INTELLIGENT SYSTEMS FOR OPTIMAL AND ADAPTIVE CONTROL: EXERCISE TUTORIAL BOOK

Alexander Grantcharov, Phd
(Technical University - Sofia)

Milka Uzunova, Phd
(ECAM-EPMI - Paris)

Sofia | Paris
2025
ISBN: 978-619-91764-4-3

Contents

Chapter I.
Introduction to control theory.

Before we can dive into the deeper parts, that compose a control system, we have to start with a quick overview at the simplest of levels and understand the concepts of control theory.

When thinking of control, any dynamic system can be represented as shown on the next figure (Figure 1), where what the system does, or how the state of the system changes, depends on two sets of input signals. One set is our control signals, and the other set are disturbances, which we cannot control (usually arising from the environment the system is operating in).

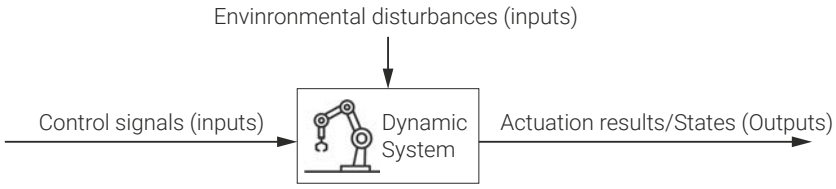


Figure 1. The highest level of abstraction of a control system

A classic example would be the process of driving a car. In a very simplistic version, pressing the acceleration (or brake) pedal affects our speed, and turning the steering wheel affects the car's direction. Those would be our control signals. On the other hand, the road might curve, might have a gradient, or there might be wind, or bumps and potholes. These would be environmental disturbances that would affect the car's direction and speed as well, but we have no control over them.

If we have the task to automate the process of driving the car, and make the car autonomous, we can start by asking ourselves the question: "Do we need to actively measure how the car is performing, as an output, in order to determine what control signals to give it?" And depending on how we answer that question, we are going to end up with one of two possible types of control systems:

I.1. Open loop control (feedforward).

The first concept we have to discuss is open loop control, shown in Figure 2. In this case we are not measuring the output (or the change in the state) of the dynamic system – for us that would be the car and its speed and direction, and we only know what we want them to be, which is our reference. So, we give that reference to the control system, and then it provides the necessary control signals. Since the control signal is fed forward to the dynamic system we want to control, and we have no measurement of what the response is, we call that control type "Open loop" or "Feed forward".

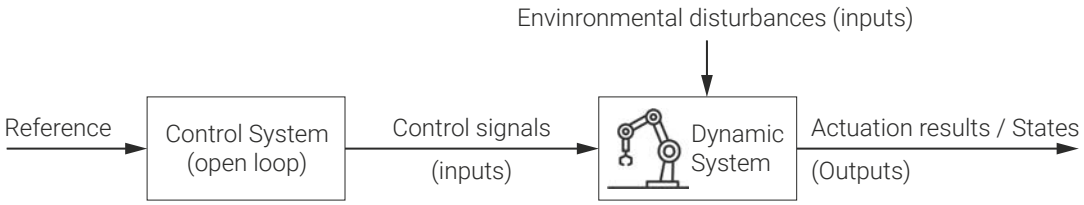


Figure 2. A schematic of the operation of an open-loop control system.

For our example with the car, let's say we want to drive the car forward at a constant speed. That means we need the steering wheel to be kept straight, and the acceleration pedal kept pressed to some level.

If we have to introduce some complexity to this example, we could ask, what if we want to keep the speed at a specific level – for example 50 km/h. We could still argue that is achievable with an open loop control, but now we have to know at what level of the acceleration is this speed achieved. That way, we will know exactly how much to press that pedal.

In order to do that, we have to have understanding of how the car works and what its responses would be for various inputs. Think of this knowledge as a function, a mathematical expression (often a system of expressions) describing the relationship between the inputs to the system and the expected outputs (changes in the system states).

Let's assume we know what is the function that governs the change of speed in respect to the position of the pedal:

$$speed = f(pedal) \qquad (I.1\text{-}1)$$

So, now we can inversely program that function in our open loop control system. We will know exactly how much to press the acceleration to get the car going at 50 km/h, because now we can feed in that speed, and as a result we will get what the pedal position should be:

$$pedal = f^{-1}(speed) \qquad (I.1\text{-}2)$$

Of course, this is a highly idealised scenario. It is fairly obvious that we can only implement open loop (feedforward) control only for systems which are robust and stable and for which we have extremely good understanding of their dynamics. A car is not such a system. In reality there are way too many variables affecting the speed and the direction over which we have no control, and which we cannot predict accurately enough in order to make them part of the reference model. In these cases, we have to start thinking of actively measuring the output of the system, as we are giving it control (input) signal, in order to correct the deviation from what we are expecting it to do, based on the reference.

I.2. Closed loop control (feedback).

Such systems, where we close the connection between the input and the output, by actively measuring the resulting state of the system we are controlling, are called closed loop control systems. We also call them feedback control systems, because effectively we feed the output back into the controller as in input – hence feedback. Just as shown on Figure 3.

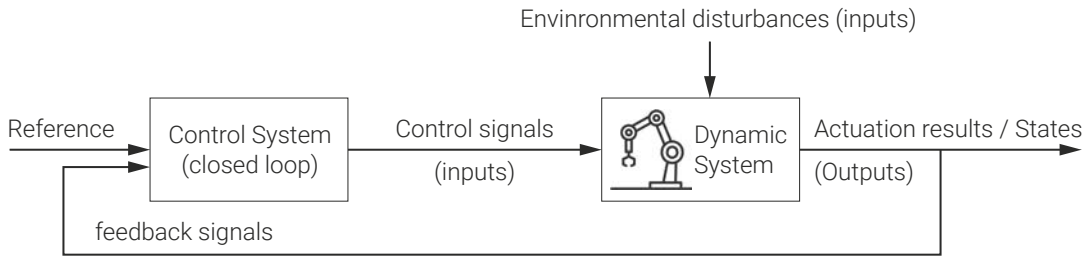


Figure 3. A schematic of the operation of a closed-loop control system.

With the existence of that feedback loop, we can see when the system deviates from the reference values (either because of the disturbances from the environment, or due to inaccuracies in our reference model). Those differences form an error, which we can use to self-correct.

Most dynamic systems, and most control applications, will require a feedback control. But with the increased flexibility that the feedback loop gives also comes high risk of destabilising the system we are controlling. The reason for this is that with feedback the system's output (future) state changes as a function or the current state:

$$\widehat{state} = f(state) \qquad (I.2\text{-}1)$$

There are many ways a feedback control system can be designed, but as we said, all of them require good knowledge of the domain and the potentially the dynamics of the system we want to control. We can classify control systems, based on specific requirements or techniques for dealing with such requirements. For example, we can classify them as Linear or Non-linear systems, Continuous or Discrete, SISO (single in, single out) and SIMO (single in, multiple out) or MISO (multiple in, single out) and MIMO (multiple in, multiple out), Uncertain or Robust, Time-Variant or Time-Invariant, Reactive or Predictive, and so on.

But regardless of the type of the control system and its field of application, we can always look on the inputs (control signals) and the outputs (states) of the system from a data flow perspective. As one way or another, all control systems rely on data to operate. It can be dynamically measured (especially in the case of a feed-back or closed loop control solution), or the data could have been collected and pre-processed in order to obtain the reference model that the system will use (in the case of a feed-forward or open loop control solution).

So, on a very abstract level if we consider the data flow, a control system would need to have a way to collect data, then some way to process it, a decision-making part, and finally, an optional actuation module. And of course, a feedback loop (as part of the data collection) if it implements closed loop control.

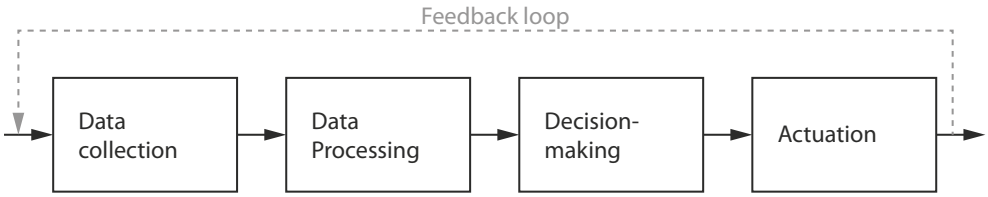


Figure 4. A high-level abstraction of a control system

This exercise book is meant to help you understand and build some simple control systems or the algorithms used for the data-processing and decision making. Especially when talking about intelligent and predictive control and will cover to some extend the data collection and the actuation possibilities.

Chapter II.

Machine learning, intelligence, optimality & adaptability: multivariable control systems application.

Machine learning is the collective term encompassing various tools from informatics, statistics and other science fields (mostly part of data science) that deals with building methods (algorithms) to create computers systems that are able to learn and adapt without following explicit instructions (programming). This is mostly achieved by drawing inferences from patterns in data.

Machine learning is considered to be part of artificial intelligence as a scientific field. Often you might here the use of the term artificial intelligence applied to some software or cyber-physical product. You have to know that humanity is still rather far from creating a true (general) artificial intelligence, and in all cases when this term is used, it is actually meant to describe some form of Machine Learning (ML) that is used for deriving the used internal logic in a system.

And when talking of Intelligence in systems, what usually is meant is that the system is able to mimic (approximate) the decision-making logic of a person (an expert) in that field, or it has predictive capabilities, similar to how an expert might use intuition to make a decision in a state of uncertainty and lack of information. Which is why you can also see these systems being called expert systems. Not all expert systems can be considered intelligent, but almost all intelligent systems are expert systems by design. They might be simple (expert in one specific domain) or complex (expert subsystems of different domains chained together). Both the machine learning and the implementation of "Intelligence" concern the data processing part of a control system.

II.1. Main types of data processing tasks

There are many ways a control system can acquire data or act upon a decision, but what sets apart these systems is their data processing and decision-making logic, which is dependent on their purpose. There are few operations, which are at the heart of the control system’s data processing and decision-making modules. It is not necessary for all of them to be present in a system, but it is almost always some combination of them:

II.1.1) Forecasting / Prediction

Forecasting is the process of using historical data to inform a future outcome. Forecasting is heavily used in any industrial or business environment. Forecasting demand, customer behaviour, resource expenditure, physical process development, etc. In a broader sense forecasting is usually applied on a particular parameter (or group of parameters), which are in some way related to each other, and it is usually in respect to a specific process.

Of course, forecasting is another word for predicting (the future) and we all understand that even the simplest of processes will have plenty of factors that will affect how they develop in time. That is the reason why predicting the future (exactly) is impossible, but if we make sure we account for enough of the major factors, and make some necessary assumptions, we can get an approximate forecast. And that is essentially what any forecasting algorithm in a control system does, it attempts to approximate the process by using measurements of the constituent factors in a formulated relationship.

II.1.2) Classification

Classification is something we do all the time. It is the process of assigning things to groups (or labelling them) according to some common factors, for example animals vs. plants, adults vs. children, red squares vs. green circles, good vs. defective items. As long as we can determine which factor (or group of factors) characteristically describe the given class of objects/subjects, then we can formulate that class, and teach an automated system to recognise those characteristic factors (also known as features), regardless of what data types we use to describe those features.

II.1.3) Clusterisation (Clustering)

Clusterisation, is another form of grouping. Unlike classification though, with clusterisation we usually don't know which of the parameters, that can be used to describe an object, are the characteristic ones. In a sense, we don't have a prior knowledge of what differentiates one cluster to the other. Even more, those objects my not cleanly fall into specific classes, but applying clustering to them will tell us how similar are they to one another.

II.1.4) Optimisation

Optimisation, in a more general sense, is the process of finding a solution that uses certain resources in the best possible way (the most efficient way, given a desired outcome). In a mathematical sense, it is the process of finding the minimum or maximum of an objective function. You can think of the objective function as a formulation of the desired outcome, which depends on various parameters (as arguments of that function), and the task is to find what combination of values for those parameters will yield the minimum or maximum value for the outcome.

Optimisation, as a mathematical tool, is used in all of the previously mentioned processes. For example, finding a function that describes the historical data, in order to use it as a prediction (forecasting) model is achieved through some form of optimisation. Making a classification model to learn the features that can be used to recognise one class from another also uses form of optimisation. Clustering, being usually reduced to a geometric task, where each sample is defined by its factors in a multidimensional space, uses optimization to find those samples that that have shortest distances between them, which is also done through optimisation. We will go through a few different optimisation algorithms, regardless if they are used as a stand-alone tool or as part of another tool.

II.1.5) Decision-making

It is exactly what is sounds, the process of making a decision. Some decisions are easy, if you have a binary logic where if some condition is met then some action should happen. But rarely in the real world we can we depend on exact values. We live in a probabilistic universe and each measurement inherently comes with a degree of uncertainty. That uncertainty becomes even larger when talking about predicted values.

In order to control anything, a control system will have to have some decision-making logic. Depending on the type of control system, and the data it uses (and the data processing algorithms), the decision making can be simpler or more complex. But at the end of day this is what will dictate what control signal comes out of the control system.

Chapter III.
Mathematical modelling

You should have noticed that we mentioned models and formulation when discussing the different kinds of algorithms that might be part of a control system. It should be clear, that by formulation we mean the construction of a formula, a mathematical expression, that comprises of different parameters that describe the specific process we are considering. We can almost interchangeably use the term "model", "formula" and "function" when talking about the mathematical representation of a process. Of course, we have to keep in mind, sometimes a process might be too complex, and the model of the process might have to be expressed (or formulated) with more than one or function.

Regardless of the type of control system we choose, it will inevitably use some mathematical model to base the control signals on. Generally speaking, there are two ways to create a mathematical model of a process (or a dynamic system whose state changes with time). We can either construct one from first principles, which in mathematics is also known as axioms, or we can use data describing the behaviour of the system and try to fit a function that most accurately approximates those data points. That way we will have an approximation of the behaviour of that system.

III.1. Knowledge models, using first principles (axioms and the laws of physics)

If we get back to our car example from Chapter I, we can model the car in a very simplified way, by representing it as a mass with attached wheels, using a spring and a damper, representing the suspension. For even more simplicity we can also limit the system in only one plane (Figure 5).

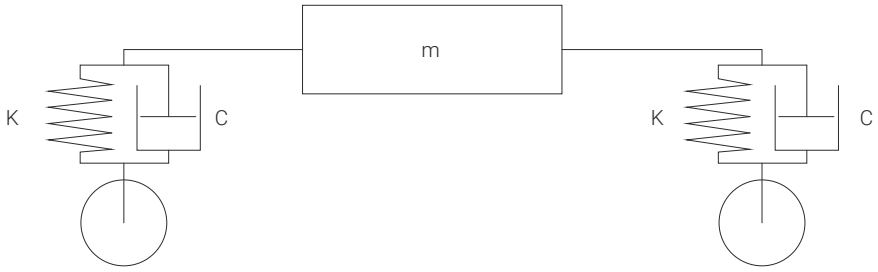


Figure 5. A simplified representation of a car in two dimensions.

Knowing Hook's law (III.1-1) where the force (*F*) equals the spring constant (*k*) times the displacement (*x*), describing the relationship between the force and the behaviour of elastic elements, and Newton's second law (III.1-2) describing the relationship between force, mass and acceleration, we can write down the equations that govern the displacement of the car. We can also analyse its speed and acceleration, as they are the first and second derivative of the displacement.

$$F = -kx \qquad \textit{(III.1-1)}$$

$$F = ma \qquad \textit{(III.1-2)}$$

But there is another way. If you either don't know what the system is comprised of, or it is too complex to be described by a few laws of physics, you could try to approximate its behaviour.

III.2. Behavioural or data-driven modelling, using System Identification

Sometimes processes are too complex to try to find the underlying physical law that governs every little element in them. Sometimes, we just want one single function, that would account for all the complexity, and approximate the process if not perfect, just well enough. That means we have to extract (or identify) the model from the way the system behaves. That usually involves taking measurements of the various states (outputs) of the system for different inputs, and then fitting a function to those results. That process is called System identification, and it involves curve fitting in some parametric space. And when talking about curve fitting, that is an optimisation process.

Let's continue the car example, but this time assume that we don't know any parameters of the car (like the mass or the elastic constants of the spring, and so on). Instead, we have an actual car ready, and we can just drive it. If we want to derive the function that dictates speed in respect to the position of the accelerator pedal, we could just take measurement of how much we have pressed the pedal and what speed the car manages to achieve. We can do that for many different positions of the pedal. If want to keep things simple, we can lock all other parameters of such a system. For example, we will only concern ourselves with a perfectly flat road with no gradients and no bumps and holes, and no winds, etc. And of course, we are completely ignoring the temporal element in the data (how long did it take at this angle of the pedal to reach the steady state). If we plot the recorded data points, we might get something like this:

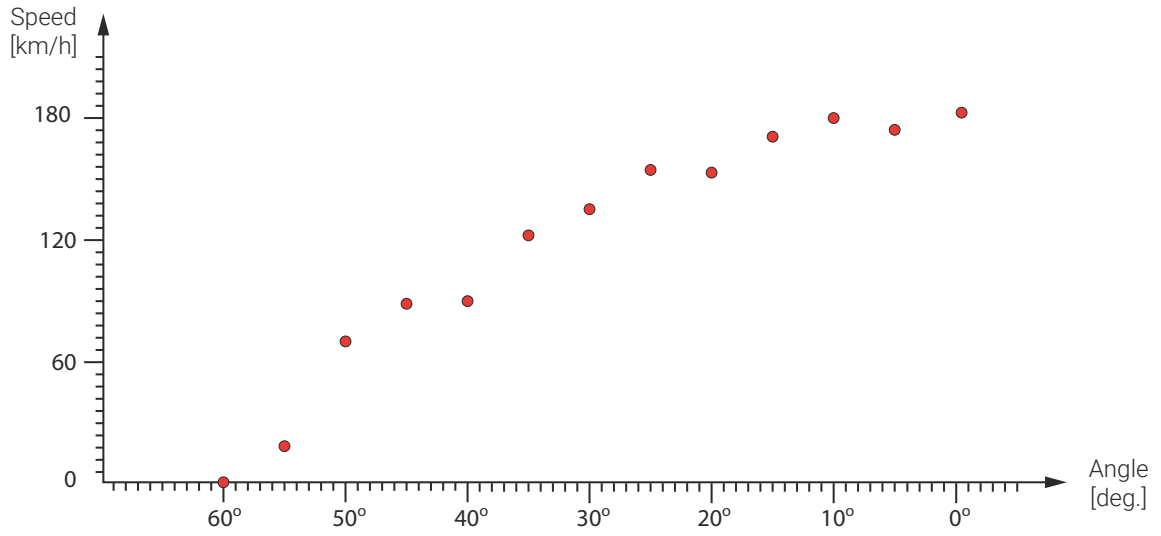


Figure 6. A figurative example of what those accelerator-pedal-angle to speed data points might look like when plotted.

We can see that at the maximum angle (the pedal has not been pressed yet) the car is stationary, and the speed is 0 km/h. When pressed to a maximum (pedal to the metal), at 0 degrees, the speed we get flats out at about 180 km/h. So now, if we want to build a system that controls the speed of the car, we just need the function that dictates the relationship between the pedal angle and the speed, and we could implement some actuation device that presses the pedal the necessary amount to get the desired speed.

To get that function we just need to find which one fits best to the data we have. We want something that rises fast and then flats out as we reach the limit of the car (where even if we could press the pedal more, the car wouldn't get any faster). For example, an exponential function like this:

$$Speed = A\left(1 - e^{\left(\frac{Angle-C}{B}\right)}\right) \qquad (III.2\text{-}1)$$

where *e* is the Euler number and A, B and C are the constants that define the specific shape of the curve (A and B need to be positive). Using a software with curve-fitting capabilities, we would get the specific curve that fits those data points. That means we will know the exact A, B and C that we need to use in our function, so that if we input the specific *Angle*, we will get the correct *Speed*.

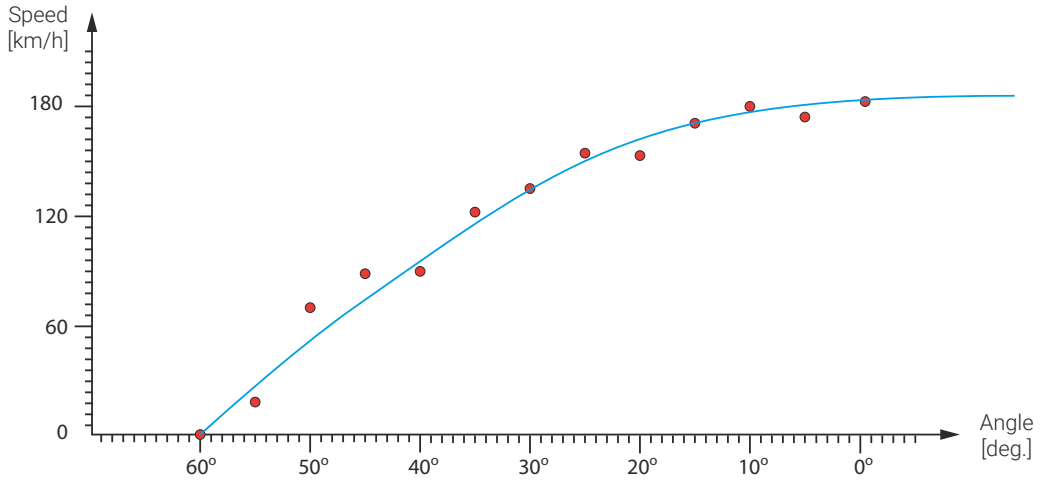


Figure 7. A figurative example of what the fitted curve (in blue) of an exponential-plateau function could look like when applied to the example data.

Both a knowledge-based and a data-based model of a system can be used successfully in a control solution. Choosing which approach to take will very much depend on the type of system and its complexity, and on the knowledge or data that we possess, or can acquire about that system.

Before continuing we need to make a clarification. If someone analyses the text of this exercise book, I'm sure the word system will have the largest occurrence. You need to understand that a system is generally any set of things that work together as part of something larger. So pretty much anything can be viewed as a system. And any system can be comprised of other sub-systems, and at the same time be part of a larger system. It is entirely up to you to decide where you draw the line of control. What exactly you want, need or can control. The process of establishing the boundaries of the system you want to control, and the limits of the control system, is called scoping. If you don't put limits, you will quickly find that any system is a rabbit hole that runs overwhelmingly deep.

But of course, in order to scope the control system, and then develop it, we need to understand the actual system (process) that it will control. When we talk about systems that need control, we are usually talking about dynamic systems (that change in time), or in other words – processes. A process might be physical (concerning some material flow), which is typically the case in an industrial environment, or it could be purely virtual (digital information flow).

In order to understand the process, we need to execute it multiple times, to examine and analyse it, preferably as it is running. The same way we "acquired" the data points for the car speed. That way, we get better and in-depth knowledge, and can better model it, which means we can also better scope the control system. As you can imagine, the more you do something the better you become at it. But see the loop here? We need to understand the process in order to control it, but we need to control it to run it, so we can understand it... In real-life this is called a trial-and-error approach. But when talking about industrial installations, trials and errors cost money. Lots of money. This is why we rather use prototypes and simulations.

Chapter IV.
MatLab / SimuLink Models and Simulations

Simulations allow us to make a virtual representation of a process and run it instead of the real one. Any mistake we make we can correct by modifying the model and rerunning the simulations. In this exercise book, most of the simulations we will run, we will build in MatLab, and more specifically in Simulink, a package especially created for that purpose.

In Matlab terms, and generally in industrial terms as well, when talking about process simulation we distinguish three main groups:

- **Energy flow (electrical):**
  (Motors, energy production systems, positioning and speed control, power control etc.);

- **Fluid flow:**
  (hydraulic or pneumatic pumps, fluid level control and inflow speed/quantity control etc.);

- **Thermo-dynamic:**
  (heat exchangers, heating and cooling control, temperature control, etc.).

That classification is related to the physical process and the corresponding mathematical representation of the physical process. The setpoints of the process (what we referred to as input reference in our Control Theory explanations) and the outputs, depend on the entire installation. That means all the sensors and actuators, and everything else that is part of plant that needs to be controlled, as well as the mathematical model that describes the relationship between those elements (the signals in the system). A summarised presentation of these control systems (either as open or closed loop control) is presented on Figure 8 below.

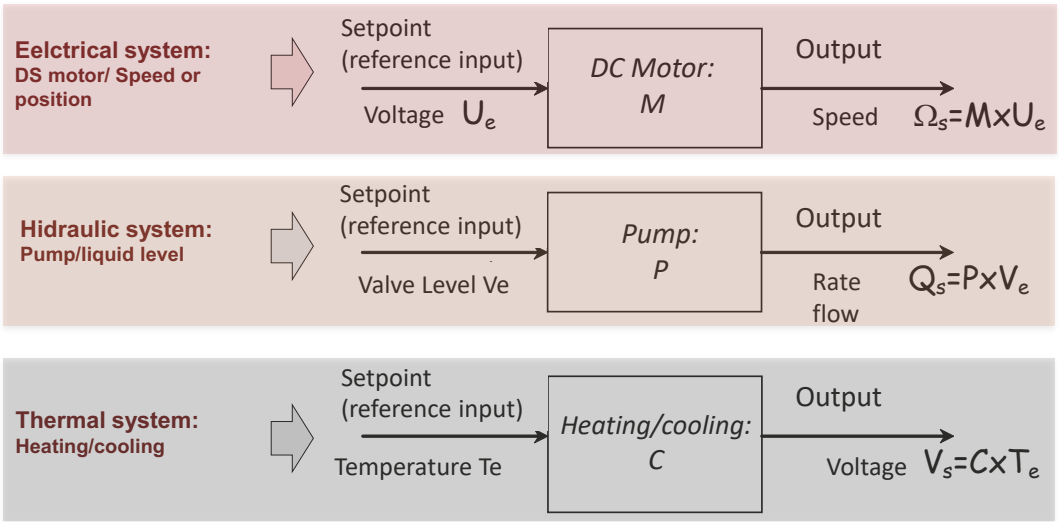


Figure 8. Control systems and parameters for electrical, hydraulic and thermal plants. The dashed line represents the closed loop version of the control diagram.

In the following, we will develop that examples when we identify and validate the obtained system (in open loop system) and when we apply a control strategy on plant (close loop control system).

IV.1. How to model

And in order to understand the process, one of the first things you can do is to build a graph, connecting all the steps of the process, accounting for all of the dependencies and relevant resources needed on every step. This process is called Process mapping.

A graphical presentation of the methodology is presented on Figure 9. This methodology helps us not only to understand the process, but also to obtain the mathematical presentation of the real process/plant (using either physical laws and axioms, element specific relations, or system identification procedure). All this with the aim to apply a control strategy (to close the system) such that we have stable and reliable performance, optimal control and a functional decision-making within the control system. Here are the steps in the process mapping procedure:

- **Create a process flow diagram:**
  if we have enough information and knowledge on the real system. For example, in an electrical system, when after applying the laws of conservation of energy, or current/tension relations, we can obtain the mathematical model of the system. Those are differential equations or some known transfer function.

- **Linearisation if necessary?**
  The first step leads to the definition of the mathematical model (polynomial function, matrix representation, differential equation, etc.). If the system seems to be non-linear, this is the time to choose to linearise it or not, which has its implications for the further steps in the modelling and simulation.

- **Parameter Identification:**
  Making sure all the parameters of the physical system are accounted for. If we continue the example of an electrical system from above, those would be: voltage, resistance, capacity, etc. For a hydraulic system they might be: geometry and volume of the tank, pressure, etc.

- **Model validation:**
  At this step we want to make sure the model makes sense, and when run produces meaningful output. We do this in the software environment of our choosing. In our case that would be MatLab. Here we study the system behaviour in an open loop (steady state) control environment, using step response.

- **Analyse and modify:**
  After the validation step, if the model is not conforming to the real system, we obviously need to analyse what is wrong with it. Did we model it wrong? Did we miss to identify and include a variable/parameter? If the simulation is producing unexpected results, we can go back to previous steps to correct our model and re-run the sim.

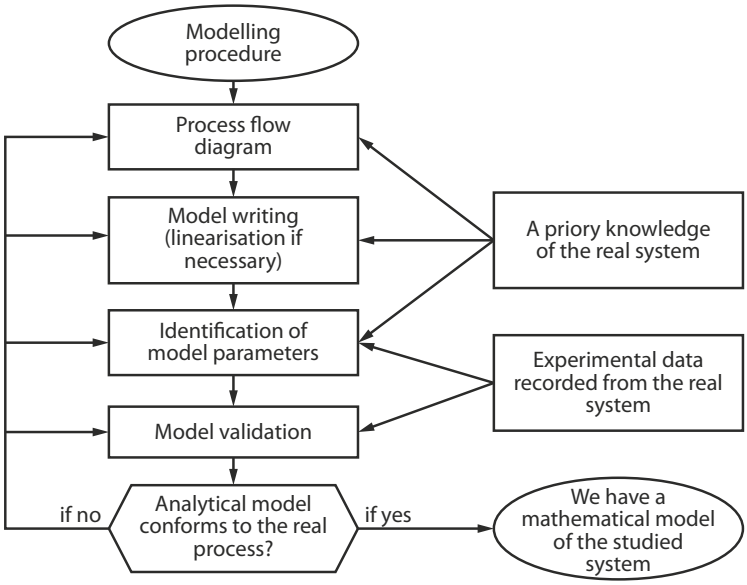


Figure 9. Process mapping methodology steps.

As discussed in the previous chapter, determining the mathematical model for the process (or for more complex processes - any separate part of the process) can be achieved in two ways, either by following first principles and the laws of physics (Knowledge Model) or by System Identification (Behavioural model or Data driven model).

IV.2. How to model processes and systems

For a linearly behaving system (or linearised discrete system), for which we know a priori the mathematical equation (knowledge model), we will need to set the transfer function of the process/system/plant in order to create a block diagram in MatLab's Simulink. We can then use that visual model to study the performance of the system. The analytical modelling tool, used for such cases is the Laplace transform. It is used for the resolution of differential equations, calculation of the transfer function of systems, system response analysis and analysis of the performance of systems. The resolution process is presented in Figure 10 below.

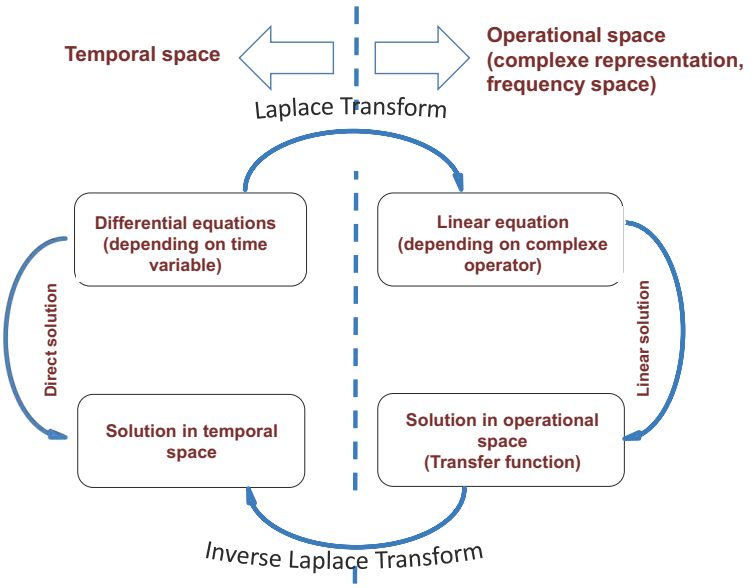


Figure 10. Process mapping based for "knowledge model" using Laplace Transform analytical modelling tool.

Here are some other blocks we can use in our Simulink diagrams, and what they do in regard to our mathematical modelling of real-world processes:

Proportional relation (constant):
Many systems, such as speed reductors, potentiometers, sensors, electrical resistors, etc., are modelled by a simple proportionality relationship between the input and the output, called component Gain.

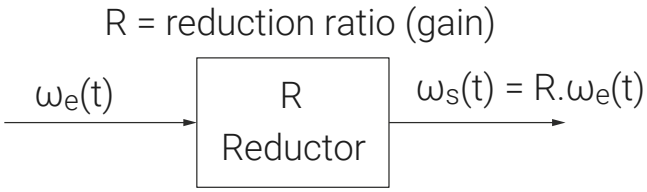


Figure 11. The Constant building block (Gain), used to model proportional relationship. What it does is to simply multiply the input by a constant value (the gain). In the case of a reductor that would be R – the reduction ratio.

IV.2.1) Integral/differential relation (1st order equation)

Another example is the stepper motor, house heat transfer, tank system or some filters – the behaviour is similar to a Resistor-Inductor (RL) system or a Resistor-Capacitor (RC) electrical system (circuit). Using a knowledge model (based on physical laws) and a Laplace transform, we can obtain the following example:

Step 1 – **Choose a real-world system** – for example a stepper motor, similar to the one shown in Figure 12:

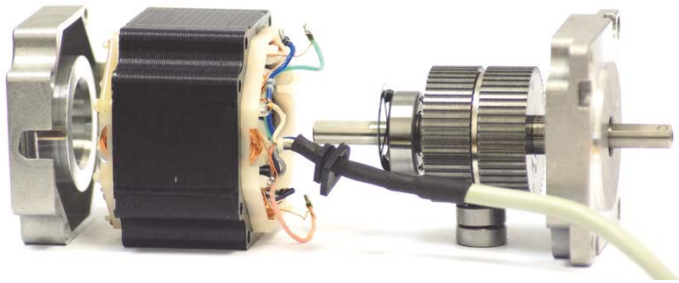


Figure 12. Example of a stepper motor model.

Step 2 – **Create a knowledge model**. As we discussed for a stepper motor that could be an RL (Resistor – Inductor) circuit, like the one shown in Figure 13:

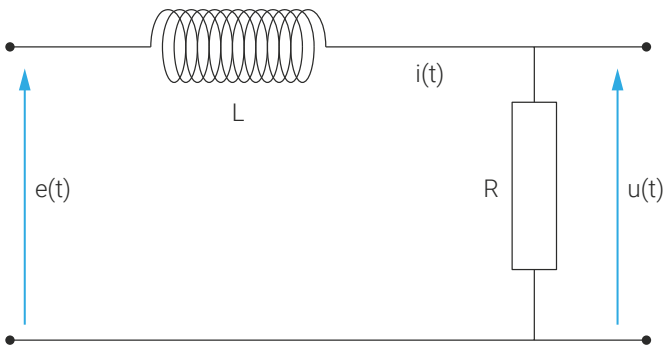


Figure 13. RL circuit as an electrical representation (model) of a stepper motor.

Step 3 – **Create an open loop control system** using the supply voltage (in respect to time) e(t) as an input to the system, and the resulting voltage (in respect to time) u(t) as the output (state) of the system (Figure 14):

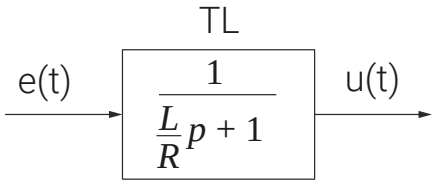


Figure 14. Graphical representation of an open loop control system, based on an RL circuit model.

Step 4 – **Validate and study the model**. We can do that by simulating a 12-volt (tension) step input for some amount of time, and then recording and studying the step output of the system (Figure 15):

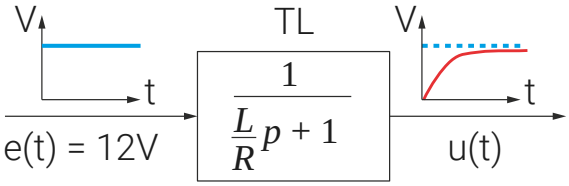


Figure 15. Graphical representation of the simulation of the open loop control system, based on an RL circuit model.

The transfer function, which gives the relationship between the input and output, that we have to use for this model is a differential equation of the 1st order, which as we previously discussed can be introduced in the simulation with a constant gain component, such that:

$$TF(p) = \frac{U(p)}{E(p)} = \frac{\mathrm{k}}{\tau p + 1} \qquad (IV.2\text{-}1)$$

where p is the input parameter, k is the static gain of the system and τ is the time constant. The relations given bellow with (IV.2-2), (IV.2-3) and (IV.2-4) represent the analytical solution (explained in Step 2 above) of the modelling, based on the LT (Laplace Transform) tool.

$$e(t) = L\frac{di(t)}{dt} + u(t)$$

$$u(t) = Ri(t) \rightarrow e(t) = \frac{L}{R}\frac{du(t)}{dt} + u(t) \qquad (IV.2\text{-}2)$$

$$LT \Longrightarrow E(p) = \tfrac{L}{R}pU(p) + U(p) = U(p)\left(\tfrac{L}{R}p + 1\right),$$
$$\text{where } p = jw - complex\ operator$$

Then, the transfer function (the model of the real-life system) is denoted as TF. Given the necessary parameters of the operational space as arguments, it gives an operational solution, as follows:

$$TF(p) = \frac{U(p)}{E(p)} = \frac{1}{\frac{L}{R}p + 1} \qquad (IV.2\text{-}3)$$

where we took the equation from (IV.2-1) and replaced the gain k with 1, and for the time constant τ we use the quotient of the inductance and resistance values of the components in our RL circuit. And finally, if we want to see the evolution in time, we need a temporal solution:

$$ILT \Longrightarrow \frac{U(t)}{E(t)} = e^{-\frac{L}{R}t} \qquad (IV.2\text{-}4)$$

IV.2.2) Integral/differential relation (2nd order equation)

Examples for systems that require 2nd order differential equations for their modelling could be a DC motor or a mass-spring-damper system (the same car suspension example we used earlier). Let's break the process into steps again.

Step 1 – **Examining the real-life system of a car wheel and its suspension**, similar to the depicted in Figure 16.

Figure 16. Illustrative example of a car suspension, including the wheel, the spring-damper set and part of the axel. We can imagine the mass of the car attached on top of it.

Step 2 – **Creating a knowledge model**. We already did that once, remember Figure 5? We can simplify it further to just one wheel and attach the mass on top, so that we get Figure 17. Note that we simplified the real-world system even more, by only taking the displacement in the x direction.

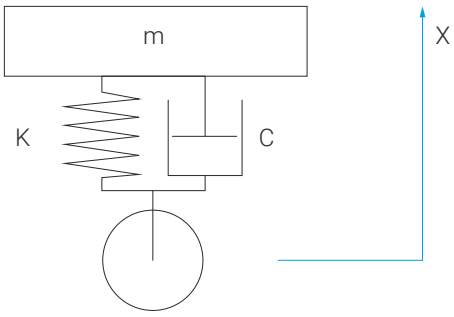


Figure 17. simplified kinematic model of the mass-spring-damper system that a car can be represented with. We ignore all other degrees of freedom except the linear travel in the x direction (the suspension movement).

Step 3 – **Derive the equations that govern the relationship** between the components and write them down. As we discussed in the beginning of the chapter, for a mass-spring-damper

system that would be Hook's law (III.1-1) and Newton's second law (III.1-2). Based on them we can write the following:

$$\begin{cases} \ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 = u \\ where: \omega_n = \sqrt{\frac{k}{m}}, \qquad \zeta = \frac{c}{2m\omega_n}, u = \frac{F_{external}}{m} \end{cases} \tag{IV.2-5}$$

This system represents a second order differential equation where we have derived the equation of the motion, and we found the sum of the forces on the mass m. On one side we have the external force on the mass u and from other side the undamped natural frequency and the damping ratio. In this relation the undamped natural frequency (natural angular frequency) is presented by ω_n, when we have zero driving force with mass equal to m and spring stiffness k. The damping is related to the loss of energy of an oscillating system due to dissipation.

in the control theory (as in this system) the damping ratio ζ characterizes the effect of oscillations, as introduced by the spring in the system. For the second order differential equation the damping coefficient (ratio) can have values from 0 to 1, when the system is underdamped, and the solution will have oscillatory component. Systems with a damping ratio equal to 1 or higher are overdamped, namely without oscillation.

This approach of modelling of the system with second order differential equation is also applied to electrical and other systems. With that said, if instead of the mass-spring-damper system we had chosen the DC electric motor to model, those 2nd order differential equations would have looked like this:

$$\begin{cases} K_t i_a - J_m \frac{d\dot{\theta a}}{dt} - b_m \dot{\theta a} - T_L = 0 \\ where: \frac{d\dot{\theta a}}{dt} = \frac{1}{J_m}\left(K_t i_a + b_m \dot{\theta a} + T_L\right) \end{cases} \tag{IV.2-6}$$

The physical parameters of the DC electric motor are the following: $J_m$moment of inertia of the rotor, b_m motor viscous friction constant, K_t motor torque constant, T_L the applied torque. This torque produces an angular velocity according to the inertia and the friction of the motor and load. Comparing the mechanical and electrical domain, we can observe that the applied mechanical power (torque) corresponds to an electrical power dissipated by the electro-magnetic force in the armature of the motor. The motor's mechanical inertia is equivalent to an inductance and the friction is equivalent of a resistance component.

Step 4 – Build a model in a software environment and simulate it. To do that we have to learn how to work with MatLab and the SimuLink modelling and simulation environment. Which is exactly what we are going to do next.

IV.3. MatLab and the SimuLink environment

To build a block diagram system model, and to make a computational analysis in the MatLab environment, we need to first learn how to create SimuLink models, add SimuLink blocks and set their properties. And finally, how to run the simulation and record the results.

Below is quick tutorial on how to do all of that for any the following exercises in this chapter:

IV.3.1) Start Simulink

Simulink is a block diagram environment for multidomain simulation and Model-Based Design in MatLab. It allows you to place blocks on a canvas, representing various parts or operations in your model.

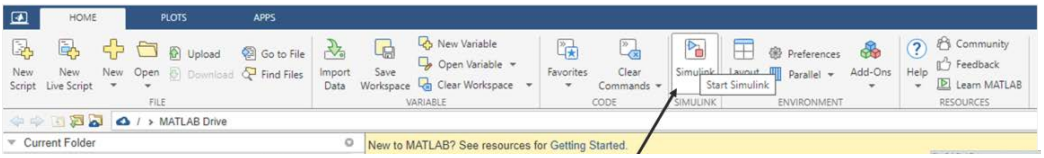


Figure 18. Clicking on the Simulink icon in the MatLab's toolstrip will launch the package.

IV.3.2) Create a BLANK MODEL.

Although Simulink comes with many templates, for various systems and scenarios, our simple model doesn't require much initial structure or settings, so we will learn how to build things from scratch. We can start with a blank model and go from there:

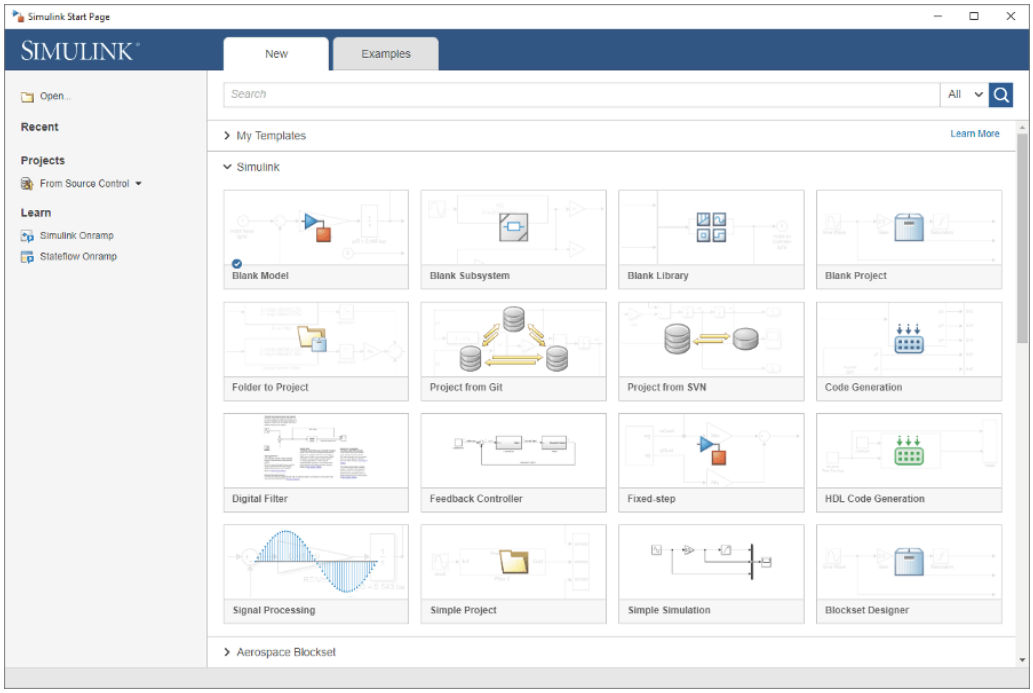


Figure 19. From Simulink's start page select the first option Blank Model.

IV.3.3) Add the transfer function

The transfer function is one the most important blocks. In most cases it is the de facto model that we use to feed the input signal in and get the resulting system state (output) out. This block is found in the model browser.

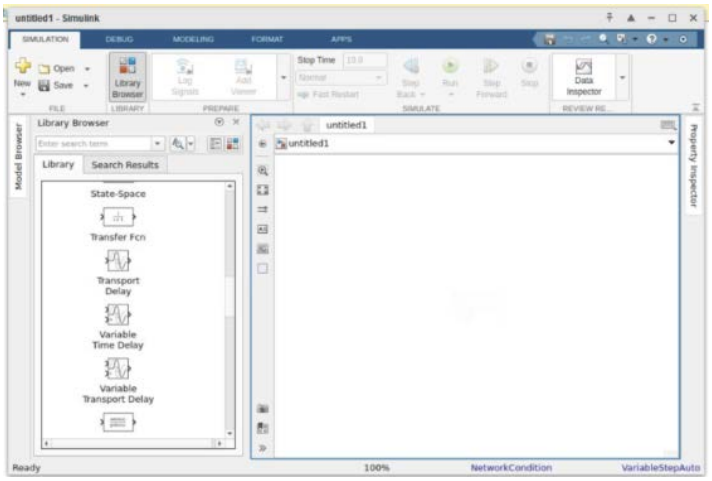


Figure 20. Simulink's library browser allows you to drag and drop a wide variety of transfer functions. You can use the search to find the one you need.

Then open the LIBRARY BROWSER and choose **Simulink** ⇒ **Continuous**
Then drag the block named **Transfer function** into the blank model workspace.

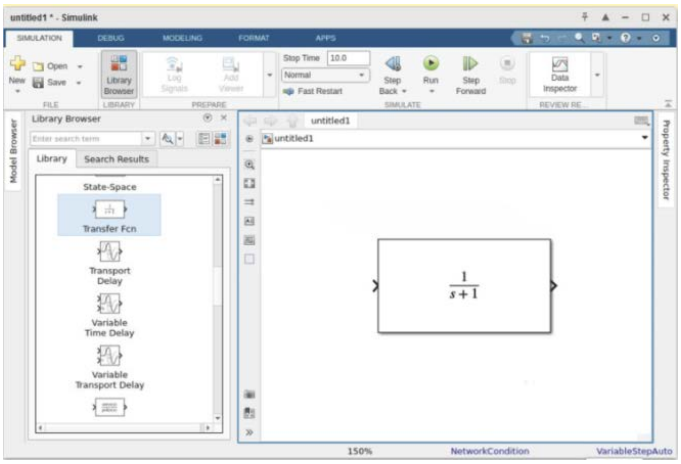


Figure 21. From Simulink's Library Browser drag the transfer function block and place it on the plot.

IV.3.4) Add other building blocks

The library browser allows you to place my different blocks in your simulation. We will start with the three most common ones:

The Step input signal:
This is only one of the possible input blocks, but it is one of the most common ones. Think of it as a switch. It starts as no signal and then as some point (which we can set) it jumps to fully open and gives the maximum amount of input. Graphically speaking it is represented by a step function. It can be found in the "Source" section of the library:

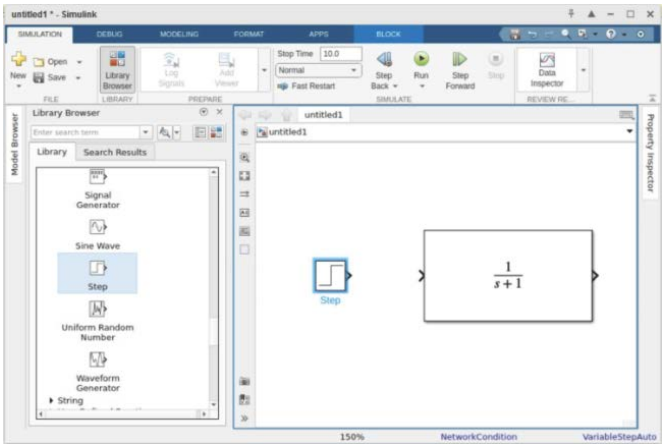


Figure 22. From Simulink's Library Browser drag the step block and place it on the plot.

Mathematical operators:

If you need to sum or subtract two signals, you can use the relevant operator blocks. For example, if we implement a feedback loop (closed loop control system), we need to find the error (difference) between the reference (input) signal and the actual (output) signal of the system. That can be achieved with a subtraction block in the following way:

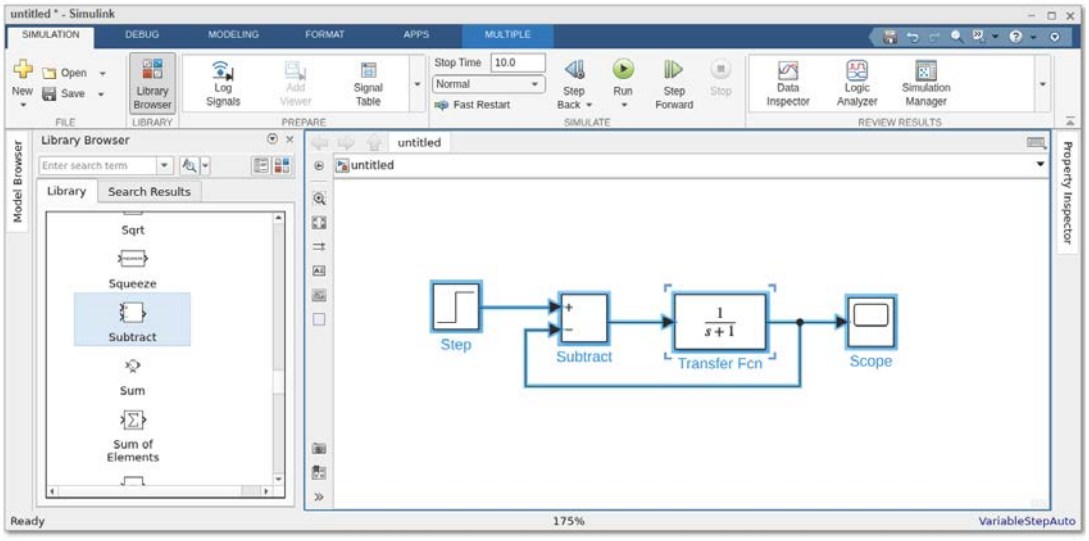


Figure 23. An example of a simple closed loop control system (with a feedback loop).

Scope:

The scope block can be found under the "Sinks" section of the library. It is essentially a block that the signal pours into so it can be measured (over time). The block gives graphical representation of the recorded data. You can see the scope connected to the output of the system in Figure 23

IV.3.5) Step 5: Set block parameters

All the blocks we will be using for our models allow for various settings to be configured (their parameters). Those are usually opened by double-clicking on the block icon.

The step block parameters are:

- **Step time**: the time (in seconds) when the step (the sudden increase) occurs. By default the value is 1 sec;
- **Initial value**: the value before the step. If we want to use this as a switch, then the initial value should be 0, but of course you can use it in different ways;
- **Final value**: the value after the step (the increased value we get to);
- **Sample time**: this is the sampling rate of the scope. think of it as the frequency with which the signal is measured during the simulation run.

For the transfer function we need to provide the coefficients for the polynomial in the denominator. For example, let's imagine we had to use a linear relationship transfer function. As we know linear equations are first order polynomials. If we write the line equation in its general form, it would be:

$$f(x) = ax + b$$

where a and b are the constants in front of the various powers of the variable. If we write is so that the powers of the variable are visible it would have looked like this:

$$f(x) = ax^1 + bx^0$$

When defining the transfer function block, MatLab expects us to supply a vector containing these constants. So, for the line equation:

$$f(x) = x + 1$$

we will have to enter the vector $[1 \quad 1]$, same as what is shown in Figure 24.

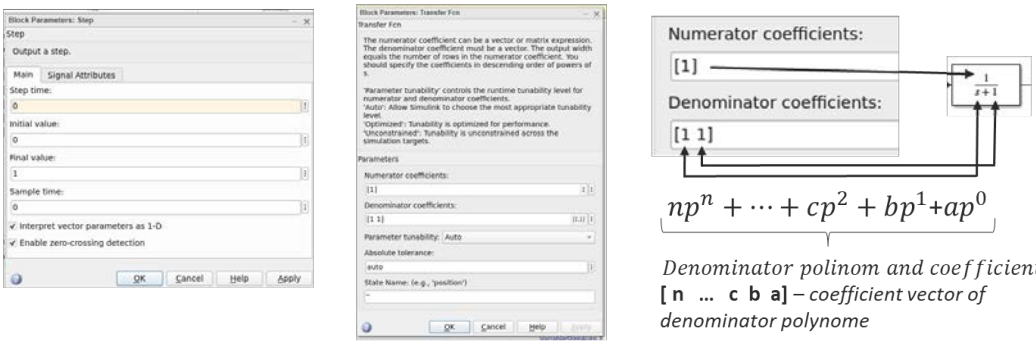


Figure 24. Block parameter windows for the step input block and the transfer function block.

IV.3.6) Step 6: Run simulation and visualise the results

Once you have set all the necessary parameters of the blocks that comprise your system, you can run the simulation by clicking on the "Run" (play) button in the MatLab taskbar, under the Simulation tab. If you want to see the results in the scope you have to highlight it before starting the simulation run.

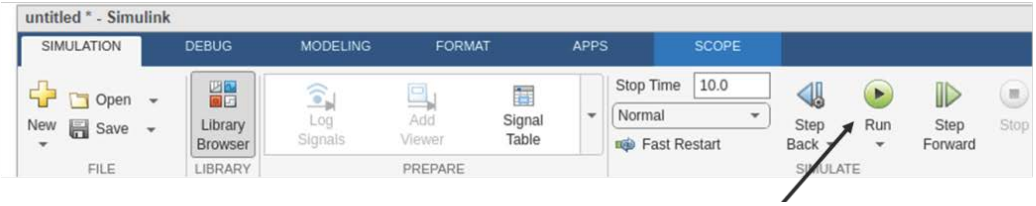


Figure 25. Click the "Run" button to launch the simulation.

You can view the graphical representation of the results by double clicking the scope block, once the simulation is over, as shown in Figure 26.

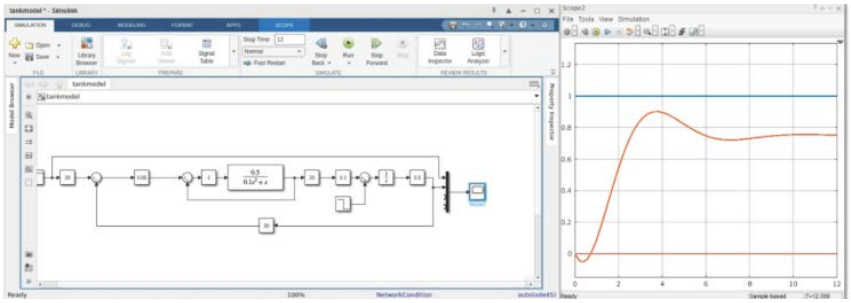


Figure 26. Once the simulation is done, double click the scope block to view the results.

Now that we know how to access the SimuLink environment and to place building blocks for our block diagram models, let's build some examples and simulate them.

IV.4. Home heating system modelling

Exercise IV.4-1: Create a simulated home heating system. Implement an open loop and a closed loop control and discuss the differences.

Our task will be to examine the concept of a home heating system and find a suitable way to represent it in a MatLab simulated environment. Then, we will analyse the difference in controlling the heating system using both an open loop and a closed loop control solution.

Our first step will be to determine the parameters of the system and assign them to some variables Figure 27:

- **Disturbance**:
  This will be the outdoor temperature, which is affecting our home, and over which we have no control. Let's assign it to the variable Theta_external: θ_{ext};
- **Home temperature**:
  This is our output variable, the indoor temperature in our home, which we can control. Let's assign it to the variable Theta_internal: θ_{int};
- Reference temperature:
  This is our input variable (also known as reference or the set point). It is the temperature we would like to have. Let's assign it to the variable Theta_c: θ_c;
- **Controlled variable**:
  Although we have a reference temperature, we do not directly input temperature in the system. What we actually control is some heating equipment (let's say a burner). What

really drives the temperature up or down is how much power we give to that burner. So, let's assign that to a variable Power_burner: P_b;

- **Regulated variable**:
  initial flow temperature θ_d, that value depend on the external temperature, considered also as disturbance for the open loop system $\theta_d = \theta_d = f(\theta_{ext})$,
- **Boiler**:
  Let's denote it with C,
- **Regulator**:
  This is our controller, which we will use to set the temperature. Let's assign it to be R;
- **External temperature sensor**:
  At some point we would like to measure the outside disturbance on our system. In our case that is the outside temperature. When we need to introduce it in the model, we can denote it with S.

For this heating open loop system, the process control includes all the material and technical devices implemented to maintain a physical quantity (temperature) to be regulated, equal to a desired value, called setpoint. For this example, we consider only the outdoor temperature for the decision-making process and in this way to increase or decrease the power of the boiler.

When disturbances (open window, open door, losses), or changes of setpoint occur (changing of the outdoor temperature), the regulation causes a corrective action (increasing or decreasing the power of the boiler) on a physical variable of the process, called controlled variable (or control).

For that open loop heating system (Figure 27), we do not consider the indoor temperature and the comparison of that measurement with the outdoor temperature.
The main inconvenient of the open loop systems is that we don't know a priori at what value the set value will stabilize and in how long time. Moreover, the set value will vary according to the disturbances.

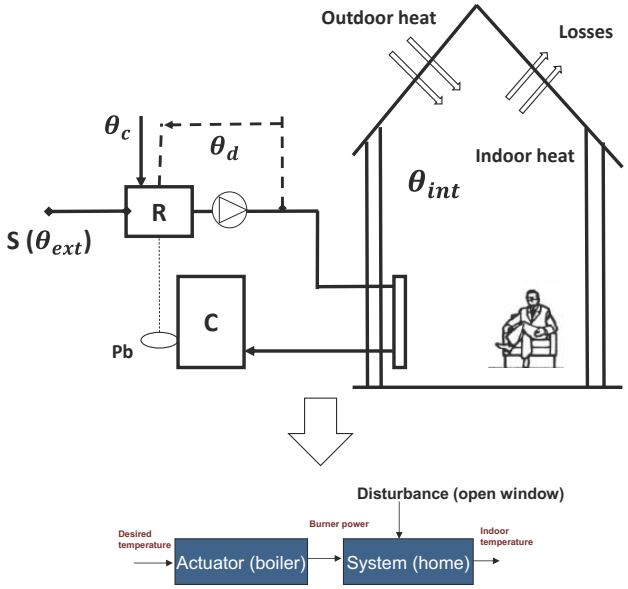


Figure 27. Open loop control system: heating plant (home heating system).

For the numerical study case we have the statement of the problem:

- Cold house ($T_{ext} = 0$°C);

- At t=0, we turn on the heating system;
- The system: air + total heat capacity walls C;
- We assume that the temperature is uniform: T(t);
- The heating power is constant: Φ_s;
- Outgoing exchanges (losses): Φ_p;
- What is the evolution of T = f(t)?

Fort his heating system and base on the first law of thermodynamics we have:
PRODUCTION = STORAGE (in the air and the walls) + OUTGOING EXCHANGES (heat losses) as shown in the system of equations bellow (IV.4-1):

$$\begin{cases} \Phi_{source} = \frac{dU}{dt} + \Phi_p \\ \Phi_{source} = C\frac{dT(t)}{dt} + \Phi_{pertes} \\ \Phi_{pertes} = \frac{1}{R_{th}}(T(t) - T_{ext}) \end{cases} \quad \text{(IV.4-1)}$$

We can conclude that for this system we have an approximation with an electrical system as shown on Figure 28

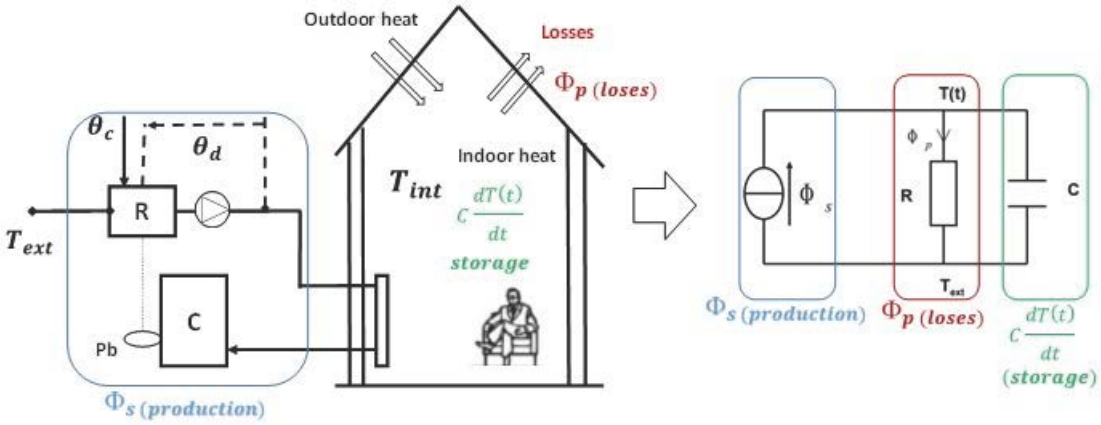


Figure 28. Heating system (left) and approximation with electrical model (right).

As final relation described the house system, we have the system of equations (IV.4-2):

$$\begin{cases} \Phi_{source} = C\frac{dT(t)}{dt} + \frac{1}{R_{th}}(T(t) - T_{ext}) \\ \frac{dT(t)}{dt} = \frac{1}{C}\left[\Phi_{source} - \frac{1}{R_{th}}(T(t) - T_{ext})\right] \end{cases} \quad \text{(IV.4-2)}$$

For this study in steady state, we have the following technical parameters and initial conditions:

- $T_{ext} = 0°C$;
- $R_{th} = 2{,}85.10^{-3} KW^{-1}$;
- $C = 6{,}27.10^{-6} JW^{-1}$;
- $T(t=0) = 0°C$;
- $T_{\infty} = 20°C$;
- $\Phi_{source} = 7kW$.

For the installation and the open loop system, based on an equation X and the created Matlab/Simulink model (Figure 29 left). As setpoint we have the step of 7 kW of energy

source. As result, we have as output of steady state (step response), the indoor variation of the temperature as presented on the scope in Figure 29 right:

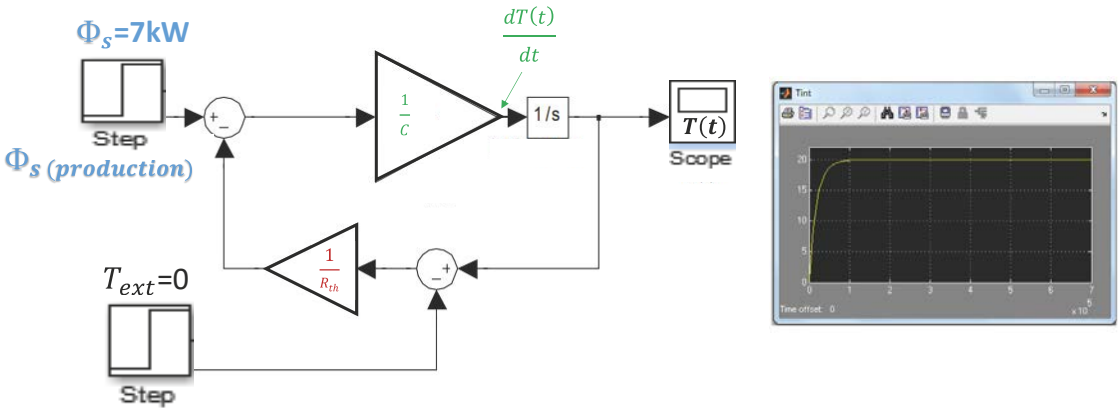


Figure 29. Heating system: steady state Matlab/Simulink model (left) and indoor temperature variation (right).

