from degrees to radians:

$$\text{angle in radians} = \frac{(\text{angle in degrees}) \cdot \pi}{180}$$

The results should then be written to file. The formatting must be on a similar form as the following example:

```
--- Book with mass 4.33 kg ---
theta = 0.93 rad
coefficient of friction = 0.34
needed friction force is 8.69 N

coefficient of friction = 0.2
needed friction force is 5.11 N

coefficient of friction = 0.55
needed friction force is 14.06 N

coefficient of friction = 0.4
needed friction force is 10.23 N
```

Filename: `slide_books_friction.py`

Exercise 4.5 - Newton's law of gravitation

We saw in exercise 2.4 how the gravitational force between two objects interacts using Newton's gravitational law:

$$F = G\frac{m_1 m_2}{r^2}$$

where m_1 and m_2 are the mass of the objects and r the distance between them.

The constant G is the gravitational constant which has value

$$G = 6.674 \times 10^{-11}\,\mathrm{m^3\,kg^{-2}\,s^{-1}}$$

We will see how much the gravitational force interacts between one object with mass M kg and N other objects. The i-th object has mass m_i kg and distance r_i from the object with mass M.

Write a program which reads a file and uses the values from the file to calculate and present the total gravitational force which acts on the object with mass M.

You can assume that the files that your program should be able to read has the same structure as this example:

```
0.2     0.0034
3       0.495e-5
0.15    2
2.5     0.000029348
1.3948  4.56
4.5     0.0294
```

The name of this file is `newton_objects.dat` which can be found here: (lenke-til-fil).

The i-th line contains information about the i-th object. The first value is m_i and the second r_i.

The program should take the mass M as first argument in the command-line and a filename to the file containing data of the N objects as a second argument.

The program must then do the following in a `try-except`-block:

- Display an appropriate message and quit if the user forgets to write the mass M or the filename as arguments to the commandline.
  In other words, the program must display the error message and quit if an `IndexError` has occurred.

- Display an appropriate message and quit if the user writes in something for M which cannot be converted to float.
  The error which will occur is a `ValueError`.
- Display an appropriate message and quit if the user writes a path to a file which does not exists the program.
  The error which occur will be an `IOError`.

Let your program read `newton_objects.dat` and set $M = 0.7\,\mathrm{kg}$.

Filename: `newton_gravitation_file.py`

Exercise 5.1 - Pulling crates

We will look at the case where N connected crates are being pulled by someone:

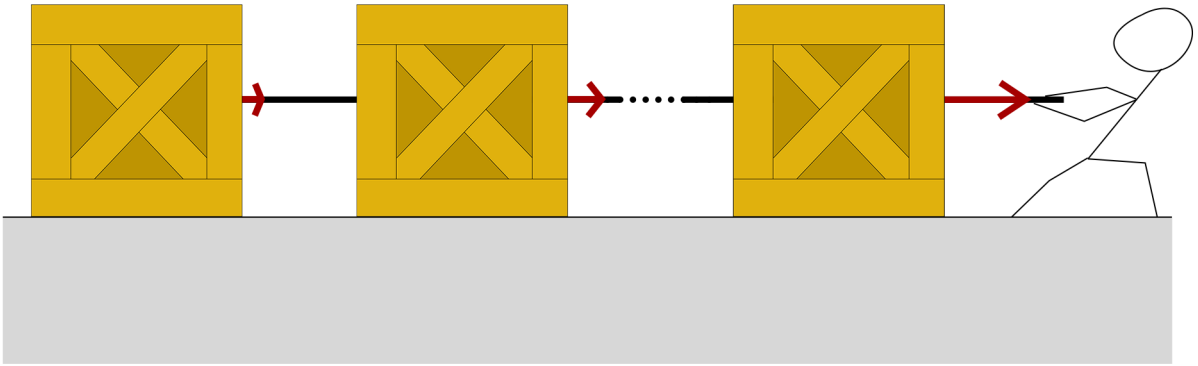

Figure 2: Illustration of a person pulling some creates. The dots indicates that there are arbitrary number of crates and the red arrows the direction and strength of the force exterted on the crates.

a)

We have measured that every i-th create is influenced by a force of $30 + 5 \cdot i\,\mathrm{N}$.

Write a program which generates a list of the forces which acts on every i-th create. Here we will look at $N = 10$ crates.

b)

It turns out that the forces measured in a), turned out to be too large. Divide the values form a) by 2 *without* using for or while loops. Find the sum of the values, also without for or while loops, and then print the result.

Filename: `pull_crates.py`

Exercise 5.2 - Plotting relativistic against classical momentum

The momentum of an object with mass m and velocity v is defined differently in classical and relativistic physics:

$$p_{clas} = m \cdot v$$
$$p_{rel} = m \cdot v \cdot \gamma, \qquad \gamma = \frac{1}{\sqrt{1 - \frac{v^2}{c^2}}}$$

The speed of light is defined as $c \approx 3 \times 10^8\,\mathrm{m/s}$.

Set $m = 5\,\mathrm{kg}$, and plot the two functions in the same plot, with velocities evenly distributed in the interval $v \in [0c,\ 0.9c] = [0\,\mathrm{m/s},\ 2.7 \times 10^8\,\mathrm{m/s}]$.

Filename: `momentum_plot.py`

Exercise 5.3 - capacitor discharge

Physical introduction: A capacitor is simply two metal plates set up parallell to each other. We can charge up each plate with positive and negative electic charges by connecting them to a

battery. If we remove the battery and connect the two plates together, the charges will flow from the negative plate to the positive plate, and the capacitor will discharge. To avoid the electrical current becoming infinite, we connect a resistor between the plates. We now have an RC-circuit.

The charge Q of a capacitor that discharges in a RC-circuit, is given as

$$Q(t) = CV\mathrm{e}^{-t/RC}$$

The following program calculates this discharge for $n = 1000$ time-steps over an interval $t = 10\,\mathrm{s}$. The capacitor has a *capacitance* of $C = 0.007\,\mathrm{F}$, an initial voltage $V_0 = 50\,\mathrm{V}$, and the resistor has a resistance $R = 350\,\Omega$.

```
import numpy as np
import matplotlib.pyplot as plt

def I(t, R, C, V0):
    return C*V0*np.exp(-t/(R*C))

V0 = 50.0
R = 350.0
C = 0.007

t = 10
n = 1000
dt = float(t)/n

t_list = []
I_list = []
for i in range(n):
    t_list.append(dt*i)
    I_list.append([I(t, R, C, V0))

plt.plot(t_list, I_list)
plt.show()
```

Copy the program and confirm that it works as intended.

Vectorize the program such that `t_list` and `I_list` is replaced by numpy vectors, and the for loops are replaced by vector operations. Confirm that the result remains the same.

