

Course Reader

Simulating real-life systems

Leo Polak, Bart Bozon, Hildo Bijl

NLE

Contents

[1. Introduction 3](#_Toc56502240)

[2. Agent-based systems 4](#_Toc56502241)

[2.1. Agent-based modeling and simulation 4](#_Toc56502242)

[2.2. Considerations 9](#_Toc56502243)

[2.3. Applications 11](#_Toc56502244)

[3. Systems in physics with one independent variable 12](#_Toc56502245)

[3.1. Modeling and simulation of systems changing in time 12](#_Toc56502246)

[3.2. Adding complexity 16](#_Toc56502247)

[3.3. Orbital mechanics: The earth moon system 19](#_Toc56502248)

[4. Systems in physics with multiple independent variables 26](#_Toc56502249)

[4.1. Modeling and simulation of one-dimensional systems changing in time 26](#_Toc56502250)

[4.2. One-dimensional systems with external effects 33](#_Toc56502251)

[4.3. One-dimensional systems with discontinuities 37](#_Toc56502252)

[4.4. Modelling and simulation of two-dimensional systems changing in time 41](#_Toc56502253)

[4.5. A complex 2D case 46](#_Toc56502254)

[4.6. Taking advantage of rotational symmetry with cylindrical coordinates 54](#_Toc56502255)

[5. Linear system theory 56](#_Toc56502256)

[5.1. The state-space form for linear systems 56](#_Toc56502257)

[5.2. Stability of continuous-time linear systems 60](#_Toc56502258)

[5.3. Stability of discrete-time linear systems 64](#_Toc56502259)

[5.4. The state-space form for nonlinear systems 67](#_Toc56502260)

[5.5. Stability of continuous-time nonlinear systems 70](#_Toc56502261)

[6. Modelling Event-based systems using Simpy 72](#_Toc56502262)

[6.1. Introduction 72](#_Toc56502263)

[6.2. Process interactions 73](#_Toc56502264)

[6.3. An elaborate example 75](#_Toc56502265)

[References 79](#_Toc56502266)

[A. Basics of Python 80](#_Toc56502267)

[A.1. Installing Python 80](#_Toc56502268)

[A.2. Basics of Python 80](#_Toc56502269)

[B. Complex numbers 81](#_Toc56502270)

[B.1. A history of numbers 81](#_Toc56502271)

[B.2. Solving equations with complex numbers 82](#_Toc56502272)

[B.3. Visualizing, adding and subtracting complex numbers 82](#_Toc56502273)

[B.4. Multiplying and dividing complex numbers 83](#_Toc56502274)

[B.5. The magnitude and argument of complex numbers 84](#_Toc56502275)

[B.6. Complex exponents and Euler’s formula 85](#_Toc56502276)

[B.7. Complex numbers in Python 86](#_Toc56502277)

[C. Differential equations 88](#_Toc56502278)

[C.1. Basics of differential equations 88](#_Toc56502279)

[C.2. Solving second-order linear differential equations 92](#_Toc56502280)

[C.3. Solving general linear differential equations 95](#_Toc56502281)

[C.4. Partial differential equations 97](#_Toc56502282)

[D. Linear algebra: vectors and matrices 100](#_Toc56502283)

[D.1. What are vectors? 100](#_Toc56502284)

[D.2. Calculating with vectors 102](#_Toc56502285)

[D.3. What are matrices? 105](#_Toc56502286)

[D.4. Calculating with matrices 108](#_Toc56502287)

[D.5. Other matrix operations 111](#_Toc56502288)

[D.6. Solving systems of linear equations 113](#_Toc56502289)

[D.7. Eigenvalues and eigenvectors 116](#_Toc56502290)

[E. Taylor polynomials 119](#_Toc56502291)

[E.1. Taylor polynomials for scalar functions 119](#_Toc56502292)

[E.2. Finding finite difference formulas for derivatives with Taylor polynomials 124](#_Toc56502293)

[E.3. Taylor polynomials for vector functions 126](#_Toc56502294)

[F. Excercises & answers 128](#_Toc56502295)

# Introduction

When developing an engineering product, it is of vital importance to be able to predict what will happen to it in various scenarios. Just like the engineers of the Apollo project managed to safely land a capsule on the moon, predicting its flight in advance through equations, so should we be able to analyse our products before we even make them.

There is one key difference between us and the Apollo engineers. While they mainly worked with pen and paper, we have the computational power of computers at our disposal. We can turn equations into full-fledged simulations, allowing us to quickly analyse a large variety of situations. That is the main subject of this course.

The process of turning a real-life system into a simulation always has five important steps.

1. **Analysis/Equations**: using so-called first principle methods from physics, economics, biology, chemistry or whatever field we are working with, we set up a variety of equations. These equations provide us a starting point.
2. **Modelling**: using various simplifying assumptions, we turn these equations into a mathematical model of our system.
3. **Simulation**: through computer programming we implement the mathematical model into a simulation. We verify whether the simulation matches the predicted behaviour.
4. **Validation**: we compare our simulation with reality and either fix or explain the differences.

For most of the example projects in this reader, we only walk through the first three steps.

This reader starts off with relatively easy simulations, but gradually the complexity increases. Along the way both new mathematical techniques and simulation tools are introduced. Chapter 2 introduces agent-based systems. Chapter 3 introduces modelling and simulations of physical quantities that change in time. In chapter 4 this is extended to physical quantities that also vary in space. We then dive deeper into the mathematics behind linear systems in Chapter 5. Chapter 6 discusses agent-based systems using Simpy. We encourage readers to work through the reader in chronological fashion, doing the exercises as they come.

# Agent-based systems

Agent-based modelling can be used to model systems in which a number of agents behave according to a set of rules. For instance, it can be used to model behaviour of people, such as traffic flows in a train station. It can be used as well for physics systems. An example is the marble track assignment. For physics systems a related method is Monte Carlo simulations.[[1]](#footnote-2)

## Agent-based modeling and simulation

Let’s take the following example: modelling pedestrians on an intersection. How could such an example be turned into an agent-based model?

Each agent-based model consists of the following items:

1) A method of generating agents. When an agent is generated, certain properties of the agent can be assigned as well (for instance the maximum speed of that specific pedestrian). Ideally these values should be randomized so that the simulation is more realistic.

2) Rules for the agent. For each agent a set of rules is needed which describes the actions the agent must or can take. For our example: Which direction should the pedestrian go? With which speed? Should he stand still? Most of the times not all the real life equivalent rules can be incorporated in our simulation. So approximations have to be made.

3) A method for removing agents. For example, for the boarding of a plane the removing method could be that they sit down on their assigned seat.

4) A method to execute item 1, 2 and 3. As we will simulate using computers, the actions will be executed each time step. So time is discreet. We also need a lot of administration. What is the state of each agent at each time step?

5) A method to extract data and information from the simulation. For instance, average waiting time for an intersection. Or the amount of people run over because of too slow crossing of the intersection. By running the simulation multiple times, statistically sound outcomes can be determined.

Determining the content of these items generally requires some research to properly substantiate the choices made. However, sometimes simple observations and general knowledge are sufficient to quickly build a simple simulation that can provide valuable insights. The simulation can then still be improved at a later instance by applying insights gained from research, as well as from the simple simulation.

### A simple example, a crowed hallway

We will start with a simple simulation. From the left people are entering a hallway, they leave to the right. Halfway through the hallway there is a constriction present. The simulation uses a “grid” technique. This is not the only possibility, but a relatively simple one.

Let’s fill in het items mentioned in the previous paragraph:

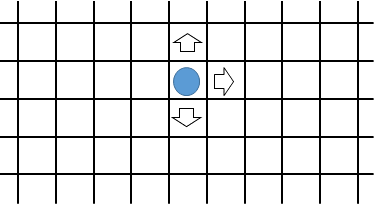
1) Generating agents

We will assume that the agent enters on the left (-position is 0). Each agent is created equally, no specific attributes are contributed to the agents. The generation rate is pre-set by the user, 1 equals 1 agent generated per time step, 2 equals (on average) 1 agent generated per 2 time steps. The height (-position) is randomly generated. If the position is occupied by another agent, a new position is tried for a total of 10 times. If after 10 times no free position was assigned, this agent is not generated. This will result in an actual generation rate (and hallway throughput rate) that is lower than the pre-set generation rate. This will be a sign of the hallway clogging up.

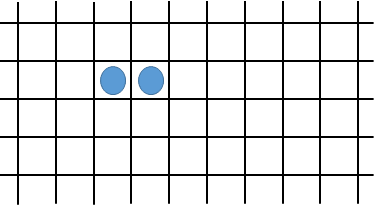
2) Rules for the agents.

The rules for the agent are:

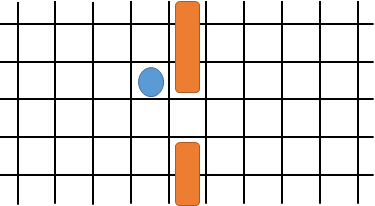
1. Each agent moves on a rectangular grid. The agent can only move vertically or horizontally. (This is a simplification of reality to make the simulation less complex)



1. The agents have a 16.6% probability to do nothing.
2. The agents have a 33.3% probability to go up or down (each 16.6%).
3. The agents have a 50% probability to go right.
4. If an agent wants to move to an occupied area the move is not executed. So the left blue agent below cannot move to the right.



1. The constriction is realized as an opening in a barrier. If an agent wants to move into the barrier, the move is not executed. So the blue agent below cannot move to the right.



1. The agent cannot move outside the grid. If an agent wants to move outside the grid, the move is not executed. In practice this means that they do not occupy the upper and lower borders of the grid.

3) Removing agents

If the agent reaches the right side of the grid they are removed.

4) Execution and administration

Now that the rules are clear we need tools for our administration. We will use numpy arrays for this. It is possible to use python structures like list or dicts for this but we advise strongly against using structures other than numpy arrays. Python threats data structures in a rather peculiar way (if a data structure is copied the reference to the structure is made the same, the data is not actually copied!), in the past we have seen numerous problems because of this[[2]](#footnote-3). We need the following:

1. A large 2-dimensional array in which we can keep track if a square is occupied. A second 2-dimensional array in which the position of the barrier is indicated.
2. Multiple arrays in which we can keep track of the agents:

* an array with the status of the agent. Possible state: not yet started, active, and finished.
* an array with the x-coordinate.
* an array with the y-coordinate.

5) Data extraction

The data we want to extract is the throughput of the hallway vs the pre-set generation rate.

Run the following code:

# -\*- coding: utf-8 -\*-

"""

Created on Mon Oct 12 16:37:38 2020

@author: bart.bozon

"""

**import** numpy as np

**import** random

max\_x=20

max\_y=10

max\_agents=1000

# 2-d matrices

field = np.zeros((max\_y,max\_x), np.uint8)

barrier = np.zeros((max\_y,max\_x), np.uint8)

# administration per agent

x\_pos = np.zeros((max\_agents+1), np.uint8)

y\_pos = np.zeros((max\_agents+1), np.uint8)

status = np.zeros((max\_agents+1), np.uint16)

# Status

# 0 not yet started

# 1 active

# 2 finished

barrier [0,5]=1

barrier [1,5]=1

barrier [2,5]=1

barrier [3,5]=1

barrier [4,5]=1

barrier [max\_y-4,5]=1

barrier [max\_y-3,5]=1

barrier [max\_y-2,5]=1

barrier [max\_y-1,5]=1

**print**("the barrier:")

**print** (barrier)

**print** ()

# do you want a pause? uncomment the next line

#dummy=input("press return in print window to continue")

**def** position\_allowed(y,x):

**if** y<1 **or** x<1:

**return** False        # Too close to the edge : rule g

**elif** y>max\_y-1 **or** x>max\_x-1:

**return** False    # Too close to the edge : rule g

**elif** barrier[y,x]>0:

**return** False      # There is a barrier : rule f

**elif** field[y,x]>0:

**return** False     # Position is not free : rule e

**else**:

**return** True

generation\_rate=5

# start with an empty slate

amount\_agents=0

time =0

field.fill(0)

x\_pos.fill(0)

y\_pos.fill(0)

status.fill(0)

**while** time<1000:

    #execute rules

**if** amount\_agents>0:

**for** i **in** range (1,max\_agents+1):

**if** status[i]==1:    # is the agent active?

                dice=random.randint(0,5)  # throw a dice to determine walking distance

**if** dice ==0 :   #rule b

**pass**

                    #do nothing

**if** dice ==1 :   # rule c

                    #down

**if** position\_allowed(y\_pos[i]+1,x\_pos[i]):

                        field[y\_pos[i],x\_pos[i]]=0

                        y\_pos[i]=y\_pos[i]+1

                        field[y\_pos[i],x\_pos[i]]=1

**if** dice ==2 :   # rule c

                    #up

**if** position\_allowed(y\_pos[i]-1,x\_pos[i]):

                       field[y\_pos[i],x\_pos[i]]=0

                       y\_pos[i]=y\_pos[i]-1

                       field[y\_pos[i],x\_pos[i]]=1

**if** dice ==3 **or** dice ==4 **or** dice ==5:    # rule d

                    #right

**if** position\_allowed(y\_pos[i],x\_pos[i]+1):

                       field[y\_pos[i],x\_pos[i]]=0

                       x\_pos[i]=x\_pos[i]+1

                       field[y\_pos[i],x\_pos[i]]=1

**if** x\_pos[i] == max\_x-1:     # removing agents

                    field[y\_pos[i],x\_pos[i]]=0

                    status[i]=2

    # determine if new agent is generated

**if** random.randint(1,generation\_rate)==1:

        #Check free position:

        succeeded=False

        teller=0

**while** (**not**(succeeded)**and** teller<10):

            pos\_x=1

            pos\_y=random.randint(2,max\_y-2)

**if** field[pos\_y,pos\_x]==0:

                succeeded=True

            teller=teller+1

**if** succeeded:

            # generate agent

            amount\_agents=amount\_agents+1

            x\_pos[amount\_agents]=pos\_x

            y\_pos[amount\_agents]=pos\_y

            field[y\_pos[amount\_agents],x\_pos[amount\_agents]]=1

            status[amount\_agents]=1

    time=time+1

**print**(field)

**print**()

    # do you want a pause? uncomment the next line

    # dummy=input("press return in print window for next value")

# data extraction

num\_finished\_agents=0

**for** i **in** range (1,max\_agents+1):  # how many agents made it till the end?

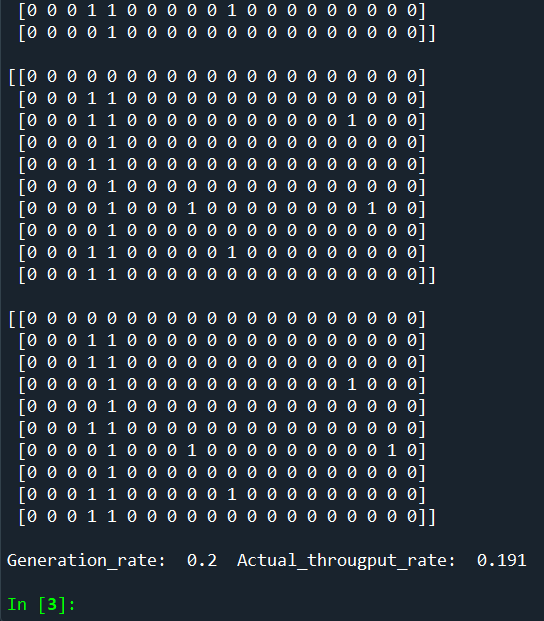
**if** status[i]==2:

        num\_finished\_agents=num\_finished\_agents+1

**print** ("Generation\_rate: ",1/generation\_rate, " Actual\_througput\_rate: ",num\_finished\_agents/time)

Script 2.1: Code for a crowed hallway with a constriction.

If you run the code you will get the output in a very simple form (later we will look at more sophisticated output methods). The matrix with the occupied places is printed out a large number of times. At the end the pre-set generation rate vs the actual throughput is given.



As you can see, the pre-set generation rate, which can be viewed as the target for generation or hallway throughput is higher than the actual throughput. If we run the simulation a number of times we can produce the graph shown in Figure 2.1.

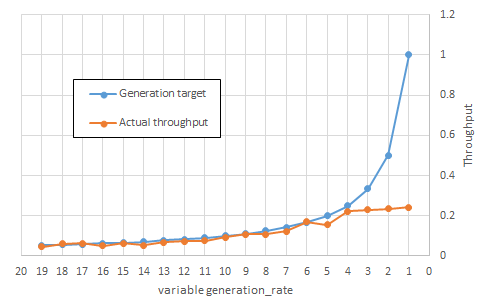


Figure 2.1: Result from the hallway simulation.

Starting at a throughput of 0.2 we see a discrepancy between the targeted generation rate and the actual. When you observe the simulation closely you will see the emergence of traffic jams. The theoretically throughput of the hole in the barrier should be in the order of 0.5 (as half the moves are directed to the right). Put even before that we see the emergence of behaviour which does not automatically follow from the simple rules of the system.

Please note that this simulation behaves similar to a physics simulation of particles travelling through a constricted pipe.

**Exercise 2.1.** Remove the # below the print(field) line. Now you can observe each time step.

**Exercise 2.2.** Play around with the value of the generation\_rate. Can you reproduce (parts of) the graph above?

**Exercise 2.3.** Remove the print(field) line. Notice the huge increase in speed. Python is very slow with printing.

**Exercise 2.4.** Expand the code such that the agent can also move diagonally.

**Exercise 2.5.** Optional: It is kind of lame to have to input different values for generation\_rate. Change the code such that a number of generation\_rates are evaluated.

**Exercise 2.6.** Optional: Use plt.plot to produce a more aesthetically pleasing output.

## Considerations

When modelling agent-based systems a lot of choices have to be made. A number of choices will be detailed below.

### Approximations

The hardest choice in designing an agent-based model is determining which approximations to make. Reality is too complex to incorporate completely in a model. Because of this it is essential to validate your model with measurements of the real life system. Deviations between the outcomes of the model and reality are always the fault of the model (if your measurements of reality are done correctly). The goal is to make incremental improvements to the model to come ever closer to reality.

### Grid or no grid

In the example of the hallway a grid was used. This is not necessary. Without a grid positions of agents can also be stored in a multiple 1-dimensional arrays. The advantage of using a grid is that the rules can be simpler and often the simulation is quicker. The advantage of not using a grid is that the conditions can be made more lifelike (generally objects do not move in discrete steps, but much more fluently). An example of this is the simulation in 3.3.4 in which the orbits of the moon and the earth are simulated. If you do choose a grid, please consider that it does not have to be rectangular. It could be hexagonal or differently spaced/dimensioned. It could also be that an agent can have lots of different statuses, some of them on a grid, some of them in other containing objects. If you would like to model for instance an amusement park, it could be a wise choice to mix a grid (the space between the rides) with a que (waiting que) and containers (e.g. the rides).

### Concurrency

In the example of the hallway the actions of the agents were calculated and immediately implemented for each agent. In reality this could introduce a bias in the outcome of the simulation. It is often preferred to first calculate the actions for all the agents and then implement the action. For this, normally a shadow bookkeeping is kept. The new positions and actions are implemented on the shadow accountancy. After calculating all the actions the shadow accountancy is swapped with the primary accountancy. Due to the nature of Python we strongly advise to use deep copy for this.

### Choice of your agents

Let’s take an example of a simulation of a factory. Agent-based modelling is very well suited for this. But what is your choice of the agent? It could be the products. But it could also be the machines. Both choices are possible. If you choose the machines the products become anonymous. No information on a specific product can be obtained. On the other hand it could very well be that the simulation is simpler than in the case of the products being the agents. You must choose, but choose wisely (Indiana Jones, 1989).

### Visualisations

Especially with agent-based simulations, videos are a strong method to “sell” your simulation. A way to make a video is to save your graphs (e.g. png’s) to the hard drive and compile a video of the set of png’s (using a tool such as makeavi for instance).

Below an example piece of code is given, which generates a set of png’s. (You probably have to install mathplotlib library.)

# -\*- coding: utf-8 -\*-

"""

Created on Tue Oct 13 19:47:12 2020

@author: bart.bozon

"""

**import** numpy as np

**import** random

**import** matplotlib.pyplot as plt

x = np.zeros((10),np.uint8)

y = np.zeros((10),np.uint8)

**for** i **in** range (1,15):

**for** j **in** range (1,10):

        x[j]=random.randint(0,10)

        y[j]=random.randint(0,10)

    plt.scatter(x,y,c='b')

    plt.title('example @ Bart Bozon')

    plt.ylim (0,10)

    plt.xlim (0,10)

    plt.xlabel('nice axis')

    plt.ylabel('really nice')

    fname = 'name-%03d.png' % i

    plt.savefig(fname)

    plt.show()

**Exercise 2.7.** Use this code to adapt the code of the crowded hallway to make a movie of the hallway.

### Methods

There are a large number of methods available to implement certain behaviour in agents. An example of such a method is Dijkstra's algorithm. With this algorithm the shortest path can be determined between two points.

**Exercise 2.8.** Study the algorithm in literature. How would you implement it in code?

Several other algorithm’s for other issues are available in literature.

## Applications

The applications of agent-based modelling are multitude. It can be used to model human behaviour, animal behaviour, factories, physical objects and many, many more. Below a few examples are described.

### Factory

The production flow of a factory is well suited for an agent-based model. Possible goals of such a simulation are:

* Optimal factory layout to minimise routing distances and times.
* Throughput vs production equipment determination.
* Insight in congestions, traffic jams, etc.

### Pandemic

Human behaviour can be simulated to determine the properties of a contagious disease. Goals could be:

* Effect of preventive measures (e.g. social distancing, quarantining).
* Determining spreading velocities, spreading distance.
* Changing characteristics of the disease (e.g. lethality or contagiousness) and determining effect.

### Celastial bodies

Agent-base modelling can also be used to simulate the movements of planets and moons. In chapter 3.3 this simulation is examined.

### Traffic flows

Agent-based modelling could also be used to simulate traffic flows. Goals could be:

* Optimal road layout.
* Capacity planning.

### Gas or fluid simulations

Gases and fluids consist ultimately of separate particles (1st order approximation, they actually consists of fluctuations of quantum fields). So gases and fluids can also be modelled as agents. Most of the times this is not the first choice (the models of chapter 4 are more suited for this).

# Systems in physics with one independent variable

In this chapter we will focus on modelling and simulation of systems in physics with one independent variable. Generally this will be time. Most physical systems can be described by a differential equation and in the case of one independent variable this is called an Ordinary Differential Equation. For a number of examples we will look at the equations, turn them into a model, analyse the model and set up a Python simulation.

## Modeling and simulation of systems changing in time

To model a system in physics we first have to study the laws which govern the systems. We will see that most systems are described by differential equations. Differential equations describe a system by formulating how it changes with time.

### Physics of a skydiver

As an introduction to a physics system, we will go through the modelling of the motion of a skydiver. It is a system consisting of an object with mass falling towards the surface of the earth. The force that causes the downward motion is the gravitational force. According to Newton’s law of gravity, this force is proportional to the mass and varies with the distance between the mass and earths centre as

|  |  |
| --- | --- |
|  | (3.1) |

where N/(kg-2m-2) is the gravitational constant and kg is the mass of the earth. The minus sign comes from the fact that the force is in principle a vector and here we have chosen the direction towards the centre of the earth as negative. More on vectors can be found in Appendix D, sections D.1 and D.2.

For a motion over a short distance relative to the earth’s radius, the distance is almost constant. For a falling trajectory near the earth’s surface, it is therefore a good approximation to replace by the earths radius m. The expression for the gravitational force then simplifies to

|  |  |
| --- | --- |
|  | (3.2) |

where .

**Exercise 3.9.** Calculate the percentage change of the gravitational force going from a height of 3 km to the surface of the earth according to equation (3.1).

In case of the skydiver, there will also be a force due to friction with air. To get a quick model up and running, we will neglect this for now. A motion of a body where the only force acting on it is the gravitational force is called *free fall*.

To describe the motion we need a relation between the forces and the resulting motion. Within classical mechanics, which is appropriate for this system, this can be expressed by Newton’s second law

|  |  |
| --- | --- |
|  | (3.3) |

which states that the acceleration of an object multiplied by its mass is equal to the resulting force on the object. Since in this simplified case there is only the constant gravitational force, we have , and we obtain as our model

|  |  |
| --- | --- |
|  | (3.4) |

This expression can be understood and analysed as a *differential equation*, as is done in the next section.

### The ordinairy differential equation

We will now look at (3.4) as a differential equation. More details on differential equations and derivatives can be found in appendix C.

Acceleration is the change of velocity in time. This is mathematically expressed as

|  |  |
| --- | --- |
|  | (3.5) |

Here we have added behind the acceleration to indicate that it is a function of time. It is also true that is a function of time, but on the right hand side the can be omitted for ease of writing, because this dependence is already clear from the way the derivative is written. Another way to write the right hand side of (3.5) is .

With (3.5) equation (3.4) becomes

|  |  |
| --- | --- |
|  | (3.6) |

This is a first order differential equation of the simplest form. (It is first order because it doesn’t contain any higher order derivatives than the first derivative.) It states that the first derivative of the velocity with respect to time is equal to the constant . Hence, must be a linear function with slope . To find the function for a specific situation, we would also need to know the velocity at some moment. For example we could have . Then, .

Of course, also the position of the skydiver is of interest. The relation is that velocity is the change of position in time. This is mathematically expressed as,

|  |  |
| --- | --- |
|  | (3.7) |

With this equation (3.6) becomes a second order differential equation:

|  |  |
| --- | --- |
|  | (3.8) |

The general analytic solution to this equation is a quadratic function of the form . The two constants and are fixed by specifying at two moments, referred to as boundary conditions (although they need not be at a boundary), or by specifying (*t*) and at some moment, referred to as initial values (although they need not be initial).

### Discretizing the system

To analyse the system, we first try to find the equilibrium state. This is the state (the values of and ) such that, if the skydiver has this position and velocity , he will keep that position and velocity.

To find the equilibrium state for a continuous-time system, we must set the derivatives and equal to zero and find the values of and for which the resulting equations hold. If we do this for the differential equation (3.6), then we find

|  |  |
| --- | --- |
|  | (3.9) |

This equation obviously never holds. It will be false, regardless of the values of and . This means that our system does not have an equilibrium state. It will always continue to change. Intuitively this makes sense: in the absence of friction, our skydiver will always continue to pick up speed.

We could also try to find an analytical solution for our differential equations. For most practical systems this is very hard, if not impossible. For our simple system it is possible to find such a solution.

**Exercise** **3.10.** Verify that and satisfy the differential equations (3.6) and (3.8) for any time and for any initial position and velocity . Also verify that the initial conditions are met: that we have and whenever .

Using this analytical solution, we can visualize how the velocity develops over time. However, we can also find a difference equation and use that to set up a simulation. To do so, we must first *discretize* our model of the system.

To discretize the model, we must pick a time step . We then make a fundamental assumption:

*We assume that, during the full time step , all our parameters are constant.*

This assumption is of course not true, but as long as is sufficiently small, it approximately holds. Thanks to this assumptions, we may rewrite (3.6) as

|  |  |
| --- | --- |
|  | (3.10) |

Also, because , we can write this as

|  |  |
| --- | --- |
|  | (3.11) |

Alternatively, we may also write this as

|  |  |
| --- | --- |
|  | (3.12) |

where is the velocity at the time step, with time . With this result, we have turned our differential equation into a finite difference equation, which we can solve numerically with the aid of a computer. For more mathematical details behind finite difference approximations, you can read Appendix E up to Section E.2.4.

Equation (3.7) can be turned into a difference equation in an identical way, resulting in

|  |  |
| --- | --- |
|  | (3.13) |

which can be written as

|  |  |
| --- | --- |
|  | (3.14) |

However, we can calculate first and use this value in the calculation of as

|  |  |
| --- | --- |
|  | (3.15) |

It turns out that in some cases this approach leads to much better results, because of better energy conservation. There are also more advanced schemes, which might be required to get realistic results when simulating complex systems. In this rather simple system it actually doesn’t matter much, both (3.14) and (3.15) turn out to give good results, as long as the time step is set with care.

**Exercise 3.11.** Use the finite difference formula for the second derivative given in appendix E, section E.2.2, to turn (3.8) into a finite difference equation.

### Influence of the time step

Once more we can visualize the way the velocity and distance evolve. This time we will apply both methods – the analytical solution and the simulated one – and check the differences. The code is listed in Script 3.1 and the resulting plot follows in Figure 3.1.

"""

@author: Hildo Bijl

Fall simulation without friction

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

# Define settings.

endTime = 10 # The time (seconds) that we simulate.

dt = 1 # The time step (seconds) that we use in the discretization.

v0 = 0 # The initial velocity [m/s].

s0 = 0 # The initial position [m].

g = 9.81 # The gravitational acceleration [m/s^2].

# Set up variables.

time = np.arange(0, endTime + dt, dt) # A list with all times we want to plot at.

vAnalytical = np.zeros(len(time)) # A list for the velocity (analytical).

sAnalytical = np.zeros(len(time)) # A list for the distance (analytical).

vSimulated = np.zeros(len(time)) # A list for the velocity (simulated).

sSimulated = np.zeros(len(time)) # A list for the distance (simulated).

vSimulated[0] = v0

sSimulated[0] = s0

# Run analytical solution.

**for** i **in** range(0, len(time)):

    t = time[i]

    vAnalytical[i] = -g\*t + v0

    sAnalytical[i] = -0.5\*g\*t\*\*2 + v0\*t + s0

# Run simulation.

**for** i **in** range(1, len(time)):

    vSimulated[i] = vSimulated[i-1] - g\*dt

    sSimulated[i] = sSimulated[i-1] + vSimulated[i-1]\*dt

# Display results.

plt.plot(time, sAnalytical, marker=".")

plt.plot(time, sSimulated, marker=".")

plt.xlabel('Time [s]')

plt.ylabel('Distance [m]')

plt.show()

# Analyze outcome.

**print**("Distance traveled after", endTime, "seconds")

**print**("Analytical:", sAnalytical[-1])

**print**("Simulated:", sSimulated[-1])

Script 3.1: Both the analytical solution and simulation of the skydiver in one script. This allows us to compare their outcomes.

|  |  |
| --- | --- |
|  |  |

Figure 3.1: The analytical solution (blue) and the simulated outcome (orange) for time steps (left) and (right).

In this case the two solutions are different. If the time step is larger they differ more (roughly 10% in this case for , while if the time step is smaller they are more similar (roughly 1% difference with ). It is important to remember that the analytical solution is as accurate as can be. It’s the simulated outcome that is off. The larger the time step is, the more it will be off.

**Exercise 3.12.** Implement Script 3.1 and experiment with the time step . How does the difference between the analytical and the simulated outcome vary based on the time step? Is there a time step sufficiently small that this difference disappears? Which time step would you recommend?

The lesson here is that, whenever we discretize a system, we introduce errors. To keep these errors small, we must use a sufficiently small time step. If we do not, then our simulation results will be meaningless. Properly choosing the time step is hence crucial. In the next example we will examine more how this works in practice.

Does our current model agree with physical expectations? As one can see from the exact solution given in exercise 3.10 and the simulations in Figure 3.1, the velocity of the skydiver keeps on increasing indefinitely. This is not what happens in reality, because at a certain velocity, the friction with the air will equal gravity and the skydiver will have reached so-called *terminal velocity*. In section 3.1.1 we left this out intentionally to get started quickly, but now it is clear we must add this to properly model the skydiver.

## Adding complexity

When making a simulation, you always want to start off as simple as possible. Make lots of simplifying assumptions, approximate things if that makes it easier to work with, and neglect effects wherever possible. You want to have a very basic simulation up and running as soon as possible. If it then turns out that the results don’t match your expectations, you can always still add complexity.

This is exactly what we will be doing now. Our free fall simulation was really accurate for the first few seconds, but it was not so realistic later on. That is because we did not take into account friction. Let us do so now and note the differences.

### Physics of friction

In an actual fall in the earth’s atmosphere there would be air friction, also called drag. For relatively high velocities the so-called drag equation is a good approximation.[[3]](#footnote-4) It relates the drag force to the velocity as

|  |  |
| --- | --- |
|  | (3.16) |

where depends on the air density, the cross-sectional area of the falling object and the so-called drag coefficient, which depends on the shape of the object and the air flow pattern. If is negative the plus sign should be taken, otherwise the minus sign, since the drag force is always opposite to the velocity. (If you want, you can be write this more neatly as , where is the absolute value of . To keep things simple, we will stick to (3.16).)

Now the resultant force on the falling object is , where we have taken the plus sign because we know the velocity is downward, which we have chosen as negative. With this, Newton’s second law gives

|  |  |
| --- | --- |
|  | (3.17) |

### Rederiving the ordinairy differential equation

Plugging (3.5) into (3.17), dividing by and rearranging, the first order differential equation for the skydiver becomes

|  |  |
| --- | --- |
|  | (3.18) |

To describe a specific situation, also some initial value must be supplied.

### (Lack of) An analytical solution

As usual, we start our analysis by finding the equilibrium state.

**Exercise 3.13.** Find the value of such that in the differential equation (3.18). Explain what the resulting expression means, from a physics point of view.

**Exercise** **3.14.** Use the values and . Calculate the terminal velocity of the skydiver. Verify that this is a sensible outcome.

For this example it is very difficult to derive an analytical solution. It is possible though, and the result equals (philosophicalmath, 2017)

|  |  |
| --- | --- |
|  | (3.19) |

where the hyperbolic tangent and the inverse hyperbolic tangent are defined as

|  |  |
| --- | --- |
|  | (3.20) |
|  | (3.21) |

**Exercise 3.15.** Verify that, if , the velocity then equals its equilibrium value.

Usually, if solving the differential equation is so difficult, we do not bother finding an analytical solution. So from now on, let’s assume that we do not know the solution (3.19). The only thing we can do then is discretize the system and set up a simulation.

**Exercise 3.16.** Discretize the differential equation (3.18): turn it into a difference equation.

### Tuning the time step

As usual, we can turn the difference equation into a Python simulation. The resulting code is shown below.