As result, we have the obtained 20 degrees in the house. This open loop study of the system does not consider the disturbances (variation of external temperature). For the simulation, we will consider that we have a disturbance (a sudden drop in the outdoor temperature) with a value of -5°C, after 3.5 seconds from the start of the simulation. That leads to the results presented on Figure 30, where we distinguish the influence of the drop of the outdoor temperature on the indoor temperature.

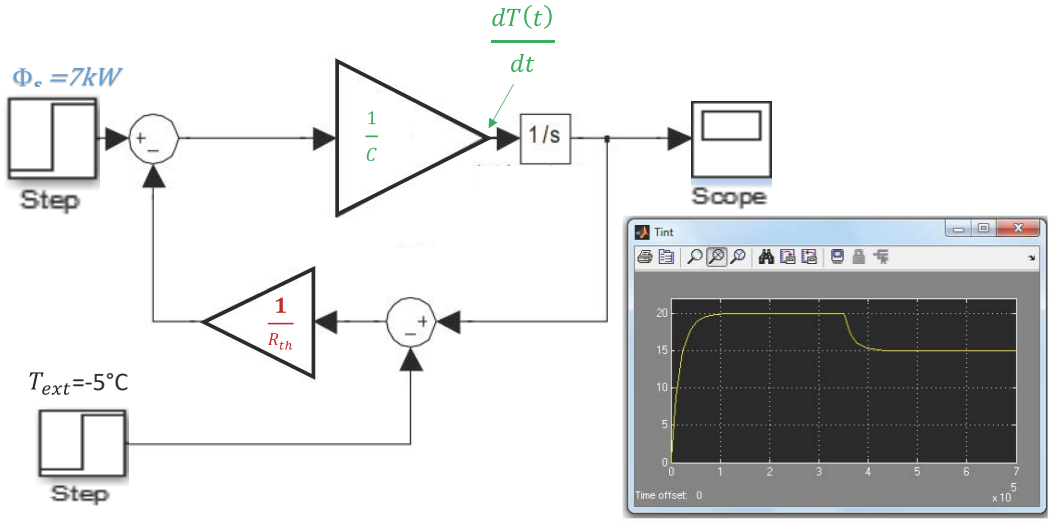


Figure 30. Heating system: steady state Matlab/Simulink model (left) and indoor temperature variation (right) under disturbances

In order to avoid and to compensate the disturbances which influence the system, and to control the indoor temperature, we will now look into the closed loop version of what we did so far.

In order to avoid all disturbances, we need to apply control to the system. The correction of the inconvenient overview of the open loop systems, are presented when we close with feedback the system and we impose some reference as input of the system. The idea is to maintain this level of the controlled output (indoor temperature) as the reference temperature and allow for zero error (Figure 31). If the error is not equal to zero, a correction action will be applied on the boiler element.

For the closed loop control system, we will introduce the following:

u_t, which is the control signal applied to the system,
$\varepsilon(t) = \theta_c - \theta_{int}$ (in Figure 32 it is represented as $\varepsilon(t) = \theta_c - \theta_{int}$, which is the error, the gap of temperature between the setpoint Φ_{source} (setpoint of desired temperature, or the input) and the indoor measured temperature T(t) (which is our output).

This measurement is compared with the setpoint to generate a deviation (ε) signal. Depending on the sign and amplitude of the error, the control law will dose the supply (u) of the resistor so that the temperature in the boiler (radiator) remains as close as possible to the setpoint.

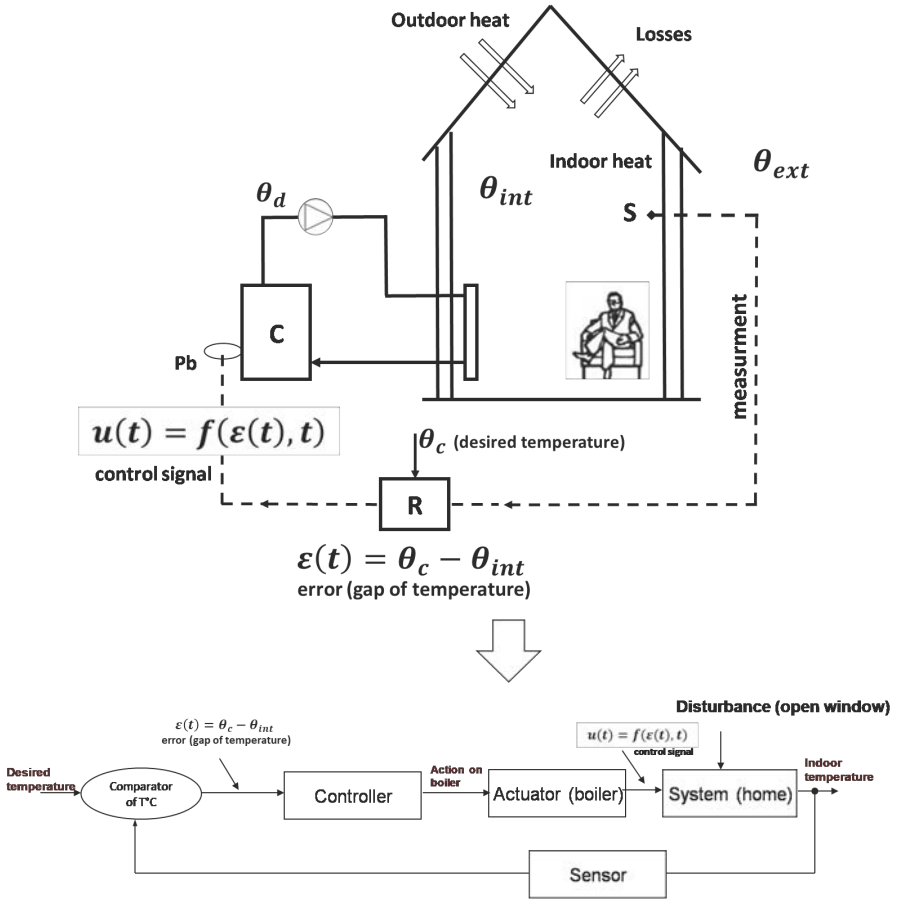


Figure 31. Close loop control system: heating plant (home heating system).

For this close loop system and based on the systems of equations in (IV.4-1) and (IV.4-2), we can apply the methodology in mapping process and to obtain the transfer function described with the system (IV.4-3) in order to control the system under disturbances, parametrical variation and to optimize it. For the control system the input is presented by the value Φ_{source} (setpoint) and the output for the system is the indoor temperature T(t).

$$\begin{cases} \Phi_{source}(t) = C\frac{dT(t)}{dt} + \frac{1}{R_{th}}(T(t) - T_{ext}) \\ C\frac{dT(t)}{dt} + \frac{1}{R_{th}}(T(t) - T_{ext}) = \Phi_{source}(t) \\ LT \Rightarrow CpT(p) + \frac{1}{R_{th}}T(p) - \frac{1}{R_{th}}T_{ext} = \Phi_{source}(p) \\ T(p)\left[Cp + \frac{1}{R_{th}}\right] - \frac{1}{R_{th}}T_{ext} = \Phi_{source}(p) \\ TF(p) = \frac{T(p)}{\Phi_{source}(p)} = \frac{1}{\left[Cp + \frac{1}{R_{th}}\right]} + \frac{1}{R_{th}}T_{ext} = \frac{R_{th}}{[R_{th}Cp+1]} + \frac{1}{R_{th}}T_{ext}, \\ (initial\ conditions\ \frac{1}{R_{th}}T_{ext} = C_{const}) \\ TF(p) = \frac{2{,}85.10^{-3}}{[2{,}85.10^{-3}.6{,}27.10^{-6}p+1]} + C_{const} = \frac{2{,}85.10^{-3}}{[17865p+1]} + C_{const} \end{cases} \quad \text{(IV.4-3)}$$

Depending on the regulation element, one option is the “on-off temperature control”. For the simulation and study of that system in soft environment, we use as a control element a hysteresis element with upper and lower limits. Therefore, in that way if the upper margin value of the regulator is exceeded, the system stops its operation, until the sensor detects the drop below the value of the lower limit defined in the regulator. Once this lower limit has been exceeded, the boiler is put back into operation to maintain the value of the temperature in the habitat around the set value. To create the Simulink model, we use the transfer function derived in (IV.4-3). For the simulation study, shown in Figure 32, we use the following elements for the close loop system:

- Step source block (setpoint) – desired indoor temperature 20°C;
- Transfer function block – the system and actuator model;
- Controller block – presented the on-off system controller;
- Output scope block – visualization of the output temperature.

The visualization of the controlled temperature is presented on Figure 32. We can observe that when we have a disturbance (open window) the controller switches on/off the boiler in order to maintain the average desired value between upper and lower limits (19 and 21°C). The controller eliminates the disturbance influence.

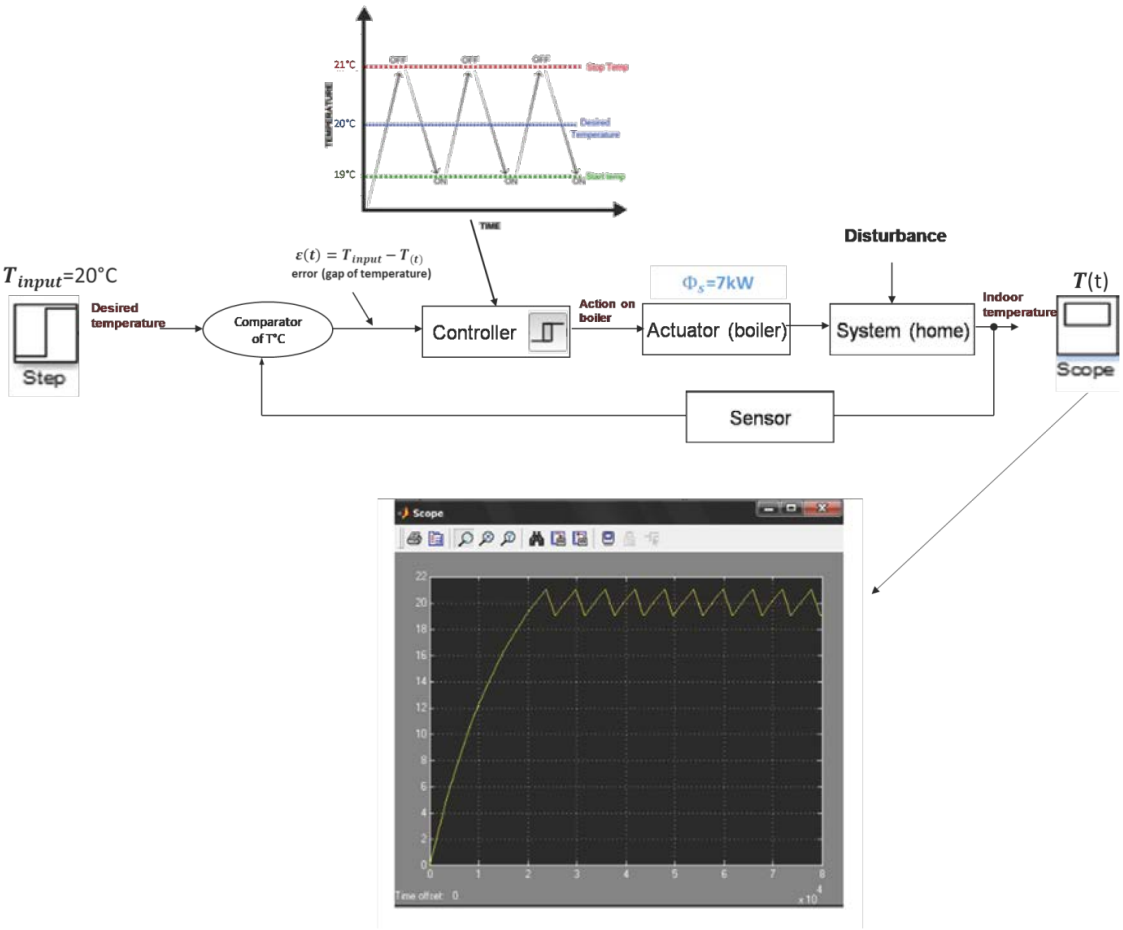


Figure 32. On-off temperature control of the house

IV.5. DC Motor modelling

In our next example we will see how we can simulate a DC motor and create a speed control system for it. Our purpose will be to examine the differences between the open loop and closed loop control solutions – how the two affect the functionality, the measurement and the setpoints.

Exercise IV.5-1: Create a simulated heating plant using a DC motor. Implement an open and closed loop control system for the motor speed.

Let's start with the open loop. For a DC motor system, we have the following technical devices and elements (comprising the electrical and mechanical part of the system):

- battery or power supply;
- resistor and inductor;
- DC motor rotor;
- the output (rotational speed, for the respective mass, as a function of time) $\omega_m(t)$

From an electrical point of view, the DC motor can be modelled as a system, whose input is the control voltage $U_m(t)$ of the rotor and the output is the rotational speed $\omega_m(t)$ of the motor shaft as shown in Figure 33. The rotor is modelled by a resistance in series with an inductance and a back EMF (a counter-electromotive force).

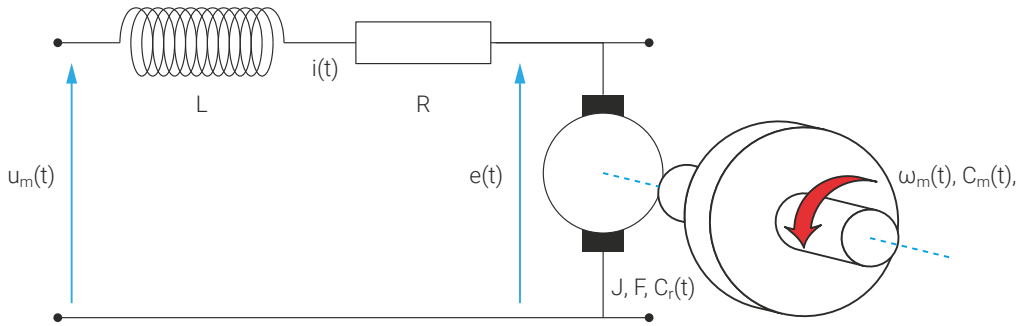


Figure 33. Electro-mechanical drawing of a DC motor system

The DC motor knowledge model is given in (IV.5-1):

$$\begin{cases} u_m(t) = e(t) + R.i(t) + L.\frac{di(t)}{dt} \\ e(t) = K_e.\omega(t);\ c_m(t) = K_c.i(t) \\ c_m(t) - c_r(t) - f.\omega(pt) = J.\frac{d\omega(t)}{dt} \end{cases} \quad \text{(IV.5-1)}$$

For this exercise we will have the following technical parameters and initial conditions:

- $u_m(t)\ [V]$ - power supply (battery);
- $i(t)$ [A] – consumed current;
- $e(t)[V]$ – back-electromotive voltage;
- R– Resistance;
- L - Inductance;
- $Ke[V/\frac{rad}{s}]$ – the coefficient of EMF (electromotive force);

- $\omega(t)[\frac{rad}{s}]$ – the rotational speed of the motor shaft;
- $f\ \left[N.\frac{m}{\frac{rad}{s}}\right]$ – total friction parameter;
- $J\ [kg.m^2\ ]$ – total inertia reduced to the motor axis;
- $Kc\ [N.m/A]$ – coupling constant;
- $C_m(t)[N.m]$ – the torque of the motor;
- $C_r(t)[N.m]$ – the resistant torque on the motor shaft.

In the case where all the initial conditions are zero, the Laplace transforms of these equations are:

- ohm's law in the armature circuit (IV.5-2);
- the equation of electromagnetism in the motor (IV.5-3);
- the equation of motor shaft dynamics (IV.5-4).

$$\{U_m(p) = E(p) + R.I(p) + L.p.I(p)\ \Rightarrow\ \left\{\frac{I(p)}{U_m(p)-E(p)} = \frac{1}{R+L.p}\right. \quad \text{(IV.5-2)}$$

$$\begin{cases} E(p) = K_e.\Omega(p) \\ C_m(p) = K_c.I(p) \end{cases} \Rightarrow \left\{\frac{E(p)}{\Omega(p)} = K_e;\ \frac{C_m(p)}{I(p)} = K_c\right. \quad \text{(IV.5-3)}$$

$$\left\{C_m(p) - C_r(p) - f.\Omega(p) = J.p.\Omega(p)\ \Rightarrow\ \left\{\frac{\Omega(p)}{C_m(p)-C_r(p)} = \frac{1}{f+J.p}\right.\right. \quad \text{(IV.5-4)}$$

Thereby obtained total transfer function and open loop system (Figure 33) for a DC motor is presented on the block diagram below (Figure 34).

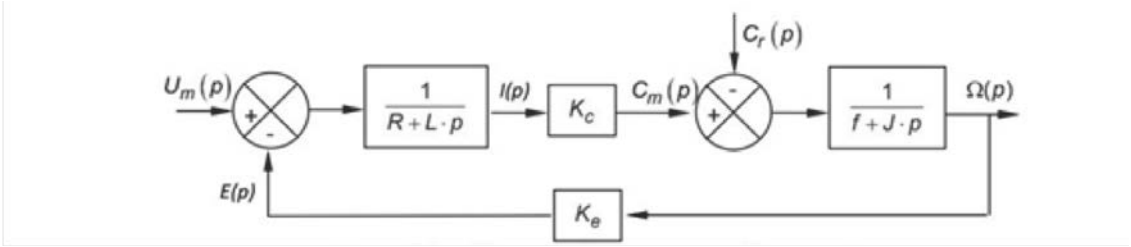


Figure 34. Open-loop model of a DC motor in Simulink (MatLab).

$C_r(t)$ is the force which opposes the rotational movement of the motor, and which tends to slow it down (disturbance). If we assume that this force is equal to zero, we obtain a second order transfer function model for this DC motor.

For the steady state system simulation, equivalent block diagram for the open loop system and builder Simulink model are presented on (Figure 36 left). As we know the speed of rotation is proportional to the voltage applied to the motor and the results in the software environment for that DC motor is presented on (Figure 36 right). The result we obtain is a linear relationship between the output rotation speed and the different input supply voltages.

For the simulation model in SimuLink (Figure 35) the used elements are the following:

- **input**: the setpoint of voltage;
- **constant (gain) block**: for the resistance and inductance elements, the K_c, K_e and the inertia element J;
- **add:** – addition (summation) block: for the feedback and the disturbance signal;
- **integral block**: in order to introduce the derivation of current (electrical part) and rotational speed of the DC motor (mechanical part).

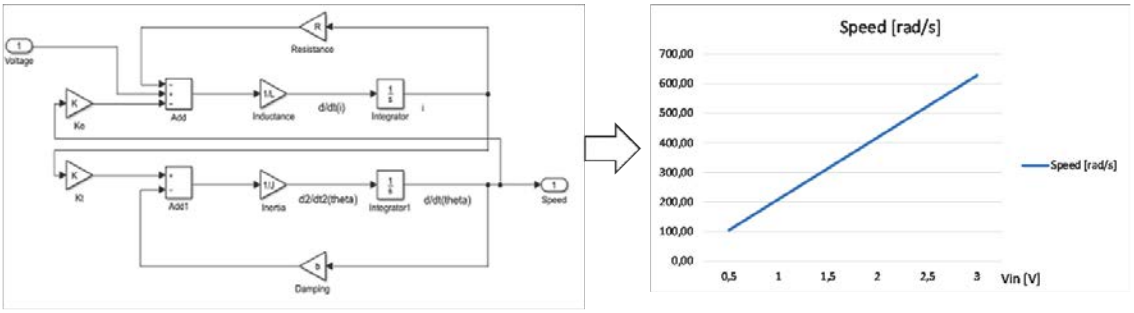


Figure 35. Open loop control system of a DC motor: MatLab/Simulink model (left) and resulting relationship between the various input supply voltages and the output rotational speed (right).

For the simulation study in the Simulink environment, we will use the following values for the different parameters:

R = 2.0;
L = 0.5;
K_c = 0.1;
K_e = 0.1;
J = 0.02;
f = 0.2.

In the following we will use the obtained TF model in order to make a simulation analysis of the study state and the controlled system. The corresponding transfer function of that open loop system (Figure 34) is as presented in equation (IV.5-5).

$$\begin{cases} \dfrac{\Omega(p)}{U_m(p)} = \dfrac{\frac{k_c}{(R+L.p)(f+J.p)}}{1+\frac{k_c}{(R+L.p)(f+J.p)}k_e} = \dfrac{k_c}{(R+L.p)(f+J.p)+k_c.k_e} \\ \dfrac{\Omega(p)}{U_m(p)} = \dfrac{0{,}1}{(2+0{,}5.p)(0{,}2+0{,}02.p)+0{,}01} = \dfrac{0{,}1}{0{,}01p^2+0{,}14p+(0{,}4+0{,}01)} = \dfrac{0{,}244}{0{,}024p^2+0{,}34p+1} \\ TF(p) = \dfrac{k}{\frac{1}{\omega_n^2}p^2+\frac{2\xi}{\omega_n}p+1} \begin{cases} k - \ static\ gain\ of\ the\ system = 0.244 \\ \xi - \ damping\ coeficien\ = 1.097 \\ \omega n - \ natural\ frequency = 6.45 \end{cases} \end{cases} \quad \text{(IV.5-5)}$$

For that system we can identify the system parameters and to make the step analysis as shown in Figure 36.For this study we consider that we have not disturbances $C_r = 0$ (load disturbance).

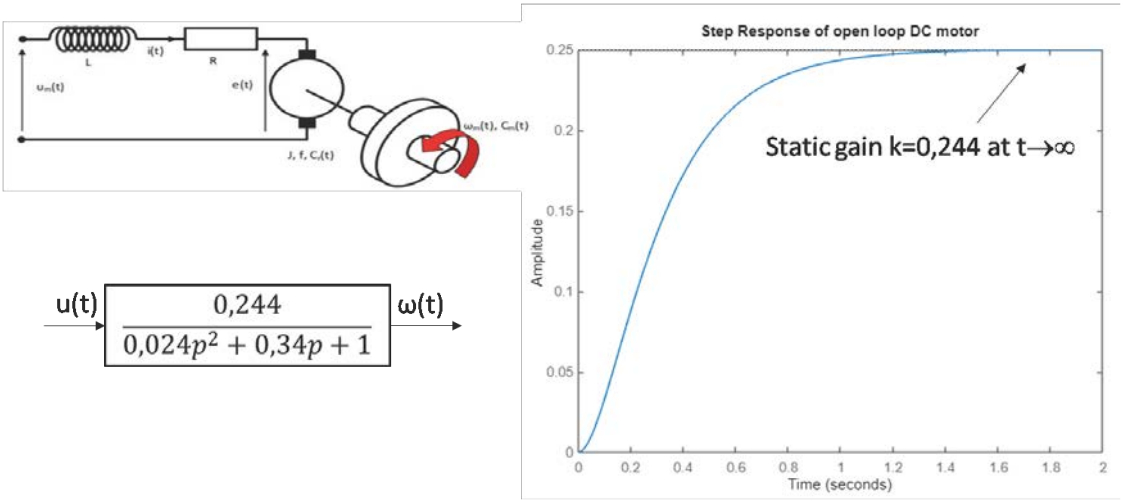


Figure 36. Simulation results with an open loop control system of a DC Motor.

Our next step would be to take the same system, but instead of open-loop control try to implement a closed-loop control system. For that aim we can use a simple structure to command the angular velocity of the DC motor. But we have to note that this proportional corrector (constant) will not perform correctly when load disturbances appear.

When we apply a load disturbance (C_r) for the case of non-controlled system at time *t = 4s* to t=8s (negative signal applied on system) as presented in Figure 37, we can see that the disturbance influences the output of the system.

For the simulation model in Simulink (Figure 36) the used elements are the following:

- **step input:** the setpoint of voltage;
- **transfer function:** the obtained second order TF of the DC motor (IV.5-5) – the ratio between the output (angular velocity) and the input (feed voltage);
- **signal builder:** the signal builder helps us to build the load disturbance signal variation;
- **sum:** sum block to add the input and the disturbance;
- **scope:** allows us to record the output of the system in a graphical way. A step response of the open loop DC motor under the given disturbances.

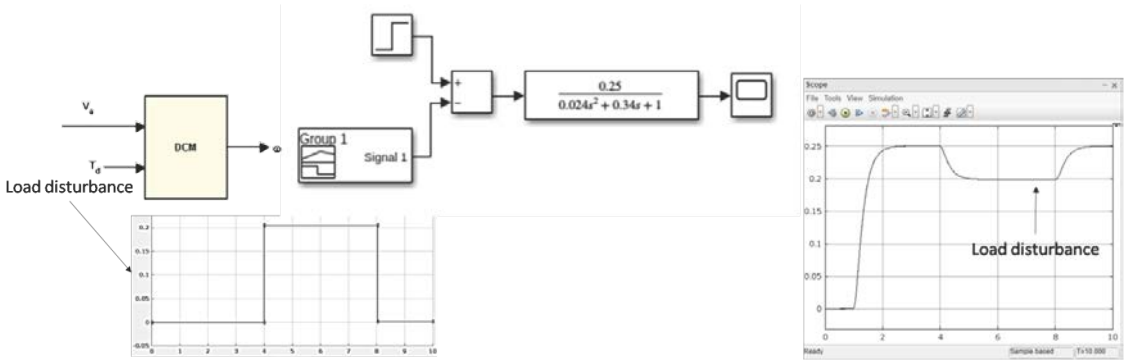


Figure 37. Load disturbance influence on a DC motor

For the same system of DC motor, we apply an open loop control with proportional regulator in order to study the influence of disturbances (Figure 39).

When we add control element in the open system, in order to avoid the influence of disturbances on the load, we introduce in the Simulink model the PID controller block – using only the proportional part P(s) (integral and differential part equal to zero). We observe that under control the system fixe the positional error (final value position) is reduced considerably, but the disturbance influence is still presented on the Figure 38.

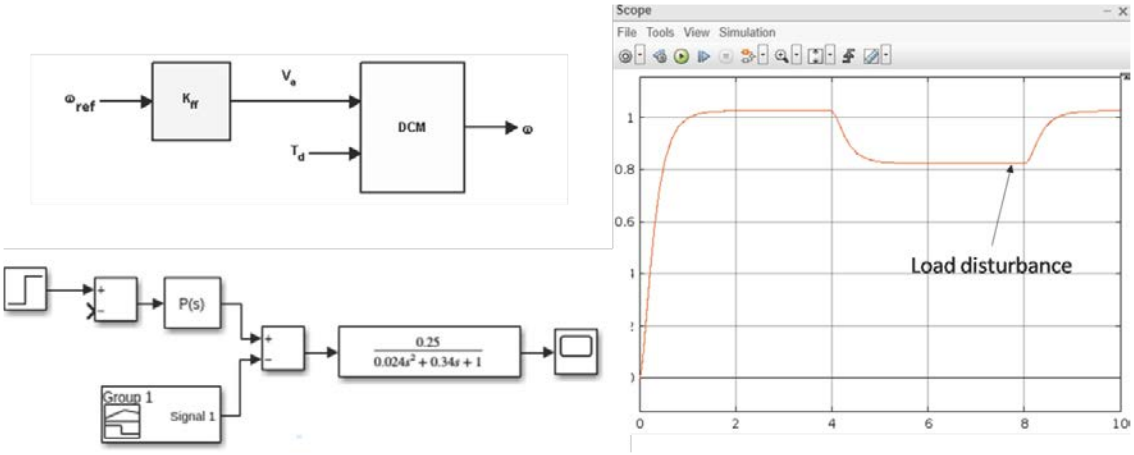


Figure 38. Open loop proportional control system under load disturbances

Clearly, feed forward control handles load disturbances poorly. In the following, we try the integral feedback control design (PI controller). This controller has a compensation influence when we have load disturbances on DC motor system.

For that close loop system, we obtain as results (Figure 39). A graphical comparison analysis is made (Figure 39 right) for the open control system (P regulator under disturbances on load) and close loop control system (PI controller under disturbances on load).

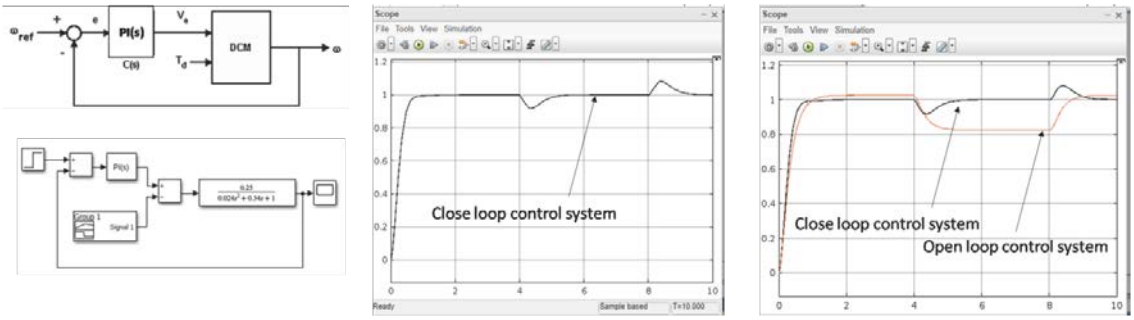


Figure 39. Close loop control system of DC motor under load disturbances

From Figure 39 it becomes apparent that the closed loop control takes into account the disturbances and considerably reduces their influences.

IV.6. Liquid level control system modelling

In our next example we will simulate a liquid level control system by creating a SimuLink model for it. Our purpose will be to examine the differences between the open loop and closed loop control solutions and how the two affect the functionality, the measurement and the setpoints. Our main goal will be to apply a control strategy in order to maintain a constant level in the tank system, taking into account the disturbances in the system.

Exercise IV.6-1: Create a simulated liquid level control system. Implement an open and closed loop control for the tank's intake valve.

The example of tank system, with liquid level control we have the following technical device and the corresponding block diagram of the system (Figure 40 and Figure 41). The goal is that the desired level n should follow a setpoint value n_c, displayed by a potentiometer. And this should be true even if the leakage rate q_f varies.

To achieve this, an error voltage is applied: $\varepsilon(t)= v_c(t)-v_n(t)$, amplified by an amplifier (with gain A_1) to valve position feedback. This servo includes a second amplifier (with gain A_2) which supplies the armature of a DC motor. The motor shaft drives a reducer whose output defines the position of the valve and therefore modifies the inlet flow q_e. The valve position is measured by a sensor mounted on the motor shaft. The different components have the following characteristics:

- **Tank** : maximum level n_{max} = 0.5 m, section S = 0.5 [m2];
- **Level sensor**: output voltage $v_n = \lambda . n$ with λ = 20 [V/m];
- **Setpoint potentiometer**: graduated from 0 to n_{max} delivering $v_c = \lambda . n_c$ with λ = 20 [V/m];
- **Amplifiers 1 and 2**: gains A_1 and A_2;
- **Motor M**: DC motor supplied by the armature, with transfer function (IV.6-1) with K_m=0.5 [rad/s.V] et τ_m=0.1 [s]

$$\frac{\theta_m(p)}{E(p)} = \frac{K_m}{p(1+\tau_m p)} \quad \text{(IV.6-1)}$$

- **Motor position sensor**:

$$v_m = k_c \theta_m, \text{ where } k_c = 1\ [V/rad] \quad \text{(IV.6-2)}$$

- **Reducer reduction ratio:**

$$\frac{1}{v} = \frac{\theta_m}{\theta_v} = 20 \quad \text{(IV.6-3)}$$

- **Valve flow:**

$$q_e(t) - q_f(t) = s\frac{dn(t)}{dt}\ [m^3/s.rad] \quad \text{(IV.6-4)}$$

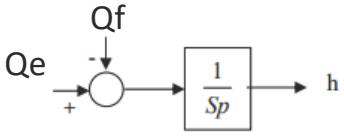


Figure 40. Open loop model of a liquid tank in SimuLink

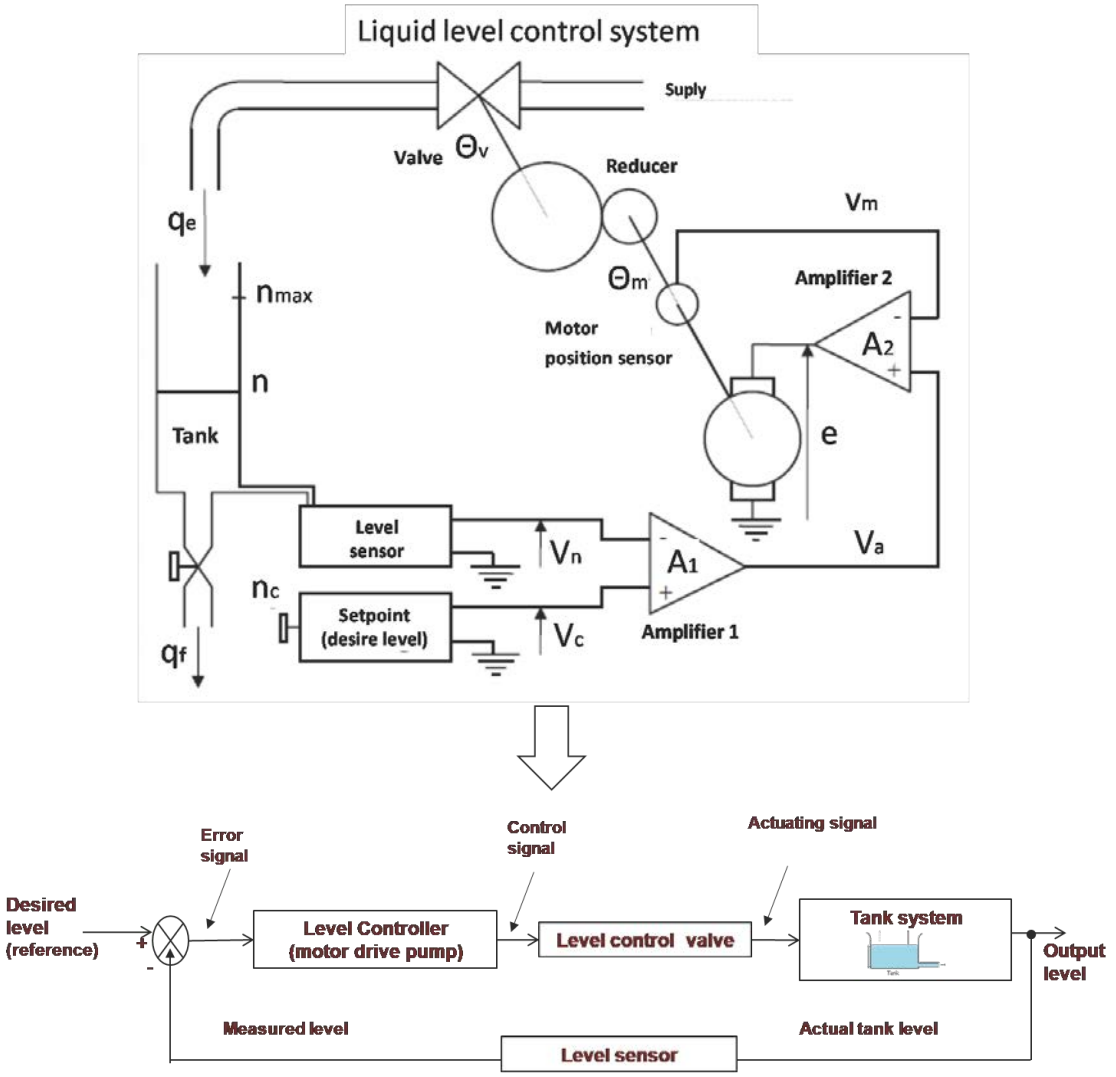


Figure 41. Controlled tank system: liquid level control (closed loop system).

Based on the technical scheme and all characteristics mentioned before we obtain the fowling block diagram (Figure 42). We can identify the technical devices and the internal input/output for the system, gathered in 4 sets: tank system, valve, motor with amplifier A_2, measurement (setpoint and output tank level measurement) and error with amplifier A_1.

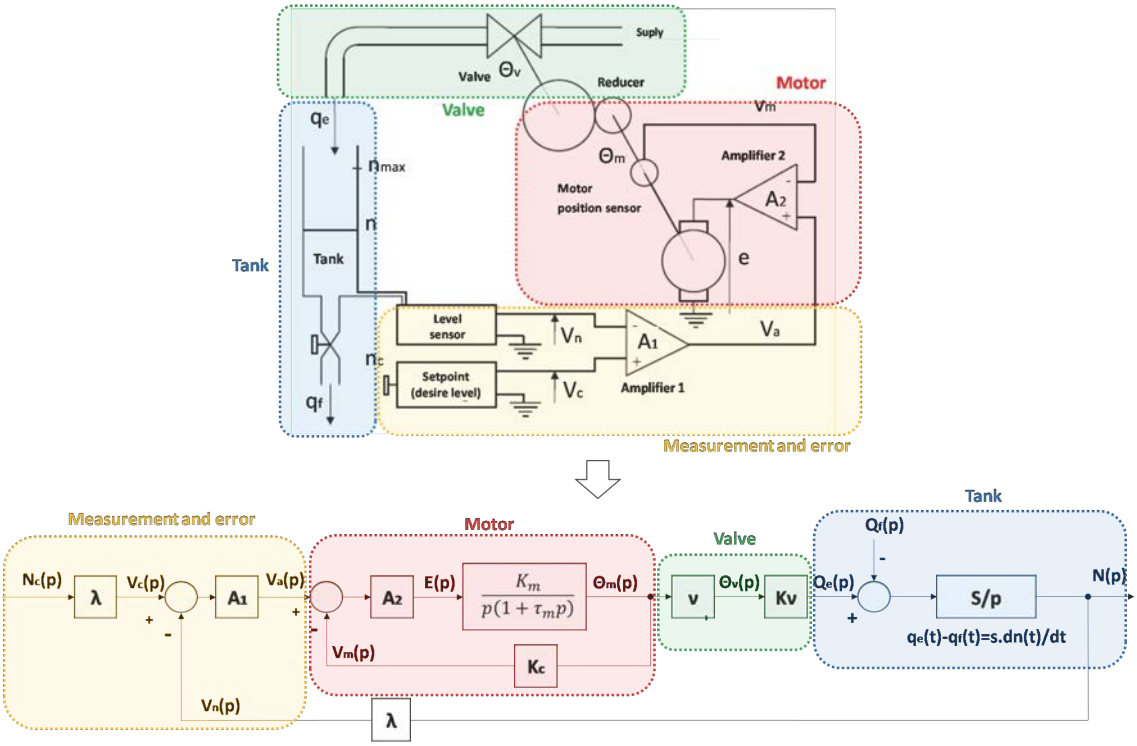


Figure 42. Controlled tank system identification sets.

We build the block diagram using the numerical values and the Simulink elements as presented on Figure 43, Figure 44 and Figure 45.

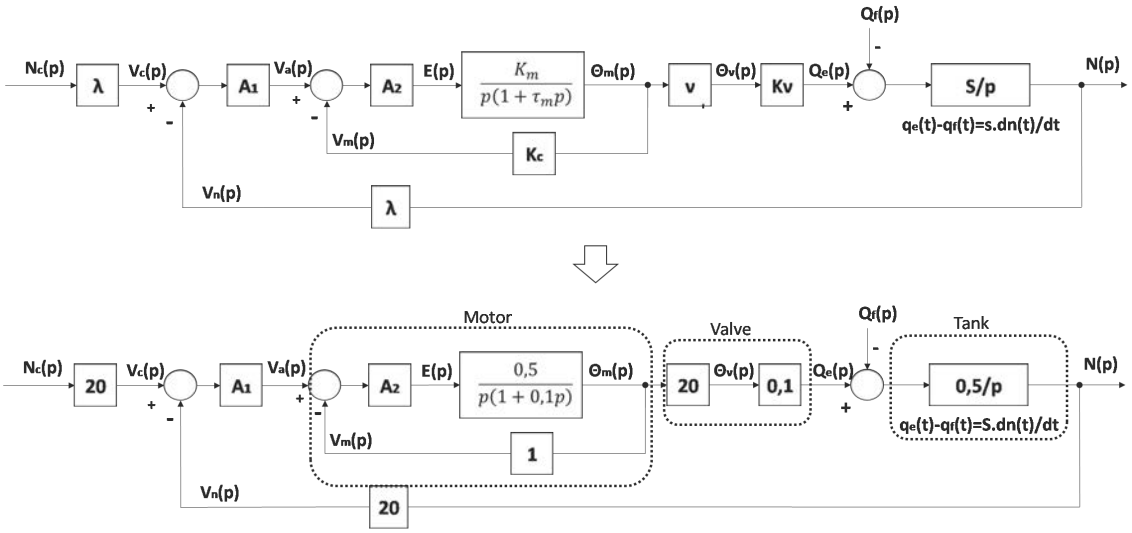


Figure 43. Tank liquid level control system: real model block diagram.

The close loop study shows that without disturbance (Q_f =0), the level reaches without error the assumed constant setpoint level Nc (setpoint level). When we adjust the amplifier constant A_1 we influence the position of the control valve, namely the opening position of the controlled valve and the flow rate entering the tank (Figure 44).

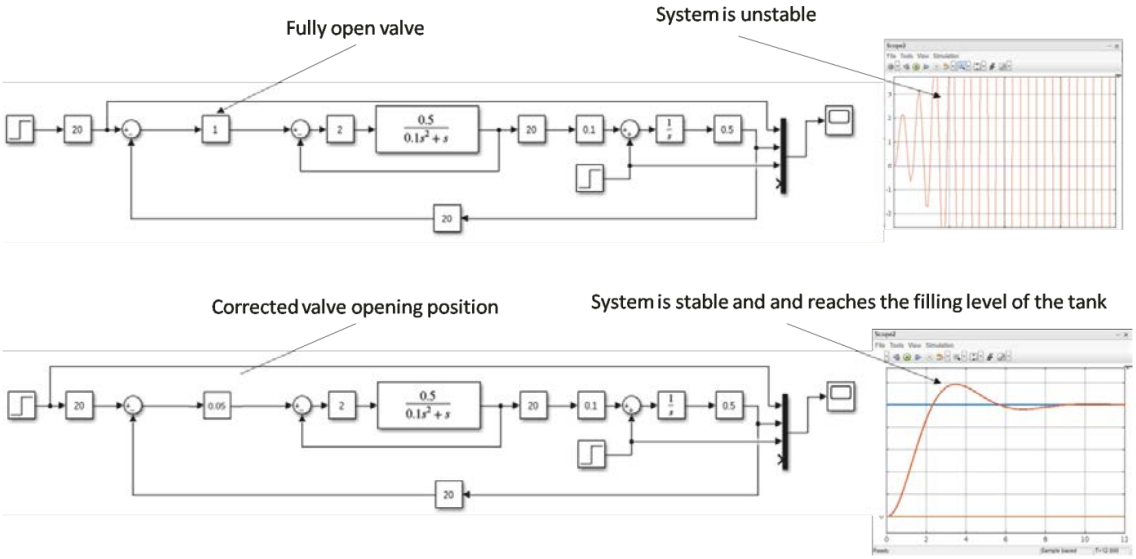


Figure 44. Close loop liquid level study: without control on the valve (up) and with control on the opening position on the valve (down).

After applying disturbance flow Q_f on the output of the tank (Q_f = 50% Q_e), we obtain the following results (Figure 45).

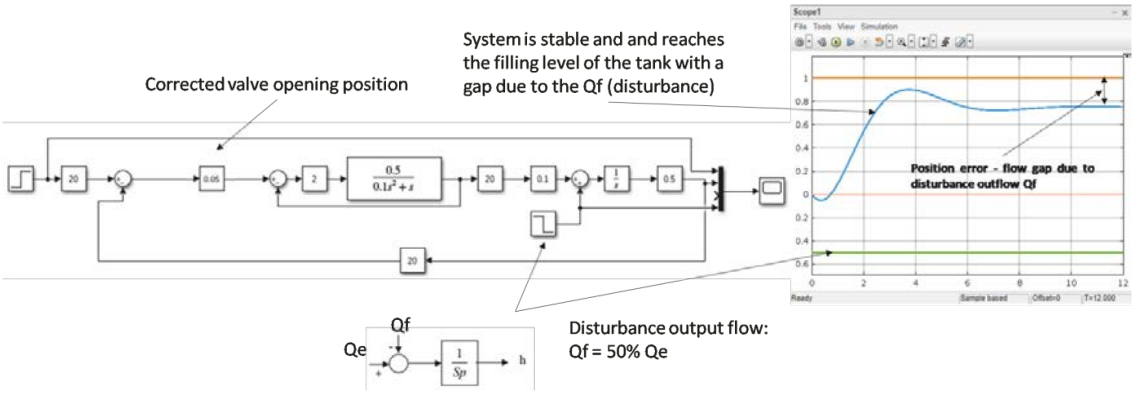


Figure 45. Close loop liquid level study: with control on the opening position on the valve and disturbance flow Qf.

This study shows that the system is stable and reaches the filling level of the tank with a gap due to the Q_f (disturbance), but without exceeding the maximum admitted level in the tank.

IV.7. Wind energy production system modelling

For our next example we will simulate a wind energy production system. We need to create a speed control system for it. Again, our purpose will be to examine the differences between the open loop and closed loop control solutions and determine how the two affect the functionality, the measurement and the setpoints. For this example, we will assume that the energy production system is connected to a battery in the output. The limit of the battery capacity is one of the reasons why we need to apply control (in order to avoid an overload).

Exercise IV.7-1: Create a simulated wind energy production system. Implement an open and closed loop control for the systems battery charging control.

With advancement of digitalization and the available computational power, companies are more and more often looking to build capabilities for simulating entire processes in real-time. Digital twins were initially conceived as a way to simulate the operation of a specific machine, in order to try out different settings scenarios, since it is much cheaper to test variations in a simulation, rather than spent significant amount of money to reconfigure a machine (while also interrupting the production process it is part of) in order to check if you will get better results.

The constructive elements (functional units) of a closed loop control of a small wind energy production system (wind turbine) would be (as shown on Figure 46):

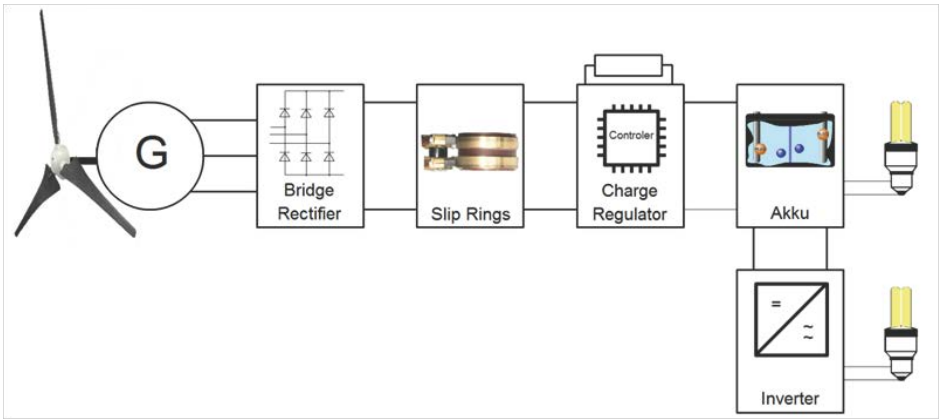


Figure 46. Closed loop control system of small wind energy production system.

- rotor with blades (spinning due to wind),
- weathercock (to self-position the blades in respect to the wind),
- alternator (synchronous generator with permanent excitation),
- rectifier (to convert the AC to DC),
- slip rings (for the transmission of energy),
- charge regulator (to limit the rate at which the battery is fed),
- accumulator (the battery to store the energy in),
- inverter (for operating mains voltage devices).

For this exercise we will conduct two studies: one for an open loop (for identification of the system) and one for closed loop. For the closed loop we will introduce the charge controller, assuring constant voltage in the output of the wind production system, which is the final charging voltage of the accumulator, which we will use for our control feedback.

Modelling and validation

The block diagram of this energy production system is presented below (Figure 47). Our aim will be to identify and to understand the relation between wind force (input for the system) and energy production (output of the system).

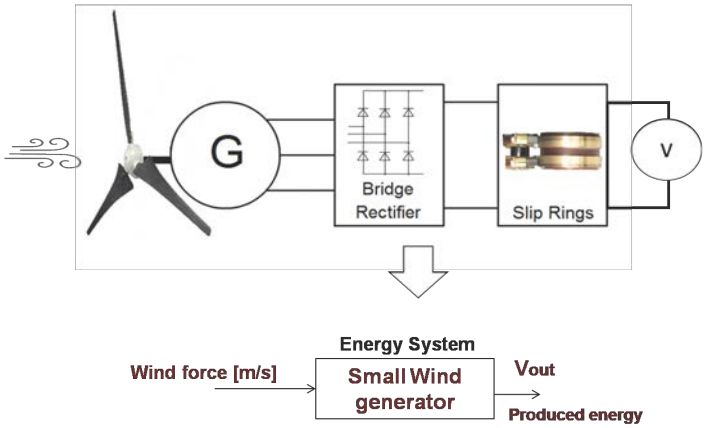


Figure 47. Open loop energy production system: small wind generator.

We will simulate wind with different wind forces and identify the amount of produced energy translated as output voltage, measured on the output of the system. The figure below (Figure 48) presents the actual experimental stand in the laboratory, and how it is connected in reference to real simulation system, based on the scheme in Figure 47.

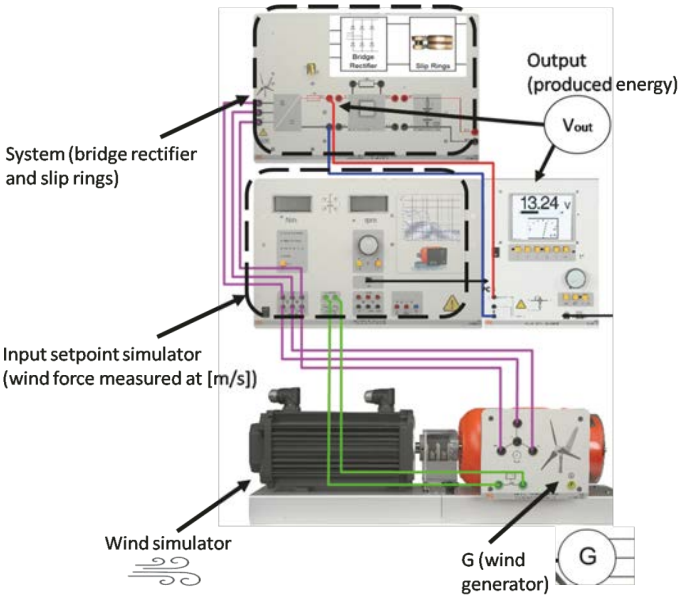


Figure 48. Practical stand of small wind energy production system (open loop): identification procedure.

For the purposes of this exercise, we will apply wind speeds between 0 [m/s] and 12 [m/s], which is the upper limit of the wind speed that is supported by our experimental (laboratory) equipment. If we ran a simple experiment to measure the output of the generator based on the wind speed (at a step of 0.5 or 1 m/s) we can see what is the input to output relation of our generator (Figure 49).

Wind Speed [m/s]	0	3,5	4	5	5,5	6	6,5	7	7,5	8	8,5	9	9,5	10	11	12
V_{out} [V]	0	1,5	10	14	20	22	25	25,5	28	30	32	35	37	41	43	47

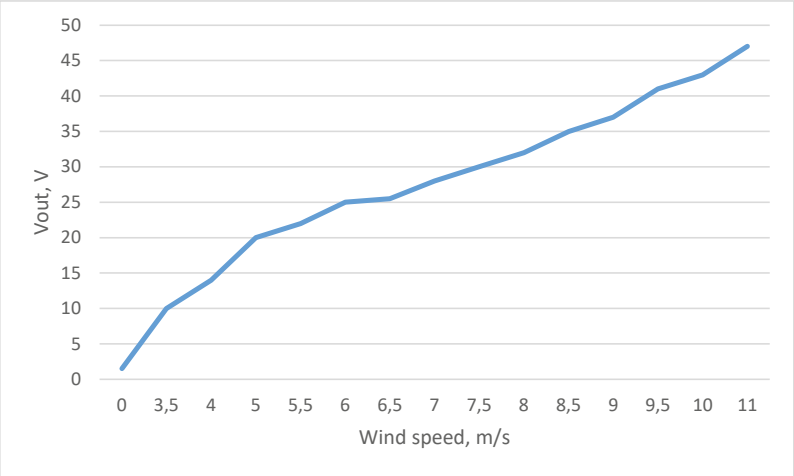


Figure 49. Variation of the output voltage of the wind generator at different wind speeds.

From the results we can conclude that for this open loop system the relation between the wind speed variation and the produced voltage at the output (or electrical power) can be assumed to be linear. With this knowledge we can also assume that the open-loop control system will just use the necessary wind speed as a reference value, expecting to get the desired output voltage.

For the closed loop version of the control system, we will add the charge regulator and the accumulator (battery), as presented on the Figure 50.

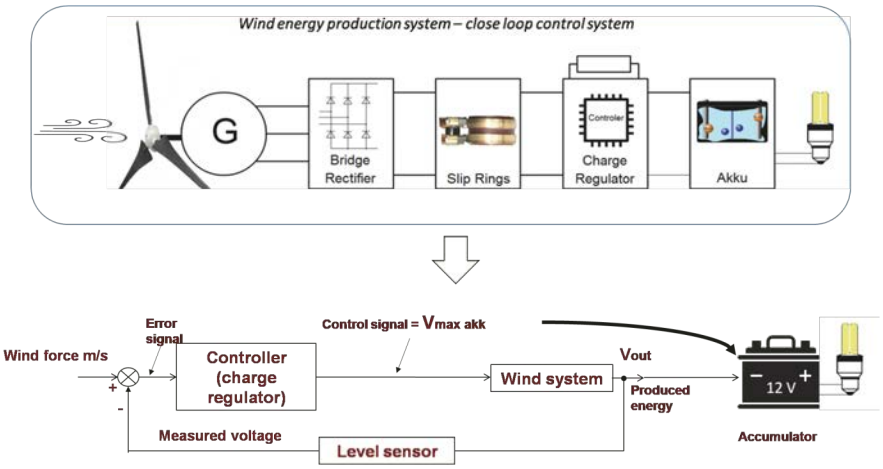


Figure 50. Close loop energy production system: small wind generator.

The charge regulator ensures optimal charging of the batteries. At the same time, it is used for regulating the wind turbine. Energy that cannot be stored by the accumulators is fed to load resistors via PWM (pulse width modulator). Thus, the wind turbine continues to operate correctly under electrical load, even if the batteries are fully charged and the wind speeds are high.

The charge controller operates on the principle of constant voltage (Figure 47). The voltage of the accumulators is limited to the final charging voltage of the accumulators. If the wind is too weak (low alternator voltage), the accumulators are not charged.
The experimental stand is presented in Figure 51.

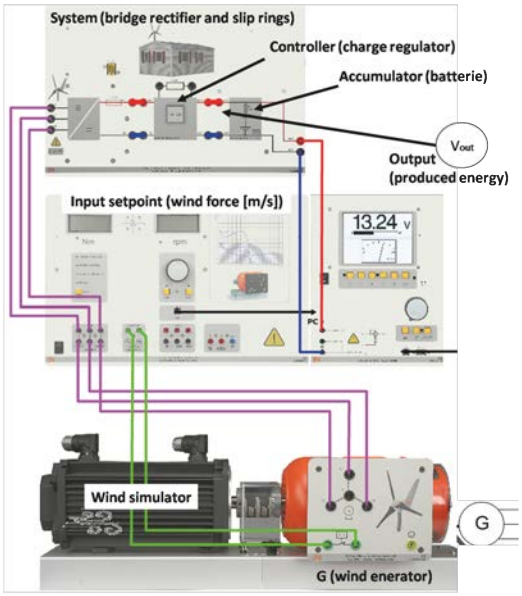


Figure 51. Practical stand of small wind energy production control system (closed loop), with charge regulator and accumulator (storage energy element).

The graphical result from control simulation is presented in Figure 52. On the left is the produced energy and measured voltage in the output of system [V] for the different wind speeds. And on the right is the voltage measured on the output of the accumulator for the same variation of wind speed.

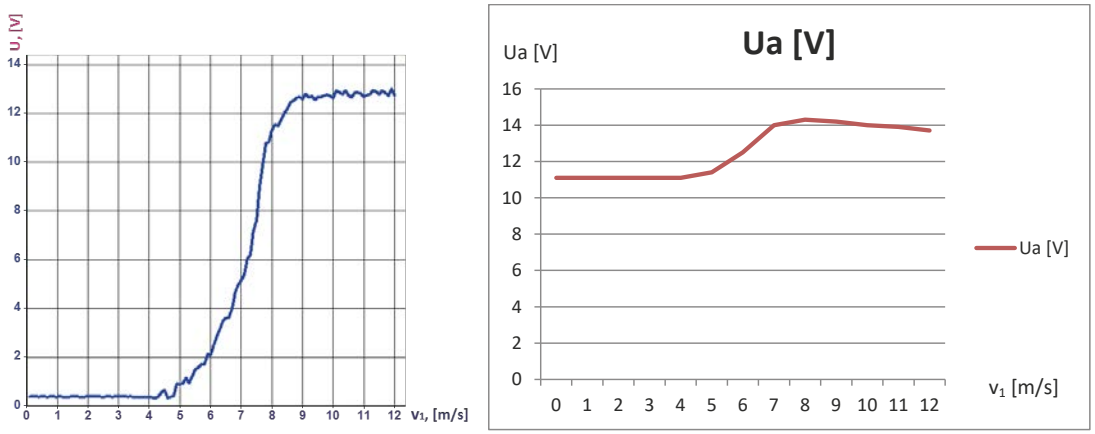


Figure 52. Output voltage related to wind speed variation: output of controller (left) and output of accumulator (right).

It can be concluded that when the wind speed increases, the alternator voltage also increases and is limited by the charge regulator from about 8 [m/s]. In this way the regulator protects the lead-acid accumulator from an overvoltage which will be harmful for it (because starting at a voltage of around 13.8 [V], the accumulator begins to bubble). Here the battery voltage remains almost constant and at high wind speeds increases to the final charging voltage of approx. 14.4 [V]

Chapter V.
Data acquisition, storage and processing. Connecting to sensors and programable logic controllers.

As should be very obvious by now, control systems can't work without data. Either initial (in the case of an open loop) or continuous stream of data (in the case of a closed loop). There are significant differences in what data looks like (data shape and format) between the different applications and methods of acquiring, storing and processing it. As this is not intended to be a deep dive in the topic of data formats, we are only going to go as deep as necessary to get a basic understanding and to facilitate getting our data in a working condition, where it will be in a useful state for our control purposes.

Let's start with the easier case – pre-recorded and already stored data. Almost universally data is stored in tabular format. Usually, features are stored in different columns, where the column (header) name is the name of the feature (parameter, factor, etc). Rows are used for storing different samples or observations, each of which contains a value for the various parameters in the columns. Of course, this may vary depending on the specific application.

Data could be stored in a simple file, like a Microsoft Excel file (.xls or .xlsx), or it could be in a more universal format as a Comma Separated Value file (.csv), which is just a text file that follows a convention that each value is separated by a comma (,) and each new sample (record row) starts on a new line. Of course, although the name suggests that the delimiter is comma, one can use a custom delimiter to separate the values. Imagine your values are whole sentences and they contain punctuation, like commas. That would lead to confusion where a new value starts and ends. So, one can create a .csv file that uses semicolon (;) or some other symbol (a "tab" character for example) to separate the values.

Another way to store already acquired data would be in a database structure. We are not going to go into the differences of different database technologies and architectures, but for the purposes of this exercise book just keep in mind that most databases still use tables for representing data points. Databases provide convenient (programmatic) way to filter and/or request data from them.

On the other side we have steaming data, which usually comes one value at a time, with some known or unknown frequency. Often it might be asynchronous. That means that either the values for the different parameters might not be coming all at once, or following the same order every time. Or for longer data types (for example text, image, video) it might be coming as characters or bites, rather than the whole thing at once. For contrast, with pre-recorded data we can request the whole record with all of its corresponding values, or the entire image, or video all at once, instead of receiving it bit by bit. There are advantages and disadvantages to using either approach, and of course, sometimes we just have to work with whatever the case allows. Usually, the decision goes down to efficiency.

For the purposes of this exercise book, we are going to go through examples that will use pre-recorded data in files, but we are going to mention that live data can be read from sensors. There are many ways to get sensor data, but in an industrial environment almost always sensors will be connected to a SCADA (Supervisory Control And Data Acquisition) system. A SCADA system is usually a software solution running on a computer workstation in an industrial grade computer network, connected to one or more PLCs (Programable Logic Controllers), which in turn are communicating

with the sensors and actuators. The SCADA system usually has a graphical user interface (GUI) which allows the monitoring and control of those sensors and actuators and is also responsible for storing logs of their values (hence the name).

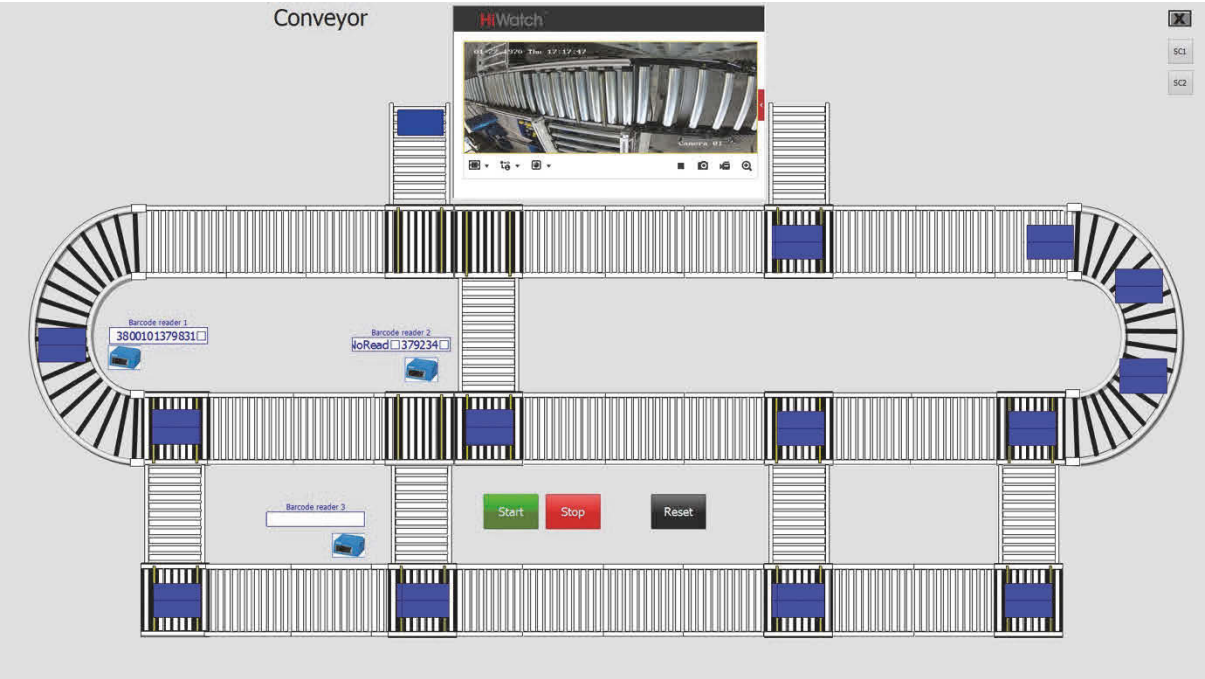


Figure 53. A simple SCADA graphical user interface for a roller conveyor in one of the laboratories of the Technical University in Sofia.