Filename: `capacitor_vectorization.py`

Exercise 5.4 - Planck's Law

Planck's law describes how much energy a black body (usually a star) emmits at different wavelengths of electromagnetic radiation. The law is given as

$$B(\lambda) = \frac{2hc^2}{\lambda^5}\frac{1}{\mathrm{e}^{\frac{hc}{\lambda kT}} - 1}$$

where T is the temperature of the star, $h = 6.62 \times 10^{-34}\,\mathrm{J\,s}$ is Planck's constant, $k = 1.38 \times 10^{-23}\,\mathrm{J/K}$ is Boltzmanns constant, and the speed of light is still $c \approx 3 \times 10^8\,\mathrm{m/s}$. The plots that comes from plotting this function are called 'Planck curves', and is a common sight in physics books.

a)

Use the temperature of the sun, $T = 5800\,\mathrm{K}$, and plot $B(\lambda)$ against wavelengths in the interval $\lambda \in [10\,\mathrm{nm},\ 3000\,\mathrm{nm}]$. Note that these wavelengths are in nanometers, while the function takes wavelengths in meters.

b)

Include also the Planck curves of the stars Alpha Centauri A ($T = 25\,000\,\mathrm{K}$), and Proxima Centauri ($T = 2400\,\mathrm{K}$), in the same plot as the Sun.

c)

Wien's displacement law tells us that we should find the peak of the Planck curve at the wavelength

$$\lambda_{max} = \frac{b}{T}$$

where $b = 2.9 \cdot 10^{-3}\,\mathrm{Km}$ is called Wien's displacement constant.

Expand the plot from b) by adding three vertical lines at $x = \lambda_{max}$ for each of the three stars, and confirm that the lines corresponds to the peak of the curves.

Hint: You can make vertical lines in Python with the function `matplotlib.pyplot.axvline(x=)`

Filename: `Planck_curves.py`

Exercise 5.5 - Oscilating spring

A rock of mass m is hung from a spring, and dragged down a length A. Upon release, the rock will oscilate up and down with a vertical position given as

$$y(t) = A \cdot \mathrm{e}^{-\gamma t} \cos\left(\sqrt{\frac{k}{m}} \cdot t\right)$$

Here, $y = 0$ corresponds to the vertical position of the rock when just hanging still from the spring. Set $k = 4\,\mathrm{kg/s^2}$ and $\gamma = 0.15\,\mathrm{s^{-1}}$. Assume the rock has a mass $m = 9\,\mathrm{kg}$, and that you drag the rock down a length $A = 0.3\,\mathrm{m}$.

a)

Create empty arrays `t_array` and `y_array` of size 101. Use a for loop to fill them with time values in the range $[0\,\mathrm{s},\ 25\,\mathrm{s}]$, and the corresponding $y(t)$ values.

b)

Vectorize your program by instead using the `linspace` function to generate the array `t_array`, and send it into a function `y(t)` to genereate the array `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. Get correct units on each axis.

Filename: `oscilating_spring.py`

Exercise 5.6 - Planatary motion

The motion of a planet orbiting a star can be described by its distance to its star as a function of its anglular position:

$$r(\theta) = \frac{p}{1 + e\cos(\theta)} \tag{5}$$

where

- θ simply represents where in its orbit the planet is (in radians). $\theta = 0$ is angle where the planet is closest to it's star.
- p is just a parameter representing the size of the orbit, which we will set to 1 AU. [6]
- e is the *eccentrisity* of the orbit, which says how eliptical the orbit is. $e = 0$ represents a circular orbit, and $e \geq 1$ means that the planet is not even in an orbit around the star, and simply traveling by.

a)

Plot the planet's distance to the star as a function of angular position for eccentrities $e = 0$, $e = 0.5$, and $e = 0.8$ in the same plot. Use $\theta \in [0,\ 2\pi]$ to include one full orbit of the planet. Get correct units on both axis.

b)

You shall now plot the actual orbits of the planet, for each of the three eccentrities. To make the plotting easier, we decompose equation (5) into x and y coordinates:

$$x(\theta) = r(\theta)\sin(\theta), \qquad y(\theta) = r(\theta)\cos(\theta)$$

Use these equations to plot $y(\theta)$ against $x(\theta)$, still using $\theta \in [0,\ 2\pi]$, for each of the three eccentrities.

Remember that both $r(\theta)$ and θ are arrays, and the resulting $x(\theta)$ and $y(\theta)$ are also arrays, and can simply be plotted against each other as they are.

Hint: Matplotlib doesn't keep axis proportional by default, meaning that a circular motion might look elliptical on the plot. If you include the line `matplotlib.pyplot.axis('equal')`, the x and y coordinates will be of equal scale. You can also include the star itself, to make the plot more visually appealing. Do this by plotting a point at (0,0), and specify it as a yellow dot: `matplotlib.pyplot.plot(0, 0, 'yo')`

Filename: `planetary_motion.py`

Exercise 5.7 - Estimate the value of Planck's constant

It is possible to use some data points to estimate the value of Planck's constant h. To do so, we will use Einstein's equation describing the photoelectric effect[7].

The equation found by Einstein is:

$$E_k = hf - W$$

where h is Planck's constant, f is the frequency of the light, W is the work which is needed to free the electrons and E_k is the electrons maximum kinetic energy.

Suppose we have gotten some measurements from one experiment where we have sent light with different frequencies f such that the electrons got a maximum kinetic energy $E_{k,max}$:

$f/10^{15}$ Hz	1.18	0.96	0.82	0.74
$E_{k,max}/10^{-19}$ J	3.12	1.57	0.8	0.22

The frequencies of light f in the table has been divided by 10^{15}. Similar goes for the maximum kinetic energies E_k in the table where the values haven been divided by 10^{-19}. Remember, when you have to use these values it is necessary to multiply the values of f and $E_{k,max}$ by 10^{15} and 10^{-19} respectively!

[6] This is the average distance from our sun to the earth. We will use it as our distance-unit in all plots and calculations.

[7] The equation came to life when Einstein wanted to explain why the photoelectric effect occurs, i.e why electrons can set free from metal if the light as a high enough frequency.

a)

Write a program which plots the measurements as points. The values for f and $E_{k,max}$ should be in arrays. Plot the values for f along the x-axis and the values for $E_{k,max}$ along the y-axis.

Hint: To plot the points, it is enough to send an additional parameter to `plt.plot` to indicate that you want to plot points instead of lines. For instance, if you want red points, you could send `'ro'` as the additional parameter to `plt.plot`.

b)

Numpy has a function named `np.polyfit(x,y,1)` which finds the slope a and the y-intercept b to a straight line $ax + b = y$ which is *as close as possible* the given points for `x` (values along the x-axis) and `y` (values along the y-axis).

If we look back at Einstein's equation, we can see that $a = h$ and $b = -W$ in our case[8]! We can therefore use our measurement to estimate h and W.

Extend your program from a) such that it calls `np.polyfit(`f`,` $E_{k,max}$`, 1)` and stores the values for h and $-W$. Let your program write out the value of the estimated h.

Your program should give the following estimate for h (in unit J s):

```
6.50987e-34
```

which is pretty close to the value $h = 6.626 \times 10^{-34}$ J s!

c)

Plot the data points from a) along with the straight line $y = ax + b$ where a and b are the estimated values for h and $-W$ respectively.

Filename: `estimate_h.py`

Exercise 5.8 - Pendulum

We will investigate how to program a simple model of the motion of a pendulum.
The positions along the x- and y-axis of the pendulum can be modeled as such:

$$x(t) = L \sin(\theta(t))$$
$$y(t) = -L \cos(\theta(t))$$

where $\theta(t)$ is the angle of the pendulum at a time t and L is the length of the pendulum.