*"""*

*@author: Hildo Bijl*

*Fall simulation with friction*

*"""*

***import****numpy as np*

**import** matplotlib.pyplot as plt

# Define settings.

endTime = 60 # The time (seconds) that we simulate.

dt = 1 # The time step (seconds) that we use in the discretization.

v0 = 0 # The initial velocity [m/s].

s0 = 0 # The initial position [m].

g = 9.81 # The gravitational acceleration [m/s^2].

b = 0.25 # The friction coefficient [N/(m/s)^2].

m = 80 # The mass of the skydiver [kg].

# Set up variables.

time = np.arange(0, endTime + dt, dt) # A list with all times we want to plot at.

v = np.zeros(len(time)) # A list for the velocity.

s = np.zeros(len(time)) # A list for the distance.

v[0] = v0

s[0] = s0

# Run simulation.

**for** i **in** range(1, len(time)):

    v[i] = v[i-1] + (b/m\*v[i-1]\*\*2 - g)\*dt

    s[i] = s[i-1] + v[i]\*dt

# Display results.

plt.plot(time, s)

plt.xlabel('Time [s]')

plt.ylabel('Distance [m]')

plt.show()

**print**("Distance traveled after", endTime, "seconds:", round(s[-1],3), "meters")

**print**("Final velocity:", round(v[-1],3), "m/s")

Script 3.2: Code for the skydiver with friction added to the simulation. The analytical solution has not been implemented.

|  |  |
| --- | --- |
|  |  |

Figure 3.2: The distance travelled (left) and the velocity (right) of the skydiver, when friction is added. Note the terminal velocity.

The first thing we should do is verify our simulation. Do the results make sense?

**Exercise 3.17.** Compare the terminal velocity from Figure 3.2 with the one you calculated at Exercise 3.14.

Next, we should investigate the influence of the time factor.

**Exercise 3.18.** Does the terminal velocity of the simulation depend on the time step ? Explain why/why not.

**Exercise 3.19.** Implement the code from the previous script. Start with a very small time step of . Run the simulation and note the final distance traveled. You can see this as the “fully accurate” simulation, akin to the analytical solution from before. Next, start increasing the time step and keep track of how the distance travelled changes. What time step do you still consider acceptable? Which time step would you recommend for the simulation?

## Orbital mechanics: The earth moon system

In this chapter we will simulate the movement of celestial bodies.

### Physics of the earth moon system

As the earth and moon move through the ultrahigh vacuum of space, they experience no friction. There will only be gravitational forces acting on these two bodies. In principle, there are gravitational forces coming from all other celestial bodies, such as the Sun and Mars, but we are only interested in the motion of the earth and moon relative to each other. The influence of these external forces on this relative motion is negligible, because they cause very similar accelerations for both bodies. Hence, we will neglect them here.

Thus, we only have the gravitational force between the earth and the moon, which in this case is

|  |  |
| --- | --- |
|  | (3.22) |

Now, is the distance between the center of the moon and the center of the earth and is the mass of the moon.

For both bodies Newton’s second law can be used, resulting in two coupled differential equations. To simplify this complex situation we make the obvious assumption that the earth is standing still. Another complication is that the motion will happen in a plane, so we should take care of the fact that in this plane the force, acceleration, velocity and position of the moon are described by two-dimensional vectors. For more information on vectors, see Appendix D.

### The equations of motion

Taking a perpendicular -coordinate system with the origin at the centre of the earth, we can treat the force, acceleration, velocity and position vectors of the moon with in terms of their -, and a - component. Specifically, we can write Newton’s second law separately for both components.

|  |  |
| --- | --- |
|  | (3.23) |

where and are the components of the acceleration vector and and are the components of the gravitational force. From Figure 3.3 we can see that the components of can be related to the magnitude and the position coordinates as

|  |  |
| --- | --- |
|  | (3.24) |

The magnitude of is still given by (3.22).

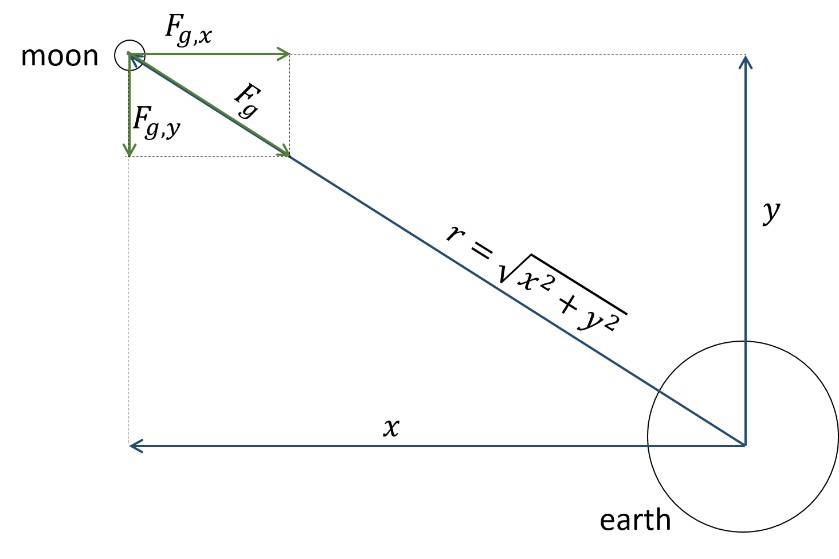


Figure 3.3: Schematic drawing of the earth-moon system, depicting the components  
and magnitudes of the position and force vectors in the plane of motion.

Combining (3.22), (3.24) and (3.25), writing the acceleration as the second derivative of position with respect to time and dividing by , the coupled differential equations that model the earth-moon system become

|  |  |
| --- | --- |
|  | (3.25) |

It is possible to obtain analytic solutions for these equations (see for example (McComb, 1999)), but we will restrain ourselves to computational analysis and visualization of the results.

### Discretizing the system

For this system, we could determine the equilibrium state. Not surprisingly, the only way in which a moon can stand still is if it already inside the earth: . Naturally this case never occurs in real life, so we will ignore it. However, this does establish that a moon has to keep moving to remain in the sky.

Next, we could also try to determine analytical solutions for the trajectory of the moons. When we have a single planet and a moon – a so-called two-body problem – then this can still be done. The mathematics will be too complicated to discuss here, but the shapes will be so-called conical cross-sections: circles, ellipsoids and hyperboles. However, when there are more than two objects, we are dealing with a multi-body problem. No general analytical solution is known for this problem, and hence we will not search for it here.

Instead, we will simply discretize the system equations. This is done identically to what we did in Section 3.1.3, except we now do it twice: once for the x-direction and once for the y-direction. As a result, we have the change in velocity

|  |  |
| --- | --- |
|  | (3.26) |

Similarly, we can calculate the change in position

|  |  |
| --- | --- |
|  | (3.27) |

When there are multiple bodies involved, we first need to calculate the sum of all forces acting on each planet, in each direction. Subsequently we need to divide by the respective body’s mass to find its acceleration for each direction, and then numerically integrate it twice, as shown above, to update the body’s position. This then needs to be done for every body at every time step to complete the simulation.

### Visualizing the system behavior

The simulation is very similar to the simulations from the previous chapters. The main complicating factor is the addition of a second axis.

"""

@author: Bart Bozon

eart-moon system

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

**import** math

# Define settings.

arraysize =4000   # number of simulation points

Rstart=384400000  # starting distance of the moon [m]

Vstart=1022       # starting speed [m/s]

G =6.674E-11      # gravity constant

Mmoon=7.342E22    # mass moon [kg]

Mearth=5.972E24   # mass earth [kg]

dt =1000          # timestep [s]

# Set up variables.

x =np.zeros(arraysize)  # array with x positions

y =np.zeros(arraysize)  # array with y positions

vx =np.zeros(arraysize) # array with speed in x direction

vy =np.zeros(arraysize) # array with speed in y direction

x[0]=Rstart             # starting position

vy[0]=Vstart            # starting speed

# Run simulation.

**for** t1 **in** range (0,arraysize-1):

    rsquare=x[t1]\*x[t1]+y[t1]\*y[t1]   # the distance (earth-moon)^2

    Ftotal=-G\*(Mmoon\*Mearth)/(rsquare) # total force

    vx[t1+1] = vx[t1]+dt\*Ftotal\*x[t1]/(Mmoon\*math.sqrt(rsquare))

    vy[t1+1] = vy[t1]+dt\*Ftotal\*y[t1]/(Mmoon\*math.sqrt(rsquare))

    x[t1+1] = x[t1]+vx[t1+1]\*dt

    y[t1+1] = y[t1]+vy[t1+1]\*dt

# Display results.

x\_axis = np.arange(0, arraysize\*(dt/(24\*3600)), dt/(24\*3600))

plt.plot(x\_axis,x, x\_axis,y)

plt.xlabel('t[d]')

plt.ylabel('[m]')

plt.show()

Script 3.3: Code for the simulation of the earth – moon system.

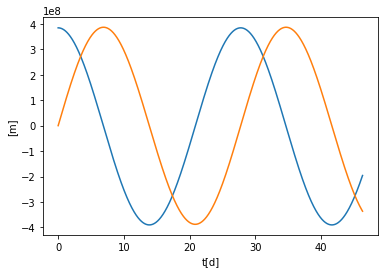


Figure 3.4: The x [blue] and y [orange] positions of the moon relative to the earth.

### [Optional] Graphical simulation of the system

In this step we will make a “real time” simulation in which we will graphically depict the moving planets. First we will need to install the TkInter library for python. TkInter is a GuiProgramming toolkit for Python. See the installation manual on canvas on how to install TkInter. In the script below planets can be created by an init function. After invoking the root.mainloop() the visualization is displayed.

"""

@author: Bart Bozon

TkInter script to simulate a solar system

"""

**try**:

    # for Python2

**from** Tkinter **import** Tk,Canvas   ## notice capitalized T in Tkinter

**except** ImportError:

    # for Python3

**from** tkinter **import** Tk,Canvas

**import** math

# defining constants

g\_constant =1   # the gravity constant is not correct.

                # the simulation takes place in a imaginary universe

delta\_t=0.1

# defining a class with planets and methods

**class** Planets:

**def** \_\_init\_\_(self, canvas):

        self.planets=0

        self.mass=[]

        self.x= []

        self.y=[]

        self.displaysize=[]

        self.vx=[]

        self.vy=[]

        self.ax=[]

        self.ay=[]

        self.planet=[]

**def** init\_planet(self,canvas,mass,displaysize,x,y,vx,vy):

        self.canvas = canvas

        self.mass.append(mass)

        self.x.append(x)

        self.y.append(y)

        self.vx.append(vx)

        self.vy.append(vy)

        self.ax.append(0)

        self.ay.append(0)

        self.displaysize.append(displaysize)

        self.planet.append(canvas.create\_oval(self.x[self.planets]-self.displaysize[self.planets]/2, self.y[self.planets]-self.displaysize[self.planets]/2, self.x[self.planets]+self.displaysize[self.planets]/2, self.y[self.planets]+self.displaysize[self.planets]/2, fill="red"))

        self.planets=self.planets+1;

**def** move\_planets(self):

**for** t1 **in** range (0,self.planets):

            force\_x=0

            force\_y=0

**for** t2 **in** range (0,self.planets):

**if** **not**(t1==t2):

                    distance\_squared = ((self.x[t1]-self.x[t2])\*\*2+(self.y[t1]-self.y[t2])\*\*2)

                    distance=math.sqrt(distance\_squared)

**if** (distance<1): distance = 1   # this is an overflow protection

                    force\_x=force\_x-g\_constant\*self.mass[t1]\*self.mass[t2]\*(self.x[t1]-self.x[t2])/(distance\*distance\_squared)

                    force\_y=force\_y-g\_constant\*self.mass[t1]\*self.mass[t2]\*(self.y[t1]-self.y[t2])/(distance\*distance\_squared)

            self.ax[t1]=force\_x/self.mass[t1]

            self.ay[t1]=force\_y/self.mass[t1]

            self.vx[t1]=self.vx[t1]+self.ax[t1]\*delta\_t

            self.vy[t1]=self.vy[t1]+self.ay[t1]\*delta\_t

**for** t1 **in** range (0,self.planets):

**if** (self.x[t1]+self.vx[t1]\*delta\_t<0) **or** (self.x[t1]+self.vx[t1]\*delta\_t>1200) :

                self.vx[t1]=-self.vx[t1]/2

**if** (self.y[t1]+self.vy[t1]\*delta\_t<0) **or** (self.y[t1]+self.vy[t1]\*delta\_t>600) :

                self.vy[t1]=-self.vy[t1]/2

            self.x[t1]=self.x[t1]+self.vx[t1]\*delta\_t

            self.y[t1]=self.y[t1]+self.vy[t1]\*delta\_t

**for** t1 **in** range (0,self.planets):

             self.canvas.move(self.planet[t1], self.vx[t1]\*delta\_t,self.vy[t1]\*delta\_t)

        self.canvas.after(5, self.move\_planets)

# Initialize root Window and canvas.

root = Tk()

root.title("planet")

root.resizable(False,False)

canvas = Canvas(root, width = 700, height = 600)

canvas.pack()

# Creating 1 sun, 1 asteroid, 2 planets and 2 moons.

planets = Planets(canvas)

planets.init\_planet(canvas,1000,5,350,300,-0.03,0)

planets.init\_planet(canvas,5,5,350,200,3.15-0.03,0)

planets.init\_planet(canvas,5,5,350,100,2.29-0.03,0)

planets.init\_planet(canvas,0.05,1,350,210,3.95-0.03,0)

planets.init\_planet(canvas,0.05,1,350,120,2.85-0.03,0)

planets.init\_planet(canvas,0.01,1,50,150,0,0.7)

planets.move\_planets()

# Run simulation.

root.mainloop()

Script 3.4: Code for the simulation of a arbritrary solar system in an imaginary universe.

A separate window will be opened by spyder in which the simulation is depicted. Check the commando toolbar for the window.

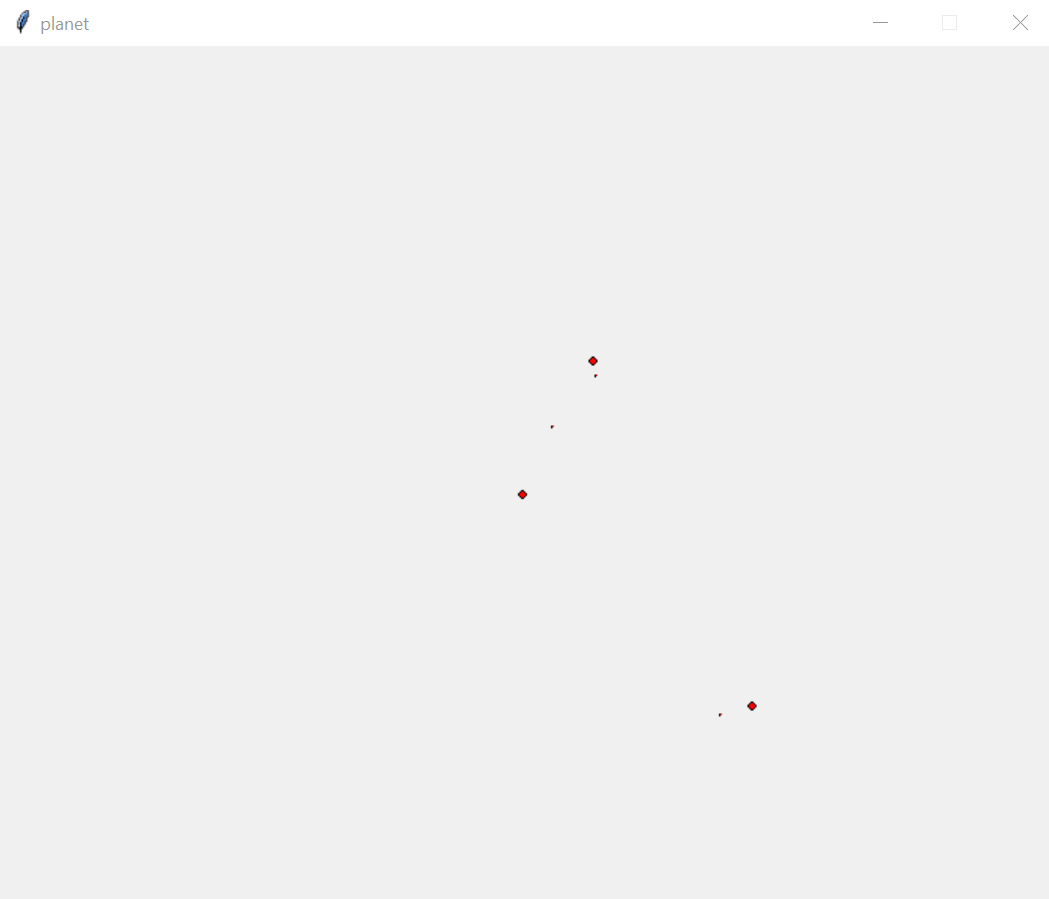


Figure 3.5: A screenshot of the simulation of the solar system.

# Systems in physics with multiple independent variables

In this chapter we will focus on modelling and simulation of systems in physics with multiple independent variables. Generally this will be spatially distributed systems where one or more parameters vary over both position and time. An example is the temperature in a room. For each position in the room the temperature can be slightly different, while the whole spatial temperature distribution can varying in time. In case of multiple independent variables the differential equation is called a Partial Differential Equation. Again, for every example, we look at the equations, turn them into a model, analyse the model and set up a Python simulation.

## Modeling and simulation of one-dimensional systems changing in time

The last example in the previous chapter seemed to be a two-dimensional system, however the simulation only handled a few separate objects in a two-dimensional space. If a certain parameter needs to be determined for all the positions in space the situation becomes more complicated. The temperature distribution in a room or a piece of metal is described by the temperature at each position in space. To simulate such a system the key point is to divide the system in small “cells”. The temperature of 1 cell will (for small time steps) only be influenced by its nearest neighbours. If the cells are small enough and the times step short enough the equations for one cell are relatively straightforward. These relatively simple equations can be performed for all the cells each time step. In this way the temperature distribution and variation over time can be computed. In this chapter we will start with a relative simple case, a diffusion profile in a long narrow tube.

### Physics of concentration diffusion

If you carefully put a drop of syrup in a glass of water, you will find that the colour gradually spreads over the entire glass, even if you do not stir or shake the glass (If you do this spreading will go much faster). This mixing is caused by a process called diffusion, which is caused by the random motion of the molecules. It also occurs in gasses and solids.

Diffusion occurs when the concentration profile of some solute varies spatially in its solvent. For example the concentration of chlorine in water. The diffusion causes the solute to spread until the concentration is the same everywhere. If present, flow of the solvent would also influence the concentration profile. In this chapter we will assume that there is no flow.

The concentration profile can, in general, vary in three spatial dimensions and in time. To keep things relatively simple we will restrict ourselves to a problem with one spatial dimension. The system we will study is a straight tube of length with a fixed cross-section , filled with water with a chlorine concentration profile that varies only in the direction along the length of the tube and in time.

### The partial differential equation

The system just described has only one spatial dimension. The diffusion equation, or Fick’s Law, then states that the diffusion flux , which is the amount of solute passing per unit time and area at position and time , is proportional and opposite to the concentration gradient . (In a three-dimensional problem the gradient is a more complicated object, see appendix C, section C.4.1, but in one dimension it is simply the spatial derivative.) That is,

|  |  |
| --- | --- |
|  | (4.1) |

where the constant of proportionality is the so-called diffusion constant, which can depend on temperature, pressure, solute and solvent. If is given in mol/m3, then gives the amount of solute passing per unit area in mol/(s·m2). If is given in kg/m3, then is in (s·m2). The curly ’s are used to indicate that it is a partial derivative (see appendix C, section C.4.1).

In general, due to this diffusion, the concentration changes in time, and therefore also the diffusion flux. The equation that governs this can be obtained by invoking mass conservation, which in one dimension and in terms of concentration reads

|  |  |
| --- | --- |
|  | (4.2) |

Combining this with Fick’s Law one can obtain

|  |  |
| --- | --- |
|  | (4.3) |

This equation is first order in time and second order in space. Given an initial condition and two boundary conditions, it will have a unique solution. The initial condition is simply the concentration profile at some (initial) time , which must already comply with the boundary conditions. The boundary conditions must specify what happens at the ends of the tubes. The concentration could be fixed in time to a certain value if it is connected to a very large (stirred) tank of that concentration, possibly zero of course. Or there could be a dead end, in which case the gradient must be zero there, because otherwise there would be diffusion. More types of boundary conditions are possible, but will not be considered here.

As before, we will first look at the equilibrium distribution. In equilibrium and Fick’s second law, equation, becomes

|  |  |
| --- | --- |
|  | (4.4) |

Integrating twice gives the general solution

|  |  |
| --- | --- |
|  | (4.5) |

We have found that the equilibrium concentration profiles are linear. Say, the length of the tube is and at one end, , the concentration is fixed at and at the other end, , the concentration is fixed at , then the concentration profile is

|  |  |
| --- | --- |
|  | (4.6) |

To study the time dependence, we will resort to computational methods.

### Turning a PDE into a difference equation

Let us now take the following case: a rectangular tube (length 10 cm, width and depth 1.5 cm) with water with a concentration profile of chlorine Cl-. The concentration along the tube at the start of the simulation is zero. Only at the right end a fixed concentration (of 0.001) is maintained. To simplify the system we will assume that the chlorine concentration is constant in the direction perpendicular to the length of the tube. The diffusion coefficient of Cl- is m2/s.

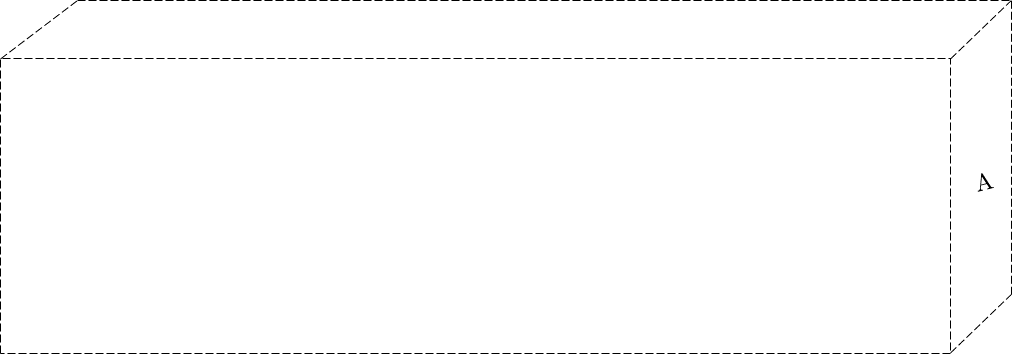


Figure 4.1: A rectangular tube filled with water.

We want to simulate the diffusion of Cl- through the tube over time. For this we divide the tube in very small slices as depicted in Figure 4.2. Each slice has a volume of . We also chop up time into small steps .

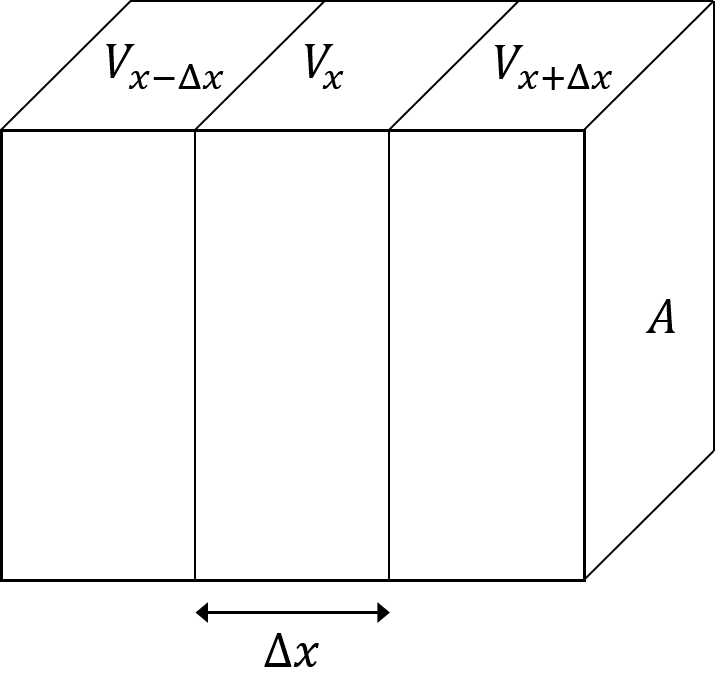


Figure 4.2: A small slice of the rectangular tube.

We will use Fick’s first law (4.1) to determine the concentration flux between each couple of slices. These fluxes change in time, but as long as the slices are thin enough and time is chopped up into small enough steps, it is reasonable to assume that within each time step the concentration in each slice is constant. Then the flow rate of chlorine from slice to slice per unit area is equal to:

|  |  |
| --- | --- |
|  | (4.7) |

and the flow rate from to per unit area is equal to:

|  |  |
| --- | --- |
|  | (4.8) |

The change of the concentration in volume after time step is then given by:

|  |  |
| --- | --- |
|  | (4.9) |

The left part is the change of the total amount of Cl- in volume (please note, concentration x volume = total amount). The right part is the flow rate of Cl- towards the volume. Because this simplifies to:

|  |  |
| --- | --- |
|  | (4.10) |

Which gives:

|  |  |
| --- | --- |
|  | (4.11) |

From this we can deduce to the following difference equation:

|  |  |
| --- | --- |
|  | (4.12) |

Where the index n denotes the passing of time.

A different way to achieve the same result is via Taylor series. The Taylor series (truncated after the third term) is given by (see also appendix E.1.3):

|  |  |  |
| --- | --- | --- |
|  |  | (4.13) |
|  |  | (4.14) |

Adding the two together gives

|  |  |
| --- | --- |
|  | (4.15) |

Combining this with Fick’s second law (4.3) (taking ) gives again (4.12).

### Visualizing the spatial difussion proces

Below the script to simulate the diffusion profile. Please note that the difference equation from the previous paragraph can be easily recognized in the code.

"""

@author: Bart Bozon

script to simulate a diffusion profile

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

# Set up variables.

arraysize=100

conc\_profile\_old =np.zeros(arraysize)

conc\_profile\_new =np.zeros(arraysize)

# defining constants

D=1.25e-9

dx=0.001 # 1 mm grid, 100 points, = 10 cm

dt=100     # 100 seconde

x\_axis = np.arange(0.0,arraysize\*dx, dx)

conc\_profile\_new[arraysize-1]=0.001

# Run simulation.

plt.plot(x\_axis,conc\_profile\_old)

**for** t **in** range (0,1001):

    conc\_profile\_old[arraysize-1]=0.001

**for** x **in** range (1,arraysize-1):

        conc\_profile\_new[x]=conc\_profile\_old[x]+dt\*D\*(-2\*conc\_profile\_old[x]+conc\_profile\_old[x-1]+conc\_profile\_old[x+1])/(dx\*dx)

    conc\_profile\_new[0]=conc\_profile\_old[0]+dt\*D\*(-1\*conc\_profile\_old[0]+conc\_profile\_old[1])/(dx\*dx)

**if** (t%100==0):

       plt.plot(x\_axis,conc\_profile\_old)

    conc\_profile\_old=conc\_profile\_new

# Display results.

plt.xlabel('X position [m]')

plt.ylabel('Concentration')

plt.show()

Script 4.1: Code for the simulation of a diffusion profile.

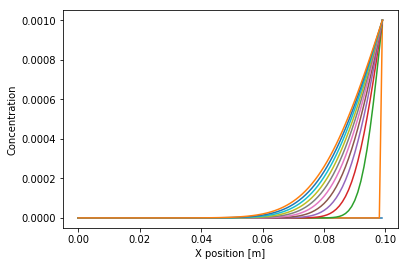


Figure 4.3: A screenshot of the simulation of a diffusion profile.

This seems to be a peculiar result (very, very slow).

**Exercise 4.1.** Verify with the following calculator the result of the simulation:

<http://www.physiologyweb.com/calculators/diffusion_time_calculator.html>

### Visualizing the Temperature profile

The previous simulation is very similar to heat conduction in a solid rod (see also the next paragraph). The difference equation for heat conduction is given by:

|  |  |
| --- | --- |
|  | (4.16) |

, the thermal diffusivity, is given by:

|  |  |
| --- | --- |
|  | (4.17) |

For aluminium it is equal to 9.7 x 10-5 m2/s. The following code simulates the temperature distribution in a metal rod over time.

"""

@author: Bart Bozon

script to simulate a temperature profile

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

# Set up variables.

arraysize=100

temp\_profile\_old =np.zeros(arraysize)

temp\_profile\_new =np.zeros(arraysize)

# defining constants

alfa=9.7e-5 # alfa of aLuminium

dx=0.001 # 1 mm grid, 100 points, = 0.1 m

dt=0.001  # 10 seconde

x\_axis = np.arange(0.0,arraysize\*dx, dx)

# Run simulation.

plt.plot(x\_axis,temp\_profile\_old)

**for** t **in** range (0,10001):

    temp\_profile\_old[arraysize-1]=100

    temp\_profile\_old[0]=0

**for** x **in** range (1,arraysize-1):

        temp\_profile\_new[x]=temp\_profile\_old[x]+dt/(dx\*dx)\*alfa\*(-2\*temp\_profile\_old[x]+temp\_profile\_old[x-1]+temp\_profile\_old[x+1])

**if** (t%(500)==0):

       plt.plot(x\_axis,temp\_profile\_old)

    temp\_profile\_old=temp\_profile\_new

# Display results.

plt.xlabel('x [m]')

plt.ylabel('Temperature [c]')

plt.show()

**print** ("elapsed time", t\* dt ,"seconds")

Script 4.2: Code for the simulation of a temperature profile.

Please note the boundary conditions have been chosen differently than for the diffusion profile. In this case both ends of the rod were kept on a constant temperature.

The result:

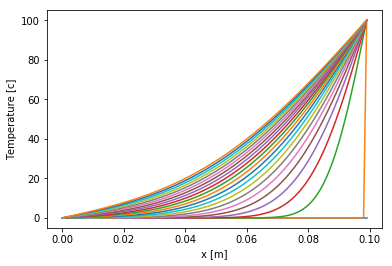


Figure 4.4: A screenshot of the simulation of a temperature profile in a metal rod.

**Exercise** **4.2.** Run the simulation for a different material; stainless steel.

**Exercise** **4.3.** Change the material back to aluminium. Change the boundary condition in the simulation such that the left end of the rod is totally isolated. Change the step time to 0.002 second.

**Exercise 4.4.** Change the boundary conditions such that the rod is totally isolated. Start with the situation that the middle 2 centimeters is 100°C and the rest of the rod 0°c. Change to 0.0005.

If we would like to see the temperature vs time at a specific point we could use the following code:

"""

@author: Bart Bozon

script to simulate a temperature profile with time axis

"""

**import** copy

**import** numpy as np

**import** matplotlib.pyplot as plt

# Set up variables.

arraysize=100

temp\_profile\_old =np.zeros(arraysize)

temp\_profile\_new =np.zeros(arraysize)

# defining constants

alfa=9.7e-5 # alfa of aLuminium

dx=0.001 # 1 mm grid, 100 points, = 0.1 m

dt=0.001     # 100 seconde

simulation\_time=50

steps\_to\_show = 50

temp\_points=np.zeros(steps\_to\_show+1)

temp\_axis =np.zeros(steps\_to\_show+1)

x\_axis = np.arange(0.0,arraysize\*dx, dx)

temp\_profile\_old[arraysize-1]=0.001

temp\_teller=0

# Run simulation.

plt.plot(x\_axis,temp\_profile\_old)

**for** t **in** range (0,int(simulation\_time/dx)+1):

    temp\_profile\_old[arraysize-1]=100

    temp\_profile\_old[0]=0

**for** x **in** range (1,arraysize-1):

        temp\_profile\_new[x]=temp\_profile\_old[x]+dt/(dx\*dx)\*alfa\*(-2\*temp\_profile\_old[x]+temp\_profile\_old[x-1]+temp\_profile\_old[x+1])

    temp\_profile\_new[0]=temp\_profile\_old[0]+dt/(dx\*dx)\*alfa\*(-1\*temp\_profile\_old[0]+temp\_profile\_old[1])

**if** (t%(int((simulation\_time/dx)/steps\_to\_show))==0):

       plt.plot(x\_axis,temp\_profile\_old)

       temp\_points[temp\_teller]=temp\_profile\_new[int(arraysize/2)]

       temp\_axis[temp\_teller]=t\*dt

       temp\_teller=temp\_teller+1

    temp\_profile\_old=copy.deepcopy(temp\_profile\_new)

# Display results.

plt.xlabel('x [m]')

plt.ylabel('temperature [c]')

plt.show()

**print** ("elapsed time", t\* dt ,"seconds")

**print** ("time elapsed per line",simulation\_time/steps\_to\_show,"seconds" )

plt.plot(temp\_axis,temp\_points)

plt.xlabel('time [s]')

plt.ylabel('temperature [c]')

plt.show()

Script 4.3: Code for the simulation of a temperature profile with time axis.

This gives an extra graph in which the temp vs time for a point is given:

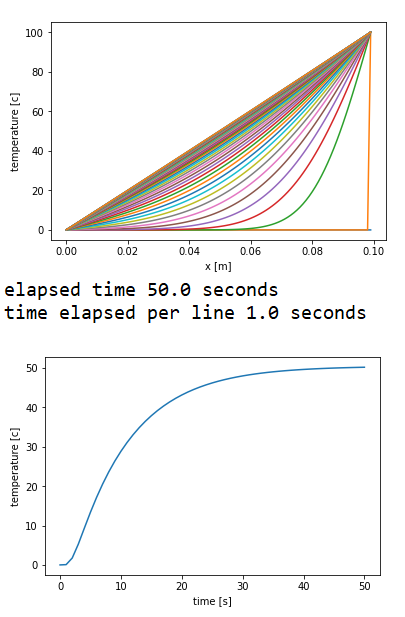


Figure 4.5: A screenshot of the simulation of a temperature profile in a metal rod at a specific point.