Often a control solution is implemented as logic within the actual PLC, but sometimes if the control solution requires more complex logic than what the PLC can provide as capabilities, the control solution has to be implemented as an additional (higher layer) on top of the existing PLCs and their SCADA.

Reading data, and giving control signals to the actuators often has to pass though the SCADA that is already communicating with the PLC that those sensors and actuators are connected to. Although there are many different architectures and different grades of PLCs and SCADA systems, which provide a plethora of connectivity protocols, almost all modern SCADA systems come with an industry standard way of communication – the OPC protocol.

OPC stands for Open Platform Communication, and its latest unified architecture paradigm allows various industrial systems, regardless of the provider, to communicate with each other, exchanging sensor data and control signals, archived information and various alarms and warnings. We will use the OPC protocol to read data from industrial sensors and give control signals to the connected actuators. But for simplicity we will focus on data you can easily get access to even outside an industrial or academic environment.

When working with data, almost always, the first steps will be loading/reading the data and visualizing it. This will give us better understanding what we are dealing with. We will start with loading data from a local source (file), which is the most common and simplest of options.

V.1. Reading/Loading data from local files

For this exercise we can use almost any data processing software. We can do it in an office suit application, like Microsoft Excel, Mac Numbers, Google Sheets, etc. or we can use a high-level programming language, like MatLab, Mathematica, R or Python, which will give us more flexibility.

We will go with Python, because it is a free-to-use environment, and also the industry standard language for data processing. We will specifically use Jupyter Notebook, which is a Python package that allows us to use line-by-line execution of our code (much like in MatLab's or Mathematica's command line), which makes it ideal for educational purposes and better understanding what each line of code does.

The file we will be working with is the monthly energy consumption records of the Newcastle City Library, for the period between 2011 and 2018 [1].

Thus, our first steps will be to load the file in our working environment and apply the necessary manipulations on it, in order to bring it to the necessary state, to continue with the exercises in the next chapters.

Exercise V.1-1: Load data from a local file and store it in a Python DataFrame variable.

Let's start by importing the necessary Python package for our task. The package we will be using is called Pandas and is the industry standard when it comes to data processing:

```
import pandas as pd
```

The first line of this code imports the package "Pandas" under the alias "pd", for shortened way to address it. Of course, this line of code will only work if Pandas is already installed.

Our next step would be to create the variable that will contain the data and reading the into it. Let's choose the name "energy" (with lower case first letter) for that variable.

Pandas has many available methods (functions) to read data from different sources. The ones usually used for local file are read_table and read_csv. They have their differences, but mostly because of the default parameters. For example, read_table by default expects the delimiter in the file to be a tabulator, whereas the read_csv expects it to be a comma. If we take a quick look at our file, using any text editing software, we will see that the delimiter is a comma ",". So, we can safely use the read_csv method in this instance:

```
energy = pd.read_csv('electricity-consumption-from-2011-2.csv')
```

The second line creates a variable called "energy", which is going to hold the data. We use the pd (Pandas) read_csv method, giving it the path to our .csv file.

If we did not receive any errors, we can assume that the process of reading and storing the data in the variable was successful. But in order to check that with certainty, we can simply ask Python to tell us what the contents of the variable is by simply calling its name:

```
energy
```

	Date	2011	2012	2013	2014	2015	2016	2017	2018
0	January	107777	109117	109582	108664	99481	99216	92638	88817
1	February	105383	110473	98032	96733	92002	90775	83980	83308
2	March	113136	109980	107506	107003	101952	99088	96655	94745
3	April	109962	101340	106052	103672	102460	102398	92196	91480
4	May	113648	112608	112458	110164	107326	106117	120704	108294
5	June	117221	109970	115136	111866	108332	105383	100999	106438
6	July	119465	118610	132643	117742	117825	107382	100498	108504
7	August	122614	126080	126571	111763	120904	106759	101640	104708
8	September	117136	118628	118080	109739	113004	101052	101914	94309
9	October	113141	113414	119439	110934	115316	100852	109249	101912
10	November	108130	107416	103904	105752	105581	95170	100499	94563
11	December	107326	103808	103434	99785	100189	93223	90119	91112

We can see that the data is successfully loaded. We can also see it is a table with 9 columns and 12 rows. A column for the month and then a column for each year between 2011 and 2018. So, we have 8 years times 12 months, or a total of 96 data points to work with for our next exercises.

V.2. Reading data from online sources

Often data resides online, on a server with some specific access policy. It might be a "cloud" service specifically created to collect and serve data resources, or it might be a simple web server, hosting a web page with useful information. Either case, we want to be able to read the data from these places.

V.2.1) Reading data using an API

The ideal scenario would be that the online source for our data has an API (**application programming interface)** which will allow us to directly request the data from the server in a format that that is ready for processing. Usually APIs, in this case a Web API, will allow us to make requests for specific data, or to declare in what form we want the data. Often it will allow for multiple settings to be transmitted in the request, so that data can be filtered before being sent back to us.

For example, lets imagine you want to train a computer vision system to recognize some specific object. In order to make such system, you will need lots of images of that specific object, shot in various angles, lighting conditions, backgrounds, etc. Instead of going and manually searching for such images online, and manually saving them on your computer, and then filtering through every one of them, considering you will need many thousands of them, you could instead use an image search API. So, you can write a script that will make that search for you, while specifying specific setting for it (for example if you want the image to be portrait or landscape format, do you want photos only or paintings and illustrations, resolution, etc.), and then also programmatically pre-process and save those images.

An ideal source for such data would be Google Image search, but unfortunately Google's APIs are not free, and for this exercise we want a source that has a free to use API, so that there are no limitations for you to do this on your own. Thus, for that particular exercise, we are

going to use an image bank (a web site that provides paid or free licensed images), in our case we are going to use **Pixabay.com** API [2] which provides free to use images.

Exercise V.2-1: Write a script to gather images from an online source using an API which that source provides.

We will, as usually, start by importing the necessary libraries, or parts of libraries, which we will need to accomplish our task:

```
import requests
import ipyplot
import pandas as pd
from PIL import Image
```

requests library allows us to make HTTP requests to remote servers;
ipyplot is a small package that offers flexible visualisation of images inside Notebook cells;
PIL is Pillow, a package for image processing. We are only going to need the Image object from it.

Almost all APIs require the use of authentication keys, this is how they track how connects to the server and does what. This is how paid APIs know how much to charge you. Although Pixabay is a free service, and you can use it as a guest, you still need to create a profile in order to use the API.

You can use your own key, which for at least rudimentary privacy you can paste in a text file, and read from there.

```
key_file = open("pixabay_api_key.txt")
key = key_file.read()
```

The first line of code opens the key text file and stores it as an object in the "key_file" variable. The second line reads the contents of the file and stores it (as a string) in the "key" variable.

Now that we are prepared to connect to the API, we need to come up with our request, what exactly do we want from it. It is always advisable to read the documentation [3] of the API that you are going to use. It will tell you what parameters you can use in your requests.

Let's start with the main thing – the search string, this is what you would have typed in the search field of the website if you were to do this manually. For our example we are going to look for "red old car".

The Pixabay website provides all sorts of visual content, photos and video, both naturally photographed and computer generated, they also have hand painted or drawn illustrations. So, we will have to specify what kind of content we are looking for, otherwise we will get a mixture of everything. Let's go for photos only. And we also want to limit the format of the photos to only landscapes (horizontal images)

```
search_string = 'red%20old%20car'
image_type = 'photo'
orientation = 'horizontal'
pretty = 'true'
```

The "%20" string is and HTML representation of blank space (URL encoding). We assign our parameters (in string format) to variables of our own choosing, which we are going to use in the next lines of code, to construct the request URL. The "pretty = true" parameter is important for us, because it will ask the server to transmit the results in a well-structured form, which is easier to read from a human. In a production environment you would set this to "false" to save traffic and processing time.

It is now time to form our request. We are going to create a variable "url" which will hold it. We are going to use formatted string, where we will insert our parameter variables:

```
url = f'https://pixabay.com/api/?key={key}
                                  &q={search_string}
                                  &image_type={image_type}
                                  &orientation={orientation}
                                  &pretty={pretty}'
```

If you call the "url" variable now you will see it is one uninterrupted string, where all the variables are replaced with their content. Now we can use that URL to make the request to the pixabay server. We are going to use the HTTP "get" function from the "request" library:

```
results = requests.get(url)
```

If we now print out the "results" variable we will see it gives us a coded response:

```
results
```

```
<Response [200]>
```

Probably most of you have seen the "error 404 – Not found" on a website when trying to access a page that does not exist. The "200" response we get from the server is another one of these HTTP Status Codes [4]. It means everything went OK and the request was successfully fulfilled.

Our next step is to decode the results and assign them to a variable. Results from APIs usually come in JavaScript Object Notation (JSON) format [5]. That information is available in the API documentation:

```
results_data = results.json()
```

We assign the JSON formatted content of the results object to the "results_data" variable.

If we print out the contents of the "result_data" variable now, we will see the actual JSON string. Here is a small excerpt from it:

```
{'total': 361,
 'totalHits': 361,
 'hits': [{'id': 1197800,
   'pageURL': 'https://pixabay.com/photos/oldtimer-car-old-car-convertible-
1197800/',
   'type': 'photo',
   'tags': 'oldtimer, car, old car',
   'previewURL': 'https://cdn.pixabay.com/photo/2016/02/13/13/11/oldtimer-
1197800_150.jpg',
   'previewWidth': 150,
   'previewHeight': 99,
   'webformatURL':
'https://pixabay.com/get/g7b0da11ee99374c826c62db1c9d82a520184b32744af9c3579a3
8e5edf53db9f02795f60ec0e3a770d35d6ce8c5030a4_640.jpg',
   'webformatWidth': 640,
   'webformatHeight': 426,
   'largeImageURL':
'https://pixabay.com/get/gabf6935b710ad2bbfbf787312a17dfedaf00645aa70d6e96ed25
d625351b52a8c2a7deae9b639f4c71ff6238fb6fe09805ab489c445dbaefdb388cd56ef86910_1
280.jpg',
   'imageWidth': 5760,
   'imageHeight': 3840,
   'imageSize': 10114533,
   'views': 520100,
   'downloads': 385635,
   'collections': 646,
```

```
   'likes': 756,
   'comments': 103,
   'user_id': 2019050,
   'user': 'Noel_Bauza',
   'userImageURL': 'https://cdn.pixabay.com/user/2021/03/27/13-38-22-
879_250x250.png'},
  {'id': 2705402,…
```

I have marked in yellow the first interesting piece of information – the total amount of found photos corresponding to our search criteria is 361 (in my case). We can also see (marked in blue) where the actual results start. They are an array (or a list) of items. We know that because they are constrained in square brackets (the opening bracket is marked in red). Then each element is a dictionary. The opening and closing bracket of the main dictionary are marked in green. Dictionary elements are just pairs of keys and values.

Let's make a test and assign one of these returned results to a variable. Since we know these are images, we will name the variable "image_result".

```
image_result = results_data['hits'][0]
image_url = image_result['largeImageURL']
image_data = requests.get(image_url, stream=True)
```

We take the first element (0 based indexing!) of the ['hits'] list, of our JSON "results_data" variable. The second line of code looks into that one element (which as we established is a dictionary) and looks for the 'largeImageURL' key value and assigns it to the "image_url" variable. Finally, in the third line of code, we use the requests library again to get that image data, from that specific URL.

The data that comes from the "requests" library is in raw format so it needs to be decoded. For that reason, we will use the Pillow library and call the Image.open() method on our raw data, and then store the result of that operation to the object to the "img" variable.

```
img = Image.open(image_data.raw)
```

Let's see what we got as our first result:

```
img
```

Figure 54: The first image from our Pixabay API request, with ID: 1197800.

Let's get a few more of these 361 images (when you do this exercise you might get different number of results). In order to easily visualize the images in a next step, we will first create an empty list, which we will use to store the path and filename of the images we save:

```
image_list = []
```

Now, we need to go through that list of results and extract some of the images. We want to look for their IDs, which we will use for naming them, and also their URLs, where we get the actual image data from:

```
for image in results_data['hits'][:10]:
    name = str(image['id'])
    img_url = image['largeImageURL']
    image_data = requests.get(img_url, stream=True)
    with open(f'images/{name + img_url[-4:]}', 'wb') as output_file:
        output_file.write(image_data.content)
        image_list.append(f'images/{name + img_url[-4:]}')
```

The first line of code opens a for loop that goes through the first 10 [from start : to 10] of the 'hits' in our "results_data" and stores them in a temporary variable "image". The second line of code is in the loop. It looks for the 'id' key in the dictionary of values of that first "image" JSON object, and converts that ID to a string, before storing it in the "name" variable. We then do the same for the 'largeImageURL' of the image, soring it to "img_url" variable. Next, we use that URL to get the raw image data (using requests) and store it in the "image_data" variable. Then we open an empty file in the /images folder (inside our project folder). The name of the file consists of the "name" variable (holding the ID) concatenated with the last 4 [to -4 : from end] characters of the "img_url" (which as you can guess are the extension of the file, for example .jpg). With that file open as "output_file" variable we write the .content of the raw image data in it. The last line of code stores the path and name of the file in the "image_list" we created in the previous step.

If we now look in our "images" folder, in our project folder on our computer, we should see 10 files (probably .jpg) named with the respective ID numbers. We can quickly visualise them in our Jupyter Notebook, using the ipyplot library:

```
ipyplot.plot_images(image_list, max_images=10, img_width=180)
```

We use the "image_list" we created, containing the filenames of our downloaded/saved images, and we ask them to be plotted as a grid of images, with 180 px width per image, and a maximum of 10 images (there should only be 10 anyways).

We can also demonstrate the usage of an API with non-image data with the **Random User Generator API** [6]**.** This is a great tool for generating random user data. One can use it to generate any number of random users and associated data, while also being able to specify gender, nationality, and many other settings.

Exercise V.2-2: Write a script to extract data from an online source using an API which that source provides.

As usual, we start by importing the packages we will use:

```
import requests
import pandas as pd
```

Let's define the API URL and make a request:

```
url = 'https://randomuser.me/api/'
response = requests.get(url)
```

Again, just to check if everything is OK, we can call the "response" variable and see what HTTP response code we got:

```
response
```

```
<Response [200]>
```

If it is [200] then we got whatever the API has to offer. Let's see what the contents of the response looks like when formatted into JSON format:

```
response.json()
```

```
{'results': [{'gender': 'male',
   'name': {'title': 'Mr', 'first': 'آدرین', 'last': 'کریمی'},
   'location': {'street': {'number': 1883, 'name': 'شهید محمد منتظری'},
    'city': 'شهریار',
    'state': 'قم',
    'country': 'Iran',
    'postcode': 31272,
    'coordinates': {'latitude': '12.9858', 'longitude': '-50.4702'},
    'timezone': {'offset': '-4:00',
     'description': 'Atlantic Time (Canada), Caracas, La Paz'}},
   'email': 'adryn.khrymy@example.com',
   'login': {'uuid': '0d437983-e47e-451a-a2ef-209d17381eae',
    'username': 'tinywolf248',
    'password': 'bowser',
    'salt': '4pD3vgMN',
    'md5': 'ec0b76df47f049c3744758d95ae15442',
    'sha1': 'ecfd1fc8130806f18b6abd736c31df5ec7411069',
    'sha256':
'23934f742425f3e670d6679f72ccb2c9288a4c1103a2d18dd855db268f700635'},
   'dob': {'date': '1947-05-02T01:26:57.897Z', 'age': 75},
   'registered': {'date': '2014-02-17T05:49:23.363Z', 'age': 8},
   'phone': '014-25186974',
   'cell': '0982-123-5434',
   'id': {'name': '', 'value': None},
   'picture': {'large': 'https://randomuser.me/api/portraits/men/86.jpg',
    'medium': 'https://randomuser.me/api/portraits/med/men/86.jpg',
    'thumbnail': 'https://randomuser.me/api/portraits/thumb/men/86.jpg'},
   'nat': 'IR'}],
 'info': {'seed': '1a1707455bbf578b',
  'results': 1,
  'page': 1,
  'version': '1.4'}}
```

Since the API gives completely random (synthetically generated, i.e. FAKE) user information, if we do not specify any parameters, we will get exactly that – random user data. The "person" you will get will have completely different personal information than the one I got here.

We can see that the response is again formatted as a list (of one single element, enclosed in square brackets – marked in red), and that element is a dictionary with some of the values being sub-dictionaries. We know that there are sub-dictionaries because of all the other curly brackets (sub-elements) in the main dictionary – marked in teal (for second tier) and yellow (for third tier). You can see the hierarchy by following the text indentation.

For example, the key 'location' has a value which is a dictionary. That sub-dictionary has its own keys ('street', 'city', 'state', etc.), and some of them even have a third level of sub-dictionary elements in them (for example: 'street', 'coordinates' and 'timezone').

We can narrow the criteria for the API responses by supplying some parameters. Let's request the server to get us only female users of (seemingly) French nationality:

```
url = 'https://randomuser.me/api/?gender=female&nat=fr'
response = requests.get(url).json()
response
```

```
{'results': [{'gender': 'female',
   'name': {'title': 'Ms', 'first': 'Eloïse', 'last': 'Louis'},
   'location': {'street': {'number': 4948, 'name': 'Rue Barrème'},
    'city': 'Aulnay-sous-Bois',
    'state': 'Nièvre',
    'country': 'France',
    …
```

You can see we hardcoded the 'gender=female' and 'nat=fr' in the request URL.

Although we can hardcode the parameters in the URL string ourselves, there is another way to do it, which will give you more flexibility (and is easier to read), using a dictionary element, where the parameter name is the key, followed by the respective value. That should give us similar French sounding user, probably a different person, of course:

```
query_params = {"gender": "female", "nat": "fr"}
url = 'https://randomuser.me/api/'
response = requests.get(url, params=query_params).json()
response
```

In the first line of code the "query_params" variable has been assigned a dictionary, where the keys "gender" and "nat" (nationality) have been added with their desired values. Then, in the third line of code we use the requests.get() method by supplying an extra argument "params" which we assign to be our predefined "quesry_params" variable.

Now, let's see how we can make a number of requests to the API, and store the information in a Pandas DataFrame. We can see that the API only returns one person per request (unlike the Pixabay API from Exercise V.2-1, which gave us all results that matched our criteria). That means we need to call the API as many times as "fake" users we want to generate. We can do this with a for loop, that we will call 10 times (to get 10 user records):

```
# Set the URL to the completely random request string
url = 'https://randomuser.me/api/'
users = [] #Create an empty list to store the users on every iteration

for i in range(10):  #We only want to make 10 requests to the API
    response = requests.get(url)  #Make the actual request
    response_data = response.json()  #Store the JSON response

    # Select the first (and only) record in the response
    random_person = response_data['results'][0]

    user_title = random_person['name']['title']  #Get the title value
    user_name = random_person['name']['first']  #Get the name value
    user_family = random_person['name']['last']  #Get the family value

    # Reconstruct the address by converting the street number
    # to a string and concatinating it with an empty space,
    # and then with the street name
    user_address = str(random_person['location']['street']['number'])
    + ' ' + random_person['location']['street']['name']

    user_city = random_person['location']['city'] #Get the city value
    user_country = random_person['location']['country'] #Get country
    user_email = random_person['email']  #Get the email value
    user_photo = random_person['picture']['large']  #Get the photo URL

    #On every step of the loop append the new data to the "users" list
```

```
    users.append({'title': user_title,
                  'name': user_name,
                  'family': user_family,
                  'address': user_address,
                  'city': user_city,
                  'country': user_country,
                  'email': user_email,
                  'photo': user_photo})
```

The comments in the code block should be enough to understand the logic behind our actions, but there are a few things we need to acknowledge. To save time we are only choosing some of the fields to store. You can see the structure of our records for the 'users' list is completely flat and only contains 'title', 'name', 'family', 'address', 'city', 'country', 'email' and 'photo'. Of course, if you want, you can modify the code to include other fields too. We are flattening the sub-dictionary structure to make things easier to read. That means that we are taking the otherwise separate values for street name and street number and slapping them tother in one single string value. We also bring to the surface the 'location' data into their own first-level fields.

And finally, we take the filled in "users" list and convert it into a Pandas DataFrame. Let's see what it looks like:

```
users = pd.DataFrame(users)
users
```

	title	name	family	address	city	country	email	photo
0	Miss	Christy	Daniels	9807 Ash Dr	Seattle	United States	christy.daniels@example.com	https://randomuser.me/api/portraits/women/52.jpg
1	Ms	Adriana	Kircher	8966 Parkstraße	Nienburg (Weser)	Germany	adriana.kircher@example.com	https://randomuser.me/api/portraits/women/26.jpg
2	Ms	Kristin	Osullivan	5506 Church Lane	Sligo	Ireland	kristin.osullivan@example.com	https://randomuser.me/api/portraits/women/2.jpg
3	Mr	Hector	Lambert	9246 Rue Andre-Gide	Lille	France	hector.lambert@example.com	https://randomuser.me/api/portraits/men/24.jpg
4	Mr	Bartel	Lever	5928 Agnietenplaats	Afferden Lb	Netherlands	bartel.lever@example.com	https://randomuser.me/api/portraits/men/47.jpg
5	Miss	Kerttu	Nurmi	6243 Hämeenkatu	Luhanka	Finland	kerttu.nurmi@example.com	https://randomuser.me/api/portraits/women/85.jpg
6	Mr	Väinö	Aalto	2874 Hermiankatu	Karstula	Finland	vaino.aalto@example.com	https://randomuser.me/api/portraits/men/60.jpg
7	Mrs	Zorica	Ivanišević	8140 Trinaestog Oktobra	Nova Crnja	Serbia	zorica.ivanisevic@example.com	https://randomuser.me/api/portraits/women/41.jpg
8	Mr	Laurent	Kreukniet	7279 Kleine Kalverweide	Muiden	Netherlands	laurent.kreukniet@example.com	https://randomuser.me/api/portraits/men/97.jpg
9	Miss	Lotta	Kurtti	8222 Fredrikinkatu	Kerava	Finland	lotta.kurtti@example.com	https://randomuser.me/api/portraits/women/22.jpg

As always, let's save our DataFrame to a .csv file for future use:

```
users.to_csv('10-random-users.csv')
```

V.2.2) Reading data using "raw" HTML code

APIs are great, but not all servers have an API. Often when you need data from an online source you might have to extract it from a web page, which has been created for a human to read and understand. In these cases, you will have to analyse the page (as all pages are different in structure) and write a specific script to read that specific page. The thing is, once you understand how to do it, it becomes relatively easy to make it for any page that you can request via HTTP protocol.

Let's say your system has to rely on some market data which is not readily available through an API or a file export. Or maybe you want data from some sensors which publish their readings online, on a web page, rather than through a programable interface.

Exercise V.2-3: Write a script that can extract and save data from a regular HTML page, and then save it in a Pandas DataFrame for future use.

As before, we have to import the packages we need for our task at hand:

```
import requests
import pandas as pd
from bs4 import BeautifulSoup
```

bs4 is the common name of BeautifulSoup 4, which is a library for parsing HTML and XML structured data. We will use the main library, called just BeautifulSoup.

Next, we need to decide which web page to target. For this example, let's assume we need a dataset containing the top picks amazon will give us, if we search for "iphone". Our first task would be to access that search manually and see the page that our browser will give us:

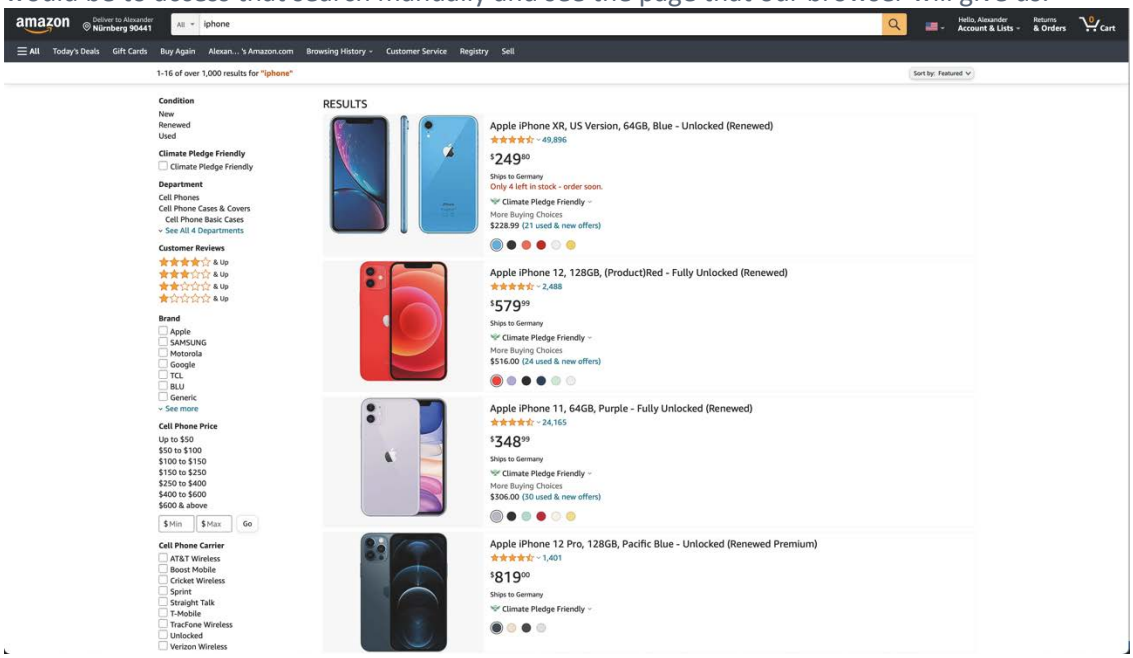
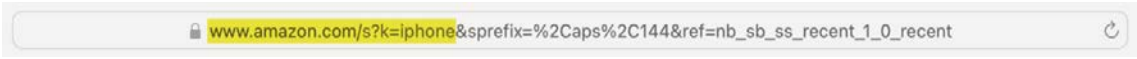


Figure 55: here is what I got at the moment of writing this.

Obviously, you will get a different set of results. Also, depending on when you access Amazon's web site, it might even look entirely different. But however it looks, make notice of the web address your browser shows:



That is what we need to use as a URL for our requests.get() function. We don't have to use the entire url, as the information there is irrelevant to our task. We just need the search term (s?k=iphone). So, let's define it in a url variable as we usually do:

```
url = 'https://www.amazon.com/s?k=iphone'
```

Before we ask Amazon to give us the result of this online search, we need to have one more additional step. Unlike APIs, which are specifically built to accept requests from a machine, Amazon tries to block unnecessary traffic, and will not give you results, unless you look like a real user, browsing the web site. You can easily test that if you skip the "header" variable bellow:

```
header = ({'User-Agent': 'Mozilla/5.0 (X11; Linux x86_64)',
           'AppleWebKit/537.36 (KHTML, like Gecko)'
           'Chrome/44.0.2403.157 Safari/537.36'
           'Accept-Language': 'en-US, en;q=0.5'})
response = requests.get(url, headers=header)
```

We define a "header" variable, containing the header text, which will get transferred with your request URL. What we are doing with it is impersonating a browser, so that Amazon thinks the originator of the request is a real user. The second line of code it the actual request, but you can see it now includes a header, that takes it value from what we have defined above it.

Let's check if this trickery worked and Amazon gave us the necessary HTML page:

```
response
<Response [200]>
```

It seems everything is OK. Next, we need to parse the result, by telling the BeautifulSoup library that it is looking for an html structure:

```
parsed_response = BeautifulSoup(response.text, 'html.parser')
```

We pass the .text content of the response to BeautifulSoup (while telling it to use an html parser) and the result of this operation is stored to the "parsed_response" variable.

And here comes the forensic analysis of the HTML page, that we need to do, in order to find out what the structure is, and where the different element of it (which we are interested in) are contained. We are specifically looking for ways to identify those tags that hold these assets (text, numerical values, images, etc.).

To keep this example relatively simple, let's just focus on few pieces of information, that we will be recording. We want to have the name of the listing, the price, the rating of the seller and the base the rating was calculated for (how many people voted).

We will need to look at the HTML source code of the loaded page (in our browser) and look for the container of the content that we are interested in, so that we can isolate it from the rest of the information on the page. All modern browsers have developer tools which we can use. Usually, you can just right click on a page (or specific element) and select "Inspect Element". We will be specifically looking for the "id" or "class" values of the different "containers".

To test out our approach, we can first try to only extract the top-level container, that has all the elements in it.

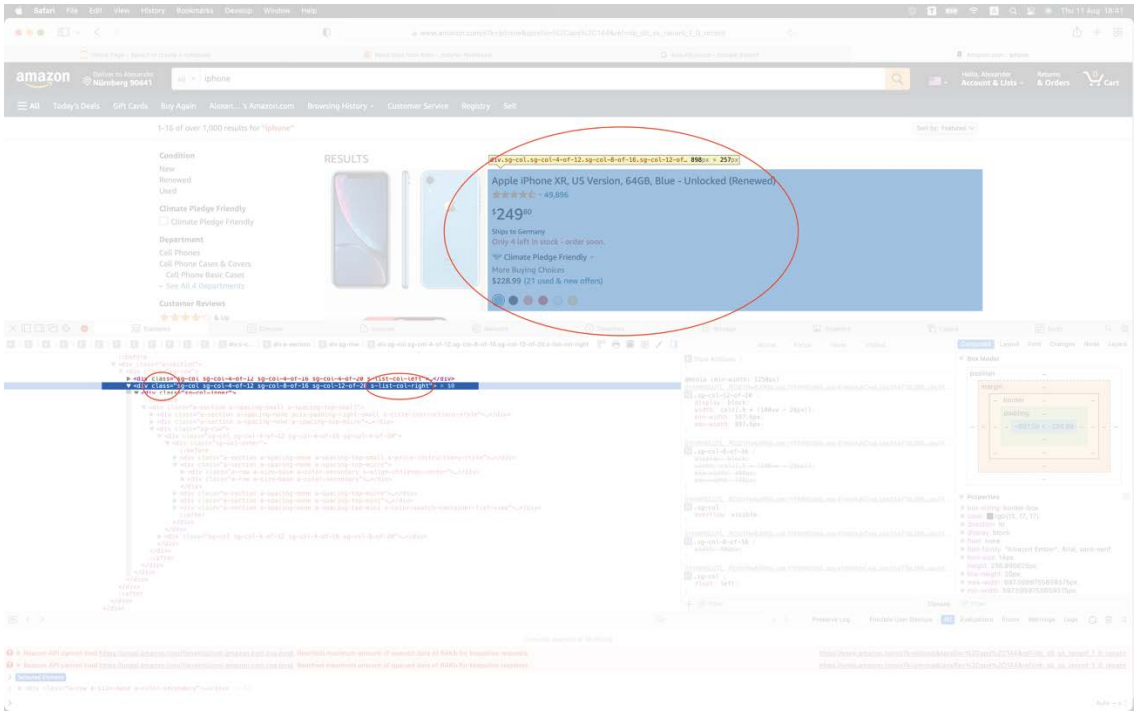


Figure 56: The image shows the top level that contains all the pieces of information we are interested in.

After some digging, we find out that all the information we are interested in is contained in a element with many associated classes, but one of them sounds meaningful "s-list-col-right". We can assume it means something like "search list column right", which as we can see is exactly where our target information is. So, let's take a new variable and place all elements that have this class in it:

```
listed_items = parsed_response.findAll('div', class_='s-list-col-right')
```

This line of code finds all the 'div' elements in the "parsed_response" that have a class of 's-list-col-right' and places them in the variable called "listed_items".

Next, we need to go through the sub elements of that element and find the ones that contain what we need. Let's first create an empty list to store the information we find.

After some digging, we find out the following 4 containers and their selectors:

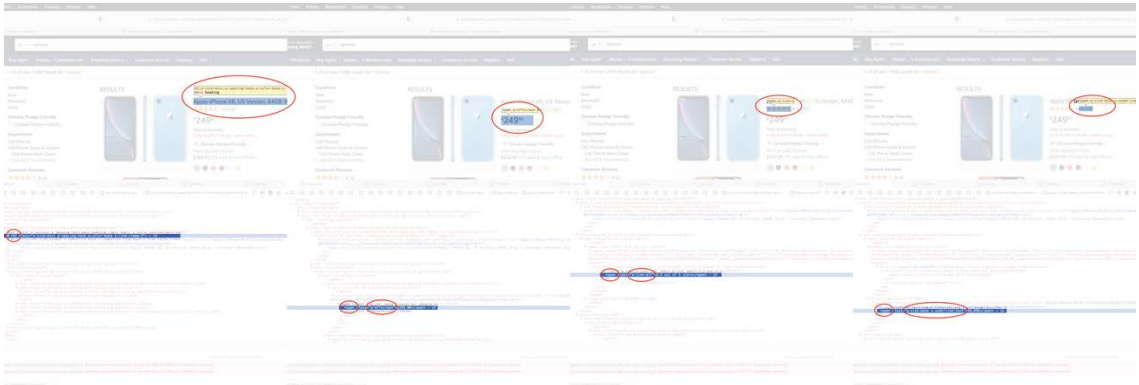


Figure 57: Screenshots from the HTML inspector in the browser, showing the necessary tags and their classes.

We found out that the name (title) of the listing is always enclosed in a <h2> tag (which is exactly what this tag is meant for – titles). So, we safely target it directly without the need to narrow down the filtering, by specifying classes.

The price is enclosed in a tag, but since there are a lot of different tags in the page, we have to narrow down by using the class of the element. The price has a class of 'a-offscreen'.

Next is the rating. Although the rating is not given in text (or number) on the page, but with stars, we can still extract meaningful info. Because the designers of the page made it to be understood by systems for the visually impaired. So, we find a tag with a class of 'a-icon-alt' which contains exactly what we need.

And finally, the rating base we find in a with classes 'a-size-base' and 's-underline-text'. We are going to use both:

```
items = []
for element in parsed_response.findAll('div',class_='s-list-col-right'):
    items.append([
        element.find('h2').text,
        element.find('span', class_='a-offscreen').text,
        element.find('span', class_='a-icon-alt').text,
        element.find('span', class_='a-size-base s-underline-text').text])
```

The first line of code creates an empty 'items' list to store the results. Then we use a for loop, which iterates through the elements in the "parsed_response" that are of type and have a class of 's-list-col-right' and store them in a temporary variable "element". Then we start appending the "items" list we previously created with the values of the element we found.

Finally, we can convert the list we filled with information to a Pandas DataFrame, using the "title", "price", "rating" and "rating_base" as column names:

```
phones = pd.DataFrame(items, columns=['title', 'price', 'rating',
'rating_base'])
```

We take the "items" list and pass it through the pd (Pandas) DataFrame object creator (by also specifying the necessary column names). The result of this operation is stored in the "phones" variable.

Let's call out our new DataFrame and see the results of our operations:

```
phones
```

	title	price	rating	rating_base
0	Apple iPhone XR, US Version, 64GB, Blue - Unlo...	$249.99	4.5 out of 5 stars	49,450
1	Apple iPhone 12 Pro, 128GB, Pacific Blue - Unl...	$829.00	4.3 out of 5 stars	1,352
2	Apple iPhone 11, 64GB, Purple - Fully Unlocked...	$358.99	4.4 out of 5 stars	23,696
3	Apple iPhone 12, 128GB, (Product)Red - Fully U...	$579.99	4.3 out of 5 stars	2,366
4	Apple iPhone 11 Pro, 256GB, Midnight Green - U...	$554.99	4.3 out of 5 stars	12,672
5	Apple iPhone X, US Version, 64GB, Silver - Unl...	$249.99	4.2 out of 5 stars	20,298
6	Apple iPhone 8, 64GB, Gold - Unlocked (Renewed)	$189.99	4.4 out of 5 stars	46,156
7	Apple iPhone 12 Pro Max, 128GB, Graphite - Ful...	$830.00	4.3 out of 5 stars	944
8	Apple iPhone 13, 128GB, Red - Unlocked (Renewe...	$814.99	4.6 out of 5 stars	297
9	Apple iPhone SE (2nd Generation), US Version, ...	$199.99	4.5 out of 5 stars	5,480
10	Apple iPhone 11 Pro Max, 256GB, Space Gray - U...	$649.95	4.4 out of 5 stars	10,406
11	Apple iPhone 7, 32GB, Silver - Unlocked (Renew...	$139.95	4.2 out of 5 stars	49,291
12	Apple iPhone 12 Pro Max, 128GB, Pacific Blue -...	$939.00	4.5 out of 5 stars	209
13	Apple iPhone 8 Plus, US Version, 64GB, Gold - ...	$229.99	4.4 out of 5 stars	22,798
14	Apple iPhone 12 Mini, 64GB, Purple - Unlocked ...	$529.99	4.4 out of 5 stars	2,615
15	Apple iPhone XS Max, US Version, 64GB, Space G...	$339.95	4.4 out of 5 stars	13,857

And, of course, don't forget to save your results to a .csv file for later use:

```
phones.to_csv('phones-on-amazon.csv')
```

V.3. Rudimentary pre-processing

Usually, before the data can be passed through the processing pipeline and used for modelling, it has to be pre-processed. That might involve:

- **Cleaning** – removing of unnecessary pieces, fields and/or samples of data;
- **Rearranging and reordering** – changing the order of columns and/or rows, or even transposing the data-set;
- **Reformatting** – for example converting from one date-time format to another, changing the decimal symbol, dealing with text encodings and so on;
- **Mapping and encoding** – most often when categorical linguistic values have to be translated to numerical representations;
- **Scaling** – usually necessary when getting raw voltage data from analog sensors that needs to be converted to the specific units of the phenomenon the sensor measures.
- **Normalisation and standardisation** – technically speaking normalisation and standardisation are a form of scaling, but using them is most often related to bringing different measured phenomena to the same (and often between 0 and 1) numeric scale, so they can be used to make sure that one data field does not dominate (outweigh) the others in the modelling phase.

We are going to pay a bit more attention to the Mapping/Encoding and the Normalisation/Standardisation methods, as they are clearer in terms of their application, unlike all other potential processes, which are much more diverse and case specific. We will, however, see examples of all the above as we progress with the exercises in this book.

V.3.1) Mapping and Encoding

Mapping and encoding are related concepts. Mapping refers to the process of associating elements from one set to elements in another set, establishing a relationship between the elements of the two sets. The purpose of mapping is to define how the elements correspond to each other. For example, we can have the following two sets of values:

```
set1 = [excellent, very good, good, average, poor, terrible]

set2 = [1, 2, 3, 4, 5, 6]
```

The process of mapping one set to the other will create associations between the linguistic values and the numerical values. The mapping can be one-to-one, one-to-many, many-to-one and many-to-many. For the purposes of this simple example, let's imagine we want a one-to-one mapping. That means one element in `set1` is mapped to only one element of `set2`, like this:

```
 excellent = 6
 very good = 5
      good = 4
   average = 3
      poor = 2
  terrible = 1
```

By creating the mapping between the two sets of values we've also encoded them. This means we converted the information from one format to another. Encoding involves representing the data using a specific scheme or method to make it suitable the specific process we want to use it for (storage, transmission, calculation, etc.). Usually, one format works better for the specific application than another. For example, for machine learning (calculations) we need the data to be numerical and not in categorical text format.

There are many different encoding methods, and they all have their advantages and disadvantages. We are only going to mention three of the most often used ones and see how to use them in practise.

a) Ordinal Encoding

What we just did with the two sets from our mapping example is called Ordinal Encoding. During this process we map each level (presented as a linguistic value) to an individual number. But also, as the name suggests there is order in the set – "excellent" is higher than "very good", so the numerical value of "excellent" has to also be higher than that used for "very good". There are a lot of details we will not cover here, but you have to keep in mind, what increments are assumed in the encoded set and how you can accommodate further granularity if necessary.

b) One hot encoding

One-hot encoding is one of the first encoding schemes we can think of when we see categorical values. It is useful when there are only few categories for that field of the data-set. It maps each level of a category to its own column, and each record (row) is either in that category (true) of not in that category (false). We use 1 to represent "true" and 0 for "false".

For example, think about the field (column) "gender" in a data table. Depending on the type of research the data came from, that field might contain just a handful of potential values. For simplicity, let's imagine only two categories were recorded "male" and "female". If we want to encode them as numbers, we can of course use ordinal encoding and simply encode one of the values as 1 and the other as 2. But now, think about those numbers from a calculation perspective. Because there is ordinality in the encoded values, that automatically implies one of them is "more" (higher) than the other (2 is greater than 1). That can bias your model significantly. Using one-hot encoding can mitigate that risk. Of course, as we mentioned, if you do that with a field that can take many different values (categories) that will lead to the artificial generation of as many new columns in the data set as potential categories there were. Also, because one category is true for each sample, a lot of artificial zeros will be recorded to fill in for the "false" categories. That might have other unwanted effects down the calculation road due to the sparsity of the values.

c) Target and James-Stein encoding

These encodings use knowledge of the target variable to do the encoding. Target encoding replaces each category with the average value of the target for rows with that category.

Imagine we had a column in our data-set named "Profession". It can take many different values, so using one-hot encoding will create too many new columns and introduce a lot of sparsity. Also, we don't want to create artificial bias by making the model think one profession is "better" than another, so using ordinal encoding is also not a good idea. What we can do is search for a target that already is representative for those categories. For this example, the target might be the salary for each profession (or, more precisely, the average salary for every specific profession recorded in the data-set). So now, we can replace each "teacher" value in the profession column with the average salary for teachers.

V.3.2) Normalisation and Standardisation.

Both processes apply scaling to the features (the parameters describing the state of the system) aiming to make sure they are on almost the same scale so that each feature is equally important passing them through a machine learning algorithm. It is important that you understand the difference between the two operations and why we need them.

a) Normalisation

As we mentioned briefly in the beginning, normalisation is useful (and in fact almost mandatory) when we need to use values of different order of magnitude in the same optimisation operation.

Imagine we are trying to create a model that can forecast the energy usage in a building, based on a variety of parameters of the building and some inhabitancy factors. Let's say the two most important are the year the building was built (maybe it has to do with the quality of materials and insulation) and the occupancy rate at the moment of prediction. We can imagine that the year value is in the thousands whereas the occupancy rate is between 0 and 1 (where 0 means the building is empty, or 0% occupied, and 1 means full, or 100% of people are in their rooms/at their workplaces).

I hope you can see how the drastic difference in the numbers' order of magnitude means even a relatively minor change in one factor, let say 10 years difference for the building construction (which is almost negligible when comparing buildings) massively outweighs even a relatively huge change in the other factor, let's say a difference of 0.5 (from completely empty to 50% full building).

To avoid feeding these raw values in the model and risking biasing it towards factors with naturally higher values, we can normalise all values to use the same scale -> 0 to 1.

Consider the occupancy factor we just used as an example. Behind those percentages there is some specific information, may be number of work desks used out of all available desks. In order to get those percentage values, they were normalised. Let me show you how to normalise the years for the building construction as well.

Let's imagine we have the following buildings in our data set:

Building #	Year of construction
1	1890
2	1903
3	1941
4	1972
5	1999
6	2007
7	2016

The same way the minimum number of occupants means 0% and the maximum number means 100%, we can say that the oldest building is built at the 0th % of the timeline and the newest is at the 100th % of the timeline. Then everything in between has to scale in that timeline. We can use the following formula to do the calculations for every year:

$$X_{normalised} = \frac{X_{actual} - X_{min}}{X_{max} - X_{min}} \quad (V.3\text{-}1)$$

So, for building number 4 that calculation would like this:

$$Normalised\ year_{building\ 4} = \frac{1972 - 1890}{2016 - 1890} \approx 0.65 \quad (V.3\text{-}2)$$

Basically, we take the position of the value on the timeline, or how far away from the beginning we are (which is the actual year minus the start/minimum year) and we divide by the range of our timeline (which is the difference between the max and min years). That gives us the percentage of the timeline for that position.

We could normalise the year values based on the information in the data-set, or we can make some assumptions. For example, we might encounter buildings that are older than 1890, so we might want to set the start of the scale (minimum year) further back the timeline. We can also expect newer buildings than 2016, so we might want to use the current year as the max year. Regardless of how we decide to approach the normalisation, the result will always be a number in the range between zero and one [0,1].

b) Standardisation

Standardisation on the other hand means to scale the data in such a way, that the mean of all values is set to zero (0) and the standard deviation becomes one (1), which is also known as a unit variance. In order to achieve that the feature values are transformed by subtracting the minimum (the beginning of the scale) and dividing by the standard deviation. You can often encounter the result of this process described as a Z-score.

$$X_{standard} = \frac{X_{actual} - \mu}{\sigma} \quad (V.3\text{-}3)$$

where μ is the mean of the values we are standardising and σ is the standard deviation from the mean. By standardising (or Z-score normalizing) the values are rescaled in such a way, that they'll have the properties of a Gaussian distribution (with $\mu = 0$ and $\sigma = 1$).

For an example, let's take the IQ (intelligence coefficient) of a random population. This is a phenomenon for which we know to expect a normal distribution. In other words, we know the average (mean value) is IQ of 100 and then in both directions there are less and less people with progressively higher or lower IQ scores.

If we wanted to use that data in a model with another normally distributed phenomenon, let's say the exam scores of students, which has completely different scale (for example 2 to 6 for most of Europe), then it makes sense to standardise the two sets so that they are both on the same scale. We do that exactly for the same reasons as with the normalisation example above.

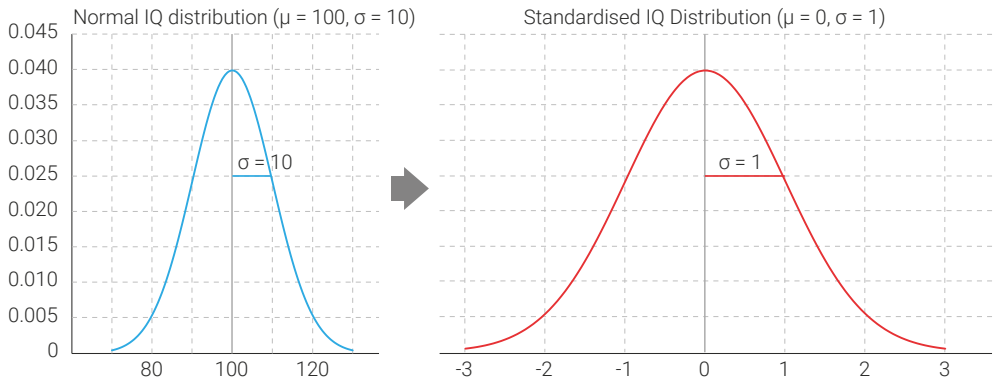


Figure 58. Example of using standardisation (Z-score normalisation) rescaling to convert the IQ values in a data set to a standard scale with a mean of 0 and a standard deviation of 1

Most often we will use normalisation rather than standardization. The main reason for that is that standardisation assumes that the data we are working with is normally distributed (as we need the mean and the standard deviation to calculate a standardized value), and more often than not, the data we will need to work with is not normally distributed.

V.3.3) Indexing, Rearranging and Reshaping arrays

Almost always, when working with data, we will be storing it in some specific data structure which we will be choosing conveniently for the specific application (or data type). Usually it will be a matrix (2 or more dimensional array), list (1-dimentional array or vector) or dictionary (a key and value pairs). Although different software products, or even different packages and libraries within these environments, might have different ways to represent some data structures, some things are universal across environments and platforms.

Each of the exercises in the book explain various ways to work with these data structures, and it surely makes more sense for you to understand the operations in context, rather than doing isolated experiments here.

We already started working with lists, and dictionaries (as well as Pandas DataFrames) in the previous exercises, so you should already have some understanding. But to continue with the

next topics we need to prepare one of our datasets by transforming and rearranging the data in it. So, let's do that.

Exercise V.3-1: Load the Energy Consumption data from Exercise V.1-1 and rearrange it in a timeseries format (labelled column vector). Convert the dates to datetime type records for further ease of use.

In Exercise V.1-1 we loaded the dataset in its original export format. You remember it had 9 columns and 12 rows. We now need to apply the necessary manipulations on it, in order to bring it to the state necessary for further processing.

When working with data, almost always your first steps will be loading the data set and visualizing it. This will give you a better understanding what are you dealing with. So, to start we need to read the .csv file into a data-frame variable. We will again use Pandas:

```
import pandas as pd
energy = pd.read_csv('electricity-consumption-from-2011-2.csv')
```

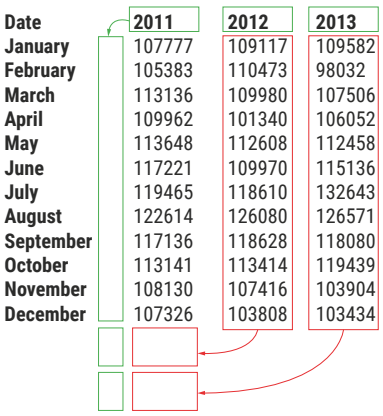
Since we are discussing timeseries, and in order to process the data further, we will need to rearrange it, so that from a 2-dimensional array of size 9×12 (1 row of dates and 8×12 data values) it has to become a 1-dimentional array (1×96 of only the energy values). So, we need a column for every date and just one row of data values.

For efficiency purposes we will do all the manipulation in an empty Python list, rather than in the Pandas DataFrame, and when we are done, we will convert the list to a DataFrame object. Thus, our first step is to create an empty list:

```
energy_reshaped_list = []
```

Create an empty list to store the newly reshaped data.

Then comes the interesting part. We need a way to populate that new list with 96 other lists, containing two values, one combining the month and the year, and one for the actual energy consumption for that date (month). You can think of it as a large list (of size 1×96) with nested 96 small lists with two values (so, a size of 1×2).



Date	2011	2012	2013
January	107777	109117	109582
February	105383	110473	98032
March	113136	109980	107506
April	109962	101340	106052
May	113648	112608	112458
June	117221	109970	115136
July	119465	118610	132643
August	122614	126080	126571
September	117136	118628	118080
October	113141	113414	119439
November	108130	107416	103904
December	107326	103808	103434

Date:	Jan. 2011	Feb. 2011	Mar. 2011	Apr. 2011	May. 2011	Jun. 2011	...	Sep. 2018	Oct. 2018	Nov. 2018	Dec. 2018
Value:	107777	105383	113136	109962	113648	117221	...	94309	101912	94563	91112

Figure 59: We need to go from a matrix representation of months and years (top part of the figure) to a time-series representation – a value for each month (bottom part of the figure).

There are many approaches to this task, but we will go with a rather clumsy method, using "for" loops. The reason for this is that you need practice understanding loops and this is a great opportunity for that. Not only we are going to use loops, but we will use nested loops:

```
# Iterating through all the years (the columns)
for current_year in energy.columns.values:
    # Iterate through all the months (the rows in in every column)
    for i, current_month in enumerate(energy['Date']):
        column_name = str(current_month + ' ' + current_year)
        energy_reshaped_list.append([column_name, energy[current_year][i]])
```

First, we iterate though all the years. We do this by accessing the columns.values of the energy DataFrame. Then while the first year (column) is selected, we use a second "for" loop to iterate through all the rows (months). In that second loop we use two variables – "i" for the index of the row and current_month for the actual string value of the month's name, which is contained in the energy['Date'] column. Then we create a new variable called column_name which we regenerate on every iteration to contain the current_month (coming from the bottom-level loop) concatenated with the current_year (coming from the top-level loop). Then we address each element of the energy DataFrame, using the name of the column (year) and the index of the row (energy[current_year][i]) and together with the newly formed column_name we append them to the empty energy_reshaped_list.

We can see wether our code worked as intended by calling the name of our reshaped list:

```
energy_reshaped_list
```

```
[['January Date', 'January'],
 ['February Date', 'February'],
 ['March Date', 'March'],
 ['April Date', 'April'],
 ['May Date', 'May'],
 ['June Date', 'June'],
 ['July Date', 'July'],
 ['August Date', 'August'],
 ['September Date', 'September'],
 ['October Date', 'October'],
 ['November Date', 'November'],
 ['December Date', 'December'],
 ['January 2011', 107777],
 ['February 2011', 105383],
 ['March 2011', 113136],
 ['April 2011', 109962],
 ['May 2011', 113648],
 ['June 2011', 117221],
 ['July 2011', 119465],
 ['August 2011', 122614],
 ['September 2011', 117136],
 ['October 2011', 113141],
 ['November 2011', 108130],
 ['December 2011', 107326],
 ['January 2012', 109117],
 ['February 2012', 110473],
 ['March 2012', 109980],...
```

It seems our nested loops did the job. But there is a small problem. Because we were taking each cell from the original DataFrame, that means out first 12 mini lists contain meaningless records. If you remember our DataFrame contained 9 columns, but only 8 were years and 1 was just month names. So now the reshaped list has 9×12 instead of 8×12 values. They have been marked in red in the output above. We need to remove those from our new reshaped list before we can continue.

OPTIONAL EXERCISE: Try to modify the loops so that we don't have these first 12 meaningless records in our reshaped list.

Again, there are many ways to get a list where the first "n" elements have been removed, but the most efficient one would be this:

```
del energy_reshaped_list[:12]
```

What this code does is to delete the elements starting from the start and up to element twelve. We could have skipped that step with a simple edit of the outer "for" loop in the previous step, omitting the first column, that generates those unnecessary combinations. Can you think what that edit could have looked like?

If we call again the reshaped list, we should see that it now starts directly from the January 2011 value of 107777 Wh.

```
energy_reshaped_list
```

Our next step will be to convert the Python list we were using back to a Pandas DataFrame, in order to later continue working with the data:

```
energy_reshaped = pd.DataFrame(energy_reshaped_list,
                               columns=['Date', 'Energy'])
```

We are creating a new variable called energy_reshaped and assigning to it a Pandas (pd) DataFrame object that is formed by our "energy_reshaped_list", and using the column names "Date" and "Energy" (because we know the list contains a pair of values for each.

If we now call the newly create 'energy_reshaped' DataFrame, we should see exactly the result we were expecting:

```
print(energy_reshaped.columns.values)
energy_reshaped.head(10)
```

The first line of this code calls the values of the columns of the "energy" DataFrame. The second line calls for the first (10) records (from the head) of the DataFrame.

```
['Date' 'Energy']
```

	Date	Energy
0	January 2011	107777
1	February 2011	105383
2	March 2011	113136
3	April 2011	109962
4	May 2011	113648
5	June 2011	117221
6	July 2011	119465
7	August 2011	122614
8	September 2011	117136
9	October 2011	113141

The result of this code tells us that the values in our reshaped list were correctly converted to a Pandas DataFrame, with two columns (one "Date" and one "Energy"). We can also see the results of the second line of code, which is a table containing the first 10 records of data in the DataFrame.

Our next step will be to manipulate the Date column records. We know the "Date" column contains dates (in our case a month name and a year), so it is a good idea to make sure Pandas understand that these values are dates (datetime type data), rather than just some random text (string type data). To do so, we can ask Pandas co convert the contents of that column to datetime type data, using the following code:

```
energy_reshaped['Date'] = pd.to_datetime(energy_reshaped['Date'])
```

We are asking Pandas to replace the contents of the "Date" column with the result of the function "to_datetime", which we feed with the current contents of that same column of values.

What we do that way is to replace the "Date" column of our "energy" DataFrame with the result of the function "to_datetime" which takes the original content of the column and parses it looking for date/time data. The function can take other arguments, like for example the specific format of the fields, which in our case is not necessary. To test that the operation worked fine we can invoke one of the fields, but request Pandas to tell us only the year of the field. If the operation of converting them went OK we should get a correct answer.

```
energy_reshaped['Date'][13].year
```

Running the above line asks for the 13th row (technically the 14th, as the data frame is indexed starting at 0 for the first row), which naturally gives 2012. In fact, we can easily conclude that if we ask for just the month we should get February, as it is the second month after the 12 months from the first rows (i.e., 14 rows of months).

```
energy_reshaped['Date'][13].month_name()
```

Or, we could have asked not for the month's name, but just the month number, which would have resulted in an integer representation of the month.

```
energy_reshaped['Date'][13].month
```

Now, that the data is stored in a DataFrame of the correct (suitable) dimension, and the dates are in a format (data type) that Pandas understands, we can start really asking and answering questions related to the data. But before that it might be wise to save the newly reshaped DataFrame in a new file, so we don't have to go through the reshaping process again when we next want to load and work with it.

```
energy_reshaped.to_csv('electricity-consumption-from-2011-2-PREPROCESSED.csv')
```

We use the "to_csv" method of the DataFrame to convert and save it. I have chosen to use the original filename, but also adding an additional word at the end to remind us this is the pre-processed version. This will give us a lot of flexibility for the next exercises.

Chapter VI.
Forecasting (timeseries)

Usually, forecasting is done by building a mathematical model of the process you want to forecast, using historical data. For example, if you want to know how much electric power you are going to need next month, you could just see how much you used the previous months. Of course, that approach is a bit too simplistic, as there are a lot of factors that might influence your power consumption.

From a supply chain management perspective, and specifically value creation or optimisation, forecasting is a vital function. Forecasting gives a company the opportunity to take advantage of cost savings, inventory reductions, and performance improvement opportunities, which would be unavailable without it. Even though, sometimes, forecasting is not formally described as a reoccurring managerial activity and responsibility, in fact it accounts for a substantial part of the time spent for operational management in any company. Planning budgets, planning schedules, planning any form of resource spending, even if done in your mind, requires forecasting. It might be purely based on intuition, but then again, intuition is just deciding by experience, or in other words - using historical data.

Usually, such historical data is recorded in certain intervals. These intervals can vary, depending on the process and kind of data being recorded. You can record some values every year, every month, every day, hour, minute or second. Obviously, the longer you record data, the more accurate your model might be.

Such interval data is also known as time series. The reason for this is that the physical process, that the data is coming from, is observed as progress over a period of time, thus we will have a series of values over some passed time. These time series represent an ordered sequence of values of the given (process) variable, offset (measured) at equal intervals of time. The particular variable can be almost any physical quantity (e.g., temperature, pressure, concentration of a substance, the intensity of a flow, population size, etc.).

Interval data analysis takes into account the fact that individual measurements taken at different time points often have inherent structure (e.g., autocorrelation, trend, or seasonal variation) that needs to be investigated.

Using models with interval data has a dual application. On the one hand, they provide an opportunity to gain insight into the structure of the data and to understand the mechanisms by which they arise, and on the other hand, a given distribution model can be applied and forecasting and management can be undertaken.

To demonstrate the various ways, we can approach forecasting we are going to use the public dataset of the energy consumption of the New Castle City Library, recorded every month between 2011 and 2018, published on the https://data.world repository [1]. If you have done the exercises from the previous chapter, you should remember we already have done some pre-processing on this dataset.

VI.1. Average

Let us start with determining some averages. We can ask a few questions here, and although using a simple average would be considered overly simplistic in most scenarios, this is a good first step in forecasting. Let's go over some hypothetic scenarios:

1) How much energy is the library going to consume next month?

2) Can we achieve better accuracy if we take an average for specific months, rather than all months?

The average value is calculated by summing all interval values and dividing the sum by their number (i.e., the number of intervals). We can formulate this as follows:

$$\bar{y} = \frac{\sum_{i=1}^{n}(y_i)}{n} \qquad (VI.1\text{-}1)$$

Exercise VI.1-1: Use the average of historical energy consumption data, to forecast future consumption, using Python.

Let's start by loading the additional packages we will need. Other than Pandas, we will need the following libraries:

```
import math
import pandas as pd
import matplotlib.pyplot as plt
from statistics import mean
from sklearn.metrics import mean_squared_error
```

math - because of the square root function;
PyPlot from MatPlotLib for visualisations;
the mean function from the statistics package;
and finally, the mean_square_error function from SKLearn package.

We will make use of the preprocessed file, which we saved in Exercise V.3-1. Let's load it into a variable using Pandas.

```
energy = pd.read_csv('electricity-consumption-from-2011-2-PREPROCESSED.csv')
energy['Date'] = pd.to_datetime(energy['Date'])
```

We are also remembering to convert the dates from strings to Date/Time objects.

In order to test if the Average of the energy consumption is a good predictor of future consumption, we will have to split the dataset we have in two. We will use the last known year of data as test data, and we will base our average on the previous 7 years.

```
energy_train = energy.head(84)
energy_test = energy.tail(12)
```

With these two lines of code, we create two new Pandas DataFrame objects. The first one for "training" purposes, containing 84 records (7 years of data) from the start of our original dataset, and the second DataFrame, for "testing" purposes, containing the last 12 records of our original dataset.

After doing that reassignment of the data, we will have to reset the new indexes of the testing dataset, to start from 0, and also drop the column containing the old indexs. We do that as follows:

```
energy_test = energy_test.reset_index()
energy_test.drop('index', axis='columns', inplace=True)
```

Now let's calculate the average of all consumption during the first 7 years, using the values in the training set:

```
total_average = mean(energy_train['Energy'])
print(total_average)
```

We use the "mean" function from the statistics package and we assign the result of this operation to the variable total_average. By printing the value, we see the average consumption is 107451.37 Kwh

```
107451.36904761905
```

Now we can plot the data. Our X axis will be the dates and our Y axis will be the energy values. We also plot a straight line using the total_average value as a constant.

```
plt.plot(energy_train['Date'], energy_train['Energy'],
         color='blue', linewidth=2)
plt.plot(energy_test['Date'], energy_test['Energy'],
         color='green', linewidth=2)
plt.axhline(y=total_average,
         color='red', linestyle='--', linewidth=2)
plt.rcParams['figure.figsize'] = [20, 8];
plt.xticks(rotation=45, ha='right');
```

We are also applying some formatting commands to pyplot to stylize the graphic better. We are using different colours and line styles for the plots, also changing the size of the plot and rotating the X axis labels (the dates) at 45 degrees for better readability.

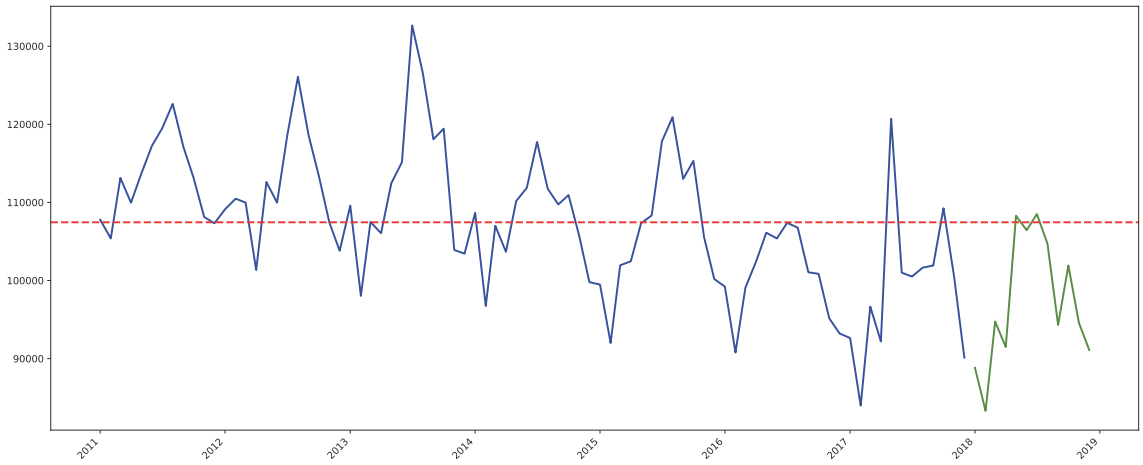


Figure 60. The red line represents the average consumption for the first 7 years (blue values).

Looking at the figure, obviously the average is not very close to the actual values for the last year (the one we wanted to predict). But having them, gives us an opportunity to compare our prediction (the average) to those values and have a numerical estimate of how far from reality we are. That is usually done by calculating the mean square error (MSE).