By assuming that the pendulum swings back and forth at only small angles, it is possible to find that $\theta(t)$ is:

$$\theta(t) = \theta_0 \cos(\omega t)$$

where θ_0 is the angle the pendulum starts at and $\omega = \sqrt{\frac{g}{L}}$ (where $g = 9.81\,\mathrm{m/s}$) is the angular velocity[9].

Write a program which calculates $x(t)$ and $y(t)$ for $N = 1000$ timepoints between 0 og $T = 1$ seconds.
Let $\theta_0 = \frac{\pi}{6}$ and $L = 0.75\,\mathrm{m}$. The program must then plot the positions along the x- and y-axis.

Important: Your program must not use any for- og while loops to do the calculations. It is necessary to use arrays.

Filename: `pendulum.py`

[8] If you are unsure about this statement: plug in $a = h$,$b = -W$,$x = f$ and $y = E_{k,max}$ into the equation for a straight line and see that you get Einstein's equation

[9] The angular velocity tells us how fast the pendulum swings.

Exercise 5.9 - Projectile motion

We will now take a step back to exercise 3.3 where we made a program which simulated a simple game of throwing a ball at a target. In this exercise we will look at how the motion of the ball is dependent of the angle θ.

The motion of the ball along the x-axis $x(t)$ and the motion of the ball along the y-axis $y(t)$ can be modeled as such:

$$x(t) = v_0 t \cos\theta$$
$$y(t) = -\frac{1}{2}gt^2 + v_0 t \sin\theta$$

where $g = 9.81\,\mathrm{m/s^2}$.

Write a program which generates 1000 values for t between 0 and $T = \dfrac{3.5}{v_0 \cos\theta}$ s.Let $v_0 = 16\,\mathrm{m/s}$ and plot $x(t)$ together with $y(t)$ for $\theta = \dfrac{\pi}{6}, \dfrac{\pi}{4}$ and $\dfrac{\pi}{3}$.
Make sure that your program uses arrays and is vectorized by passing the arrays to functions.

Filename: `plot_throw_ball.py`

Exercise 5.10 - Angular wavefunction

Short introduction: In this exercise we will take a closer look at some functions which are called *Legendre functions* which takes $\cos\theta$ as argument. The functions is used, along with other functions, to describe a wavefunction to a particle in three dimensions. A wavefunction describes at which state a particle has and can be used to find the probability of finding the particle in a specified area.

We will look closer at the following functions:

$$P_2^0(\theta) = \frac{1}{2}\left(3\cos^2\theta - 1\right)$$

$$P_2^1(\theta) = 3\sin\theta\cos\theta$$

$$P_2^2(\theta) = 3\sin^2\theta$$

where θ has values between 0 and π.

Write a program where P_2^0, P_2^1 and P_2^2 are functions. Let θ be an array which has 1000 uniformly distributed values between 0 and π. The program should do the following for every function P_2^i:

1) Calculates $R_i = \left|P_2^i(\theta)\right|$
2) Calculates $a = R_i \sin\theta$ and $b = R_i \cos\theta$ without using a for loop. Here you will see that having defined the functions and let θ be an array will be of much help.
3) Plots b along the y-axis together with a along the x-axis and a mirrored along the y-axis. To do the mirroring, one could simply plot $-a$ along the x-axis together with b along the y-axis. The program is therefore supposed to make two graphs in the same plot, preferably both graphs having the same color.

Hint: Take advantage of the fact that functions can also be stored in lists. Use a for loop to iterate through the functions.

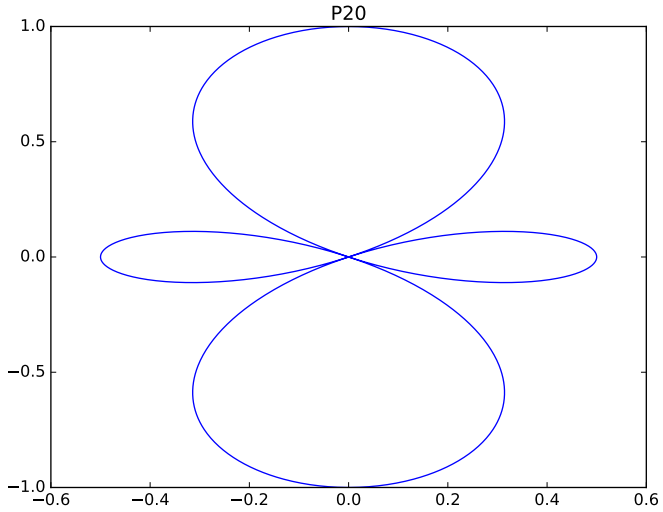

Figure 3: An example of what you should get after plotting $P_0^2(\theta)$.

Filename: `angular_wavefunction.py`

Exercise 6.1 - Solar system

The following file contains information about average distance from the sun, the mass, and the radius of a selection of celestial bodies in our solar system.

file reference

```
Object     Distance[km]     Mass[kg]     Radius[km]

Sun        0                1.99e30      695700
Mercury    5.79e7           3.30e23      2440
Venus      1.08e8           4.87e24      6052
Earth      1.50e8           5.97e24      6378
Mars       2.28e8           6.42e23      3396
Jupiter    7.79e8           1.90e27      71492
Saturn     1.43e9           5.68e26      60268
Uranus     2.87e9           8.68e25      25559
Neptune    4.49e9           1.02e26      24764
Pluto      5.90e9           1.46e22      1185
```

a)

Read the file, and write it's content into three dictionaries, `distance`, `mass` and `radius`, with the name of the celestial objects as keys.

b)

Make a new dictionary, `densities`, which contains the density of each celestial object. Density is mass divided by volume, in kg/m^3 (note that the radius in the file is not in meters). You may assume that all the objects are perfect spheres, meaning that their volumes are given as $V = 4/3\pi r^3$.

c)

Write the information, including the densities, to a new file, with more or less same layout as the original (your code from exercise a) should be able to read it).

d) (Voluntary exercise)

Combine the information into a nested dictionary, `solar_system`, such that for instance `solar_system[Jupiter][radius]` would yield the radius of Jupiter.

Test your new dictionary by calculating how many earths you would have to align to reach from the Sun to Pluto.

Filename: `solar_system_dict.py`

Exercise 6.2 - Read and use physical constants

It is quite often we have use for a selection of physical constant to model different systems in physics. However, it might get challenging after a while to memorize all the values by heart.

Instead of going the old way by memorizing the values, we can now create a file containing the physical values and then write a program which extracts the values to use in a specified problem.

a)

Write a program which reads `physics_constant.dat`.
The file has a format where the names of the constant of arbitrary number of characters is written before a colon, ':'. The constant's symbol is to be found after the colon, after the symbol is the value of the constant and at the end of each line is the unit of the constant.