## One-dimensional systems with external effects

In this section we will continue with a similar one-dimensional system, heat diffusion (or conduction) through a straight rod with constant cross section (the differential equation is actually the same). However, we will now add external influences to it and study the effects.

### Physics of heat diffusion in one dimension

Fourier’s Law describes the rate of flow of heat energy per unit area caused by a temperature gradient

|  |  |
| --- | --- |
|  | (4.18) |

where is the temperature at position and time , and is the thermal conductivity of the material. Combining this with the heat capacity of the material and conservation of energy, this can be turned into the Heat Equation (see for example (van den Akker & Mudde, 1998)), which describes the temperature variation in a solid in time and space as a result of heat diffusion. In one dimension it reads

|  |  |
| --- | --- |
|  | (4.19) |

where is the thermal diffusivity, which depends on thermal conductivity , heat capacity and density of the solid as

|  |  |
| --- | --- |
|  | (4.20) |

Interestingly, Fourier’s Law and the heat equation are mathematically equivalent to the Fick’s First and Second Law respectively.[[4]](#footnote-5) If the material is not the same over the entire length of the rod, these quantities will depend on and equation (4.19) is not valid in that form. This will be discussed in more detail in section 4.3.

### Adding heat production

The heat conducting rod could be producing heat itself. For example, if an electric current would be passing through. Alternatively, small heat sources could be places inside the rod. (Small enough to neglect their disturbance on the heat diffusion through the rod as a whole.) It also possible that heat is extracted from certain areas through cooling. Such heat drains will often be referred to as sources as well, because mathematically it is the except for a minus sign. The general case would be a continuous distribution of space and time dependent heat sources with a net power density . This can be added to the heat equation as

|  |  |
| --- | --- |
|  | (4.21) |

Concerning boundary and initial conditions, the situation is similar to diffusion, but with different physical meaning. The addition of heat sources does not change that.

Some initial condition must be specified, which is the initial temperature profile at . For the boundary, different conditions are possible. One common situation is to assume a fixed temperature, which could be caused for example by an ice bath, which keeps at 273 K. Another common situation is to assume perfect isolation at a boundary. Then the heat flow should be zero, which means that is zero at this end.

Completely perfect isolation is not very realistic. In general, the heat flow at the boundary will be proportional to the temperature difference and the heat transfer coefficient at the boundary. For example, if the temperature just outside the boundary at is , then we will have outside the boundary at is , then we will have

|  |  |
| --- | --- |
|  | (4.22) |

In principle, boundary conditions can also very in time, but we will not treat such problems in this reader.

### Analytic solutions for the equilibrium distribution

In section 4.1.2 we found for Fick’s Second Law describing diffusion in one-dimension that the equilibrium concentration profile is linear. Since the heat equation without internal heat sources is mathematically the same, the equilibrium one-dimensional temperature profile outside heat sources will be linear as well.

**Exercise 4.5.** Determine the temperature profile for a rod of length with a fixed temperature at and at .

Inside the heat sources the situation is more complicated for analytical analysis. In case the power density inside a heat source is constant over its length, the equilibrium temperature profile inside it will be quadratic.

**Exercise** **4.6.** Show that the equilibrium temperature profile inside a heat source with constant power density over its length is quadratic.

### Converging the simulation to the equilibrium distribution

We will now introduce a heater and a cooler in the rod itself. This means that for a specific slice an extra term needs to be added to the difference equation:

|  |  |
| --- | --- |
|  | (4.23) |

Where [m3] is equal to the volume of the slice and [J.s-1.m-1] is the net energy flow produced by the heater and the cooler per length of rod (so for the heater will be positive and for the cooler it will be negative). [J.kg-1.K-1] is the specific heat capacity and [kg.m-3] the density.

As the volume is equal to the frontal area of the rod times , we get:

|  |  |
| --- | --- |
|  | (4.24) |

We see that disappears from the second part of the equation. This is what we would expect, the rise of temperature with time should not be dependent on the thickness of the slice.

We will define the following situation:

A rectangular aluminium rod (length 10 cm, width and depth 1.5 cm) with a heater from = 1 till = 2 cm. A cooler from = 8 till = 9 cm. The temperature along the rod at the start of the simulation is 0°C. The rod is perfectly isolated, so no heat is lost via its sides. The values for the heater and cooler is 10 watt total. This means for , 10 watt per 1 cm so = 1000 [J·s-1·m-1].

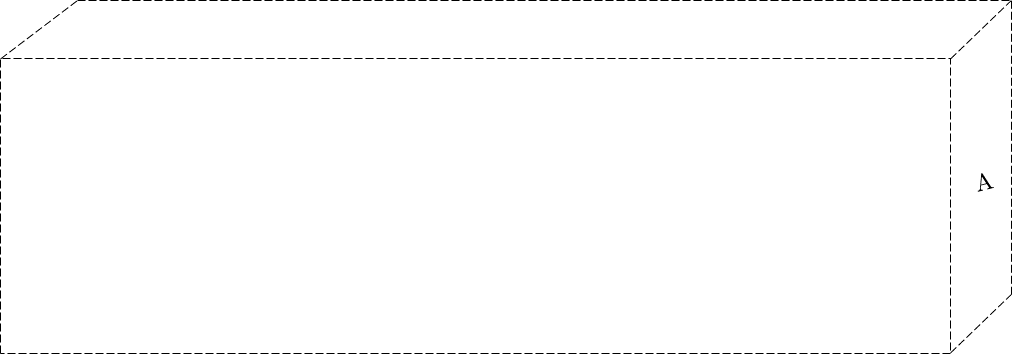


Figure 4.6: A rectangular aluminum rod.

Below is the script to simulate the previous described situation.

"""

@author: Bart Bozon

script to simulate a temperature profile with a heater and cooler

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

**import** copy

# Set up variables.

arraysize=100

temp\_profile\_old =np.zeros(arraysize)

temp\_profile\_new =np.zeros(arraysize)

pnetto =np.zeros(arraysize)

pnetto[10:20]=1000

pnetto[80:90]=-1000

# defining constants

alfa=9.7e-5 # alfa of aLuminium

dx=0.001 # 1 mm grid, 100 points, = 0.1 m

dt=0.0005  # 10 seconde

A=0.015 \* 0.015

rho=2700

c= 897

x\_axis = np.arange(0.0,arraysize\*dx, dx)

# Run simulation.

plt.plot(x\_axis,temp\_profile\_old)

**for** t **in** range (0,10001):

    temp\_profile\_old[arraysize-1]=temp\_profile\_old[arraysize-2]

    temp\_profile\_old[0]=temp\_profile\_old[1]

**for** x **in** range (1,arraysize-1):

        temp\_profile\_new[x]=temp\_profile\_old[x]+dt\*pnetto[x]/(A\*rho\*c)+dt/(dx\*dx)\*alfa\*(-2\*temp\_profile\_old[x]+temp\_profile\_old[x-1]+temp\_profile\_old[x+1])

**if** (t%(500)==0):

       plt.plot(x\_axis,temp\_profile\_old)

    temp\_profile\_old=copy.deepcopy(temp\_profile\_new)

# Display results.

plt.xlabel('x [m]')

plt.ylabel('temperature [c]')

plt.show()

**print** ("elapsed time", t\* dt ,"seconds")

Script 4.4: Code for the simulation of a temperature profile.

The result:

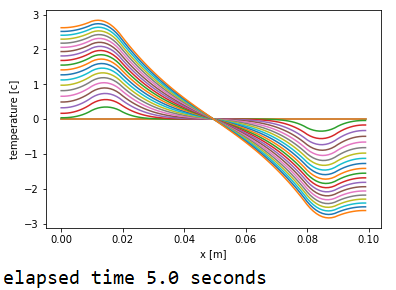


Figure 4.7: Heating profile in an aluminium rod.

**Exercise 4.7.** Run the simulation for 10 time as long. What is the in steady state?

**Exercise 4.8.** In steady state the heat energy flow in Fourier’s law (4.18) is constant. Argue what its value should be in the area between the heater and the cooler and use this to check if the simulation is correct.

**Exercise** **4.9.** (Extra challenge) Determine the steady state temperature profile over the entire length of the rod. Hints: Use Fourier’s Law to determine the temperature difference over the part in between the heater and cooler (from to cm), use the fact that the temperature in the centre is zero, and use the result of exercise 4.6 to determine inside the heater and cooler. The latter requires also to use that is *continuous* everywhere, meaning that both and its derivative do not jump from one value to another at any point in space. Plot the result and compare this to the simulation. What is between the outer ends of the rod?

## One-dimensional systems with discontinuities

In this section we will continue with a similar one-dimensional system. However, we will now add discontinuities in the internal composition and study the effects.

### Physics of heat diffusion through multiple materials

Our rod can consist of segments of different materials. The border between two materials can be very abrupt, but it can also be very gradual. As mentioned in 4.2.1, in this case equation (4.19) is not valid in that form, because the thermal conductivity , heat capacity and density will vary over the length of the rod. Next we will introduce the proper form of the one-dimensional heat equation for this situation.

### The border between the two materials

If the material is not the same over the entire length of the rod, , and will depend on . In that case the heat equation can be written as

|  |  |
| --- | --- |
|  | (4.25) |

Abrupt changes in the properties can be mathematically incorporated using the Heaviside step function, see for example (Borrelli & Coleman, 2004). In the extra challenging exercise 4.9 it was necessary to use the fact that has to be continuous. In case of abrupt material changes this is no longer the case; now the analytic solution for has to be *piecewise continuous*, meaning that its derivative can jump from one value to another. The value of can of course not jump from one value to another. (We will not go into the full mathematical methods to deal with this, but without too much of a problem we will find the steady state solution over such an abrupt change in an area without heat sources in 4.3.3.) In numerical methods the volume of the material is divided into small discrete pieces and it is just a question of properly assigning the properties to each piece.

Let’s now look in detail at this problem. Below we have part of a metal rod consisting of two materials (metal a and metal b) for which the boundary is between and .

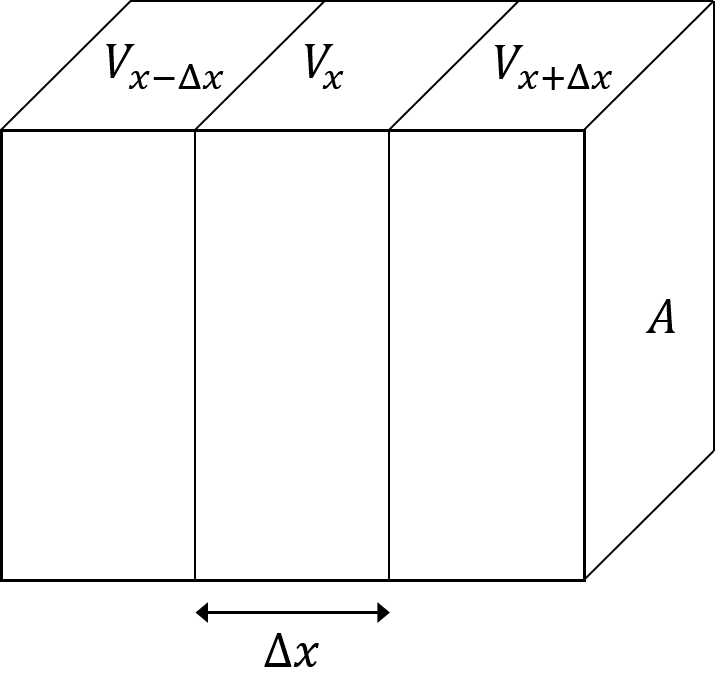


Figure 4.8: A part of a metal rod consisting of two materials (metal a and metal b) for which the boundary is between and .

The heat flow from towards can then be approximated by:

|  |  |
| --- | --- |
|  | (4.26) |

Key for understanding this is that the conduction between and is determined by the worst conductor. If = 10 and = 0.1 the formula gives 0.19 for the total k (which is logic as half of the material is the bad conductor). When = 10 and = 10 the formula gives 10 for the total (which also appeals to our sense of a perfect world).

The change in temperature of is given by:

|  |  |
| --- | --- |
|  | (4.27) |

This results in the following difference equation:

|  |  |
| --- | --- |
|  | (4.28) |

In the following program this is used to simulate the previous rod, but now half of the material is aluminium and the other half is stainless steel:

"""

@author: Bart Bozon

script to simulate a temperature profile with ALU **and** SS

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

**import** copy

# Set up variables.

arraysize=100

temp\_profile\_old =np.zeros(arraysize)

temp\_profile\_new =np.zeros(arraysize)

pnetto =np.zeros(arraysize)

pnetto[10:20]=1000

pnetto[80:90]=-1000

# defining constants

k =np.zeros(arraysize)

rho =np.zeros(arraysize)

c =np.zeros(arraysize)

k\_alu=237 # values of aLuminium

rho\_alu=2700

c\_alu= 897

k\_ss=14.4 # values of stainless steel

rho\_ss=7500

c\_ss= 502

k[0:49]=k\_alu

k[49:100]=k\_ss

c[0:49]=c\_alu

c[49:100]=c\_ss

rho[0:49]=rho\_alu

rho[49:100]=rho\_ss

dx=0.001 # 1 mm grid, 100 points, = 0.1 m

dt=0.0005  # 5 seconde

A=0.015 \* 0.015

x\_axis = np.arange(0.0,arraysize\*dx, dx)

# Run simulation.

plt.plot(x\_axis,temp\_profile\_old)

**for** t **in** range (0,10001):

    temp\_profile\_old[arraysize-1]=temp\_profile\_old[arraysize-2]

    temp\_profile\_old[0]=temp\_profile\_old[1]

**for** x **in** range (1,arraysize-1):

        temp\_profile\_new[x]=temp\_profile\_old[x]+dt\*pnetto[x]/(A\*rho[x]\*c[x])+dt/(dx\*dx)/(rho[x]\*c[x])\*(-temp\_profile\_old[x]\*(2\*k[x]\*k[x-1]/(k[x]+k[x-1]))-temp\_profile\_old[x]\*(2\*k[x]\*k[x+1]/(k[x]+k[x+1]))+temp\_profile\_old[x-1]\*(2\*k[x]\*k[x-1]/(k[x]+k[x-1]))+temp\_profile\_old[x+1]\*(2\*k[x]\*k[x+1]/(k[x]+k[x+1])))

**if** (t%(500)==0):

       plt.plot(x\_axis,temp\_profile\_old)

    temp\_profile\_old=copy.deepcopy(temp\_profile\_new)

# Display results.

plt.xlabel('x [m]')

plt.ylabel('temperature [c]')

plt.show()

**print** ("elapsed time", t\* dt ,"seconds")

**print** (np.sum(temp\_profile\_old[0:49]))

**print** (np.sum(temp\_profile\_old[49:100]))

Script 4.5: Code for the simulation of a temperature profile with multiple materials.

The result:

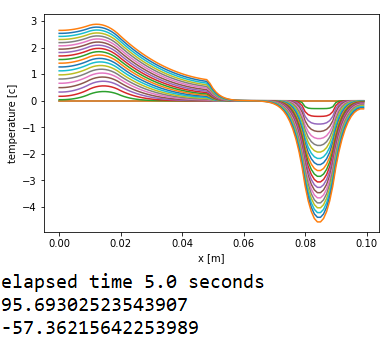


Figure 4.9: Heating profile in an aluminium/stainless steel rod.

**Exercise 4.10.** Compare this result with Figure 4.7. Do you understand why the highest temperature is higher in the stainless steel part?

**Exercise 4.11.** The last two numbers of Figure 4.9 depict the integral of the temperature of respectively the aluminium and the stainless steel part. Do you understand why the cumulated temperature is higher in the aluminium part?

### The equilibrium distribution

In parts of the rod without heat sources or sinks we can find the steady state, or equilibrium distribution, over the abrupt material change by using Fourier’s Law (4.18). We will make this a bit easier for ourselves by considering the rod of length cm of the previous section, but instead of a heater and a cooler we will just fix the end temperatures to and .

In steady state the heat flow will be constant. Hence, Fourier’s Law (4.18) tells us that on the aluminium side

|  |  |
| --- | --- |
|  | (4.29) |

while on the stainless steel side

|  |  |
| --- | --- |
|  | (4.30) |

**Exercise 4.2.** Derive (4.29) and (4.30).

From this it becomes clear that at cm the slope (i.e. the derivative) of the temperature profile will change. But the temperature itself must of course be the same at cm for both parts of the solution. Hence,   . This leads to

|  |  |
| --- | --- |
|  | (4.31) |

which the mechanical engineers amongst you will probably recognize quickly as

|  |  |
| --- | --- |
|  | (4.32) |

### Converging to the equilibrium distribution

In the below script the simulated time is made sufficiently long enough to reach the equilibrium situation.

"""

@author: Bart Bozon

script to simulate a temperature profile with fixed boundary temperatures

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

**import** copy

# Set up variables.

arraysize=100

temp\_profile\_old =np.zeros(arraysize)

temp\_profile\_new =np.zeros(arraysize)

pnetto =np.zeros(arraysize)

# defining constants

k =np.zeros(arraysize)

rho =np.zeros(arraysize)

c =np.zeros(arraysize)

k\_alu=237 # values of aLuminium

rho\_alu=2700

c\_alu= 897

k\_ss=14.4 # values of stainless steel

rho\_ss=7500

c\_ss= 502

k[0:49]=k\_alu

k[49:100]=k\_ss

c[0:49]=c\_alu

c[49:100]=c\_ss

rho[0:49]=rho\_alu

rho[49:100]=rho\_ss

dx=0.001 # 1 mm grid, 100 points, = 0.1 m

dt=0.005 # Time based changed!!

A=0.015 \* 0.015

x\_axis = np.arange(0.0,arraysize\*dx, dx)

# Run simulation.

plt.plot(x\_axis,temp\_profile\_old)

**for** t **in** range (0,50001):

    temp\_profile\_old[arraysize-1]=0

    temp\_profile\_old[0]=100

**for** x **in** range (1,arraysize-1):

        temp\_profile\_new[x]=temp\_profile\_old[x]+dt\*pnetto[x]/(A\*rho[x]\*c[x])+dt/(dx\*dx)/(rho[x]\*c[x])\*(-temp\_profile\_old[x]\*(2\*k[x]\*k[x-1]/(k[x]+k[x-1]))-temp\_profile\_old[x]\*(2\*k[x]\*k[x+1]/(k[x]+k[x+1]))+temp\_profile\_old[x-1]\*(2\*k[x]\*k[x-1]/(k[x]+k[x-1]))+temp\_profile\_old[x+1]\*(2\*k[x]\*k[x+1]/(k[x]+k[x+1])))

**if** (t%(500)==0):

       plt.plot(x\_axis,temp\_profile\_old)

    temp\_profile\_old=copy.deepcopy(temp\_profile\_new)

# Display results.

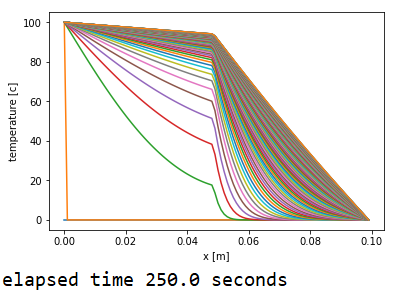
plt.xlabel('x [m]')

plt.ylabel('temperature [c]')

plt.show()

**print** ("elapsed time", t\* dt ,"seconds")

Script 4.6: Code for the simulation of a temperature profile with different materials, with fixed boundary temperatures.



## Modelling and simulation of two-dimensional systems changing in time

In this section we will start with two-dimensional systems. In addition optimisation techniques are discussed.

### Physics of heat diffusion in three dimension

In three dimension the heat equation is

|  |  |
| --- | --- |
|  | (4.33) |

Using the Laplacian, see appendix C, section C.4.1, this can be written more compactly as

|  |  |
| --- | --- |
|  | (4.34) |

Allowing for different materials, the proper heat equation is

|  |  |
| --- | --- |
|  | (4.35) |

### The two-dimensional partial differential equation

In this section we will study a two-dimensional problem. This could for example be the temperature distribution of a plate. For a single material, we should then solve

|  |  |
| --- | --- |
|  | (4.36) |

To simulate a real situation, an initial condition, an initial temperature profile , has to be specified. In addition, boundary conditions have to be specified along the entire boundary. The simplest case is just a fixed boundary temperature profile. Slightly more complex is the perfect isolation case. Other types of boundary conditions fix the gradient of in the direction perpendicular to the boundary surface (which is one-dimensional in the two-dimensional case) to some value or some function, which can involve at the boundary itself, as in (4.22). This is beyond the scope of this reader.

### Discretizing multi-dimensional PDEs

For a 2-dimensional case we have

Using the spatial discretization and labelling shown in Figure 4.10, this can be transformed into the following difference equation:

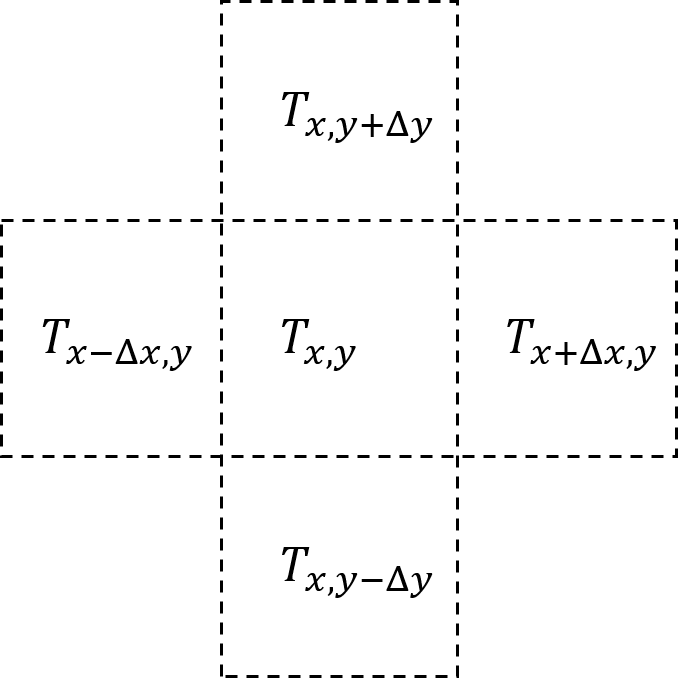
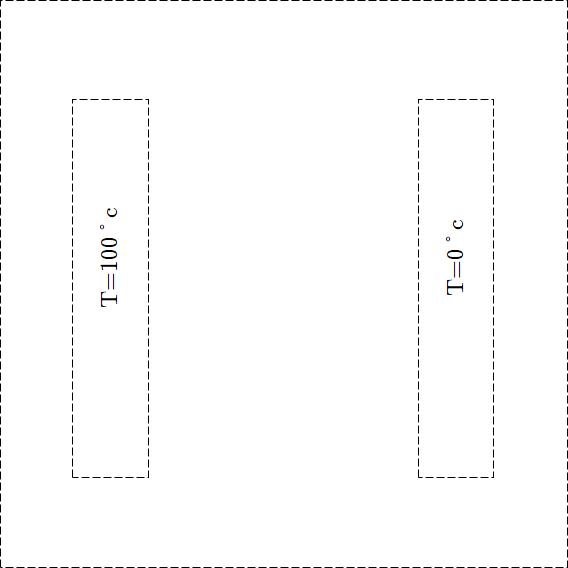


Figure 4.10: Discretization and labeling of two dimensional space.

### Visualization through heat maps

We will now simulate the following 2 dimensional case:



An aluminium plate with dimensions 15x15 cm and two zones which are held on a constant temperature of 100⁰C and 0⁰C. The starting temperature of the rest of the material is 50⁰C. The sides of the plate are again perfectly isolated.

Run the following code:

"""

@author: Bart Bozon

script to simulate a 2d temperature profile

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

**import** copy

# Set up variables& constants

arraysize=150

temp\_old =np.ones((arraysize,arraysize))

temp\_new =np.ones((arraysize,arraysize))

temp\_old=50\*temp\_old

alfa=9.7e-5 # alfa of aLuminium

dx=0.001 # 1 mm grid, 150 points, = 0.15 m

dt=0.001

# Run simulation.

**for** t **in** range (0,100001):

    temp\_old[30:120,20:30]=100

    temp\_old[30:120,120:130,]=0

**for** x **in** range (1,arraysize-1):

**for** y **in** range (1,arraysize-1):

        temp\_new[y,x]=temp\_old[y,x]+dt/(dx\*dx)\*alfa\*(-4\*temp\_old[y,x]+temp\_old[y,x-1]+temp\_old[y,x+1]+temp\_old[y+1,x]+temp\_old[y-1,x])

    temp\_new[0:arraysize-1,0]=temp\_new[0:arraysize-1,1]

    temp\_new[0:arraysize-1,arraysize-1]=temp\_new[0:arraysize-1,arraysize-2]

    temp\_new[0,0:arraysize-1]=temp\_new[1,0:arraysize-1]

    temp\_new[arraysize-1,0:arraysize-1]=temp\_new[arraysize-2,0:arraysize-1]

**if** (t%(100)==0):

        # Display results.

        im=plt.imshow(temp\_new,cmap="gist\_ncar")

        # see https://matplotlib.org/examples/color/colormaps\_reference.html

        plt.colorbar(im)

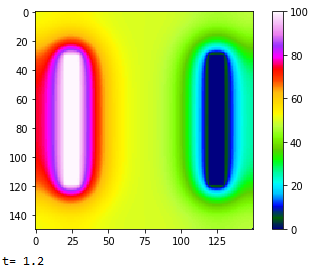
        plt.show()

**print** ("t=",t\*dt)

    temp\_old=copy.deepcopy(temp\_new)

Script 4.7: Code for the simulation of a 2D temperature profile.

We can visualize the results with so called heat maps:



As you can notice, the simulation takes quit a lot of time. One way to solve this is to increase the dt step. If you want to stop the executing of the program you can restart the kernel with ctrl+.

For this you have to place the cursor in the executing window.

**Exercise 4.12.** Increase the dt step. Save for different values a screenshot of the situation at t=1.0. When does the simulation loose it’s accuracy? When does it become unstable?

### Speed optimisation

One key thing to realize is that python is an interpreter language. This has the disadvantage that python is relatively slow in executing for loops. One way to alleviate this problem is by using matrix calculation. Python can multiply two arrays very quickly by using a built in instruction.

Run the following program in the spyder environment:

"""

@author: Bart Bozon

script to determine the speed advantage of built in instructions

"""

**import** time

**import** numpy as np

arraysize=1000

a =np.zeros(arraysize)

b =np.zeros(arraysize)

c =np.zeros(arraysize)

start\_time = time.time()

**for** t1 **in** range (0, 10000):

**for** t2 **in** range (0,arraysize-1):

        c[t2]=a[t2]\*b[t2]

elapsed\_time = time.time() - start\_time

**print**("elapsed time using for-loops", elapsed\_time)

start\_time = time.time()

**for** t1 **in** range (0, 10000):

    c=a\*b

elapsed\_time = time.time() - start\_time

**print**("elapsed time using matrix/vector", elapsed\_time)

The time difference is rather impressive:



A factor of 288 increase in speed!

A clever instruction we can use is a buildin method in numpy (we have to import numpy as np):

*rolled\_matrix*=np.roll(*original\_matrix*,*amount\_to\_roll*,*axis*)

With this instruction we can easily make reference to items like n-1 or n+1.

Try the following code:

"""

@author: Bart Bozon

Demo of np.roll

"""

**import** numpy as np

arraysize=4

a =np.zeros((arraysize,arraysize))

a[2,1]=1

**print** (a)

**print** (" 1,0 :")

**print**(np.roll(a,1,0))

**print** (" 1,1 :")

**print**(np.roll(a,1,1))

**print** (" -1,1 :")

**print**(np.roll(a,-1,1))

With this instruction we can change the program of chapter 4.4.4 to a version which runs much faster:

"""

@author: Bart Bozon

Script for 2d temperature profile, optimised for speed

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

**import** copy

# Set up variables& constants

arraysize=150

temp\_old =np.ones((arraysize,arraysize))

temp\_new =np.ones((arraysize,arraysize))

temp\_old=50\*temp\_old

alfa=9.7e-5 # alfa of aLuminium

dx=0.001 # 1 mm grid, 150 points, = 0.15 m

dt=0.001

# this function determines the delta between two shifted matrices

**def** delta\_2d (matrix):

    matrixdown=np.roll(matrix,1,0)

    matrixright=np.roll(matrix,1,1)

    deltadown =(matrixdown -matrix)

    deltaright =(matrixright -matrix)

    delta=deltadown -np.roll(deltadown,-1,0)+deltaright-np.roll(deltaright,-1,1)

**return** delta

# Run simulation.

**for** t **in** range (0,100001):

    temp\_old[30:120,20:30]=100

    temp\_old[30:120,120:130,]=0

    temp\_new=temp\_old+dt/(dx\*dx)\*alfa\*delta\_2d(temp\_old)

    temp\_new[0:arraysize-1,0]=temp\_new[0:arraysize-1,1]

    temp\_new[0:arraysize-1,arraysize-1]=temp\_new[0:arraysize-1,arraysize-2]

    temp\_new[0,0:arraysize-1]=temp\_new[1,0:arraysize-1]

    temp\_new[arraysize-1,0:arraysize-1]=temp\_new[arraysize-2,0:arraysize-1]

**if** (t%(100)==0):

        # display results.

        im=plt.imshow(temp\_new,cmap="gist\_ncar")

        # see https://matplotlib.org/examples/color/colormaps\_reference.html

        plt.colorbar(im)

        plt.show()

**print** ("t=",t\*dt)

    temp\_old=copy.deepcopy(temp\_new)

Script 4.8: Code for the simulation of a 2D temperature profile using matrices calculations.

The change in performance of the program is remarkable. Please note that due to roll over (the top of the matrix influences the bottom and the right influences the left) we have to treat the boundaries individually. In the above script it is solved by assuming perfect isolation.

## A complex 2D case

Below a script is given to simulate a complex case. Several ideas are incorporated in this script. A short description of each is given before the total script.

Matrix calulation

# functions to calculate the delta T difference between 2 points

**def** delta\_2d\_left (matrix):

    matrixleft=np.roll(matrix,1,1)

    delta =(matrixleft -matrix)

**return** delta

The script uses multiple functions for matrix calculations. This is needed because different materials connected to each other are simulated. See also the difference equation:

    # this is the actual difference equation

    temp\_new=temp\_old+pnetto\*dt/(rho\*c)+dt/(dx\*dx\*rho\*c)\*(k\_left\*delta\_2d\_left(temp\_old)+k\_right\*delta\_2d\_right(temp\_old)+k\_up\*delta\_2d\_up(temp\_old)+k\_down\*delta\_2d\_down(temp\_old))

For this the “mixed k” value needs to be calculated as well:

**def** determine\_mixed\_k\_left (matrix):

    matrixleft=np.roll(matrix,1,1)

    mixed\_k =2\*(matrixleft \*matrix)/(matrixleft+matrix)

**return** mixed\_k

Material definition

# defining material constants. We will use a dict for this    
aluminium = {

    'rho': 2700,

    'k': 237,

    'c': 897,

}

A dict is used to define the materials. The material needs to be added to the material list as well:

# we need a link between a number and the name of the material

material\_list={

        1:aluminium,

        2:stainless\_steel,

        3:teflon

}

Material assignment

# here we define the composition of our rectangle.

material\_matrix[0:arraysizey-1,0:arraysizex-1]=2

material\_matrix[10:15,10:80]=1

material\_matrix[20:25,10:80]=1

Different areas can be composed of different materials. The translation to the final material matrices is done here:

# copy all the values of the chosen materials in the different matrices y,x

**for** x **in** range (0,arraysizex):

**for** y **in** range (0,arraysizey):

        k[y,x]=material\_list[material\_matrix[y,x]]['k']

        rho[y,x]=material\_list[material\_matrix[y,x]]['rho']

        c[y,x]=material\_list[material\_matrix[y,x]]['c']

Internal heat sources or sinks

# we'll add a power source

pnetto [10:50,90:100]=2000000

We can switch off the heat source after a certain amount of time.

**if** t>500001 :

        pnetto=0\*pnetto

Visualizing the heat flow

    # we will calculate the q flow as well. This is only for display purposes

**if** (t%(10000)==0):

     qflow=abs(1/(dx\*dx\*rho\*c)\*(k\_left\*delta\_2d\_left(temp\_old)+k\_up\*delta\_2d\_up(temp\_old)))

The heat flow is calculated only once every 10000 simulations. This is because we only need it for the visuals.

Boundary conditions

    # in princple our boundaries are floating (perfectly isolated)

    temp\_new[0:arraysizey-1,0]=temp\_new[0:arraysizey-1,1]

    temp\_new[0:arraysizey-1,arraysizex-1]=temp\_new[0:arraysizey-1,arraysizex-2]

    temp\_new[0,0:arraysizex-1]=temp\_new[1,0:arraysizex-1]

    temp\_new[arraysizey-1,0:arraysizex-1]=temp\_new[arraysizey-2,0:arraysizex-1]

    # This boundary is kept at a constant temperature, we force it to 15c

    temp\_new[0:arraysizey-1,0]=15

    # for this boundary we assume it radiates energy into space

    temp\_new[0:50,arraysizex-1]=temp\_new[0:50,arraysizex-1]-5.67e-8\*dt\*(temp\_new[0:50,arraysizex-1]+273.15)\*\*4

Different boundaries can be implemented. Floating boundaries (perfect isolation), fixed or boundaries based on different heat flows (like radiation).