The way MSE is implemented in the SKLearn package requires us to have the average values either as a separate list, or as a column in our dataset. The easiest will be to create a list with as many of them as the length of our test dataset:

```
total_average_list = [total_average] * len(energy_test['Energy'])
```

Now we can calculate the MSE, and then we can go one step further, and get the square root of the MSE, which will give us the Root Mean Square Error (RMSE). RMSE is very often used as an accuracy score for predictive models, because it tells us on average how far is our prediction for the real values.

```
rmse_average = math.sqrt(mean_squared_error(energy_test['Energy'],
                                            total_average_list))
print(rmse_average)
```

After printing out the RMSE we get a value of 12889.08 Kwh. Sinse this is our first prediction model (using the average as a predictor), we have nothing to compare that value with. But in subsequent exercises we will get back to this value and see if we can get better than that.

But even now, using just a simple average, we can attempt to get a better result. Because we are dealing with energy consumption, we can safely assume there are differences in the consumption in different months. We don't even have to guess, as it is already very obvious from the visual representation of our data. We can make a logical assumption that the total average we are getting is, well, an average value for the year, so it should be closer to the actual consumption of a month, which sits somewhere between the summer and the winter (when we expect minimum and maximum of the consumption) i.e. spring, rather than summer or winter, when we would expect a larger difference. In fact, we can easily test that.

Let's see the differences between the average and the actual consumption for February and May:

```
print(energy_test['Energy'][1] - total_average)
print(energy_test['Energy'][4] - total_average)
```

Remember indexing is usually 0-based. So, February will be the 1st record, and May will be the 4th.

Thus, it probably makes sense that if we want to predict the energy consumption for a specific month, let's say March, we would probably get closer to reality if we take the average of only other March months, from the previous years.

Let's test that hypothesis and construct a new dataframe by filtering the training data:

```
energy_march = energy_train[energy_train['Date'].dt.month == 3]
```

What we do with this line of code is to create a new energy_march Pandas DataFrame, by taking all records from energy_train, where the month of the energy_train "Date" column is equal to 3 (March).

Now we can calculate the average just for the months of March:

```
average_march = mean(energy_march['Energy'])
print(average_march)
```

Let's see if the average for March from previous years is closer to the March value in our test dataframe than the total average was:

```
print(energy_test['Energy'][2] - average_march)
print(energy_test['Energy'][2] - total_average)
```

We can see that the average for just March is closer by more than 2500 Kwh to the real value than the total average (including all months).

Before we finish with this exercise, let's save our training and testing datasets as separate .csv files, so we have them ready for our next sessions, rather than having to create them from scratch:

```
energy_train.to_csv('electricity-consumption-from-2011-2017-training-set.csv',
                    index=False)
energy_test.to_csv('electricity-consumption-testing-set-2018.csv',
                    index=False)
```

The use of index=False argument skips the unnecessary writing of the row indexes in the .csv files.

Of course, our theory about the average being close to the spring and autumn months than to the summer and winter months, will break if we attempt to get the difference for June or July. That will be since there is an obvious general decline in the energy consumption (negative trend) over the entire period, that is visible from the plot. May be the library in those years invested in energy saving lights, more efficient equipment, or made other optimisations in their consumption. In all cases, they are using less and less energy every year.

When there is a trend in the data (either the process is "accelerating" and the values are increasing or it is "decelerating" and the values are getting smaller) the average is not a good way to approach forecasting the respective process. But before we get to dealing with the trend, let's see another way we can use averages.

VI.2. Weighted average

Unlike the simple average, calculating a weighted average value will take into account the different degrees of importance of individual values in the data set. When calculating a weighted average, each number in the data set is multiplied by a predetermined weight corresponding to its importance (weight) in the formation of the outcome, and only then the products obtained in this way are added to a total. The total is then divided by the sum of the weights used, in order to obtain a (weighted) average value. This can be described by the following formula:

$$\bar{y} = \frac{\sum_{i=1}^{n}(w_i y_i)}{\sum_{i=1}^{n} w_i} \qquad (VI.2\text{-}1)$$

A weighted average can be more accurate than a simple average, where all the numbers in a data set are assigned equal weight. If we consider our example of predicting the electricity consumption of a public building, it makes much more sense to give a higher weight to the latest acquired data points (last month for example), rather than the values from years ago.

Exercise VI.2-1: Forecast future energy consumption, but instead of the average use a weighted average, where some recorded values have higher significancy in the prediction.

Let's import the Pandas package, load the two datasets we saved in the previous exercise, and make sure we have converted the Date values to Date/Time objects:

```
import pandas as pd

energy_train = pd.read_csv('electricity-consumption-from-2011-2017-training-
set.csv')
energy_test = pd.read_csv('electricity-consumption-testing-set-2018.csv')
```

```
energy_train['Date'] = pd.to_datetime(energy_train['Date'])
energy_test['Date'] = pd.to_datetime(energy_test['Date'])
```

Now, since we started with March as an example from the previous exercise, and we already know what our accuracy in predicting it is, using the total average and the March monthly average. Let's continue with the same month.

We should first construct a filtered dataframe, containing only those training records that reflect the consumption in the months of March for the previous years:

```
#Filter the dataset for all rows whose date month is March (3)
energy_train_march = energy_train[energy_train["Date"].dt.month == 3]
#Restart the index of the newly created data set
energy_train_march = energy_train_march.reset_index()
#Delete the redundant 'index' column, which appeared after the reset
energy_train_march.drop('index', axis='columns', inplace=True)
```

The first line makes the same kind of filtering we have done previously, by asking for only get those rows of the energy_train dataset, where the month of the "Date" field is equal to 3 (March). The resulting new dataframe energy_train_march then needs to get its indexes reset and the old ones removed (dropped).

Let's call for the newly created DataFrame to check if everything is as expected:

```
energy_train_march
```

	Date	Energy
0	2011-03-01	113136
1	2012-03-01	109980
2	2013-03-01	107506
3	2014-03-01	107003
4	2015-03-01	101952
5	2016-03-01	99088
6	2017-03-01	96655

We see that we have 7 entries and they are all for March. Our next step will be to calculate a weighted average for these values. But before that we need to decide what weights we are going to use.

Our logic here would be that the closer the month is to the one we want to forecast, the more likely its consumption value will be closer to what we will need in the predicted month. In other words, last March we probably used closer to the amount of energy we did this March rather than March 7 years ago. especially considering that we spend less and less each year (visible from the data visualization in the previous exercise).

For a test, we can use these completely intuitively generated weights:
The most recent month of March (the last observed, with index 6 in our array) will have a weight of 5 (i.e. 500% or 5 times more important than another March months), and the first observed March (the oldest, with index 0) we will take with a weight of 0.1 (ie 10% or 0.1 times the importance).

Here is an example weight distribution for the filtered 7 months:

Index	Weight	Description
[0]	10%	The first March we know of (furthest from our time) we take with only 0.1 importance (weight).
[1]	20%	The next one with 0.2 importance (weight).
[2]	50%	The third March's impact on our forecast will be 0.5 times its value.
[3]	100%	Then 1.0 time.
[4]	200%	As we are getting closer to the present time, we increase the weights. This month's value we take 2.0 times.
[5]	300%	The semi-last March we take 3.0 times.
[6]	500%	The latest March (last year's record) we take with the highest importance of 5 times.

Let's apply that to our data:

```
total_march_energy_weighted = (
energy_train_march['Energy'][6] * 5.0 +
energy_train_march['Energy'][5] * 3.0 +
energy_train_march['Energy'][4] * 2.0 +
energy_train_march['Energy'][3] * 1.0 +
energy_train_march['Energy'][2] * 0.5 +
energy_train_march['Energy'][1] * 0.2 +
energy_train_march['Energy'][0] * 0.1
)
```

In these lines of code, we take the values for the energy consumption (from the respective column and row) and we multiply them by our chosen weights, and then summing them in a new variable called total_march_energy_weighted.

Following Equation (VI.2-1) we now have to divide the sum of the weighted values to the sum of the weights themselves:

```
weighted_march_average = total_march_energy_weighted / 11.8
print(weighted_march_average)
```

Since the total sum of the weights we used is 11.8, that is what we need to use as a divisor as well.

```
99873.61016949153
```

Finally, to assess how good our prediction is, we can find the error between our prediction (the weighted average of all March months) and the actual March value from the test array. To be consistent in the metric used over the past (and future exercises) we will use the root mean square error (RMSE). The out-of-the-box RMSE function we used in the previous exercise requires an array of data, and cannot work with single values, so we'll need to apply the RMSE formula ourselves:

```
math.sqrt(((weighted_march_average - energy_test['Energy'][2])**2)/2)
3626.4750289097474
```

Obviously, we got much closer to the real value, and if we spend some time tweaking the weights, we can get even closer. Of course, that is a very limited model, which can only be used for that specific month and cannot be applied to the rest of the testing data.

VI.3. Moving average

An alternative to simple or weighted averaging is to calculate an average over a smaller data period comprising only a few intervals. The averaging process is performed by calculating an average for each subsequent period.

The main reason for doing this is that way we can negate the effects of noise or other kinds of fluctuations in the data, while still being led by the idea that we are more likely to spend as much energy as the average of the last few days, rather than an average over the whole period of data, or some distant data point.

For calculating such a Simple Moving Average, we can think of the period we are averaging over as a sliding window, which we move one step forward with every new observation (value). Essentially, it is an equally weighted mean of the previous *n* data points.

$$\bar{y}_{SM} = \frac{y_n + y_{n+1} + \cdots + y_{M-(n-1)}}{M} = \frac{1}{M}\sum_{i=0}^{n-1} y_{M-i} \qquad (VI.3\text{-}1)$$

So, for calculating the next average value the window needs to slide one value (hence "moving average"). That way, a new value will be added into the sum, and the leading value from the previous time period will be dropped out. Since the new period average will include most of the same values, in order to optimize the operations, we don't need to sum all values again. What we can do for every new period is:

$$\bar{y}_{SM} = \bar{y}_{SM\_previous} + \frac{1}{n}(y_M - y_{M-n}) \qquad (VI.3\text{-}2)$$

That way, regardless of how long the period (averaging window) is, we only do a full summation the first time, and then have to only do two operations. Let's see how we can implement that in Python.

Exercise VI.3-1: Again, forecast the future energy consumption, but this time use a moving average.

As per usual, we need to import the packages we will be working with.

```
import pandas as pd
import matplotlib.pyplot as plt
import math
from sklearn.metrics import mean_squared_error
```

As usual, we will use Pandas to import and manage the data structure, and we will need the plotting library to visualise our results. We will again use the root of the mean squared error to determine how good our prediction is, so we will need the math and the mean_square_error libraries.

Let's start with reading the data sets and storing it in a Pandas Dataframe variable.

```
energy = pd.read_table('electricity-consumption-from-2011-2-
PREPROCESSED.csv')
```

Next, we again need to make sure Pandas understand that the data in one of our columns is of the datetime type.

```
energy['Date'] = pd.to_datetime(energy['Date'])
```

Now we are ready to start working on the moving average. The first thing we will do here is to select the size of the period we want to average over (the so called "window"). Let's start with a small value, for example 3. That means we look at the last 3 values we have, we average them, and we assume that the next value that comes will be that average, forecasting it on the basis of that previous period.

In order to do that we will also need to create a new empty list, to store the averages we calculate on every step. We can later use that list of forecasted values (all the moving averages) to plot it and see how it looks in comparison to our actual (real-world) data.

```
window_size = 3
moving_averages = []
```

We define the size of the averaging window and create an empty list to store our moving average values in.

Our next step would be to create a loop which will iterate through all the elements of the dataset, calculate each average, and then append it to the empty list we created to store those average values.

```
i = 0
# Loop through all the elements of the "Energy" column
# and take all possible periods of length equal to the window_size
while i < len(energy['Energy']) - window_size + 1:
    # Store elements from i to i+window_size
    # in list to get the current window
    current_window = energy['Energy'][i:i + window_size]

    # Calculate the average of current window
    current_window_average = round(sum(current_window)/window_size, 2)

    # Store the average of current window in moving average list
    moving_averages.append(current_window_average)

    # Shift window to right by one position
    i += 1
```

First line of code creates an iterator variable and sets it to zero. We then use a while loop, which will run until the iterator is less than the length of the dataset minus the size of the averaging window (in our case 3) plus one. Inside the loop we first make a subset of the values, which are in the current window and store them in the very unimaginatively called variable 'current_window'. For example, in the first iteration it will take the values from the 0 to 3rd row, on the next iteration it will move the window one value and take the interval from 1 to 4, then 2 to 5, etc. The next step is to sum the values in the current_window, divide by the length of the window and then round to the second digit. Finally, we append the average to the list of moving averages and we increment the iterator.

Let's plot the list of (moving) averages against the actual data and see how it looks.

```
plt.plot(energy['Date'], energy['Energy'],
         color='blue', linewidth=1)
plt.plot(energy['Date'][2:96], moving_averages,
         color='r', linestyle='--', linewidth=1)
plt.rcParams['figure.figsize'] = [16, 7]
plt.xticks(rotation=45, ha='right');
```

We first plot the real consumption data. We give this line 'blue' colour and thickness of 1. We then plot the moving averages, starting from the 3rd datapoint, but we choose the colour to be 'r' (shorthand for red) and we change the line style to be dashed, keeping the thickness of 1. Finally, we give some parameters to the plot figure, just to make it a bit more visually appealing – specific size and rotation of the x-axis labels.

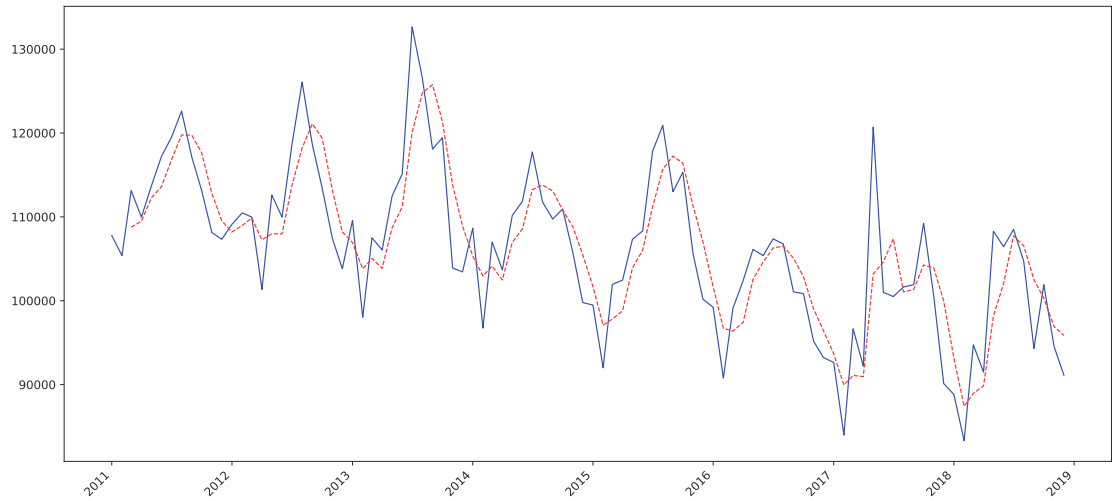


Figure 61. The red (dashed) line represents the moving average with a window of 3 values. The blue line is the real consumption data.

We can see that the moving average follows the features of the real data while also smoothing them. This is, of course, perfectly normal and expected from an averaging operation. In fact, averaging is often used for processing noisy data, so that by applying some smoothing the baseline becomes clear.

Let us see how good our predictions were, using the moving averages, compared to the actual data.

```
rmse_average = math.sqrt(mean_squared_error(energy['Energy'][2:96],
                         moving_averages))
print(rmse_average)
```

```
4821.501506878859
```

That is way better than what we previously achieved, when using the simple average and weighted average. Of course, we have to acknowledge something important here – we are getting relatively accurate values, but only one value ahead. So, if we need to forecast 12 months ahead, they will all be the same one value we get from the past few datapoints.

If we take a closer look at the graph in Figure 61 we can observe there is a slight shift between the ups and downs of the two sets. That kind of lag is expected. Can you think of the reason? In fact, can you find the mistake we made when plotting the forecasted values? Let's see what will happen if we increase the "window" size.

```
window_size = 6
moving_averages = []
```

Redefine the window size and re-initialise the empty list for the average values.

```
i = 0
# Loop through the array to consider
# every window of size 6
while i < len(energy['Energy']) - window_size + 1:
    # Store elements from i to i+window_size
    # in list to get the current window
    current_window = energy['Energy'][i:i + window_size]

    # Calculate the average of current window
```

```
    current_window_average = round(sum(current_window)/window_size, 2)

    # Store the average of current window in moving average list
    moving_averages.append(current_window_average)

    # Shift window to right by one position
    i += 1
```

Rerun the loop to calculate the new averages and store them in the empty list.

```
plt.plot(energy['Date'], energy['Energy'],
         color='blue', linewidth=1)
plt.plot(energy['Date'][5:96], moving_averages,
         color='r', linestyle='--', linewidth=1)
plt.rcParams['figure.figsize'] = [16, 7]
plt.xticks(rotation=45, ha='right')
```

Plot the new curve on top of the actual data.

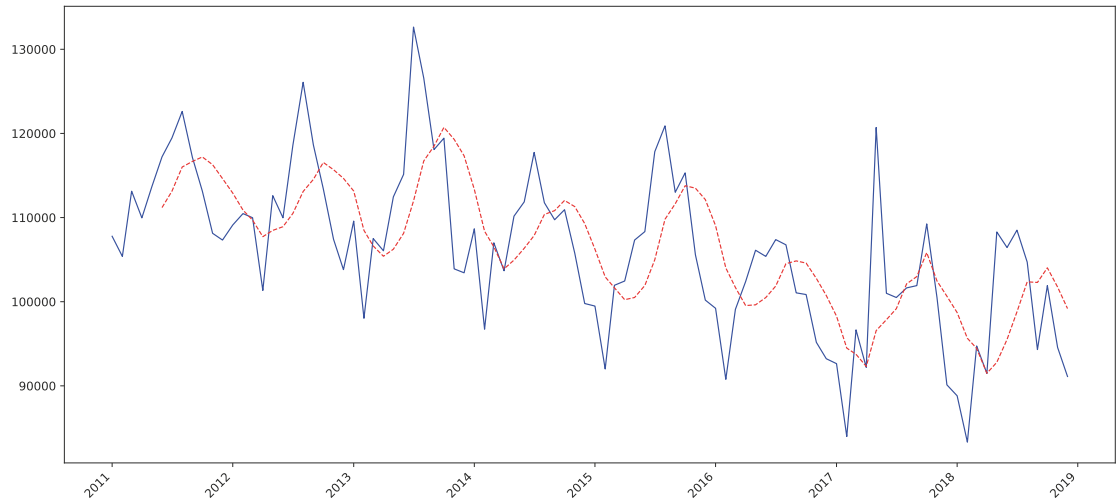


Figure 62. Same as the previous figure, but the red dashed line has been achieved using a 6-period averaging window.

We can see the curve representing the moving average values is getting smother, less peaks and dips, other than the obvious seasonal ones. Of course, the lag is also increasing, since we are getting the first value after only after we have already accumulated 6 previous values.

Let's increase the window size one more time:

```
window_size = 12
moving_averages = []
```

```
i = 0
# Loop through the array to consider
# every window of size 12
while i < len(energy['Energy']) - window_size + 1:
    # Store elements from i to i+window_size
    # in list to get the current window
    current_window = energy['Energy'][i:i + window_size]

    # Calculate the average of current window
    current_window_average = round(sum(current_window)/window_size, 2)
```

```
    # Store the average of current window in moving average list
    moving_averages.append(current_window_average)

    # Shift window to right by one position
    i += 1

plt.plot(energy['Date'], energy['Energy'],
         color='blue', linewidth=1)
plt.plot(energy['Date'][11:96], moving_averages,
         color='r', linestyle='--', linewidth=1)
plt.rcParams['figure.figsize'] = [16, 7]
plt.xticks(rotation=45, ha='right')
```

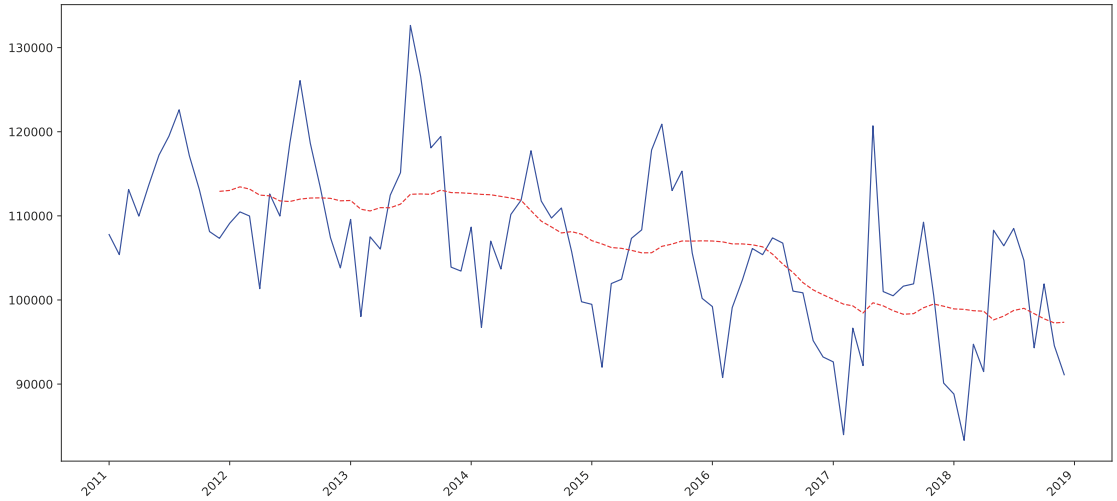


Figure 63. Same as the previous figure, but the red dashed line has been achieved using a 12-period averaging window.

With every increase of the averaging window, the forecasted data gets smoother and smoother, starting to approach a line, trending down as the library was consuming less and less energy each year. Make a mental note of that, as we will discuss it in the next exercises.

Moving average is particularly simple method to implement, but it comes with limitations. First of all, we can only forecast the next consecutive value. True, we can recursively feed it back into the algorithm, in order to get more values, but you can't just forecast a value at an arbitrary time in the future. We will address that when we get to more complex models. For now, let's see how we can improve on the moving average method.

VI.4. Exponential smoothing and forecasting (exponentially weighted moving average)

As we already observed, in a given series of interval measured data, there is always more or less some random variation in the values. These random fluctuations can be artificially generated noise due to the method of measurement or the method of operation of the particular measuring instrument, or they can be natural micro-fluctuations in the process itself, due to certain physical phenomena. A commonly used technique to reduce the effect of such fluctuations is the so-called smoothing. We already saw this effect in the previous exercise.

In essence, the simple exponential smoothing is not so different than the weighted moving average we already used. The difference is mainly in the weights used. The simple exponential smoothing is in reality an exponentially weighted moving average. Unlike the simple moving average, which considers past observations equally, the exponential smoothing assigns exponentially decreasing weights over time. This means that exponential smoothing places a stronger emphasis on more recent observations.

In reality, we already achieved the same effect by manually assigning weights, in our weighted moving average example. The difference here is the weights distribution is exponential, and is controlled by one single parameter (indicated as an alpha coefficient) which can have a value between 0 and 1. We can calculate (forecast) the next value in a series using the following formula:

$$\hat{y}_{t+1} = \alpha y_t + (1-\alpha)\hat{y}_{t-1} \qquad (VI.4\text{-}1)$$

where the $\hat{y}_{t+1}$is the forecasted value for the next period (we assume *t* is now) and is equal to the smoothing coefficient alpha times the last known (actually observed) value y_t, and finally the term $\hat{y}_{t-1}$is the previous forecast.

Let us see how well this method will model our data, and if it can achieve better forecasting results than we already did with previous methods.

Exercise VI.4-1: Use an exponential smoothing method to predict the next value in the timeseries.

As before, we start by importing the necessary packages.

```
import pandas as pd
import matplotlib.pyplot as plt
import math
from sklearn.metrics import mean_squared_error
```

This time we will load the whole data set, and as before, we will convert the Date column to datetime format.

```
energy = pd.read_csv('electricity-consumption-from-2011-2-
PREPROCESSED.csv', sep=';')
energy['Date'] = pd.to_datetime(energy['Date'])
```

Our next step will be to generate an empty list to store the forecasted (smoothed) values. We will also create a variable called ‘predicted’ where we will store the result of every prediction. And then, we need variable called ‘alpha’ to store the smoothing coefficient (just as in formula (VI.4-1))

```
exponential_moving_averages = []
predicted = energy['Energy'][0]
alpha = 0.2
```

We are assigning a 0.2 value to the alpha coefficient, which is just a randomly selected initial value. In reality this parameter has to be optimised for, so that a good working one can be found for the specific process we are forecasting. Since this is a demonstration 0.2 will be just fine, and in later exercises we will go through an optimisation process, finding the best possible smoothing coefficients. We also need an initial “predicted” value to use when we enter the loop for the first time. We can just make it equal to the actual value.

Now we are ready to run the smoothing on the entire dataset. In order to do that we will need a “for” loop. It will iterate through all the values in the energy Dataframe and calculate a prediction (using the formula) for every value. Then the current prediction is appended to the empty list of smoothed (forecasted) values, and the loop moves to the next value.

```
for actual_value in energy['Energy']:
    predicted = alpha * actual_value + (1 - alpha) * predicted
    exponential_moving_averages.append(predicted)
```

When the loop is finished, the ‘exponential_moving_averages’ list should contain all of the predictions (made from the perspective of every time point in the data set). We can plot the forecasted (smoothed) values on top of the actual values, to see the difference.

```
plt.plot(energy['Date'], energy['Energy'],color='blue', linewidth=1)
plt.plot(energy['Date'], exponential_moving_averages, color='r',
linestyle='--', linewidth=1)
plt.rcParams['figure.figsize'] = [16, 7]
plt.xticks(rotation=45, ha='right');
```

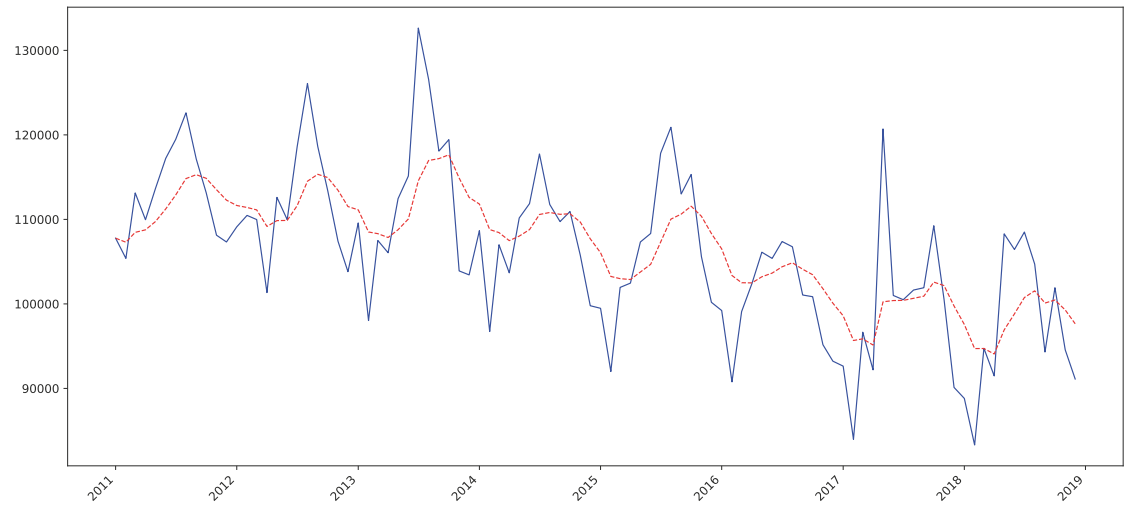


Figure 64. It is worth noting that the shape of the red dashed curve is similar with the one on Figure 62. It is the value of the alpha coefficient that dictates the significance of the previous values in the series in the forecasted value (in an exponentially decreasing order) which has a similar effect like the choosing of a longer or shorter averaging window in the simple moving average method.

As we previously discussed, the simple moving average methods (including the single exponential method) are good for forecasting only one (next) future value from the "now" standpoint. In other words, we can only predict one period ahead.

The method, of course, is recursive, so we can easily generate more values ahead (by using every prediction as the basis for every next prediction), but in reality, that will just generate the same value.

Of course, from the conveniens of hindsight, we can estimate how good the predictions were, if we take all separate values and compare them to the actual (observed) values. Same as we did in previous exercises. We can do that by again calling the ‘mean_square_error’ function for the array of actual values (in the energy Dataframe) and the predicted values in the ‘exponential_moving_average’ list and then taking the square root of the result.

```
rmse_average = math.sqrt(mean_squared_error(energy['Energy'],
exponential_moving_averages))
print(rmse_average)
```

```
6385.2245020024075
```

I advise you to try running all the code again, but this time changing the 'alpha' value. Experiment with values even closer to 0 or 1 to see how the smoothed curve will change, and we can discuss why is that happening.

It is now clear that the simple exponential smoothing did not beat the forecasting accuracy of the simple moving average and the weighted moving average. And although by experimenting with different alpha values we can get better result, there is a good reason why this is not the best method for this particular dataset.

Simple exponential smoothing only has one parameter (alpha) that controls the rate of influence from historical observations. We demonstrated that values closer to 1 mean that the model pays little attention to past observations, while smaller values force more of the hist historical values to be considered during predictions. Because there is only one parameter and it affects the weighting of the level (or the magnitude) of the value, the method is not well suited for dealing with time-series data that has trend or/and seasonality, since they would require separate parameters to account for them. From all the graphs we plotted so far, it is fairly obvious our data has both trend and seasonality.

So, let us see how we can enhance the simple exponential smoothing model to also care about trend.

VI.5. Double exponential smoothing and forecasting

Double exponential smoothing is also known as Holt's exponential smoothing [7]. It includes two exponentially weighted moving averages: one for the smoothed values of *the level*, and another one for the *trend*. Double exponential smoothing is suitable for time-series data that has trend but with no seasonality component.

This builds on the single exponential smoothing technique with an additional smoothing factor (beta) to control the change in the trend (or the slope) of the observed process. For our data we will assume that the trend is additive in nature (as opposed to multiplicative), so, to get the next (forecasted) value, we need to add the smoothed level and trend together:

$$\hat{y}_{t+1} = l_t + b_t \qquad (VI.5\text{-}1)$$

$$l_t = \alpha y_t + (1-\alpha)(l_{t-1} + b_{t-1}) \qquad (VI.5\text{-}2)$$

$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1} \qquad (VI.5\text{-}3)$$

where l_t is the smoothed level and b_t is the trend component in the forecast, $y_t$is the last actual (observed) data point, $l_{t-1}$is the level of the previous period, and $b_{t-1}$is the trend in the previous period. Although we assumed additive trend, the method also allows modelling processes with exponential trend ("multiplicative"), but then the level and trend components in the formula need to be multiplied.

In the previous exercise, when dealing with single exponential smoothing, we coded our own smoothing loop, using the provided formula. I am encouraging you to try to do the same with the double exponential method, as a homework, but this time we are going to use a library of statistical models, which has the exponential smoothing methods built in.

Exercise VI.5-1: Use a double exponential smoothing method to predict the next (future) value in the timeseries, while accounting for the trend in the data.

Let's start by importing the necessary packages. The new package here is the TSA (Time Series Analysis) package from the 'statsmodels' library. We will be specifically importing the Holt-Winters methods, and for convenience we will assign them an alias 'tsa' (same way we assign 'pd' to Pandas, or 'plt' to pyplot).

```
import math
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.tsa.holtwinters as tsa
from sklearn.metrics import mean_squared_error
```

Our next step is to load the data. For this exercise we will need training and testing data. We can either load the already split datasets (we have them saved in .csv formatted files in previous exercises), or we can load the whole dataset and do the splitting here. I'll go with loading the whole file because I want to show you other tips and trick regarding loading data with Pandas.

```
energy = pd.read_csv('electricity-consumption-from-2011-2-
PREPROCESSED.csv', index_col='Date')
```

You should already be quite familiar with the method for loading data from a .csv file to a Pandas Dataframe. The new thing here is that we will assign an index column, rather than let pandas generate one on its own. Remember our dataset has two columns, a "Date" column and an "Energy" column. Let's use the dates column as an index.

Then, we have to split the data in two arrays, one for training the model, and one for testing purposes. We can use the index (Date) column to split the data. As before, we can use the last year (2018) for testing, and everything up to December 2017 for training.

```
energy_train = energy[ :'December 2017']
energy_test = energy['January 2018': ]
```

What we are essentially saying with these two lines of code is, make an 'energy_train' variable and store in it all values from the 'energy' Dataframe from the start of the array to the 'December 2017' record. The second line does the same, but we start at 'January 2018' till the end [start : end].

Next, we might need to work with the dates (in the index), so we better convert them from string type data to datetime type data. Also, when talking about Timeseries data, as we discussed in the beginning of this chapter, we are always having a constant time period length. Usually, the packages we are using should be able to guess what the frequency of the data ingestion was (in our case is monthly), but just to make sure everything works correctly, we should make sure the 'freq' (frequency) attribute of the Dataframe index is correct. We can force Pandas to infer the correct frequency by using the necessary function:

```
energy_train.index = pd.to_datetime(energy_train.index)
energy_test.index = pd.to_datetime(energy_test.index)
```

```
energy_train.index.freq = pd.infer_freq(energy_train.index)
energy_test.index.freq = pd.infer_freq(energy_test.index)
```

The first two lines convert the indexes of the training and testing datasets to a datetime format, and the second two lines ask Pandas (pd) to infer the frequency of the values and store that frequency in the 'freq' attribute of the respective Dataframe index.

We are now ready to define our model and train it. We are going to create a new object 'des' (Double Exponential Smoothing) and initiate an instance of the ExponentialSmoothing class in the 'tsa' package. We will give it the necessary parameters to make it a double exponential model. That means we need to assign the trend type ('additive' or 'add', for linear, or 'multiplicative' or 'mul' for exponential trend). In our case, at least visually from the graphs, it seems like the trend in our data is linear.

Next, we say that we are not going to use dampening of the trend. Dampening has to do with the horizon of the forecasting (how far ahead we want to predict the values). The presumption is the trend will not continue to increase or decrease indefinitely. Thus, the damped trend method adds a dampening parameter, so that the trend converges to a constant value in the future (it flattens the trend). We are not going to use that parameter, so we will say the dampening equals 'False'.

We are also going to ignore the seasonality in the data. As we discussed the double exponential method deals with level and trend. In order to simulate that we will turn off the seasonality smoothing parameter by saying seasonality is 'None'. In the next exercise, when using triple exponential smoothing, we will use the same function, but we will then model it with all three parameters. Speaking of parameters, let's define them now, we will again use alpha for the level, but we will also add beta now, for the trend:

```
alpha = 0.5
beta = 0.5
```

We are again going to start with some completely randomly chosen values for alpha and beta, but we can later experiment with changing them.

For now, let's see how we'll do with just the two:

```
des = tsa.ExponentialSmoothing(energy_train, trend='additive',
                               damped_trend=False, seasonal=None)
des_model = des.fit(smooting_level= alpha, smooting_trend=beta)
```

In the first line we create the model by passing in the training dataset and assigning 'additive' trend. The second line we ask for the model to be trained (fitted) to our existing (training) data, using the two smoothing parameters.

Now, the 'des_model' variable contains the trained model. We can use it to forecast the next value(s) in the series.

```
energy_forecasted = des_model.forecast(steps=len(energy_test))
```

Let's convert the list of forecasted values in a Pandas Dataframe and make sure the dates in the index column are interpreted correctly as datetime format.

```
energy_forecasted = pd.DataFrame(energy_forecasted,
columns=['Energy'])
energy_forecasted.index = pd.to_datetime(energy_forecasted.index)
```

Finally, let's get everything plotted out, so we can better see the results of our actions

```
plt.plot(energy_train, color='blue', linewidth=1)
plt.plot(energy_test, color='green', linewidth=1)
plt.plot(energy_forecasted, color='red', linestyle='--', linewidth=1)
plt.rcParams['figure.figsize'] = [16, 7]
plt.xticks(rotation=45, ha='right');
```

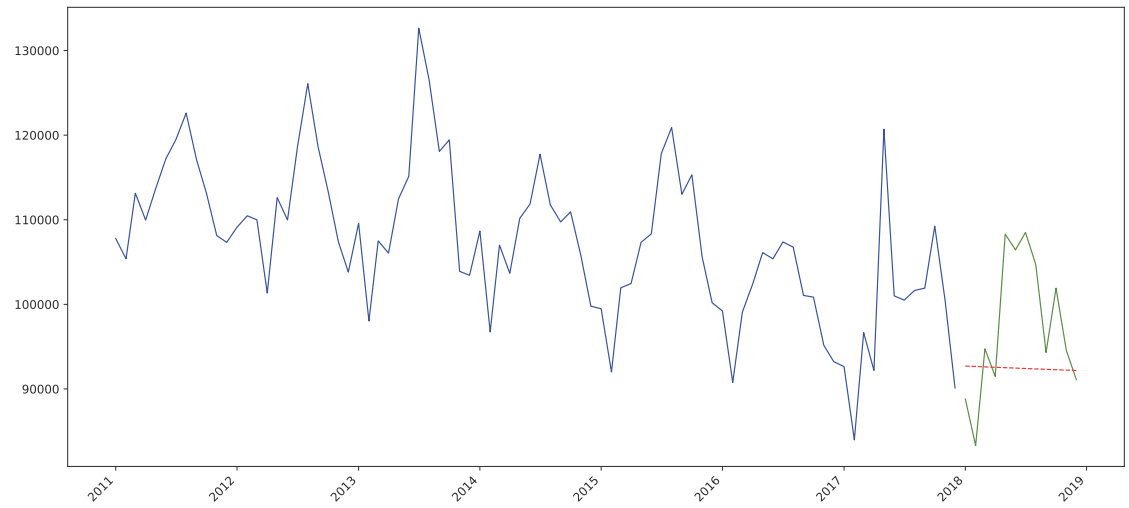


Figure 65. We can clearly see the forecasted values in a slight linear decline (based on the trend in the data). The gradient of that trend will depend on the smoothing coefficient we chose for it (beta). See what happens when you change it...

And finally, to get some better understanding of how are we doing in terms of predictive power, we can calculate the square root of the mean squared error (between the forecasts and our test dataset), as we did with the previous models:

```
rmse_average = math.sqrt(mean_squared_error(energy_test['Energy'],
energy_forecasted['Energy']))
print(rmse_average)
```

```
9430.408497121443
```

We are getting a worse performance than some of the more trivial methods, but remember than all of them only allowed for one forecast at a time, whereas now we are getting all 12 values ahead!

Of course, the relatively high error we get against the test data set is not only due to the fact that we are now comparing with the entire test set from the position of a single month in the past, but we are also completely ignoring the fact there is one more significant factor in our data, that we need to address – seasonality.

VI.6. Triple exponential smoothing and forecasting

In order to get the seasonal component, we have to introduce one more smoothing parameter and thus one more element to the overall smoothing and forecasting equation [8].

$$\hat{y}_{t+1} = l_t + mb_t + s_{t-p+1+(m-1)mod\ p} \qquad (VI.6\text{-}1)$$

$$l_t = \alpha(y_t - s_{t-p}) + (1-\alpha)(l_{t-1} + b_{t-1}) \qquad (VI.6\text{-}2)$$

$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1} \quad \text{(VI.6-3)}$$

$$s_t = \gamma(y_t - l_t) + (1-\gamma)s_{t-p} \quad \text{(VI.6-4)}$$

where m is the horizon of the forecast (how many steps ahead we want a forecast for), p is the length of one season (in time periods), $s_{t-p+1+(m-1)mod\ p}$ is just a way to describe that in order to get the seasonal component, you have to take into account the last known point at the forecasted position from the previous season.

Please note, that the method requires you to know the season length in advance. For most timeseries (where forecasting is necessary) that is true. Even if there is no distinctly obvious seasonality, one can always assume one (or more) seasonal effects are in place, especially when real-world phenomena are being sampled. There are ways to modify the Holt-Winters method to account for sub-seasons within the overall (longer) season, as well as other exponential methods, developed specifically to deal with more complex seasonality, but they are outside the scope of this exercise [9].

Much like we had to distinguish the type of trend we are dealing with, when applying the double exponential method, here we also have to assign a type for the seasonal component, which can be either additive or multiplicative. Think of the Additive seasonality as a constant that is added to the equation and does not depend on where in time we are. We can say the effect of the seasonality on the data is constant in respect to the trend. The Multiplicative version of seasonality means that there is change over time for how large of an effect that seasonality has on our values. Think of it as having a proportional effect, which is tied to the trend of the process (in time).

One very simple way to determine which one to use (if not immediately obvious from the graphical representation of our data), is to compare the residuals after the historical data we have is decomposed to its components. It is important to notice that equation (VI.6-1) adds the three components together, assuming an additive nature of trend and seasonality. But if it turns out that seasonality is multiplicative, the formula will have to account for that by multiplying the seasonal component. Let's see which one it will be:

Exercise VI.6-1: Use a triple exponential smoothing method to predict the next (future) value in the timeseries, while accounting for both the trend and the seasonality in the data.

As always, we start by importing the additional libraries we will need. And this time we will need quite a few extra libraries.

```
import math
import itertools
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import statsmodels.tsa.holtwinters as hw
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
```

The math library we still need because of the square root function when calculating the RMSE. We will need the itertools methods when constructing a list of permutations of the smoothing coefficients, aiming to find which combination (permutation) of parameters will be optimal for our data set. We need numpy to construct an array of equaly spaced values (again, for the needs of smoothing coefficients optimisation) later on. We are going to change the alias we use for the Holt-Winters set of methods we import from tsa from tsa (that we previously used) to hw,

for more clarity. We will also import the seasonal_decompose function from the same library. The pandas and the final two should already be familiar from previous exercises.

Our first step is to import the dataset we will be using into a Pandas Dataframe object. Like we did in the previous exercise, we will use the Date column as an index for the timeseries.

```
energy = pd.read_csv('electricity-consumption-from-2011-2-
PREPROCESSED.csv', index_col='Date')
```

We will, again, split the data in two – a training and a testing set. The training will contain the records up to December 2017 and the test set will contain the last year of records, from January 2018 till the end of the data set.

```
energy_train = energy[:'December 2017']
energy_test = energy['January 2018':]
```

Like we did in the previous exercise, we will convert the index values from the string type (which they are at the time of import) to a datetime type objects, so we can address them easier if we need to. We will also make sure that the frequency of the timeseries is well defined, so we will ask Pandas to infer it from the available data and then store it in the necessary parameter of the index column.

```
energy_train.index = pd.to_datetime(energy_train.index)
energy_test.index = pd.to_datetime(energy_test.index)
energy_train.index.freq = pd.infer_freq(energy_train.index)
energy_test.index.freq = pd.infer_freq(energy_test.index)
```

We are now ready to start the real work with using the Triple Exponential Smoothing and forecasting method (also known as Holt-Winter's method) to understand better the development of the observed process and possibly get good at forecasting it.

Previously we went directly to creating a model and training (fitting) it with the data. This time let's spend some time to do a timeseries decomposition. That means extracting the trend and the seasonal components within the data and getting the residuals. This will allow to better define what our model should look like.

```
decompose_result = seasonal_decompose(energy['Energy'], period=12,
model='additive')
```

The tsa library contains a function to do that, and we already imported it in the beginning of the notebook. We only need to supply the data set, state the seasonality period (in our case 12 months) and also define the type of the seasonality – additive or multiplicative. We will start with 'additive'.

After running this code, which depending on your computer's hardware might take a few seconds, we will get a decompose result. This object includes a plot() method, which we are going to use. So, we will set a figure variable 'fig' and use it to store the plot in it. Then, we can set additional parameters to the figure, like the size and the rotation of the x-axis labels.

```
fig = decompose_result.plot()
fig.set_size_inches((16, 12))
fig.tight_layout()  # Tight layout to realign things
fig.autofmt_xdate(rotation=45, ha='right')
```

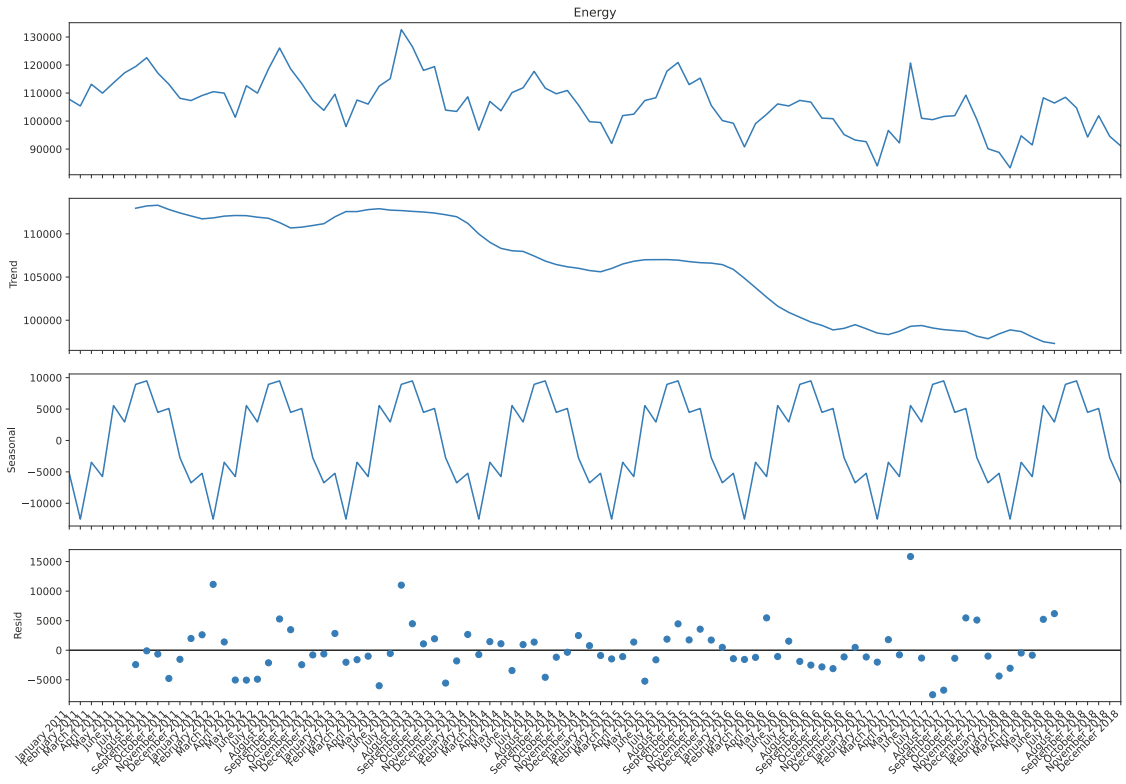


Figure 66. The first graph is the actual data, the second one is the trend component, the third is the seasonal component and what is left after removing trend and seasonality (so called residual) is plotted in the fourth graph.

Other than the fact that the seasonal component in the data set is very well defined and almost identical over the eight years of records, the interesting part are the residuals. The residuals are what is left of the data when the trend and the seasonality are accounted for. Residuals are calculated by the difference between the smoothed (predicted or fitted) value and the actual observed value at every position. Generally speaking, a good forecasting method will yield residuals that satisfy the following conditions:

- They are uncorrelated: If there is correlation between residuals, then there is additional information in the data, which was not well represented by the currently selected trend and seasonality components.
- The residuals have zero mean: If the residuals do not have a mean equalling zero, then the smoothing (fitted) curve passed tendentiously through the data points and left most of them on one side. It basically means the forecasts we get from that model will be biased.

We can see that the residuals in Figure 66 are distributed somewhat randomly around the 0 of the value axis. Of course, our first thought should be to try to run the decomposition again but using different type of seasonality or/and trend. We can do that by running the same line of code (seasonal decompose) but changing the seasonality type to be ‘multiplicative’. If we then plot the results again, we will have a similar figure, but the residuals will tell a different story:

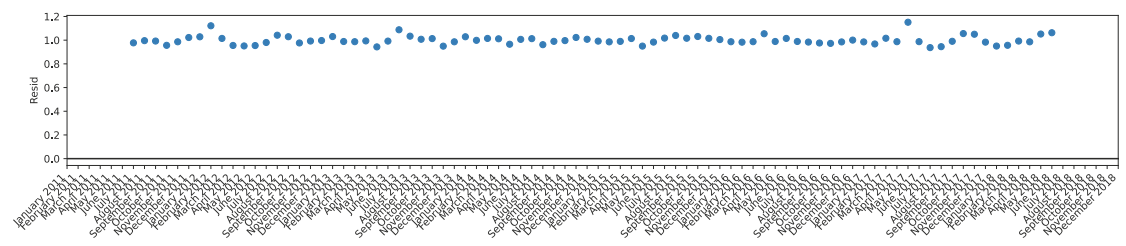


Figure 67. With these new results we can clearly see that there is less variability in the residuals, and they are much more tightly packed around the value of 1.

We can, of course, check with greater precision the mean of the two residual sets (for the 'additive' and the 'multiplicative' models), by running the following code for both of them:

```
print(np.mean(decompose_result.resid))
```

It will use the mean() function in Numpy and apply it to the set of residuals, which is stored in the .resid property of the decompose_result object. If you do that, you will see that the 'additive' method will give a residual mean of about 40 kWh, whereas the 'multiplicative' method will return a residual of about 1 kWh, which was already obvious by looking at Figure 67.

Judging purely from those values, we can conclude that the 'multiplicative' version of the method is closer to having a zeroed residual mean, than the 'additive'. But the reason they are not zero comes from the fact both methods failed to converge to a minimum (of the MSE) when fitting the training data. We could go further and also check for corelation, but this is not in the scope of this exercise, and besides, in order to do that we will have to go through the regression analysis topic, which follows after this one. For now, we will just choose the 'multiplicative' method, as we suspect it will yield better results (based on the residuals mean).

Now that we know, what kind of model we need to train, in order to forecast future values, we need to create it, using the already familiar functionality, and make sure we specify the type of seasonality to be 'multiplicative' (can be shortened to just 'mul').

```
tes = hw.ExponentialSmoothing(energy_train, trend='mul',
damped_trend=False, seasonal='mul', seasonal_periods=12)
```

We should not forget to set our smoothing coefficients. We previously used alpha and beta for the level and trend respectively, but this time we will need one more – gamma, for the seasonal component.

```
alpha = 0.5
beta = 0.5
gamma = 0.5
```

We are again going to start with some completely randomly chosen values for alpha, beta and gamma.

Next, we train the model, by trying to fit it to the training (historical) data. Pay attention for warnings in the output field – did the optimisation algorithm (used during training) manage to converge to an optimum?

```
tes_model = tes.fit(smooting_level= alpha,
                    smooting_trend=beta,
                    smooting_seasonal=gamma)
```

The TES (Triple Exponential Smoothing) model is trained and stored in the tes_model object. Now we can use it to forecast new values. Let's do that:

```
energy_forecasted = tes_model.forecast(steps=len(energy_test))
```

With this line of code we are creating a new variable called energy_forecasted and storing in it the result of the .forecast method of out trained method (pun not intended :). The .forecast() method of the trained model object

requires a forecasting horizon (how many steps ahead to forecast). We define those to be equal to the length of the testing data set (which is currently 12).

We can then convert the results of the forecasting (which are returned as a list) to a Pandas Dataframe. Also, for consistency purposes, we should convert the index of the results in a datetime format.

```
energy_forecasted = pd.DataFrame(energy_forecasted,columns=['Energy'])
energy_forecasted.index = pd.to_datetime(energy_forecasted.index)
```

Let us see how these forecasted values look like, visually compared to the test data set:

```
plt.plot(energy_train, color='blue', linewidth=1)
plt.plot(energy_test, color='green', linewidth=1)
plt.plot(energy_forecasted, color='red', linestyle='--', linewidth=1)
plt.rcParams['figure.figsize'] = [16, 7]
plt.xticks(rotation=45, ha='right');
```

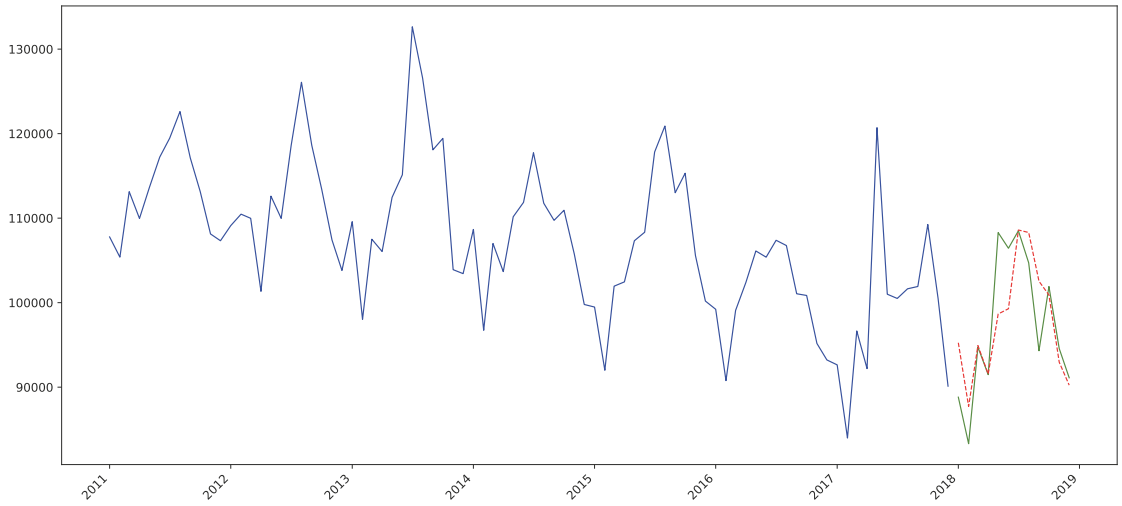


Figure 68. The green line represents the test values, and the red dashed line are the forecasted values.

We can see the forecasted values are rather close to the actual values. But how close? Let's run the MSE and get its root to find out:

```
rmse_average = math.sqrt(mean_squared_error(energy_test['Energy'],
energy_forecasted['Energy']))
print(rmse_average)
```

```
4943.8800184029005
```

That is so much better than everything else we have managed to get so far. And that is, regardless of the fact that we could, for completionism reasons, train an 'additive' model and then run the forecasting with it, and actually get a little bit lower error that way. That is true, you can try it yourself if you want. And we could make a conclusion, that we made a mistake choosing 'multiplicative' over 'additive'. But that would only be true if we stopped here. In reality, up until now, we were just trying to get an idea where to focus our efforts.

Our next step will actually be to optimise the smoothing parameters and get an even better forecasting model. In order to do that we want to test which combination of smoothing

parameters (alpha – for the level, beta – for the trend, and gamma – for the seasonality) will provide a model which forecasts bests (with least error).

To achieve that we first need to create a list of possible values for all of our smoothing coefficients. It is up to us to decide how many and how precise values we will use. To keep things relatively simple and to limit the time we spend optimising, I would suggest we take 9 value options per coefficient. We start with a minimum value of 0,1 and increment with a step of 0,1 up to a maximum value of 0,9.

We could, of course, set our coefficient variables to be equal to a list that looks like this: [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]. But instead of manually entering values for the tree variables, we can use a function to create those distributions:

```
alphas = np.arange(0.10, 1, 0.10)
betas = np.arange(0.10, 1, 0.10)
gammas = np.arange(0.10, 1, 0.10)
seasonality = 12
steps = len(energy_test)
```

We use the Numpy arange() function. It takes a range (with a starting value and an upper limit value) and a step size value, and generates the in-between values in that range, that follow that step. A note to remember is that in Python ranges DO NOT include the upper limit. So, we set the upper limit to be 1, but it will not be reached, the list of values will stop at the last step before reaching it, which is 0.9

Next we need to generate a list of permutations (unique combinations of the three coefficients). As you know 3 items with 9 possible options for every means there are 9 to the power of 3 possible ways to arrange those. That means we need to create a list of 9*9*9 combination for a total of 729 options to explore.

We can do that with the itertools library. It contains a method .product() which returns the Cartesian Product of the sets it is fed. Which is exactly what we want. We will supply the three sets of coefficient we created in the previous step, and the .product() method will return all combinations between them. We than store that returned result into a new list variable ‘abg’:

```
abg = list(itertools.product(alphas, betas, gammas))
```

Here comes the more complex part. We have to create an algorithm that takes all of the necessary information and runs the optimisation process, finally returning which values to alpha, beta and gamma got the best results. Ideally, we want to be able to run this algorithm multiple time, so a good idea will be to make it into a python function that we can call when we need it.

Let’s define that function. We will call it tes_optimiser. It needs to take as arguments the following: the training data set, the testing data set, the list of combinations of coefficients to try, the type of the trend and the type of the seasonality (for the smoothing model), as well as the seasonality period and how many steps of forecasting the model should do.

The seasonality period length we already defined two steps up in the code, where we declared seasonality = 12. We also set the steps to be equal the length of the testing data set.

Then, inside the function we need to create empty variables for the three smoothing coefficients, where we will store the running best value. Every time a combination of values performs better than anything before them, we store that combination in these variables.

Then, we create a model with the assigned trend and seasonality type, we train (fit) it using the smoothing coefficients in the current iteration of the loop (running through all combinations in the 'abg' list). We use it to predict some values (as many as the length of the testing set). Then we compare the predictions to the testing set. We calculate the mean absolute error (mae) for this iteration and compare it to what we have stored in the running variable 'best_mae'. This is why we needed to set the mae to be an infinite floating-point number at the start of the loop, so that it can be compared against, regardless of how large the current mae turns out. On every iteration, if the 'mae' is smaller than what was the previous smaller 'mae', the set of alpha, beta and gamma of that respective iteration will be stored in their temp variables.

We can print out the currently running combination, so that we know the optimisation algorithm is working. At the end of the loop, when all combinations in the 'abg' list have been tested, and the best one is stored in the respective variables (best_alpha, best_beta and best_gamma), they can be printed and then returned as an output of the function.

```
def tes_optimiser(train,
                  test,
                  abg,
                  trend_mode='add', #If no value – use 'add'
                  seasonal_mode='add', #If no value – use 'add'
                  seasonal_period=seasonality,
                  step=steps):

    """
        This is a function which optimises the smoothing coefficients
        for a Triple Exponential Smoothing model
    """

    #Create empty variables to store
    #the optimised values in when we get them
    best_alpha = None
    best_beta = None
    best_gamma = None
    #infinite float value allow for easy comparison,
    #regardless of how large the compared value is
    best_mae = float('inf')
    #create the model
    tes_model = hw.ExponentialSmoothing(train,
                                        trend=trend_mode,
                                        seasonal=seasonal_mode,
                                        seasonal_periods=seasonal_period)
    #we run a loop to iterate every combination of coefficient
    #values and check how well the model performs
    for comb in abg:

        #train the model with the current iteration of coefficients
        trained_model = tes_model.fit(smoothing_level=comb[0],
                                      smoothing_trend=comb[1],
                                      smoothing_seasonal=comb[2])
        #make predictions
        predictions = trained_model.forecast(step)

        #chaeck what error these predictions lead to
        mae = mean_absolute_error(test, predictions)

        #if the error is smaller the best we have found so far:
        if mae < best_mae:
```

```
            best_alpha = comb[0] #store the current iteration alpha
            best_beta = comb[1] #store the current iteration beta
            best_gamma = comb[2] #store the current iteration gamma
            best_mae = mae #store the current error

    #At the end of the loop print the best values
    print(f'best_alpha: {round(best_alpha, 2)},
            best_beta: {round(best_beta, 2)},
            best_gamma: {round(best_gamma, 2)},
            best_mae: {round(best_mae, 4)}')
    #return the best values as an output of this function
    return best_alpha, best_beta, best_gamma, best_mae
```

Now that we have the function defined, we can run it with the arguments we want. This will take a while, depending on your computer's hardware it may take a few minutes to finish this optimisation process.

```
best_alpha, best_beta, best_gamma, best_mae =
                          tes_optimiser(energy_train, energy_test, abg)
```

```
best_alpha: 0.1, best_beta: 0.1, best_gamma: 0.4, best_mae: 2673.899
```

Finally, we know exactly what smoothing coefficients to use from now on, for forecasting this particular process, using a triple exponential (Holt-Winters) smoothing method. We can now create a best model and train it using those coefficient values.

```
best_tes_model = tes.fit(smoothing_level=best_alpha,
                         smoothing_trend=best_beta,
                         smoothing_seasonal=best_gamma)
```

Let forecast some values. As usual, we will forecast 12 values (as the length of our testing set).

```
energy_forecast_best = best_tes_model.forecast(steps=len(energy_test))
```

Like before, we convert the list of forecasted values in a Pandas Dataframe object and make sure the index column (containing the dates) is in a datetime format.

```
energy_forecast_best = pd.DataFrame(energy_forecast_best,
                                    columns=['Energy'])
energy_forecast_best.index =pd.to_datetime(energy_forecast_best.index)
```

Let's plot the newly acquired set of forecasted values:

```
plt.plot(energy_train, color='blue', linewidth=1)
plt.plot(energy_test, color='green', linewidth=1)
plt.plot(energy_forecast_best, color='r', linestyle='--', linewidth=1)
plt.rcParams['figure.figsize'] = [16, 7]
plt.xticks(rotation=45, ha='right');
```

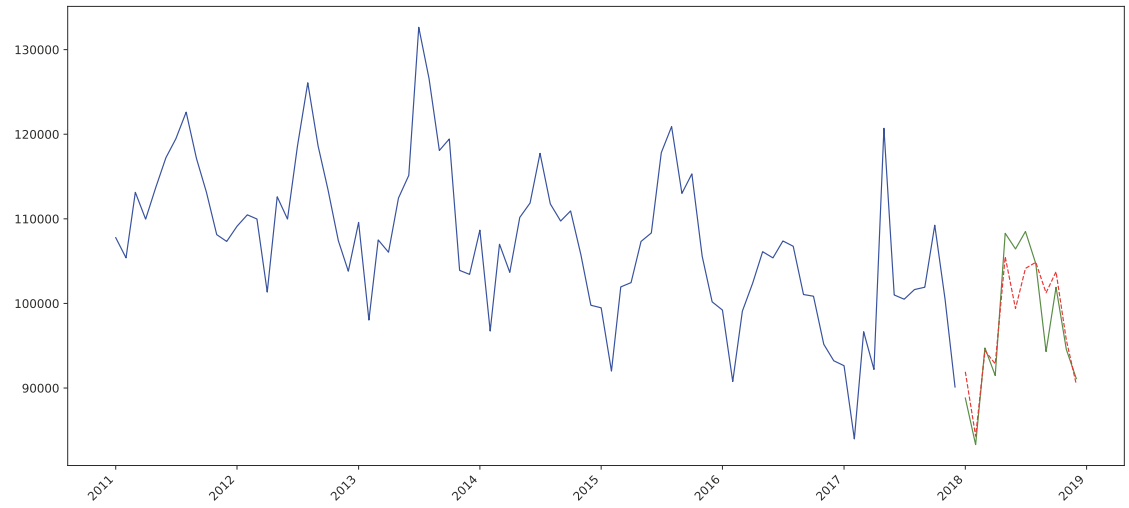


Figure 69. The green line is the test data set (last year of actually observed data). The red dashed line is the forecast, using the combination of best values for the smoothing coefficients, found during the optimisation process.

Looking at Figure 69 it might not look much different than what we got on Figure 68 but there is only one way to know for sure, we need to measure the error again. Let's calculate the RMSE for the model with these coefficients:

```
rmse_average = math.sqrt(mean_squared_error(energy_test['Energy'],
energy_forecasted_best['Energy']))
print(rmse_average)
```

```
3612.6298174699123
```

It seems our optimisation efforts were well rewarded. We managed to lower the error even more. just to put that into perspective, if we take the average of all consumption (which we did in the first exercise of this chapter) we got 107451.37 kWh of energy consumption. The 3612,63 value we got as an error, means we are able to forecast with and average error of ± 3,34%, which is not bad at all.

VI.7. Regression method

The Regression method is part of correlation analysis, which is a statistical data processing method used to study coefficients (correlations) between variables. The purpose of the correlation model is to identify the degree of association between two phenomena. If the purpose of the study is focused on forecasting a specific variable, based on information from another variable, then a regression method is applied.