The program must store the constants in a dictionary. The symbol of the constant have to be the key to get the corresponding value of the constant.

b)

Bohr found a model on how one could model the energy levels E_n to a hydrogen atom. The levels are found to be:

$$E_n = -\frac{k_e^2 m_e e^4}{2\hbar^2}\frac{1}{n^2}$$

Use your dictionary from a) to get the necessary constant which is used in the model and test if your program gives that:

$$\begin{aligned}\frac{k_e^2 m_e e^4}{2\hbar^2} &\simeq 2.18 \times 10^{-18}\,\mathrm{J} \\ &\simeq 13.6\,\mathrm{eV}\end{aligned}$$

where k_e is Coulomb's constant, m_e the mass of an electron, e the elementary charge and $\hbar$ is Planck's reduced constant (you can find it in the file as `hbar`).

Filename: `constants_hydrogen.py`

Exercise 6.3 - Dynamical frictions

A dynamical friction force from a surface on which moving objects moves across [10].

The dynamical friction force μ_D is

$$\mu_D = N\mu$$

where μ is the friction coefficient between the material of the surface and the material of the object and N is the normal force which acts on the object [11].

We assume that the object moves in horizontal direction. Through this assumption, one can find that the normal force N is:

$$N = mg$$

where m is the mass of the object and $g = 9.81\,\mathrm{m/s^2}$.

a)

Write a program which reads the file `friction_coefficients_data.dat` (which can be found here: link-to-file) which is a table consisting of friction coefficients for selected pairs of materials.

The first line tells us which pair of materials we are looking at, and the second line consists of corresponding coefficients of friction .

The program must store the pairs of materials and the values of their coefficient of friction in seperate lists. You do not need to worry about (yet) the hyphen (`'-'`) between the materials.

b)

Now your program has two lists; one for the pairs of materials and one for the values of the coefficients of friction.

For every pair of material, let the program extract the material to the object (material given before the hyphen), material to the surface (after the hyphen) and corresponding coefficient of friction μ.

Let the program calculate the dynamical friction force. The force must then be displayed such that it is clearly stated which material the object and the surface is made of.

[10] The contact points between the objects and the surface must be in movement - hence the force is dynamic.

[11] The normal force is the force which acts on the object from the surface where the object and the surface is in contact with each other.

Let the object have mass $m = 2.5\,\text{kg}$.

Filename: `dynamic_friction_pair.py`

Exercise 6.4 - On Earth

We have been working a lot with the gravitational acceleration $g = 9.81\,\text{m/s}^2$ without worrying about where on Earth we are located. In fact, the value of g is dependent on where we are located (more precisely: on which altitude and latitude we are located)[12]!

In the file `data_different_g.dat` can you find (rounded) calculated values for g for a selection of cities.

a)

Write a program which reads the file `data_different_g.dat` and creates a dictionary where g for every city is stored.

For instance, the acceleration of gravity in Stockholm if $g = 9.818\,\text{m/s}^2$. Your dictionary should be made such that if you send `Stockholm` as a key to the dictionary it should return 9.818, the gravitational acceleration in Stockholm.

We will assume that a city will either have one space in its name (for instance San Francisco), or no space (for instance Paris).

The last line containing only '`-`'s marks the end of the given cities.

Hint: Use `split()` for every line to get a list of strings which were originally separated by spaces. If a name contains one space the list will be of 1 unit longer than the list with name without space. One could therefore check the length of the lists to see if the name of the city has a space or not.

Slicing will also be useful when extracting the values of g.

b)

Extend your program from a) such that it reads the file `on_Earth.dat`. The file contains several lines, each consisting of two cities seperated by `to`.

For every city in `on_Earth.dat`, your program must extract the name of the two cities and use the dictionary from a) to print the acceleration of gravitation of both cities seperated by a `to` (in the same manner as in the file `on_Earth.dat`, just with the corresponding values of g instead of the name of the cities).

This example shows how the format should be:

```
9.813 to 9.8
9.805 to 9.816
9.82 to 9.78
```

The last line in the file however, marks the end of the file. You can assume that no city begins with '`On`'.

Hint: Again, `split()` is your friend. Exploit the assumption that the cities has names containing either one or no space, no city begins with '`On`' and there is always a `to` which separates the two cities.

For the second city (after '`to`'), one could use `' '.join(line[start:])` to get the whole name. The index `start` is dependent on the first city having a name with or without a space.

Filename: `different_g.py`

[12] This is due to the centrifugal force - a force which is due to the rotational motion of the Earth.

Exercise 7.1 - Planet class

a)

Write a class `Planet`, with a contructor which takes in the planet's *name*, *radius*, and *mass*, and saves them. Include also a method `density`, which returns the density of the planet in $\mathrm{kg/m^3}$, and a method `print_info`, which writes all known information about the planet to the terminal, including it's density. (call the `density` method from inside `print_info` method)

b)

Create an instance of the class called `planet1`, representing Earth. Add a new attribute to this instance called `population`, with a value `7497486172`.

Add the following line to your program, and ensure that you get the print stated below:

```
print planet1.name, " has a population of ", planet1.population
```

```
>>> Earth has a population of 7497486172
```

Filename: `Planet.py`

Exercise 7.2 - Coulomb's law

Coulomb's law describes the force which interacts between two charged point masses [13].

We will now make a simple model to find the force interacting between the two charged particles.

Coulomb's law states that:

$$F = k_e \frac{q_1 q_2}{r^2}$$

where $k_e = 8.988 \times 10^9\,\mathrm{Nm^2\,/C^2}$, q_1 is the charge of one particles, q_2 is the charge of the other particle and r is the distance between the particles.

a)

Define a class which takes in the position of the particle (as an array consisting of the x- and y-coordinate of the particle's position) and charge q of the particle as parameters to the constructor.

The class must also have a function which takes in another particle as parameter, and calculates the force which interacts between the particles using Coulomb's law. The absolute value of the found force should then be returned.

Hint: To calculate the distance r between the particles, one can use `np.linalg.norm`.

b)

Define a test function *outside* the class definition to test if your implementation of Coulomb's law gives you desired results. Let your program call on the test function.

A test your program should perform, is to create two particles whereas the particles has distance 30 mm between them. Let the charge of one particle be $-1.602 \cdot 10^{-19}\,\mathrm{C}$ and the charge of the other particle be $1.602 \cdot 10^{-19}\,\mathrm{C}$.

Your program should calculate that the force between the particles is:

$$F = 2.565\,833\,688 \times 10^{-15}\,\mathrm{N}$$

Filename: `Particle_Coulomb.py`

[13] A point mass is a very small objects which is so small that it does not occupy any space, but is still present. Small bodies, such as elementary particles is usually thought as point masses. Coulumb's law can also be applied to larger, sphere-shaped objects.

Exercise 7.3 - Unidentified flying object

a)

You shall write a class, `ObjectMovement`, which calculates the movement of an object which flies freely through the air. We ignore air resistance, and do our calculations in two dimensions - one horizontal axis x, and one vertical axis y.

Begin the class with a constructor, for saving the object's initial position `(x0, y0)` and initial velocity `(vx0, vy0)`. Include also the acceleration of gravity, $a = -9.81\,\mathrm{m/s^2}$, preferably as a key-word variable.

Give the class two methods `position` and `velocity`, which takes in a timepoint t, and returns the objects position or velocity at that time. You may use the following equations of movement:

$$s(t) = s_0 + v_0 t + \frac{1}{2}at^2, \qquad v(t) = v_0 + at$$

Remember that you can decompose the equations in x and y direction and calculate them individually. Gravity's acceleration is then only in the y direction.