Recording the data

# this is used for the graph at the end

steps\_to\_show = 500

temp\_points=np.zeros((11,steps\_to\_show+1))

temp\_axis =np.zeros(steps\_to\_show+1)

temp\_teller=0

An array is created to record the data at key points. This can than be used in excel to compare the data to actual measurements. Saving of the file to HD:

**if** (t%(100000)==0):

         # each x seconds (simualation time) the file is saved

         file = open('testfile.txt','w')

**for** t **in** range (0,500):

            file.write(str(temp\_axis[t]))

            file.write(',')

The actual script:

"""

@author: Bart Bozon

Script for 2d temperature profile, optimised for speed

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

**import** copy

# functions to calculate the delta T difference between 2 points

**def** delta\_2d\_left (matrix):

    matrixleft=np.roll(matrix,1,1)

    delta =(matrixleft -matrix)

**return** delta

**def** delta\_2d\_right (matrix):

    matrixright=np.roll(matrix,-1,1)

    delta =(matrixright -matrix)

**return** delta

**def** delta\_2d\_down (matrix):

    matrixright=np.roll(matrix,-1,0)

    delta =(matrixright -matrix)

**return** delta

**def** delta\_2d\_up (matrix):

    matrixright=np.roll(matrix,1,0)

    delta =(matrixright -matrix)

**return** delta

# functions to calculate mixed value of k

**def** determine\_mixed\_k\_left (matrix):

    matrixleft=np.roll(matrix,1,1)

    mixed\_k =2\*(matrixleft \*matrix)/(matrixleft+matrix)

**return** mixed\_k

**def** determine\_mixed\_k\_up (matrix):

    matrixleft=np.roll(matrix,1,0)

    mixed\_k =2\*(matrixleft \*matrix)/(matrixleft+matrix)

**return** mixed\_k

**def** determine\_mixed\_k\_right (matrix):

    matrixleft=np.roll(matrix,-1,1)

    mixed\_k =2\*(matrixleft \*matrix)/(matrixleft+matrix)

**return** mixed\_k

**def** determine\_mixed\_k\_down (matrix):

    matrixleft=np.roll(matrix,-1,0)

    mixed\_k =2\*(matrixleft \*matrix)/(matrixleft+matrix)

**return** mixed\_k

# Set up variables& constants

arraysizey=100

arraysizex=200

temp\_old =np.ones((arraysizey,arraysizex))

temp\_new =np.ones((arraysizey,arraysizex))

temp\_old=20\*temp\_old

material\_matrix=np.ones((arraysizey,arraysizex))

k =np.ones((arraysizey,arraysizex))

rho =np.ones((arraysizey,arraysizex))

c =np.ones((arraysizey,arraysizex))

pnetto = np.zeros((arraysizey,arraysizex))

qflow = np.zeros((arraysizey,arraysizex))

dx=0.001 # 1 mm grid, 150 points, = 0.15 m

dt=0.001

# defining material constants. We will use a dict for this

aluminium = {

    'rho': 2700,

    'k': 237,

    'c': 897,

}

stainless\_steel={

    'rho': 7500,

    'k': 14.4,

    'c': 502,

}

teflon={

    'rho': 2200,

    'k': 0.25,

    'c': 1000,

}

# we need a link between a number and the name of the material

material\_list={

        1:aluminium,

        2:stainless\_steel,

        3:teflon

}

# here we define the composition of our rectangle.

material\_matrix[0:arraysizey-1,0:arraysizex-1]=2

material\_matrix[10:15,10:80]=1

material\_matrix[20:25,10:80]=1

material\_matrix[30:35,10:80]=1

material\_matrix[40:45,10:80]=1

material\_matrix[50:55,10:80]=1

material\_matrix[20:35,110:180]=1

material\_matrix[40:75,30:60]=3

material\_matrix[10:25,140:150]=3

material\_matrix[30:50,140:150]=3

material\_matrix[15:20,160:arraysizex-1]=3

material\_matrix[0:20,160:165]=3

# we'll add a power source

pnetto [10:50,90:100]=2000000

# display of the material

plt.figure(figsize = (16,4))

im=plt.imshow(material\_matrix)

plt.colorbar(im)

plt.show()

# copy all the values of the chosen materials in the different matrices y,x

**for** x **in** range (0,arraysizex):

**for** y **in** range (0,arraysizey):

        k[y,x]=material\_list[material\_matrix[y,x]]['k']

        rho[y,x]=material\_list[material\_matrix[y,x]]['rho']

        c[y,x]=material\_list[material\_matrix[y,x]]['c']

# determine the mixed k

k\_left=determine\_mixed\_k\_left(k)

k\_right=determine\_mixed\_k\_right(k)

k\_up=determine\_mixed\_k\_up(k)

k\_down=determine\_mixed\_k\_down(k)

# this is used for the graph at the end

steps\_to\_show = 500

temp\_points=np.zeros((11,steps\_to\_show+1))

temp\_axis =np.zeros(steps\_to\_show+1)

temp\_teller=0

# Run simulation.

**for** t **in** range (0,1000001):

**if** t>500001 :

        pnetto=0\*pnetto

    # we force part of the material to 30 c

    temp\_old[80:85,170:180]=30

    # we force part of the material to 10 c

    temp\_old[80:85,15:25]=10

    # this is the actual difference equation

    temp\_new=temp\_old+pnetto\*dt/(rho\*c)+dt/(dx\*dx\*rho\*c)\*(k\_left\*delta\_2d\_left(temp\_old)+k\_right\*delta\_2d\_right(temp\_old)+k\_up\*delta\_2d\_up(temp\_old)+k\_down\*delta\_2d\_down(temp\_old))

    # we will calculate the q flow as well. This is only for display purposes

**if** (t%(10000)==0):

     qflow=abs(1/(dx\*dx\*rho\*c)\*(k\_left\*delta\_2d\_left(temp\_old)+k\_up\*delta\_2d\_up(temp\_old)))

    # the boundaries are incorrectly calculated (due to the matrix roll over). We force them zero

    qflow [0:arraysizey,0]=0

    qflow [0:arraysizey,arraysizex-1]=0

    qflow [0,0:arraysizex]=0

    qflow [arraysizey-1,0:arraysizex]=0

    # in princple our boundaries are floating (perfectly isolated)

    temp\_new[0:arraysizey-1,0]=temp\_new[0:arraysizey-1,1]

    temp\_new[0:arraysizey-1,arraysizex-1]=temp\_new[0:arraysizey-1,arraysizex-2]

    temp\_new[0,0:arraysizex-1]=temp\_new[1,0:arraysizex-1]

    temp\_new[arraysizey-1,0:arraysizex-1]=temp\_new[arraysizey-2,0:arraysizex-1]

    # This boundary is kept at a constant temperature, we force it to 15c

    temp\_new[0:arraysizey-1,0]=15

    # for this boundary we assume it radiates energy into space

    temp\_new[0:50,arraysizex-1]=temp\_new[0:50,arraysizex-1]-5.67e-8\*dt\*(temp\_new[0:50,arraysizex-1]+273.15)\*\*4

    # the plots are shown each x seconds.

**if** (t%(10000)==0):

        # display results.

        plt.figure(figsize = (16,4))

        im=plt.imshow(temp\_new,cmap="gist\_ncar")

        # see https://matplotlib.org/examples/color/colormaps\_reference.html

        plt.colorbar(im)

        plt.show()

        plt.figure(figsize = (16,4))

        im2=plt.imshow(qflow,cmap="hot")

        # see https://matplotlib.org/examples/color/colormaps\_reference.html

        plt.colorbar(im2)

        plt.show()

**print** ("t=",t\*dt)

        # these points are used to make the graph

        temp\_points[0,temp\_teller]=temp\_new[40,10]

        temp\_points[1,temp\_teller]=temp\_new[40,30]

        temp\_points[2,temp\_teller]=temp\_new[40,50]

        temp\_points[3,temp\_teller]=temp\_new[40,70]

        temp\_points[4,temp\_teller]=temp\_new[40,90]

        temp\_points[5,temp\_teller]=temp\_new[80,25]

        temp\_points[6,temp\_teller]=temp\_new[80,50]

        temp\_points[7,temp\_teller]=temp\_new[80,75]

        temp\_axis[temp\_teller]=t\*dt

        temp\_teller=temp\_teller+1

        plt.plot(temp\_axis,temp\_points[0])

        plt.plot(temp\_axis,temp\_points[1])

        plt.plot(temp\_axis,temp\_points[2])

        plt.plot(temp\_axis,temp\_points[3])

        plt.plot(temp\_axis,temp\_points[4])

        plt.plot(temp\_axis,temp\_points[5])

        plt.plot(temp\_axis,temp\_points[6])

        plt.plot(temp\_axis,temp\_points[7])

        plt.show()

**if** (t%(100000)==0):

         # each x seconds (simualation time) the file is saved

         file = open('testfile.txt','w')

**for** t **in** range (0,500):

            file.write(str(temp\_axis[t]))

            file.write(',')

            file.write(str(temp\_points[0][t]))

            file.write(',')

            file.write(str(temp\_points[1][t]))

            file.write(',')

            file.write(str(temp\_points[2][t]))

            file.write(',')

            file.write(str(temp\_points[3][t]))

            file.write(',')

            file.write(str(temp\_points[4][t]))

            file.write(',')

            file.write(str(temp\_points[5][t]))

            file.write(',')

            file.write(str(temp\_points[6][t]))

            file.write(',')

            file.write(str(temp\_points[7][t]))

            file.write(',')

            file.write(str(temp\_points[8][t]))

            file.write(',')

            file.write(str(temp\_points[9][t]))

            file.write(',')

            file.write(str(temp\_points[10][t]))

            file.write('\n')

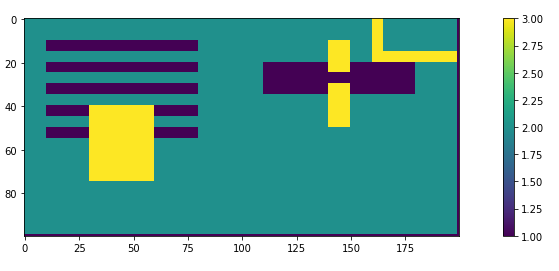
         file.close()

    temp\_old=copy.deepcopy(temp\_new)

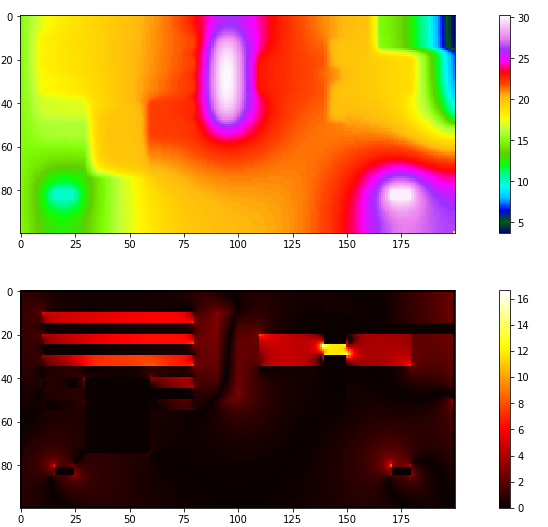
Script 4.9: Code for the simulation of a complex 2D case.

The result of the simulation:

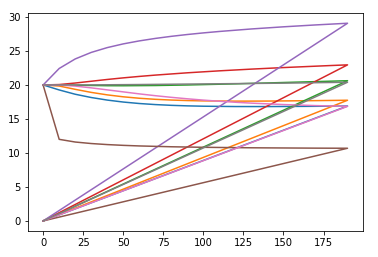
The material composition:



The temperature and heat flow:



The temperature vs time graph:



## Taking advantage of rotational symmetry with cylindrical coordinates

To model the temperature profile of a three dimensional systems it is in principle necessary to include all three spatial dimensions in the heat equation, which we wrote in in its most general form in (4.35). However, in some cases it is possible to use the symmetry of the situation to reduce the complexity. Perhaps the most basic and also a common case is cylindrical symmetry. A system that has an axis around which any rotation leaves the system unchanged has cylindrical symmetry. Hence the heating of a cylindrical rod is such a case, as long as there are no variations in heating sources or material changes when going around the axis of the cylinder that should be taken into account.

The relation between the standard (so-called Cartesian) coordinates and cylindrical coordinates is depicted in Figure 4.11. From this figure, it can be seen that these coordinate systems are related as

|  |  |
| --- | --- |
|  | (4.37) |

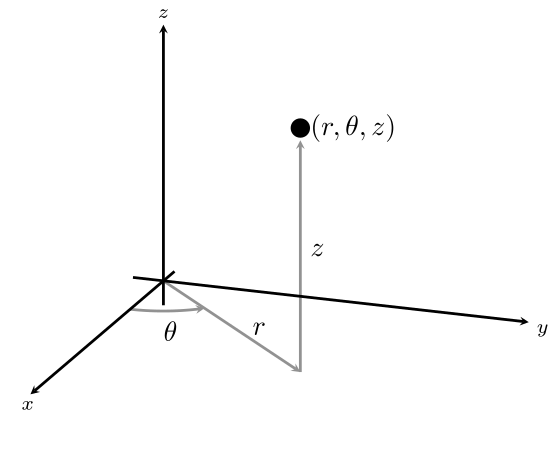


Figure 4.11: Depiction of the relation between cylindrical and Cartesian coordinates.

Through some math it can be shown that in these coordinates the heat equation (4.35) can be written as

|  |  |
| --- | --- |
|  | (4.38) |

(See for example (Griffiths, 2005)). For compactness, the possible coordinate dependencies of , , and are not displayed here. This hardly seems like a simplification. However, when the system under study has cylindrical symmetry, nothing will vary along the direction of , that is what cylindrical symmetry means. Hence,    and the equation reduces to

|  |  |
| --- | --- |
|  | (4.39) |

This is still not a simple equation, but the fact that we got rid of one entire coordinate dependence, that we reduced the problem from a 3-dimensional to a 2-dimensional problem, is a huge advantage.

# Linear system theory

In this chapter we take another perspective on the mathematical modelling of physical systems. This requires some knowledge of the methods and notation of linear algebra and matrices, which can be found in appendix D. This chapter will have a relatively high focus on the “analysis” (mathematical) subsection of each section.

## The state-space form for linear systems

In this section we will put a linear differential equation in its state-space form. (Prior required knowledge is paragraphs D.1 and D.2 from the appendix.) Concepts like “state”, “equilibrium state” and “coordinate transformations” will be introduced.

### Physics: the mass-spring system

Vibrations, or oscillations, are a very fundamental part of the universe. The most everyday example is a mass on a spring, as depicted in

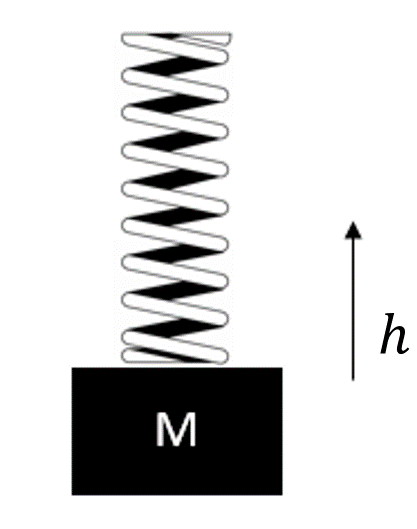


Figure 5.1: A mass hanging from a spring.

In the case of an ideal spring the force is proportional and in the opposite direction to the height displacement, , from the equilibrium position of the spring. Mathematically

|  |  |
| --- | --- |
|  | (5.1) |

where is the spring constant. In addition, there is the gravitational force, which we take as in (3.2). Newton’s second law then becomes

|  |  |
| --- | --- |
|  | (5.2) |

How to tackle such a differential equation is discussed in the Appendix C, section C.2. One way to write the solution is

|  |  |
| --- | --- |
|  | (5.3) |

This describes the oscillation of the mass spring system around the equilibrium position of the mass-spring system . The constants and determine the amplitude and the initial position and could be fixed by some initial position and velocity. The constant is the radial frequency and is related to the mass and the spring constant , as you will show in the exercise.

**Exercise 5.1.** Show that (5.3) is indeed a solution and determine the expression for in terms of and .

**Exercise 5.2.** A mass of 1kg is fixed to a spring with a spring constant 10 N/m. The mass is released from rest at a distance of 50 cm from its equilibrium. Determine the position and velocity of the mass 10 seconds after it is released.

### Modeling: the state-space form

When dealing with dynamic systems, we could work with differential equations like the one from equation (5.2). In practice, it is easier to write our system differently: we want to put our system in the *state-space form*. To do so, there are various steps that we should take.

**Step 1: find the state vector**

We first must find state vector that fully describes the state of our system. That is, if we know the state vector , then we know exactly what is currently going on in our system. Given this state and the system model, we could then also predict everything that will happen in the future in our system.

**Exercise 5.3.** If you know the height , do you then have sufficient data to predict the future motion of the mass? Or does this motion also depend on something else?

In this case, the state of our system is given by , where is the velocity of the mass. (Officially there are multiple options for the state vector, but this is the easiest one.) More generally, if our differential equation has a parameter and its ’th derivative in it, then the state *must* contain the parameters , but *not* .

**Step 2: write the system in state-space form**

Now that we know the state, we want to write the dynamics of the system in the form

|  |  |
| --- | --- |
|  | (5.4) |

where is some vector-valued function that only depends on the parameters from the state . If this notation confuses you, you can read up on vectors in Appendix D and on vector functions in E.

For our example system, the state equals . Its time-derivative hence equals

|  |  |
| --- | --- |
|  | (5.5) |

Our goal now is to find some expression

|  |  |
| --- | --- |
|  | (5.6) |

where on the dots there should be expressions containing the state-parameters and . Expressing in the state-parameters and is still easy here, because equals . We may hence write

|  |  |
| --- | --- |
|  | (5.7) |

The second part is more difficult. We want to express (or equivalently ) in only and . To do so, we must use our differential equation (5.2). We then find

|  |  |
| --- | --- |
|  | (5.8) |

We have now put our system in state-space form!

An extra step that we often do is simplify the notation. We then write the state vector as . In other words, we define the first state parameter as the height , and the second state parameter as the velocity . We can then write

|  |  |
| --- | --- |
|  | (5.9) |

This is our mass-spring system in state-space notation . If you feed this to a simulation package, it can easily run the simulation for you.

**Step 3: check for linearity**

If we look at equation (5.9), we can notice something interesting: the function (the right-hand side) depends *linearly* on the state . That is, there are only terms of the form with some constant and some state parameter. (An extra constant like is also allowed.) When this is all the case, we call the system *linear*. In any other case the system is *nonlinear*.

The advantage of linear systems is that we can write them in an even more useful form: the matrix form

|  |  |
| --- | --- |
|  | (5.10) |

Here, the matrix and the vector are constant. That is, they do not depend on the state in any way. For our example system we can write

|  |  |
| --- | --- |
|  | (5.11) |

This is the linear system in its state-space representation.

**Exercise 5.4.** Write down the matrix and the vector for this system. That is, extract them from (5.11).

### Analysis: the equilibrium state and deviations from it

We continue by analysing the system. As usual, we start by finding the equilibrium state . Per definition, if the system starts in this state, then the state remains constant. So it is the state for which , or equivalently for which .

**Exercise 5.5.** For the mass-spring system in state-space form (5.9), find the equilibrium state . Write it in vector form. Does your answer physically make sense? Subsequently also find using its linear form (5.11) by applying a matrix inverse. Do you get the same answer?

When analysing systems, it is common to only look at deviations from the equilibrium state . So in the case of our mass-spring system, we don’t care about the actual height of the mass, but only about its displacement from its equilibrium.

To make this easier mathematically, we define the *displacement* as . That is, we define and . We now express our whole system in this new displacement state .

**Exercise 5.6.** Write the mass-spring system in the form . That is, find expressions for and , expressing them in and . (Hint: substitute and .)

**Exercise 5.7.** For this new system description, find the equilibrium state. Explain the result you find.

**Exercise 5.8.** Also write the system in the form . What is ? Is this always the case?

### Simulation: simulating a system in state-space form

"""

@author: Bart Bozon

Spring in state space form with euler

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

# Define settings.

endTime = 10 # The time (seconds) that we simulate.

dt = 0.0001 # The time step (seconds) that we use in the discretization.

g = 9.81 # The gravitational acceleration [m/s^2].

# Set up variables.

time = np.arange(0, endTime + dt, dt) # A list with all times we want to plot at.

vSimulated = np.zeros(len(time)) # A list for the velocity (simulated).

sSimulated = np.zeros(len(time)) # A list for the distance (simulated).

k=10

m=1

x=np.array([0,0])

A=np.array ([[0,1],

             [-k/m,0]])

b=([0,g])

# Run simulation.

**for** i **in** range(1, len(time)):

    x=x+dt\*(A.dot(x)+b)

    sSimulated[i] = x[0]

    vSimulated[i] = x[1]

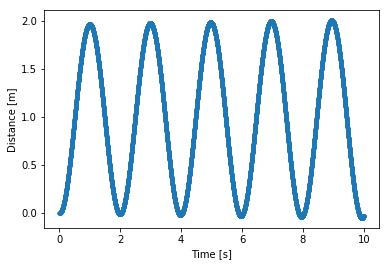
# Display results.

plt.plot(time, sSimulated, marker=".")

plt.xlabel('Time [s]')

plt.ylabel('Distance [m]')

plt.show()



## Stability of continuous-time linear systems

By including friction in our model of the mass-spring system, the question of stability becomes more interesting and we will be able to demonstrate some of the power of the state-space form introduced in paragraph 5.1.2. The math described in the appendix D, section D.3, is prerequisite.

### Physics: adding friction to the mass-spring system

Now we will add friction the mass-spring system. We will assume that the system will not be disturbed not too far away from its equilibrium position, such that the velocity will always be relatively small and there is no turbulence of the surrounding gas or liquid. In this case it is a reasonable approximation to assume that the drag force is proportional to the velocity:[[5]](#footnote-6)

|  |  |
| --- | --- |
| . | (5.12) |

Replacing by and adding this to equation (5.2) results in

|  |  |
| --- | --- |
|  | (5.13) |

as the differential equation for the mass-spring system including friction. How to tackle such a differential equation is discussed in the Appendix C, section C.2.

### Modeling: extending the state-space form

Just like previously, we will go through the various steps of putting equation (5.13) into state space form.

**Exercise 5.9.** Determine a state for this mass-spring-damper system.

**Exercise 5.10.** Write the system in the form .

**Exercise 5.11.** Determine if the system is linear. If it is, write it as. What are and ?

### Analysis: stability of the equilibrium state

As usual, we start our analysis by finding the equilibrium state.

**Exercise 5.12.** Determine the equilibrium state of the mass-spring-damper system.

**Exercise 5.13.** Write the mass-spring-damper system, through the displacement , in the form .

We have now written our system in the form . This is very powerful, as there are many mathematical techniques to analytically calculate how the state displacement evolves over time. One of them is through matrix exponents. If we put the system in an initial displacement , then the displacement at any later time is given by

|  |  |
| --- | --- |
|  | (5.14) |

However, matrix exponents are difficult to calculate. A method that is used more often is the so-called *modal analysis*. We then first find the eigenvalues of the matrix . (For background on eigenvalues, see Appendix D, Section D.7.) These eigenvalues have corresponding eigenvectors . It can then be mathematically proven (we will not do so) that we can write

|  |  |
| --- | --- |
|  | (5.15) |

The constants here depend on the initial state that we put our system in. Each of the terms in the above expression are then called *modes* of our system.

**Exercise 5.14.** For the mass-spring-damper system, take , and . Find the two eigenvalues of . Find the corresponding eigenvectors. Then, determine the response for an initial state .

It may happen that some of the eigenvalues are complex numbers, like . (For a crash course on complex numbers, see Appendix B.) In that case, the above expression will also include sine and cosine terms: we will have oscillations. The specific terms that get added will include and . So the imaginary part of the complex number will specify the frequency of the oscillation, while the real part determines whether this oscillation grows (if ) or damps out (if ). We will not discuss in detail how exactly we can then find the response , as there are many mathematical intricacies involved that are not relevant for us.

For us, the most important thing to determine is whether the state displacement eventually goes to zero as grows, or whether diverges and becomes infinitely large. To figure that out, we have to look at the eigenvalues . If there is any eigenvalue that is positive (like ) or has a positive real part (like ), then will grow infinitely large. After all, the solution will have a term that will grow to infinity as . The system is then said to be *unstable*. If, however, *all* eigenvalues are negative (like ) or have negative real parts (like ) then the system is *stable*.

You may note that the eigenvalues here fulfil exactly the same role as the *characteristic roots* of a differential equations. (See Appendix C, Section C.3.2.) As a result, we can set up the following general rules, identical to those for characteristic roots.

* If *all* eigenvalues have a negative real part, then the equilibrium state is *stable*. We have .
* If there is *any* eigenvalue with a positive real part, then the equilibrium state is *unstable*. Alternatively, if there are two or more identical eigenvalues with a zero real part, then it is also unstable. .
* In any other case, the equilibrium state is marginally unstable. This is the case when some eigenvalues have a zero real part, but these eigenvalues are all different. Other eigenvalues may still have a negative real part. In this case may converge to a constant non-zero value or continue to oscillate forever.

These rules may initially be a bit confusing. It helps to visualize them. If you draw the eigenvalues of your system in Figure 5.2, you immediately see whether all eigenvalues are safe, or if there’s one eigenvalue destabilizing your system.

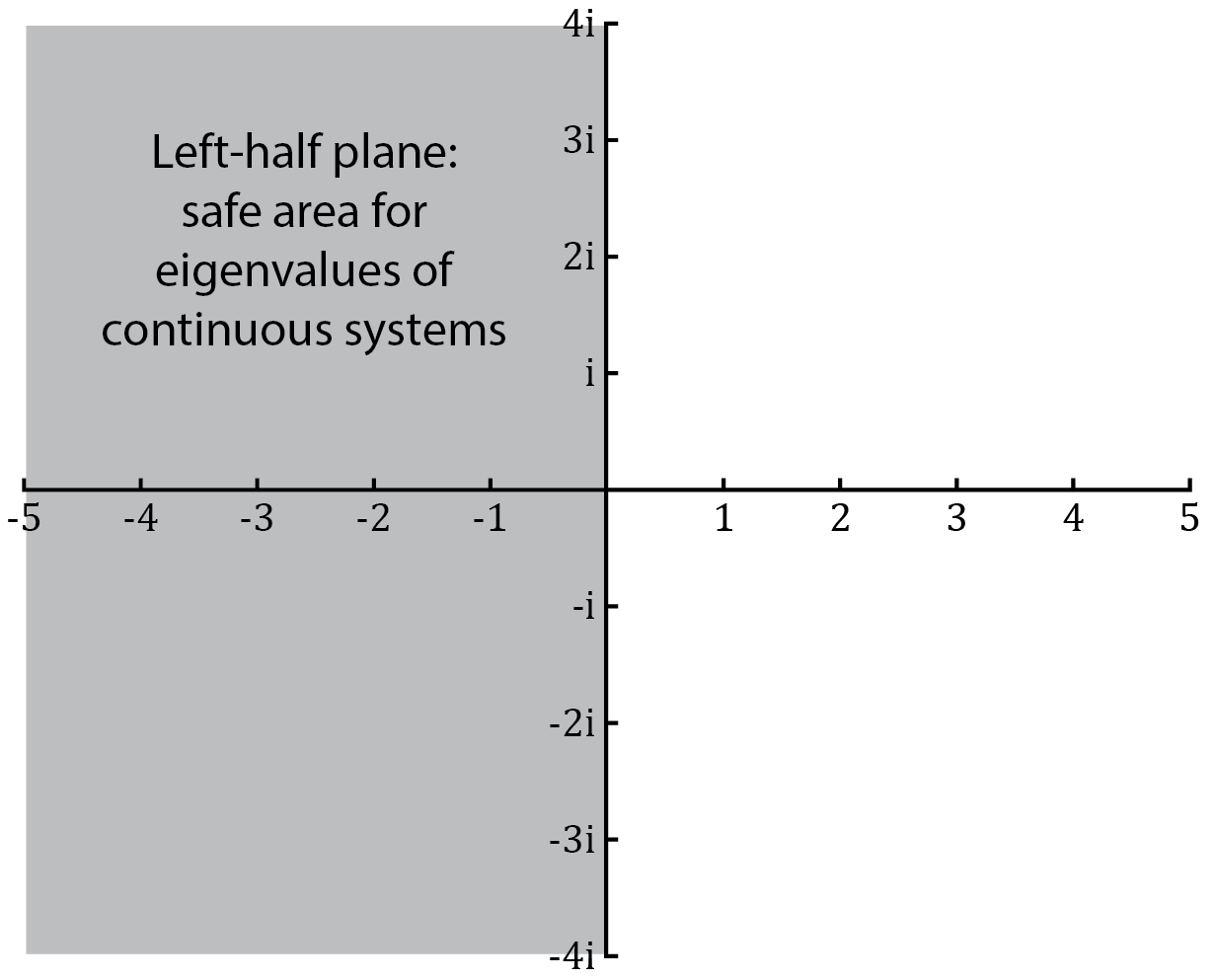


Figure 5.2: For continuous-time systems, all eigenvalues must be in the left-half plane to ensure stability.

**Exercise 5.15.** If an equilibrium state has eigenvalues , what can we say about its stability?

**Exercise 5.16.** If an equilibrium state has eigenvalues , what can we say about its stability?

**Exercise 5.17.** If an equilibrium state has eigenvalues , what can we say about its stability?

**Exercise 5.18.** If an equilibrium state has eigenvalues , what can we say about its stability?

**Exercise 5.19.** If an equilibrium state has eigenvalues , what can we say about its stability?

We can apply these rules to our mass-spring-damper system.

**Exercise 5.20.** Use values , and . Find the eigenvalues of the matrix . What can we say about the stability of the system? Does this make sense?

**Exercise 5.21.** Use values , and . Find the eigenvalues of the matrix . What can we say about the stability of the system? Does this make sense?

**Exercise 5.22.** For generic values of , and , calculate the eigenvalues of the matrix , expressing them in , and . Verify your expression by comparing it with the previous two exercises. Assume that . What conditions must hold for and to ensure stability? And when do we have marginal stability? Explain what happens physically in each of these cases.

### Simulation: comparison of system response with eigenvalues

Below a script is given in which the solutions are found and compared with a euler simulation.

"""

@author: Bart Bozon

Spring in state space form. Solved with euler and state space

"""

**import** numpy as np

**from** numpy **import** linalg as LA

**import** matplotlib.pyplot as plt

**import** cmath

**from** numpy.linalg **import** inv

# Define settings.

endTime = 10 # The time (seconds) that we simulate.

dt = 0.0001 # The time step (seconds) that we use in the discretization.

g = 9.81 # The gravitational acceleration [m/s^2].

# Set up variables.

time = np.arange(0, endTime + dt, dt) # A list with all times we want to plot at.

vSimulated = np.zeros(len(time)) # A list for the velocity (simulated).

sSimulated = np.zeros(len(time)) # A list for the distance (simulated).

Resultsreal = np.zeros((2,len(time)),dtype=np.complex\_)

k=10

m=1

xstart=np.array([1,0])

x=xstart

A=np.array ([[0,1],

             [-k/m,0]])

b=([0,0])

w,v=(LA.eig(A))

**print** ("========   STATE SPACE =========")

**print** ("The eigen values are:")

**print** (w)

**print**("The eigen vectors are:")

**print** (v[:,0])

**print** (v[:,1])

**print** ("================================")

# Use differential equations (in matrix form) to find the solution (and compare with state space)

**for** i **in** range(0, len(time)):

    sSimulated[i] = x[0]

    vSimulated[i] = x[1]

    x=x+dt\*(A.dot(x)+b)

# Display results.

plt.plot(time, sSimulated, marker=".")

plt.title('Position, simulated with euler', loc='left')

plt.xlabel('Time [s]')

plt.ylabel('Distance [m]')

plt.show()

plt.plot(time, vSimulated, marker=".")

plt.title('Velocity, simulated with euler', loc='left')

plt.xlabel('Time [s]')

plt.ylabel('Velocity [ms-1]')

plt.show()

# use state space to find solution

vinv=inv (v.transpose())

c=xstart.dot(vinv)

**for** i **in** range(0, len(time)):

    Resultsreal[0,i]=c[0]\*cmath.exp(w[0]\*i\*dt)\*v[0,0]+c[1]\*cmath.exp(w[1]\*i\*dt)\*v[0,1]

    Resultsreal[1,i]=c[0]\*cmath.exp(w[0]\*i\*dt)\*v[1,0]+c[1]\*cmath.exp(w[1]\*i\*dt)\*v[1,1]

plt.plot(time, Resultsreal[0].real, marker=".")

plt.title('Position, calculated with state space', loc='left')

plt.xlabel('Time [s]')

plt.ylabel('Distance [m]')

plt.show()

plt.plot(time, Resultsreal[1].real, marker=".")

plt.title('Velocity, calculated with state space', loc='left')

plt.xlabel('Time [s]')

plt.ylabel('Velocity [ms-1]')

plt.show()

**print** ('Strech the printing area to enable correct formating of below equations')

**print**('The exact solutions for the deviations from the equilibrium state is given by :')

**print**()

**print** ('                        ({0.real:.2f} + {0.imag:.2f}i)'.format(w[0]),'                                      ({0.real:.2f} + {0.imag:.2f}i)'.format(w[1]))

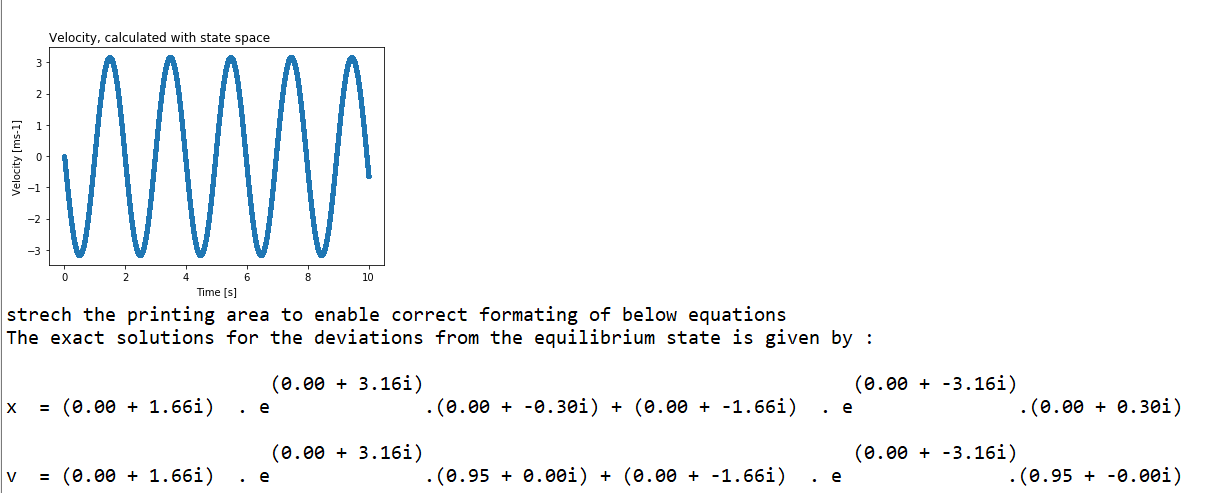
**print** ('x  = ({0.real:.2f} + {0.imag:.2f}i)'.format(c[0]),' . e              .({0.real:.2f} + {0.imag:.2f}i)'.format(v[0,0]),'+ ({0.real:.2f} + {0.imag:.2f}i)'.format(c[1]),' . e              ','.({0.real:.2f} + {0.imag:.2f}i)'.format(v[0,1]))

**print**()

**print** ('                        ({0.real:.2f} + {0.imag:.2f}i)'.format(w[0]),'                                      ({0.real:.2f} + {0.imag:.2f}i)'.format(w[1]))

**print** ('v  = ({0.real:.2f} + {0.imag:.2f}i)'.format(c[0]),' . e              .({0.real:.2f} + {0.imag:.2f}i)'.format(v[1,0]),'+ ({0.real:.2f} + {0.imag:.2f}i)'.format(c[1]),' . e              ','.({0.real:.2f} + {0.imag:.2f}i)'.format(v[1,1]))

Part of the output:



## Stability of discrete-time linear systems

We just studied the stability of continuous-time systems. In Chapter 3 we saw that systems can be either continuous or discrete. So how does stability work for discrete-time systems? That’s what we’ll figure out next.