A regression model shows changes in one variable as a function of changes, or differences in fixed values, of another variable. Analysis compares correlation coefficients between one or more pairs of variables to establish statistical relationships between them. Usually, we denote the independent variable (the factor) with x, and y stands for the dependent variable (the outcome). For this exercise we will assume the relationship between the variables is linear. In reality it can be much more complex, may include more than one factor (input, or independent variable) and can follow any kind of curve you can think of. The most common ones, which are also available in all data processing software packages are linear, power,

exponential, logarithmic, polynomial. We will exclusively look into linear regression, where the forecasted value $\hat{y}_i$ is calculated with a simple line equation:

$$\hat{y}_i = mx_i + b \qquad \textit{(VI.7-1)}$$

where m and b are the slope and the intercept of the line, and are calculated such that the total sum of squares TSS of the difference between the calculated and observed values of y, is minimised:

$$\arg\min\,(TSS) = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i-1}^{n}(y_i - mx_i - b)^2 = \sum_{i=1}^{n}(\hat{\epsilon}_i)^2 \qquad \textit{(VI.7-2)}$$

where y_i is the actual value of the i^{th} observation, $\hat{y}_i$ is the predicted value for the same position, and $\hat{\epsilon}_i$ is the error (residual), which is the difference between the two.

The relation between the analysed variables can be real, i.e. there is a causal (functional) relationship between them, or statistical (stochastic) dependence. For example, the relationship between x and y may be so significant that if one knows what value X will take, applying the model can automatically obtain the value of the dependent variable Y. In such situations, the relationship between x and y is said to be functional. For example, if we assume that the relationship between the marketing budget of a company and the volume of sales have a functional relationship, and based on historical data we have built a regression model, knowing the budget for the next year we can predict the sales revenue.

Of course, not all processes have functional (causal) dependence. As mentioned before, many processes are too complex, and depend on many factors, that are difficult to fully account for. For example, it is obvious that the amount of precipitation is related to the volume of the agricultural harvest, but this does not mean that the relationship between the amount of precipitation x [l/m^2] and the yield y [kg/m^2] is functional. In addition to rainfall, yield is affected by other factors, such as the type of soil, the type and amount of fertilisers used, the number of sunny days during the period, etc. In these situations, a change in one quantity affects another only statistically (on average), i.e. we have a statistical relationship between the values. Think of it this way, if the relation is statistical, when the input parameter changes, the average value of the output changes. Statistical dependence is called correlation.

We can judge the quality of the regression model by several indicators, the main one of which is the coefficient of determination R^2. It explains what percentage of the variation in the output can be attributed to the variation in the input data. Or, in other words, what percentage of the deviation of the values (from the mean) of the outcome, due to the regression, is expected (can be explained) by the changes in the factor variable. Respectively, the coefficient indicating the degree of uncertainty in the model is K^2. The coefficient of determination R^2 and the uncertainty coefficient K^2 add up to 1 (100%).

When interpreting the results of the correlation analysis, we describe the direction of the relationship (positive or negative), based on whether there is a direct or inverse relationship between the variables. This means that if we have a positive sign of the relationship, an increase in the importance of one variable will lead to an increase in the importance of the other, and vice versa - if the sign is negative, then an increase in the importance of one variable leads to a decrease in the importance of the other.

The other important indicator is the so-called p-value. The p-value is an indicator of whether the null hypothesis is likely (and how much). In practice, the p-value is the probability of getting a result that is at least as extreme as the result obtained if we assume that the null hypothesis is true (i.e., we assume that the factor magnitude has no effect on the result). A low p-value (lower than 0.05) indicates that the null hypothesis can be rejected. In other words, a predictor with a low p-value is statistically significant and changes in the factor magnitude really affect the outcome. We can judge the degree of association by the corresponding value of R^2.

In regression analysis, we generally want our model to have statistically significant variables (low p-value) and to have a high R^2 value. When a model has a low P and low R^2 it means that the model has significant variables but does not explain the variability well. At first glance, this seems strange and seems to have no logic, but let's look at some examples:

Exercise VI.7-1: Forecast future sales volume (revenue), based on the allocated marketing budget, using linear regression.

We are going to use a different data set this time. We need a data set for which it will be better suited using linear regression as a prediction model, than the Library Energy Consumption data, that we used until now.

The sales to marketing budget data set contains information for the sales volume of a company (in thousands of currency units), against the marketing budget (in thousands of currency units), that has been used to achieve that sales volume [10]. Our task will be to find out if there is correlation between these two variables and to create a linear regression model, that can be used to forecast what budget will be necessary to achieve a specific (target) sales volume. Or, if we know what budget is planned for next year, then what amount of sales can we expect.

Let's start by importing the necessary additional packages we will need. The new one here is the Statistics Models (statsmodels) API, which we will call under the alias 'sm'. It contains the regression modelling tools we need for this exercise. All other packages we have already used in previous exercises, and you should be familiar with them by now.

```
import math
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
from sklearn.metrics import mean_squared_error
```

If we open the text (.csv) file, containing the data, we will see that it has two columns of data, one for the marketing budget and one for the revenue that corresponds to that budget. I am going to rename the column names, to give them shorter designations. I'll change the "marketing_budget(thousands)" to be just "m_budget" and the "actual_sales(millions)" will be just "sales". We can do that in the text file. Let's load those values in a Pandas DataFrame:

```
sales = pd.read_csv('sales_dataset.csv')
```

As we previously discussed, it is always a good idea to visualize your data before you start doing anything else. That way you will have at least a rough idea what you are dealing with. Depending on how we frame the question, our visualization might look differently. For example, if we assume we know the budget, and we want to forecast the sales, then it makes

sense that the budget is on the X axis (the factor, or the independent variable), and the dependent variable, the one we want to forecast, which depends on the budget, will be on the Y axis. For this visualisation we are going to use a different plot kind – a scatter plot. Unlike before, when we used lines (or curves) that connect consecutive values, here the values are not ordered in any specific way. So, a line between the dots will only create a mess. We are more interested in the separate observations (data points), rather than their order, as there is no temporal element:

```
plt.scatter(sales['m_budget'], sales['sales'], color='blue', s=3)
plt.rcParams['figure.figsize'] = [13, 4]
```

The .scatter method of the plot class takes X and Y arrays (we can actually just use the Pandas Dataframe object as a single argument, but for clarity, I am distinguishing between the two columns). The argument 's' is the size of the data point marker.

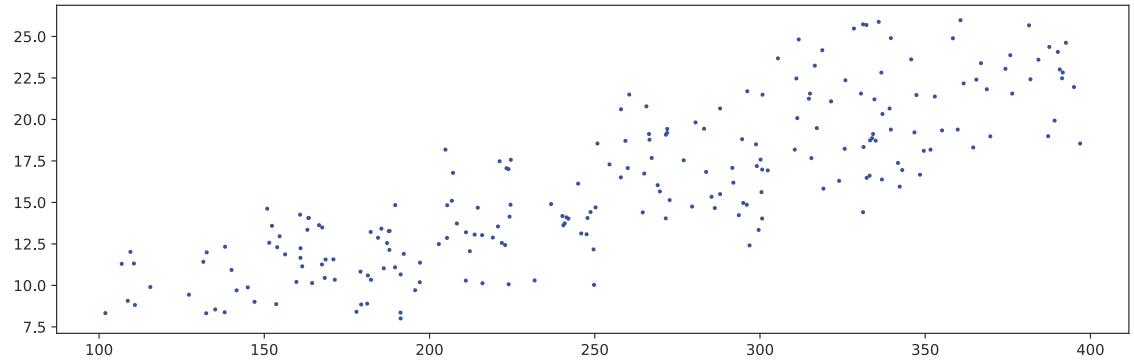


Figure 70. A scatter plot of all the data point in our data set. The X-axis is the marketing budget (in thousands currency) and the Y-axis is the achieved revenue (in millions currency)

I hope you can see why I have chosen this data set to apply linear regression on it. It should be quite obvious that there is a clear upwards trend in the relationship between the marketing budget and the revenue.

In order to train and test the model, which we are about to create, once again we need to split the data into two sets. Previously, when doing this we took advantage of the index of the Dataframe. This time, we want to take random rather than sequential records, so we are going to use a different approach. The .sample method does exactly what we want. We only have to specify what fraction of the records we want – let's say 80% should do the trick.

```
#take 85% of the samples at random and place them in a training set
sales_train = sales.sample(frac=0.85, random_state=42)
#take the original set and subtract the training set from it for test
sales_test = sales.drop(sales_train.index)
```

The random_state variable is a seed for the RNG (random number generator) which will allow us to have the same set of "random" numbers every time the code is ran. That way, the chosen records to go in both sets will be the same for everyone, every time this cell of the notebook is ran. Then, whatever records were selected for the training set are dropped from the original and rest are saved in the test data set.

It is time to start creating the linear regression model. Our first task is to determine the factor (the X values). Since we chose to forecast the revenue, based on the budget, then the budget will be our factor.

```
#adds an intercept (constant component) to the analisys
X = sm.add_constant(sales_train['m_budget'])
```

We will call the .add_constant method of the statistics models package and pass the values of the 'm_budget' column.

Now we can create the linear regression model, using the ordinary least squares (OLS) fitting method. We will pass in the Y values (our sales_train set) and the X values (in the already created constant object with the same name).

```
reg_model = sm.OLS(sales_train['sales'], X)
```

It is time to let the curve fitting optimization run. To do that, we call the .fit() function, as we have done with previous models as well.

```
trained_reg_model = reg_model.fit()
```

We now have the trained regression model, stored in the "trained_reg_model" object. If we want to know what the model looks like mathematically (the equation of the line that represents the line that was fitted to the data), we need to extract the parameters of the model – the slope and the intercept. These are stored in the .params property of the trained model. It is an array with two elements, the first element (with index 0) is the intercept, and the second element (with index 1) is the slope.

We can create a new string variable, let's call it 'equation', and build a string that is formatted to contain the necessary information:

```
equation = f'reg equation: ŷ = {round(trained_reg_model.params[1],
2)}x + {round(trained_reg_model.params[0], 2)}'
print(equation)
```

The 'f' in front of the string quotations means we want to format the string that follows. The curly brackets {} are used to inject code inside the string. That allows us to place the parameters of the model in the text of the equation (rounding them to the second digit after the decimal point first).

```
reg equation: ŷ = 0.05x + 2.88
```

We can see that a forecasted value ($\hat{y}$) can be acquired by multiplying the slope (0.05 in our case) with the value of the planned marketing budget (x) and then added to the intercept (2.88 in our case). That way we can get a projected sales volume for any budget.

It would be nice to visualise the model on top of the existing data. To do that we could use the model to acquire the theoretical revenue values for all the existing marketing budget values in the data set, and then plot them.

```
#Create ŷ values = m * x + b for every x in the training set
sales_train['predicted'] = [trained_reg_model.params[1] * x +
trained_reg_model.params[0] for x in sales_train['m_budget']]
```

With this code we create a new column 'predicted' in our DataFrame. We then use an inline loop to iterate through all the values of marketing budgets in the 'm_budget' column (stored in the temporary loop variable 'x') and use those values with the parameters we know to generate predictions.

Now our DataFrame has all the data necessary to plot everything we want to see. We can get the scatter plot of the original data and the line plot of the 'predicted' values on top of it. Let's do that:

```
plt.scatter(sales_train['m_budget'], sales_train['sales'],
            color='blue', s=3)
plt.plot(sales_train['m_budget'], sales_train['predicted'],
            color='red', linewidth=1, label=equation)
plt.legend(fontsize='medium', loc='upper left')
plt.rcParams['figure.figsize'] = [13, 4]
```

Notice we are using the 'equation' string we constructed as an input to the label of the line plot. Then on the next line of code we assign the legend of the graphic (which is essentially that label) to be placed in the upper left corner.

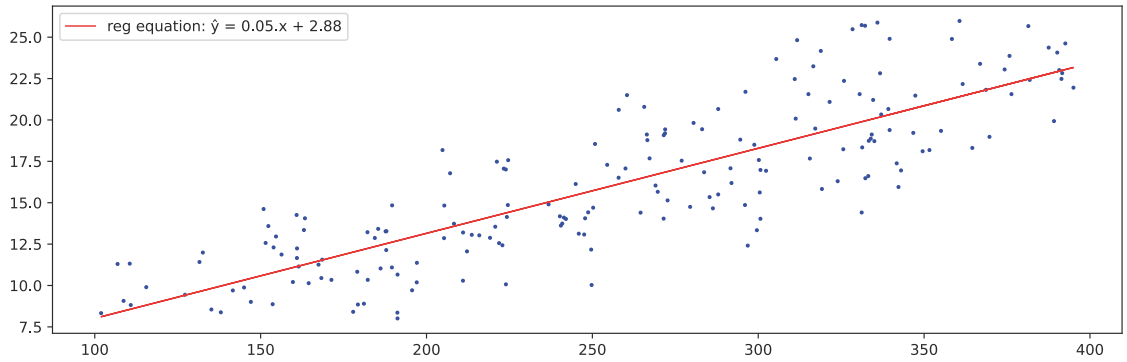


Figure 71. Scatter plot of the training data set with added trend line (linear regression prediction model). The equation of the model can be seen in the upper left corner.

Now we are ready to generate some predictions, based on the budgets we already have stored in our test data set, which we can then compare to the actual sales values, and get an estimate of the error of our prediction model.

We will approach this the same way we did when we constructed the values for plotting the regression line in Figure 71, using an in-line loop which iterates through all the marketing budget values in the test data set, and then calculated a predicted revenue (sales) value for that budget, using the line equation. The results will be stored, again, in a new column in the test data set called 'predicted':

```
sales_test['predicted'] = [trained_reg_model.params[1] * x +
trained_reg_model.params[0] for x in sales_test['m_budget']]
```

We can now use the new 'predicted' column to create a new plot, with just the test data. But this time, instead of a line, let's also plot the predicted values as points, so we can better understand what is happening:

```
plt.scatter(sales_test['m_budget'], sales_test['sales'],
            color='green', s=3)
plt.scatter(sales_test['m_budget'], sales_test['predicted'],
            color='red', s=5, label=equation)
plt.legend(fontsize='medium', loc='upper left')
plt.rcParams['figure.figsize'] = [13, 4]
```

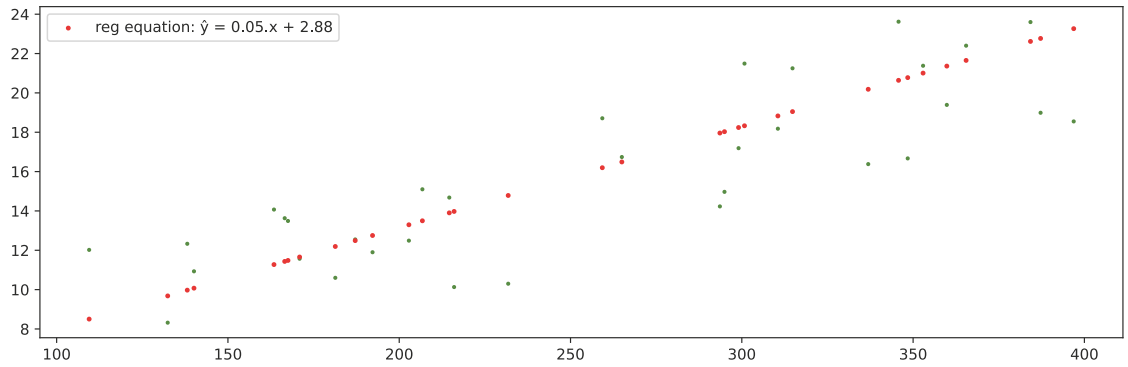


Figure 72. We can see all the red points (predicted values) corresponding to all the green points (actual values). It is obvious that the predicted ones lie on the line, that is defined by the regression equation.

If we calculate the difference between each red point and its corresponding green point, we will get the errors in those positions. The sum of the squares of those errors is what we use to calculate the mean squared error. So, let's do that and see how well is our prediction model doing:

```
rmse_average = math.sqrt(mean_squared_error(sales_test['sales'],
                                            sales_test['predicted']))
print(rmse_average)
```

```
2.507341911931452
```

It seems, on average, we will be about 2.5 million in revenue wrong in our prediction, if using this model. Of course, we don't have anything to compare that accuracy to, at the moment. The only thing we could do, to get a better understanding if our model is even appropriate, is to get the parameters of the model and see if they make sense.

We discussed in the beginning that there are two things we are particularly interested in, the R^2 and P-value, because those are the ones that tell us if there is strong correlation between the factor and the dependant variable, and how much of the value that we are attempting to predict is based on the change of the factor.

Let's see what are these parameters for our model. We can print the summary of the curve fitting process, which is stored in the .summary property of the trained model object:

```
print(trained_reg_model.summary())
```

```
OLS Regression Results
==============================================================================
Dep. Variable:                  sales   R-squared:                       0.707
Model:                            OLS   Adj. R-squared:                  0.705
Method:                 Least Squares   F-statistic:                     450.7
Date:                Wed, 28 Sep 2022   Prob (F-statistic):           1.06e-51
Time:                        15:31:43   Log-Likelihood:                -444.74
No. Observations:                 189   AIC:                             893.5
Df Residuals:                     187   BIC:                             900.0
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          2.8778      0.652      4.411      0.000       1.591       4.165
m_budget       0.0514      0.002     21.229      0.000       0.047       0.056
==============================================================================
Omnibus:                        3.042   Durbin-Watson:                   1.987
Prob(Omnibus):                  0.218   Jarque-Bera (JB):                2.816
Skew:                           0.225   Prob(JB):                        0.245
Kurtosis:                       2.606   Cond. No.                         945.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correct
ly specified.
```

Figure 73. Summary of the curve fitting process for the linear regression model. Coloured are the parameters we are particularly interested in – in blue is the R^2, in yellow is the p-value, and in green are the slope and the intercept of the line equation.

I've coloured the parts that we are specifically interested in. First of all, in green I've pointed out the two parameters of the line equation – the slope and the intercept. We already found them in their designated property, but we can also find them here.

Then, we can see in light blue that the R^2 is 0.707. That is a decent value, it basically means about 70% of the change in the revenue can be explained with the change of the marketing budget. The other 30% are due to other factors, which are not part of this prediction model.

The equivalent of the p-value, for this specific Python statistics package, marked in yellow, is the F-statistic. It is also known as F-test and shows the overall significance of the model. It indicates whether our linear regression model provides a better fit to the data, than a model that contains no independent variables. It is a measure of the probability that the null-hypothesis is true. The fact, that it is a very small number ($1.06*10^{-51}$) means we can safely discard the null hypothesis and assume our model is statistically significant.

VI.8. Recurrent neural network (RNN)

Artificial Neural Networks (ANNs or just NNs) are a powerful Machine Learning tool that can be employed for various data processing and decision-making tasks. ANNs are inspired by the structure of the human brain, and the way biological neurons work, and in an abstract way mimic the way we process information. If you think about it, we acquire information with our sensors (eyesight, touch, smell, hearing, etc.) and then that information (depending the type of it) leads to different decisions, based on our learned experiences. For example, if your phone rings and an unknown person is on the other end, you can easily guess (classify) the person's gender and relative age (young / old). The reason you can do that is because over the years you have built a natural association between the features of speech (sound) and

people's gender or age. In your brain, there are neurons, which are responsible for (will activate if) those sound features (frequency, amplitude, etc.) are present in the stimulus (the input information).

Much the same, when we want to create and teach an artificial neuron network to do the same task, we take the sound and represent it as a sequence of values that we feed to a network of neurons. A huge subset of neural networks are the so called Recurrent Neural Networks (RNNs). Unlike feed-forward ANN models (like Multi-Layer Perceptrons - MLPs and Support Vector Machines - SVMs), where each new sample of data fed through the network is a stand-alone case, RNNs allow the output of the network (or any of its layers) to be fed back in as an input.

Of course, there are many different ways you can configure the feedback loop, but the important part is that in a way, it allows the network (or specific layer of it) to have memory, to remember what was the previous state it was in. This design allows those networks to recognize patterns in sequences of data. Where new information is not stand-alone, but might need to be interpreted in some past context.

In this example we will use one of the most prominent types of RNNs - the Long Short-Term Memory network (LSTM) to predict the next value in a time series, something LSTMs are really good at. LSTMs are also often used for Natural Language Processing (NLP), both audio and text (like suggestive typing, translations and so on). They are also used for genome sequencing and other tasks that involve data in a sequence. The reason why we are going to use LSTM instead of classic RNN is that classic RNN architectures have a problem with "remembering" information for too many steps back. There is a fundamental limit to what their memory can handle. Whereas LSTMs are specifically invented to handle Long-Term holding of information (hence the name).

A typical LSTM cell structure is shown on the next figure. You can see there are three input channels to the cell, The characteristic one is the *C-channel* which is the state (and the input is taken from the output of the previous step in time C_{t-1}). You can think of this channel as the memory of the system. It is easy for information to flow through it unchanged, or with very few alterations. On every new step (new input) something can be added or removed from that memory, based on the internal structure.

The internal structure consists of gates, which regulate what gets added or removed from the state. The first one is the "forget" gate. It decides what to remove (forget) from the state. That happens by passing the input signal through an activation function (usually sigmoid), which decides which parts of the state will be "forgotten" and then doing an element-wise multiplication to execute that action.

Next is the addition of new input to the state. To do that update we again use a sigmoid layer to determine good candidates for input (i_t) and then we do another element-wise multiplication with the respective part of the state ($\tilde{C}_t$), decided by the weights received from a hyperbolic tangent activation function. The result then goes through an element-wise summation with the current state to produce the new state C_t.

And finally, we need to decide what is going to be the output. We take a filtered version of the input and run it through yet another sigmoid layer to get only the parts we want to output (O_t). Then we pass the current state through a hyperbolic tangent to receive the necessary

weights to multiply the selective ouput with. The result of theis element-wise multiplication becomes the actual output of the cell (Y_t)

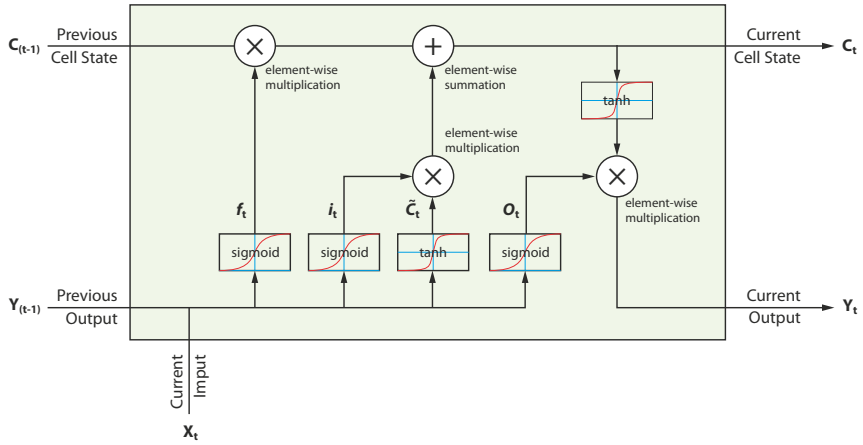


Figure 74. Typical structure of a LSTM cell.

Nowadays almost everyone using RNNs is in fact using a version of an LSTM. And a lot of what was previously done with LSTMs is now moved to attention-based architectures (transformers), like the ones used in modern LLMs (Large Language Models) like GPT, Lamma, Gemini and others. But for our simple example LSTM would still be a very good solution.

Exercise VI.8-1: Use Python to build a LSTM RNN and train it to predict the next value in the timeseries.

Let's start by importing the necessary libraries.

The first few (math, Pandas, Numpy, Matplotlib and Mean Squared Error) should be very familiar to you by now, so let's see what the rest are for.

MinMaxScaler is part of the pre-processing package of SKLearn and is essentially a normalisation algorithm. We will discuss it in more detail when we get to it in the code. Sequential is the class that describes RNN models. It is part of the Keras package, running on top of Tensorflow framework [11].

Next, we import the three kinds of neural layers that we are going to use. The LSTM is obvious, but we also need Dropout layers. These are layers that help us avoid one very significant problem with RNNs - the overfitting. We will discuss what that is, and how the Dropout layer works, when we get to that part of the code. One last part of the neural architecture is the Dense layer, which is the final layer in our network, which is responsible for changing the dimension of the output to match our needs. We will discuss it in more detail as we get to it.

And finally, we are going to import the set_random_seed method (part of the Keras utilities package), which will allow us access to the seed for the random number generator, which in turn will allow us reproducibility of our results:

```
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

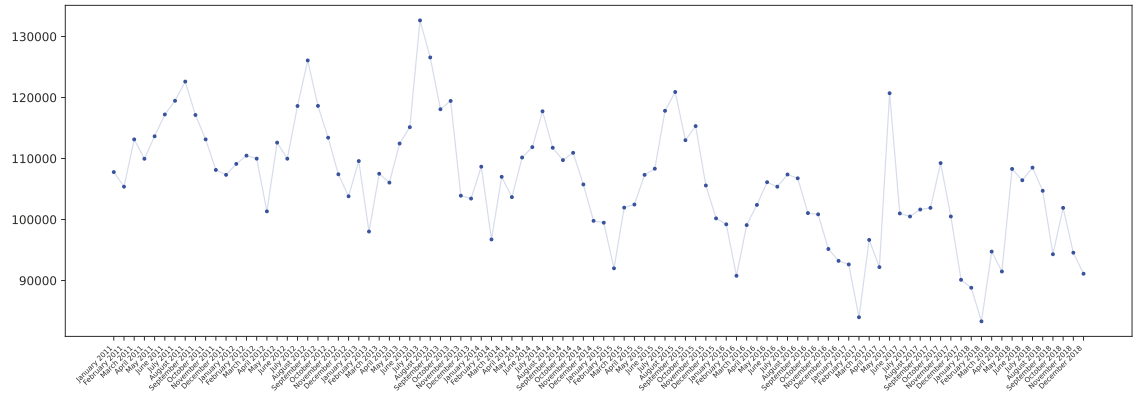
```
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import set_random_seed
```

Let's load the full dataset, using the already well familiar Pandas package. Make sure you load the already pre-processed data file. The result we'll store in the 'energy' variable:

```
energy = pd.read_csv('data/electricity-consumption-from-2011-2-
PREPROCESSED.csv')
```

As usual, our first task is to visualise the data, so that we have good visual reference to what we'll be working with:

```
plt.plot(energy['Date'], energy['Energy'],
         color='blue', linewidth=1, alpha=0.2)
plt.scatter(energy['Date'], energy['Energy'],
         color='blue', s=5)
plt.rcParams['figure.figsize'] = [16, 5]
plt.xticks(rotation=45, ha='right', fontsize=6)
```



Since the LSTM neural network we will be using only needs the sequential data points (the levels of the variable we want to predict), we can strip the data set from anything that won't be necessary. Let's store those energy levels in a new variable 'energy_only'. To remove the unnecessary parts we can use the .drop( ) method of the Pandas DataFrame, and tell it to drop the 'Date' column:

```
energy_only = energy.drop('Date', axis='columns')
print(len(energy_only))
```
```
96
```

Having that clean dataset, we can now move towards constructing a training and testing subsets from it. An important step, when generating training data is making sure we have normalised the values. The reason we normalise data when doing machine learning is to remove the scale factor of different phenomena, which would otherwise affect the weights of the features they might be representing, during training. The difference in scale can make one factor's influence be completely lost if another factor is using much higher order of magnitude values.

This is where the MinMaxScaler() that we imported in the beginning will be used. As the name suggests, it takes some set of values and scales them based on the range (max - min) of that set, using the normalisation formula (V.3-1) we discussed when talking about pre-processing data.

```
scaler = MinMaxScaler()
```

After we have defined the scaler that we will use to normalise the data, we can do the actual normalisation. We will use the .fit_transform( ) method of the scaler class and feed it our 'energy_only' dataset. The result of the scaling we will assign to a new variable called 'normalised_data':

```
normalised_data = scaler.fit_transform(energy_only)
```

As with previous approaches to prediction on this dataset, we will have to split it into training and testing subsets. The training part will be all the records except the last 12 (the last year), which we will actually try to predict. So, the 'trainig_data' variable will be formed by taking the normalised data from start up to the length of it, minus the last 12 records:

```
training_data = normalised_data[:len(energy) - 12]
#Another way to achieve the same result would be to simply do:
#training_data = normalised_data[:- 12] # which will take everything
from the start up to -12th value
```

The test set is just the last 12 records. Pay attention that we will construct the test set from the original Pandas object, as we will want to compare that with the prediction. Of course, we will have to de-normalise the prediction value first, but we will worry about that when we get there. First, let's take the values from -12 to the end of the original Pandas dataframe:

```
energy_test = energy[-12:]
```

Let's make sure we got the right set:

```
print(energy_test)
```

	Date	Energy
84	January 2018	88817
85	February 2018	83308
86	March 2018	94745
87	April 2018	91480
88	May 2018	108294
89	June 2018	106438
90	July 2018	108504
91	August 2018	104708
92	September 2018	94309
93	October 2018	101912
94	November 2018	94563
95	December 2018	91112

Next, we have to define the length of the prediction. As we said, we want to predict a full year of values, so the prediction window is 12 (months):

```
window_size = 12
```

In order to approach this time series as a supervised learning problem we need to organise the data into two sets, one consisting of training inputs and another one consisting of the actual values, that those inputs should have produced. This way the existing observations at the previous time step(s) are used as an input to forecast the observation at the current time step.

Since we are using a data with known seasonality of 12 months, we will assume a 12-value input, which we want to produce a forecast for the next point in time. Thus, we have to construct a matrix (an array) of training examples, of length 12 to be our inputs, and also, we will need a set of actual values (in a classification problem that would have been our "labels"), which the network can use to compare its own prediction to (during training).

To achieve that we will need to iterate through the training data and split it accordingly. One set with chunks of 12 values and one set with single values for the actual values corresponding to those chunks.

We then add them to their respective lists and then convert them to Numpy arrays, because the training algorithm is made to work with Numpy arrays, rather than lists.

First, we will create the two empty lists, one to contain all the chunks and one for the "true" values. We will name them "training_set" and "actual_data_set".

Then we start a loop, which needs to generate the 72 chunks of 12 values. If you examine the next figure, you will notice that if we start moving the 12-value window along the set of 84 energy values, we can only fit 72 windows (84 - 12 = 72). To store these chunks in the newly created "training_set" we use the .append( ) method on every iteration of the loop. What we append is this part of the training data, which is between the i^{th} and the $i + 12^{th}$ index. That ensures that we always store 12 values and we move one step ahead on every iteration (as i changes).

Then, within the same loop we also append the actual data, which is basically the value that follows the current window of 12 values. So, we take the $(i+12)^{th}$ value.

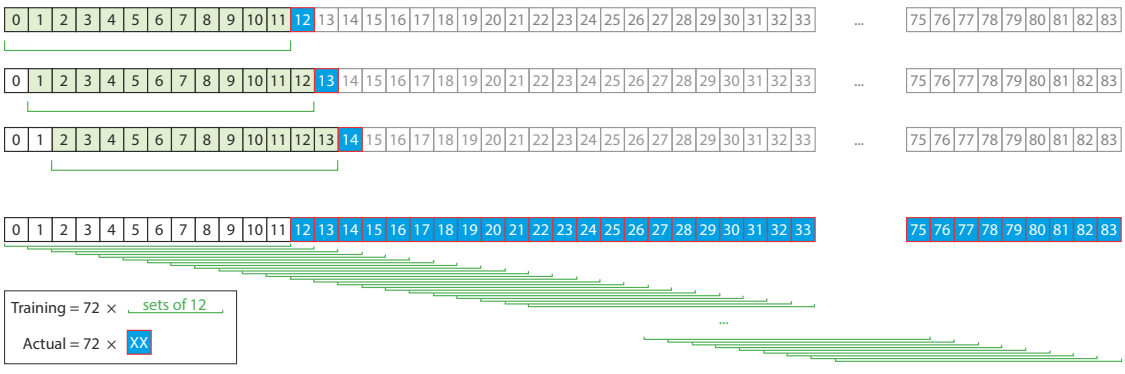


Figure 75. The process of creating the training data set.

At the end, when we exit the loop, we take the two lists and convert them to Numpy (np) arrays, replacing the contents of the same variable:

```
training_set = []
actual_data_set = []


for i in range(len(training_data) - window_size):
    training_set.append(training_data[i: i + window_size, 0])
    actual_data_set.append(training_data[i + window_size, 0])

training_set = np.array(training_set)
actual_data_set = np.array(actual_data_set)
```

Just to make sure we have what we need, we can make a fast check by printing the shape of the two arrays. We should get a two-dimensional array of 72 by 12 values for the "training_set" and a 1-dimensional array of just 72 values for the "actual_data_set":

```
print(training_set.shape)
print(actual_data_set.shape)
(72, 12)
(72,)
```

Finally, we need to do one more pre-processing operation before we can feed the data to the neural network. We need to convert the 2-dimentional array into 3-dimentional array. The reason we do this has to do with the way sequence inputs work in LSTM layers.

Technically speaking, we can view any 2-dimentional array as 3-dementional by simply thinking of the position of the values (the column and the row) as two of the dimensions, and then the magnitude, or the actual value content as the third dimension. We simply have to ask Numpy to introduce an "empty" dimension in the array in order to do that.

We will use the .reshape() method in Numpy, which takes the original array and then we supply it with the dimensions we want. Of course, this will only work if the new dimensions can be achieved (there are enough values). Since we are only introducing an empty dimension, without actually changing the number of columns or rows, we don't have to worry about that now. So, we want the new array to have the same number of dimensions as the old one (which we call by the index 0 and 1 of the .shape() method) and then followed by a 1, to indicate that third dimension:

```
training_set = np.reshape(training_set, (training_set.shape[0],
training_set.shape[1], 1))
```

Let's check if that operation was successful. If we now print the shape of the "training_set" variable instead of 72 by 12 we should get 72 by 12 by 1:

```
print(training_set.shape)
(72, 12l, 1)
```

With this we are done with the preliminary steps, and we are ready to start building the recurrent neural network (RNN). We will create a new variable which we will name "rnn" for simplicity, and in it we will store an instance of the Sequential() RNN class, which we already imported in the beginning:

```
rnn = Sequential()
```

Next, we need to set the number of neurons in the network and the dropout value. Both of these parameters we will choose somewhat at random. The truth is there isn't a formula to calculate what a good number of neurons is, or what the dropout probability should be. It is something that one has to experiment with or optimise for.

We will define the number of neurons to be 80, but you can, of course, experiment with other values too.

For the dropout, let me first explain what it is, and why we use it, and we will then choose a value for it.

Dropout is one of the most widespread methods for solving the overfitting problem when training neural networks. Essentially what it does is to randomly nullify the weight of neurons, ignoring their values for the current training pass, based on some probability value. The reason we want to do that is to make sure the network doesn't rely too heavily on a small subset of neurons (which is what happens if the network is overfitted), but rather make it rely more on its whole structure with all its neurons. There are various types of dropout and it can be applied to various layers in the network, but we will not dive that deep as to explore those subtleties during this exercise.

We will use 15% probability a random neuron’s weight is turned into a zero.

```
number_of_neurons = 80
dropout_amount = 0.15
```

We can now start building the layers of our network. We start by adding the input layer. we use LSTM class (layer) as we want a Long Short-Term Memory network architecture. For that layer we can define a lot of parameters, but we will rely on the default values for most of them.

The important parameters are the number of neurons (units), which for this layer can simply be the size of our window of values (12), the shape of the input, which will be 12 by 1 (two-dimensional) array. And finally, the call to return sequences instead of states. The reason we want to switch the return_sequences from False (the default value) to True is that this will allow us to access the hidden states of the layer, and this is a necessity if we are going to stack more than one LSTM layer (which we want to do in this example). The more layers and the more neurons in a layer the better the predictions of a network (usually). But that also can lead to overfitting, which will get us the best predictions on the given data, but will perform very poorly on new data. To avoid overfitting and to increase the generalisation capabilities of our network, as we discussed, we will employ dropout layers. So, we add a dropout layer immediately after the input layer, and we give it the predefined dropout probability, which we have stored in the "dropout_amount" variable:

```
rnn.add(LSTM(units=window_size, return_sequences=True,
                                input_shape=(window_size, 1)))
rnn.add(Dropout(dropout_amount))
```

We can continue stacking LSTM layers, followed by dropout layers for as large of a network as we want. The only difference here is the number of neurons in the internal layers. We can make them equal to the predefined "number_of_neurons" variable. For the dropout layers we continue providing the same probability.

```
rnn.add(LSTM(units=number_of_neurons, return_sequences=True))
rnn.add(Dropout(dropout_amount))
rnn.add(LSTM(units=number_of_neurons, return_sequences=True))
rnn.add(Dropout(dropout_amount))
rnn.add(LSTM(units=number_of_neurons))
rnn.add(Dropout(dropout_amount))
```

When we are done with the internal structure of the network, we have to add a Dense layer. A dense layer is a fully connected layer, meaning that each neuron in it receives input from all neurons of the previous layer. Dense layers are used for changing the dimensionality of the output from the preceding layer. Essentially, we want to collapse all of the neurons of the previous layer to single output value (the forecast). So, we give the Dense layer a neuron count of 1 (units=1). We do not supply any additional parameters, most important of which is the activation function (like we did with classification networks), this means that the Dense layer will have a linear relationship with the previous layer. You can learn more about the Dense() class (layer) in the Keras documentation [12].

```
rnn.add(Dense(units=1))
```

Before we get to the compilation of the entire network, we are going to use a specific value for the random seed, much like we did with previous exercises. That will lock the random number generator to always produce the same values, and thus allow us repeatability in our experiments. I will set the random seed to 42, because it is the answer to life, the universe and everything, but you can use any number you like.

Finally, we can compile the RNN network. We will use the ADAM (adaptive moment estimator) optimiser, and the MSE (mean squared error) for the loss function:

```
set_random_seed(42)
rnn.compile(optimizer='adam', loss='mean_squared_error')
```

With this, we are now ready to let the network learn. As with models in previous exercises, this is done using the .fit() method. We supply the training set, the labels (actual values), the batch size and the epochs:

```
rnn.fit(training_set, actual_data_set, epochs=100, batch_size=12)
```

Now that we have a trained network model, we can use it to make predictions. Before we can do that though, we have to prepare input vectors for testing. Much like we prepared the training data into chunks of 12 values, we have to do the same here. We need to address the normalised data set and take from it 12 chucks (one chunk for every prediction we want to make, and we want to make 12 predictions), and each chunk will have 12 existing values.

Assuming we are at the end of the 84 values (that's what we used for training), we want to start one full window length back, at index 72, and take the values up to index 83. That chunk of 12 values will predict the value for the next moment in time (index 84).

Then, at the next moment in time when the actual value at index 84 comes we include it in the chunk and drop the first one (index72). Thus, the second chunk starts at 73 and ends in 84. And so on, until the last, 12th chunk, will start at index 83 and end at index 94 (the last actual value we have).

Just to demonstrate how we are going to generate the indexes, we can use this simple loop:

```
for i in range(12):
    print(list(range(84-12+i, 84+i)))
```

```
[72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83]
[73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84]
[74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85]
[75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86]
[76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87]
[77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88]
[78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89]
[79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90]
[80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91]
[81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92]
[82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93]
[83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94]
```

Now we can employ this to get the values behind those indexes and store them in a list, so that we can then feed them as inputs to the network.

Let's first create the empty list, that we'll use to store the chunks, we will name it "x_test_values".

Then we declare the loop that will run for as many iterations as the length of the testing set (12), just as in the example above.

Then we take a temporary variable "x" which on every iteration will contain the values of "normalised_data" at the list of indexes that we generate the same way as in the example above.

Finally, we append the chunk of values stored in "x" to the list "x_test_values":

```
x_test_values = []

for i in range(len(energy_test)):
    x = normalised_data[list(range(len(training_data) - window_size +
i, len(training_data) + i))]
    x_test_values.append(x)
```

As you remember, the neural network takes values as arrays, so again, we need to convert the list to a Numpy array:

```
x_test_values = np.array(x_test_values)
```

Just to make sure we have the correct shape of the input we can call the .shape( ) method. We expect to have an array of 12 chunks, each of them with 12 values. And because we received "normalised_data" as a result of the Scaler in the beginning of the exercise, it is already of shape 96 by 1, so we expect our chunks to also have that extra "depth" dimension. Thus, we expect the "x_test_values" array to be 12 by 12 by 1:

```
print(x_test_values.shape)
```

```
(12, 12, 1)
```

Great! Now we run predictions with these input vectors. Let's feed the "x_test_values" to the .predict( ) method of the trained network. We will store the predictions in a new variable which we will very unimaginatively name "predictions":

```
predictions = rnn.predict(x_test_values)
```

Before we can compare those predictions to our test data set, we have to remember that both the input and the output of the network is normalised, so we have to run an inverse_transform on the predicted values in order to get them at scale with the real ones. We do that by supplying the predictions to the .inverse_transform( ) method of the scaler we already have defined. The result of that operation we store in a new variable named "unscaled_predictions":

```
unscaled_predictions = scaler.inverse_transform(predictions)
```

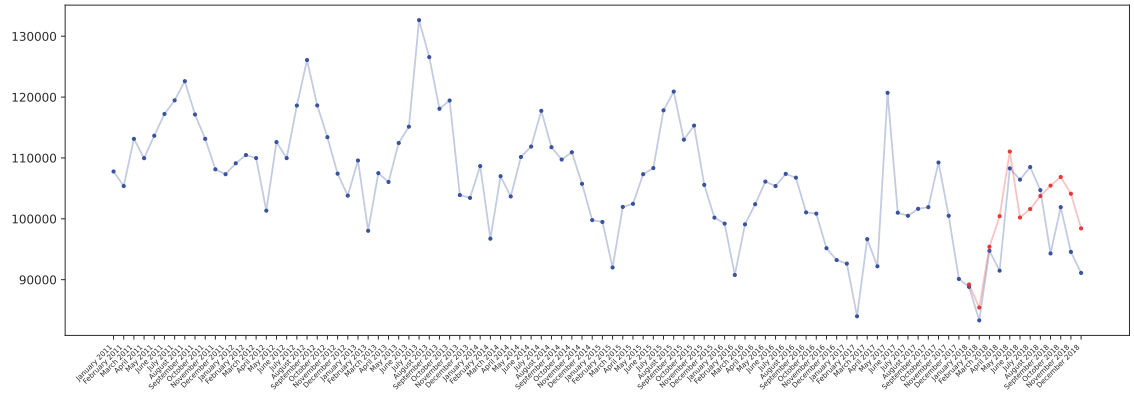
We can see the difference between the normalised and the scaled back versions of the predictions if we print the two variables. We can see that the normalised ones are between 0 and 1 (as expected) and the scaled back ones are in the same order of magnitude as the real energy values:

```
print(predictions)
print(unscaled_predictions)
[[0.16241932]
 [0.11729858]
 [0.29742387]
 [0.37860218]
 [0.5488351 ]
 [0.37479046]
 [0.39805374]
 [0.43250585]
 [0.45471022]
 [0.46675268]
 [0.40056193]
 [0.30698258]]
[[ 91320.96 ]
 [ 89094.93 ]
 [ 97981.41 ]
 [101986.336]
 [110384.78 ]
 [101798.29 ]
 [102945.98 ]
 [104645.68 ]
 [105741.13 ]
 [106335.234]
 [103069.72 ]
 [ 98452.984]]
```

In order to really see how close to the real value we are, we can again visually compare them. Let's plot the entire energy data set (in blue), and then on top of it let's plot the unscaled_predicted values (in red):

```
plt.plot(energy['Date'], energy['Energy'],
         color='blue', linewidth=1.5, alpha=0.3)
plt.scatter(energy['Date'], energy['Energy'],
            color='blue', s=6)
plt.plot(energy_test['Date'], unscaled_predictions,
         color='red', linewidth=1.5, alpha=0.3)
plt.scatter(energy_test['Date'], unscaled_predictions,
```

```
            color='red', s=6)
plt.rcParams['figure.figsize'] = [16, 7]
plt.xticks(rotation=45, ha='right', fontsize=6)
```



We can see that we got a rather good result. Of course, in order to get a numerical estimate of our accuracy, we can again use the root of the mean squared error (RMSE) function and give it the two data sets:

```
rmse_average = math.sqrt(mean_squared_error(energy_test['Energy'],
unscaled_predictions))
print(rmse_average)
6415.62264104657
```

Obviously, we are comparably close in accuracy as some of the other prediction methods we used. And this is without any optimisation of the hyper parameters of the neural network.

Chapter VII.
Clusterisation (grouping)

We can think of clusterisation as a form of classification, but one where we don't know what the classes are (or rather what characterises the classes). We only know how many of them are (or at least how many we expect there to be) in the data.

Unlike classification, where the neural networks are trained with a "teacher" or an expert supervisor (supervised learning), and generally where there is an established knowledge of which parameters are to be used as predictors, employing clusterisation means we lack that knowledge, and we rather let the computer tell us what the similarities in the recorded data are.

To better understand the case for clusterisation we can think of two examples. Let's imagine we have a lot of records of bank card transactions and maybe we want to build a system that would flag a newly incoming transaction as either normal or potentially fraudulent. Again, for the sake of the example, let's assume we have no knowledge what a fraudulent transaction looks like (what symptoms to look for to distinguish it from a "normal" transaction).

In this case we could use a clustering algorithm and tell it we want it to group the records into two groups. And when the algorithm is done with the task, we can review the transactions that fall in each of the groups (clusters), and try to analyse their features, to understand what similarities it found in them.

Although theoretically clustering algorithms could be based on various "likeness" estimation approaches, almost all applications tend to fall in the geometric distance estimation. If we go back to the transactions example above, that means that the algorithm will look for records which are geometrically close to one another, based on their coordinates in a multidimensional space, where each dimension is one of the transaction features (e.g. the amount, the country, the currency, etc.)

Of course, a more complex approach could be taken, where the features that have an ordinal numerical representation (e.g. amount) are passed through a distance-based algorithm and cardinal values (countries, currencies, etc.) are passed through a sameness-based algorithm. For our exercise we will consider one of the most widely used distance-based algorithms – the K-means.

VII.1. K-means

K-means is a method for unsupervised clusterisation (grouping), which can distribute a number of samples into k clusters. So, for a given set of observations (x_1, x_2, …, x_n), where each observation may be a multidimensional vector (the sample is described by many parameters), the k-means method aims to partition the n samples in k (≤ n) groups/clusters $S = \{S_1, S_2, \ldots, S_k\}$, so that the within-cluster sum of squares (WCSS is the sum of squared distance between each point and the centroid in the cluster) is minimised. Or, in mathematical notation, that means finding:

$$arg\min_{S} \sum_{i=1}^{k} \sum_{j=1}^{n} \left\| x_j - \mu_i \right\|^2 \qquad \textit{(VII.1-1)}$$

where k are the number of clusters (centres), n are the number of samples (records) which need to be assigned to a group (cluster), x_j is the value (the coordinate) of the j^{th} sample and μ is the value (the current coordinate) of the centre of the cluster that sample belongs to.

The algorithm runs iteratively and, on each iteration, finds the records which are closest to the current cluster centre, and then moves the centre to represent better the current selection of datapoints that correspond to it. Then it repeats the operation with the new centre coordinates. That continues until the centres stop moving, and the clusters are in their final composition. Let's see it in action:

Exercise VII.1-1: Use Python to build K-Means clusterisation algorithm, finding the centres of k groups of samples.

Let's import the libraries we will use:

```
import ipywidgets as widgets
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import datasets as ds
from sklearn.cluster import KMeans
from PIL import Image
```

We will start this exercise with a simple example in 1-dimension before we move to examples in more dimensions. Let's take the following set of numbers:

```
X = [1, 2, 5, 7, 9, 10, 11, 13, 27, 33, 34, 37,
     40, 42, 51, 76, 109, 127, 144, 163, 200]
```

which we want to group into $k = 3$ clusters. Usually, the method is initialized by choosing random values for the initial coordinates of the centres. Often the practice is to randomly use some of the already existing values in the array for initial centres.

Then, using the formula above, we calculate the distance of each element of the array to each of the centres of the three clusters. The smallest of these distances indicates that this element belongs to the group of the corresponding centre.

When all the elements are distributed, we need to make an adjustment to the locations of the centres so that they better reflect the already distributed values. For our one-dimensional case, this is done by finding the arithmetic mean of the clustered values. For the multidimensional case, this must be done for every single dimension.

This process is repeated until a state of the computational operation is reached, in which another iteration does not lead to a reallocation of the elements of the array.

For this first example, let's initially choose the centres to be equal to three of the existing values in the array. Let these be **1**, **7** and **200**:

```
centers = [1, 7, 200]
```

As usual, I would advise that our first step is to visualise the data so that we can better visualise its distribution.

```
plt.scatter(X, np.ones_like(X), s=10)
plt.scatter(centers, np.ones_like(centers), s=50, color='r',alpha=0.5)
plt.gca().margins(x=0.01)
plt.xticks(X, rotation=45, ha='right', fontsize=5)
plt.yticks([])
plt.ylim(0.5, 1.5)
plt.rcParams['figure.figsize'] = [16, 2]
```

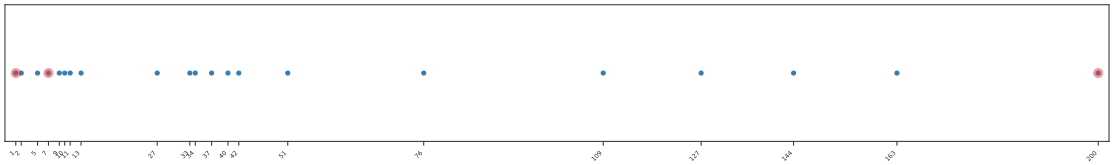


Figure 76. In blue are all the values from the set, and in red are the initial centres.

In the first iteration we look at all the numbers in the set and check which centre are they closest to. Naturally 1 and 2 are closer to **1** than to any of the other two centres. Then 5, 7, 9, 10, … up to 76 are closer to 7 than to the centres at **1** or **200**. And finally, 109, 127,… up to 200 are obviously closer to **200**.

But now we need to calculate where the centres will move to, knowing which values fall under them during first iteration. Since we are only working in 1-dimention it is easy to just find the average of the values, to get the new centre value. So, the first centre will go at 1.5 (the average of 1 and 2). The second centre will go to 28.21 (the average of the numbers from 5 to 76 in our set) and the third centre goes to 148.6

On the second iteration we again try to distribute the numbers in the three clusters, but we now use the newly calculated centres. We do this again in a third iteration (again with newly calculated centres). When we attempt a fourth iteration, we will see that the centres (and of course the distribution of the numbers) will not change. So, the third iteration is last and those are the final centre points.

First Iteration			
μ	**1**	**7**	**200**
x_i	1	5	109
	2	7	127
		9	144
		10	163
		11	200
		13	
		27	
		33	
		34	
		37	
		40	
		42	
		51	
		76	
SUM	3	395	743
New μ	1,50	28,21	148,60

Second Iteration			
μ	**1,5**	**28,21**	**148,6**
x_i	1	27	109
	2	33	127
	5	34	144
	7	37	163
	9	40	200
	10	42	
	11	51	
	13	76	
SUM	58	340	743
New μ	7,25	42,5	148,6

Third Iteration			
μ	**7,25**	**42,5**	**148,6**
x_i	1	27	109
	2	33	127
	5	34	144
	7	37	163
	9	40	200
	10	42	
	11	51	
	13	76	
SUM	58	340	743
New μ	7,25	42,5	148,6

Figure 77. The μ value is the centre for the current iteration. Every iteration the centre moves to better reflect the points that fall in that group.

Let's do the same example but using the capabilities of the SKlearn library. Let's convert our list of "X" values into a Pandas object, which we'll name "X_data". Let's also add another column to it (we'll just name it "Y"), which we'll fill with units. For this we will use the .ones_like() function in numpy. We do this because we'll need both X and Y values for visualisation, and also most K-means implementations require at least two-dimensional data:

```
X_data = pd.DataFrame(X, columns=['X'])
X_data['Y'] = np.ones_like(X)
```

Our next step is to create a K-means model by assigning it to the "D1_kmeans" object. For the value of k we will put 3, this is done by specifying the argument "n_clusters". "Training" the model is again performed by calling the .fit() function, passing our array of "X_data" values as an argument:

```
D1_kmeans = KMeans(n_clusters=3)
D1_kmeans.fit(X_data)
```

Once the model is "trained", i.e. grouping (clustering) of our values is done, we can see which value it will assign which label (i.e., which group it will associate it with). For this purpose, as in previous exercises, we use the .predict() function, passing it our array of values, and we will save the result of this action in a new variable "predicted_X_labels.

Besides the labels, the other most important thing in the whole process is to determine where the centres are for each of the groups. These values are stored (during training) in the .cluster_centers_ property, in the trained model. We can turn them into a Pandas DataFrame object, which we'll name "predicted_X_centers", containing two columns ('Centers_X' and 'Centers_Y'):

```
predicted_X_labels = D1_kmeans.predict(X_data)
predicted_X_centers = pd.DataFrame(D1_kmeans.cluster_centers_,
columns=['Centers_X', 'Centers_Y'])
```

Let's see if the centres of the clusters that our SKLearn model found match the centres we determined at the beginning:

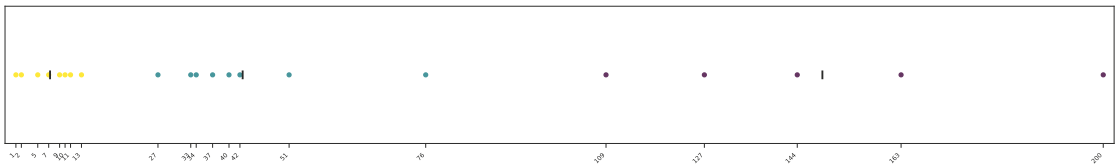
```
print(predicted_X_centers)
```

	Centers_X	Centers_Y
0	7.25	1.0
1	148.6	1.0
2	42.5	1.0

As expected, we got the same results. We can visualise the three centres on our figure. For a tag we will use the symbol "|" with size s=50, and we will also colour code the dots, that represent our set values, so that it is apparent which one belons to which centre. For that purpose, we can pass the cluster membership codes (that we now have in the "predicted_X_labels" variable and choose a colourmap to use, for example "veridis" [13]):

```
plt.scatter(X, np.ones_like(X), c=predicted_X_labels,
            cmap='viridis', s=10)
plt.scatter(predicted_X_centers['Centers_X'],
predicted_X_centers['Centers_Y'], marker='|', s=50, color='black')
plt.gca().margins(x=0.01)
plt.xticks(X, rotation=45, ha='right', fontsize=5)
plt.yticks([])
```

```
plt.ylim(0.5, 1.5)
plt.rcParams['figure.figsize'] = [16, 2]
```



Hopefully this 1-dimensional example helped you understand conceptually what the K-means algorithm does. Now we can apply it to higher dimensions of data. To do that, this time, we will generate our own (synthetic) data to work with. Here is where the datasets (ds) package of the Keras library will come handy. The SKLearn library contains ready-made functions for generating data of different shapes and parameters. We'll start with blobs.

The function we will use is .make_blobs() from SKLearn's datasets package (which we have abbreviated as "ds"). As arguments, we'll say that we want 500 records spread over 5 centres with a standard deviation of 0.9 (i.e. reasonably well clustered around the centres), and to be able to repeat the code execution with the same values, we'll say that the state of random number generator will be equal to 1.

This function will return two arrays. One will contain the records themselves and the other will contain the labels to each record (i.e. which record refers to which group). We will assign these two arrays to two variables, "blobs_data" and "blobs_center_labels" respectively:

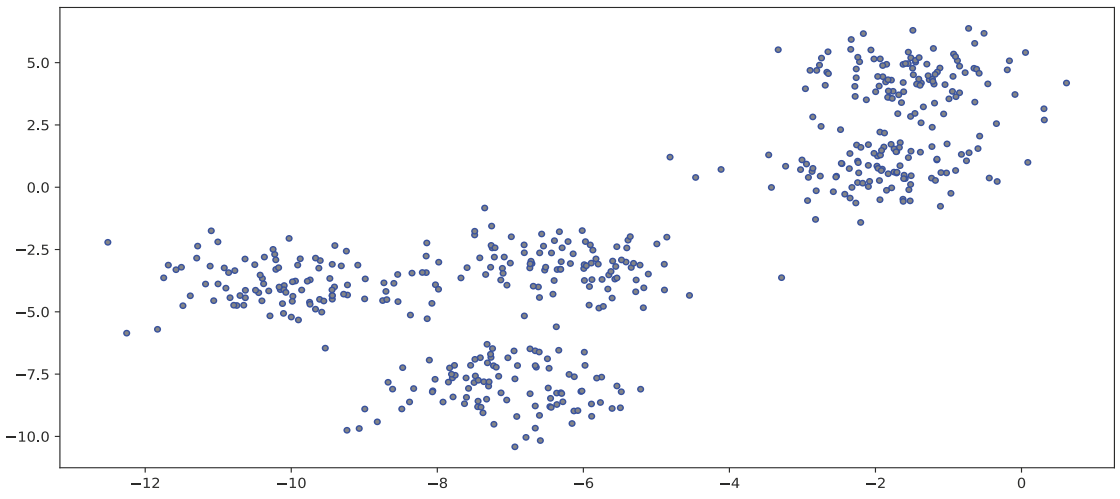
```
blobs_data, blobs_center_labels = ds.make_blobs(n_samples=500,
centers=5, cluster_std=0.90, random_state=1)
```

To make things a bit easier to read, let's again store the data into a Pandas DataFrame table with two columns "X" and "Y":

```
blobs_data = pd.DataFrame(blobs_data, columns=['X', 'Y'])
```

Let's see what we got as a result of generating the data. For the purpose of the scatterplot function, we will pass the newly generated array:

```
plt.scatter(blobs_data['X'], blobs_data['Y'], s=15, color='b',
facecolor='grey');
plt.rcParams['figure.figsize'] = [13, 6]
```



If you're not happy with the result, you can go back to the code we used to generate the blobs and change some parameters. For example, you can change the standard deviation, quantity, and state of the random number generator. This will result in a completely different distribution of values in this two-dimensional space.

If we are satisfied with the result, we can proceed to train a new model to cluster the new array. For the value of k (the number of clusters), we specify 5, since this is the value we used to generate the array. If you want, you can experiment with how the clustering will go if you ask the model to distribute the values into more or less than 5 groups.

We perform the training again with the .fit() function, while passing in the generated data:

```
kmeans_blobs = KMeans(n_clusters=5)
kmeans_blobs.fit(blobs_data)
```

We follow the sequence of actions from the one-dimensional array. First, we run the data through the .predict() function to get the labels that we will assign to the "blobs_predicted_labels" variable. Next, we take the coordinates of the cluster centres and convert them into a Pandas DataFrame, while specifying the column names:

```
blobs_predicted_labels = kmeans_blobs.predict(blobs_data)
blobs_centers = pd.DataFrame(kmeans_blobs.cluster_centers_,
columns=['Centers_X', 'Centers_Y'])
```

We can see what the labels look like. We see that they are values between 0 and 4, i.e. a total of 5 possible options, as there are 5 groups. The labels' indexes correspond to the indexes of the data we are clustering:

```
print(blobs_predicted_labels)
```

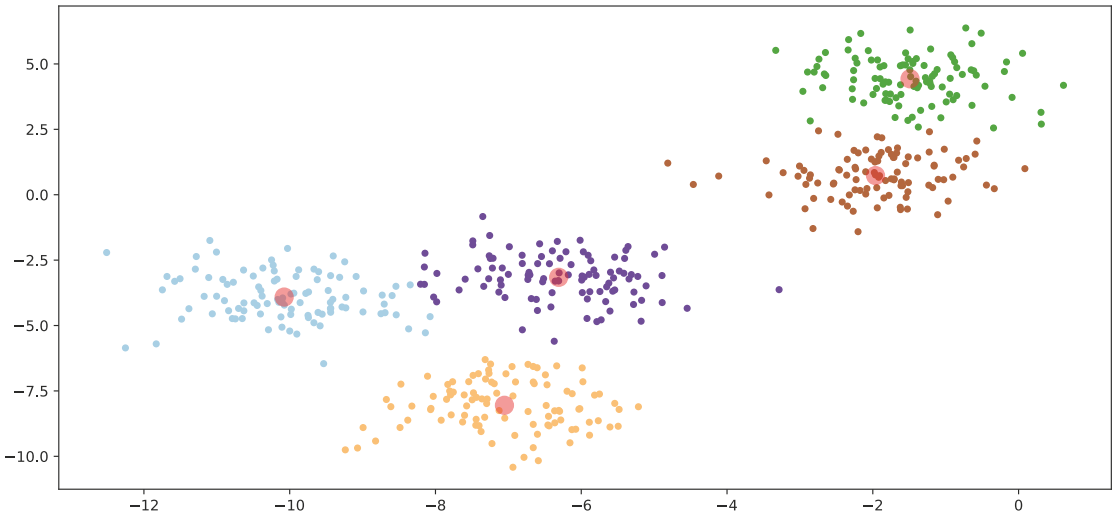
```
[4, 1, 2, 0, 2, 3, 4, 1, 3, 0, 2, 2, 1, 4, 3, 3, 0, 0, 4, 1, 3, 1,
 2, 4, 4, 1, 2, 3, 2, 1, 1, 0, 2, 1, 1, 3, 0, 4, 4, 0, 3, 3, 3, 1,
 2, 2, 2, 3, 1, 1, 3, 0, 0, 2, 2, 4, 3, 0, 2, 0, 0, 4, 4, 3, 2, 4,
 1, 1, 3, 4, 0, 2, 3, 1, 0, 1, 0, 3, 1, 1, 2, 3, 4, 3, 2, 4, 0, 4,
 4, 2, 1, 4, 2, 0, 3, 1, 2, 3, 1, 1, 4, 2, 1, 3, 1, 2, 1, 3, 4, 4,
 3, 0, 0, 4, 0, 4, 3, 4, 1, 4, 3, 1, 0, 3, 3, 1, 3, 4, 1, 0, 1, 4,
 3, 3, 1, 0, 2, 3, 1, 4, 3, 2, 0, 3, 4, 4, 4, 1, 2, 2, 2, 0, 3, 4,
 1, 3, 0, 3, 1, 1, 0, 3, 0, 3, 3, 4, 0, 0, 3, 1, 4, 0, 3, 1, 1, 4,
 3, 0, 3, 3, 2, 0, 1, 2, 4, 2, 0, 0, 1, 2, 0, 2, 4, 3, 0, 1, 2, 4,
 2, 2, 1, 4, 2, 0, 3, 2, 2, 0, 1, 0, 2, 2, 0, 0, 0, 0, 2, 0, 0, 4,
 0, 4, 1, 0, 0, 1, 0, 3, 3, 3, 0, 4, 2, 3, 1, 3, 4, 3, 0, 1, 4, 1,
 1, 3, 2, 2, 2, 2, 1, 3, 2, 3, 1, 4, 4, 1, 1, 4, 3, 1, 0, 3, 1, 4,
 3, 3, 4, 1, 2, 2, 0, 0, 3, 1, 4, 4, 1, 4, 0, 1, 4, 2, 4, 0, 2, 4,
 2, 4, 4, 2, 0, 0, 3, 1, 4, 2, 2, 4, 1, 3, 0, 2, 0, 2, 1, 3, 0, 2,
 0, 2, 2, 3, 4, 2, 1, 1, 4, 4, 1, 0, 3, 4, 2, 2, 4, 4, 2, 3, 4, 3,
 2, 2, 2, 4, 3, 1, 2, 4, 3, 1, 4, 3, 2, 1, 4, 4, 3, 4, 4, 3, 3, 0,
 2, 1, 4, 3, 2, 2, 4, 3, 4, 1, 4, 0, 2, 0, 4, 1, 1, 2, 4, 0, 2, 0,
 2, 0, 0, 0, 0, 4, 0, 4, 4, 3, 4, 2, 3, 2, 0, 3, 3, 4, 2, 0, 2, 3,
 2, 0, 0, 4, 3, 2, 0, 1, 4, 1, 1, 2, 3, 1, 2, 3, 3, 2, 0, 0, 3, 3,
 0, 1, 3, 0, 1, 4, 0, 2, 4, 2, 1, 3, 4, 3, 1, 3, 1, 0, 4, 0, 1, 4,
 2, 2, 0, 3, 4, 0, 1, 2, 1, 1, 0, 3, 2, 1, 2, 1, 1, 1, 2, 2, 2, 0,
 0, 1, 4, 1, 4, 1, 0, 0, 4, 3, 4, 0, 2, 0, 2, 3, 4, 1, 2, 4, 2, 4,
 0, 1, 4, 3, 0, 3, 1, 0, 3, 3, 0, 2, 1, 4, 3, 1]
```

We can visualize the different groups by using the predicted labels to colour the individual points in a different colour. All Matplotlib visualization functions allow this. I.e. we'll choose

the labels as blob colour codes c=blobs_predicted_labels, and we'll pass the colours themselves as a colormap (cmap). The different possible colourmaps can be seen in Matplotlib's documentation [13].

We will also visualize the centres of the clusters. For this we will use the coordinates from the model that we saved in the Pandas DataFrame object "blobs_centers". We will colour their markers red, make them with size s=150 and transparency (alpha-channel) of 50%:

```
plt.scatter(blobs_data['X'], blobs_data['Y'], s=15,
c=blobs_predicted_labels, cmap='Paired');
plt.scatter(blobs_centers['Centers_X'], blobs_centers['Centers_Y'],
c='red', s=150, alpha=0.5);
plt.rcParams['figure.figsize'] = [13, 6]
```



We can see that the clustering algorithm does quite well with this data. In fact, if you want to really understand how well it's doing, and have some sort of quantitative assessment, you can generate different datasets and go through the clustering of them all. Then compare the labels that the generating function gives you when the data is created with those that the clustering algorithm predicts. That way you'll know quantitatively what accuracy you can expect (for different types of datasets). I'll leave that to you as homework.

While K-means is an extremely good clustering method, it has its limitations that you should be aware of. When clusters are observed in the data, that cannot be linearly separated, i.e. you cannot draw a mental line between groups, but rather have to use curves, or even hypercurves (when working with more dimensions), then you may need to look for other clustering or classification tools.

To demonstrate this, let's generate synthetic data again. This time we'll use the make_moons() and make_circles() functions, which, as their names suggest, generate data distributed in the shape of opposite crescents or (concentric) circles.

For the crescents, we'll want 300 points and 7% noise. Again, we will freeze the state of the random number generator to have reproducibility of the results of the function. For the circles, we will stake two sets of points (i.e. we want to form two concentric circles), one with 500 and one with 200 points. The aspect ratio is the ratio between the inner and outer radiuses, which we will choose to be 30%. We'll also set a noise of 8% (standard deviation of the data from the mean/radius), and again lock the state of the random number generator:

```
moons_data, moons_labels =ds.make_moons(300,noise=0.07,random_state=1)
moons_data = pd.DataFrame(moons_data, columns=['X', 'Y'])

circle_data, circle_labels = ds.make_circles((500, 200), factor=0.3,
                            shuffle=True, noise=0.08, random_state=42)
circle_data = pd.DataFrame(circle_data, columns=['X', 'Y'])
```

We go the same way as with the previous data. First, we create a pattern, specifying that we want 2 clusters (since we created two crescents and two concentric circles). We then train the models using the .fit() function:

```
kmeans_moons = KMeans(n_clusters=2, random_state=0)
kmeans_moons.fit(moons_data)

kmeans_circles = KMeans(n_clusters=2, random_state=0)
kmeans_circles.fit(circle_data)
```

We use a trained model to predict the labels of the points in crescent form (i.e., which of the two crescents the model will assign them to). We also take the coordinates of the two centres, saving them to a Pandas object:

```
moons_predicted_labels = kmeans_moons.predict(moons_data)
moons_centers = pd.DataFrame(kmeans_moons.cluster_centers_,
                             columns=['Centers_X', 'Centers_Y'])
```

Let's do the same for the concentric circles:

```
circles_predicted_labels = kmeans_circles.predict(circle_data)
circle_centers = pd.DataFrame(kmeans_circles.cluster_centers_,
                              columns=['Centers_X', 'Centers_Y'])
```

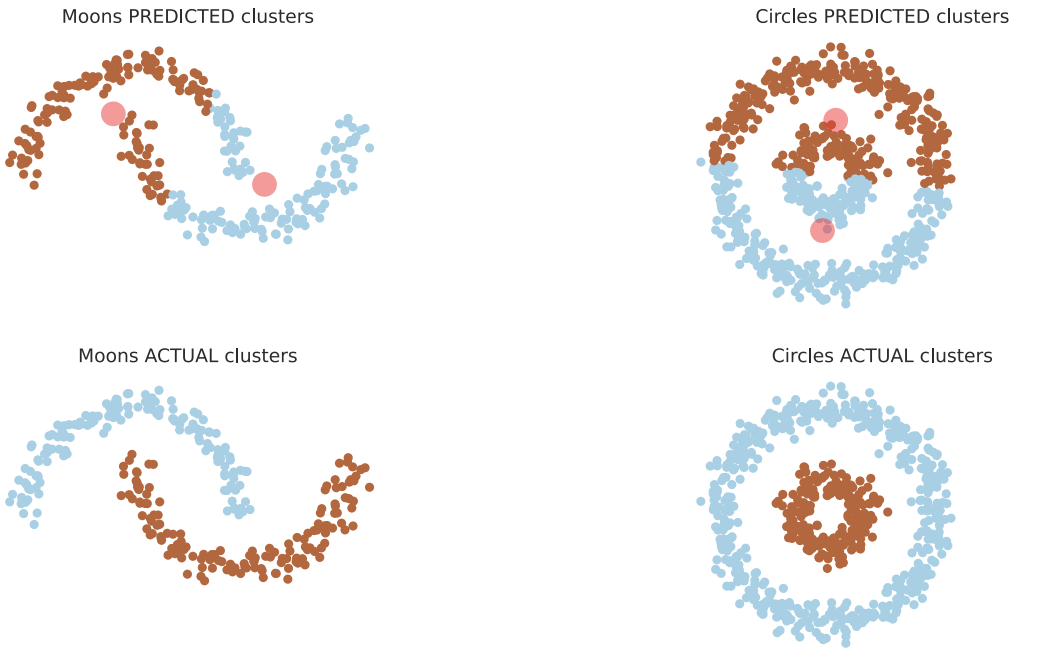
And finally, we are ready to visualise them. To be able to do a comparative analysis we will want to visualise both the original data and the clustered (distributed into groups) points. We'll want to do this for both the crescents and the circles. This means we will need a total of 4 plots. To combine 4 plots into one figure we will use the function provided for this in Matplotlib, and each separate plot will be a sub-plot of the main figure.

We call the subplots() function telling it that we want to divide the space into 2 rows with 2 plots per row. We also specify that we want "sharex" and "sharey" to be True (i.e. axes to be shared, which will assure equal scaling in both directions). We assign the result of the function to the variable "fig" and the set of subplots "ax1", "ax2", "ax3" and "ax4" distributed in a matrix with two rows and two columns.

In the first subplot, we will visualise the moons with the clusters predicted by the model (using the labels to colour the points). In the second subplot, we visualise the circles, again with the clusters predicted by the model, using the colour code labels. On the bottom row of subplots, we'll show how we actually expected the points to be clustered by color-coding them with their original labels that we got from when these arrays were generated:

```
fig, ([ax1,ax2],[ax3,ax4])= plt.subplots(2,2,sharex=True,sharey=True)
fig.subplots_adjust(wspace=0.05)

ax1.scatter(moons_data['X'], moons_data['Y'], s=15,
            c=moons_predicted_labels, cmap='Paired');
ax1.scatter(moons_centers['Centers_X'], moons_centers['Centers_Y'],
```

```
            c='red', s=150, alpha=0.5);
ax1.set_title('Moons PREDICTED clusters', fontsize=10)
ax1.set_aspect('equal', 'box')
ax1.axis('off')

ax2.scatter(circle_data['X'], circle_data['Y'], s=15,
            c=circles_predicted_labels, cmap='Paired');
ax2.scatter(circle_centers['Centers_X'], circle_centers['Centers_Y'],
           c='red', s=150, alpha=0.5);
ax2.set_title('Circles PREDICTED clusters', fontsize=10)
ax2.set_aspect('equal', 'box')
ax2.axis('off')

ax3.scatter(moons_data['X'], moons_data['Y'], s=15,
            c=moons_labels, cmap='Paired');
ax3.set_title('Moons ACTUAL clusters', fontsize=10)
ax3.set_aspect('equal', 'box')
ax3.axis('off')

ax4.scatter(circle_data['X'], circle_data['Y'], s=15,
            c=circle_labels, cmap='Paired');
ax4.set_title('Circles ACTUAL clusters', fontsize=10)
ax4.set_aspect('equal', 'box')
ax4.axis('off')
```



It is obvious that K-means fails to recognise groups (clusters) in space that cannot be separated by a straight line. In such situations, in order to associate the points with the correct cluster, a much better approach would be to use some classification algorithm that can do a non-linear boundary between the classes. For example supervised learning classification models such as SVM (Support Vector Machine) or MLP (Multi-Layer Perceptron). Of course, it is good to keep in mind that in order to create such models, and use supervised learning, we will have to feed not only the raw values, but also the labels indicating which record falls into which group. We will do this when we get to the classification chapter of the exercises.

To complete the current exercise, we will look at an example from practice. I assume you are all familiar with the .gif graphic file format? Do you know the main advantage of .gif files over other graphic formats (eg .jpg or .png)?

Other than the fact they can have multiple frames, and thus be animated, .gif files work with the so-called indexed colour space, which means that there is a set of pre-defined colours, and each pixel has a label that tells that pixel which colour it should be rendered in. This is very reminiscent of our data arrays and their labels, isn't it? This is not accidental. Indeed, when saving a .gif file, the software with which you create a gif version of the image uses a clustering algorithm to find out which are those few colours (it's up to you to specify how many there will be, i.e. k =?) around which all other colours in the image can be clustered. Essentially, we can use the K-means method in three-dimensional space. Why 3-dimensional? Because in general, the images in our computer are stored as a matrix containing the RGB (Red, Green, Blue) values for each pixel, so we have 3 channels of colour data.

We can demonstrate the action of K-means applied to an image to see this effect. Let's load one of the images we saved during the data acquisition exercise, when we used Pixabay's API. We are going to use the Pillow library (PIL) and the .open() method of its Image class:

```
image = Image.open('Resources/API/saved-images/1197800.jpg')
```

We can see what this image looks like by simply calling the variable that contains it, or by using the matplotlib .imshow() method:

```
plt.imshow(image)
```

Our next step is to convert the image to a Numpy array. We will store the result of this conversion in a new variable "np_image":

```
np_image = np.array(image)
```

We can make sure the result is what we wanted if we call the .shape method, which returns the shape of Numpy arrays. For our image, we expect to get an array that has as many rows as the height of the image (in pixels) and as many columns as the width of the image (in pixels), and the depth of the array should be equal to the number of colour channels, i.e. 3 (for GRB images):

```
print(np_image.shape)
(853, 1280, 3)
```

To make our work easier, we can normalize the values in the array using the already familiar formula (V.3-1).

Since the intensity values, that make up the colours, range from 0 to 255, which is due to their 8bit (2^8 = 256) colour representation. That means their span ($X_{max} - X_{min}$) is exactly 255, we just need to divide each value in our array by 255. We save the normalisation result in a new variable named "normalised_image_data":

```
normalised_image_data = np_image / 255.0
```

Our next step is to convert the 3-dimensional array to 2-dimensional one. We need to take all the rows of the image and line them up next to each other, for all three colour channels. I.e. for the 853 x 1280 pixels with 3 channels, we will get 3 rows (one for each channel) with 1091840 values (= 853 x 1280). We will save the result of the resizing in the "data" variable:

```
data = normalised_image_data.reshape(np_image.shape[0] *
                                     np_image.shape[1], 3)
```

We can check that we have achieved the desired result by calling the .shape() method on the resized array again:

```
print(data.shape)
(1091840, 3)
```

Now is the time to determine how many groups we want to distribute the colours into. I.e. effectively how many colours we will use to map all of the colours from the original image. You can experiment with this value, but for starters, let's go with 3:

```
k = 3
```

We create the model by specifying that the number of clusters k will be equal to 3. After which we let it "learn" using the familiar .fit() method:

```
kmeans = KMeans(n_clusters=k)
kmeans.fit(data)
```

We use .predict() with the normalised and resized data to get the labels for each pixel (i.e. which of the 3 colours/centres it is assigned to):

```
color_labels = kmeans.predict(data)
```

Then we need to replace the values of each pixel with the value of the corresponding group average colour (i.e. the centre value). For this purpose, we can take the "color_labels" variable, which contains the predicted labels, and pass it to be used as indexes for the three

colours (found in the .cluster_centers_ property of the trained model). We save the result of this action in a new variable "new_colors":

```
new_colors = kmeans.cluster_centers_[color_labels]
```

Now we need to restore the original shape of the array. From 1091840 x 3 we need to make it 853 x 1280 x 3 again. So, we will use .reshape() method again, specifying the size of the original image (np_image.shape). We will store the result of the resizing in a new variable "colour_compressed_image":

```
colour_compressed_image = new_colors.reshape(np_image.shape)
```

All that remains is to convert the Numpy array back into a Pillow image:

```
colour_compressed_image =
Image.fromarray(np.uint8(colour_compressed_image*255))
```

If we now call the "colour_compressed_image" variable we should get the image with colours matched to the determined 3-centre colours:

```
plt.imshow(colour_compressed_image)
```

We can visualise the original and the processed image in a figure with two sub-plots, so we can more easily compare the two:

```
images_fig, (im1, im2) = plt.subplots(1, 2)
fig.subplots_adjust(wspace=0.05)

im1.imshow(image)
im1.set_title('Original Image')
im1.set_xticks([])
```

```
im1.set_yticks([])

im2.imshow(colour_compressed_image)
im2.set_title(f'{k}-colours Image')
im2.set_xticks([])
im2.set_yticks([])
```

Chapter VIII.
Detection & Classification using Artificial Neural Networks

Very often, the internal decision-making logic of a control system will need a way to classify certain system states or objects.

Unlike the k-means clusterisation algorithm, which from Machine Learning standpoint uses **unsupervised learning** to find the internal relationships between datapoints, Artificial Neuron Networks (ANNs) require the so-called **supervised learning**.

The **unsupervised learning** approach means we don't need to "supervise" the learning process by preparing learning examples or providing any guidance. We don't need any understanding about the classes (groups, clusters), only their count (how many of them there are).

With **supervised learning** we need examples. We need a "supervisor" who understands what is it in the data that categorises each datapoint as one class or another. Someone (or something) that can label examples of datapoints with the correct class (or group) which we can then feed to the model, so it will learn what features of those data points have anything to do with their class and to what extent.

The most common neural network architecture is feed-forward with back-propagation of the error (we feed the input data at the beginning of the network, and it spreads forward from one layer to the next, until it reaches the end of the network). During training the output is compared with a predefined (expected) output. That predefined output is the result of the supervisor "labelling" some number of samples with the correct class. These samples comprise the training dataset, that must be prepared before feeding them to the network for training.

The difference between the expected output and the actual output is the error. The size of the error tells the network how close/far it is from how it is expected to work. Then, an optimisation algorithm is used on the error function, to find such parameters for each neuron, so that the next iteration the output will be closer to what is expected. Finally, when all training samples have been used and the parameters of the network are tuned so that all training samples, when fed through the network, produce the expected result – the network is ready to start working with new (unseen) examples, which were not part of the training set.

VIII.1. Support Vector Machine

Support Vector Machines (SVMs) are type of Artificial Neural Network (ANN) which are very often used for classification or regression problems. The main goal of the SVM is to find a plane (or hyperplane, when considering multi-dimensional problems) that best divides the feature space, that contains the various classes we are interested in.

Let’s again use the example with the “normal” and “fraudulent” bank transactions. Let’s assume our task is to train a system to classify them (this time based on known, already classified, historic examples). For example, the two feature dimensions of the data could be the country and the currency. Those would be our X_1 and X_2 coordinates (Figure 78).

What the SVM will do is attempt to find such a line (plane or hyper-plane) that divides the space that the known (training) examples occupy, such that the space between the closest point is at its maximum.

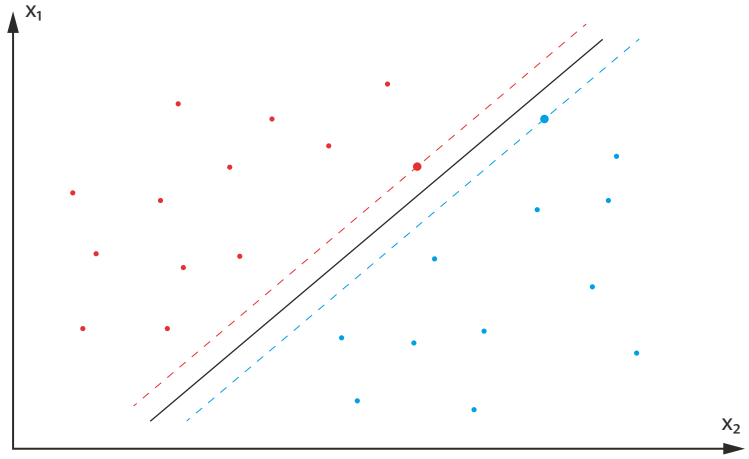


Figure 78. Example of a two-dimensional feature space, divided with a classification boundary with some margin around it. The coordinates of the larger red and blue dots (lying on the margins) would be the supporting vector for the margin of the respective class

The distance between the diving line or plane to the nearest point would be the classification margin, and the points that are closest to the boundary are the supporting vector (as in the vector of coordinates, that support the margin).

Exercise VIII.1-1: Use Python to build SVM classification algorithm.

We can approach the moons and circles dataset, which we used in the clusterisation Exercise VII.1-1, and this time try to classify the points either belonging to one or the other of the moons/circles, by using an SVM.

Let's import the packages we'll need. Other than MatplotLib, which we will use for visualisation purposes, almost everything else we need we'll get from SciKit Learn (SKlearn) [14]. the datasets package (imported as "ds") will allow us to generate our own synthetic data, same as we did with the clusterisation exercise. The train_test_split function will allow us to create training and testing datasets by splitting the synthetically generated data. Confusion Matrix is a way to represent the confusion of the network when training (which class of objects was confused with which other class). The ConfusionMatrixDisplay allows us to visualise the confusion matrix. DecisionBoundaryDisplay will allow us to see the classification decision boundaries. And finally, the actual neural architecture, the SVC (Support Vector Classification) class part of the SVM package. The reason why there is a distinction in the SVM package is that as we mentioned Support Vector Machines can be used for regression, so there is another class SVR that implements that particular architecture. If you want, you can try to use that on the Energy Consumption problem from previous exercises.

```
import matplotlib.pylab as plt
from sklearn import datasets as ds
from sklearn.model_selection import train_test_split
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.svm import SVC  # Support Vector Classification
```

Let's recreate the dataset we used in the clusterisation exercise. We will use the same parameters for the moons and circles. We store the data points in the respective "moons_data" and "circles_data" variables, and we store the membership labels (actual classes) in the respective "moons_labels" and "circles_labels" variables:

```
moons_data, moons_labels = ds.make_moons(300, noise=0.07,
                                         random_state=42)
circles_data, circles_labels = ds.make_circles((500, 200), factor=0.3,
                               shuffle=True, noise=0.08, random_state=42)
```

As we discussed, unlike clusterisation, and generally unsupervised machine learning, where we can use all the data, and let the algorithm find patterns and relationships in it, with supervised learning models we need to split the data into training and testing datasets.

So first, we take the "moons_data" and the "moons_labels" and pass them as input to the train_test_split() function. The result is four variables, the first two contain the training and testing values, and the other two contain the training and testing labels. The splitting rule we use is 40% of the samples to be used for testing (60% for training), and of course, we set the random number generator's state to a number of our choice, so that we have repeatability.

Then we do the same for the circles as well:

```
moons_data_train, moons_data_test, moons_labels_train,
moons_labels_test = train_test_split(moons_data, moons_labels,
                                     test_size=0.4, random_state=42)

circles_data_train, circles_data_test, circles_labels_train,
circles_labels_test = train_test_split(circles_data, circles_labels,
                                       test_size=0.4, random_state=42)
```

Now that we have the training and testing data split, it is time to create the SVM. We are going to call the SVC class that we imported in the beginning and pass in the parameters of the network. There are few important parameters that we need to consider.

First of all, the kernel of the model. By default, the sklearn SVC implementation uses Radial Bias Function (RBF) as a kernel [15]. We can choose other options as well, for example a sigmoid or a linear function, but for this example we will stick to the default, but feel free to try the others as well, and see the differences in the results.

The next is the gamma parameter, which defines how far the influence of a single training example reaches. A low value would mean "longer" reach, and a high value would mean "closer" reach. The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors.

The C parameter trades off correct classification of training examples against maximization of the decision function's margin. For larger values of C, a smaller margin will be accepted if the decision function is better at classifying all training points correctly. A lower C will encourage a larger margin, therefore a simpler decision function, at the cost of training accuracy. In other words, C behaves as a regularization parameter in the SVM [16]. By default, C will be set to 1, so we will stick to that value as well.

```
# training a linear SVM classifier
moons_svm = SVC(kernel='rbf', gamma=2, C=1)
```

Once the network is created we can start the training process by invoking the well familiar .fit() method on it, and passing as an input the training data and the training labels:

```
moons_svm.fit(moons_data_train, moons_labels_train)
```

When the network finishes learning we can use the .predict() method to test how it works. If we pass in the testing portion of our dataset the network should classify those points (which it has not seen during training). The result of the classification we can save in a new variable "moons_svm_predicted_labels":

```
moons_svm_predicted_labels = moons_svm.predict(moons_data_test)
```

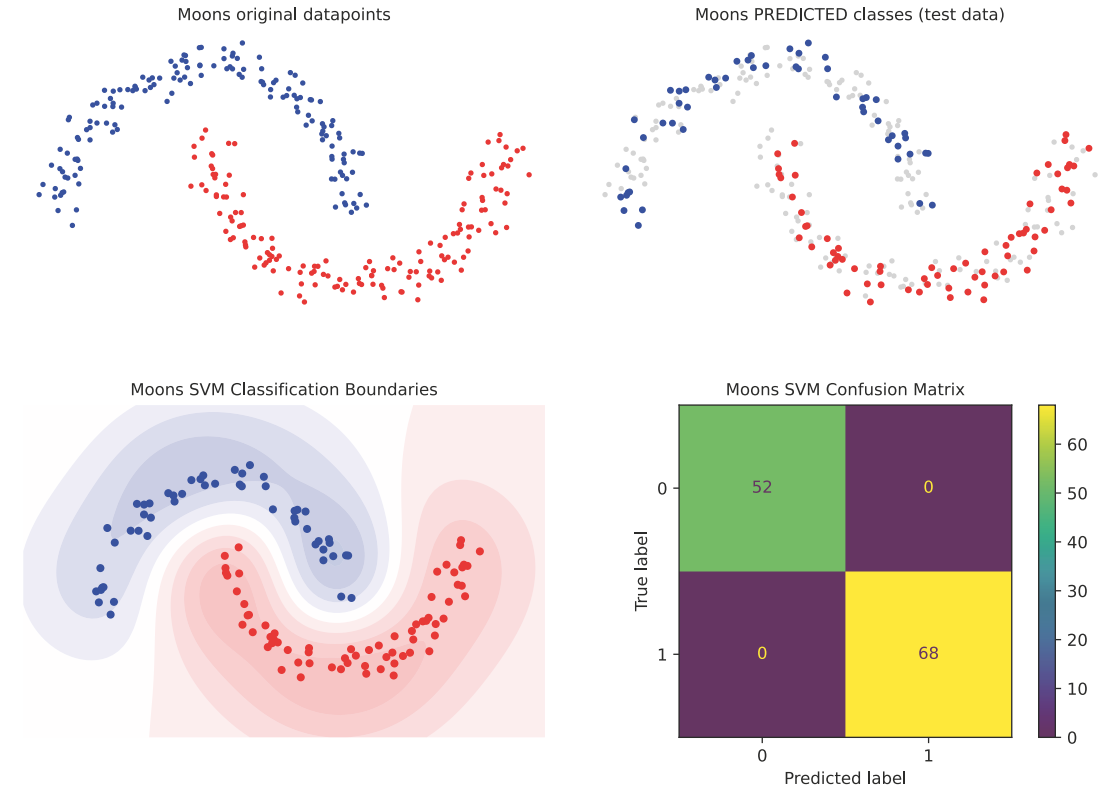
We can get an estimation of the network's accuracy (the percentage of the total test samples which were correctly classified) by calling the .score() method of the trained model. We pass as input the test dataset as well as the test labels. The result we can store in a new variable named "moons_svm_score". Let's print it out:

```
# model accuracy for X_test
moons_svm_score = moons_svm.score(moons_data_test, moons_labels_test)
print(moons_svm_score)
1.0
```

The score of 1.0 (100%) means that the model classified all test data points correctly. We can see that visually if we construct a matplotlib figure with the test data predictions (in blue and red) on top of the training data (in grey), the points do in fact follow the shape of the two moons (top-right subplot). We can see the original data in the top-left subplot, and the classification boundaries (learnt during training) in the bottom-left subplot.
Finally, we also plot the confusion matrix in the bottom-right subplot. We can see that out of the 120 training samples (60% of the total 200 points) 52 were correctly classified as class 0 (the first moon) and 68 were correctly classified as class 1 (the second moon). Thus, we have 0 false classifications on any of the two classes:

```
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2)
fig.subplots_adjust(wspace=0.05)
fig.set_size_inches(12, 8)

ax1.scatter(moons_data[:, 0], moons_data[:, 1],
            s=5, c=moons_labels, cmap='bwr');
ax1.set_title('Moons original datapoints', fontsize=10)
ax1.set_aspect('equal', 'box')
ax1.axis('off')

ax2.scatter(moons_data[:, 0], moons_data[:, 1], s=5, c='lightgrey');
ax2.scatter(moons_data_test[:, 0], moons_data_test[:, 1],
            s=10, c=moons_svm_predicted_labels, cmap='bwr');
ax2.set_title('Moons PREDICTED classes (test data)', fontsize=10)
ax2.set_aspect('equal', 'box')
ax2.axis('off')

DecisionBoundaryDisplay.from_estimator(moons_svm, moons_data,
cmap='bwr', alpha=0.3, ax=ax3, eps=0.5)
ax3.scatter(moons_data_test[:, 0], moons_data_test[:, 1],
            s=15, c=moons_svm_predicted_labels, cmap='bwr');
ax3.set_title('Moons SVM Classification Boundaries', fontsize=10)
ax3.set_aspect('equal', 'box')
ax3.axis('off')
```

```
ConfusionMatrixDisplay.from_estimator(moons_svm, moons_data_test,
moons_labels_test, ax=ax4)
ax4.set_title('Moons SVM Confusion Matrix', fontsize=10)
ax4.set_aspect('equal', 'box')
```



We are done with the moons, but if you want – try to increase the noise during data generation and see if you still get a perfect result. Now, let's move to the second dataset – the circles. Let's go through the same process for them as well. Create a new SVM model, which we will store with the name "circles_svm". This time we will not bother to define the kernel and the *C* variable, as we using the default values for them. We just need to pass in the value for the gamma parameter. Then we let the model train with the training dataset we prepared in the beginning:

```
# training a linear SVM classifier
circles_svm = SVC(gamma=2)
circles_svm.fit(circles_data_train, circles_labels_train)
```

Once the model has finished training, we can again use the .predict() method to make it classify the testing portion of the data:

```
circles_svm_predicted_labels = circles_svm.predict(circles_data_test)
```

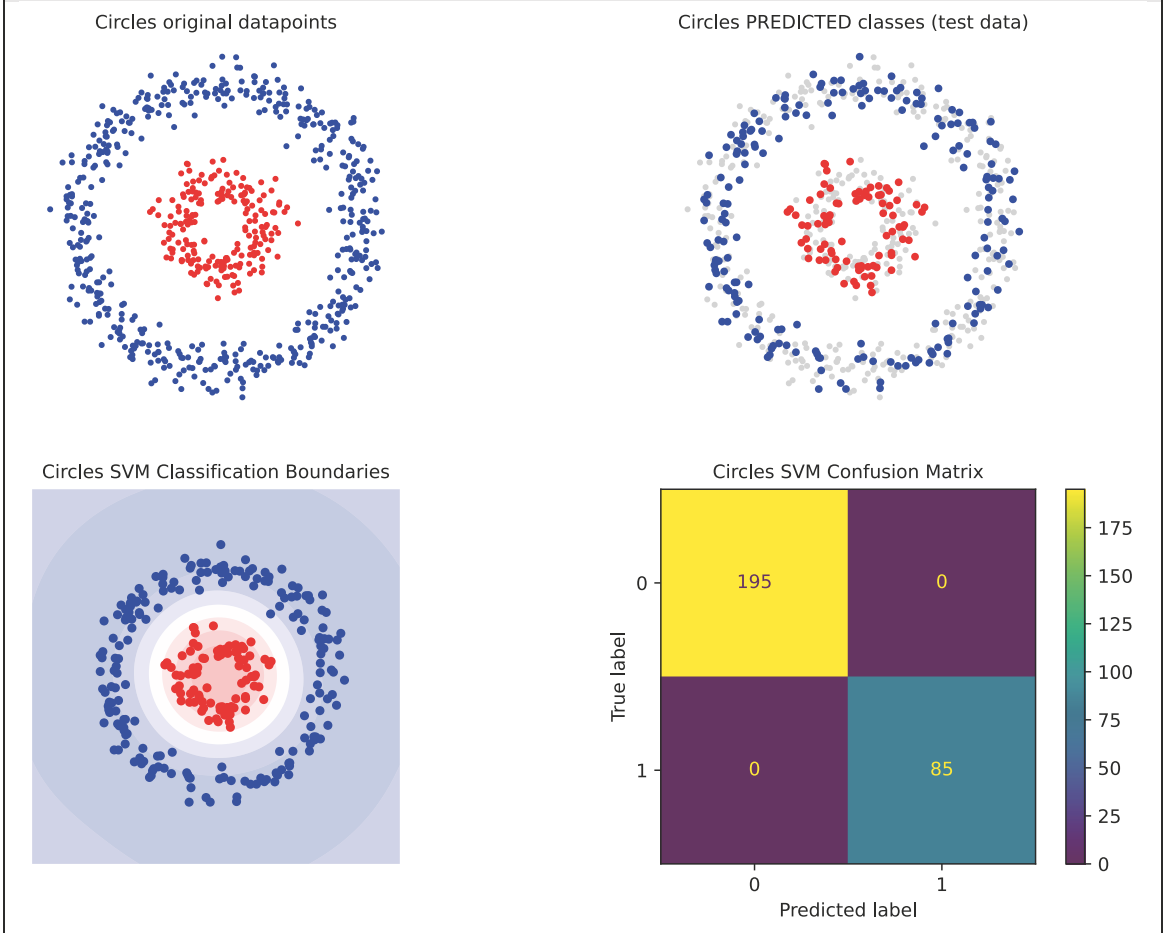
With this synthetic dataset, with its limited volume of data, as well as the fact that we are intentionally generating it to be relatively clean, we do not expect the accuracy to really drop, but just in case, let's evaluate it and print it out, like we did for the moon's model:

```
# model accuracy for X_test
circles_svm_score = circles_svm.score(circles_data_test,
                                      circles_labels_test)
print(circles_svm_score)
```

```
1.0
```

And finally, let us visualise it the same way we did with the moons:

```
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2)
fig.subplots_adjust(wspace=0.05)
fig.set_size_inches(12, 8)
ax1.scatter(circles_data[:, 0], circles_data[:, 1], s=5,
            c=circles_labels, cmap='bwr');
ax1.set_title('Circles original datapoints', fontsize=10)
ax1.set_aspect('equal', 'box')
ax1.axis('off')

ax2.scatter(circles_data[:, 0], circles_data[:, 1], s=5,
            c='lightgrey');
ax2.scatter(circles_data_test[:, 0], circles_data_test[:, 1], s=10,
c=circles_svm_predicted_labels, cmap='bwr');
ax2.set_title('Circles PREDICTED classes (test data)', fontsize=10)
ax2.set_aspect('equal', 'box')
ax2.axis('off')

DecisionBoundaryDisplay.from_estimator(circles_svm, circles_data,
                                       cmap='bwr', alpha=0.3, ax=ax3, eps=0.5)
ax3.scatter(circles_data_test[:, 0], circles_data_test[:, 1], s=15,
                           c=circles_svm_predicted_labels, cmap='bwr');
ax3.set_title('Circles SVM Classification Boundaries', fontsize=10)
ax3.set_aspect('equal', 'box')
ax3.axis('off')
ConfusionMatrixDisplay.from_estimator(circles_svm, circles_data_test,
                                      circles_labels_test, ax=ax4)
ax4.set_title('Circles SVM Confusion Matrix', fontsize=10)
ax4.set_aspect('equal', 'box')
```



VIII.2. Multilayer perceptron

Multilayer perceptron is another artificial neural network architechture, which is good at classification tasks. It also has good generalisation properties and can be applied to wide variety of classification problems. Perceptrons, in general, have one caveat, and that is they can only generate a linear boundary between the classes. Considering the shapes our data forms (moons and circles) it is obvious, that a single line will not be able to separate the space in a way that makes them really apart. Especially so for the circles.

But, if we stack enough perceptrons together to form multiple layers of them (hence the name MLP), where every layer is responsible for a small linear part of a larger curved classification boundary, then we can employ the MLP for non-linear classification. Let's try.

Exercise VIII.2-1: Use Python to build and train an MLP classification model.

We will again use the moons and circles dataset, which we used in the clusterisation Exercise VII.1-1 and in the SVM Exercise VIII.1-1, but this time we will apply an MLP model to it.

Let's import the packages we'll need. We will again use MatplotLib for visualisation purposes, everything else we need we'll get from SciKit Learn [14]. The datasets package (imported as "ds") will allow us to generate the same synthetic data we used before. The train_test_split function will allow us to create training and testing datasets by splitting the synthetically generated data. We will again use ConfusionMatrixDisplay and DecisionBoundaryDisplay to visualise the performance of the neural model. And finally, the actual neural architecture, the MLPClassifier.

```
import matplotlib.pylab as plt
from sklearn import datasets as ds
from sklearn.model_selection import train_test_split
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.inspection import DecisionBoundaryDisplay
from sklearn.neural_network import MLPClassifier
```

Let's recreate the moons and circles data one more time. Again, we will use the same parameters for the moons and circles as we did in previous exercises. We store the data points in the respective "moons_data" and "circles_data" variables, and we store the membership labels (actual classes) in the respective "moons_labels" and "circles_labels" variables:

```
moons_data, moons_labels = ds.make_moons(300, noise=0.07,
random_state=42)
circles_data, circles_labels = ds.make_circles((500, 200), factor=0.3,
shuffle=True, noise=0.08, random_state=42)
```

There is no difference in the process of splitting the data into training and testing subsets, and we again use a 40/60 split, and the same value for the random number generator seed:

```
moons_data_train, moons_data_test, moons_labels_train,
moons_labels_test = train_test_split(moons_data, moons_labels,
                                     test_size=0.4, random_state=42)
circles_data_train, circles_data_test, circles_labels_train,
circles_labels_test = train_test_split(circles_data, circles_labels,
                                       test_size=0.4, random_state=42)
```

Let's create the model. We use the MLPClassifier class and define the parameters of the network. All the parameters we will use have default values, and we will make use of most of them, but for some we might want to experiment with.

For example, we might want to try out different hidden layer composition, different value for alpha, different number of iteration limit (for the optimisation solver to reach convergence), different activation function for the perceptrons, and so on. We will store the model structure in the variable "moons_mlp" and let it train, using the .fit() method, with the supplied training data and labels:

```
moons_mlp = MLPClassifier(hidden_layer_sizes=(100,), alpha=0.001,
max_iter=2000, solver='adam', activation='relu', random_state=1)
moons_mlp.fit(moons_data_train, moons_labels_train)  # Train the model
```

Once the model is trained we can use the .predict() method to generate predictions for the testing portion of the dataset:

```
# Use the trained MLP to predict the classes of the test set
moons_mlp_predicted_labels = moons_mlp.predict(moons_data_test)
```

We can also, just like we did before, calculate the accuracy score for the trained model, by passing in the test dataset and the test labels, to see what percentage of the samples will get classified correctly. Let's print out the result:

```
moons_mlp_score = moons_mlp.score(moons_data_test, moons_labels_test)
print(moons_mlp_score)
1.0
```

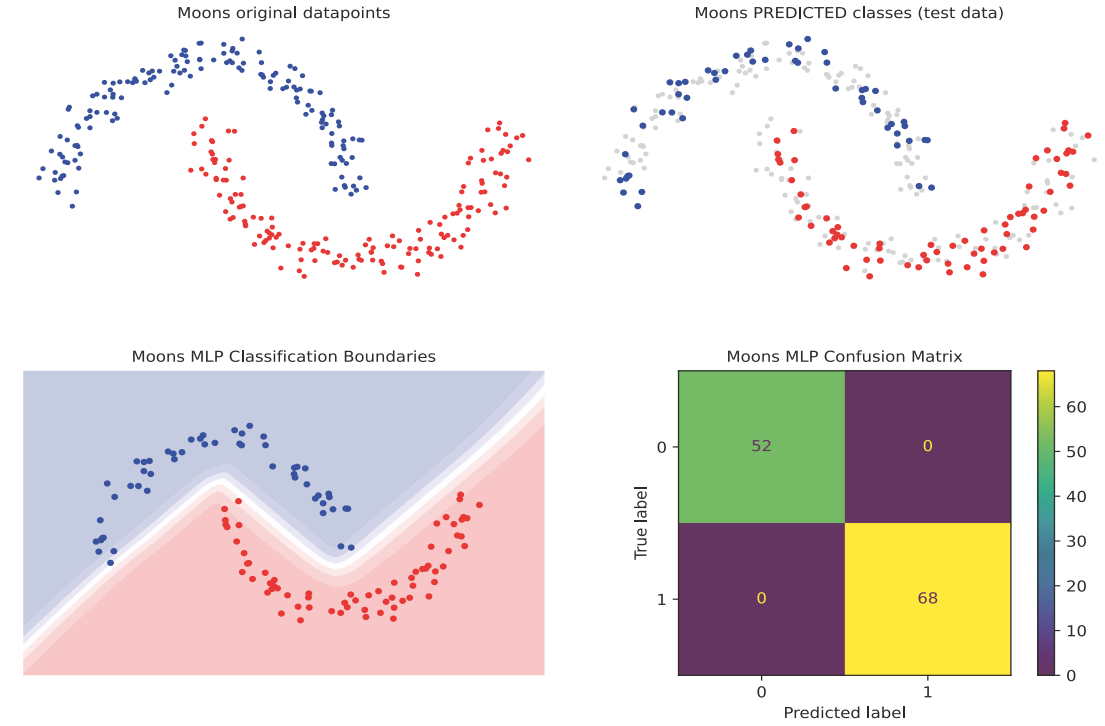
Once again, we get a perfect score. Of course, if we have let the model with its default parameters it might have not achieved that, and I encourage you to try out different values.

Let us visualise the results. We will use a 2 by 2 grid of subplots. In the top-left we will plot the original data with the actual class labels in blue and red ('bwr' colour map). In the top-right subplot we will have the original points in lightgrey colour, and then on top of them the predicted classification from the model (again in blue and red).

On the bottom row we will display the classification boundary (on top of the original data) and then next to it, in the last subplot, the confusion matrix:

```
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2)
fig.subplots_adjust(wspace=0.05)
fig.set_size_inches(12, 8)

ax1.scatter(moons_data[:, 0], moons_data[:, 1],
            s=5, c=moons_labels, cmap='bwr');
ax1.set_title('Moons original datapoints', fontsize=10)
ax1.set_aspect('equal', 'box')
ax1.axis('off')

ax2.scatter(moons_data[:, 0], moons_data[:, 1], s=5, c='lightgrey');
ax2.scatter(moons_data_test[:, 0], moons_data_test[:, 1], s=10,
c=moons_mlp_predicted_labels, cmap='bwr');
ax2.set_title('Moons PREDICTED classes (test data)', fontsize=10)
ax2.set_aspect('equal', 'box')
ax2.axis('off')
```

```
DecisionBoundaryDisplay.from_estimator(moons_mlp, moons_data,
cmap='bwr', alpha=0.3, ax=ax3, eps=0.5)
ax3.scatter(moons_data_test[:, 0], moons_data_test[:, 1], s=10,
c=moons_mlp_predicted_labels, cmap='bwr');
ax3.set_title('Moons MLP Classification Boundaries', fontsize=10)
ax3.set_aspect('equal', 'box')
ax3.axis('off')

ConfusionMatrixDisplay.from_estimator(moons_mlp, moons_data_test,
moons_labels_test, ax=ax4)
ax4.set_title('Moons MLP Confusion Matrix', fontsize=10)
ax4.set_aspect('equal', 'box')
```



It is easy to compare the classification boundaries of the MLP model to the boundaries of the SVM model we did in the previous exercise. We can see how the MLP has built the boundary from somewhat straight sections, unlike the smoother version the SVM did.

Let's finish the exercise by quickly doing the same steps for the circles dataset. We are going to use exactly the same parameters, but store the model in a new object named "circles_mlp". We then use the .fit() method to train it and then get predictions using the .predict() method:

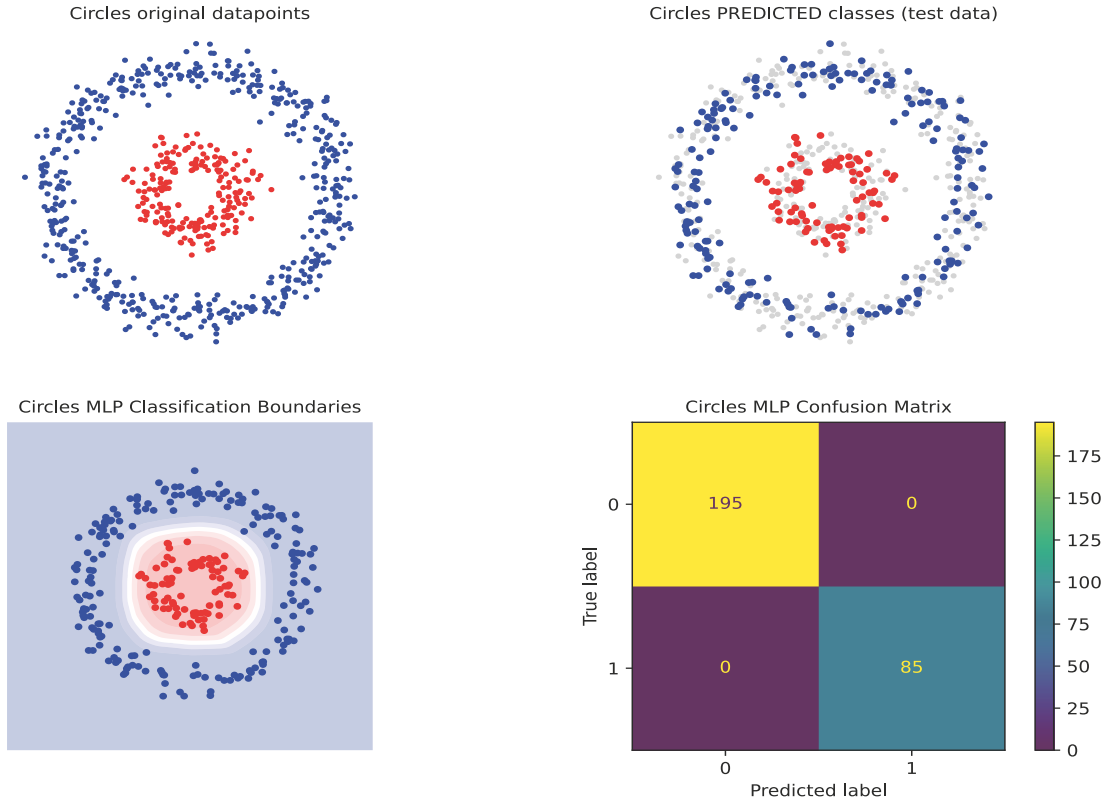
```
circles_mlp = MLPClassifier(hidden_layer_sizes=(100,), max_iter=1000,
solver='adam', activation='relu', random_state=1)
circles_mlp.fit(circles_data_train, circles_labels_train)
circles_mlp_predicted_labels = circles_mlp.predict(circles_data_test)
```

Let's see what is the accuracy score of this model:

```
circles_mlp_score = circles_mlp.score(circles_data_test,
                                      circles_labels_test)
print(circles_mlp_score)
1.0
```

Of course, not much surprise with the 100% accuracy, as the data is very clear. Let's visualise the results and see the difference in the classification boundaries:

```
fig, [[ax1, ax2], [ax3, ax4]] = plt.subplots(2, 2)
fig.subplots_adjust(wspace=0.05)
fig.set_size_inches(12, 8)

ax1.scatter(circles_data[:, 0], circles_data[:, 1], s=5,
c=circles_labels, cmap='bwr');
ax1.set_title('Circles original datapoints', fontsize=10)
ax1.set_aspect('equal', 'box')
ax1.axis('off')

ax2.scatter(circles_data[:, 0], circles_data[:, 1], s=5,
c='lightgrey');
ax2.scatter(circles_data_test[:, 0], circles_data_test[:, 1], s=10,
c=circles_mlp_predicted_labels, cmap='bwr');
ax2.set_title('Circles PREDICTED classes (test data)', fontsize=10)
ax2.set_aspect('equal', 'box')
ax2.axis('off')

DecisionBoundaryDisplay.from_estimator(circles_mlp, circles_data,
cmap='bwr', alpha=0.3, ax=ax3, eps=0.5)
ax3.scatter(circles_data_test[:, 0], circles_data_test[:, 1], s=10,
c=circles_mlp_predicted_labels, cmap='bwr');
ax3.set_title('Circles MLP Classification Boundaries', fontsize=10)
ax3.set_aspect('equal', 'box')
ax3.axis('off')

ConfusionMatrixDisplay.from_estimator(circles_mlp, circles_data_test,
circles_labels_test, ax=ax4)
ax4.set_title('Circles MLP Confusion Matrix', fontsize=10)
ax4.set_aspect('equal', 'box')
```



VIII.3. Object classification and detection with convolutional neural networks

Neural networks are often used in the field of computer vision for **recognition (classification)** and **finding (localisation)** of objects in images. These processes can be applied both to a still image (photo) and to a video, which is itself a series of still frames. Of course, when working with video, one must consider the processing time of a single frame and how it fits within the video speed (frames per second). If the recognition cannot be performed within the time that one frame is replaced by another from the video source, this may lead to undesirable consequences.

These types of neural networks have a huge number of neurons, and their training requires a significant number of calculations, which makes it very resource-consuming. This is based on the fact that images are made up of pixels, and every single pixel in the image needs to be fed as input to the network. This combined with the depth of the network (the number of hidden layers and all the neurons in them) results in the accumulation of a huge amount of optimization parameters during training.

Furthermore, the fact that images are some of the heaviest files in terms of their size in computer memory means that loading the training array into memory also requires substantial resources. At least thousands and often millions or billions of images are used during training. This means that it is practically impossible to keep the entire training set in memory. For this, the training uses parts (batches) containing a sample of the total amount of images, thus the entire training array is used on small combinations of records.

Although the training process is slow and resource-intensive, once trained the network can perform the necessary classifications on the input image relatively quickly, and in practice run in real time even from a relatively low-power devices. The time to perform a classification depends on the architecture of the trained network and the size of the input image.

The training of convolutional neural networks is supervised, i.e. is performed using pre-labelled data. In the computer vision sense, this means that every image must be accompanied with an additional package of information containing which classes of objects are present in the image, and when it comes to finding (localising) objects - that information should also include the area where they are located in the frame, in the form of pixel coordinates). Below in the exercise we will demonstrate the process of labelling an image.

During training, the neural network tries to "learn" what transformations it needs to make on the images, to find the most representative features to associate with the given classes of objects. This process is called feature extraction.

For example, let's imagine an intelligent computer vision based quality control system in a production assembly line. During training, it should be fed a large number of pictures of various defects that we want it to learn to detect (many pictures for each class of defect). If one of the defect classes is a crack in a given casting, then the network must learn to recognize what a crack looks like. What are the features, in terms of the placement and relationship of the pixels in the area of the image where the crack is contained. The training image set should contain pictures of many different cracks (for our particular part), preferably in different lighting conditions and different angles, unless these are conditions we can control in the operational environment.

As mentioned above, in order to extract the features of the objects (classes), during training, the network applies different transformations (filters) to the image, through which it can obtain different effects in the image. Such transformations can be inverting the image from colour to grayscale or even black and white (bitmap), can be related to changing the size or rotation of the frame, changing the contrast or colour of the image, finding edges, applying blur, sharp, etc.

This type of image filtering (changing the values of the pixels in the image) is related to the application of the mathematical operation convolution. Convolution, in general terms, is the multiplication of the images of two functions. Mathematically, it can be represented as:

$$s(x) = f(x) * g(x) = \int_{-\infty}^{\infty} f(x-\tau)g(\tau)d\tau \qquad \textit{(VIII.3-1)}$$

where the convolution operation is denoted by the symbol (*). It is performed between two functions of x, one is $f(x)$ and the other is $g(x)$. The result of the convolution is a new function $s(x)$.

In the special case of convolutional neural networks, the convolution is performed between the input image of the network (represented as a matrix of pixel values) and a filter (kernel) that contains different weights to be applied to a given pixel (and its surrounding area).

The purpose of this exercise is to demonstrate this process in action. Within the exercise, we will also demonstrate how to prepare images for supervised training of convolutional neural networks, and finally how to use an already trained network for a specific application.

Exercise VIII.3-1: Use Python to convolutionally apply filter on an image.

Before we get to use a convolutional neural network, we should first understand well how the convolutional kernels (filters) work.

Let's start by importing some libraries. The most important library we will be working with when dealing with images for computer vision applications is CV2 (Computer Vision 2). It will allow us to open/capture images and video from files and video devices (cameras) and manipulate them. The Math library we will need mostly for the square root function. Numpy is the standard library when working with arrays (or matrices), and images are inherently arrays of pixels. Also, NumPy has a built-in convolution function which we can use. SciPy is a library with scientific functions, and also has a convolution implementation we can try. And finally, we need Matplotlib for visualisation purposes:

```
import cv2
import math
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from matplotlib.patches import Rectangle
```

For the purposes of this exercise, we can think of the convolution as a moving weighted average between two arrays. One array, let's call it base, is our original data that we want to

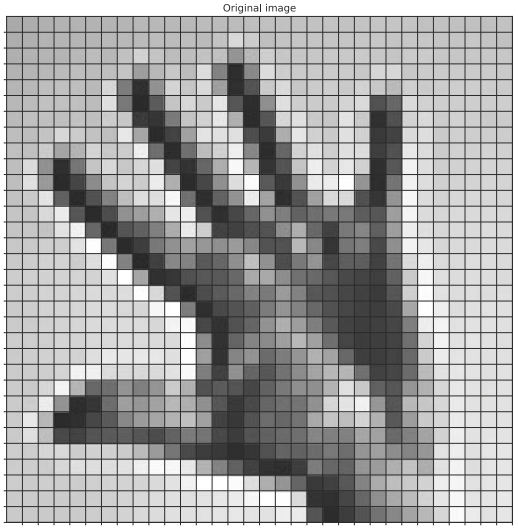
process through convolution, and the other array contains the weights that we'll use for the weighted moving average. Let's load our base array first. For now, this will be a small sample image of a 32 x 32 pixels of the palm of a hand. To make our task easier, we will use a grayscale image, so we only work with a 2-dimensional array, instead of 3-dimensional.

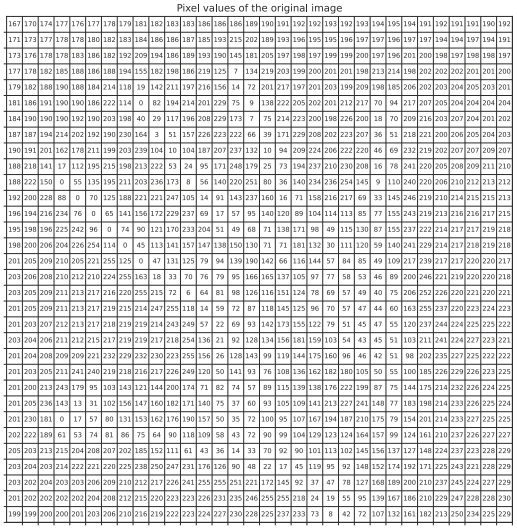
We use the .imread() function of CV2 and pass in the path to our image file, as well as the necessary setting for converting the image to grayscale when reading it in. The result we store in a variable named "hand_image_bw":

```
hand_image_bw = cv2.imread('Resources/CNN/hand-shot-32.png',
cv2.IMREAD_GRAYSCALE)
```

We can visualise this image on the screen. Let's show it both in its original form (the left subplot) and as a matrix of the values of the different pixels (the right subplot):

```
# Create the figure object and an array ot two subplots (1 row and 2
columns)
fig, [ax1, ax2] = plt.subplots(1, 2, sharex=True, sharey=True)

# Use the first subplot to show the original image
ax1.imshow(hand_image_bw, cmap='gray')
# Move the grid ticks to sit in between the pixes
# instead in the middle of the pixels
# Hide tick labels on the x axis
ax1.set_xticks([x-0.5 for x in list(range(1,len(hand_image_bw)+1))],
range(1,len(hand_image_bw)+1), visible=False)
# Hide tick labels on the y axis
ax1.set_yticks([x-0.5 for x in list(range(1,len(hand_image_bw)+1))],
range(1,len(hand_image_bw)+1), visible=False)
#Add grid lines on all major ticks
ax1.grid(which="major", color='black', linestyle='-', linewidth=0.5)
#Set the title of the subplot
ax1.set_title('Original image')

# Use the second subplot to draw an empty (white) array
# with the same size as the original image
ax2.imshow(np.zeros_like(hand_image_bw), cmap='Greys')
# Go through all the pixel values and show them as text
# on top of the empty image
for i in range(hand_image_bw.shape[0]):
    for j in range(hand_image_bw.shape[1]):
        text = ax2.text(j, i, hand_image_bw[i, j], ha="center",
                        va="center", fontsize=8, color="black")
# Move the grid ticks to sit in between the pixes
# instead in the middle of the pixels
ax2.set_xticks([x-0.5 for x in list(range(1,len(hand_image_bw)+1))],
range(1,len(hand_image_bw)+1), visible=False)  # Hide X tick labels
ax2.set_yticks([x-0.5 for x in list(range(1,len(hand_image_bw)+1))],
range(1,len(hand_image_bw)+1), visible=False)  # Hide Y tick labels
ax2.grid(which="major", color='black', linestyle='-', linewidth=0.5)
#Add grid lines on all major ticks
ax2.set_title('Pixel values of the original image')  #Set the title of
the subplot

# Make the figure take the entire width and fix the height to width
ratio, making sure it is square
fig.set_size_inches(24, 12)
```





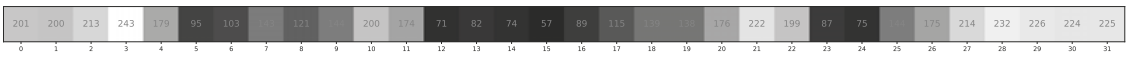
Before proceeding with applying filters to the image, let's understand the operation itself by first applying it to a small one-dimensional region of the original image. To do this, we'll take one row of pixels, for example the 24th row (index 23 of the base image array), and write it into a new variable we'll name "image_strip".

Then, because MatplotLib's image/array visualisation function named .imshow() requires 2-dimensional arrays, and so does the convolution function from the Numpy library, we'll need to extend the dimensionality of our vector (row) of pixel values by adding a second dimension. We'll do this with the .expand_dims() function from the Numpy library. As input we give the array that we want to expand and the direction (axis):

```
image_strip = hand_image_bw[23]
image_strip_exp = np.expand_dims(image_strip, axis=0)
```

Let's visualise this one row along with its pixel (colour intensity) values:

```
plt.imshow(image_strip_exp, cmap='gray')
# Go through all the pixel values and show them as text
# on top of the pixels of the image
for i in range(len(image_strip_exp[0])):
    text = plt.text(i, 0, image_strip_exp[0][i], ha="center",
va="center", fontsize=14, color="grey")
plt.xticks(range(32))
plt.yticks([])
plt.rcParams["figure.figsize"] = (32,1)
```



Our next step is to define the filter kernel with which we will perform the convolution. To begin with, let's start with the most elementary kernel, where all the weights are the same and their sum is equal to 1. One feature that we need to consider when talking about convolutions, in the sense of filter kernels applied to an image, is that the kernels must have a central value and be symmetrical in every dimension. I.e., If we are talking about a one-dimensional core, it can be of 3, 5, 7, etc. values. Thus, we have one central value and a correspondingly equal number of values from the centre outwards. If we are talking about a two-dimensional core, it should be a square of size 3x3, 5x5, 7x7, etc.

The reason is that we use the kernel to process one particular pixel, i.e. we need to be able to centre the kernel on that pixel, which we won't be able to do with a kernel of an even number of values.

For our example 1-dimensional kernel, we can use the smallest possible case of 3 values:

```
kernel_1d = np.array([1/3, 1/3, 1/3])
```

Before we use a built-in function to convolute the two arrays, let's apply our kernel to a few pixels manually, just to see how it actually works, and then we can also compare the result with the built-in function.

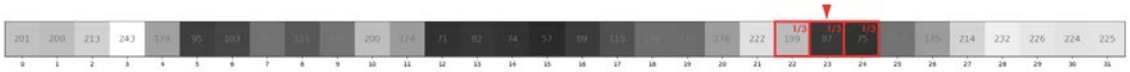
Let's choose which pixels to apply the test kernel to. For example, let's take the 23rd, 24th, and 25th pixels (with array indices 22, 23, and 24, respectively). They (for the sample image) have values of 199, 87 and 75.

Let's start with the first pixel, with an index of 22 and a value of 199. We apply the kernel to it so that the middle value overlaps it, like so:

We multiply the weights in the kernel by the values of the corresponding pixels below them, we then sum the results of this multiplication, and divide by the sum of the weights. As mentioned at the beginning, this results in a weighted average. In effect, what we are doing is replacing the pixel value with a new one that includes the influence of its neighbouring pixels as well. In this case, using the same weights for all the pixels involved in the calculation, we make an average blur (mean blur). We can draw an analogy with the moving average and exponential methods, which work similarly. Essentially, the result is a weighted smoothed array.

So, the new value for that pixel will be: $\frac{\frac{1}{3}222+\frac{1}{3}199+\frac{1}{3}87}{\frac{1}{3}+\frac{1}{3}+\frac{1}{3}} = 169.33$

Of course, we can't use a decimal value for the pixel intensity, so we'll take the integer part of it, namely: **169**. We follow the same procedure with the next pixel in the row. I hope you now see how that process is very similar to a weighted moving average. We drag the kernel one pixel to the right and repeat the operation:



We get: $\frac{\frac{1}{3}199+\frac{1}{3}87+\frac{1}{3}75}{\frac{1}{3}+\frac{1}{3}+\frac{1}{3}} = 120.33$ and therefore the new value for that pixel will be **120**.

Then we drag another pixel to the right and again we calculate its weighted value:

We get: $\frac{\frac{1}{3}87+\frac{1}{3}75+\frac{1}{3}144}{\frac{1}{3}+\frac{1}{3}+\frac{1}{3}} = 102$

Of course, as we mentioned in the beginning, we can use predefined functions to do this for us. There are several Python libraries that contain a function to convolute two arrays. Some run faster than others, depending on the implementation. We won't go in depth in this exercise, but it's important to keep this in mind when working with many large arrays, as is the case with convolutional neural networks. For our purposes, we'll start with the standard .convolve() function from the Numpy library. It takes as arguments the two arrays it needs to convolve, which in our case are the row of pixels 'image_strip' and our filter kernel 'kernel_1d'.