Write also a test-function, `test_pos_vel`, which tests that the position and velocity given form the `position` and `velocity` methods matches exact values from a calculator within a tolerance.

b)

Because the only force acting on the object is gravity, which is a conservative force, the sum of *kinetic* and *potential* energy should be conserved (equal at any two points in time). Kinetic energy is given as $E_k = \frac{1}{2}mv^2$, and potential energy (on the surface of the Earth) $E_p = mgy, \quad g = 9.81\,\mathrm{m/s^2}$.

Remember that $v = \sqrt{v_x^2 + v_y^2}$.

Write a function `test_energy_conservation`, which calculates E_k and E_p at two chosen timepoints, and confirms that they are equal within a tolerance. Set $m = 1\,\mathrm{kg}$.

Filename: `UFO.py`

Exercise 7.4 - Runners at different inclined slopes

We will now define a class which can be used to represent a runner which runs down different hills with different angles of inclination.

a)

The runners has mass $m\,\mathrm{kg}$ and an initial velocity of $v_0\,\mathrm{m/s}$.

Define the class which represents a runner and make an instance of the class with mass $m = 30\,\mathrm{kg}$, initial velocity $v_0 = 5\,\mathrm{m/s}$ and $\theta = 30°$. The mass m, initial velocity v_0 and the inclination angle θ must be given as parameters to the constructor.

b)

Extend your class from a) such that it contains a function `__str__` (this function is called automatically when we try to print the instance).
The function will return a string which contains information about the mass m of the runner, initial velocity v_0 and the inclination angle on the hill which the runner sprints at. Make sure that the returned string clearly states what each value represents.

If we try to print the instance from a), it should get something like

```
Sprinter with
mass: 80 kg
initial velocity: 5 m/s
angle: 30 degrees
```

c)

Extends the class such that it contains a function which calculates how much time the runner will use to run d meters down the hill.
We will make a simple assumption that the driving force of the runner (the force which makes the runner run) is $F_{\mathrm{d}} = 400\,\mathrm{N}$.
Extend your class with a function which takes the value of d as parameter. The time used by the runner to sprint d meters can be shown to be

$$T = -\frac{v_0}{g\sin\theta + \frac{1}{m}400} + \frac{\sqrt{v_0^2 + 2d\left(g\sin\theta + \frac{1}{m}400\right)}}{g\sin\theta + \frac{1}{m}400}$$

Use your newly defined function to write out how much time the runner from a) will use.

Filename: `Runner.py`

Exercise 7.5 - Center of mass

When we are working with several objects, it is often convenient to work with the *center of mass* of the objects. The center of mass is a position which is often used as a reference point.

For N objects where j-th object has mass m_j and position $\vec{r}_j$, the center of mass is defined to be:

$$\vec{R} = \frac{m_1\vec{r}_1 + m_2\vec{r}_2 + \cdots + m_N\vec{r}_N}{m_1 + m_2 + \cdots + m_N}$$

In this task we will represent the object's position by using arrays with length 2. The first element in the array will represent the x- coordinate of the object, and the second element will represent the y-coordinate of the object.

We will now look at the center of mass between a selection of particles.

a)

Define a particle class which takes the position of the particle and the mass of the particle as parameters to the constructor.

b)

Initialize five instances of particles where j-th particle has position $r_j = (j, 2\cdot j)$ and mass $m_j = j \cdot 10^{-30}\,\mathrm{kg}$.

Hint: One way one could initialize the instances, is by creating them through a for loop and store them in a list.

c)

Let your program find the center of mass between the five particles which you initialized in b).

Filename: `center_of_mass.py`

Exercise 8.1 - Conservation of energy

In this exercise we will use a random number generator to confirm a physical phenomena. This phenomena is called *conservation of energy*. Conservation of energy is an important phenomena which we often use to solve and describe physical systems.

The total energy E of a body can be divided into two groups: the kinetic energy E_k and potential energy E_p. There exists also some other different types of energies, but we will only focus on the kinetic energy and the potential energy.

The total energy E of the body can be written as $E = E_k + E_p$. By having conservation of energy means that for *any time* t we measure the total energy of the body we will always get the same result.

We can write this as

$$E_0 = E_1$$
$$E_{k,0} + E_{p,0} = E_{k,1} + E_{p,1}$$

where E_0 and E_1 are the total energies measured at the different times t_0 and t_1.

The kinetic and potential energy of a body which is only affected by gravity is

$$E_k = \frac{1}{2}mv(t)^2$$
$$E_p = mgy(t)$$

Assume that we have a ball which is being thrown straight up. The height $y(t)$ at a certain time t can by found by using

$$y(t) = -gt^2 + v_0 t$$
$$v(t) = -gt + v_0$$

where $g = 9.81\,\mathrm{m/s^2}$ and $v_0\,\mathrm{m/s}$ is the velocity the ball is thrown with.

Define a test function which takes N, m and v_0 as parameters and tests if we have conservation of energy.

Let the function generate N random values for the time t_0 and t_1. The values have to be between 0 and $\frac{v_0}{g}$.
For every value of t_0, the program have to calculate E_0. The same should be done for every value of t_1, that is calculate the total energy E_1.

Test if the total energies E_0 and E_1 are almost equal. Use `assert` and write a descriptive message if it happens that your program do not have conservation of energy.

Let $N = 100$, $m = 0.057\,\mathrm{kg}$ and $v_0 = 17\,\mathrm{m/s}$ and call the test function using these parameters.

Hint: A lot of this code can be vectorized - take a look at the end of page 510 in the book for how tests could be vectorized.

Filename: `check_energy_conservation.py`

Exercise 8.2 - Randomized Decay

We have previously studied how a mass of a radioactive material decays over time, using a purely analytical formula:

$$N(t) = N_0 \mathrm{e}^{-t/\tau} \tag{6}$$

With our knowledge of random numbers, we can now take a more numerical approach, modelling each atom in the material individually.

Each atom in a radioactive material has a chance p of decaying each second. We will model the material as an array, with each element in the array representing an atom. The elements can have values either 1, representing that the atom has not yet decayed, or 0, representing that the atom has decayed.

We will be looking at the nitrogen isotope *nitrogen-16*, where every atom has a chance $p = 0.0926$ (or 9.26%) of decaying each second.

a)

We will first study a single second of decay. Create an array of 40 atoms. Loop over each atom, and decide if it decays or not. You can do this by generating a random number in the interval $[0,\ 1]$, and compare it to p. Remember to generate a new number for each atom, or they will all end up either decaying or not at the same time.

Print the new array of atoms and confirm that a few of the atoms actually decayed.

b)

Repeat the process for 30 seconds by wrapping your code in another for loop.

Implement also another array that keeps track of how many atoms remains over time. You can do this by using the `numpy.sum()` function on the array of atoms after each time-step[14]. Make sure that both the original amount of atoms before the first decay, and the last number of atoms after the last decay are included.

c)

Plot the remaining atoms as a function of time. Put also the analytical solution to the decay in the same plot, given by equation (6).

where N_0 is the initial amount of atoms. $\tau = 10.3\,\mathrm{s}$, and is directly derived from p.

Try to increasing the number of atoms, and (hopefully) watch your approximation become more and more like the analytical solution.