### Biology: populations of rabbits and foxes

In this section we will set up a simplified predator/prey system. A more realistic description is given by the Lotka-Volterra equations, but these are not linear, which is what we still want for now. The logic of our system is assumed to be as follows.

* Every month a rabbit gets 1.2 child (ignoring gender). The rabbit also stays alive, so without foxes the population more than doubles.
* Every month a fox eats twelve rabbits.
* Every month 20% of the foxes die (with or without rabbits).
* The more rabbits there are, the more young foxes are born. After all, then it’s easier to hunt them, so only the male needs to hunt while the female can bear children. For every fifty extra rabbits, another fox is born.

We should find the equations

|  |  |
| --- | --- |
|  | (5.16) |

We will study this discrete-time linear system of equations in more detail in next sections.

### Modeling: the discrete state-space form

A convenient mathematical way to model this system is through a matrix expression. To set this up, we define the state as

|  |  |
| --- | --- |
|  | (5.17) |

We can now model the system through a single matrix expression

|  |  |
| --- | --- |
|  | (5.18) |

If we know the matrix , then this equation captures our entire system dynamics.

**Exercise 5.23.** Determine the matrix for our example system. Use equation (5.16).

### Analysis: stability of the equilibrium position

In the previous section we found that our model of the rabbit/fox population problem can be written in the form . To analyse the stability, we must look at the eigenvalues of the matrix . For our particular matrix , the eigenvalues are with eigenvector and with eigenvector . (Yes, we get round numbers as eigenvalues. That is definitely not always the case, but is due to conveniently chosen -matrices.) It can now be shown that the analytical solution to our population problem equals

|  |  |
| --- | --- |
|  | (5.19) |

The coefficients and depend on the initial conditions of the system.

**Exercise 5.24.** Assume that we start with rabbits and foxes. What are and then? What will happen with the population of the rabbits and the foxes?

**Exercise 5.25.** Assume that we start with rabbits and foxes. What are and then? What will happen with the population of the rabbits and the foxes?

**Exercise 5.26.** Assume that we start with rabbits and foxes. What are and then? What will happen with the population of the rabbits and the foxes?

**Exercise 5.27.** Assume that we start with rabbits and foxes. What are and then? What will happen with the population of the rabbits and the foxes?

**Exercise 5.28.** Assume that we start with rabbits and foxes. What are and then? What will happen with the population of the rabbits and the foxes?

If we modify our -matrix, we will get different system dynamics. We could for instance adjust it to

|  |  |
| --- | --- |
|  | (5.20) |

In this case our eigenvalues will be and . This means that there will be oscillations in the populations of our rabbits and foxes. Alternatively, we could use the slightly different

|  |  |
| --- | --- |
|  | (5.21) |

which results in eigenvalues and . Once more there will be oscillations. However, as you will see in the simulation section, the first of these two -matrices causes our population to grow. (We have instability.) The second, however, will eventually result in a complete extinction of both rabbits and foxes: we have stability.

To see why this is the case, we have to look at the rules for stability of discrete-time systems. These require us to know the eigenvalues of .

* If *all* eigenvalues have a magnitude smaller than one, then the system state is *stable*.
* If there is *any* eigenvalue with a magnitude larger than one, then the system is *unstable*.
* In any other case, the system is marginally unstable. This is the case when some eigenvalues have a magnitude of exactly one, while other eigenvalues have a magnitude smaller than one. In this case may converge to a constant non-zero value () or continue to oscillate forever ().

These requirements are visualized in Figure 5.3.

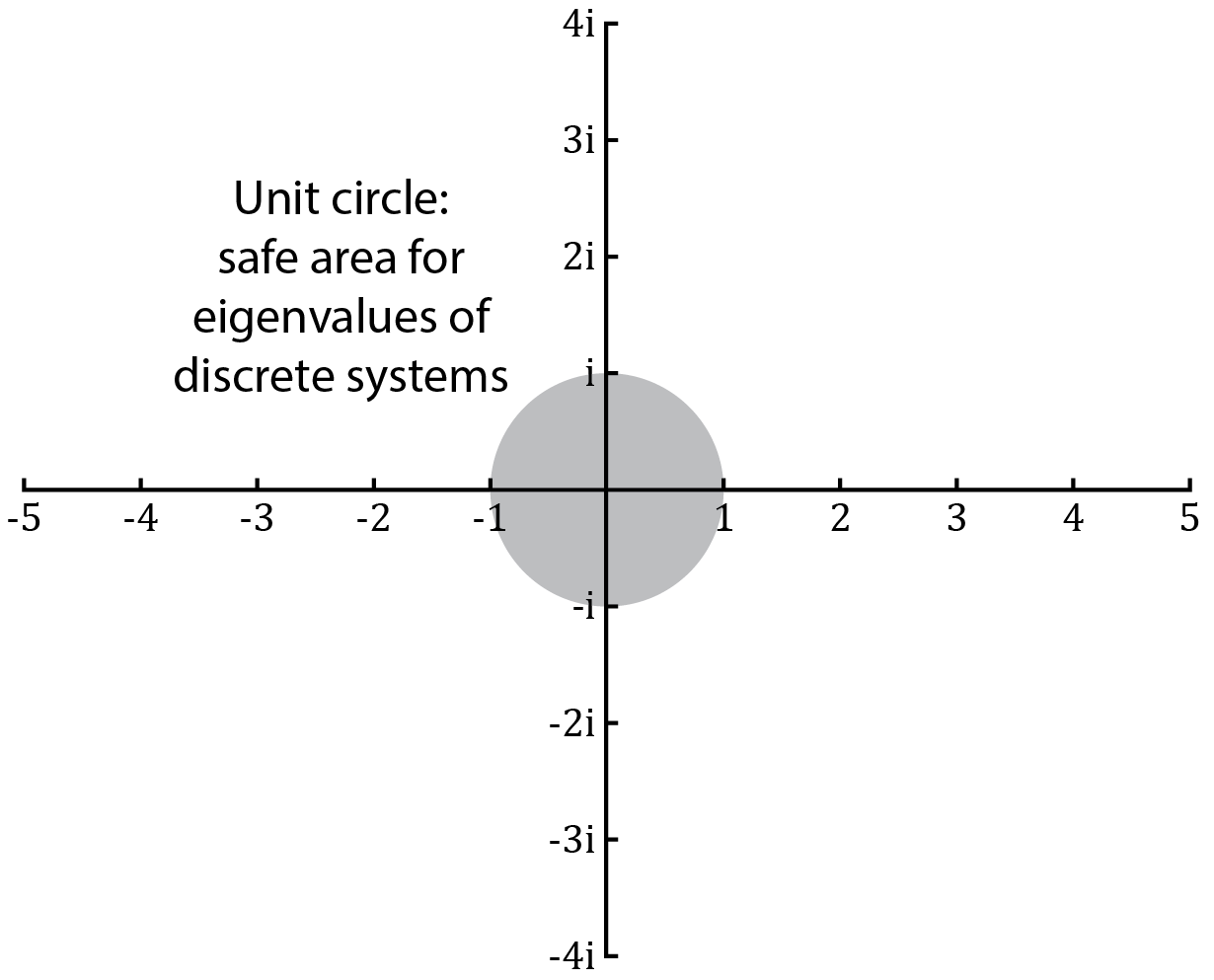


Figure 5.3: For discrete-time systems, to ensure stability, all eigenvalues must be in the unit circle: their magnitude must be less than 1.

**Exercise 5.29.** Check, for the matrices from (5.20) and (5.21) whether the resulting system is stable or not.

**Exercise 5.30.** For the matrix

|  |  |
| --- | --- |
|  | (5.22) |

check whether the system is stable or unstable. That is, find the eigenvalues and apply the above rules. Also find the corresponding eigenvectors. Assuming that we start with rabbits and foxes, what will happen with the population of the rabbits and the foxes?

### Simulation: comparison of the system response with eigenvalues

The previous scripts can be used to find eigenvalues and compare with the actual system.

## The state-space form for nonlinear systems

In this section we will study a nonlinear system. We will put it in state-space form and then linearize it. This enables us to determine the equilibrium states.

### Physics: the pendulum

The simple pendulum is a mass connected with a stiff rod to a rotation point, as depicted in Figure 5.4. We assume that the mass of the rod can be neglected, that there is no friction and that the pendulum is on earth. Then the only force present is the gravitational force acting vertically down on the mass at the end of the rod.

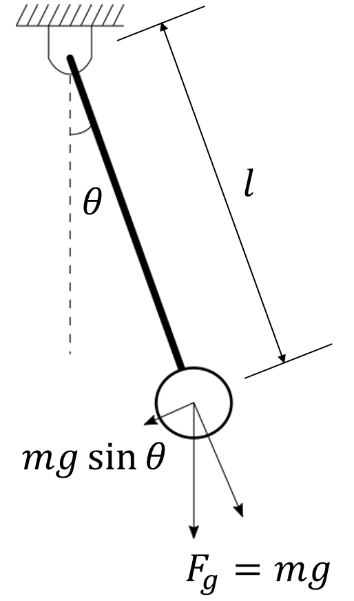


Figure 5.4: Drawing of the pendulum depicting the variables and parameters of importance.

At a displacement angle of the rod from its vertical position, there will be a restoring force, equal to , pushing it back towards the vertical position. The mass describes a path that is the arc of a circle. We can define coordinate along this path, which assigns zero to the vertical position and the displacement along the arc to all other position, positive to the right, negative to the left. Newton’s Second Law then tells us that

|  |  |
| --- | --- |
|  | (5.23) |

The minus sign is added because for positive the restoring force is in the direction of negative . Since the length of the rod is fixed, this simplifies to

|  |  |
| --- | --- |
|  | (5.24) |

### Modeling: the nonlinear state-space form

As usual, we turn the above differential equation into state-space form. For this, we need to determine the state. Since the highest derivative is , and since is also present, the state will be

|  |  |
| --- | --- |
|  | (0.25) |

So it contains the angle and the angular velocity . Alternatively, we could write this in the default notation

|  |  |
| --- | --- |
|  | (5.26) |

where and . Subsequently, we should find the function such that . Similar to how we found equation (5.9), we can now determine this as

|  |  |
| --- | --- |
|  | (5.27) |

We should also check if this is linear. If so, we can write it as . However, with a term this is obviously not the case. We can hence directly move on to the analysis phase.

### Analysis: multiple equilibrium points

To analyze the system, we first find the equilibrium states. Whereas linear systems usually only have *one* equilibrium state , nonlinear systems often have multiple. Not only but also is an equilibrium state.

**Exercise 5.31.** Find *all* equilibrium states for the pendulum. How many equilibrium states are there? What do they physically represent?

Linear systems only had one equilibrium state. This equilibrium state was either stable or unstable (or in special cases marginally stable). As a result, we could say that the whole linear system was either stable or unstable. For nonlinear systems this is different. Nonlinear systems can have multiple equilibrium states, with some being stable and others being unstable. As a result, we cannot say that a nonlinear system is stable or unstable. Only equilibrium states can be stable or unstable.

How do we determine whether an equilibrium state of a nonlinear system is stable or not? That is difficult, since we do not have an matrix to determine the eigenvalues of. However, we can find one by *linearizing* the system. We do so using the methods explained in Appendix E, Section E.3.3. We then linearize about the equilibrium state . The linearization follows as

|  |  |
| --- | --- |
|  | (5.28) |

We will take a look at the various terms in this equation.

**Exercise 5.32.** What is the value of , given that is an equilibrium state?

The Jacobian equals the matrix

|  |  |
| --- | --- |
|  | (5.29) |

Finally, the term is the displacement from the equilibrium state. This means that, after linearizing, we can actually write our system as

|  |  |
| --- | --- |
|  | (5.30) |

where the -matrix equals the Jacobian shown above!

Let’s apply this new theory. Let’s find the first, for any state . From (5.27) we can find

|  |  |
| --- | --- |
|  | (5.31) |

To find , we must insert the equilibrium state that we want to linearize the system about. Let’s first consider the state . Inserting this into the above expression gives us

|  |  |
| --- | --- |
|  | (5.32) |

With the above matrix , our linearized system can be described by the expression . And now that we can write our system in a linear form, we can analyse the stability!

**Exercise** **5.33.** Find the eigenvalues of the -matrix of the linearized system. Is the equilibrium state stable, unstable or marginally stable? Does this correspond to your expectations? Note that and are positive values.

**Exercise 5.34.** For the equilibrium state , find the corresponding -matrix. Once more find the eigenvalues. What can you say about the stability of this equilibrium state? Does this match your expectations?

If we want to be mathematically accurate (which we do), then we should adjust the rules for stability slightly. Suppose that we take a nonlinear system, linearize it, find a matrix such that , and subsequently calculate its eigenvalues, then the following rules hold concerning stability.

* If *all* eigenvalues have a negative real part, then the equilibrium state is *stable*. We have .
* If there is *any* eigenvalue with a positive real part, then the equilibrium state is *unstable*: .
* In any other case, we cannot say anything conclusive about the stability. The reason is that only holds *approximately*. More advanced methods (like Lyapunov methods) are needed to learn more about the stability of the corresponding equilibrium state.

**Exercise 5.35.** Redo exercise 5.33. Can we say anything conclusive about the stability of the equilibrium state using mathematical techniques? Can we guarantee (marginal) stability? And what do the laws of physics (e.g. conservation of energy) say?

### Simulation: simulate nonlinear state-space systems

"""

@author: bart.bozon

A pendulum

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

arraysize =4000

l =1

g =9.81

dt=0.001

theta =np.zeros(arraysize)

theta\_dot =np.zeros(arraysize)

theta\_dot\_dot =np.zeros(arraysize)

theta [0] = 0.1\*3.1415

**for** t1 **in** range (0,arraysize-1):

    theta\_dot\_dot[t1+1] = -g\*l\*np.sin(theta[t1])

    theta\_dot[t1+1] = theta\_dot[t1]+theta\_dot\_dot[t1+1]\*dt

    theta[t1+1]=theta[t1]+theta\_dot[t1+1]\*dt

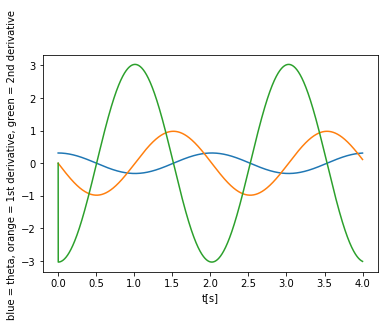
x\_axis = np.arange(0, arraysize\*dt, dt)

plt.plot(x\_axis,theta,x\_axis,theta\_dot,x\_axis,theta\_dot\_dot)

plt.xlabel('t[s]')

plt.ylabel('blue = theta, orange = 1st derivative, green = 2nd derivative ')

plt.show()



## Stability of continuous-time nonlinear systems

In this section we add a bit of complexity to the nonlinear system described in the previous section and apply the state-space method and linearization to study its stability properties.

### Physics: adding friction to the pendulum

Now we will add friction to the simple pendulum in a similar way as was done in section 5.2.1 for the mass spring system. That is, we add a drag force that is proportional and opposite to the velocity of the mass as in equation (5.12). In this case we can write the velocity as . Adding this force to equation of motion for the pendulum, equation (5.23), we obtain

|  |  |
| --- | --- |
|  | (5.33) |

As before, since and is fixed, we can rewrite this as

|  |  |
| --- | --- |
|  | (5.34) |

### Modeling: extending the state-space form

For this adjusted system, we can do exactly the same analysis. Since you are now up-to-speed with all the required methods, we’ll mostly leave that up to you.

**Exercise 5.36.** Put the nonlinear differential equation (5.34) into state-space form .

### Analysis: linearization and stability of the equilibrium positions

**Exercise 5.37.** Find *all* equilibrium states for the pendulum.

**Exercise 5.38.** For each equilibrium state, linearize the system around that equilibrium state and determine its stability. Use the fact that , , and are all positive numbers.

**Exercise 5.39.** Imagine that would be negative. What does this change about the stability of the equilibrium points. What is the corresponding physical meaning? Similarly, imagine that would be negative. What does this change about the stability of the equilibrium points. What is the corresponding meaning?

**Exercise 5.40.** What condition must hold for the parameters , , and to ensure that the eigenvalues are not complex? (That is, to ensure that they are purely real numbers.) Physically, what does this mean?

### Simulation: comparison of nonlinear systems with linearizations

The pendulum script can be compared to the results of the previous scripts (the one with the eigenvalues). Do both and compare the results.

"""

@author: bart.bozon

A pendulum

"""

**import** numpy as np

**import** matplotlib.pyplot as plt

arraysize =4000

l =1

g =9.81

dt=0.001

theta =np.zeros(arraysize)

theta\_dot =np.zeros(arraysize)

theta\_dot\_dot =np.zeros(arraysize)

theta [0] = 0.1\*3.1415

**for** t1 **in** range (0,arraysize-1):

    theta\_dot\_dot[t1+1] = -g\*l\*np.sin(theta[t1])

    theta\_dot[t1+1] = theta\_dot[t1]+theta\_dot\_dot[t1+1]\*dt

    theta[t1+1]=theta[t1]+theta\_dot[t1+1]\*dt

x\_axis = np.arange(0, arraysize\*dt, dt)

plt.plot(x\_axis,theta,x\_axis,theta\_dot,x\_axis,theta\_dot\_dot)

plt.xlabel('t[s]')

plt.ylabel('blue = theta, orange = 1st derivative, green = 2nd derivative ')

plt.show()

# Modelling Event-based systems using Simpy

To model event-based systems (e.g. ques in supermarkets or manufacturing goods in a factory) SimPy can be used. The authors of this reader do not recommend this method if your programming skills are moderate.

See the installation manual on canvas on how to install SimPy. Most of this chapter originated from the website: [https://simpy.readthedocs.io](https://simpy.readthedocs.io/). Large volumes of text are copied from this source, this will be indicated by text in *italics*.

## Introduction

*“SimPy is a process-based discrete-event simulation framework based on standard Python. Processes in SimPy are defined by Python generator functions and may, for example, be used to model active components like customers, vehicles or agents. SimPy also provides various types of shared resources to model limited capacity congestion points (like servers, checkout counters and tunnels). Simulations can be performed “as fast as possible”, in real time (wall clock time) or by manually stepping through the events.”*

A short example simulating two clocks ticking in different time intervals looks like this:

"""

@author: https://simpy.readthedocs.io

simple script to simulate clocks

"""

**import** simpy

**def** clock(env, name, tick):

**while** True:

**print**(name, env.now)

**yield** env.timeout(tick)

env = simpy.Environment()

env.process(clock(env, 'fast', 0.5))

env.process(clock(env, 'slow', 1))

env.run(until=10)

*“SimPy is a discrete-event simulation library. The behavior of active components (like vehicles, customers or messages) is modeled with processes. All processes live in an environment.”*

In the above code the environment is created with this statement:

env = simpy.Environment()

*“ They interact with the environment and with each other via events. Processes are described by simple Python generators. You can call them process function or process method, depending on whether it is a normal function or method of a class. During their lifetime, they create events and yield them in order to wait for them to be triggered.”*

This is the process function (or generator):

**def** clock(env, name, tick):

**while** True:

**print**(name, env.now)

**yield** env.timeout(tick)

Yields means “zwichten, afstaan” in dutch. So the process function “releases” the control to the environment. Behind the yield statement the conditions are given under which the process is triggered again.

*When a process yields an event, the process gets suspended. SimPy resumes the process, when the event occurs (we say that the event is triggered). Multiple processes can wait for the same event. SimPy resumes them in the same order in which they yielded that event. An important event type is the Timeout. Events of this type are triggered after a certain amount of (simulated) time has passed. They allow a process to sleep (or hold its state) for the given time.”*

In the above example the event is triggered if the tick has proceeded the amount as is specified during the creation of the process. In the above example two clocks were created (the fast and the slow) which trigger at different elapsed ticks. We will again illustrate this Timout event with an example, a car.

*" The car will alternately drive and park for a while. When it starts driving (or parking), it will print the current simulation time. The car’s behavior is described in an infinite loop. Remember, this function is a generator. Though it will never terminate, it will pass the control flow back to the simulation once a yield statement is reached. Once the yielded event is triggered (“it occurs”), the simulation will resume the function at this statement. As I said before, our car switches between the states parking and driving. It announces its new state by printing a message and the current simulation time (as returned by the Environment.now property). It then calls the Environment.timeout() factory function to create a Timeout event. This event describes the point in time the car is done parking (or driving, respectively). By yielding the event, it signals the simulation that it wants to wait for the event to occur.”*

""""""

@author: https://simpy.readthedocs.io

simple script to simulate a car

"""

**import** simpy

**def** car(env):

**while** True:

**print**('Start parking at %d' % env.now)

         parking\_duration = 5

**yield** env.timeout(parking\_duration)

**print**('Start driving at %d' % env.now)

         trip\_duration = 2

**yield** env.timeout(trip\_duration)

env = simpy.Environment()

env.process(car(env))

env.run(until=15)

## Process interactions

*“The Process instance that is returned by Environment.process() can be utilized for process interactions. The two most common examples for this are to wait for another process to finish and to interrupt another process while it is waiting for an event.*

*As it happens, a SimPy Process can be used like an event (technically, a process actually is an event). If you yield it, you are resumed once the process has finished. Imagine a car-wash simulation where cars enter the car-wash and wait for the washing process to finish. Or an airport simulation where passengers have to wait until a security check finishes. Lets assume that the car from our last example magically became an electric vehicle. Electric vehicles usually take a lot of time charging their batteries after a trip. They have to wait until their battery is charged before they can start driving again. We can model this with an additional charge() process for our car. Therefore, we refactor our car to be a class with two process methods: run() (which is the original car() process function) and charge().”*

"""

@author: https://simpy.readthedocs.io

simple script to simulate a car, using a class

"""

**import** simpy

**class** Car(object):

**def** \_\_init\_\_(self, env):

         self.env = env

         # Start the run process everytime an instance is created.

         self.action = env.process(self.run())

**def** run(self):

**while** True:

**print**('Start parking and charging at %d' % self.env.now)

             charge\_duration = 5

             # We yield the process that process() returns

             # to wait for it to finish

**yield** self.env.process(self.charge(charge\_duration))

             # The charge process has finished and

             # we can start driving again.

**print**('Start driving at %d' % self.env.now)

             trip\_duration = 2

**yield** self.env.timeout(trip\_duration)

**def** charge(self, duration):

**yield** self.env.timeout(duration)

env = simpy.Environment()

car = Car(env)

env.run(until=15)

*“The run process is automatically started when Car is instantiated. A new charge process is started every time the vehicle starts parking. By yielding the Process instance that Environment.process() returns, the run process starts waiting for it to finish”*

## An elaborate example

In the below code a supermarket is modelled and simulated. The code creates at random times (determined by customer\_creation\_time\_high/low) new customers. They shop for random times (shop\_time\_high/low) after which they try to check out at a cash register. The purpose of the program is to determine the amount of cash registers the supermarket has to apply to keep the rows in check.

"""

@author: Bart Bozon

elaborate script to simulate a supermarket

"""

**import** simpy

**import** random

# the parameters of your supermarket

number\_of\_cash\_registers=5

number\_of\_allowed\_shoppers\_in\_supermarket=20

customer\_creation\_time\_low =5

customer\_creation\_time\_up =20

shop\_time\_low =10

shop\_time\_up =160

cash\_time\_low =10

cash\_time\_up =60

# start of the program

customer\_counter=0

**def** create\_customers(env):

**while** True:

**global** customer\_counter

    customer\_counter=customer\_counter+1

**print**('%3d : customer %d created' %(env.now,customer\_counter))

    customer(env,'cust-'+chr(int(customer\_counter / 10)+48)+chr(customer\_counter%10+48),customer\_counter)

**yield** env.timeout(random.randint(customer\_creation\_time\_low,customer\_creation\_time\_up))

**class** MonitoredResource(simpy.Resource):

**def** \_\_init\_\_(self, \*args, \*\*kwargs):

         super().\_\_init\_\_(\*args, \*\*kwargs)

         self.data = []

**def** request(self, \*args, \*\*kwargs):

         self.data.append((self.\_env.now,1,self.count,  len(self.queue)))

**return** super().request(\*args, \*\*kwargs)

**def** release(self, \*args, \*\*kwargs):

         self.data.append((self.\_env.now,-1,self.count, len(self.queue)))

**return** super().release(\*args, \*\*kwargs)

**class** customer(object):

**def** \_\_init\_\_(self, env,name,number):

         self.env = env

         self.action = env.process(self.run())

         self.name=name

         self.number=number

**def** run(self):

             #while True:

             shop\_duration = random.randint(shop\_time\_low,shop\_time\_up)

**yield** self.env.process(self.shopping\_cap(shop\_duration))

             cash\_register\_duration = random.randint(cash\_time\_low,cash\_time\_up)

**yield** self.env.process(self.cash\_reg(cash\_register\_duration))

**print**('%3d : %s finished at cashregister' % (self.env.now,self.name))

**print**('%3d : %s starts driving home' % (self.env.now,self.name))

             trip\_duration = 2

**yield** self.env.timeout(trip\_duration)

**def** cash\_reg(self, duration):

      # Request one of its charging spots

**print**('%3d : %s wants to pay at cash\_register (time needed to pay %d)' % (env.now,self.name,duration))

      with cash\_register.request() as req:

**yield** req

**print**('%3d : %s is at cashier' % (env.now,self.name))

**yield** env.timeout(duration)

**print**('%3d : %s leaving the cash\_register' % (env.now,self.name))

**def** shopping\_cap(self, duration):

      # Request one of its charging spots

**print**('%3d : %s wants to shop for time %d' % (env.now,self.name,duration))

      with shopping\_capacity.request() as req:

**yield** req

**print**('%3d : %s is shopping' % (env.now,self.name))

**yield** env.timeout(duration)

**print**('%3d : %s is finished with shopping' % (env.now,self.name))

env = simpy.Environment()

shopping\_capacity = MonitoredResource(env, capacity=number\_of\_allowed\_shoppers\_in\_supermarket)

cash\_register = MonitoredResource(env, capacity=number\_of\_cash\_registers)

env.process(create\_customers(env))

# if you want a que of customers at the start of the shopping time

**for** customer\_counter **in** range (3):

    customer(env,'cust-'+chr(int(customer\_counter / 10)+48)+chr(customer\_counter%10+48),customer\_counter)

env.run(until = 500)

**print** ("Shopping")

**print** ("time, request (1) or release (-1), resources occupied, que length")

**for** i **in** range (len(shopping\_capacity.data)):

**print**(shopping\_capacity.data [i])

**print** ()

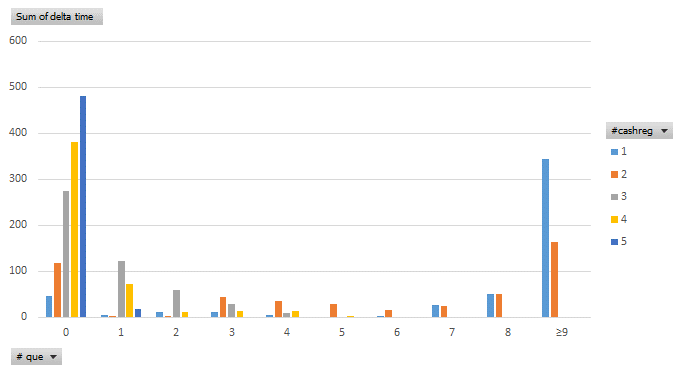
**print** ("Cash register")

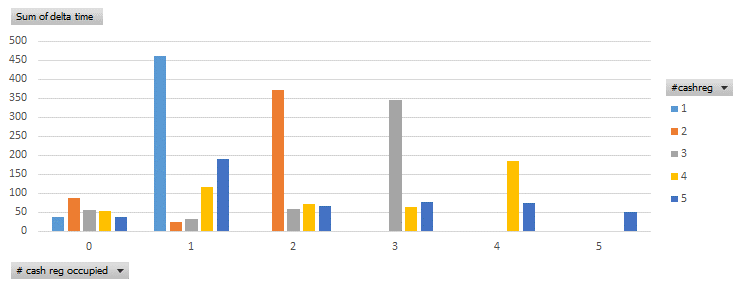
**print** ("time, request (1) or release (-1), resources occupied, que length")

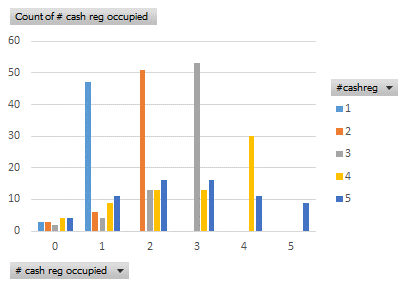
**for** i **in** range (len(cash\_register.data)):

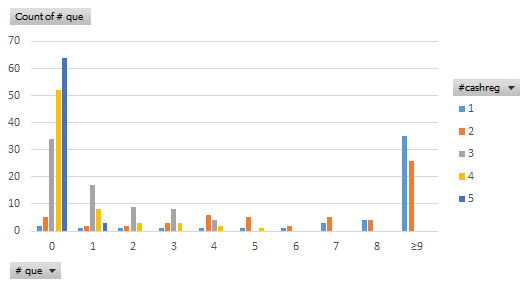
**print**(cash\_register.data[i])

The program outputs array’s of data which can be analysed with excel. In the below graphs the impact of # of cash registers on row length is visualised (using pivot chart in excel).









# References

Borrelli, R. L., & Coleman, C. S. (2004). *Differential Equations: A Modeling Perspective.* John Wiley & Sons.

*Fick's laws of diffusion*. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Fick%27s\_laws\_of\_diffusion#Fick's\_first\_law

Griffiths, D. J. (2005). *Introduction to electrodynamics.*

McComb, W. (1999). *Dynamics and relativity.* Oxford University Press.

philosophicalmath. (2017, October 21). *Phylosophical Math*. Retrieved from https://philosophicalmath.wordpress.com/2017/10/21/terminal-velocity-derivation/

Pritchard, P. J., & Mitchell, J. W. (2016). *Fox and McDonald's Introduction to Fluid Mechanics.* John Wiley & Sons.

van den Akker, H., & Mudde, R. F. (1998). *Fysische transportverschijnselen.* Delft: Delft University Press.

# Basics of Python

## Installing Python

See the “Installation manual python-anaconda” on installing the python environment and the associated libraries.

## Basics of Python

For a basic introduction into python see:

[https://www.codecademy.com/learn/learn-python](https://www.codecademy.com/learn/learn-python-2)

We will use python 3 in this course. Regretfully the python 3 course is not free anymore. You can follow the python 2 course. The main difference between python 2 and 3 is the print statement.

# Complex numbers

When mathematically analyzing systems, it is useful to know what complex numbers are and how you can calculate with them. This appendix gives you a crash course. Alternatively, you can also read Chapter 7 of the Engineering Mathematics book.

## A history of numbers

A long time ago people learned to count. They knew the positive numbers, 1, 2, 3, 4, …. They learned to add them together, discovering that . They also learned the opposite – subtraction – and saw that But while addition was always possible, subtraction had its troubles. Because what is ?

Then one day someone invented the concept of a “negative number”. Perhaps he or she said, “If five minus three is two, then three minus five is *minus* two!” People were a bit negative about it at first: they didn’t like it. “You can’t have minus two sheep!” they said. But eventually they came around, and ever since subtractions always work. You just have to follow the proper rules.

Next, people invented multiplications. They learned that . They also learned the opposite – division – and found that . Multiplications were lovely, since they always worked. Divisions were a different story though, because what is ?

No one knew, until one day someone decided to introduce fractions. “We just leave it written as , and we now also call this a number!” he or she may have said. For a while people were confused. “You can’t have sheep!” they said. But ever since they came around, humanity has been able to calculate with fractions. You just have to follow the proper rules.

Next came the invention of the square. People learned to square numbers, saying . They also figured out the inverse – the square root – and found that . Squaring numbers was great, but square roots were annoying. For example, what is ? It was not possible to write it as a fraction, or as any number people knew at the time.