```
result = np.convolve(image_strip, kernel_1d)
```

This is the time to address a little "problem" that arises with image convolutions. It will become very apparent when we visualise the result of the convolution over the entire array.

Because we use a kernel that takes information from the surrounding pixels, this inevitably leads to a conflict when processing the outermost pixels (first and last column, and first and last row). On one hand, inevitably, the resulting array after the convolution will have larger size than the original image. The larger the size of the kernel, the more additional pixels will be generated in the process. And on the other hand, the lack of colour information outside the image frame makes the processing of these edge pixels tricky. There are different approaches to deal with the "problem", that depend on the type of image, its size, or the nature and size of the filter itself. For example, we can apply the filter kernel so that we ignore the pixels under the kernel that are outside the range of the base image (i.e., ignore their weight/impact). Or, another approach would be to duplicate the edge pixels in advance, so that there is material to use the convolution with. Then, after the filtering is done, we could crop the extra pixels to restore the original size. In any case, whichever method we choose, unless the image is extremely small in size (number of pixels), the difference in the effect of the different methods will be negligible.

Since we are working with a relatively small image, this effect will be quite visible. We will partially solve it by simply cropping the resulting array after the convolution, in order to get back to the size of the original (base) image, but the effect of the lack of colour information will remain (i.e. the convolution will treat everything outside the image as 0, which is black, and we will expect to see a darkening of the image frame (the outermost pixels).

Since we already have the result of the convolution stored in the variable 'result', our next step is to crop the extra pixels, so that we again have an image (row) of 32 pixels.

Then we artificially add depth (another dimension) to the result so that MatplotLib's .imshow() function can visualize it:

```
# Crop the result taking only the pixels between the first and last
result = result[1:33]
result_exp = np.expand_dims(result, axis=0) # Add a dimension
```

Let’s visualise the original row and the resulting row (after the convolution) one below the other, so we can easily compare them.

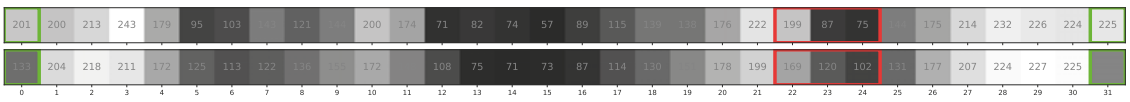
With a red rectangle we will mark the area for which we manually applied the convolutional kernel a while ago. We can see that indeed for the three processed pixels we got the same values: **169**, **120** and **102**.

We also see the darkening effect on the outermost pixels (marked in green). The first and last pixels became 133 and 149 respectively, which is quite a lower (darker) value than the previous 201 and 225. In fact, we can easily verify that the actual integer part of:

$\frac{1}{3}(0 + 201 + 200) = 133$ and $\frac{1}{3}(226 + 224 + 225) = 149.$

Comparing the two rows of pixels, it is obvious that the lower row (the result of the convolution) is a "smoothed" version of the upper one, with smoother transitions. This is, for example, clearly visible for pixels with index 4 to 6, 11 to 13 or 22 to 24, which we last worked with:

```
fig, [ax1, ax2] = plt.subplots(2, 1, sharex=True, sharey=True)

ax1.imshow(image_strip_exp, cmap='gray')
for i in range(len(image_strip_exp[0])):
    text = ax1.text(i, 0, image_strip_exp[0][i], ha="center",
va="center", fontsize=14, color="grey")
ax1.add_patch(Rectangle((21.5, -0.5), 3, 1, edgecolor='red',
fill=False, lw=5))
ax1.add_patch(Rectangle((-0.5, -0.5), 1, 1, edgecolor='lime',
fill=False, lw=5))
ax1.add_patch(Rectangle((30.5, -0.5), 1, 1, edgecolor='lime',
fill=False, lw=5))

ax2.imshow(result_exp, cmap='gray')
for i in range(len(result_exp[0])):
    text = ax2.text(i, 0, int(result_exp[0][i]), ha="center",
va="center", fontsize=14, color="grey")
ax2.add_patch(Rectangle((21.5, -0.5), 3, 1, edgecolor='red',
fill=False, lw=5))
ax2.add_patch(Rectangle((-0.5, -0.5), 1, 1, edgecolor='lime',
fill=False, lw=5))
ax2.add_patch(Rectangle((30.5, -0.5), 1, 1, edgecolor='lime',
fill=False, lw=5))
ax2.set_xticks(range(32))
ax2.yaxis.set_major_locator(ticker.NullLocator())

fig.set_size_inches(32, 2)
```



Now that we know how to apply a convolutional kernel to a one-dimensional array, it's time to move on to two-dimensional arrays and kernels.

If we want to create a mean blur filter, equivalent to what we used so far, but in two dimensions, we can create an array of size 3 x 3 (9 elements) and fill it with equal values (1/9) :

```
kernel_2d = np.array([[1/9, 1/9, 1/9],
                      [1/9, 1/9, 1/9],
                      [1/9, 1/9, 1/9]])
```

Let's do a convolution between our original image (not just the slice, but the full hand image) with the newly created 2D kernel. Again, as with the one-dimensional version, the kernel

moves pixel by pixel, and row by row, until it has processed all the pixels. This time, let's use the SciPy version of the convolution function. There is a significant difference in the implementation between the NumPy and the SciPy libraries. For one, the NumPy version of the only works with one-dimensional arrays. The SciPy not only has a 2D convolution, but also uses FFT (Fast Fourier Transform) to do so, which is computationally much faster. The SciPy convolution is part of the signal processing package.

We will save the result in the 'kernel_2d' variable:

```
result_2d = sp.signal.convolve2d(hand_image_bw, kernel_2d)
```

In order to get a better understanding, let's visualise both the kernel itself, the original image and processed (result of the convolution) image in the same figure (using 1 row with 3 subplots):

```
fig2, [ax1, ax2, ax3] = plt.subplots(1, 3)

ax1.imshow(kernel_2d, cmap='Greys')
ax1.tick_params(left=False, labelleft=False, labelbottom=False,
bottom=False)
ax1.grid(which="major", color='black', linestyle='-', linewidth=1)
ax1.title.set_text('Mean Blur Kernel')
for i in range(kernel_2d.shape[0]):
    for j in range(kernel_2d.shape[1]):
        text = ax1.text(j, i, round(kernel_2d[i, j], 4), ha='center',
va='center', color='black')

ax1.set_xticks([x - 0.5 for x in list(range(1, len(kernel_2d) + 1))],
range(1, len(kernel_2d) + 1), visible=False)
ax1.set_yticks([x - 0.5 for x in list(range(1, len(kernel_2d) + 1))],
range(1, len(kernel_2d) + 1), visible=False)

ax2.imshow(hand_image_bw, cmap='gray')
ax2.tick_params(left=False, labelleft=False, labelbottom=False,
bottom=False)
ax2.title.set_text('Original Image')

ax3.imshow(result_2d[1:33, 1:33], cmap='gray')
ax3.tick_params(left=False, labelleft=False, labelbottom=False,
bottom=False)
ax3.title.set_text('Average Blurred Image')

fig2.set_size_inches(12, 4)
```

Mean Blur Kernel

0.1111	0.1111	0.1111
0.1111	0.1111	0.1111
0.1111	0.1111	0.1111

Original Image

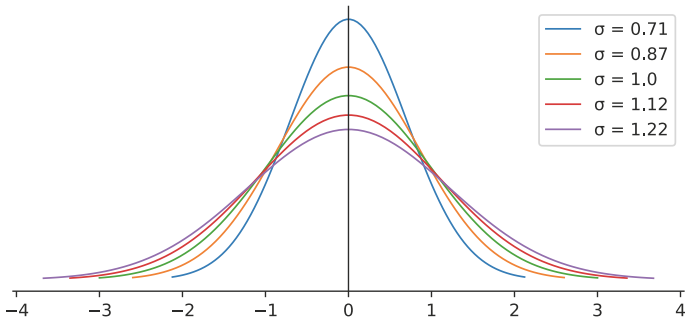
Average Blurred Image

We can very clearly see the dark frame of pixels, which results from the 0 values used (the non-existent colour information) beyond the image size. As obvious, when the kernel is composed of equal weights, the surrounding pixels (relative to the processed pixel) have equal weight in the formation of the resulting value. But of course, depending on the kernel values, the convolution can produce different results.

An example of a more complex kernel is one with a normal distribution (Gaussian), which in practice will result in the so-called Gaussian Blur. This is probably the most common type of blurring, as it gives much more attention to border regions (edges) in the image. The main difference between Gaussian blur and average blur is that with Gaussian we give significant priority to the middle pixel (which is processed), relative to its neighbours. Thus, when the processed pixel is close to an edge (i.e., a sharp change in intensity), the blur will not introduce too much information from neighbouring pixels.

As you probably know, the normal distribution is bell-shaped. The standard deviation of this curve σ (sigma), meaning the average distance to the median μ (mu), will determine how wide the bell is. Thus, a large standard deviation will make the curve very wide (sweepy), and a small standard deviation will make it narrow and steep. Here's what different normal distributions look like, according to the value of their average squared deviations (variance) for example, in the range of 0.5 to 1.5:

```
mu = 0
variance = [0.5, 0.75, 1, 1.25, 1.5]

fig3 = plt.figure()
ax = fig3.gca()
for var in variance:
    sigma = math.sqrt(var)
    x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
    ax.plot(x, sp.stats.norm.pdf(x, mu, sigma), label=f'σ =
{round(sigma, 2)}', linewidth=1)

ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.tick_params(left=False, labelleft=False , labelbottom=True,
bottom=True)
ax.legend(loc ='upper right')
fig3.set_size_inches(7,3)
```



In terms of the effect of the convolution, this means that the narrower the distribution of values in the kernel (with a small standard deviation, i.e. low sigma value), the more the result of the convolution will account for the details in the image (where there is contrast between adjacent pixels).

Below we will define some examples of different effects that can be applied convolutionally and their kernels:

- mean blur
- Gaussian blur
- sharpening
- search for horizontal edges (top/bottom Sobel)
- search for vertical edges (left/right Sobel)
- embossing
- outline

```
mean_blur =np.array([[1/3, 1/3, 1/3],
                     [1/3, 1/3, 1/3],
                     [1/3, 1/3, 1/3]])
gaussian_blur = np.array([[1/16, 1/8, 1/16],
                          [1/8,  1/4, 1/8],
                          [1/16, 1/8, 1/16]])
sharpen = np.array([[ 0, -1,  0],
                    [-1,  3, -1],
                    [ 0, -1,  0]])
top_sobel = np.array([[ 1,  2,  1],
                      [ 0,  0,  0],
                      [-1, -2, -1]])
bottom_sobel = np.array([[-1, -2, -1],
                         [ 0,  0,  0],
                         [ 1,  2,  1]])
left_sobel = np.array([[1, 0, -1],
                       [2, 0, -2],
                       [1, 0, -1]])
right_sobel = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]])
emboss = np.array([[-2, -1, 0],
                   [-1,  1, 1],
                   [ 0,  1, 2]])
outline = np.array([[-1, -1, -1],
                    [-1,  9, -1],
                    [-1, -1, -1]])
```

With the kernels created, we can now define a dictionary of filters, in which we will associate each of the above arrays (values) with the specific name of the filter (dictionary keys). This will allow us to easily iterate through them for our next visualisation:

```
filters = {'Mean Blur':mean_blur,
           'Gaussian Blur':gaussian_blur,
           'Sharpen':sharpen,
           'Top Sobel':top_sobel,
           'Bottom Sobel':bottom_sobel,
           'Left Sobel':left_sobel,
           'Right Sobel':right_sobel,
           'Emboss':emboss,
           'Outline':outline}
```

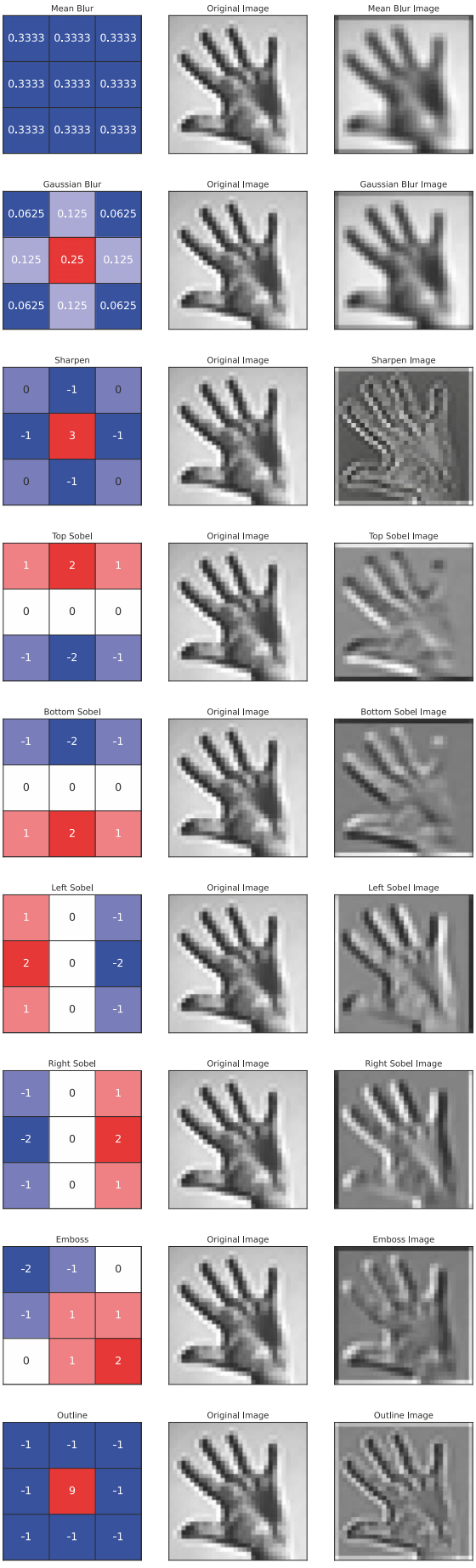
Let’s take all of these different kernels and see what will be the result of convoluting them with our base image (of the hand). For this purpose, we will create an empty list 'results' in which we will be storing the results. Next, we'll create a MatplotLib figure with 18 subplots

spread over 9 rows of 3 columns. On each row, we will show the **kernel**, **the original image**, and **the convolution result**.

We'll do this with a loop that will go through all kernels in the 'filters' dictionary, taking the index of the kernel (value) in the dictionary and its key (the filter name):

```
# Create an empty list for the results
results = []

# Create a figure with 18 subplots (9 rows and 3 columns). The subplot
axis are stored in the 'axs' list
fig4, axs = plt.subplots(9, 3)

# Enumerate all elements of the filters dictionary
for idx, kernel in enumerate(filters):

    ### FIRST COLUMN ###
    # In the first (0 index) column show the kernel pixels (just
colour), using oposing colours colourmap (blue/white/red)
    axs[idx, 0].imshow(filters[kernel], cmap='bwr')
    # fix the X and Y axis tick marks, so that they are in the
    # beginning and end of pixes, rather than in the middle
    axs[idx, 0].set_xticks([x - 0.5 for x in list(range(1,
len(kernel_2d) + 1))], range(1, len(kernel_2d) + 1), visible=False)
    axs[idx, 0].set_yticks([x - 0.5 for x in list(range(1,
len(kernel_2d) + 1))], range(1, len(kernel_2d) + 1), visible=False)
    # hide the tick marks and labels from the axis
    axs[idx, 0].tick_params(left=False, labelleft=False,
                            labelbottom=False, bottom=False)
    # Add grid lines at the place of the major tick marks
    axs[idx, 0].grid(which="major", color='black',
                     linestyle='-', linewidth=1)
    # Set the title of the plot to be the key of the current
dictionary item (which holds the filter name)
    axs[idx, 0].title.set_text(f'{kernel}')
    # Display the values of the kernel on top of the 'pixels'
    for i in range(filters[kernel].shape[0]):
        for j in range(filters[kernel].shape[1]):
            # If the value is 0 make the text colour black
            if filters[kernel][i, j] == 0:
                text = axs[idx, 0].text(j, i,
                round(filters[kernel][i, j], 4),
                ha='center', va='center', fontsize=15, color='black')
            # if it is anything else (+ or - value) make it white
            else:
                text = axs[idx, 0].text(j, i,
                round(filters[kernel][i, j], 4),
                ha='center', va='center', fontsize=15, color='white')

    # Use the current kernel to do a convolution with the original
    # image and append the result to the list
    results.append(sp.signal.convolve2d(hand_image_bw,
                                        filters[kernel]))

    ### SECOND COLUMN ###
    # In the second (1 index) column show the original image
    axs[idx, 1].imshow(hand_image_bw, cmap='gray')
    # Set the title
    axs[idx, 1].title.set_text('Original Image')
```

```
    # Hide the tick marks and the labels
    axs[idx, 1].tick_params(left=False, labelleft=False,
                            labelbottom=False, bottom=False)

    ### THIRD COLUMN ###
    # In the third (2 index) column show the convoluted image
    # (which was just apended to the results list)
    # crop the extra pixels from the front and the end
    axs[idx, 2].imshow(results[idx][1:33, 1:33], cmap='gray')
    # Set the plot title
    axs[idx, 2].title.set_text(f'{kernel} Image')
    # Hide the tick marks and the labels
    axs[idx, 2].tick_params(left=False, labelleft=False,
                            labelbottom=False, bottom=False)

# Set the figure size and save it as a file (if you want)
fig4.set_size_inches(12, 40)
```


Mean Blur
0.3333 0.3333 0.3333
0.3333 0.3333 0.3333
0.3333 0.3333 0.3333
Original Image
Mean Blur Image
Gaussian Blur
0.0625 0.125 0.0625
0.125 0.25 0.125
0.0625 0.125 0.0625
Original Image
Gaussian Blur Image
Sharpen
0 -1 0
-1 3 -1
0 -1 0
Original Image
Sharpen Image
Top Sobel
1 2 1
0 0 0
-1 -2 -1
Original Image
Top Sobel Image
Bottom Sobel
-1 -2 -1
0 0 0
1 2 1
Original Image
Bottom Sobel Image
Left Sobel
1 0 -1
2 0 -2
1 0 -1
Original Image
Left Sobel Image
Right Sobel
-1 0 1
-2 0 2
-1 0 1
Original Image
Right Sobel Image
Emboss
-2 -1 0
-1 1 1
0 1 2
Original Image
Emboss Image
Outline
-1 -1 -1
-1 9 -1
-1 -1 -1
Original Image
Outline Image

Now that you are familiar with convolutions, it is time to dive into Convolutional Neural Networks (CNNs). Let’s discuss the architecture of those networks and how they work.

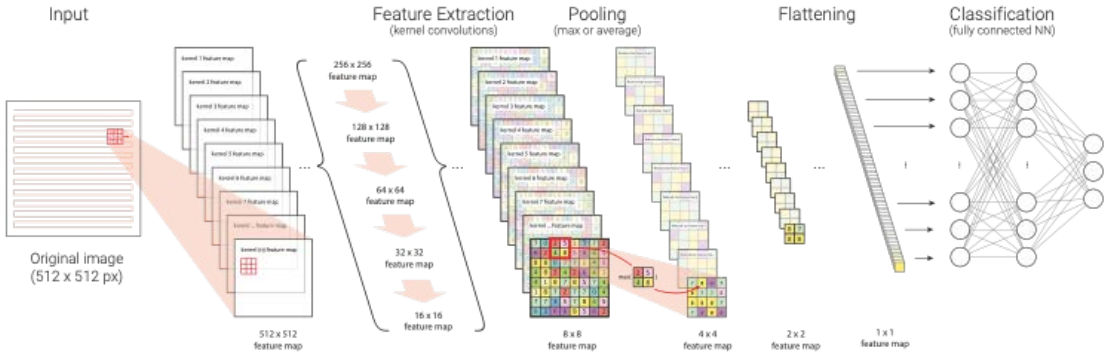


Figure 79. A general representation of the structure of a Convolutional Neural Network.

Figure 79 shows the main elements in the architecture of convolutional neural networks. At the input we feed an image. It goes through a series of processes, the first of which is to be subjected to various convolutional kernels (like the 9 example kernels which we tested in the previous exercise). These would result in separate convolutions, each representing a specific "map" of features (edges, spots, textures, etc.). After that, it goes through a pooling operation. Essentially, this is halving the size of the resulting images (feature maps) after the convolutions. This can be done in two ways, either by averaging the values of neighbouring pixels (average pooling) or by taking the maximum of the values (max pooling). In the diagram above we have shown an example of Max Pooling. Thus, each area of 2x2 pixels is equated to one (average or maximum of them) pixel.

We do this for several reasons. On the one hand, we want to reduce the number of pixels we work with, in order to save computing power. On the other hand, this way we enable our model to learn with a higher degree of abstraction what defines the objects we want to recognize. Also, in this way, with each subsequent step we remove the spatial information about the features, which means that the model will learn to recognize the objects no matter where (or how big) they are in the input image.

This series of steps is usually repeated until the resulting images (after the convolution and pooling operations) are too small to apply more convolutional kernels. After that, we have to flatten the last obtained features to a one-dimensional vector. Then the flattened vector is fed to a neural network of fully connected neurons. At the output of the network, we will have as many output neurons as classes we want to recognize.

As mentioned in the beginning of this topic, during training we feed the model labelled images and it tries to guess which filter kernels (with what weights in them) to apply, so that it finds enough characteristic features that represent the objects good enough. This is done by a process of (error) signal backpropagation. Because the examples are labelled, the model knows what is in the output. It then tries to fit the weights of each previous layer so that it gets the same output result.

Exercise VIII.3-2: Use Python to build and train a convolutional neural network to recognize and locate objects in images.

Let's imagine we are building a control system that relies on hand gestures to manipulate some machine. In order to achieve that we will have to first teach the system to recognise what a hand is, what are the different landmarks (fingers, finger joints, finger tips, palm, etc.) that comprise a hand. For that reason, we will have to build a learning dataset of hand images in various environmental conditions and then label them with all the landmarks that we would want that system to learn to distinguish.

So, our first task will be to understand how to label images. Different tools and different annotation file formats can be used for this purpose. Some of them are open source and free to use, others are paid. Some are installed locally, and others can be used online through a web interface. Their use largely depends on the project and the neural network framework that will be used. We will use LabelImg [17], as shown on Figure 80. Some alternatives are Label Studio, which is developed by the same team but has more advanced capabilities and a web interface, or Computer Vision Annotation Tool (CVAT), or even using an already trained model to go through all the images in a new dataset and label them automatically.

To use the tool, we must install it through the command line (Terminal for Unix-based machines, Command Prompt or Power Prompt for Windows-based machines) with the already familiar procedure "pip install labelimg". After that we can launch the tool from our project folder by simply typing the name in the terminal: "labelimg"

The tool allows us to open one or multiple images (in a folder) and to specify the available classes (objects) for each of them. In addition, from the toolbar we can select the button for marking a rectangular selection, and in this way indicate where exactly in the image the object of the corresponding class is located:

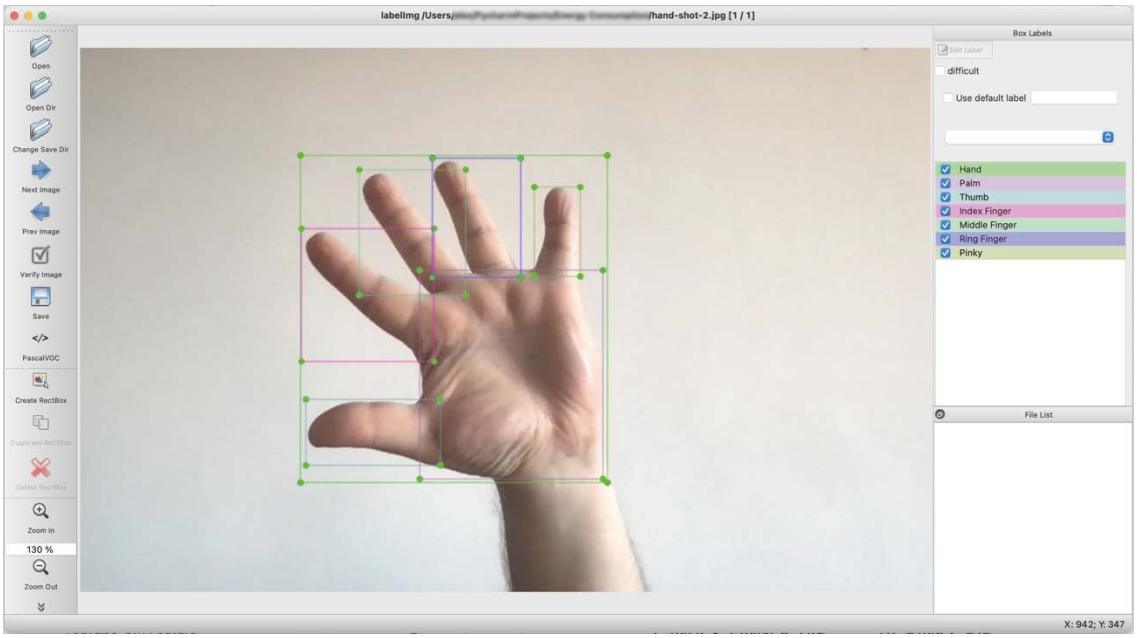


Figure 80.LabelImg interface, allowing images to be manually labelled with the necessary classes.

Once prepared, the images can be divided into separate groups for training, testing and validation of the model.

Building and training a convolutional neural network is not easy or fast. You could attempt to do so on your home computer, given you have the resources, but for the purpose of this exercise, we will use a pretrained convolutional network to recognize (classify and localise) classes of objects in images. The model we will use is MediaPipe Hands [18], which is a hand landmark recognition model (fingers, phalanges, palms, etc.). MediaPipe is a Google project and is free to use. The footage we will work with will be taken from a webcam connected to our computer, regardless of whether it is built-in or connected via USB. If you don't have a camera, you can use an already recorded video (you can find and download one off the Internet), or just a photo from your phone.

Let's start by importing the necessary libraries. We only need three, the Open Computer Vision 2 (cv2), which will allow us to capture video stream from our camera and manipulate it. The MediaPipe library we already discussed, we are going to import it as "mp". And finally, Matplotlib pyplot for visualisation purposes:

```
import cv2
import mediapipe as mp
import matplotlib.pyplot as plt
```

We'll start by creating a video capture object, using the open computer vision library (cv2). The value in the brackets is the index number of the camera we want to use in the list of available cameras in the system. If you only have one it will be at index 0, but if you have more you might have to change that number accordingly:

```
cap = cv2.VideoCapture(0)
```

The video object we created has a .read( ) method that returns two things, a Boolean value (True or False) depending on whether it successfully read a frame, as well as the video frame (image), in case of a successful read. We can store these two results (the successful read and the frame) in two variables. In 'ret' (short for retrieval) we will keep information about whether the frame was successfully read, and in 'frame' we will store the image itself.

Let's run the below line of code that will read a frame (take a picture) from the camera. Since we are going to build a CNN for hand landmark detection and classification it might be a good idea the photo we take to be of a hand. So, hold your hand in front of the camera and run the next line of code. We will later feed that image trough the classification model:

```
ret, frame = cap.read()
```

After executing the above line, if your camera has an indication (e.g. an LED) that shows when the camera is triggered, it should turn on. We can check if a frame was successfully read by checking the value of the 'ret' variable:

```
print(ret)
```

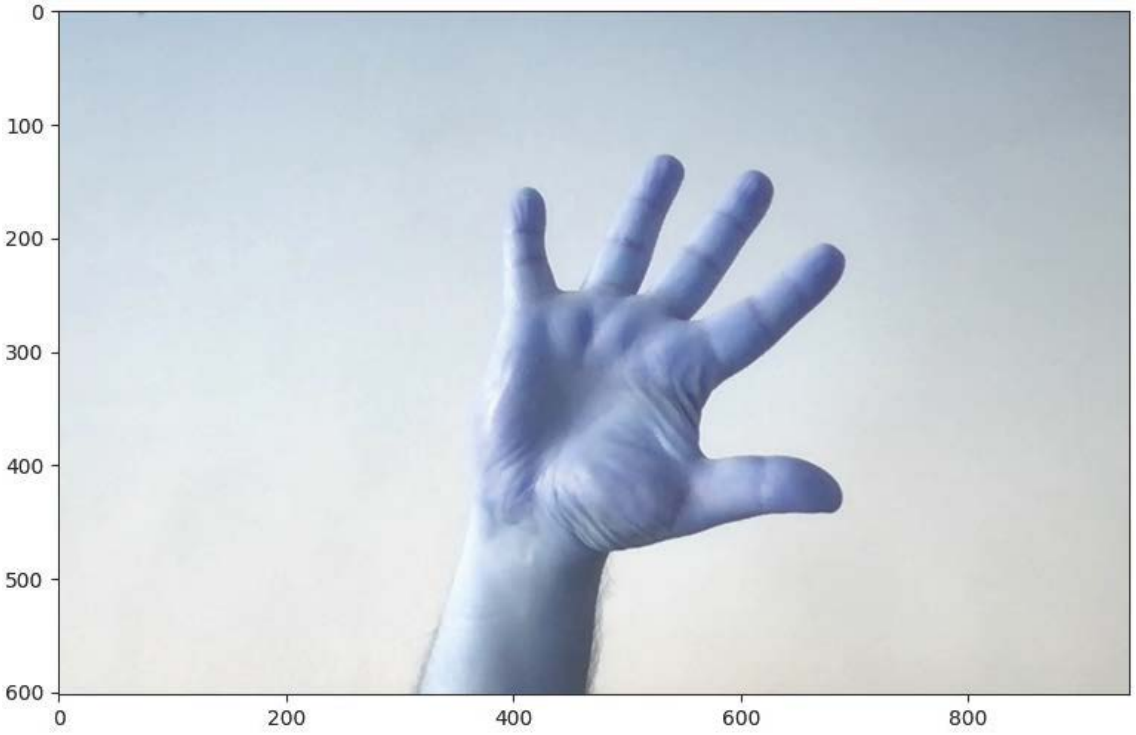
If the value in 'ret' is "True" then we have successfully read a frame. We can visualise it using MatplotLib. Note that the frames come in the form of a Numpy array, which means that if you try to render them simply by calling them as a variable, you will only get the numerical values of the pixels, but not their colour interpretation.

We can verify this by printing the type of the 'frame' variable. We can also check the array size and its depth (number of colour channels) by printing the array shape:

```
print(type(frame))
print(frame.shape)
```

We can use the .imshow() function of Matplotlib's pyplot to draw the frame. This function takes any array-like object and interprets the values as pixel colour intensity:

```
plt.imshow(frame)
plt.rcParams['figure.figsize'] = [10,6]
```



You'll immediately notice two peculiarities of the footage coming from the CV2 captured frame. First, the colours don't look quite right. This is because CV2 arranges the colour channels slightly differently. By default, images on your computer (intended to be displayed on a screen) are stored in an RGB (Red, Green, Blue) colour scale. CV2 arranges the channels in BGR (Blue, Green, Red). So, in the preview above, the red and blue channels are swapped and thus the image looks bluish.

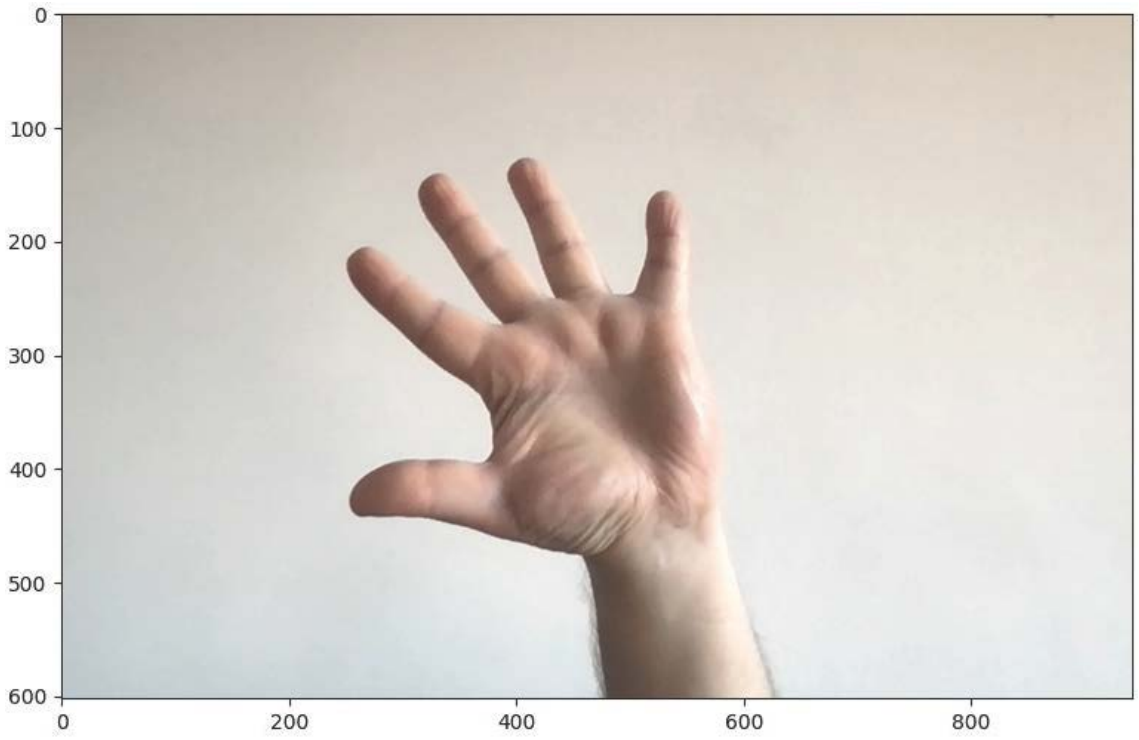
Second, the shot is a mirror image of what we expect to see. In reality, the shot is true from the point of view of a person looking across from us, but we are accustomed when we look at our own image on a screen (through the camera) to expect to see ourselves as in a mirror.

We can "fix" both "problems" by manipulating the frame. First, we'll convert it from BGR to RGB, and then we'll flip it horizontally (i.e., mirror it over the vertical axis):

```
frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
#0=x axis (vertical flip), 1=y axis (horizontal flip), -1=both axis
frame_rgb = cv2.flip(frame_rgb, 1)
```

Let's visualise the manipulated frame, which we just stored in the new variable "frame_rgb":

```
plt.imshow(frame_rgb)
plt.rcParams['figure.figsize'] = [10,6]
```



Now that we have a frame in the required format, we can feed it to MediaPipe's trained convolutional model to see if (and how well) it does at finding the hand in the frame.

Our first step is to create an object containing the model. We will name our object 'mp_hands', and we will assign it an instance of the hands model (part of the solutions package of MediaPipe).

Next, we'll create the 'hands_static' variable, which will be a reference to the model's Hands class, and pass in some parameters. For example, we'll specify that we'll use it for a static image (we'll change this parameter later when we apply the model to a video stream), and we'll also limit the number of hands the model will track to 1. (i.e., if there are more than one hand in the frame, we will store "landmark" information only on the first one detected. We will also set a minimum model confidence threshold in our classification of 0.6. So, if the model is not at least 60% sure that the object is a hand it will be ignored. We can do the same not only for classification, but also for finding the object in the frame (localisation/tracking).

Finally, we need to create a reference to the hand's landmarks drawing function as well. Let's name it 'mp_draw'. We will use it to draw the different points of the hand on the frame:

```
mp_hands = mp.solutions.hands
hands_static = mp_hands.Hands(static_image_mode=True, max_num_hands=1,
min_detection_confidence=0.6, min_tracking_confidence=0.6)
mp_draw = mp.solutions.drawing_utils
```

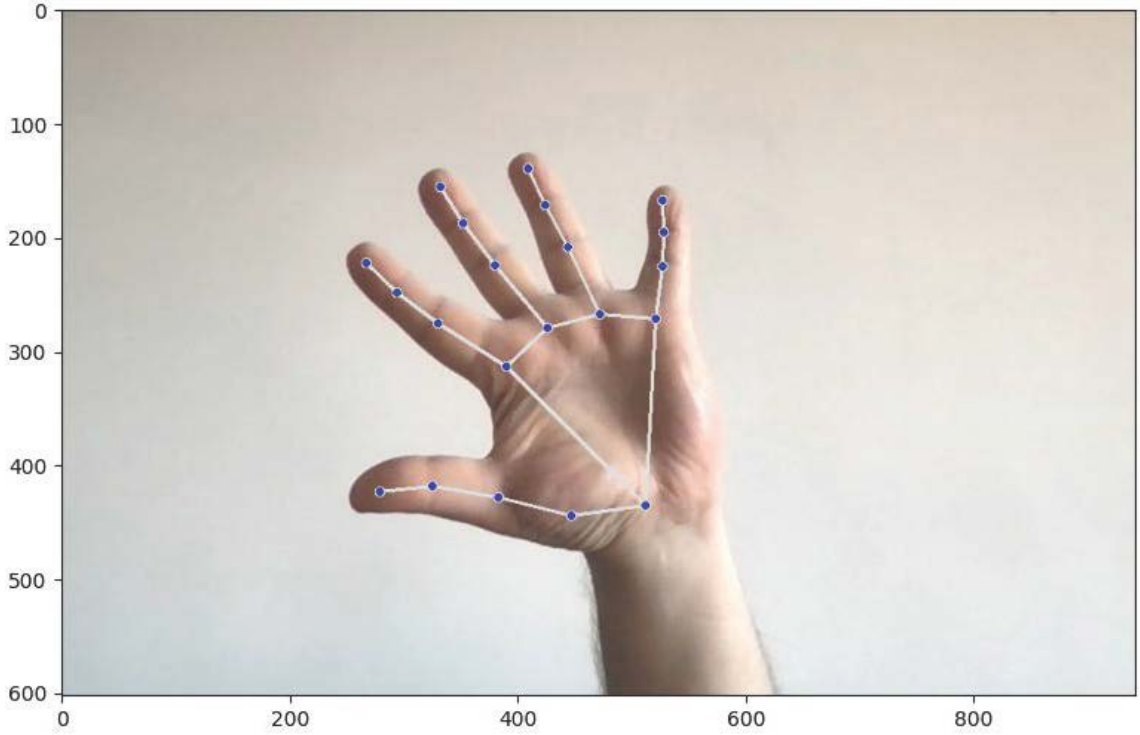
Next, we'll pass the frame to the .process() function of the model and assign the result of the classification/recognition in a new variable "result".
Then, we will take whatever hand landmarks have been found (they are stored in the .multi_hand_landmarks property of the result object) and store them in a new variable "hand_landmarks".

Since the "hand_landmarks" variable can be empty (if no hand or landmarks have been found in the provided image), before we draw them on the frame, we have to make sure the variable is not empty. To do this we can use a simple conditional (if) statement. If it is True, then we use the draw_landmarks function. We pass the frame we want to draw on top of, and then we define which landmarks (for which hand) we want to draw. In our case (as we only have one had) it will be in index 0. Finally, we can also supply an additional setting to the function that tells it that we want the landmarks to be drawn with connection lines between them (otherwise they will be just joint points):

```
result = hands_static.process(frame_rgb)
hand_landmarks = result.multi_hand_landmarks

if hand_landmarks:
    mp_draw.draw_landmarks(frame_rgb, hand_landmarks[0],
                           mp_hands.HAND_CONNECTIONS)
```

Let's see what we have drawn. We use pyplot to visualise the RGB frame again:

```
plt.imshow(frame_rgb)
plt.rcParams['figure.figsize'] = [10,6]
```



The video recognition process is not significantly different, except that we need to make a loop where we constantly ask the camera for new frames. Before we continue, however, let's create a new version of the model to work with dynamic instead of static frames (i.e., video):

```
hands_dynamic = mp_hands.Hands(static_image_mode=False,
                           max_num_hands=1, min_detection_confidence=0.6,
                           min_tracking_confidence=0.6)
```

We will use a 'while' loop. In it, the first thing we do is read a new frame on each iteration. Then, if we have a successfully read frame (by checking the "ret" variable), we mirror it and convert it to RGB.

Then we feed the converted frame to the new dynamic classification model and save the recognition results in the 'result' variable.

If the model has found a hand in the frame, for each hand (in our case we have limited it to 1, but there can be more) we extract the coordinates of its "Landmarks". The model documentation describes 21 points (observations) that can be classified and located in the frame.

For the purposes of this exercise, we will use the tip of the index finger as an example. The coordinates that the model returns are normalised (so that they are not dependent on the size of the image). This means that they are values between 0 and 1, where 0 is the start of the corresponding frame axis and 1 means the end. So, 0.5 would be in the centre. For example, if this point has normalised coordinates (0.3, 0.5) it means that it is located at 30% of the horizontal dimension and 50% of the vertical dimension. To get the real coordinates, as pixel indices, we need to multiply the normalised coordinates by the corresponding image size. We store the de-normalised result in the variables 'x_finger' and 'y_finger' respectively.

As we go through each hand landmark, we also have to draw them on the frame. For this purpose, we will use the drawing tool 'mp_draw' which we have already addressed above. We feed it the frame we want to draw on, as well as the corresponding elements. In our case, this is each 'hand_landmark' object from the loop we iterate them with. After that we can add additional arguments about the way and style of drawing. The only thing we will add is to specify that we want the points to be connected by lines (HAND_CONNECTIONS).

We use the original frame, not the one converted to RGB. The reason is that CV2's .imshow( ) function expects to receive a CV2 frame, which as we've already established is in BGR channel arrangement by default. And to save ourselves one conversion of the processed frame back from RGB to BGR, we can just use the already existing original frame.

We can also add a text line in the frame that shows the *x* and *y* coordinates of the index fingertip, using the .putText() function of open CV.

Then we submit the frame (with the already drawn points and connections) to .imshow() to be displayed. Note that this happens outside of the loop that iterates the landmarks and after checking for existence of the classification results. Otherwise, we would only display a frame when there is a hand in it, and we want to see what the camera sees even if there are no hands or landmarks in the frame.

After that we can also put a small code section that will trigger a break in the camera display loop if we press a specific key (in our case 'q'). That will allow us to stop the otherwise endless "while True" loop.

After exiting the loop, we release the video device and close all windows (associated with the video stream):

```
while True:
    ret, frame = cap.read()

    # if the frame was retreived successfully
    if ret:
        frame = cv2.flip(frame, 1)  # Flip frame horizontally
```

```
        framergb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB) #RGB Convert
        # Process the frame and store the results:
        result = hands_dynamic.process(framergb)

        # If there are results
        if result.multi_hand_landmarks:
            # Extract the landmarks from the results
            # and iterate through them
            for hand_landmark in result.multi_hand_landmarks:
                # Get normalised index finger coordinates for x and y
                # Multiply them by the respective image size
                # to get actual pixel coordinates
                x_finger =
int(hand_landmark.landmark[mp_hands.HandLandmark.INDEX_FINGER_TIP].x *
frame.shape[1])
                y_finger =
int(hand_landmark.landmark[mp_hands.HandLandmark.INDEX_FINGER_TIP].y *
frame.shape[0])
                # Draw the landmarks on the (original) frame
                mp_draw.draw_landmarks(frame, hand_landmark,
                                       mp_hands.HAND_CONNECTIONS)
                # Place the coordinates text on the frame
                cv2.putText(frame,
f'Index finger tip coordinates: x={x_finger}, y={y_finger}',
(20, 30), cv2.FONT_HERSHEY_PLAIN, 2, (0, 0, 255), 2)

        # Show the image
        cv2.imshow(f'camera', frame)

    # Check if the 'q' key is pressed to break the loop
    if cv2.waitKey(25) & 0xFF == ord('q'):
        break

# When the loop is broken release the Capture device
# and close all windows
cap.release()
cv2.destroyAllWindows()
```

The MediaPipe model enables creative solutions in a variety of areas. For example, an automatic sign language translation system. Or we can create an autonomous safety system on a machine that stops the execution of a process if a worker/operator's hands are in a danger zone, and many other different applications.

Chapter IX.
Linear Optimisation

Optimisation is another task that is often required from engineers and/or managers, and as such is usually embedded in many control solutions. It can be applied to a large variety of problems, but the goal is almost always the reduction (finding a minimum) or the increase (finding a maximum) in the expenditure (cost) or gain (revenue) of some resource, given some set of criteria.

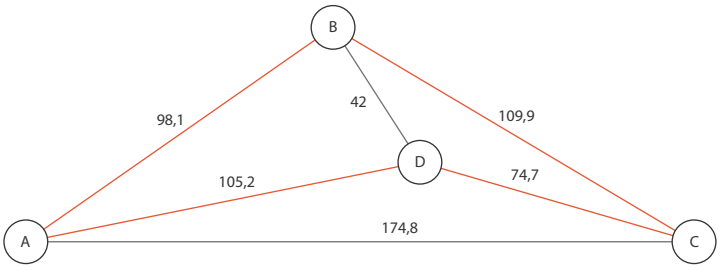
Linear optimisation specifically addresses the Process of optimising a linear function that has linear constraints. These kinds of algorithms search for an optimal solution by traversing the parametric space, defined by the objective function, until there is no more improvement (increase or decrease) in the results.

Very often Linear Optimisation is approached from a constraints perspective, looking for the best solution in a subset of feasible solutions, rather than trying to find the most optimal solution from all solutions that could exist. In that way an optimisation problem might not even have an objective function. The task might be to narrow down a very large set of possible solutions to a smaller set, or a single solution, by adding constraints to the problem. Also, depending on the used solver, the problem might need to be constrained to integer values of the used parameters and constraints. In that case we will be talking about Constraints Programming and Integer (or Mixed Integer) Optimisation.

IX.1. Traveling salesman and general routing problems

Some of the most common types of optimization tasks (in the field of linear optimisation) are the so-called routing problems. The goal is to find the most optimal path for a set of vehicles that need to reach a certain number of locations (or customers), starting from and ending at the same start/exit point. Under optimal path most often we understand the path with the lowest cost, which is usually defined as the shortest path (since in reality a large part of the logistics costs can be derived per unit distance). Of course, the task can be relatively simple or extremely complex, depending on the set of additional conditions. For example, a customer/location may only need to be visited once, or it needs to be visited within a limited time range, or maybe the vehicles have a certain volume or weight capacity, and of course, drivers might have to drive for a certain amount of time and then must have an obligatory rest, etc.

We are going to start with the simplest version, a special case of the routing problem, commonly known as the traveling salesman problem. It is when we only have one vehicle and multiple locations/customers to be visited. Let's look at a simple example represented by the graph in the following figure:



Possible routes (the optimal is in red)

ABCDA	98,1	109,9	74,7	105,2	387,9
ABDCA	98,1	42	74,7	174,8	389,6
ADBCA	105,2	42	109,9	174,8	431,9
ADCBA	105,2	74,7	109,9	98,1	387,9
ACBDA	174,8	109,9	42	105,2	431,9
ACDBA	174,8	74,7	42	98,1	389,6

Figure 81. A graph representation of a simple traveling salesman problem. Each vertex of the graph is a route with a specific distance. Assuming node 'A' is the starting one, the table shows all possible route variations.

The link values in the graph (Figure 81) represent the distances between individual locations. In this situation, we can find the shortest distance by making a list of all possible routes and then easily compare them to see which one is the shortest (the table on the right of the graph). Since the condition is to start and finish the route from/to the same location, it is normal for the shortest distance route to have two options, depending on which direction we decide to start the loop. Of course, this would not be the case if the sequence of visiting the locations was a factor in the optimisation process.

The task is relatively easy with a small number of locations, and respectively a small number of routes, that can be easily compared to each other – an exhaustive search approach. Things start to get significantly more difficult as the number of possible routes increases. For example, for the above case where we have a total of 4 locations, one of which is the origin/exit, the possible route permutations would be $3! = 6$ (as can be seen from the table). In other words, if we generalise, for n total locations and 1 output the formula would be $(n-1)!$. If we calculate the number of possible routes for 20 points, we will see that it is $(20-1)! = 19! > 1.2 * 10^{17}$ permutations. While a list with this many options can still be relatively quickly iterated and compared by a modern computer, as locations and conditions grow this exhaustive search approach starts to become extremely inefficient and time consuming. Imagine there were 150 locations? That's more than $3.8 * 10^{260}$ possibilities. With such a quantity of options, we must approach the task intelligently and take advantage of some kind of optimisation strategy that will allow us to "search" the space of possibilities without having to compare each of them with each other.

Exercise IX.1-1: Use Python to find an optimal solution to a Traveling Salesman problem.

Our firs exercise will be to find an optimal route to visit a number of European cities, starting and ending the route in one specific location.

Depending on the software package you have access to, the optimisation task may be defined differently, but all packages provide the ability to perform linear optimisation. We will focus on OR Tools for Python [19]. This package is developed by Google and publicly available for free. As usual, we'll start by importing all the libraries we need.

Apart from Pandas, which you should already be familiar with, all other libraries are probably new to you. I will briefly describe each of them and its purpose.

GeoPandas [20] is an additional library based on Pandas, which is used for working with geospatial information, like Pandas it uses tabular data structures (in GeoDataFrame format) by default. Since the sample data we will be working with are cities with their underlying information in tabular form, this library is ideal for our purposes.

Shapely and Folium are auxiliary libraries for our visualisations. Shapely will allow us to generate the lines between the coordinates of individual locations, and Folium is a library that works with < OpenStreetMap [21], which allows us to visualise our geo-spacial data on top of an interactive world map.

Haversine [22] is a package including the function of the same name [23], for calculating the distance on a spherical surface between two points. Essentially, the function calculates half of the versus (remainder to 1) of the sine of the central angle (of the sphere on whose surface they lie) of the corresponding two points.

And finally, we have to import both functions that we will use from the ORTools library. Both are part of the conditional extremum search (Extremum of an objective function given parameter constraints) package 'constraints_solver'. the functions we will use are 'pywrapcp', which is a Python wrapper over the C implementation of the optimisation algorithms, as well as the set of route enumerators in 'routing_enums_pb2'.

```
import random
import pandas as pd
import geopandas as gpd
import shapely
import folium
from haversine import haversine
from ortools.constraint_solver import pywrapcp, routing_enums_pb2
```

Knowing what libraries we will need, our next step is to get the information we will be working with. Our first objective in solving the traveling salesman problem will be to optimize the route of a vehicle so that it visits a certain number of cities, aiming for a minimal total length of the route. For this we need a list of cities and their global coordinates.

We can visit OpenDataSoft [24] and download a .csv file containing all European cities with a population of over 1000 people. If you want to test what you learned in the API exercises, you can try, as a self-exercise, to download the data automatically via their API, without having to manually download the file.

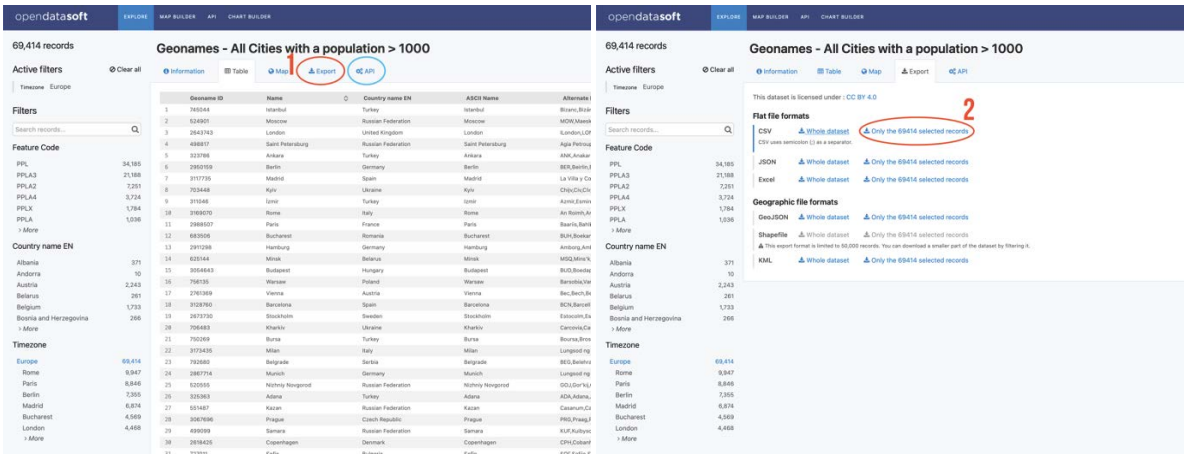


Figure 82. OpenDataSoft's website. You can see the query you need to use to get the same result by clicking on the "API" tab (marked with a blue oval in the image). Otherwise, to download the data as a .csv file, you need to click on the "Export" tab (marked as step 1 in a red oval in the image). On the next screen, you have to select the option to download the data as a .csv file (step 2). You also have to decide if you need to download ALL the eauropean cities, or just the filtered ones (with a population over 1000).

As we have done many times before, we will load the downloaded file in Pandas DataFrame format. Note that under the file download option on the website we were warned that the delimiter used is ";". We will need to make this clarification in the file read function, otherwise we will not get correct results. Also, to make things easier, we'll tell Pandas to use the city names as an index column, which we will use to refer to their respective rows of information. To do this, we specify that the index column of the newly created DataFrame will be the 'Name' column from the downloaded file:

```
all_cities = pd.read_csv(
             'data/geonames-all-cities-with-a-population-1000.csv',
             index_col='Name', delimiter=';')
```

Let's see what we got in the 'all_cities' table:

```
all_cities
```

If the data import was successful, we should see the table with all the data fields. Our next step will be to sort the table by population, so that the cities with most citizens are at the top. To do that we have to tell the .sort_values( ) method that we want the sort to be done by the 'Population' column, in descending order (ascending=False), as well as that we want the action to be performed in place (inplace=True), which will automatically replace the old table with the new ordered one:

```
all_cities.sort_values(by=['Population'], ascending=False, inplace=True)
```

Now is the time to consider how many locations we want to use for our exercise. I suggest using the first 150 cities (with the largest population) by calling 150 entries from the head of the 'all_cities' table, and assigning them to a new variable that we'll just call 'cities'.

```
cities = all_cities.head(150)
```

Our next step will be to clean the table of information we don't need. As we saw above, when we visualised the table after importing the data, there are a lot of fields (columns) that we won't need. We'll use the .drop( ) method, giving it a list of the names of these columns, so they can be dropped from the table. In order to specify that we mean columns and not rows, we will use the 'axis=1' parameter:

```
#axis=1 searches columns (header names),
#axis=0 searches rows (index names)
cities = cities.drop(['Geoname ID',
                      'ASCII Name',
                      'Alternate Names',
                      'Feature Class',
                      'Feature Code',
                      'Country Code 2',
                      'Admin1 Code',
                      'Admin2 Code',
                      'Admin3 Code',
                      'Admin4 Code',
                      'Elevation',
                      'DIgital Elevation Model',
                      'Timezone',
                      'Modification date',
                      'LABEL EN'], axis=1)
```

Since we will need to work with the coordinates of the locations (cities), we need to make sure that we have them in a suitable format. If we take a look at the 'Coordinates' column we will see that the latitude and longitude values are given consecutively (separated by a comma). I.e. the table currently treats them as a text string. The value of the third coordinate - the geographic altitude is in the 'Elevation' column, which we didn't really need and have already removed.

To be able to use the latitude and longitude, however, we'll need to extract them as separate values and make sure to store them as numeric (floating point) values. For this purpose, we will use two loops to go through the contents of the entire 'Coordinates' column, and for each coordinate ('coord') split the text string where the comma is located. We do this with the .split( ) method, specifying the comma (,) as the character to split the string on. This method returns a list

of two elements, one element is the string before the comma, the other element is the string after the comma.

I.e. to get the longitude we need to select the first (index 0) item from the list. Also, since the element after the split inherits the data type, namely a text string, we must convert it to a floating-point number.

For the latitude, we do the same, with one small but very significant detail, first, instead of the first element of the result list, we take the second element (index 1). And secondly, we notice that after the comma in the original text string there was a blank space, which currently stands at the beginning of the new element (after we did the split). I.e. to get only the numbers as characters, we need to ignore the first character (the blank space) and take everything after it [from 1 : to the end]. Then, as with the longitude, the result of this operation goes through conversion to a floating point number.

As mentioned, we do this for each row of the coordinate column, and write the resulting list of separated values into new columns, which we name abbreviated ['Lat'] for 'Latitude' and ['Lon'] for 'Longitude' respectively:

```
cities['Lat'] = [
     float(coord.split(',')[0]) for coord in cities['Coordinates']]
cities['Lon'] = [
     float(coord.split(',')[1][1:]) for coord in cities['Coordinates']]
```

Once we have successfully separated the two coordinates, and stored them separately in their corresponding columns, we can safely delete the 'Coordinates' column:

```
cities = cities.drop(['Coordinates'], axis=1)
```

Let's see what we got left with in the 'cities' table:

```
cities
```

Name	Country Code	Country name EN	Population	Lat	Lon
Istanbul	TR	Turkey	14804116	41.01384	28.94966
Moscow	RU	Russian Federation	10381222	55.75222	37.61556
London	GB	United Kingdom	8961989	51.50853	-0.12574
Saint Petersburg	RU	Russian Federation	5351935	59.93863	30.31413
Ankara	TR	Turkey	3517182	39.91987	32.85427
...	...	...	...	...	...
Tirana	AL	Albania	418495	41.32750	19.81889
Wandsbek	DE	Germany	411422	53.58334	10.08305
Palma	ES	Spain	409661	39.56939	2.65024
Szczecin	PL	Poland	407811	53.42894	14.55302
Ivanovo	RU	Russian Federation	406113	56.99719	40.97139

150 rows × 5 columns

Now that we have the coordinates of the locations, we are ready to start thinking about the distances between them. In a real environment, the distances between cities will be calculated based on the road network (if we are talking about land transport), such as roadways or railways. Naturally, additional parameters may be added to the optimisation based on the conditions of these roads (for example, roads with and without toll fees, different level of traffic load, etc). In practice, this would lead to additional variations of the routes between individual locations.

For the purposes of our example, we will idealise the situation as much as possible by simply calculating straight-line distances. Of course, talking about a straight line on the globe is a nonsense, so we need to find a way to calculate the arc lengths (the distances on the surface of the earth) that connect the coordinates of our locations.

Fortunately, there is a formula for just this purpose. As mentioned at the beginning, a great way to do this would be to use the Haversine Formula. It uses the latitude and longitude coordinates to find the distance between them along the circumference of the glove.

Essentially, we'll need a loop that iterates through each city and finds the distance to all other cities. The result will be a distance matrix.

Let's construct the loop that will populate the table with the required calculated distances.
First, as we said, we want to go through every city on our list. Then, on every iteration of the main loop, we create an empty list 'current_city_distances' in which we will store the distances to all other cities from the city we are currently processing.

We then create that second (nested) loop that goes through all the cities and runs for every city in the main loop. Here we first output a message to the console to give us an idea of which city pair we are currently processing. Then, we check if the city, to which we want to calculate the distance, is not the same city as the one we are currently processing (from which we start to calculate the distance). If it is, instead of calculating with the formula, we simply write 0. In practice, the formula will also give 0, but due to rounding errors we might get something slightly off zero. To avoid the possibility that the distance between Istanbul and Istanbul turns out to be 0.0002 km for example, it is better to set it to 0 manually, instead of relying on the Haversine formula.

Otherwise (if the city is different), we must use the Haversine formula to calculate the distance. Here we create two variables 'city1' and 'city2' to store the pairs of coordinates of the two cities. In 'city1' we assign the coordinates of the city we are processing (from the main loop), and in 'city2' we store the coordinates of the destination city (with index 'i') that we have reached in this inner loop. For 'city1' we use the .get_loc( ) method on the index column of the table (which if you remember contains the city names), passing it the name of the currently processed city contained in 'city'. The result is the index of the row that contains that city. Accordingly, we use this index to refer to the ['Lat'] and ['Lon'] columns and retrieve the corresponding values from them.

After that we feed the two variables with coordinates to the haversine function. We turn its result into an integer value (int). The reason for this is, as we mentioned in the beginning, we are using a version of a liner optimisation algorithm that uses integers. We add the converted value to the list created at the beginning of the main cycle.

At the end of each inner loop, we create a new column in the 'cities' table, which uses the name of the currently processed city (stored in the variable 'city') as the name of the column, and we write the contents of the list in it. After that, we move on to the next city of the main loop:

```
# Calculate the distances from the currently processed city
# to all other cities. Iterate for every city
for city in cities.index.values:

    # Temp list to store the distances for the currently processed city
    current_city_distances = []
```

```
    # The loop will run as many times as cities there are
    for i in range(len(cities.index.values)):
        print(f'Processing: {city} and {cities.index.values[i]}')

        # If the currently processed city is the same as the one we are
        # calculating the distance to, make the distance exactly 0.
        # Also, make sure it is an integer and not a floting-point)
        if city == cities.index.values[i]:
            current_city_distances.append(int(0))

        # if it is a different city - calculate the distance to it
        # and store it in the temp list
        else:
            city1 = (cities['Lat'][cities.index.get_loc(city)],
                     cities['Lon'][cities.index.get_loc(city)])
            city2 = (cities['Lat'][i], cities['Lon'][i])
            # use the Haversine function on the two cities:
            current_city_distances.append(int(haversine(city1, city2)))

    # Create a new column for every city (named as the city)
    # and store in it the distances to all other cities
    cities[city] = current_city_distances
```

If all went well, the 'cities' table should contain newly created columns for all cities, with corresponding integer values for the distances between them. Let's check this out:

```
cities
```

As usual, it's a good idea to visualise our data. For this purpose, we will use the Folium and GeoPandas.

Before creating the map, however, we need to generate bounds to which we want to scale the map (i.e. how much of the global map to load visually). We do this purely for aesthetic reasons. For this purpose, we create two variables 'sw_bound' and 'ne_bound', which are respectively the south-west-most and north-east-most coordinates in our list of city coordinates.

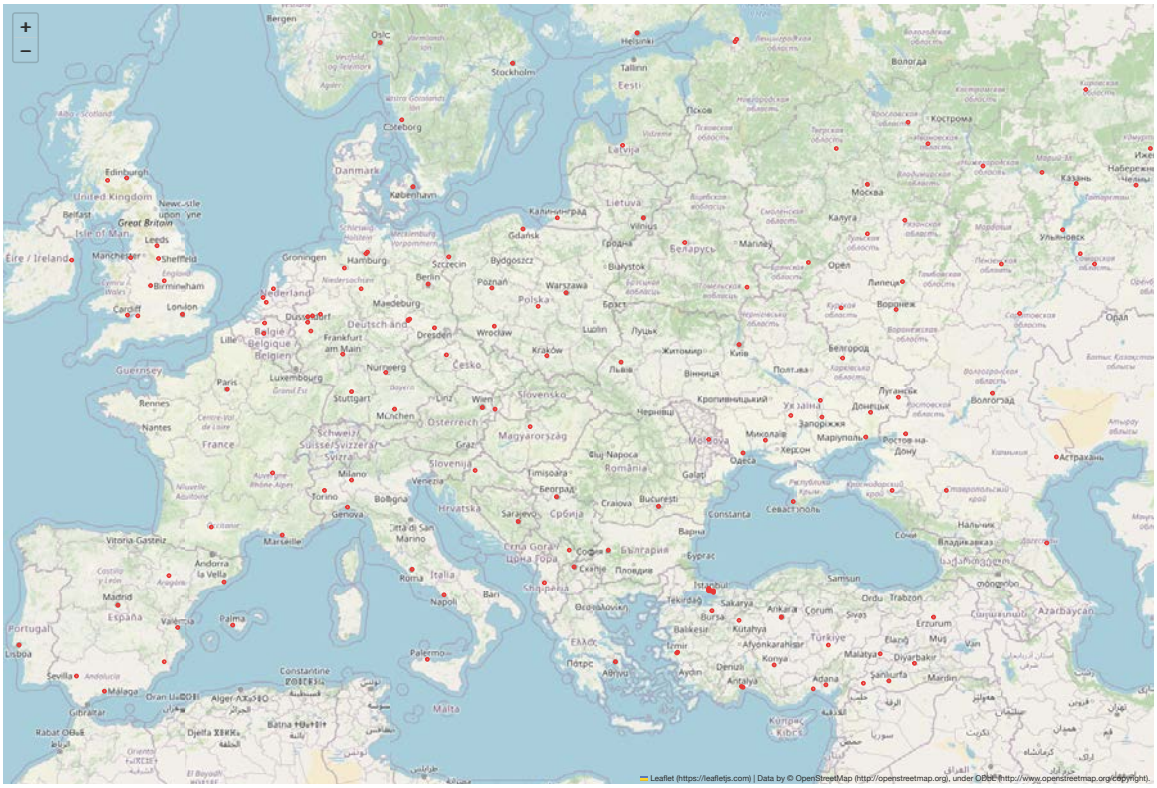
Now we can create a world map and store it in the 'world_map' variable, using the 'Map' class of the Folium library. Then we will indicate that we want the newly created map to be visually scaled so that the area within the boundaries of the southwest and northeast borders created above fits the plot area.

We then generate a new GeoPandas (gpd) GeoDataFrame structure to which we will feed our table of 150 cities. We will indicate that the objects that we want to be visualised on the map will be points with two coordinates .points_from_xy( ), and accordingly we will pass the names of the columns with the geographic coordinates to this function. Also, we will specify that we want the coordinate reference system to be EPSG Coordinate Reference System [25].

Next, we'll use the .explore( ) method on the newly created GeoPandas object, passing it the 'world_map' map to render on, and that we want the marker colour to be red:

```
# Use GeoPandas to visualise a map with the cities on it.
# Use EPSG coordinate reference system.
sw_bound = cities[['Lat', 'Lon']].min().values.tolist()
ne_bound = cities[['Lat', 'Lon']].max().values.tolist()
```

```
world_map = folium.Map()
world_map.fit_bounds([sw_bound, ne_bound])

geo_df = gpd.GeoDataFrame(cities,
geometry=gpd.points_from_xy(cities["Lon"],
                            cities["Lat"]), crs="epsg:4386")
geo_df.explore(m=world_map, color='red')
```



We are now ready to define the parameters for the optimisation task. We need three parameters:

- how many vehicles we will have (for our first example it will be only one vehicle)
- which of the locations will be the depot (start and end point)
- a distance matrix (containing the distances from any location to any other location)

```
# This is a traveling Salesman Problem, so the number of vehicles is 1.
# If we need more vehicles it will become a routing problem
vehicles = 1
```

We will choose 'Sofia' for our depot:

```
# The depot is the I/O point (which city we start from, and end in)
depot = 'Sofia'
```

Thus, since we have made the city names column to serve as index to the table, if we want to know the distance between the depot and some other city, for example Paris, we can address the cities table like so:

```
cities[depot]['Paris']
1757
```

It remains for us to separate the distance matrix into a separate variable, that we can also feed it to the optimisation algorithm. To do this, we can simply remove the columns we don't need from the 'cities' table and save the result in a new 'distance_matrix' variable:

```
distance_matrix = cities.drop(['Country Code',
                               'Country name EN',
                               'Population',
                               'Lat',
                               'Lon'], axis=1)
```

Let's see if we have what we expect in the 'distance_matrix' variable:

```
distance_matrix
```

Our next step is to set up the optimisation algorithm. First, we will create a routing manager to which we will pass the selected constraints - the number and distances between locations (via the distance matrix), the number of vehicles and the entry-exit point (the depot).

Next, we will also create the routing model, specifying the active routing manager:

```
manager = pywrapcp.RoutingIndexManager(
    len(distance_matrix),                    # Number of Nodes
    vehicles,                                # Number of vehicles
    distance_matrix.index.get_loc(depot))    # Stop point index
routing = pywrapcp.RoutingModel(manager)
```

Then, we need to define our own function to return the distance between two indices (i.e. when given the indices of two locations, it should find the locations in the distance matrix and return the distance value. We'll call the function 'distance_callback'.

The function will take as arguments the indices of the two locations, then convert the supplied indices (the next pair to be looked at) to node index numbers (i.e. those location indices in the distance matrix). The reason for this is we may have requirements for some or all locations to be visited more than once, i.e. the iteration index will not always match the location index. We will then use the converted indices (of node/location) to return the distance from the distance matrix:

```
def distance_callback(from_index, to_index):
    """Returns the distance between the two cities."""