Filename: `random_decay.py`

Exercise 8.3 - Optimal shooting angles

You have made a small ball-canon which is supposed to hit a painted area on a wall (if you have done exercise 3.4, then you are free to use and modify your program from that exercise here). The painted area begins at a height h_0 and ends at a height h_1 at the wall.

The height of the ball can be expressed by:

$$y(t) = -\frac{1}{2}gt^2 + v_0 t \sin\theta$$

where $g = 9.81\,\mathrm{m/s^2}$, $v_0\,\mathrm{m/s}$ is the velocity and θ is the angle the ball has been shoot out with.

The ball hits the wall at a time $T = \dfrac{b}{v_0 \cos\theta}$ where b is the distance (in meter) between the cannon and the wall.

[14] The sum function sums up the values of the elements, so a non-decayed atom will count as 1, and a decayed atom will count as 0.

a)

Write a program which generates N values of θ by using Python's `uniform(0,π)` or Numpy's `uniform(0,π,N)`. The values for θ shall be given as a parameter to a function which tests if the canon hits the painted area for every value of θ. The θ-s which makes the ball hit the painted area should then be stored in a list or array. The list or array has to be returned by the function.

Let $N = 1000$, $h_0 = 3\,\text{m}$, $h1 = 3.25\,\text{m}$, $b = 3.5\,\text{m}$ og $v_0 = 25\,\text{m/s}$ and run the program for these parameters.

b)

Use the list og array which was returned by the function in a) to find the mean value θ' of the angles which made the ball hit the painted area. If the list is empty, let the program generate N new values for θ until the returned list or array is not empty. This could for instance be done by making a while loop which will run as long as the list is empty.

Let your program do the same as in a), but instead using a different function to generate the values of θ. Now use Python's `random.gauss(θ',0.05)` or Numpy's `np.random.uniform(θ',0.05,N)`.

Let your program print out how many angles from a) and from this subtask made the ball hit the painted area. How is the number of angles in this subtask compared to the number of angles you found in a)?

Comments to the result in b): The values of θ in a) were generated such that every value between 0 and π were equally probable to pick. However, in b) we generated the values using a *normal distribution* centered about θ'. A normal distribution tells us how likely it is for us to get values in a area about θ'.

The value θ' will be the most likely value to get, whereas the probability of getting other values will decrease the greater the difference between the other values and θ' is. The rate of the decrease is dependent on an additional value, called the standard deviation, which is in our case 0.05 (found after some experimenting).

The larger the standard deviation is, the more probable the other values gets. Smaller the standard deviation makes the other values less probable.

Filename: `optimal_angles_shoot.py`

Exercise 8.4 - Hot gas

In this exercise we will study the velocity of particles in a hot gas. Temperature is just random movement of particles. If the gas isn't moving, the combined velocity of all particles is virtually 0, but they still move internally, relative to each other.

The particles in a gas of absolute temperature T have velocities that are normally (Gaussian) distributed around a mean 0, with a standard deviation[15] $s = \sqrt{\frac{kT}{m}}$, where $k = 1.38 \times 10^{-23}\,\text{m}^2\,\text{kg}\,\text{s}^{-2}\,\text{K}^{-1}$ is Boltzmann's constant. This distribution is true for their velocities along each axis, (v_x, v_y, v_z), individually.

We can get normally distributed numbers with the numpy function `numpy.normal(m,s,shape)`, where `shape` is the shape of the array of random numbers that will be returned. To represent `N` particles with normally distributed velocities along each axis, we will create an array with `shape = (N,3)`. This makes each element in our array represent a particle, which is itself an array of size 3, representing the velocities along each axis.

[15] The standard deviation represents the width of the distribution curve, and a higher standard diviation results in higher velocities, and a higher temperature.

a)

Set $N = 20$, $m = 10^{-22}\,\text{kg}$, $T = 300\,\text{K}$, and save the normally distributed particles from the numpy function in an array.
Use the `random.choice()` function to pick a random of the 20 particles from the array, and print it's velocities to the terminal.

b)

The average kinetic energy of the particles in the gas of temperature T is given as

$$E_k = \frac{3}{2}kT$$

If we multiply this by the number of particles, N, we get the total kinetic energy of the gas.

An alternative method for calculating the kinetic energy is to sum up the individual energies of the particles in the gas using the formula

$$E_k = \frac{1}{2}mv^2$$

Loop over every particle, and calculate their kinetic energy by inserting their absolute velocity, $|v| = \sqrt{v_x^2 + v_y^2 + v_z^2}$, into the second formula. Compare this to the result you get by using the first formula.

Filename: `gaussian_velocities.py`

Exercise 8.5 - Raindrops

In this exercise, we will take a look at the *terminal velocity* for raindrops. The terminal velocity is the velocity of a raindrop where the gravity pulls down the raindrop as much as the air resistance acts on it.

The terminal velocity v_T (in m/s) for a raindrop with radius r and assumed to be shaped as a perfect sphere is:

$$v_T = \sqrt{\frac{8r\rho_v g}{3\rho C}}$$

where $\rho_v = 1\,\text{g}/\text{c}^3\text{m} = 1000\,\text{kg}/\text{m}^3$ is the approximate density of water, $g = 9.81\,\text{m}/\text{s}^2$, $\rho = 1.293\,\text{kg}/\text{m}^3$ is the approximate density of air and $C = 0.47$ describes how much resistance the raindrop exerts on its environment (in this case the air).

a)

In this subtask you are *not* supposed to use the Numpy-module.

Make a program which generates $N = 100000$ different raindrops represented by random values for their radii in the interval $[1\,\text{mm},\ 6\,\text{mm}) = [10^{-3}\,\text{m},\ 6 \cdot 10^{-3}\,\text{m})$.

Your program is supposed to take the time on how long it takes to generate the N raindrops and thereafter find the average terminal velocity.
Your program should then display the average terminal velocity along with how much time it used.

b)

Extend your program from a) such that it does the same calculations, however by using the Numpy-module and vectorization. Write out the terminal velocity along with the runtime of your program which uses vectorization and the Numpy-module.

In this subtask you will also see that it is a huge improvement in time using vectorization and the Numpy-module compared to the time we got from a).

Hint: To take the time in seconds of a program, it is possible to use the time-module. An example of finding how long time your program uses to print 'Hello, world!' is as follows:

```
import time

time_start = time.time()             # To start taking time
print "Hello, world!"
time_used = time.time() - time_start # Find how how time we have used

print "Time used: %g seconds"%time_end
```

Filename: `raindrops.py`

Exercise 9.1 - Acceleration class

The file `constant_acceleration_class.py` contains a class `ConstantAcceleration` for calculating one dimensional movement with constant acceleration, using the equations of movement. The constructor saves the initial position, velocity, and acceleration. The class call returnes the position at a given time `t`, and the method `velocity` returns the velocity at a given time `t`.