Then one day a mathematician suggested, “We just leave it written as , or alternatively write it through its decimal expansion .” At first people didn’t like it. “How can a number not be a fraction? We can’t have sheep,” they said. But very diplomatically, to appease them, this creative mathematician called such numbers *irrational numbers* (contrary to fractions, which fall in the realm of *rational numbers*). Eventually the people who thought it was irrational to talk about such things started to accept them, and ever since square roots always work. You just have to follow the proper rules.

Or do they? In reality, the problem wasn’t solved yet. People may have settled on what means, but what is ? No one knew.

Then one day another mathematician suggested, “We can create a new number for it!” As usual, people weren’t happy. “The square root of a negative number doesn’t exist! You can’t have sheep,” they said. But this creative mathematician didn’t give up. “Let’s imagine that they do,” he said. “Let’s call them imaginary numbers! If the square root of four equals two, then the square root of *minus* four equals *imaginary 2*.” Initially people didn’t like it, but eventually they came around and accepted the new numbers.

Ever since, the letter (short for “imaginary”) was used. Per definition, it equals . It directly follows that . Through it, square roots always work. You just have to follow the proper rules.

Later on, computer scientists wanted to be different. Instead of using , they decided to write the imaginary number as . It means the exact same thing though, so don’t let those funny computer scientists confuse you.

## Solving equations with complex numbers

Per definition we have . It follows that , , and so forth. We can even write that . With this knowledge, can you solve the following exercises?

**Exercise B.1.** Find the two solutions to .

**Exercise B.2.** Find the two solutions to

**Exercise B.3.** Find the two solutions to

**Exercise B.4.** Find the two solutions to

## Visualizing, adding and subtracting complex numbers

We often write a complex number as . We could for example have .

There is a way to display a complex number. To do so, we use the so-called *complex plane*, shown in Figure B.1. In this plane, the *real numbers* (numbers without any imaginary part, like 2, -5 or ) are placed on the horizontal line. The *imaginary numbers* (numbers that only consist of a complex part, like , or ) are placed on the vertical line. The *complex numbers* (basically all numbers, including the ones with real and imaginary parts together) now span the entire plane. Numbers are subsequently drawn as arrows.

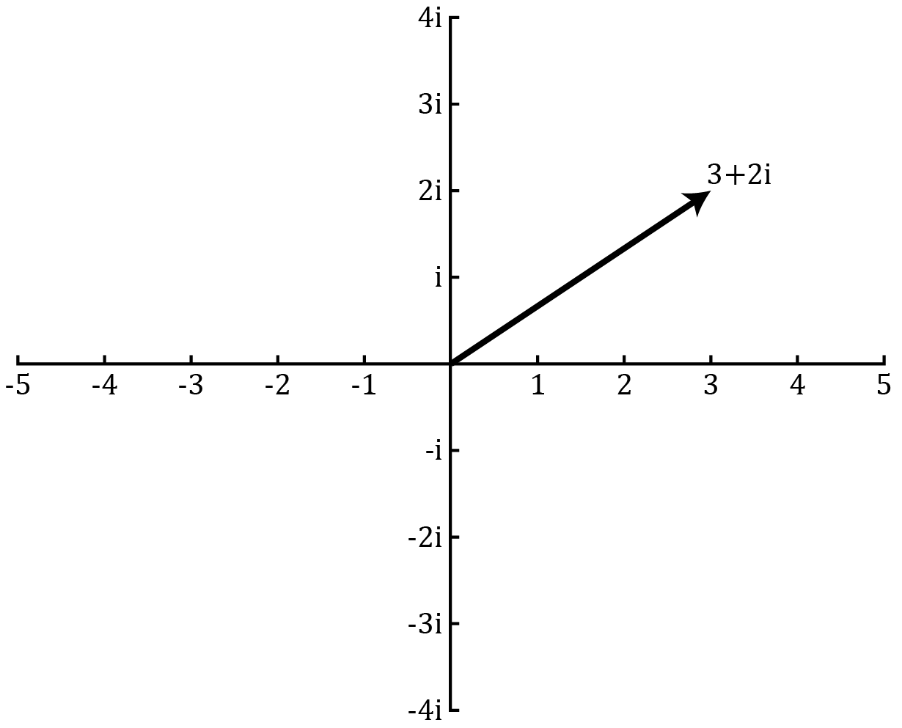


Figure B.1: The complex plane, including the example number , displayed as an arrow.

It is possible to add complex numbers. To do so, you add their real parts and complex parts together separately. Generally, if and , then

|  |  |
| --- | --- |
|  | (B.1) |

We can also visualize this in the complex plane, by putting the arrows one after the other. See Figure B.2.

**Exercise B.5.** Add the complex numbers and .

**Exercise B.6.** Add the complex numbers and .

**Exercise B.7.** Add the complex numbers and .

**Exercise B.8.** Subtract the complex numbers and . That is, calculate .

**Exercise B.9.** Find a generic rule for subtraction. That is, if and , then what is ? Write your answer in a similar form as equation (B.1).

**Exercise B.10.** If , what is equal to? Write it as . Do the same for and , with a real number.

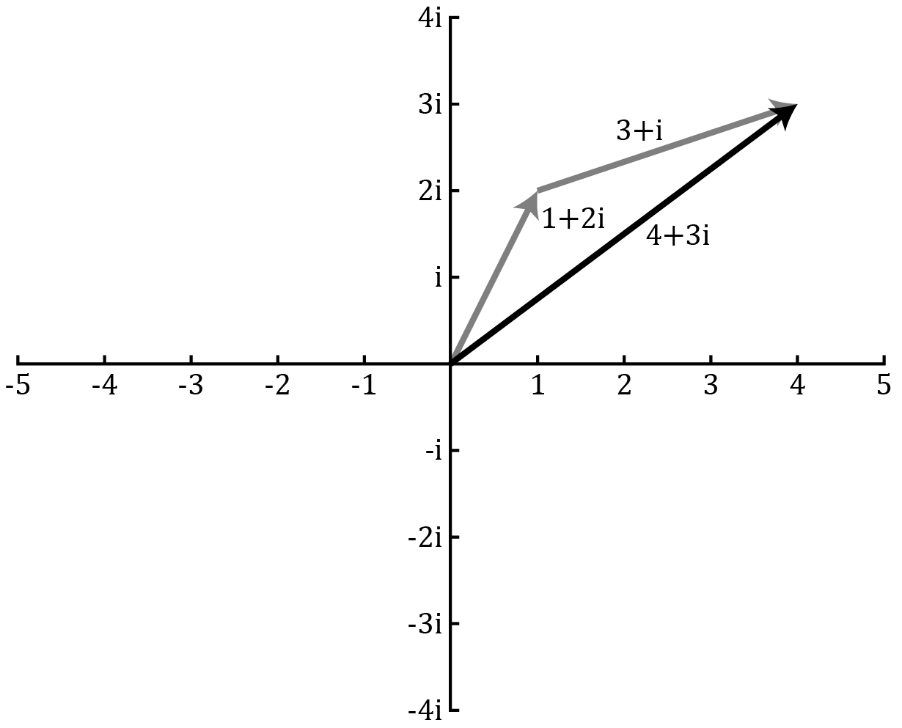


Figure B.2: Addition of complex numbers comes down to putting one arrow after another.

## Multiplying and dividing complex numbers

Just like you can multiply real numbers, it is also possible to multiply complex numbers. When calculating products, it helps to keep in mind that . So if and , then the product equals

|  |  |
| --- | --- |
|  | (B.2) |

**Exercise B.11.** Multiply the complex numbers and .

**Exercise B.12.** Multiply the complex numbers and .

**Exercise B.13.** Multiply the complex numbers and .

In the last exercise, you most likely wound up with a real number. This is no coincidence, and we can figure out why. If we take a number and invert the complex part, we find the so-called *complex conjugate*. We write this as . In the complex plane, and are each other’s mirror image, with respect to the real (horizontal) line.

**Exercise B.14.** For the complex number , find the complex conjugate . Then multiply and .

**Exercise B.15.** Prove that the product is always a real number. That is, using expression (B.2), argue why the product between a complex number and its complex conjugate has a zero imaginary part.

Next to multiplication, it is also possible to divide complex numbers. If and , then we can write the fraction as

|  |  |
| --- | --- |
|  | (B.3) |

This is not an easy expression to work with. To simplify its notation, we can apply a convenient trick. We want the denominator (the bottom) of the fraction to become a real number. After all, if we have that, we can simplify the notation to something of the form . To accomplish this, we multiply both the numerator and the denominator of the fraction by . This turns the expression into

|  |  |
| --- | --- |
|  | (B.4) |

If we subsequently expand the brackets, we find

|  |  |
| --- | --- |
|  | (B.5) |

This is as easy as we can make the expression. Now let’s apply it.

**Exercise B.16.** Divide the complex number by .

**Exercise B.17.** Divide the complex number by .

**Exercise B.18.** Find a simpler way to write . Draw the result as an arrow in the complex plane.

## The magnitude and argument of complex numbers

Every complex number has both a *magnitude* and an *argument*. The *magnitude*  is defined as the length of the arrow representing the complex number. So, if , then

|  |  |
| --- | --- |
|  | (B.6) |

At the same time, the *argument* (sometimes also referred to as the phase) is the angle in **radians** between the arrow and the real axis, where counter-clockwise counts as positive. So,

|  |  |
| --- | --- |
|  | (B.7) |

This is all visualized in Figure B.3.

**Exercise B.19.** Calculate the magnitude and argument of , and . Try to do so without a calculator, but by drawing the numbers in the complex plane.

**Exercise B.20.** Calculate the magnitude and argument of and . This time use a calculator.

**Exercise B.21.** Multiply the complex numbers and . For the outcome , find the magnitude and argument. Compare them with the magnitudes and arguments of and . What do you notice?

The results you (probably) just discovered in the exercise always hold. If you have two complex numbers and , and their product equals , then we always have

|  |  |
| --- | --- |
|  | (B.8) |

**Exercise B.22.** If , then what can you say about , given and ?

**Exercise B.23.** If , then what can you say about , given and ?

**Exercise B.24.** Using (B.6) and (B.7) to find the magnitude and argument of . Use this to write it in a simpler way.

**Exercise B.25.** If you know the argument of , then what can you say about the argument of the complex conjugate of ? Given this result, prove that is a positive real number, for any complex number .

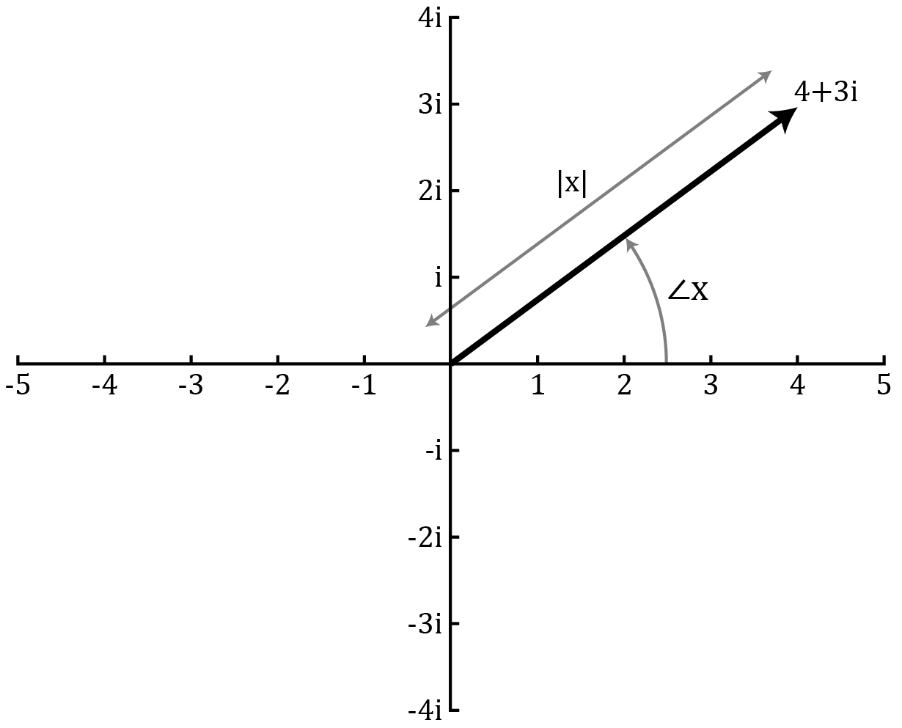


Figure B.3: The magnitude and argument of a complex number.

## Complex exponents and Euler’s formula

Let’s look at exponents. Specifically, we will study exponents of Euler’s number . (It has various special properties that will later come in useful.) We know what is, or or even .

**Exercise B.26.** Use a calculator to find the values of , , , and .

We can also use complex exponents. For example , or . To know what this means, we have to know one important rule. (It is possible to prove this, but its proof is very involved, so we will not do so here.) In general, the number , for any real value , equals the complex number with magnitude 1 and argument . Here, is always measured in radians.

From this piece of data follows *Euler’s formula*,

|  |  |
| --- | --- |
|  | (B.9) |

**Exercise B.27.** Without a calculator, find the values of , , , and . (Hint: .)

We can now use imaginary numbers as exponents. However, what happens when we take a complex number as exponent? For example, what is ? To solve that, we can use the general rule that for any and . So,

|  |  |
| --- | --- |
|  | (B.10) |

**Exercise B.28.** What is the magnitude and argument of ? What is the magnitude and argument of ? What is then the magnitude and argument of ?

A common way of writing any complex number is through its magnitude and argument. We then write it as

|  |  |
| --- | --- |
|  | (B.11) |

**Exercise B.29.** Write the complex numbers , and in the form . Try to do so without a calculator, by drawing the numbers in the complex plane.

A nice property of writing complex numbers in this notation, is that it’s very easy to multiply numbers.

**Exercise B.30.** If , prove that we can write as . (Hint: use , .)

**Exercise B.31.** For the numbers , and , find and . (Hint: if you remember Exercise B.29. then you don’t even need to calculate .)

**Exercise B.32.** If has magnitude and argument , what is then the magnitude and argument of ?

We can also invert Euler’s formula. If we do, we find

|  |  |
| --- | --- |
|  | (B.12) |
|  | (B.13) |

This may seem like a very elaborate way to write a sine or a cosine, but in some areas of mathematics this proves to be very useful.

**Exercise B.33.** Prove the above two expressions, starting from Euler’s formula (B.9).

**Exercise B.34.** A final challenge: write the complex number in the form . What are and here?

## Complex numbers in Python

Python is very well capable of working with complex numbers. The functionality is built into the way Python handles numbers. Several examples of this are shown in Script B.1.

|  |
| --- |
| 1. **import** numpy as np 2. **import** math 4. # Define a complex number. 5. x = 4 + 3j # Sets x equal to 4+3j. In computer science j is the imaginary number. 6. x = complex(4, 3) # Does the same. 7. x = 4 + 3\*((-1)\*\*0.5) # A very roundabout way of doing the same thing. 8. **print**(x) # Shows (4+3j). 10. # Adding complex numbers. 11. x = 1 + 2j 12. y = 3 + 1j # Writing "3 + j" does not work. Then Python sees "j" as a variable. 13. **print**(x + y) # Shows (4+3j). 15. # Multiplying complex numbers. 16. x = 4 - 1j 17. y = 3 + 5j 18. **print**(x\*y) # Gives (17+17j). 20. # Complex conjugate. 21. x = 4 + 3j 22. **print**(x.conjugate()) # Gives (4-3j). 23. **print**(x\*x.conjugate()) # Gives 25. 25. # Dividing complex numbers. 26. x = 17 + 17j 27. y = 3 + 5j 28. **print**(x/y) # Gives (4-1j). (Possibly with minor numerical errors.) 30. # Real and imaginary parts. 31. x = 4 + 3j 32. **print**(x.real) # Gives 4. 33. **print**(x.imag) # Gives 3. 35. # Magnitude and argument. 36. **print**(abs(x)) # Gives 5. 37. **print**(np.angle(x)) # Phase in radians. Gives 0.6435... 38. **print**(math.atan2(x.imag, x.real)) # Also phase in radians. Gives 0.6435... 39. **print**(math.degrees(np.angle(x))) # Phase in degrees. Gives 36.8699... 41. # Complex exponents. 42. **print**(math.e\*\*0j) # Gives 1. 43. **print**(np.exp(0j)) # Also gives 1. 44. **print**(np.exp(math.pi\*1j)) # Gives -1. (Possibly with minor numerical errors.) 45. **print**(np.exp(math.pi\*0.5j)) # Gives 1j. 46. **print**(np.exp(math.pi\*1.5j)) # Gives -1j. 47. **print**(4\*np.exp(2/3\*math.pi\*1j)) # Gives (-2+3.464j). 48. **print**(1j\*\*1j) # Gives 0.20788. |

Script B.1: Examples of Python working with complex numbers. As always, there are multiple ways of achieving the same outcome.

# Differential equations

Many processes and systems in the world around us can be modelled through differential equations. Understanding such equations is therefore a crucial skill for every engineer.

In this appendix we study what differential equations mean. We then also look at various ways of solving simple differential equations analytically. Analytically solving differential equations is not part of the Engineering Fundamentals, since you could always numerically simulate differential equations. However, being able to solve them does give extra insights in what solutions to differential equations will look like, and this will therefore help in analyzing systems. The theory is also discussed in Chapters 12 and 13 of the Engineering Mathematics book.

## Basics of differential equations

We start off by discussing the basics of differential equations. What are derivatives? What are differential equations? And what methods can we use to solve the most basic differential equations?

### What is a derivative?

Differential equations make use of derivatives. For this course it is assumed you know what a derivative is and how to find the derivative of a function. To make sure we are all on the same page, we do briefly discuss what a derivative intuitively comes down to.

Suppose we have some time-dependent function . (The notation “” indicates that the value of depends on the time .) You could attach a meaning to this function – for example the temperature throughout the day – but mathematically that is irrelevant. At every time this function gives us a value . We could plot these values in a graph.

The derivative is now defined as the *rate of change* of this function. You could also see it as the *slope* of the function. If is the temperature, then is the rate at which the temperature is increasing (if positive) or decreasing (if negative).

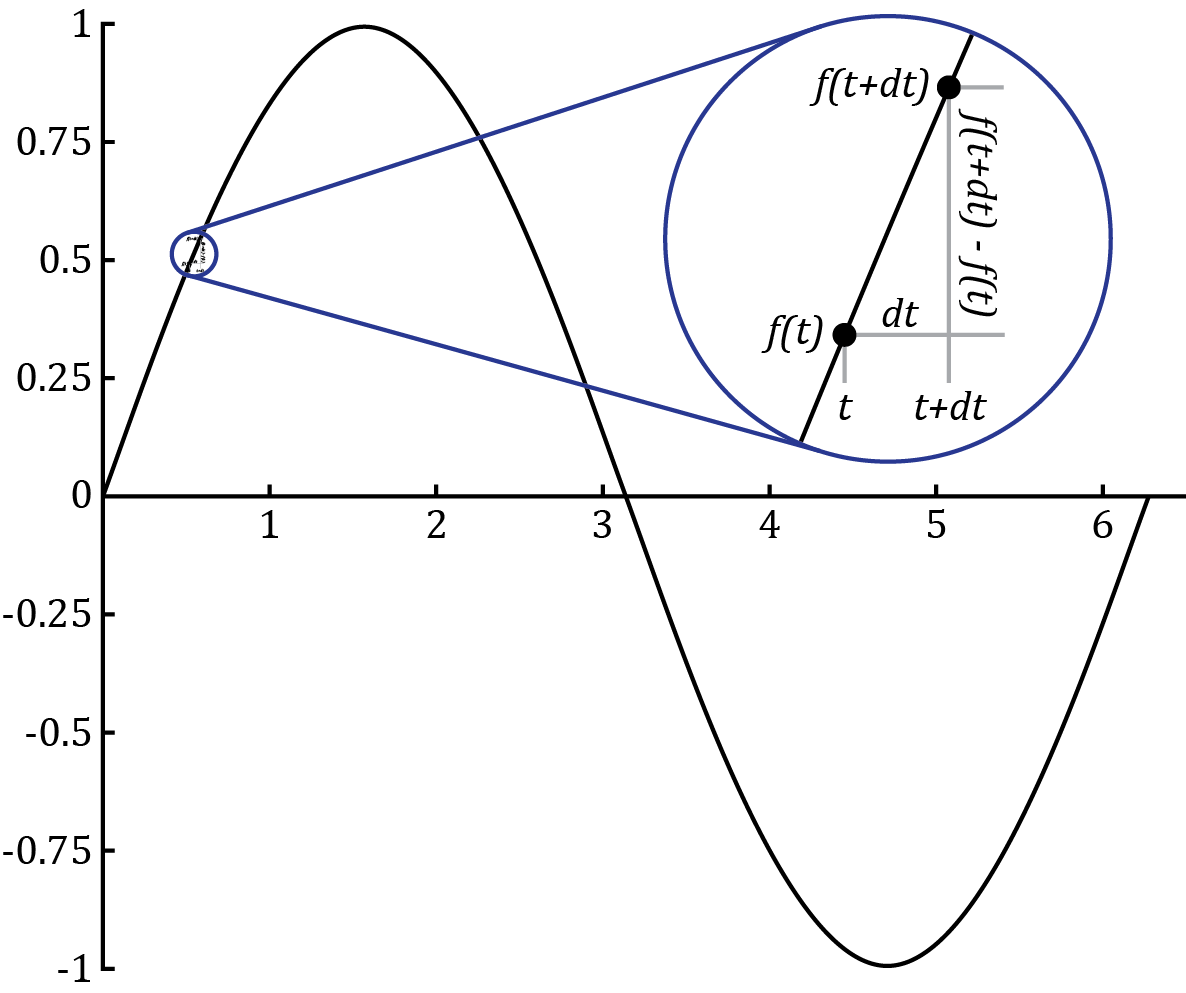


Figure C.1: A visualization of the definition of the derivative, for the function .

To find the derivative at some time , we could look at the function value and compare it to the function value a very brief moment later. We take the difference between these function values (the increase of the function value) and divide this by the time to get the increase per second; the rate of change. In an equation this is

|  |  |
| --- | --- |
|  | (C.1) |

This concept is visualized in Figure C.1, for the sample function .

**Exercise C.1.** For , calculate at . Use , and . Do you get different results? If so, why is this?

For most mathematical functions, methods exist to analytically find a function of the derivative. For example, for the function , we know the derivative is . There is a large variety of such methods to find derivatives, and we will not discuss those here. We assume that you know concepts like the chain rule, the product rule, etcetera.

**Exercise C.2.** Calculate at and compare your result to that of the previous exercise.

What we have found so far was the first derivative . However, we can also take the derivative of this function once more. This gives us the second derivative . For the function , the first derivative is and the second derivative is . Intuitively, this is the rate of change of the rate of change. If is the temperature, then you could for instance say, “The temperature is increasing ( is positive) but the rate of temperature increase is lowering ( is negative).” The second derivative could also be seen of the *curvature* of the graph of , as shown in Figure C.2.

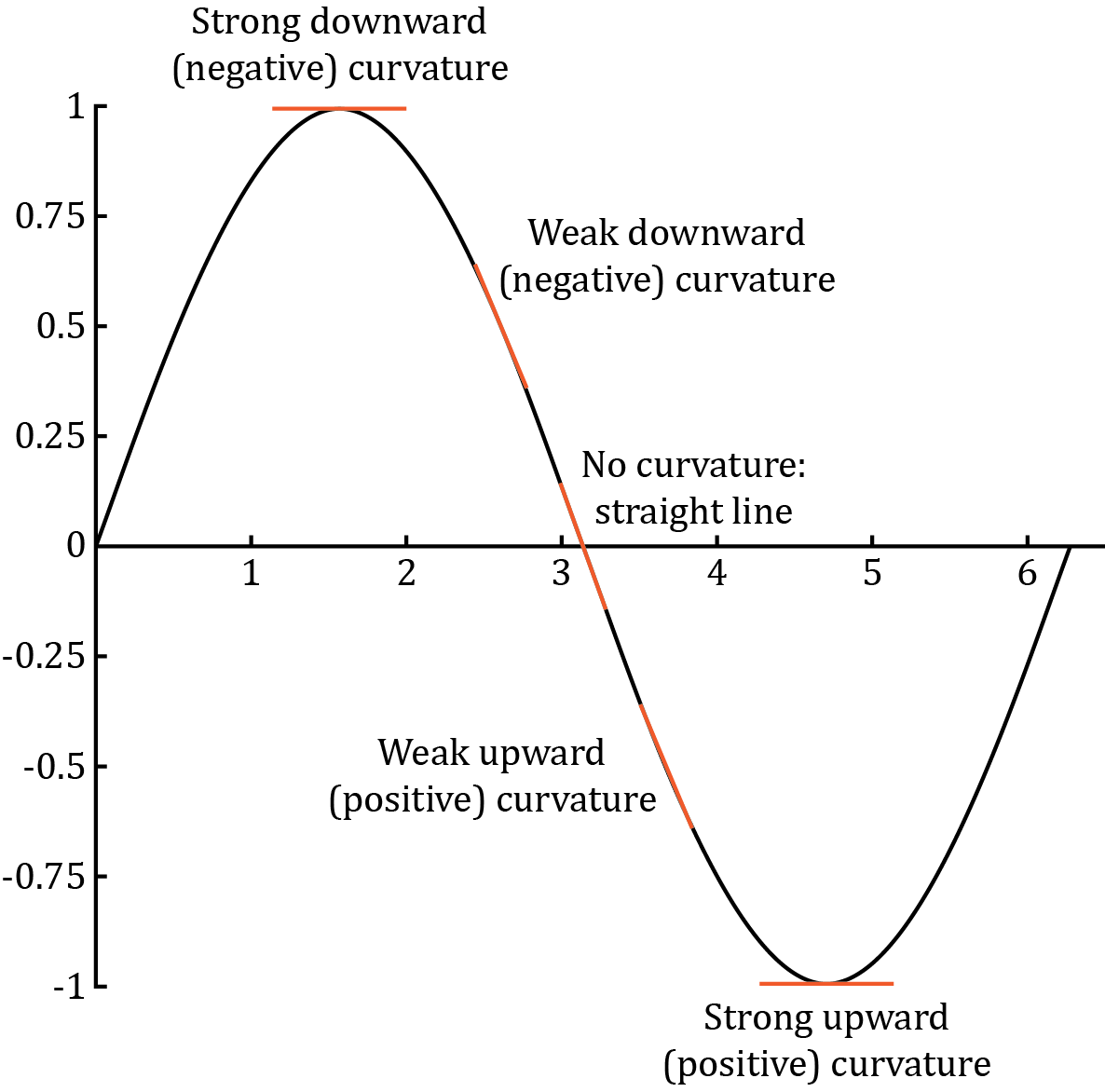


Figure C.2: The second derivative can be seen as the curvature of a function. Here is it shown for the function .

It is also possible to find a third derivative , a fourth derivative and so on. These derivatives have less of an intuitive meaning, but in some applications they may be relevant.

### What is a differential equation?

In some cases we know the derivative. For instance, if we study the population size of a group of rabbits, and we know that every day five more rabbits are born, then we get the equation

|  |  |
| --- | --- |
|  | (C.2) |

with the size of our population and the time in days. (Assuming no rabbits die or leave.) In this case simple integration (the inverse of differentiation) will give us the solution ,

|  |  |
| --- | --- |
|  | (C.3) |

This solution has a constant . No matter the value of this constant, the above equation will be a solution to (C.2). So technically we have found infinitely many solutions! However, only one applies to our rabbit conundrum. To figure out which solution applies, we must use so-called *initial conditions/boundary conditions*: what is the situation at the start/end? In our case, we could for instance specify that we initially have rabbits. We write this number as . If we then insert into the above solution, we immediately find

|  |  |
| --- | --- |
|  | (C.4) |

Hence the constant must equal the initial rabbit population . Our rabbit population therefore satisfies

|  |  |
| --- | --- |
|  | (C.5) |

This is the solution to our problem. First we solve the equation and then we determine constants through initial conditions/boundary conditions.

Though equation (C.2) is technically a differential equation, we can solve it through mere integration. As a result, we often do not even call this a differential equations. To get a non-trivial differential equation, let’s change our model. Now we say that a rabbit gets a child every days. In this case, the growth rate of our population is (on average, ignoring random fluctuations, and ignoring gender) equal to

|  |  |
| --- | --- |
|  | (C.6) |

In this equation the rate of change depends on the value of itself! If that is the case – if we have an equation with both and its derivative (or a higher derivative) – then we have a *differential equation*.

### Solving a first-order differential equation: trial and error

Solving differential equations cannot be done through differentiation/integration. It is pretty much an art in itself. Often we need to “try” functions and hope that one works.

We will apply this trial-and-error approach to equation (C.6). However, to take into account possible changes in the birth rate of our rabbits, we adjust this equation to a more general version

|  |  |
| --- | --- |
|  | (C.7) |

One trial solution that often works is the solution

|  |  |
| --- | --- |
|  | (C.8) |

with some constant that we still need to determine. We now want to know if this expression is a solution of (C.7) for some value of . To find out, we need to plug it into (C.7). When we do, this equation turns into

|  |  |
| --- | --- |
|  | (C.9) |

For to be a solution to the differential equation, the above relation must hold for *all* . That is only the case when . Hence, is a solution to our differential equation!

It is however not the only solution. When looking at the differential equation (C.7), we can note that it is *linear*. That is, there are only terms of the form , with some constant and some derivative (or potentially itself). For linear differential equations, if we know one solution (for instance ) then any linear multiple (for instance for any constant ) is *also* a solution to the differential equation. So the general solution to the differential equation is

|  |  |
| --- | --- |
|  | (C.10) |

for any constant .

**Exercise C.3.** Verify the above solution. That is, find and plug it (together with ) into (C.7). Show that the equation still holds.

To find which value of applies for us, we need to look at the so-called *boundary conditions*: what values of do we already know? In our case, we may know the initial number of rabbits . Inserting into the above solution gives us

|  |  |
| --- | --- |
|  | (C.11) |

Our boundary condition hence tells us that our constant must equal the initial population of rabbits .

**Exercise C.4.** For , plot . Calculate at time days. Next, calculate the derivative at that time using (C.1). (Choose a proper value of yourself.) Compare the two values. Do you see a link?

### Solving a first-order differential equation: the integrating factor

For some simple differential equations methods, we do not have to apply trial-and-error: methods exist to analytically solve them. One example of such a differential equation is the *linear first-order differential equation* which we just solved. (First-order means that only a function and its first derivative are present.) This solution is a bit mathematical, so if you don’t follow, feel free to skip this paragraph.

To solve (C.7), we rewrite it to

|  |  |
| --- | --- |
|  | (C.12) |

Next, we multiply this equation by the so-called *integrating factor* . This results in

|  |  |
| --- | --- |
|  | (C.13) |

Subsequently we apply the inverse product rule. That turns the above into

|  |  |
| --- | --- |
|  | (C.14) |

We integrate this equation from time to a specific time . The rules of mathematics say that, if we define to be our integration limit, we may not also use inside our integral as integrating parameters. So we replace that one by , allowing us to write

|  |  |
| --- | --- |
|  | (C.15) |

Because the derivative is the inverse of the integral, the left-hand side of the equation is readily solved. We only need to take into account the integral bounds. The right-hand integral evaluates to zero. We end up with

|  |  |
| --- | --- |
|  | (C.16) |

The integral bounds can be expanded into

|  |  |
| --- | --- |
|  | (C.17) |

Finally, the solution follows as

|  |  |
| --- | --- |
|  | (C.18) |

As you noticed, this is not the easiest method, despite it being a relatively simple differential equation. As a result, we will stick with the trial-and-error method for the remainder of this appendix.

## Solving second-order linear differential equations

Previously we considered first-order linear differential equations: we only had a function and its derivative . Now we consider second-order linear differential equations which also include . Using trial and error we try to solve them. This will result in some very useful insights.

### A differential equation with real-valued roots

Consider the differential equation

|  |  |
| --- | --- |
|  | (C.19) |

How do we solve it? We could try our earlier approach: just plug in and hope that it works. If we do, the above turns into

|  |  |
| --- | --- |
|  | (C.20) |

By pulling out of brackets, we can rewrite this to

|  |  |
| --- | --- |
|  | (C.21) |

This equation must hold for all times . We can note that the exponent is not going to equal zero. As a result, for the above expression to hold, we must have

|  |  |
| --- | --- |
|  | (C.22) |

This equation is known as the *characteristic equation* of the differential equation. Its solutions are known as the *characteristic roots*. For a second-order differential equation, there are always two solutions. For our example, we have the characteristic roots and . We hence get two solutions and . But, because the differential equation is linear, any linear combination of these solutions is also a solution. Hence, our general solution is

|  |  |
| --- | --- |
|  | (C.23) |

This solution satisfies (C.19) for any constants and . What these specific constants must be depends on our boundary conditions.

**Exercise C.5.** Verify that the above general solution indeed satisfies (C.19) for any constants and .

**Exercise C.6.** If we know that and , find the constants and .

### A differential equation with purely imaginary roots

As a next example we consider the differential equation

|  |  |
| --- | --- |
|  | (C.24) |

You may actually be able to guess the solution to this one.

**Exercise C.7.** Do you know any function for which, if we find the second derivative , we get ? So only a minus sign and a factor 4 are added? (Hint: you saw a function similar to it at Section C.1.1.)

Despite knowing the solution, we will try our trial-and-error approach to this differential equation. Inserting will give us the characteristic equation

|  |  |
| --- | --- |
|  | (C.25) |

**Exercise C.8.** Find the two solutions and for the above equation. (Hint: you will need Appendix B.)

As a result, we get the general solution

|  |  |
| --- | --- |
|  | (C.26) |

This may surprise you. After all, is a complex value! (This follows from Euler’s formula, equation (B.9).) However, is some quantity that actually occurs in real life. There is nothing complex or imaginary about it. To ensure that remains real, the constants and must also be complex. So it looks like we will have to work with complex numbers, which would be difficult.