The purpose of this exercise is to expand the functionality of the `ConstantAcceleration` class into a `LinearAcceleration` class, which can also deal with cases were the acceleration is a first order polynomial on the form $a = a_0 + jt$. The constant j is known as "jerk", and is the change in acceleration over time. The equations of movement now look like

$$x(t) = x_0 + v_0 t + \frac{1}{2} a_0 t^2 + \frac{1}{6} j t^3$$

$$v(t) = v_0 + a_0 t + \frac{1}{2} j t^2$$

Make a class `LinearAcceleration`, which borrows the functionality of `ConstantAcceleration`, but with the added ability to calculate with a given jerk. Wether you implement a 'has-a' or 'is a' relationship between the two classes are up to you, but you must borrow code from the `ConstantAcceleration` class as much as possible.

Filename: `Jerk.py`

Exercise 9.2 - Solids

In this exercise we will look at how we can made a simple model of an arbitrary solid and then a specific type of solid using inheritance.

a)

Define a class `Solid` which takes the volume of a body as parameter to the constructor. Let the class have a function which calculates the density of the body where the mass of the body is given as parameter to the function. Recall that the density is defined to be mass divided by volume.

b)

Define a class `Iron` which is a subclass of `Solid`. The constructor must take the volume of a given body (made of iron) and mass as parameters.
Define a function in `Iron` which calls the function from `Solid` to calculate and returns the density of the created body.

c)

Define a test function which initialize an instance of the class from b) with volume $0.1\,\mathrm{m}^3$ and mass 787 kg and test if the calculated density is $7870\,\mathrm{kg}/\mathrm{m}^3$. Remember to call your testfunction!

Filename: `Solid.py`

Exercise 9.3 - Moment of inertia about center of mass

The mass of a body can be looked upon as being a measurement of how difficult it is to change its velocity. We have a similar measure for rotation of a body, namely the *moment of inertia* about the center of mass of the body[16].

[16]The center of mass of a body is a position in space dependent on the distribution of mass within the shape of the body.

a)

Define a class which represents a geometric object. The constructor must only take the mass M of the object as parameter.

Initialize an instance of this class with $M = 5\,\text{kg}$.

b)

Write a class which inherits from the class defined in a). This class will represent a cylindrical object. The constructor must take as parameters the mass M of the object and its radius R.

Define a function inside this class which calculates and returns the moment of inertia to the cylindrical object. The moment of inertia I to a cylinder with mass M and radius R is found (by calculation) to be:

$$I = \frac{1}{2}MR^2$$

Initialize an instance of this class with $M = 5\,\text{kg}$ and $R = 0.75\,\text{m}$ and display its moment of inertia.

c)

Make a class which inherits from the class in b). This class shall represents a cylindrical shell with mass M and radius R.

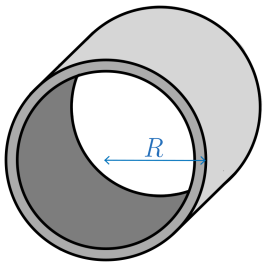

Figure 4: Illustration of a cylindrical shell with radius R.

This class must also be able to calculate and return the value of the moment of inertia of the cylindrical shell. However, in this case we can the moment of inertia calculated in the class from b). You see, it can be shown that the moment of inertia for a cylindrical shell is:

$$I = MR^2$$

We could therefore make b) calculate the moment of inertia, and then multiply its answer by 2.

Initialize an instance of this class representing a cylindrical shell with $M = 5\,\text{kg}$ and $R = 0.75\,\text{m}$ and write out its moment of inertia.

Remark: In this subtask you are supposed to write very little code. Exploit inheritance as much as possible!

Filename: `Moment_of_inertia.py`

Exercise E.1 - Boiling water

Newton's law of cooling tells us how the temperature $T(t)$ to an object changes over t minutes. It is formulated as such:

$$\frac{\mathrm{d}T(t)}{\mathrm{d}t} = -k(T(t) - T_e)$$

where T_e is the temperature to the environment and k is the rate of change in temperature of the water.

We will look closer at how the temperature to water at 15 °C changes when being boiled in an old and worn out kettle. We will use Newton's law of cooling as a simple model to the change of temperature.

At $t = 0$, we have that $T(0) = 15\,^{\circ}\mathrm{C}$. Also, let $k = 0.2$.

a)

The kettle gets a temperature of 100 °C immediately after being turned on. In other words, we have that $T_e = 100\,^{\circ}\mathrm{C}$ for all values of t.

Write a program which use `ODESolver` to calculate the temperature of the water $T(t)$. Use $N = 1000$ uniformly distributed values for t between 0 and 15.

b)

As you probably saw in a) it takes the kettle long time to make the water reach 100 °C. A friend of yours tries to fix your kettle, but unluckily manages to make it worse. You make some measurements to see how badly the kettle has become, and notices that the temperature of the kettle varies according to the following model:

$$T_e(t) = 20 \sin\left(\frac{\pi}{3}t\right) + 80$$

Extend your program from a) such that it finds the temperature $T(t)$ of the water in the broken kettle.

Let your program plot the temperature T of the water from a) along with the temperature T your program finds in this subtask.

Filename: `boiling_water.py`

Exercise E.2 - RC circuit

The object of this exercise is to study the change in electric charge Q held by a capacitor over time.

Figure 5 shows an RC-circuit, containing a capacitor, a resistor and a battery (voltage source). Our focus will be on the capacitor, which consists of two parallel metal plates. When a voltage source is connected, the metal plates gets charged with positive and negative electric charges. When the voltage source is disconnected, the electric charges flows back through the circuit in opposite direction, until the capacitor is entirely discharged, and $Q = 0\,\mathrm{C}$.

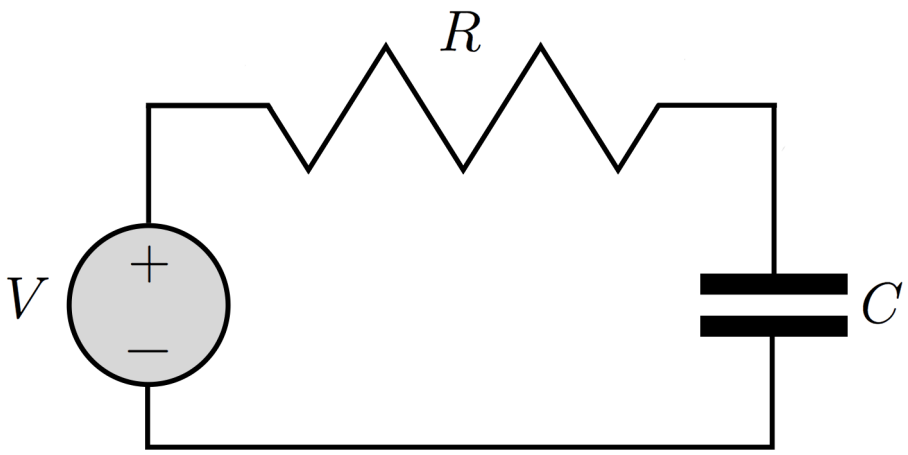

Figure 5: Illustration of a RC-circuit.

The battery provides a constant voltage V (measured in Volt) when turned on. The resistors resistance is given as R (in Ohm), and tells how well it blocks electric current. The capacitor has a capacitance C (in Farad), which tells how well it can hold electric charge.

The amount of electric charge the capacitor holds at a certain time is given by differential equation

$$Q'(t) = \frac{V}{R} - Q(t)\frac{1}{RC} \tag{7}$$

a)

Use ODESolver to solve for $Q(t)$ over the first 10 seconds with 101 timesteps. Set the battery's voltage to $V = 8\,\text{Volt}$, the capacitors capacitance to $C = 2 \times 10^{-8}\,\text{Farad}$, and the resistors resistance to $R = 10^8\,\text{Ohm}$. The capacitor has no charge, $Q_0 = 0$, at $t = 0$. Use both Forward Euler and Runge Kutta 4, and plot both results together with the analytical solution

$$Q(t) = (Q_0 - CV)\mathrm{e}^{-t/RC} + CV \tag{8}$$

Reduce the number of timesteps until you can visibly distinguish both numerical methods from the exact solution.

b)

The battery becomes unstable at $t = 10\,\text{s}$, and the voltage starts oscillating as a sinus wave. At $t = 20\,\text{s}$ the battery breaks completely. We can implement this by making the voltage a variable of time, $V(t)$. We are still solving the differential *equation* (7), but now with a voltage given as

$$V(t) = \begin{cases} 8, & t < 10 \\ 4\sin(t) + 4, & 10 < t < 20 \\ 0, & t > 20 \end{cases} \tag{9}$$

Plot the new solution, $Q(t)$ over 30 seconds with Runge Kutta 4 and 101 timepoints.

Filename: `RC.py`

Exercise E.3 - Throwing ball with air resistance

In this task we will simulate a ball being thrown where we include air resistance. To do so, we will use `ODESolver.py`.

The wind moves in the same direction as the ball is being thrown.

We must solve the following equations to find the position of the ball:

$$\begin{aligned}\frac{\mathrm{d}x}{\mathrm{d}t} &= v_x \\ \frac{\mathrm{d}v_x}{\mathrm{d}t} &= -\frac{1}{2m}\rho C\pi R^2(v_x - w)^2\end{aligned}$$

$$\begin{aligned}\frac{\mathrm{d}y}{\mathrm{d}t} &= v_y \\ \frac{\mathrm{d}v_y}{\mathrm{d}t} &= -g\end{aligned}$$

where $C = 0.47$ is a measurement on how much resistance the ball does at the air, R is the radius of the ball, m is the mass of the ball, v_x is the velocity of the ball along the x-axis, w the velocity of the wind and $g = 9.81\,\mathrm{m/s^2}$.

The initial conditions for this model are:

$$\begin{aligned}x(0) &= 0 \\ v_x(0) &= v_0\cos\theta \\ y(0) &= 0 \\ v_y(0) &= v_0\sin\theta\end{aligned}$$

where θ is the angle and v_0 the velocity (in m /s) the ball was thrown with.

a)

Define a class which contains information about this problem. The class must have the same structure as the class `Problem` at page 790 in the book (or page 745 is you use the 4th edition). The class in this exercise must have a constructor which takes in the initial conditions as a list, the radius R of the ball, the mass m of the ball and the velocity of the wind w.

The class must also contain a `__call__`-function which returns the list:

$$\texttt{[vx,}-\frac{1}{2m}\rho C\pi R^2(v_x - w)^2\texttt{,vy,}-g\texttt{]}$$

where `vx` and `vy` is found in the same matter at page 790 in the book (or 745).

b)

We will now see how the velocity of the wind affects the motion of the ball over $T = 0.5\,\mathrm{s}$.

Assume that someone throw a ball with radius $R = 0.03275\,\mathrm{m}$ and mass $m = 0.057\,\mathrm{kg}$ with an angle $\theta = \frac{\pi}{4}$ and velocity $v_0 = 10\,\mathrm{m/s}$. Also, let $x(0) = 0$ and $y(0) = 0$.

We will look at the velocities of the wind $w = -10, -5, 0, 5,$ og 10. For every velocity w, the program must do the following:

1) Initialize and instance of the class defined a) where the values of r R,m, w and list of initial conditions is sent as parameters to the constructor. The list of initial conditions `U0` might have the following structure:

   `U0 = [x0, vx0, y0, vy0]`

   where `x0` $= x(0)$, `vx0` $= v_0\cos(\theta)$, `y0` $= y(0)$ og `vy0` $= v_0\sin(\theta)$.

2) Let `ODESolver` find the positions. You are free to choose which numerical method `ODESolver` should use to solve the ODEs. Be aware to send enough timepoints to the function `solve` from `ODEsolver` such that you get reasonable results.

3) Send the found positions from step 2) to `plt.plot(x,y)`. Be aware to get the correct values from the matrix `solve` from `ODESolver` returns!

When your program has finished the above steps for every w, let your program call `plt.show()` to display the motion of the ball for every value fo w.

Important: When you have to plot several graphs in one plot, it is very important to be able to distinguish them somehow. To do so, one could use a list of strings which describes what is unique for every graph. The list can so be sent to `plt.legend` *before* `plt.show()` has been called. For instance, in our case we can do the following:

```
# Other necessary calls
labels = []
for w in [-10,-5,0,5,10]:
    # Here we will solve for x and y using ODESolver
    labels.append("w = %g"%w)
    plt.plot(x,y)

plt.legend(labels)
plt.show()
```

to be able to determine which graph corresponds to which velocity w of the wind.

Filename: `throw_air_resistance.py`

Exercise E.4 - Planetary orbits

In this exercise we will study the orbit of the earth around the sun with 2nd order ODEs. Because the earth revolves around the sun in a plane, we can model its movement in only two dimensions, which we'll call x and y.

Newtons gravitational law tells us the gravitational force between two objects. If we decompose this law into the x and y direction, and inserts it into Newtons second law, we can solve for the acceleration of Earth:

$$\frac{d^2x}{dt^2} = -G\frac{M_{sun}x}{x^2+y^2} \qquad \frac{d^2y}{dt^2} = -G\frac{M_{sun}y}{x^2+y^2} \tag{10}$$

Here, M_{sun} is the mass of the sun, G is the gravitational constant, and x and y is the distance from the Sun in x and y directions.

We will in this exercise use astronomical units instead of SI units, meaning that we will

- measure time in years (yr) instead of seconds.
- measure distances in Astronomical Units (AU) instead of meters.[17]
- measure masses in Solar Masses (SM) instead of kilograms.

This has the effect that

- the gravitational constant becomes $G = 4\pi^2 \text{AU}^3\,\text{yr}^{-2}\,\text{SM}^{-1}$.
- the mass of the Sun is $M_{sun} = 1\,\text{SM}$.
- the initial position of Earth can be set to $x = 1\,\text{AU}$, $y = 0\,\text{AU}$.
- the initial velocity of Earth can be set to $v_x = 0\,\text{AU/yr}$, $v_y = 2\pi\,\text{AU/yr}$.

Use the `ODESolver` to solve for the movement of the Earth around the Sun for 10 years using equation (10) and the initial conditions and parameters described above.
For more information on how to set up 2nd order ODEs, see the book, page 799-801.

[17] 1 AU is the average distance between the Sun and Earth

Plot y against x for the 10 years, and you should observe a circular orbit of the earth around the origin, in a distance 1 AU.

Hint: Matplotlib doesn't keep axis aligned by default, and circular orbits may look elliptical. You can change this by setting `matplotlib.pyplot.axis('equal')`

Filename: `Orbits.py`