Luckily, there is a work-around! It would be a lot more sensible to use somewhat different general solution,

|  |  |
| --- | --- |
|  | (C.27) |

**Exercise C.9.** Verify that the above solution is indeed a general solution to the differential equation (C.24).

**Exercise C.10.** For the latter solution, if we know that and , find the constants and .

**Exercise C.11.** [Optional/difficult] For the former solution , and given and , find the complex-valued constants and . Using Euler’s formula, try to simplify your final result.

The nice thing is, if we wind up with purely imaginary characteristic roots and , then we are always allowed to adjust our general solution accordingly. We replace the exponents and by a sine and a cosine. Mathematically, it can be proven (you may have accidentally done so in the last exercise) that these solutions are equivalent.

### Solving using sine and cosine

The next second-order linear differential equation that we will solve together is

|  |  |
| --- | --- |
|  | (C.28) |

**Exercise C.12.** Find the characteristic equation for this differential equation, including its roots and .

This time we have complex-valued roots! You may have noticed that and are complex conjugates. This is actually always the case: complex roots always come in pairs. If one root equals , then the other must be . (In our case we have and .) How do we deal with this?

One option is to once more use exponents. Our general solution will then be

|  |  |
| --- | --- |
|  | (C.29) |

In this case the constants and will once more be complex, and the mathematics will be very difficult. As a result, we will once more adjust our general solution. You may recall that we may write as , which in turn can be written as . This leads to the idea that we write the above as

|  |  |
| --- | --- |
|  | (C.30) |

If we apply the same trick as before, writing imaginary exponents as a sine and cosine, we get the general solution

|  |  |
| --- | --- |
|  | (C.31) |

where in our case we still have and . For complex-valued roots this is our general solution.

**Exercise C.13.** For our value of and , verify that the above expression is indeed a solution of (C.28).

**Exercise C.14.** If we know that and , find the constants and .

**Exercise C.15.** Solve the differential equation

|  |  |
| --- | --- |
|  | (C.32) |

for the initial conditions and .

### Identical characteristic roots

The last linear second-order differential equation that we will look at is

|  |  |
| --- | --- |
|  | (C.33) |

We once more determine the characteristic equation,

|  |  |
| --- | --- |
|  | (C.34) |

Because we have a second-order linear differential equation, we must get two roots to this equation. These roots are and . Yes, that’s correct: we say that the root appears twice. This is because of the term , which contains a square.

How does our solution method work if we have multiple *identical* roots? What is the general solution now? We of course have the term . However, in the case that we have two identical roots, the term (so the usual solution multiplied by ) is also a solution! The general solution is hence

|  |  |
| --- | --- |
|  | (C.35) |

**Exercise C.16.** Verify that the above solution is the general solution to (C.33). Show that it holds for any .

This rationale can even be extended further. In case the root appears three times, then is also a solution to the differential equation, and so on.

**Exercise C.17.** If we know that and , find the constants and .

## Solving general linear differential equations

So far we have considered at most second-order linear differential equations. Next, we will consider linear differential equations of any order. As we will see, things will be very similar then.

### The general solution, from the characteristic roots

We will solve the fourth-order linear differential equation

|  |  |
| --- | --- |
|  | (C.36) |

To do so, we follow our usual approach: we determine the characteristic equation

|  |  |
| --- | --- |
|  | (C.37) |

and subsequently find its roots. This is too difficult to be done analytically, but we can use Python. Through the function **print(np.roots([1,6,25,28,-60]))** we can find the roots. The result is . We see the first two roots are real. We hence get terms and in our solution. The other two roots are complex. We hence also get terms and . This gives us the general solution

|  |  |
| --- | --- |
|  | (C.38) |

We can insert this solution into the differential equation to verify it, but this will be a lot of work, so we skip this.

**Exercise C.18.** The differential equation has the characteristic equation

|  |  |
| --- | --- |
|  | (C.39) |

Find the characteristic roots and use them to formulate the general solution of the differential equation.

### Stability of the general solution

A very important concept when dealing with differential equations, and with dynamic systems in general, is that of *stability*. The main question here is, “If we wait infinitely long, will the value of converge to zero, continue to oscillate, or diverge to infinity?”

To answer this question, we must look at our general solution, and specifically at the factors . If is positive, then goes to infinity as goes to infinity. (We write this as .) However, if is negative, then converges to zero (). In the special case that , we don’t converge to 0, but we don’t diverge to infinity either.

**Exercise C.19.** For the general solutions of equation (C.23), (C.27), (C.31) and (C.35), determine the stability. That is, determine if (convergence), (divergence) or neither. Do the same for (C.38).

If the solution of a differential equation always convergences, then the corresponding system is said to be *stable*. If the solution may diverge, the system is called *unstable*. In other cases (continuous oscillation or convergence to a non-zero value) the system is *marginally stable*.

The stability of the system only depends on the characteristic roots of the differential equation. In particular, if some of the characteristic roots are complex, , then the real part is crucial here. The following important rules now hold.

* If *all* characteristic roots have a negative real part, then the system is *stable*.
* If there is *any* root with a positive real part, then the equilibrium point is *unstable*. Alternatively, if there are two or more identical roots with a zero real part, then the system is also unstable.
* In any other case, the system is marginally unstable. This is the case when some characteristic roots have a zero real part, but these roots are all different. Other roots may still have a negative real part.

**Exercise C.20.** For the differential equations (C.19), (C.24), (C.28) and (C.33), determine the stability of the corresponding system using the above rules. Do the same for differential equation (C.36).

**Exercise C.21.** For the differential equations , find the characteristic roots. Use it to determine the stability of the system. Does this make sense? Explain your answer by comparing it with the general solution.

### Homogeneous and non-homogeneous solutions

Previously, we have always considered differential equations with only terms , with some constant some derivative order (possibly equal to zero for ). These are so-called *homogeneous­* differential equations.Next, we will change this. We will add a term that doesn’t satisfy this format. When we do, we write this term on the right-hand side of the equation. The corresponding differential equation is then called *non-homogeneous*. An example of a non-homogeneous differential equation is

|  |  |
| --- | --- |
|  | (C.40) |

Note the term on the right.

To solve a non-homogeneous differential equation, we must follow two steps.

* Pretend that the differential equation is homogeneous, ignoring the right-hand term. Through this, find the *general solution to the homogeneous equation* .
* Find only a single *specific solution to the non-homogeneous equation* .

The general solution to the full non-homogeneous differential equation is then the sum of these two solutions.

Let’s apply this method to our example.

**Exercise C.22.** Find the general solution to the homogeneous equation .

**Exercise C.23.** Find a single specific solution to the non-homogeneous differential equation (C.40). (Hint: just try something for and see if it works. For example, try a constant and see what must satisfy.)

**Exercise C.24.** Add up the two solutions and . Verify that the result satisfies (C.40) for any .

**Exercise C.25.** If we know that and , find the constants and .

**Exercise C.26.** For the differential equation

|  |  |
| --- | --- |
|  | (C.41) |

find the general solution . (Hint: for the specific solution, try of a form similar to the non-homogeneous term . In this case would be a good suggestion. What must satisfy?)

We should also discuss the stability of non-homogeneous systems. This is actually easier than you might think: we say that the stability of the non-homogeneous system is *exactly the same* as that of the homogeneous system.

To see why this is so, note that our general solution equals . We now want to know whether . If the homogeneous system is unstable, then , which means that also : the non-homogeneous system is also unstable. If this is not the case, the stability depends on the specific solution . However, in practice this specific solution is of lesser importance. If the non-homogeneous term (on the right of the differential equation) remains bounded, then will often also remain bounded. That is why we decide to ignore this part of the solution when talking about stability.

The conclusion: also for non-homogeneous differential equations, to determine the stability, we only have to look at the characteristic roots.

## Partial differential equations

Up until now, we only considered functions or that depended on a single parameter. In practice, functions may depend on multiple parameters. For example, the height of a landscape depends on both the - and the -coordinate: we have a function . How do you take derivatives of such functions? And are there are also differential equations with such derivatives? How do we solve those? That is the subject of this section.

### What is a partial derivative?

Consider a function with two different input parameters. We define the so-called *partial derivative*  as the derivative of this function with respect to , assuming that remains constant. That latter part is very important. For instance, if we have a function

|  |  |
| --- | --- |
|  | (C.42) |

then to find the derivative , we must pretend that is a constant, just like the factor in the above expression. We hence get

|  |  |
| --- | --- |
|  | (C.43) |

You can see this partial derivative as the slope you’re facing if you’d walk in the direction of the -axis. This slope may also depend on the -coordinate you’re currently at, so that is why this is once more a function with two input parameters and .

In an identical way, it’s also possible to find the partial derivative with respect to .

**Exercise C.27.** For the above example function, find .

We can also take the second partial derivative of a function .

**Exercise C.28.** Find the second derivatives and .

**Exercise C.29.** Find the second derivatives by first taking the derivative with respect to and then with respect to . Next, do the same, but first taking the derivative with respect to and then with respect to . Does it make a difference, which order you use? That is, is there a difference between and ?

Often we need partial derivatives with respect to both and . For that, we have defined the so-called *differential* operator . That symbol is called *nabla* and hence it is often also referred to as the nabla operator. If we have a function , then the gradient is defined as the vector

|  |  |
| --- | --- |
|  | (E.44) |

For an introduction into vectors and matrices, see Appendix D. The gradient vector can be seen as the *direction with the highest slope*. For instance, if is very small, but is very large, then we get the largest slope by mostly walking in the direction of the -axis. This is exactly what the gradient tells us: it tells us what direction we must walk to get the largest increase of the function value .

The so-called *Hessian* matrix of is in some sense the second order gradient. It can be defined as

|  |  |
| --- | --- |
|  | (E.45) |

**Exercise C.30.** For the example function we used so far, find the Hessian matrix .

**Exercise C.31.** For , find the gradient and the Hessian matrix .

Another mathematical object that is in some sense also the second order gradient is the Laplacian . Its output is not a vector or a matrix, but a normal function. It is defined as

|  |  |
| --- | --- |
|  | (E.46) |

**Exercise C.32.** For the example function we used so far, find the Laplacian .

**Exercise C.33.** For , find the gradient and the Laplacian .

It is important to note that, in general, the nabla operator ignores the time variable. That is, acting upon functions that also depend on time, it only introduces partial derivatives with respect to spatial variables and vector and matrix components related to their directions.

### Partial differential equations and how to solve them

A partial differential equation is an equation that relates a function with one or more of its partial derivatives.

An example of this is the so-called heat equation. Imagine we have a long and thin metal rod. At every time we measure the temperature at every position along the rod’s longitudinal axis. The temperature can now be seen as a function . The so-called heat equation describes how heat spreads over time. The rate of temperature increase depends on variations of the temperature. Specifically, it depends on the second derivative . If this second derivative is positive, then the temperature at that point will increase. This results in the differential equation

with some constant that depends on the material: how much heat can it store? How well does it conduct heat? Etcetera.

Solving a partial differential equation analytically is difficult. Most technical universities offer a full third-year course on this subject alone. Usually it involves Fourier series, which is an infinite sum of sine-terms and cosine-terms. We will not discuss this here.

Instead, if we need to deal with partial derivatives, we will generally solve them numerically. To do so, we will discretize the system, using a time step and a spatial step . Subsequently, we relate the differences and to function values , , etcetera, using the methods from Appendix E, Section E.2.4. Through the resulting expressions, we can then derive an update equation to implement in our simulation. The exact details of how this is implemented are discussed in the various examples of Chapter 4.

# Linear algebra: vectors and matrices

Algebra is the branch of mathematics where we replace numbers by letters. “Solve ” is a typical algebra question. Linear algebra is the sub-branch where the equation are all linear. “Solve ” is a typical linear algebra question. Note there are only multiplications of known numbers and , and additions of these terms. There is nothing else, like squares, sines, logarithms or anything else.

Linear algebra looks easy, but when you dive deeper into it, there is a whole world behind it. In this appendix we will take only the first few steps into this world. We start by introducing vectors and matrices, and how to calculate with them. Then we examine how to use them to solve equations. Finally we examine eigenvalues and eigenvectors. The theory can also be found in Chapters 8 and 9 of the Engineering Mathematics book.

## What are vectors?

In mathematics and physics, we make a distinction between quantities that only have a magnitude, and quantities that also have a direction. A quantity that only has a magnitude (for example the pressure ) is a *scalar*. If a quantity has both a magnitude and a direction (for example, the velocity ) then it is called a *vector*.

### An example of a vector and its notation

An example of a vector is the vector shown in Figure D.1. You can see a vector as an arrow in space. It has a *component* in the horizontal direction and a *component* in the vertical direction.

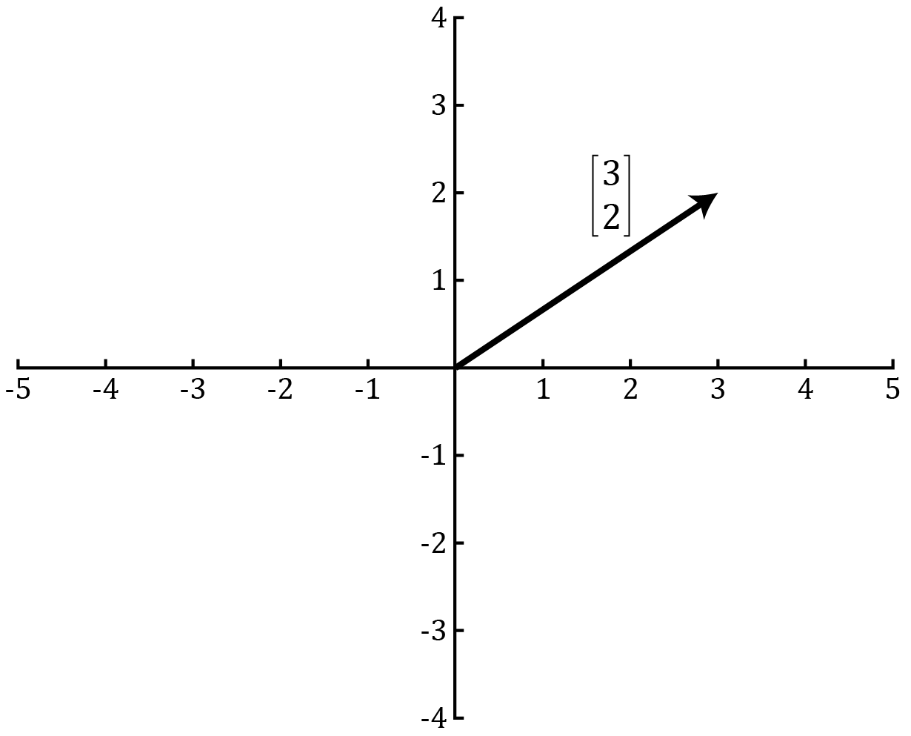


Figure D.1: An example of a vector. In this case it has a horizontal component of 3 and a vertical component of 2.

To distinguish between scalars and vectors, we always write scalars (like the pressure ) in a normal font, while vectors (like the velocity ) are written in a bold font. When writing vectors, we use square brackets to indicate the various components. For instance,

|  |  |
| --- | --- |
|  | (D.1) |

### The length of a vector

A vector has a length, also known as the *magnitude* of the vector. We write this as and we calculate it using Pythagoras’ Theorem. Hence, if we have a (generic) vector

|  |  |
| --- | --- |
|  | (D.2) |

then the magnitude of this vector equals

|  |  |
| --- | --- |
|  | (D.3) |

Note that the magnitude is a scalar quantity. It does not have any direction. In physics, instead of **,** it is also customary to write to denote the magnitude of the vector . For example, if we write , we are talking about a velocity vector (with direction), but if we write , we are dealing with the magnitude of the velocity (without direction).

**Exercise D.1.** Calculate the length of the vector .

### Notation using unit vectors

Another way to write vectors is by using the so-called *unit vectors*. We define the *unit vectors* and as the vectors of unit length along each of the axes. (Don’t confuse these unit vectors with complex numbers. They are very much different things.) So,

|  |  |
| --- | --- |
|  | (D.4) |

See Figure D.2 for a visualization of this. We can then write a vector in terms of these unit vectors. For instance, we may write

|  |  |
| --- | --- |
|  | (D.5) |

In some branches of mathematics this notation is very useful, but we will mainly stick with the bracket notation.

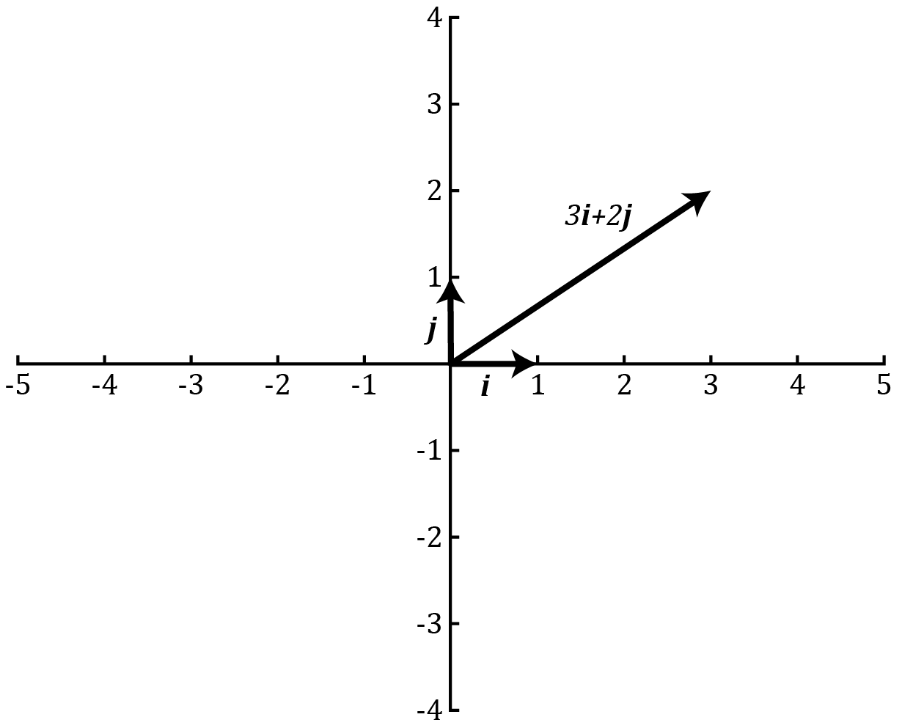


Figure D.2: Notation of vectors through unit vectors. We don't often use this, but it's good to know the notation if you find it elsewhere.

### Higher-dimensional vectors

So far we have discussed vectors in two dimensions. It is also possible to have three-dimensional vectors. (This is where vectors are different from, for instance, complex numbers, which do not have three-dimensional representations.) We can then write

|  |  |
| --- | --- |
|  | (D.6) |

Similarly, you can have vectors in four dimensions, five dimensions or any other number of dimensions. (A one-dimensional vector is a scalar.) The mathematics all work the same. For example, we can still find the magnitude of an eight-dimensional vector using Pythagoras’ Theorem.

**Exercise D.2.** Calculate the magnitude of the vector .

## Calculating with vectors

You can calculate with vectors, just like you can calculate with any other kind of number. Addition works very similarly. Multiplication is different though, as there are multiple ways to multiply vectors.

### Adding vectors

Previously we saw that a vector is like an arrow. When adding vectors, we can put the arrows one after the other. For example, if we have

|  |  |
| --- | --- |
|  | (D.7) |

then the vector equals

|  |  |
| --- | --- |
|  | (D.8) |

This is visualized in Figure D.3.

More generically, if we have vectors and , then their sum equals

|  |  |
| --- | --- |
|  | (D.9) |

So to add up vectors, we may simply add up their respective components.

Note that it is impossible to add a two-dimensional vector to a three-dimensional vector. To add vectors, they must be of the same size.

**Exercise D.3.** Calculate the sum of the vectors and .

**Exercise D.4.** Calculate the subtraction of the vectors and .

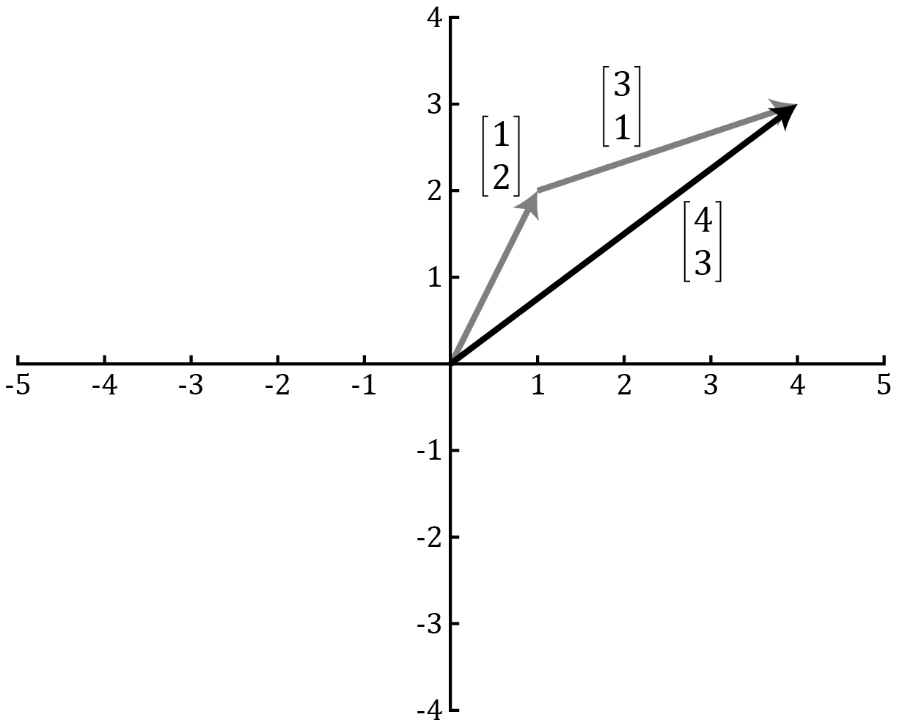


Figure D.3: To add vectors, you can put one after the other.

### Multiplying vectors: the dot product

There are multiple ways to multiply two vectors. The first method is the so-called dot product. It is defined as

|  |  |
| --- | --- |
|  | (D.10) |

So you multiply the respective components of each vector and then add everything up. Note that the dot product results in a *scalar* quantity.

The dot product can be seen as a measure of how much the vectors and point in the same direction. Let’s write the angle between and as , as shown in Figure D.4. We can then prove that the dot product equals

|  |  |
| --- | --- |
|  | (D.11) |

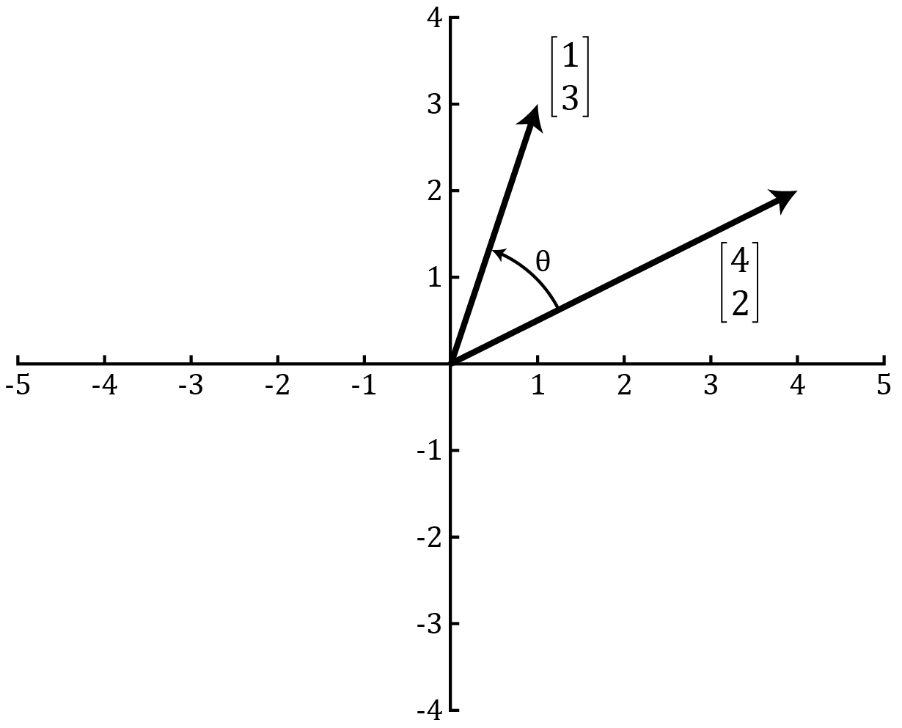


Figure D.4: The dot product equals the product of the magnitudes of both vectors times the cosine of the angle between them.

So if the two vectors point exactly in the same direction () then the dot product is simply the product of the lengths of the two vectors: . If the vectors are however orthogonal to each other () then the dot product always equals zero: . The dot product is hence a nice mathematical trick to see if two vectors are orthogonal to each other, or alternatively to find the angle between them.

**Exercise D.5.** Calculate the dot product of the vectors and .

**Exercise D.6.** Calculate the angle between the vectors and . Verify this with Figure D.4.

**Exercise D.7.** Calculate the dot product of the vectors and . Explain your answer.

**Exercise D.8.** Calculate the dot product of the vectors and . Explain your answer.

**Exercise D.9.** Writing the nabla operator as , show that results in the Laplacian as given in equation (E.46) .

### Multiplying vectors: the cross product

Another way to multiply vectors is the so-called *cross-product*. It only works for vectors of size 3. It is defined as

|  |  |
| --- | --- |
|  | (D.12) |

Note that, unlike the dot product, the cross product results in yet another three-dimensional vector.

There is also a more intuitive way of finding the cross product. First imagine (in 3D) the plane going through both and . The cross product now is the vector perpendicular to this plane. Its magnitude equals

|  |  |
| --- | --- |
|  | (D.13) |

with again the angle between and .

**Exercise D.10.** Calculate the cross product of the vectors and . Explain the outcome.

**Exercise D.11.** Calculate the cross product of the vectors and . What do you notice?

You just saw that the cross product has the peculiar property that **.**

**Exercise D.12.** Prove this property, based on the definition of the cross product.

**Exercise D.13.** Calculate the cross product of the vectors and . Explain the outcome.

**Exercise D.14.** Calculate the cross product of the vectors and . Explain the outcome.

**Exercise** **D.15.** Calculate the cross product of the vectors and . Explain the outcome.

### Vectors in Python

Python has packages that allow you to calculate with vectors. Specifically, the Numpy package allows us to do exactly that. If we give the Numpy array method a list with numbers, it will treat it as a vector. Hence, we can do all sorts of calculations with them. This is shown in Script D.1.

|  |
| --- |
| 1. **import** numpy as np 2. **import** math 4. # Calculate length. 5. x = np.array([3,-6,2]) 6. **print**(np.linalg.norm(x)) # Gives 7. 8. # Add vectors. 9. x = np.array([5,-2]) 10. y = np.array([-3,-4]) 11. **print**(x + y) # Gives [2,-6]. 12. **print**(np.add(x,y)) # Also gives [2,-6]. 14. # Subtract vectors. 15. y = np.array([3,4]) 16. **print**(x - y) # Gives [2,-6]. 17. **print**(np.subtract(x,y)) # Also gives [2,-6]. 19. # Dot product. 20. x = np.array([1,3]) 21. y = np.array([4,2]) 22. **print**(np.dot(x,y)) # Gives 10. 23. theta = math.acos(np.dot(x,y)/np.linalg.norm(x)/np.linalg.norm(y)) # In radians. 24. **print**(theta\*180/math.pi) # Gives 45 degrees. 25. **print**(math.degrees(theta)) # Also gives 45 degrees. 27. # Cross product. 28. x = np.array([1,1,0]) 29. y = np.array([0,0,1]) 30. **print**(np.cross(x,y)) # Gives [1,-1,0]. 32. # Element-wise multiplication and division. 33. x = np.array([2,3,3]) 34. y = np.array([2,2,4]) 35. **print**(x\*y) # Gives [4,6,12]. 36. **print**(np.multiply(x,y)) # Also gives [4,6,12]. 37. **print**(x/y) # Gives [1,1.5,0.75]. 38. **print**(np.divide(x,y)) # Also gives [1,1.5,0.75]. |

Script D.1: Vector calculations in Python. Note that there are often multiple ways to write something in Python.

Note that, to do multiplications, we must specifically say which type of multiplication we want. If we just use x\*y, then Python will multiply the two vectors element-wise,

|  |  |
| --- | --- |
|  | (D.14) |

Element-wise multiplications aren’t often used, so most of the time this is *not* what we want.

## What are matrices?

Next to vectors, there are also matrices. Some see matrices as a table of numbers. Others see them as a transformation. We will take a look at both these points of view.

### Matrix notation

The most common way to write a matrix is as a table of numbers. For example, we can have

|  |  |
| --- | --- |
|  | (D.15) |

As convention, we write matrices with capital letters like . These letters are not boldfaced; only vectors are.

A matrix has a size. The matrix above has a size 2 by 3. We can also write that is a matrix. The first number here denotes the height and the second number denotes the width. When a matrix has an equal height and width, then we call the matrix a *square matrix*. Note that a vector is technically also a matrix. It just happens to be an matrix. Even a scalar can be seen as a matrix, but then a matrix.

### A matrix as a transformation

Intuitively, you can see a matrix as a transformation. Take for example the matrix

|  |  |
| --- | --- |
|  | (D.16) |

We can multiply a vector with this matrix. (We will look later into how multiplication exactly works.) The resulting vector will always be a counterclockwise rotation of . This is visualized in Figure D.5.

|  |  |
| --- | --- |
|  |  |

Figure D.5: A vector before (left) and after (right) being multiplied by a transformation matrix causing rotation.

|  |  |
| --- | --- |
|  |  |

Figure D.6: A vector before (left) and after (right) being multiplied by a transformation matrix causing shearing in the x-direction.

Or take for example the matrix

|  |  |
| --- | --- |
|  | (D.17) |

This matrix shears whatever vector we multiply it with, as shown in Figure D.6. Or we can take the matrix

|  |  |
| --- | --- |
|  | (D.18) |

This matrix shrinks any vector we multiply it with, effectively halving its length. This is shown in Figure D.7.

|  |  |
| --- | --- |
|  |  |

Figure D.7: A vector before (left) and after (right) being multiplied by a transformation matrix causing shrinking.

Matrices can do a lot more than just transform vectors, but for now seeing a matrix as a transformation helps us to give matrices an intuitive meaning.

### The matrix determinant

A vector has a magnitude. Similarly, a matrix has a determinant. We write this determinant as , or sometimes as . Calculating a determinant for a 2 by 2 matrix is still manageable: if we have a matrix

|  |  |
| --- | --- |
|  | (D.19) |

then the determinant equals

|  |  |
| --- | --- |
|  | (D.20) |

For larger matrices it is more difficult to calculate the determinant, so we will not discuss that here. We just leave that to Python whenever we need it. Also, if a matrix is not square (its width and height are different) then it is not possible to calculate its determinant. The determinant is then undefined. Even Python doesn’t know how to calculate it.

What is a determinant then? We know that a matrix can be seen as a transformation. If we transform a shape (for example a square) then the determinant is the so-called *magnification factor*: it describes how much the surface area of our shape has grown. For example, if we look at the square from Figure D.5 through Figure D.7, then the determinant describes how much the area of this square has grown. (Or, for three-dimensional shapes, how much the volume has grown. Or, for four-dimensional shapes, the hypervolume.)

**Exercise D.16.** For the transformation matrices , and , calculate the determinant. Explain the outcome.

## Calculating with matrices

Just like with vectors, we can also calculate with matrices. Specifically, we look at how to add matrices, how to multiply a matrix with a vector, and how to multiply a matrix with another matrix.

### Adding matrices

Adding matrices is still relatively easy. If we have two matrices

|  |  |
| --- | --- |
|  | (D.21) |

then to find their sum we can simply add up the corresponding elements. That is,

|  |  |
| --- | --- |
|  | (D.22) |

Note that we can only add up matrices in this way of the matrices have exactly the same size. The outcome will then also be a matrix of this size. It is impossible to add up matrices of different sizes.

### Multiplying a matrix by a vector

Suppose we have a matrix and a vector of size . In that case we can multiply them using

|  |  |
| --- | --- |
|  | (D.23) |

This looks complicated, so there’s a few things we should point out. First we should note the size. If we multiply a matrix by a matrix (a vector of size 3) then we get a matrix (a vector of size 2). This is always the case. In fact, the width of the matrix always has to equal the height of the vector. Otherwise it is not possibly to multiply them.

Secondly, we should look at how the two terms in our resulting vector are calculated. To find them, we can (in our mind) pick up the vector , rotate it counterclockwise, hover it over the first row of , multiply all terms that are hovering over each other (like with , with , and so forth) and add up the result of all these multiplications. This gives us a single term: the top term of our multiplication outcome. Then we do the same for the second row, and for any other row we may have.

Another way to look at it, is to consider the *rows* of as vectors and. The product can then also be found by using the dot product, through

|  |  |
| --- | --- |
|  | (D.24) |

There are many ways to remember how to perform matrix multiplications. You just have to find a way that intuitively works for you.

**Exercise D.17.** For the vector , and for the matrices , and defined earlier, calculate **,**  and . Check your results by comparing it with Figure D.5 through Figure D.7.

**Exercise D.18.** Perform the multiplication

|  |  |
| --- | --- |
|  | (D.25) |

**Exercise D.19.** Is it allowed to multiply a vector by a square matrix ? That is, is it possible to calculate ? If so, when is it possible? If not, why is it not possible?

### Multiplying a matrix by a matrix

It is also possible to multiply a matrix with a matrix. This works nearly the same as multiplying a matrix by a vector. We only have to do the same procedure multiple times.

Suppose we multiply a matrix by a matrix . We then per definition have

|  |  |
| --- | --- |
|  | (D.26) |

This looks like a humongous equation. However, it is exactly the same as we did previously. To see how that works, first look at the size of the resulting matrix. It is a matrix. In general, if we multiply a matrix with an matrix, then the outcome will be a matrix. Do note that the *width* of the first matrix must equal the *height* of the second matrix. Otherwise the multiplication is impossible to perform.

To find the first column of our result, we must use the whole matrix , but we only have to use the first column of . In fact, if we write this column as the vector , then the first column of our resulting matrix equals . For the second column of our outcome, we must then use the second column of . So,

|  |  |
| --- | --- |
|  | (D.27) |

An alternative way to look at this, is to once again see as a matrix of dot products. We then take the dot products of the rows of and the columns of . We write this as

|  |  |
| --- | --- |
|  | (D.28) |

Again, there are multiple ways to remember how matrix multiplications exactly work. Find a way that works for you and stick with it.

**Exercise D.20.** For the matrices and defined earlier, calculate . Then use this to calculate for . Explain the outcome.

**Exercise D.21.** For the matrices and defined earlier, calculate . Then use this to calculate for . Explain the outcome.

**Exercise D.22.** Perform the multiplication

|  |  |
| --- | --- |
|  | (D.29) |

**Exercise D.23.** Perform the multiplication

|  |  |
| --- | --- |
|  | (D.30) |

**Exercise D.24.** Perform the two multiplications and . Do you get the same result?

When multiplying two matrices and , then the multiplications and are usually different! Only in some special cases does equal . In that case we say that the matrices and *commute*. Usually this is not the case. When multiplying a matrix by a matrix , it is hence important to note if we want to left-multiply by (and get , with on the left) or if we want to right-multiply by (and get , with on the right).

### The identity matrix

**Exercise D.25.** Perform the multiplication

|  |  |
| --- | --- |
|  | (D.31) |

A special matrix is the *identity matrix* . This is a square matrix filled with zeros, but having ones along its diagonal. So the identity matrix is

|  |  |
| --- | --- |
|  | (D.32) |

and the identity matrix is

|  |  |
| --- | --- |
|  | (D.33) |

The special property of this matrix is that, if we multiply *any* matrix by the identity matrix, then we still get . So and . This even works if is not square, as long as has a matching size. is per definition always square.

### Matrices in Python

Working with matrices in Python works similarly to working with vectors. After all, a vector is also a matrix. A few basic examples are shown in Script D.2.

|  |
| --- |
| 1. **import** numpy as np 3. # Matrix determinant. 4. A = np.array([[ 0.5, 0   ], 5. [ 0,   0.5 ]]) 6. **print**(np.linalg.det(A)) # Gives 0.25.  9. # Multiplying a matrix by a vector. 10. A = np.array([[ 1,  0], 11. [ 3,  -1], 12. [ 35/6, 1/2]]) 13. x = np.array([3, -5]) 14. **print**(np.dot(A, x)) # Gives [3, 14, 15]. 15. **print**(np.matmul(A, x)) # Also gives [3, 14, 15]. 17. # Multiplying a matrix by a matrix. 18. A = np.array([[ 1,  0], 19. [ 3,  -1], 20. [ 35/6, 1/2]]) 21. B = np.array([[ 3,  9], 22. [ -5, 1]]) 23. **print**(np.dot(A, B)) # Gives [[3, 14, 15], [9, 26, 53]]. 24. **print**(np.matmul(A, B)) # Also gives [[3, 14, 15], [9, 26, 53]]. |

Script D.2: Multiplying matrices in Python can also be done in various ways.

## Other matrix operations

There are various other operations we can apply to matrices. We will examine the matrix transpose, the matrix inverse and the matrix exponent.

### Matrix transpose

To find the transpose of a matrix , written as , we have to flip the rows and columns of the matrix. The first row becomes the first column, the second row becomes the second column, and so forth. For example, if

|  |  |
| --- | --- |
|  | (D.34) |

then the transpose of this matrix equals

|  |  |
| --- | --- |
|  | (D.35) |

So if the matrix is of size , then its transpose will have size .

You can envision a transpose as a “rotation” of the matrix. Imagine you draw a line from the top left to the right bottom of the matrix. Then you flip the matrix (in 3D) around this line. The resulting matrix is your transpose. (If you have transparent paper, you could actually flip the paper like this!)

**Exercise D.26.** Find the transpose of

|  |  |
| --- | --- |
|  | (D.36) |

**Exercise D.27.** Find the transpose of

|  |  |
| --- | --- |
|  | (D.37) |

If a matrix equals its transpose, and we hence have , then we call the matrix *symmetric*.

### Matrix inverse

The inverse of a square matrix is the matrix such that, if we multiply it by , we get the identity matrix. So per definition we have

|  |  |
| --- | --- |
|  | (D.38) |

It can then also be proven that also .

Not all square matrices can be inverted. A matrix that can be inverted is said to be *invertible* (or *non-singular*). A matrix that cannot be inverted is *non-invertible* (or *singular*).

A good way to check if a matrix is invertible is to calculate its determinant. A matrix is invertible if its determinant is non-zero. If however , then the matrix cannot be inverted.

For a matrix there is an equation to find the inverse. If

|  |  |
| --- | --- |
|  | (D.39) |

then the inverse equals

|  |  |
| --- | --- |
|  | (D.40) |

For larger matrices such an equation also exists, but the expressions become enormous, so then other methods are usually used to invert matrices. We will talk about those later.

**Exercise** **D.28.** For the matrix , find the inverse . Then check your answer by calculating .

**Exercise D.29.** Verify the generic expression of shown above, by multiplying it by . Do you get ?

### Matrix powers

We have seen matrix addition and multiplication. Do powers also work? What is ? The answer is not surprising, . Just multiply with itself. Similarly, , and so on.

**Exercise** **D.30.** For the matrix , find the square and the cube . What is in this case?

### Matrix exponents

When dealing with numbers, we can also calculate exponents. For instance, if , then we can calculate . But if is a square matrix, what is ?

To calculate this, we make use of the Taylor expansion of the function . (See Appendix E for a brief introduction of Taylor polynomials.) This Taylor expansion equals

|  |  |
| --- | --- |
|  | (D.41) |

For matrices, the matrix exponential is defined along this idea. So per definition

|  |  |
| --- | --- |
|  | (D.42) |

This is an infinite sum which you don’t simply calculate by hand, but computers are very well capable of finding matrix exponents.

**Exercise D.31.** For the matrix , try to guess what will be, based on the above equation. (Hint: use your result from Exercise D.30. Determine each element of individually. Start with the easier ones. Also use the expression that .) If you liked this challenge, then do the same for .

There is one case where we *can* easily calculate a matrix exponent by hand. That is the case when is a so-called *diagonal* matrix: it only has non-zero terms along its main diagonal (from the left top to the right bottom). All other (non-diagonal) terms are zero. So if equals

|  |  |
| --- | --- |
|  | (D.43) |

then we can calculate through

|  |  |
| --- | --- |
|  | (D.44) |

The lesson: in case of a diagonal matrix (and only then!) are we allowed to simply take the exponent of the individual elements.

**Exercise D.32.** For the matrix , calculate .

### Matrix operations in Python

All the calculations we just did can also be performed by Python. Examples are shown in Script D.3. Note that this time the Numpy package is not enough: we are getting scientific enough to also need the Scipy package.

|  |
| --- |
| 1. **import** numpy as np # Import an entire package. 2. **from** scipy.linalg **import** expm # Only import one function from a package. 3. **import** math 5. # Matrix transpose. 6. A = np.array([[ 0,  1   ], 7. [ 2,  3   ], 8. [ 4,  5   ]]) 9. **print**(A.transpose()) # Gives [[0, 2, 4], [1, 3, 5]]. 11. # Matrix inverse. 12. A = np.array([[ 5,  2   ], 13. [ 7,  3   ]]) 14. **print**(np.linalg.inv(A)) # Gives [[3, -2], [-7, 5]]. 16. # Matrix powers. 17. A = np.array([[ 1,  0   ], 18. [ 1,  1   ]]) 19. **print**(np.linalg.matrix\_power(A, 3)) # Gives [[1, 0], [3, 1]]. 20. **print**(A\*\*3) # Gives something else! Python does this element-wise. 22. # Matrix exponent. 23. **print**(expm(A)) # Gives [[2.718, 0], [2.718, 2.718]]. 25. # Diagonal matrix exponent. 26. A = np.array([[ 2,  0   ], 27. [ 0,  4   ]]) 28. **print**(expm(A\*math.log(3))) # Gives [[9, 0], [0, 81]]. Here we used a trick to find 3^A through the expm function: 3^A can be written as (e^log(3))^A which in turn equals e^(log(3)\*A). |

Script D.3: There are various useful functions in Python, but you have to know which ones to use.

## Solving systems of linear equations

One of the origins of linear algebra lies in solving linear equations. Let’s take a look at how that works.

### Solving a system of linear equations the old way

Imagine we have two unknown parameters and , for which we know two relations,

|  |  |
| --- | --- |
|  | (D.45) |

Note that both these relations are *linear*: they do not have any squares, higher powers, sines, logarithms or anything similar. Only multiplication of known values by some and some constant added to it. Hence, it falls under the realm of linear algebra.

We can solve this *system of equations* in various ways. You may be most familiar with the substitution method, where we solve

|  |  |
| --- | --- |
|  | (D.46) |

insert this into the other relation and solve the outcome. Instead, we will use a different method, of multiplying and subtracting equations. To do so, we first multiply the first relation by and the second one by . This turns the whole system into

|  |  |
| --- | --- |
|  | (D.47) |

Next, we subtract the second relation from the first. Note that then drops out! This gives us

|  |  |
| --- | --- |
|  | (D.48) |

We can directly find . Next, we can subtract from the very first relation we started with, . We then get

|  |  |
| --- | --- |
|  | (D.49) |

It immediately follows that . We have now solved the system of linear equations.

### Solving a system of linear equations through matrices

Let’s once more look at the system of linear equations

|  |  |
| --- | --- |
|  | (D.50) |

Using matrices and vectors, we can write this in a shorter way. We can actually write

|  |  |
| --- | --- |
|  | (D.51) |

This equality must hold. To keep things shorter, we often write the above as

|  |  |
| --- | --- |
|  | (D.52) |

with , and defined as shown above. We know and . The challenge now is to solve for .

To do so, we can apply the same method we just did. The notation is just a bit different. First we take the first row of the matrix , multiply it by and then subtract this from the second row. This gives us

|  |  |
| --- | --- |
|  | (D.53) |

Note that we must also apply this to the vector for the equality to still hold. We can simplify the above to

|  |  |
| --- | --- |
|  | (D.54) |

or, by multiplying the second row by , reduce this further to

|  |  |
| --- | --- |
|  | (D.55) |

Effectively, we have already solved for ! We can continue this method further, by subtracting two times the bottom row from the top row. We then get

|  |  |
| --- | --- |
|  | (D.56) |

Dividing the first row by (normalizing it) gives us the final result

|  |  |
| --- | --- |
|  | (D.57) |

Note that we have turned the matrix on the left into the identity matrix . We can hence directly read what the value of is: it’s the vector that is now written on the right-hand side of our equation.

This method that we have applied to solve our matrix equation is known as *Gauss-Jordan elimination*. Specifically, *Gaussian elimination* is the first half, where we turn the bottom left corner of the matrix into zeros. *Jordan elimination* is the second step, where we also turn the top right half into zeros. The full method is sometimes also called *row reduction*.

**Exercise D.33.** Apply Gauss-Jordan elimination to solve the matrix equation

|  |  |
| --- | --- |
|  | (D.58) |

### Solving a system of linear equations through a matrix inverse

Let’s look at the matrix equation again. Another way to solve this, is by left-multiplying both sides of the equation by the matrix inverse . On the left, we then get . However, (per definition) equals , and will equal . We hence get

|  |  |
| --- | --- |
|  | (D.59) |

So if we know the inverse , we can directly find !

**Exercise D.34.** Solve the matrix equation by using **.** (Hint: remember Exercise D.28.)

The big question is: how do we find the matrix inverse ? For a matrix we can use the equation we saw earlier. For bigger matrices that will not be possible. However, we can also find the matrix inverse through Gauss-Jordan elimination.

To do so, we start by writing the matrix that we want to invert, together with the identity matrix . So,

|  |  |
| --- | --- |
|  | (D.60) |

We then apply Gauss-Jordan elimination to the left half, applying exactly the same steps to the right half. So to start, we subtract times the first row from the second row, turning the bottom left element to . This gives

|  |  |
| --- | --- |
|  | (D.61) |

We then normalize the second row, making the coefficient in the left half equal to . In other words, we multiply it by . This turns the matrix into

|  |  |
| --- | --- |
|  | (D.62) |

Next, we subtract twice the second row from the first row to turn the top right element of the left half to zero,

|  |  |
| --- | --- |
|  | (D.63) |

Finally we normalize the top row by dividing by . The final outcome will be

|  |  |
| --- | --- |
|  | (D.64) |

Note that now the left half is the identity matrix , while the right half is the matrix inverse . This is how computer algorithms find matrix inverses, and how we can find them too in the absence of a computer.

**Exercise D.35.** Find the inverse of the matrix using Gauss-Jordan elimination.

**Exercise D.36.** Find the inverse of the matrix . Then calculate .

### Solving systems of linear equations with Python

Python can help you solve systems of linear equations. You can see example code to do exactly that in Script D.4.

|  |
| --- |
| 1. **import** numpy as np 3. # Solve a system of linear equations. 4. A = np.array([[ 5,  2   ], 5. [ 7,  3   ]]) 6. b = np.array([4, 5]) 7. Ai = np.linalg.inv(A) # Short for "A-inverse". Equals [[3, -2], [-7, 5]]. 8. **print**(Ai.dot(b)) # Gives [2, -3]. Note that this is A^{-1}\*b. 9. **print**(np.linalg.solve(A, b)) # Gives [2, -3]. It immediately finds A^{-1}\*b. Because it does not have to find/store A^{-1}, this is faster and more accurate. |

Script D.4: Python has a variety of ways to solve systems of linear equations.

## Eigenvalues and eigenvectors

A very important property of a matrix is its eigenvalues and its eigenvectors. Eigenvalues and eigenvectors are complicated subjects, so we will only focus on the basics. What are they and how do you find them?

### The definition of eigenvalues and eigenvectors

Whenever you multiply a matrix with a vector , then you get another vector as outcome. If this vector is a multiple of the original vector that we used, then is a so-called *eigenvector*. So an eigenvector must satisfy

|  |  |
| --- | --- |
|  | (D.65) |

for some constant value . This value then is the corresponding *eigenvalue*.

**Exercise D.37.** Multiply the matrix by the vector . Is an eigenvector of ? If so, what is the corresponding eigenvalue ?

**Exercise D.38.** Multiply the matrix by the vector . Is an eigenvector of ? If so, what is the corresponding eigenvalue ?

In the last exercise we saw that, if is an eigenvector of , then every multiple of is also an eigenvector with the same eigenvalue. This is always the case, and because of this, we do not count them as different eigenvectors. Also note that the *zero-vector* always satisfies , but we don’t count this as an eigenvector either.

### Finding eigenvalues and eigenvectors

To find eigenvalues, we make use of the above definition. We can rewrite it to ,or alternatively to

|  |  |
| --- | --- |
|  | (D.66) |

It can be proven that this equation only has a solution (other than which we ignore) whenever the determinant equals zero. We can use this equation to find . So if

|  |  |
| --- | --- |
|  | (D.67) |

then we have

|  |  |
| --- | --- |
|  | (D.68) |

and the equation to find the eigenvalues (the so-called *characteristic equation*) follows as

|  |  |
| --- | --- |
|  | (D.69) |

By solving this equation, we can find all eigenvalues.

**Exercise D.39.** For the matrix , find the eigenvalues. Afterwards, use to find the corresponding eigenvectors. (Hint: you can freely choose one element of each eigenvector.)

We see that a matrix of size has two eigenvalues, each with a corresponding eigenvector. This is usually (but not always) the case. Similarly, a matrix of size always has eigenvalues, and these usually each have a corresponding eigenvector. There are cases where eigenvalues do not have corresponding eigenvectors, but these are special cases which we will not discuss here for reasons of complexity.

Eigenvalues do not always have to be real numbers. It could very well be that eigenvalues are complex numbers. (For an introduction into complex numbers, see Appendix B.) Complex eigenvalues have the property that they always come in pairs: if is a complex eigenvalue, then its complex conjugate is also an eigenvalue. The resulting eigenvectors will also be complex-valued.

**Exercise D.40.** For the matrix , first find both eigenvalues. Next, verify that is an eigenvector for one of these eigenvalues. Then find the eigenvector for the other eigenvalue.

### Eigenvalues and eigenvectors in Python

Python has all the toolboxes ready to calculate eigenvalues and eigenvectors of matrices. Example code to calculate them is given in Script D.5.

|  |
| --- |
| 1. **import** numpy as np 3. # Find eigenvalues and eigenvectors. 4. A = np.array([[ 1,  -2  ], 5. [ 1,  4   ]]) 6. l, v = np.linalg.eig(A) # Gives first the eigenvalues and then the eigenvectors. 7. **print**(l) # Gives [2, 3]. 8. **print**(v) # Gives [[-0.89, 0.71], [0.45, -0.71]]. Note that [-0.89, 0.45] is a multiple of [-2, 1], and that [0.71, -0.71] is a multiple of [1, -1]. Python always gives the eigenvectors with length 1. |

Script D.5: An example script to calculate the eigenvalues and eigenvectors of a matrix.

# Taylor polynomials

When working with simulations, it is common to make linearizations. What are linearizations? Or more generally, what are Taylor polynomials? That is the subject of this appendix. Extra text on this can also be found in Section 10.3 of the Engineering Mathematics book.

This appendix starts with the Taylor polynomial of a regular function. We then apply this Taylor polynomial to find derivatives of functions. Finally, we examine Taylor polynomials of vector functions.

## Taylor polynomials for scalar functions

Suppose that we have some nonlinear function like . Can we also write this function as a polynomial in ? That is, we want to write as some polynomial

|  |  |
| --- | --- |
|  | (E.1) |

where ideally equals for every possible input . It turns out that this is often possible, and the resulting polynomial is known as the *Taylor polynomial*. The big question is how to find the coefficients .

### The zero order approximation

The first thing we want to find is . To do so, we insert into the function . This gives us the known value . (For example, if we take , then .) We want to equal . Hence, we must also have to equal . But if we insert into

|  |  |
| --- | --- |
|  | (E.2) |

then all we wind up with is . As a result, we must have .

We could now simply plot the function . This is what we call the *zero order approximation* of . It’s not a very good approximation, as shown in Figure E.1. That is why we usually include more terms.

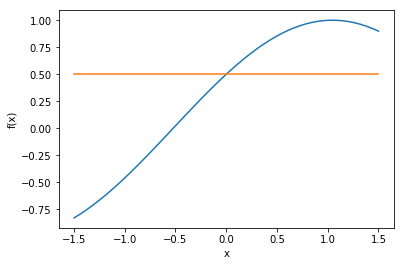


Figure E.1: The function (blue) and the zero order approximation (orange).

### The first-order approximation: the linearization

Next, consider the next coefficient . To find it, we consider the derivative . If the functions and are equal, then their derivatives must also be equal. So must equal the polynomial

|  |  |
| --- | --- |
|  | (E.3) |

Specifically, if we insert again, then we must get the same value! As a result, we must have .

The approximation that we now have,

|  |  |
| --- | --- |
|  | (E.4) |

is known as the *first-order approximation*, also known as the *linearization*, of . It is shown in Figure E.2.

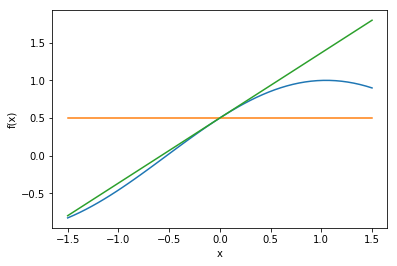


Figure E.2: An extension of Figure E.1, with the first-order approximation (the linearization) added as a green line.

### Higher-order approximations

We can find the remaining coefficients in the same way. For the coefficient , for instance, we look at the second derivative. Since we have

|  |  |
| --- | --- |
|  | (E.5) |

we must have . It follows that , and similarly

|  |  |
| --- | --- |
|  | (E.6) |

The ’th order Taylor polynomial is now given by

|  |  |
| --- | --- |
|  | (E.7) |

Some examples of Taylor polynomials of are shown in Figure E.3. You can see that, as more terms are added, the approximation becomes more accurate.

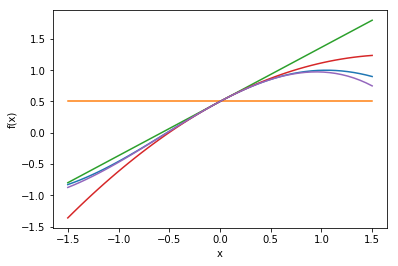


Figure E.3: An extension of Figure E.2, with the second-order approximation (red) and the third-order approximation (purple) added.

**Exercise E.1.** Find the first five terms of the Taylor polynomial of .

**Exercise E.2.** Find the first five terms of the Taylor polynomial of .

**Exercise E.3.** Find the first five terms of the Taylor polynomial of .

**Exercise E.4.** Find the first five terms of the Taylor polynomial of . Is it still the derivative of ?

**Exercise E.5.** Find the first five terms of the Taylor polynomial of .

### Approximations about a different point

So far we have made our approximation about the point . You could see this from Figure E.3: the approximation was only valid around this point. The further away we are from zero, the larger our error becomes. Can we also make our approximation around some other point (pronounced x-bar)?

The answer is yes. The key here is to write our polynomial slightly differently, through deviations from our point . That is, we write it as

|  |  |
| --- | --- |
|  | (E.8) |

We can then apply the same method to find the coefficients. If we do, we wind up with the Taylor polynomial

|  |  |
| --- | --- |
|  | (E.9) |

We could for instance find the Taylor polynomial of around . We then get the result shown in Figure E.4.

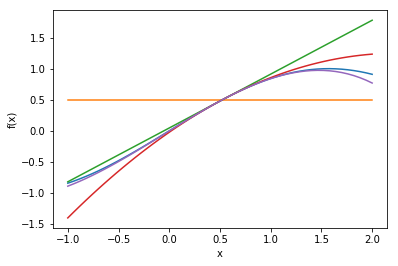


Figure E.4: Various Taylor approximations of the function around the point .

We do need to make one final note about the naming of these approximations. If we take the approximation around the zero point , like we did at the start, then the polynomial is officially called a *Maclaurin polynomial*. If we take it about any generic point (which may even be ) then it’s called a *Taylor polynomial*. The Taylor polynomial is hence a generalization of the Maclaurin polynomial.

**Exercise E.6.** Find the first five terms of the Taylor polynomial of around .

**Exercise E.7.** Find the first five terms of the Taylor polynomial of around .

**Exercise E.8.** Find the first five terms of the Taylor polynomial of around .

### Linearizations of functions with multiple inputs

A function can have multiple inputs. For example, we can have a function depending on parameters and . If we plot such a function, we will get a 3D plot that looks more or less like a landscape Figure E.5.

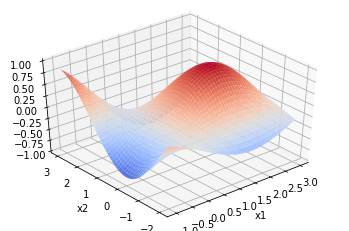


Figure E.5: A 3D plot of the function .

A shorter way to write such functions is through the vector notation. We then write it as , where the vector equals . (For information about vectors, read Appendix D on linear algebra.)

If we take the derivative of a function with multiple inputs, we should indicate with respect to which variable we take the derivative. (See Section C.4 for more background on these so-called *partial derivatives*.) For instance, we can take the derivative

|  |  |
| --- | --- |
|  | (E.10) |

Alternatively, we can take the derivative with respect to all parameters (the *gradient* ) and wind up with

|  |  |
| --- | --- |
|  | (E.11) |

**Exercise E.9.** For the example vector function , find the gradient .

This gradient is very useful if we want to linearize functions with multiple inputs. To do so, we should specify around which point we perform this linearization. The linearization itself then equals

|  |  |
| --- | --- |
|  | (E.12) |

**Exercise E.10.** Linearize around the point and .

**Exercise E.11.** Linearize around the point and .

**Exercise E.12.** Linearize around the point and .

There is a way to visualize this linearization. First of all, imagine the 3D graph of the function as a landscape; see Figure E.5. Then pick one point in this landscape; the point . The linearization now is the flat (but sloped) plane tangent to this landscape: it has exactly the same slope in every direction as the function itself does. Also, if the function is flat, like in one of the peaks or valleys, the linearization is a flat plane. For an example of this, see Figure E.6.

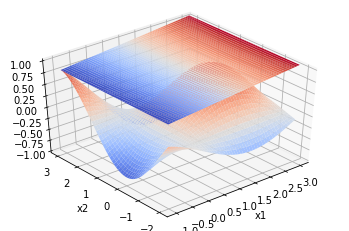


Figure E.6: A linearization of the function about the point , . Note that .

## Finding finite difference formulas for derivatives with Taylor polynomials

Suppose that we are measuring some signal; for example the temperature. We do this at fixed intervals, for example every few milliseconds. This gives us the data points

|  |  |
| --- | --- |
|  | (E.13) |

for some time . We know the value of the function at this time . Can we then also find the derivatives at that time , , and so on, solely based on these measurements? How do we do that? The answer is, with finite difference formulas that approximate these derivatives. These finite difference formulas are also used to convert differential equations, see appendix C, to finite difference equations, such that they can be solved numerically.

### Finite difference formula for the first derivative

The answer is: we can approximate them. The starting point is the ’th order Taylor polynomial

|  |  |
| --- | --- |
|  | (E.14) |

One way to obtain the derivative is to pick the data points and and plug them into the expression

|  |  |
| --- | --- |
|  | (E.15) |

This expression may intuitively make sense to you. It is after all the definition of the derivative: the rate of change of divided by the change in the time . But if we use Taylor polynomials, we can also see why this works in a different way.

To do so, we must insert time into the above Taylor polynomial. This gives us

|  |  |
| --- | --- |
|  | (E.16) |

If we plug this into our so-called *finite difference approximation*, we get

|  |  |
| --- | --- |
|  | (E.17) |

There are a variety of simplifications we can make here. First of all, drops out. Secondly, all remaining terms in the fraction have in them, so we can simplify the fraction. We remain with

|  |  |
| --- | --- |
|  | (E.18) |

Next, we must assume that the time step is very small. If it is sufficiently small (nearly zero) then all terms drop out, and we indeed have

|  |  |
| --- | --- |
|  | (E.19) |

At the same time we should note that, if the time step is not so small, and if our signal is highly curved and hence is large, that this approximation is not so accurate.

### Finite difference formula for the second derivative

We can do a similar trick to find the second derivative . The outcome will be

|  |  |
| --- | --- |
|  | (E.20) |

To see why this holds, we should also insert into our Taylor polynomial. This results in

|  |  |
| --- | --- |
|  | (E.21) |

If we plug this into the above expression, we should wind up with approximately .

**Exercise E.13.** Work out the mathematics described above. Is it indeed true that we end up with ?

### Finite difference formula for higher-order derivatives

Similar expressions can be found for higher-order derivatives. For instance, the finite-difference approximation for the third derivative equals

|  |  |
| --- | --- |
|  | (E.22) |

**Exercise E.14.** Verify that this expression holds.

The pattern that emerges here is strongly linked to the well-known Pascal’s triangle. Using it, we can pretty much guess what the expressions for the higher derivatives will be.

**Exercise E.15.** Look up Pascal’s triangle online. Use it to guess the finite-difference approximation of .

### Using the finite-difference approximations for simulations

In simulations it may occur that we know the derivative . For example, our signal may actually be the position and we know the velocity . In this case we can use the finite-difference approximation the other way around. In that case we can rewrite the finite-difference approximation

|  |  |
| --- | --- |
|  | (E.23) |

into the simulation step

|  |  |
| --- | --- |
|  | (E.24) |

This works for the first derivative. It also works for the second derivative. If we only know , then we can rewrite the finite difference approximation

|  |  |
| --- | --- |
|  | (E.25) |

into the simulation step

|  |  |
| --- | --- |
|  | (E.26) |

Similar tricks can be applied if higher-order derivatives are known.

**Exercise E.16.** Derive a similar expression when using the third derivative. That is, express in the previous values , and , the third derivative and the time step .

## Taylor polynomials for vector functions

So far we have dealt with regular (scalar) functions . We will now expand the theory to vector functions . Do make sure you have read Appendix D on linear algebra prior to reading this section to know what vectors are.

### Vector functions and their derivatives

A scalar function as we have used so far requires a time as input and gives a scalar number as output. A vector function still requires a time as input, but it gives a vector as output. We can write such a vector function as its components through

|  |  |
| --- | --- |
|  | (E.27) |

Just like scalar functions, vector functions also have derivatives. To take the derivative of a vector function, we can simply take the derivative of the individual elements

|  |  |
| --- | --- |
|  | (E.28) |

**Exercise E.17.** For the example vector function , find the derivative .

### Linearizations of vector functions

With scalar functions, it was possible to make a linearization (a first-order approximation) about a certain input . We did so in Section E.1.2 and found

|  |  |
| --- | --- |
|  | (E.29) |

We can do the same for vector functions. The expression we then get is actually identical. It equals

|  |  |
| --- | --- |
|  | (E.30) |

**Exercise E.18.** Find the linearization of the vector function around the point .

### Linearizations of vector functions with multiple inputs

In Section E.1.5 we saw that a scalar function can have multiple inputs. We then write it as or alternatively as . Vector functions can also have multiple inputs. We then write them as or shorter as . Such a vector function still consists of components, like

|  |  |
| --- | --- |
|  | (E.31) |

We can then take a partial derivative with respect to one variable, like

|  |  |
| --- | --- |
|  | (E.32) |

Alternatively, we can take the derivative with respect to all parameters similar to the *gradient*. The generalisation of the gradient to vector functions is a matrix called the *Jacobian* . For the vector function it is defined as

|  |  |
| --- | --- |
|  | (E.33) |

**Exercise E.19.** For the example vector function , find the Jacobian .

Just like we can linearize a scalar function with multiple inputs, we can also linearize a vector function with multiple inputs. First we should specify around which point we want to perform the linearization. The linearization then follows as

|  |  |
| --- | --- |
|  | (E.34) |

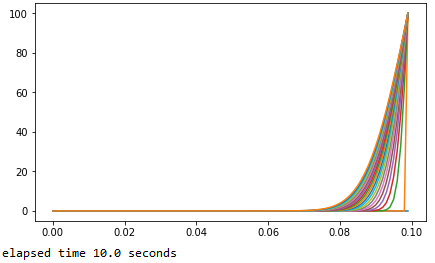
Effectively, this comes down to first linearizing in the way that we previously did, and then separately doing the same thing for .

**Exercise E.20.** Linearize the function around the point , . Write your answer using matrix notation.

# Excercises & answers

**Exercise 4.2** Run the simulation for a different material, stainless steel 304.

The thermal diffusivity of SS 304 is 4.2 x 10-6



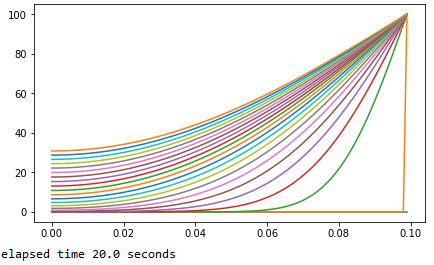
**Exercise 4.3** Change the material back to aluminium. Change the boundary condition in the simulation such that the left end of the rod is totally isolated. Change the step time to 0.002 second.

There are several ways to do this. The easiest way is to change the line:

temp\_profile\_old[0]=0

to:

temp\_profile\_old[0]=temp\_profile\_old[1]



**Exercise** Change the boundary conditions such that the rod is totally isolated. Start with the situation that the middle 2 centimeters is 100ºC and the rest of the rod 0ºc. Change the dt to 0.0005.

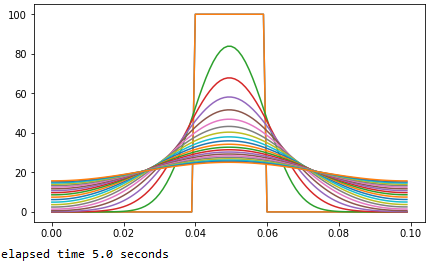
There are several ways to do this. The easiest way is to change the code as follows:

temp\_profile\_old =np.zeros(arraysize)

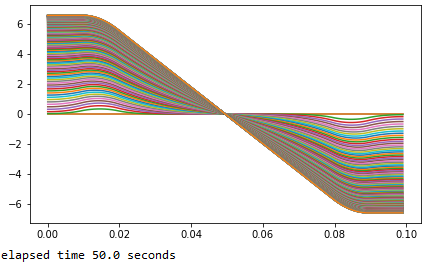
temp\_profile\_new =np.zeros(arraysize)

temp\_profile\_old[40:60]=100

alfa=9.7e-5 # alfa of aLuminium



1) Run the simulation for 10 time as long. What is the t in steady state?



2) Use the Fourier’s law for steady state to check if the simulation is correct.

Which is in rather good agreement with the simulation.

1. Monte Carlo simulations are algorithms that rely on repeated random sampling to obtain numerical results. The underlying concept is to use randomness to solve problems that might be deterministic in principle (wiki, 2020). [↑](#footnote-ref-2)
2. To solve this you can use deepcopy: <https://docs.python.org/3/library/copy.html> [↑](#footnote-ref-3)
3. What should be seen as relatively high or low velocity also depends on the size of the object involved and the viscosity of the air. Which approximations are reasonable in what situations is best captured by the so-called Reynolds number. (van den Akker & Mudde, 1998; Pritchard & Mitchell, 2016) [↑](#footnote-ref-4)
4. This is perhaps not so surprising, since heat conduction is caused by the random motion of the molecules in a solid, while a solute is spread through a solution by the random motion of the molecules. [↑](#footnote-ref-5)
5. As mentioned in a footnote in paragraph 3.2.1, what should be seen as relatively high or low velocity and which approximations are then appropriate can be determined by using the so-called Reynolds number. (van den Akker & Mudde, 1998; Pritchard & Mitchell, 2016) [↑](#footnote-ref-6)