    # Convert from routing variable Index to distance matrix NodeIndex.
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)

    return distance_matrix[distance_matrix.index[from_node]][to_node]
```

Next, we need to register the function in the routing model by calling the .RegisterTransitCallback() method and specifying the name of our function (for finding the distance between two locations). The result of this registration will be an object that we will name 'transit_callback_index':

```
transit_callback_index =
routing.RegisterTransitCallback(distance_callback)
```

We can test that the function is working correctly by calling it and passing it the indices of two points and seeing if it returns the expected result. And to show that you can't just use a simple

callback, based on the direct index input, we can call for the 150th index. Now, we only have 149 indices (since we have 150 locations and they start at index 0), so 150 would technically mean to loop back to the start.

A simpler function would have either told you there is no 150th index in the distance matrix, or it would have looped back to the 0th location in the distance matrix (which in our case is Istanbul), but because we use the conversion to node indices the 150th index will actually be the beginning of the route, which is our depot - 'Sofia'. Thus, if we call the function and request the distance between 0 and 150, we should expect to see the distance between Istanbul and Sofia, which is 501 km (according to our distance matrix). Let's see if we get that:

```
distance_callback(0, 150)
```
```
501
```

Our next step is to define the objective function, respectively the parameter for which we will search for a minimum. In our case, this is the cost of transportation, which as we mentioned at the beginning, we will assume to be the distance. In other words, the distance function will be our cost (or cost of travel) function.

Of course, as we discussed at the beginning, we can turn this into a more complex problem if the objective function includes not only the distance (for which we assume the cost per kilometre is a constant value), but also additional depreciations related to the vehicle type, speed limits, both for the vehicle and for the roads, and thus introducing a temporal element, as this will inevitably affect the completion time of each route.

But for now, we'll assume it's all about distance, and try to minimise it accordingly:

```
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

After that, we need to set the parameters of the search (the traversal of the possibility space). We will use the standard (default) search parameters by using the .DefaultRoutingSearchParameters( ) method.

```
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
```

Next, we need to define the heuristic method that will be used to find the first solution. In our case, we'll use PATH_CHEAPEST_ARC, which starts at the depot and continues with the "cheapest" (in our case, the closest next point). The method continues iteratively doing the same until it closes the route. This achieves a first solution (which is guaranteed to be a local minimum, and from which the search can be continued. Other possible strategies are described in the documentation for the routing options of ORTools.

```
search_parameters.first_solution_strategy =
(routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
```

This concludes the preparation and it is time to let the algorithm search for an optimal solution. To do this, we run the .SolveWithParameters( ) method, giving it the search parameters defined just above. We save the solution (the result of the method) in the 'solution' object:

```
solution = routing.SolveWithParameters(search_parameters)
```

When the algorithm finishes working, we can visualise the route by extracting the location indices recorded in the routing model.

For this purpose, we will first check whether we have reached a solution at all (i.e. whether there is anything written in the 'solution' object). Next, we'll create an 'index' variable that we'll make equal to the value of the first recorded index in the model (which is at position 0 in the list).

Next, we will create a variable 'route_distance' in which we will store the distance for the route. We will also create an empty 'plan' list in which we will store the ordered sequence of cities on the route. Finally, we'll create a variable 'plan_output' which we'll initialize as an empty text string, which we'll later use to generate the text we'll print out to the console.

Filling in the list with cities, calculating the distance, and generating the text string that we will print will be done in a loop. Its condition will be to run until there are no more city indices written in the 'routing' object.

Inside the loop we will first add the next city name to the empty 'plan' list. To do this, we will take the location index from the routing manager and look it up in the (index of) distance matrix. We will then append the relevant city to the 'plan_output' text string as well, along with the '->' symbols that denote the sequence.

We then move the value of the current location index into the 'previous_index' variable, and make the 'index' variable equal to the next index in the route sequence. So, within this iteration of the loop we have both a previous and a next location. Then we find the cost of moving between the two locations (which in our case is the distance between them). We do this with the .GetArcCostForVehicle( ) method, to which we pass the two indices respectively. We add the resulting distance value to the sum in 'route_distance'.

Exiting the loop, we do one final addition of a city to the 'plan' list of the route - the depot, and thus ending again at the location we started from. We also add that to the 'plan_output' string for printing.

We then print the sum of the distances that have been accumulated in 'route_distance' and print the text in 'plan_output':

```
if solution:
    index = routing.Start(0)
    route_distance = 0
    plan = []
    plan_output = ''

    while not routing.IsEnd(index):
        plan.append(distance_matrix.index[manager.IndexToNode(index)])
        plan_output +=
             f'{distance_matrix.index[manager.IndexToNode(index)]} ->'
        previous_index = index
        index = solution.Value(routing.NextVar(index))
        route_distance +=
             routing.GetArcCostForVehicle(previous_index, index, 0)

    plan.append(distance_matrix.index[manager.IndexToNode(index)])
    plan_output += distance_matrix.index[manager.IndexToNode(index)]
```

```
    print(f'Route for vehicle 0, total distance {route_distance} km.:')
    print(plan_output)
```

```
Route for vehicle 0, with total distance 31050 km.:
 Sofia -> Belgrade -> Sarajevo -> Naples -> Palermo -> Rome -> Genoa ->
Milan -> Turin -> Marseille -> Barcelona -> Palma -> Valencia -> Murcia ->
Málaga -> Sevilla -> Lisbon -> Madrid -> Zaragoza -> Toulouse -> Lyon ->
Paris -> London -> Bristol -> Cardiff -> Birmingham -> Liverpool -> Dublin
-> Glasgow -> Edinburgh -> Leeds -> Sheffield -> Leicester -> Amsterdam ->
The Hague -> Rotterdam -> Antwerpen -> Brussels -> Köln -> Düsseldorf ->
Duisburg -> Essen -> Dortmund -> Frankfurt am Main -> Stuttgart ->
Nürnberg -> Munich -> Zagreb -> Budapest -> Bratislava -> Vienna -> Prague
-> Dresden -> Leipzig -> Kleinzschocher -> Großzschocher -> Hannover ->
Bremen -> Hamburg -> Wandsbek -> Berlin -> Szczecin -> Poznań -> Wrocław -
> Łódź -> Warsaw -> Kraków -> Lviv -> Kyiv -> Homyel' -> Minsk -> Vilnius
-> Riga -> Kaliningrad -> Gdańsk -> Copenhagen -> Göteborg -> Oslo ->
Stockholm -> Helsinki -> Saint Petersburg -> Kalininskiy -> Tver ->
Yaroslavl -> Ivanovo -> Moscow -> Tula -> Ryazan’ -> Lipetsk -> Voronezh -
> Bryansk -> Kursk -> Kharkiv -> Dnipro -> Zaporizhzhya -> Kryvyy Rih ->
Mykolayiv -> Odesa -> Chisinau -> Bucharest -> Sultangazi -> Esenler ->
Bağcılar -> Bahçelievler -> Istanbul -> Üsküdar -> Umraniye -> Maltepe ->
Bursa -> Eskişehir -> Ankara -> Çankaya -> Sevastopol -> Mariupol ->
Donetsk -> Luhansk -> Rostov-na-Donu -> Krasnodar -> Stavropol’ ->
Volgograd -> Saratov -> Penza -> Ulyanovsk -> Kazan -> Cheboksary ->
Nizhniy Novgorod -> Kirov -> Izhevsk -> Naberezhnyye Chelny -> Tolyatti ->
Samara -> Astrakhan -> Makhachkala -> Erzurum -> Diyarbakır -> Malatya ->
Şanlıurfa -> Gaziantep -> Kayseri -> Adana -> Mersin -> Konya -> Muratpaşa
-> Antalya -> İzmir -> Karabağlar -> Athens -> Tirana -> Skopje ->
Pristina ->Sofia
```

We can see that the algorithm has reached a solution. The total length of the route is 31,050 km.

Of course, it would be much better to visualise the route on the map than to just leave it as a printed text string of cities. For this purpose, we need to create the necessary lines to connect the points we have previously visualised, and plot them on the map as well.

First, we will create a new empty list, which we will name 'route_lines', in which we will store the Shapely objects that we will generate.

Then, we'll use the list of cities 'plan' that we just created in a loop to retrieve the coordinates of the cities. We will use them with the .LineString( ) method to get the lines. Inside the loop, we'll check if we've reached the end of the list of cities, and if we have, we need to connect the last city back to the first. Otherwise, each currently processed city is connected to the next one in the list:

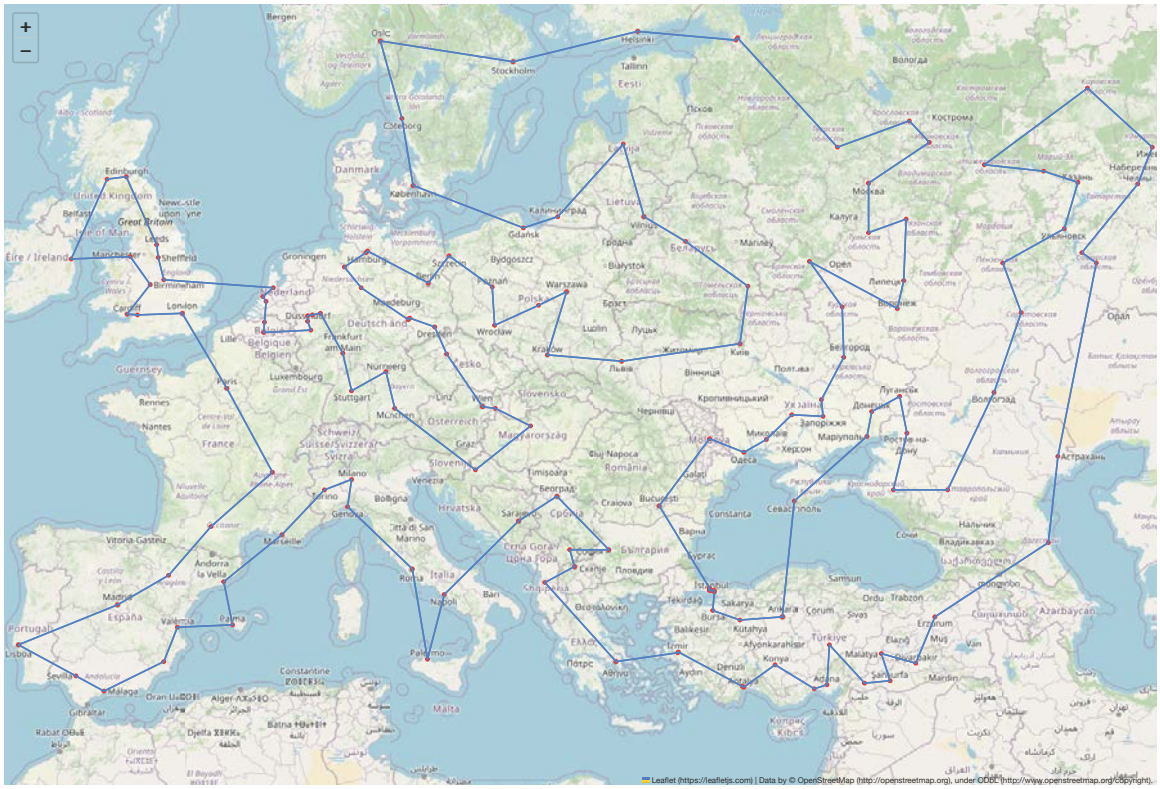
```
route_lines = []
for i in range(len(plan) - 1):

    if i == len(plan) - 1:
        route_lines.append(shapely.geometry.LineString([
                    (cities['Lon'][plan[i]],
                     cities['Lat'][plan[i]]),
                    (cities['Lon'][plan[0]],
                     cities['Lat'][plan[0]])]))
    else:
        route_lines.append(shapely.geometry.LineString([
                    (cities['Lon'][plan[i]],
                     cities['Lat'][plan[i]]),
                    (cities['Lon'][plan[i + 1]],
                     cities['Lat'][plan[i + 1]])]))
```

Similarly to Pandas, where we used to create DataFrames from lists and dictionaries, with GeoPandas we have the same functionality. We'll use this to create a GeoDataFrame named 'plan_lines' from the list of Shapely lines 'route_lines'. We'll specify that we want the lines column to be called 'geometry':

```
plan_lines = gpd.GeoDataFrame(route_lines, columns=['geometry'])
```

Again, we'll use the .explore( ) method to look at the lines, saying we want them to be plotted on top of the 'world_map' we've already generated in the beginning of the exercise (containing the locations of the cities):

```
plan_lines.explore(m=world_map)
```



Looking at the route visualised on the map, we must again take into account the fact that this is an idealised version of the problem, in which we have assumed that the locations are connected by a straight line. Here we do not take into account the fact that the roads are not straight, and that to reach some of the locations you have to circle around large volumes of water, or there is no land connection at all.

As we commented at the beginning of the topic, the traveling salesman problem is a special case in the general routing problem, where there are more than one vehicles involved. But what does it even mean to look for an optimal route for multiple vehicles. A possible answer would be that an optimal solution would be such that the sum of the distance of the routes of all vehicles is the smallest. But, if we ignore any other constraints (which we didn't have for the last example anyway), the optimal solution with such an approach will effectively be to visit all locations with just one vehicle (which effectively gets us back to the special case of the traveling salesman).

A better way to define optimal routes would be to search for the minimum of the longest single route (per vehicle), among all vehicles. Essentially, this means that we try to reach all destinations as quickly as possible (assuming our speed is constant), using all vehicles, while wanting each vehicle to travel the least possible amount of distance.

Here we need to define how many vehicles we will use for this example. I should note that while in our first exercise, with only one vehicle and 150 locations, the optimum was found relatively quickly, here, in a more generalised problem with more vehicles, the optimisation process will start to take much longer time for finding an optimum. So, note that 150 locations and 6 vehicles may you're your computer quite a few minutes to reach an optimal solution. Consider carefully how many locations and vehicles you will make this net exercise with. I will place 6 vehicles, as well as change the depot location to be the city of Paris:

Exercise IX.1-2: Use Python to find an optimal solution to a routing problem.

Our task here is similar to the previous exercise, but we will generalise the problem, so that instead of only one vehicle we will use 6. We will keep the number of cities the same (150).

We will use the same libraries as with the previous exercise, so I will not explain them again.

```
import random
import pandas as pd
import geopandas as gpd
import shapely
import folium
from haversine import haversine
from ortools.constraint_solver import pywrapcp, routing_enums_pb2
```

Since we will be using the same dataset and the same 150 cities (filtered by highest population), we will just copy the same code as we used in the previous exercise.

First read the data file, then sort it by population. Take the first 150 cities and then drop the columns we won't need:

```
all_cities = pd.read_csv('data/geonames-all-cities-with-a-population-
1000.csv', index_col='Name', delimiter=';')
all_cities.sort_values(by=['Population'], ascending=False, inplace=True)
cities = all_cities.head(150)
cities = cities.drop(['Geoname ID',
                      'ASCII Name',
                      'Alternate Names',
                      'Feature Class',
                      'Feature Code',
                      'Country Code 2',
                      'Admin1 Code',
                      'Admin2 Code',
                      'Admin3 Code',
                      'Admin4 Code',
                      'Elevation',
                      'DIgital Elevation Model',
                      'Timezone',
                      'Modification date',
                      'LABEL EN'], axis=1)
```

We have to again split the coordinates string in order to get the two parts (the Latitude and the Longitude) and store them in separate columns. Then drop the redundant 'Coordinates' column:

```
cities['Lat'] = [
     float(coord.split(',')[0]) for coord in cities['Coordinates']]
cities['Lon'] = [
     float(coord.split(',')[1][1:]) for coord in cities['Coordinates']]
cities = cities.drop(['Coordinates'], axis=1)
```

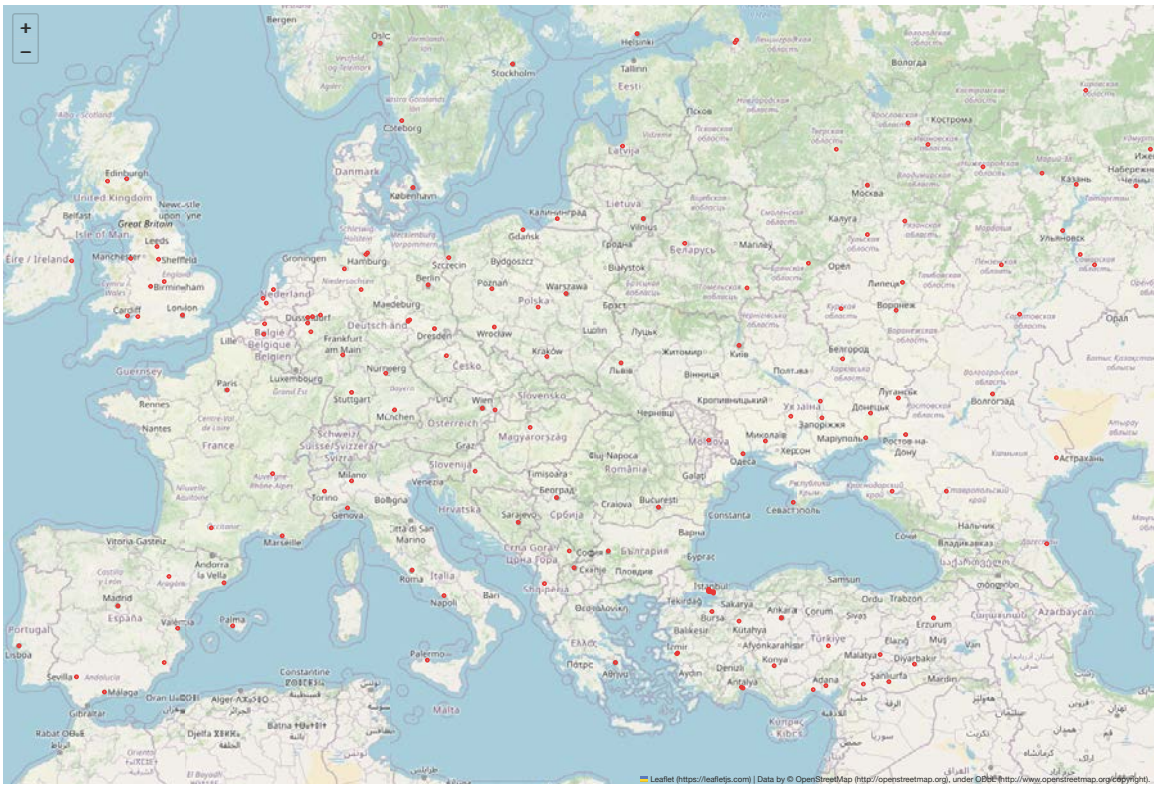
We will again need to construct a distance matrix for all cities, so we will just copy the code from the previous exercise:

```
# Calculate the distances from the currently processed city
# to all other cities. Iterate for every city
for city in cities.index.values:

    # Temp list to store the distances for the currently processed city
    current_city_distances = []

    # The loop will run as many times as cities there are
    for i in range(len(cities.index.values)):
        print(f'Processing: {city} and {cities.index.values[i]}')

        # If the currently processed city is the same as the one we are
        # calculating the distance to, make the distance exactly 0.
        # Also, make sure it is an integer and not a floting-point)
        if city == cities.index.values[i]:
            current_city_distances.append(int(0))

        # if it is a different city - calculate the distance to it
        # and store it in the temp list
        else:
            city1 = (cities['Lat'][cities.index.get_loc(city)],
                     cities['Lon'][cities.index.get_loc(city)])
            city2 = (cities['Lat'][i], cities['Lon'][i])
            # use the Haversine function on the two cities:
            current_city_distances.append(int(haversine(city1, city2)))

    # Create a new column for every city (named as the city)
    # and store in it the distances to all other cities
    cities[city] = current_city_distances

distance_matrix = cities.drop(['Country Code',
                               'Country name EN',
                               'Population',
                               'Lat',
                               'Lon'], axis=1)
```

Let's visualise the cities on the map again, to make sure we are starting from the same base as before:

```
# Use GeoPandas to visualise a map with the cities on it.
# Use EPSG coordinate reference system.
sw_bound = cities[['Lat', 'Lon']].min().values.tolist()
ne_bound = cities[['Lat', 'Lon']].max().values.tolist()

world_map = folium.Map()
```

```
world_map.fit_bounds([sw_bound, ne_bound])

geo_df = gpd.GeoDataFrame(cities,
geometry=gpd.points_from_xy(cities["Lon"],
                            cities["Lat"]), crs="epsg:4386")
geo_df.explore(m=world_map, color='red')
```



As with the previous example, our next step is to set up the optimisation algorithm. Again, we create a routing manager to which we will pass the necessary constraints - the number and distances between locations (using the distance matrix), the number of vehicles and the I/O location (the depot).

Next, we will also create the routing model, specifying the active routing manager:

```
fleet_manager = pywrapcp.RoutingIndexManager(
    len(distance_matrix),                          # Number of nodes
    fleet_vehicles,                                # Number of vehicles
    distance_matrix.index.get_loc(fleet_depot))    # Stop index
fleet_routing = pywrapcp.RoutingModel(fleet_manager)
```

For this example, we'll need to create a new function to find the distances between locations, since trying to register the old one in the new routing model (at least for me) crashes the Python core. In practice, we'll just copy the contents of the old function, but change the name to 'fleet_distance_callback'.

Then, again, we need to register this function in the new routing model by calling the .RegisterTransitCallback( ) method and specifying its name. We will name the result of this registration 'fleet_transit_callback_index'.

```
def fleet_distance_callback(from_index, to_index):
    """Returns the distance between the two cities."""
```

```
    # Convert from routing variable Index to distance matrix NodeIndex.
    from_node = fleet_manager.IndexToNode(from_index)
    to_node = fleet_manager.IndexToNode(to_index)
    return distance_matrix[distance_matrix.index[from_node]][to_node]

fleet_transit_callback_index =
            fleet_routing.RegisterTransitCallback(fleet_distance_callback)
```

Again, we need to set the objective function and pass it the 'fleet_transit_callback_index' we just recreated:

```
fleet_routing.SetArcCostEvaluatorOfAllVehicles(
                                    fleet_transit_callback_index)
```

Unlike the previous exercise, where we didn't have any constraints, for this example let's introduce conditions on the distance that individual vehicles can travel. Adding constraints is done by introducing additional dimension for the optimising algorithm.

First, we will define the name of the dimension (constraint). The variable will be named 'dimension_name' and will contain the actual name: the text string 'Distance'. We will then add this dimension to the model using the .AddDimension( ) method. The 'fleet_transit_callback_index' argument is the previously defined index that we got when registering the function that returns the distances between locations.

In addition to the index, the .AddDimension() method has the following arguments:

- **slack_max**: Maximum slack for each location. I.e., whether upon arrival the vehicle immediately leaves for the next location. In our case, this would be the maximum idle time (e.g., loading, unloading, driver rest, etc.) for each location. For our example we'll make it 0, but in reality, there will always be some range of slack for every condition we add as a dimension.
- **capacity**: Capacity is the maximum value for the relevant dimension (added condition) that can be accumulated on the road for any single vehicle. In our case, this will be the value indicating the maximum distance we give a vehicle to travel. For our case, we will set this value at 10,000 km.
- **fix_start_cumulative_to_zero**: A Boolean value that indicates whether the accumulation of the added dimension starts from zero or not. In our case it will be True because starting from the depot the vehicles will not have travelled any distance. But for other constraints this may not be true. For example, if we're talking about load capacity, maybe the vehicles don't start empty from the depot, or if we have time constraints, we might want to start after a certain time value, etc.
- **dimension_name**: A text value that specifies the name of the dimension (constraint) we could refer to. In our case, we pass the variable 'dimension_name', which we made above to contain the necessary name as a text (string) value.

```
# Add Distance constraint.
dimension_name = 'Distance'
fleet_routing.AddDimension(fleet_transit_callback_index,
                           0,      # no slack
                           10000,  # vehicle maximum travel distance
                           True,   # start acumulating distance from zero?
                           dimension_name)
```

Once we have defined this constraint, we can refer to it by name and give it a coefficient.

We can create a reference to it by using the .GetDimensionOrDie() method, passing the name of the constraint as an argument, which in our case is 'dimension_name'. If a constraint with such a name does not exist, it will stop the execution of our script (hence ‘die’).

The coefficient we set defines what part of the cost is related to this constraint. As always, when we talk about weights (coefficients) we consider them as the level of importance (the impact) we want to give to this constraint in the search for the optimum.

Since this is the only constraint for us, we can say that it has a weight of 100%. Setting the coefficient itself is done with the .SetGlobalSpanCostCoefficient( ) method. This will make the global range of distances (the maximum of the final value of the dimension minus the minimum of the starting value of the dimension) for that dimension the dominant factor in the optimisation of the objective function:

```
distance_dimension = fleet_routing.GetDimensionOrDie(dimension_name)
distance_dimension.SetGlobalSpanCostCoefficient(100)
```

We have to set the search parameters (again we will use the default ones) and define the first solution strategy:

```
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy =
(routing_enums_pb2.FirstSolutionStrategy.PATH_CHEAPEST_ARC)
```

We can proceed to the search for a solution. I remind you that depending on the processing power of your computer, this task can take a considerable amount of time. We will store the optimisation result in the newly created variable 'fleet_routing_solution':

```
# Solve the problem.
fleet_routing_solution = fleet_routing.SolveWithParameters(
                                                    search_parameters)
```

As with the previous exercise, we will also compose a loop to print the found optimal solution to the console. Conceptually there is no difference in the approach. The only significant difference is that instead of a single vehicle here we have to compile the separate routes of each of the vehicles. This means we have two loops, the first one iterates over the vehicles and the inner loop iterates over the route locations for the corresponding vehicle, the same way we did it with the traveling salesman problem:

```
# Print solution
if solution:
    max_route_distance = 0
    fleet_plan = []
    total_distance = 0

    for vehicle_id in range(fleet_vehicles):
        index = fleet_routing.Start(vehicle_id)
        route_distance = 0
        fleet_plan_output = ''
        vehicle_plan = []
        while not fleet_routing.IsEnd(index):
            vehicle_plan.append(distance_matrix.index[
                  fleet_manager.IndexToNode(index)])
            fleet_plan_output += f' {distance_matrix.index[
```

```
                fleet_manager.IndexToNode(index)]} ->'
            previous_index = index
            index = fleet_routing_solution.Value(
                fleet_routing.NextVar(index))
            route_distance +=
                 fleet_routing.GetArcCostForVehicle(previous_index,
                                                    index, vehicle_id)
        fleet_plan_output += ' ' + distance_matrix.index[
                 fleet_manager.IndexToNode(index)]
        max_route_distance = max(route_distance, max_route_distance)
        fleet_plan.append(vehicle_plan)
        total_distance += max_route_distance

        print(f'Vehicle {vehicle_id}, distance {route_distance} km.:')
        print(fleet_plan_output)
        print('\n')

    print(f'The longest single vehicle route is {max_route_distance} km,
the total distance for all routes is: {total_distance} km')

else:
    print('No solution found !')
```

```
Route for vehicle 0, with distance 7481 km.:
 Paris -> Lyon -> Marseille -> Toulouse -> Barcelona -> Zaragoza -> Madrid
-> Lisbon -> Sevilla -> Málaga -> Murcia -> Valencia -> Palma -> Palermo -
> Naples -> Dresden -> Berlin -> Copenhagen -> Amsterdam -> The Hague ->
Rotterdam -> Antwerpen -> Brussels -> Paris

Route for vehicle 1, with distance 7708 km.:
 Paris -> London -> Bristol -> Cardiff -> Birmingham -> Leicester ->
Sheffield -> Leeds -> Liverpool -> Dublin -> Glasgow -> Edinburgh -> Oslo
-> Göteborg -> Stockholm -> Helsinki -> Saint Petersburg -> Kalininskiy ->
Tver -> Moscow -> Ryazan’ -> Lipetsk -> Tula -> Bryansk -> Minsk ->
Vilnius -> Kaliningrad -> Gdańsk -> Szczecin -> Hannover -> Dortmund ->
Essen -> Duisburg -> Düsseldorf -> Paris

Route for vehicle 2, with distance 7718 km.:
 Paris -> Nürnberg -> Vienna -> Bratislava -> Budapest -> Chisinau ->
Sevastopol -> Krasnodar -> Stavropol’ -> Makhachkala -> Astrakhan ->
Saratov -> Voronezh -> Kursk -> Homyel' -> Warsaw -> Łódź -> Wrocław ->
Leipzig -> Kleinzschocher -> Großzschocher -> Köln -> Paris

Route for vehicle 3, with distance 7750 km.:
 Paris -> Bremen -> Hamburg -> Wandsbek -> Riga -> Yaroslavl -> Ivanovo ->
Nizhniy Novgorod -> Cheboksary -> Kirov -> Izhevsk -> Naberezhnyye Chelny
-> Kazan -> Ulyanovsk -> Tolyatti -> Samara -> Penza -> Poznań -> Paris

Route for vehicle 4, with distance 7042 km.:
 Paris -> Milan -> Sarajevo -> Pristina -> Skopje -> Sofia -> Bucharest ->
Odesa -> Mykolayiv -> Kryvyy Rih -> Dnipro -> Zaporizhzhya -> Mariupol ->
Rostov-na-Donu -> Volgograd -> Luhansk -> Donetsk -> Kharkiv -> Kyiv ->
Lviv -> Kraków -> Prague -> Frankfurt am Main -> Paris

Route for vehicle 5, with distance 7601 km.:
 Paris -> Turin -> Genoa -> Rome -> Tirana -> Athens -> Karabağlar ->
İzmir -> Antalya -> Muratpaşa -> Konya -> Mersin -> Adana -> Gaziantep ->
Şanlıurfa -> Diyarbakır -> Erzurum -> Malatya -> Kayseri -> Çankaya ->
Ankara -> Eskişehir -> Bursa -> Maltepe -> Umraniye -> Üsküdar -> Istanbul
-> Bahçelievler -> Bağcılar -> Esenler -> Sultangazi -> Belgrade -> Zagreb
-> Munich -> Stuttgart -> Paris
```

```
The longest single vehicle route is 7750 km, the total distance for all
routes is: 46157 km
```

If we want to visualise the routes of individual vehicles on the map with cities, we will need to compose lines (Shapely objects) for them. Again, our approach is the same as for the previous exercise, with the difference that we will use two loops. One will iterate the vehicles and the other the location names in the list:

```
fleet_route_lines = []
for veh_plan in fleet_plan:
    veh_route = []

    for i, node in enumerate(veh_plan):
        # If it is the last node on the list ->
        # loop back to the first (0 index) node to close the route
        if node == veh_plan[-1]:
            veh_route.append(shapely.geometry.LineString([
                 (cities['Lon'][node],
                  cities['Lat'][node]),
                 (cities['Lon'][veh_plan[0]],
                  cities['Lat'][veh_plan[0]])]))
        else:
            veh_route.append(shapely.geometry.LineString([
                 (cities['Lon'][node],
                  cities['Lat'][node]),
                 (cities['Lon'][veh_plan[i + 1]],
                  cities['Lat'][veh_plan[i + 1]])]))
    fleet_route_lines.append(veh_route)
```

The part of the script that we'll use to visualise the routes is a little more complex than the first example, since, for clarity, we want the individual routes to be in different colours. For this we will need to create a list of colours (as many as there are vehicles).

First let's see what colours does Folium support. We can find them in the .color_options property of the .Icon class of the library. If we pass that to a list() function we can easily print them out:

```
print(list(folium.Icon.color_options))
['blue', 'green', 'beige', 'black', 'darkgreen', 'pink', 'darkblue',
'white', 'darkred', 'darkpurple', 'orange', 'lightred', 'purple',
'lightgreen', 'gray', 'lightblue', 'lightgray', 'cadetblue', 'red']
```

I will choose 6 of them that I believe will have enough contrast on top of the map we want to visualise the lines on:

```
colours = ['blue', 'black', 'darkgreen', 'orange', 'purple', 'red']
```

We will create a new 'world_map' containing only the city points, then again, we will constrain the map's zoom level to span between the south-west-most and north-east-most coordinates in our list of city coordinates. Again, using the .explore( ) method, we will plot the cities from the 'geo_df' GeoDataFrame onto the created 'routing_map', making the markers red.

We will then need a loop to generate a child object on the map for each vehicle's route, respectively each set of lines in the 'fleet_route_lines' list, in the corresponding colour. For this purpose, we first add the child objects to a newly generated empty list 'lines_df', containing a subset of the 'geo_df' structure. To generate these sub-sets of locations we use the .query( )

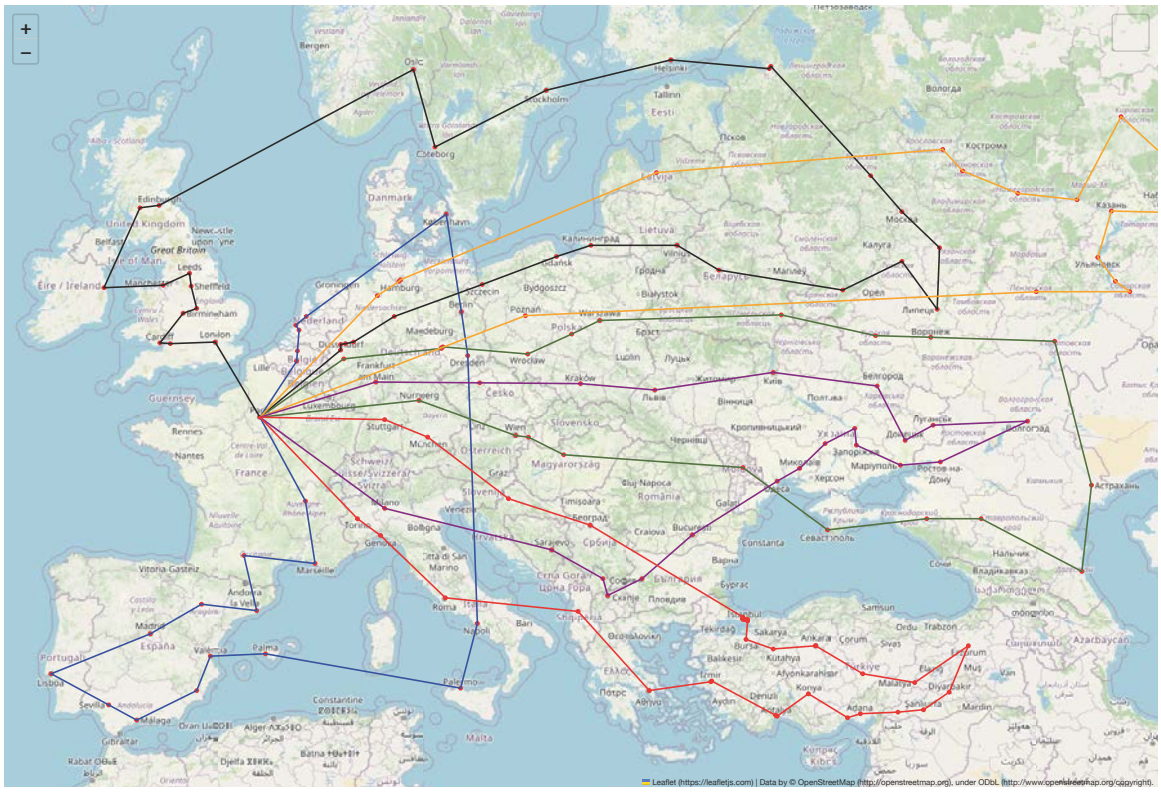
method, where as an argument we want it to search where in the index of 'geo_df' we can find the the cities from the corresponding 'fleet_plan' list. The geometry that we want to visualise on the map will come from the corresponding elements that matches those in the 'fleet_route_lines'.

Then we tell the next 'lines_df' element, that we just appended, to be rendered on the 'routing_map' (using the .explpore( ) method), and for the colour we take the next element of the 'colours' list.

Finally, we also add an interactive .LayerControl( ) element that allows individual child objects to be turned on and off on the map, using the .add_to( ) method, giving our map as an argument. The button for this layer control function is in the upper right corner of the map.

On the last line, we call the 'routing_map' variable containing the map to render it on the screen:

```
routing_map = folium.Map()
routing_map.fit_bounds([sw_bound, ne_bound])
geo_df.explore(m=routing_map, color='red', name='Locations')

lines_df = []
for i in range(len(fleet_route_lines)):
    lines_df.append(gpd.GeoDataFrame(geo_df.query(
                                        f'index in @fleet_plan[{i}]'),
                                        geometry=fleet_route_lines[i],
                                        crs="epsg:4386"))
    lines_df[i].explore(m=routing_map, color=colours[i], name=f'Line {i}')

# this is completely optional
folium.LayerControl().add_to(routing_map)
routing_map
```



Chapter X.
Decision-making

Since we are dealing with intelligent control systems, we have to pay at least some attention to their decision-making capabilities. Much like us, an automated control system will also be limited in the amount of knowledge it has (die to constraints, approximations and so on), which inevitably means making decisions in a state of uncertainty. We humans tend to rely on our intuition in such cases, which by definition is build based on our past experiences and our ability to make analogies based on similarity. If you think about it, our goal with machine learning and intelligent systems in general, is to achieve the same.

But regardless of how that decision intuition is achieved, defining it into a consistent model is paramount for the success of the control application. One of the main reasons for it is the ability to compare decisions and analyse results based on them, by having the consistency of the inner logic of the decision-making system.

There are many decision-making algorithms available for various tasks (and depending on the type of input data), like Decision trees, Markov Decision Process, Bayesian Networks and Fuzzy Logic Inference. All of them deal with the probabilistic nature of decision making in one way or another.

The one that we choose to include as an exercise here is Fuzzy logic inference, because of the closeness to human reasoning, allowing for degrees of truth rather than binary true/false values. Its simplicity and ease of understanding make it an excellent starting point for students.

X.1. Fuzzy logic and fuzzy inference systems

Often, when building control solutions, we have to acknowledge that it is people that will be working with them, and also, often those people might not be particularly technically adept. Building control solutions, that use human language as a method of communication may require you to deal with uncertainty, that comes with language interpretation.

Just to understand what I am talking about, before we continue with the exercise, let's conduct a brief experiment. Please, fill in the short survey at this link and fill in the answers. We can discuss the results in class. But the general idea is that language is fuzzy. If someone says they are cold that does not directly translate to a specific temperature value. Even more, different people will have different understanding of what temperature constitutes "cold".

When building fuzzy control systems, we usually have to rely on established rules, where some collective knowledge has been synthesised and encoded in. For example, a rule might be something like "If it is **cold** outside, put on **warm** clothes". Now we can use that rule as a prediction model. If we want to know if someone will have put warm clothes, we just need to know if the temperature outside is cold. But the use of such linguistic values (belonging to the linguistic variable "Temperature") immediately introduces uncertainty. And by linguistic variable we understand a variable whose values are not numerical but linguistic (i.e., words or expressions) [26]. Since we already started with the temperature example, let’s imagine a room environment management system that uses a linguistic variable such as "Temperature".

That variable will have values that are words, for example "cold" or "warm", instead of numeric values such as 15, 20, 30 or 40 [°C]. And while this might complicate the work of the engineers creating the systems, it may make communication with the system much more natural and understandable for the people who have to use it.

But then we have to ask ourselves, what do these linguistic values really mean. What does it mean for the temperature outside to be **cold**? We could set an exact threshold value, let's say 18 degrees Celsius, and insist that any temperature bellow it is cold, and any temperature above it is warm. But that is binary logic, which implies that 17.99 C^o is definitely cold and 18.01 C^o is definitely warm.

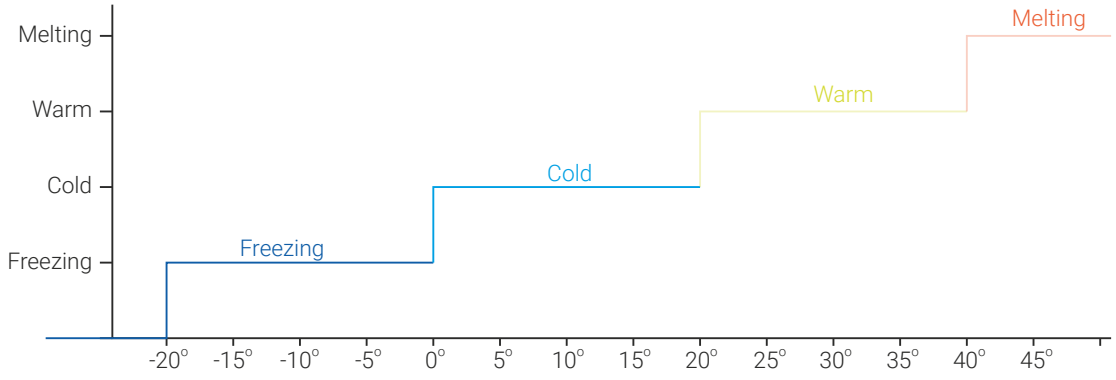


Figure 83. Graphical representation of a binary division of the temperature space in four linguistic variables.

It implies that an infinitely small step in either direction, crossing the set threshold, somehow completely changes our interpretation of the state of the environment. And that is obviously not true.

This is where fuzzy logic is very useful. When we have to work with fuzzy values (such as words), where we don't have a specific (crisp) value associated with the term. Unlike Boolean mathematics and classical logic, in fuzzy logic the truth of a statement is not binary - True vs. False, or 1 vs. 0, or for the example on the following figure, "cold" vs. "hot". Rather, there are degrees of truth for each value, which can vary between 0 and 1, where 0 mean completely false (or belonging 0% to that statement) and 1 is completely true (or 100% belonging, or we can also say, being a member of this statement).

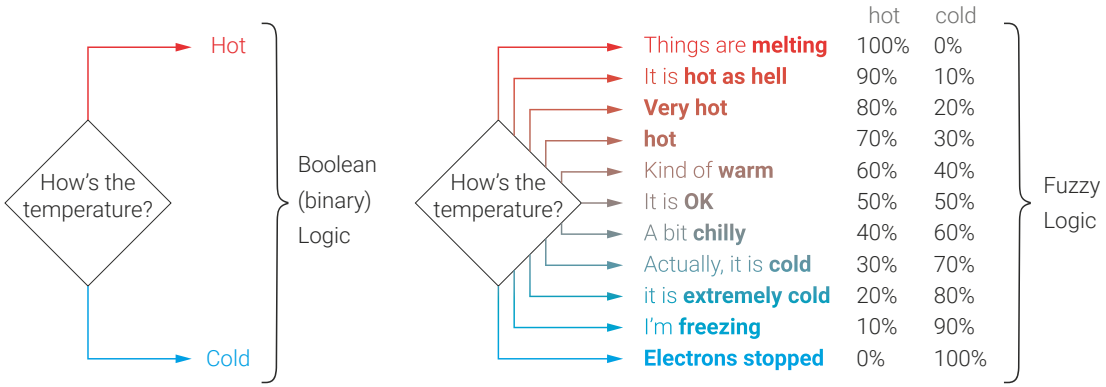


Figure 84. Graphical representation of the difference between binary and fuzzy linguistic variables and their membership.

If we take temperature as an example, a temperature of 20 degrees Celsius may belong to some extend to the "cold" value, but it also is much strongly associated with other linguistic

values. Most certainly it is under the "warm" temperature set for a lot of people. We can say that each linguistic variable (such as "Temperature") has one or more membership functions, which define to what extend each crisp value in its Universe (in our example that would be the range of possible temperatures) belongs to its linguistic values, such as "cold" or "warm". If we assume a normal distribution in such a membership function, we could have something like this:

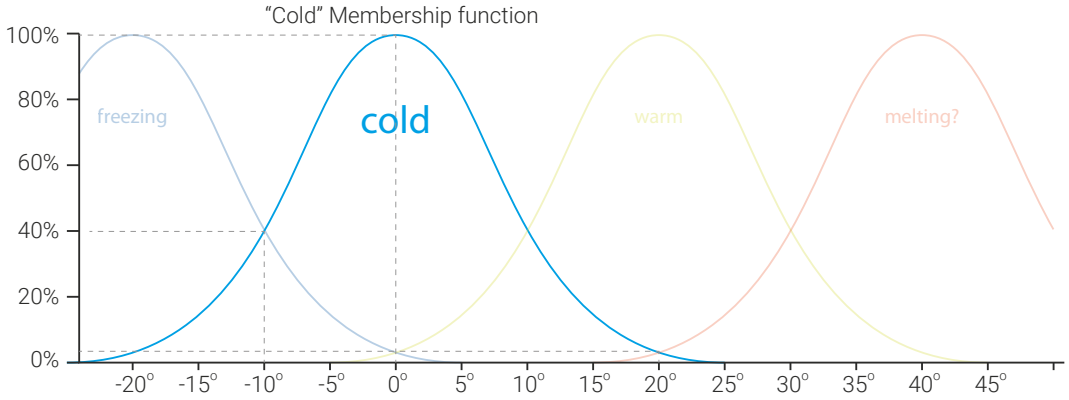


Figure 85. Graphical representation of a Gaussian membership functions of several linguistic variables.

In the above figure $0\ C^o$ belongs 100% in the "cold" set, but also about 5% in the "warm" set. $-20\ C^o$ is only 5% in the "cold" set, but it is 100% in the "freezing" set. And oppositely, $20\ C^o$ also belongs 5% in the "cold", but 100% in the "warm" set. Obviously, we can have more or less granular division of the universe of values, in terms of linguistic variables, used to describe that space.

Fuzzy logic allows us to build fuzzy inference systems, which use human interpretable rules to convert input to output. Unlike previous algorithms, which were aimed at creating a model, either by first principles, or by fitting a function to some known data, fuzzy inference doesn't need a model because it uses human set rules. It is very useful in complex processes, where the underlying mechanics of the process are not known, or there is not enough data to fit a model. As long as we have good intuition, or there are established rules, that map the fuzzy inputs and outputs, then we can derive them, or directly embed them in the inference system.

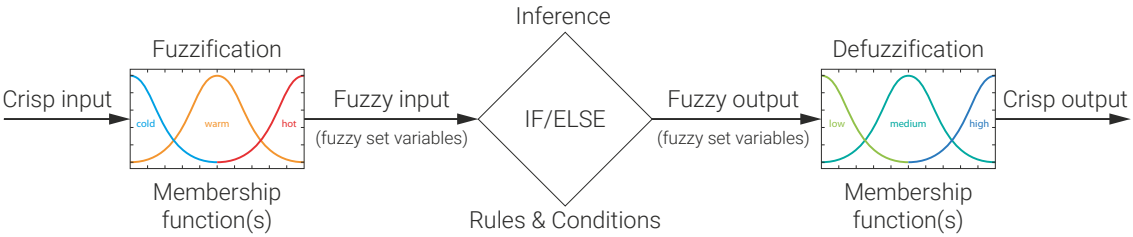


Figure 86. An abstraction of a fuzzy control system.

Exercise X.1-1: Use Python to construct a fuzzy logic inference system that suggests what to clothes to wear, based on a fuzzy temperature description.

The main library we will use for this exercise is skfuzzy (scikit-fuzzy), which is an extension of Sklearn. It contains tools and functions for working with fuzzy logic.

You are already more or less familiar with Numpy and Matplotlib. Besides matplotlib, for this exercise we will use another visualization library - Plotly. We could do without it, but Matplotlib doesn't handle 3D plots very well, or at least not when they need to be interactive.

Cycler is a specific class, part of Matplotlib, that contains a function of the same name to allow cycling through combinations of values (usually in the form of dictionaries). Used by matplotlib's plotting functions to colour the plot components. We will use it for the same purpose, in order to replace the default colours that Matplolib will use with some custom colours of our choice.

```
import numpy as np
import plotly.graph_objects as go
import matplotlib.pyplot as plt
import skfuzzy as fuzz
from skfuzzy import control as ctrl
from cycler import cycler
```

Since we are going for a temperature control example, we will have to create the input and output universes of the system. In fuzzy logic those are referred to as Antecedent and Consequent. For our case, let's assume there are two inputs, or two Antecedents - the room temperature and how well are we dressed. And there is one Consequent - the temperature control setting of the heater.

First, we have to define their universe, i.e. what is the range of possible (crisp) values that they can have. For the room temperature we will assume it can be between 5 and 35 degrees Celsius and our measurement precision is 1 degree. For creating that set, which we will then have to fuzzify, we can use the np.arange() method, giving it a start at 5, an end of 36 and a step of 1.

Similarly, we define the amount of possible clothing we have on us. Let's use some scale where each layer of clothing we have on us adds 1 to the score. Let's assume the maximum number of layers we can have is 4. Thus, for the np.arange() we use a start of 0, an end of 5 and a step of 1.

The output is the temperature control setting for the heater. We will assume it we have available an air conditioning unit that can be set to a temperature between 15 and 35 degrees Celsius. We also assume the precision of the heater thermostat is 0.5 degrees.

```
# New Antecedent/Consequent objects hold
# universe variables and membership functions
room_temp = ctrl.Antecedent(np.arange(5, 36, 1), 'room temperature')
clothing = ctrl.Antecedent(np.arange(0, 5, 1), 'amount of clothing')
temp_control = ctrl.Consequent(np.arange(15, 35.5, 0.5),
                                          'temperature control')
```

In the example I used for the introduction, we were talking about normal distribution, or Gaussian distribution of the membership. Of course, the distribution or the type of membership function will entirely depend on the specific variable we have to work with, and how we will extract knowledge about it from the real world. The membership functions can be triangular, trapezoid or any other distribution that can be formulated.

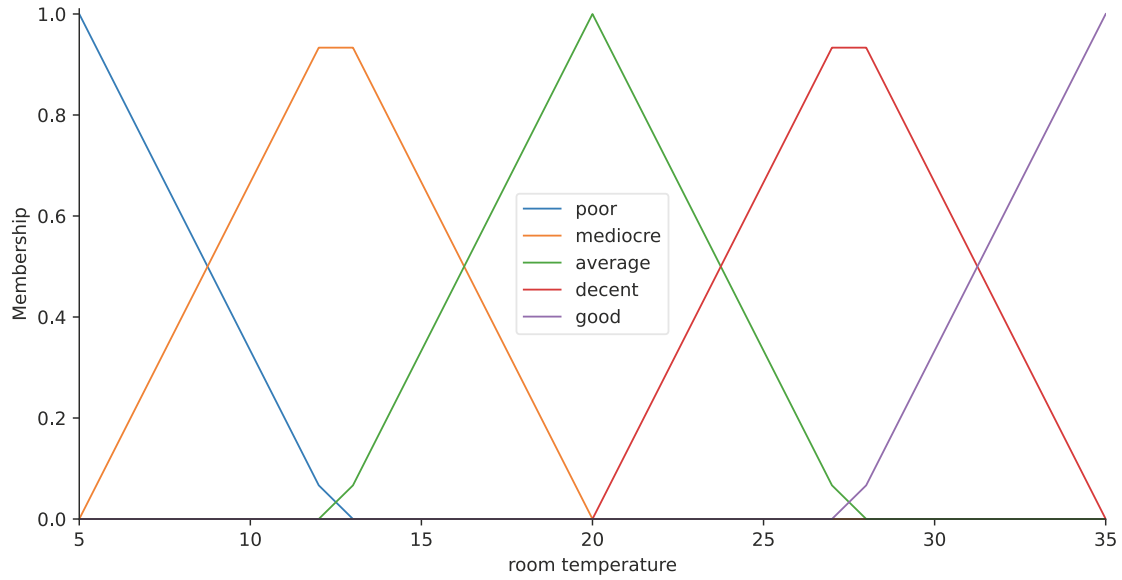
Skfuzzy allows membership functions to be defined in an automatic way (by default using a triangular function), and the defined range of the universe of values is then evenly used to cover the bases of the triangles. The number of triangles is dictated by the number of fuzzy (linguistic) values that we will use to describe the conditions of the system, and also use for the definition of the fuzzy rules.

If no specific values are provided, skfuzzy will use its own predefined set. Let's try it out:

```
room_temp.automf(5)
```

By passing the input 5, we are saying we want 5 linguistic values (fuzzy terms) to divide the universe of room temperatures in. Since we did not supply names for those terms, skfuzzy will use the default set of terms. Let's visualise the first antecedent and see what the universe looks like. For that we will use its .view( ) method:

```
room_temp.view()
```



We can see skfuzzy divided the range so that there are 5 peaks (at 5, 12.5, 20, 17.5 and 35 degrees). We can also see that the linguistic terms (values) used are:

['poor', 'mediocre', 'average', 'decent', 'good'],

which are part of the default set of 7 linguistic variables, which skfuzzy uses. Obviously, those are not particularly compatible (linguistically speaking) with describing temperature. So, let's define our own fuzzy (linguistic) values. Just to demonstrate how we can tell the autogenerator of membership functions to divide the universe into more states, and to use custom terms, let's go with 11 terms to describe temperature:

```
# Custom linguistic values, used for describing the room temperature
temp_qualitative_values = ['absolute zero',
```

```
                                'freezing',
                                'extremely cold',
                                'cold',
                                'chilly',
                                'OK',
                                'warm',
                                'hot',
                                'very hot',
                                'hot as hell',
                                'melting']
```

Now we can use the same .automf( ) function, but we will say we want 11 states, and the names are to be taken from the list that we just defined:

```
# Auto-membership function population is possible
# with .automf(3, 5, or 7), but we will use our
# 11 custom variables for this Antecedent
room_temp.automf(11, names=temp_qualitative_values)
```

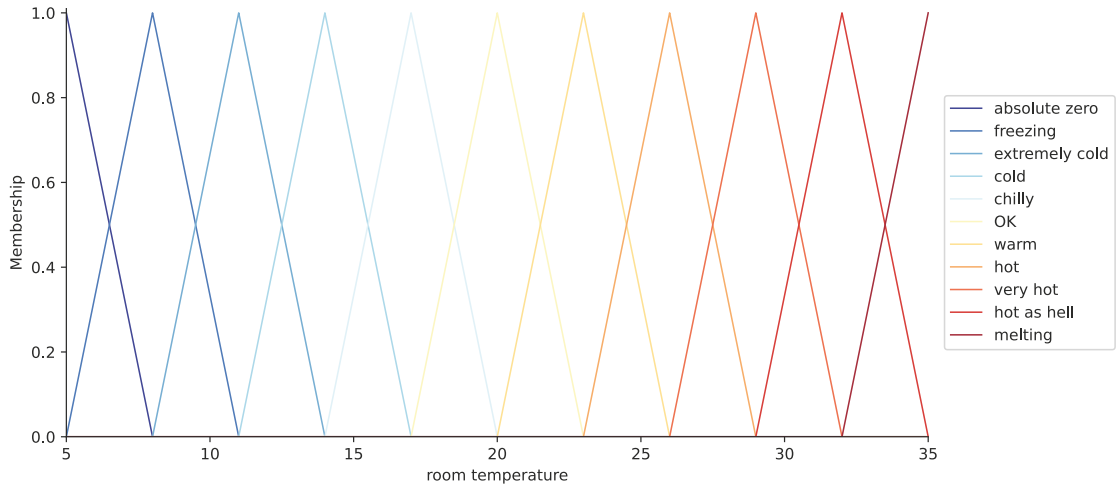
To see what have we done, we can again call the .view( ) method. But as we saw the previous time we called it, it used some random colours for the lines of the 5 triangles. Considering it is temperature we are discussing, it would be appropriate if we use a more meaningful colour scale to represent the membership functions (and the respective linguistic values).

In order to do that, since the visualization package of skfuzzy does not give us a direct option to change the colours, or select custom colormaps, we will have to be a bit more creative.

First, we will change the runtime configuration of Pyplot's colour cycler to use a different colour map. The 'RdYlBU' stands for 'RedYellowBlue'. The '_r' after the colour map name is used to tell matplotlib to reverse the order of the colours. Essentially, we want to use the colours from blue to red to convey the idea of going from cold to hot. And since we are addressing pyplot for the colourmap, we can use the opportunity to also change the size of the displayed figure.

At this time we could just use .view( ) method on the 'room_temp', and you can very well just do so. But because that will place the legend on top of the plot, and we would rather have it outside the plot, we would need to address the axes object of the underlying matplotlib plot, in order to make changes. To do that we can first create a FuzzyVariableVisualizer object, and then using the .view( ) method on it, which will expose the axis and the figure objects to us. We can then easily tell the axis to place the legend outside the border of the plot, anchoring it to the 1.01 point in the x direction and 0.5 in the y direction (those are normalised coordinates):

```
plt.rc('axes',
       prop_cycle=cycler(
              color=plt.get_cmap('RdYlBu_r')(np.linspace(0, 1,
                     len(temp_qualitative_values)))))
plt.rcParams['figure.figsize'] = [10, 5]

room_temp_viz = ctrl.visualization.FuzzyVariableVisualizer(room_temp)
fig, ax = room_temp_viz.view()

ax.legend(loc='center left', bbox_to_anchor=(1.01, 0.5))
```



As we mentioned, by default skfuzzy uses triangular membership functions, but there are other types, that can be used (as described in the documentation [27]).

For simplicity let's fall back to just 3 options for linguistically describing the room temperature (otherwise we will have to spend a lot of time defining rules later on). In order to do that we will have to reinitialise the 'room_temp' antecedent variable:

```
room_temp = ctrl.Antecedent(np.arange(5, 36, 1), 'room temperature')
```

Previously we used the 'automf' method to automatically generate membership functions. Here we can demonstrate how to define custom membership functions (for the respective linguistic value). Let's assume, that the three words we will use to describe room temperature are 'cold', 'OK' and 'warm'. Just to demonstrate the different way the various distributions (membership functions) are defined, let's make 'cold' have a trapezoid distribution, 'OK' to have Gaussian (normal) distribution, and 'warm' will have a triangular distribution.

The trapezoid membership function is defined with the .trapmf( ) method. It takes as inputs the x axis values (in our case those are the 'room_temp.universe' we defined - consisting of the values from 5 to 36 degrees Celsius), and then the x-coordinates of the four points of the trapezoid (ABCD) in the form of a list.

The Gaussian membership function generator also takes the universe as an input, but then the Gaussian curve is defined by its mean and sigma values. We will use 20 degrees as the mean and 4 degrees as the standard deviation (sigma).

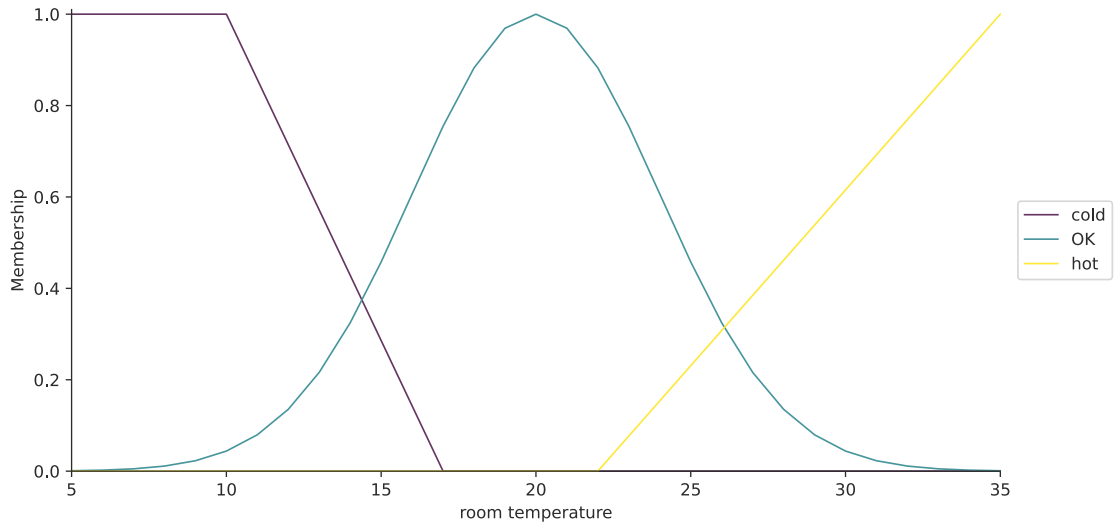
The Triangle generator also takes the universe, but then can be manually adjusted, by providing the function with a list of the x coordinates of the vertices (ABC) of the desired triangle:

```
# Custom linguistic variables, and custom membership functions
room_temp['cold'] = fuzz.trapmf(room_temp.universe, [5, 5, 10, 17])
room_temp['OK'] = fuzz.gaussmf(room_temp.universe, 20, 4)
room_temp['hot'] = fuzz.trimf(room_temp.universe, [22, 35, 35])
```

Let's generate a new visualisation for the room temperature universe and the membership distribution of the various values of the linguistic variable "Temperature":

```
plt.rc('axes',
```

```
        prop_cycle=cycler(
                color=plt.get_cmap('viridis')(np.linspace(0, 1,
                        len(room_temp.terms)))))
plt.rcParams['figure.figsize'] = [10, 5]

room_temp_viz = ctrl.visualization.FuzzyVariableVisualizer(room_temp)
fig, ax = room_temp_viz.view()

ax.legend(loc='center left', bbox_to_anchor=(1.01, 0.5))
```

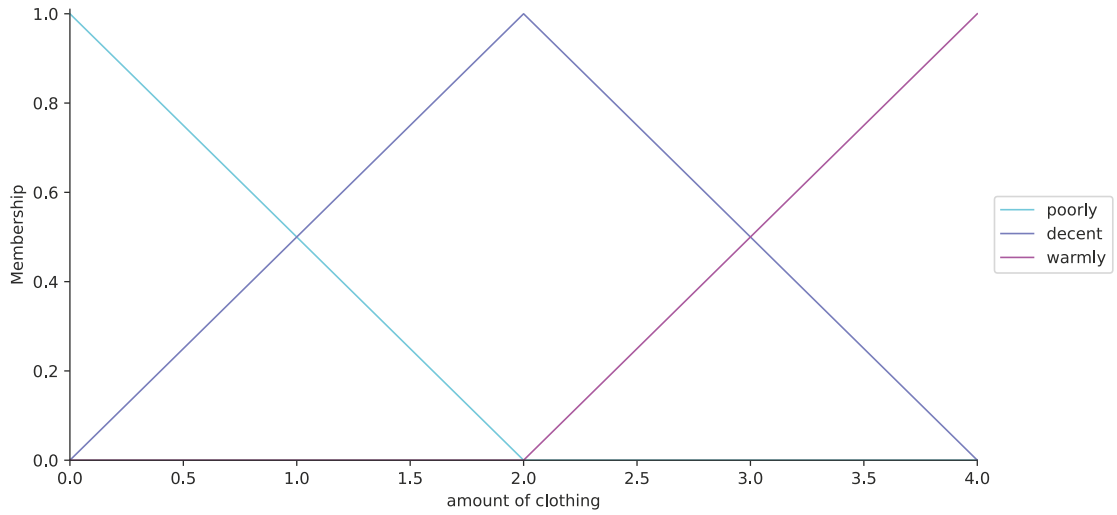


Let's define the membership functions for the values of the clothing level, We are going to use the auto generator, but we will pass our own custom linguistic values for describing how warm we are dressed:

```
# Custom linguistic values, describing how well are we dressed
clothing_qualitative_values = ['poorly',
                               'decent',
                               'warmly']

clothing.automf(3, names=clothing_qualitative_values)
```

Let's visualise that as well, changing the colormap to something different, for example the 'cool' one:

```
plt.rc('axes',
       prop_cycle=cycler(
               color=plt.get_cmap('cool')(np.linspace(0, 1,
                       len(clothing_qualitative_values)))))
plt.rcParams['figure.figsize'] = [10, 5]

clothing_viz = ctrl.visualization.FuzzyVariableVisualizer(clothing)
fig2, ax2 = clothing_viz.view()

ax2.legend(loc='center left', bbox_to_anchor=(1.01, 0.5))
```

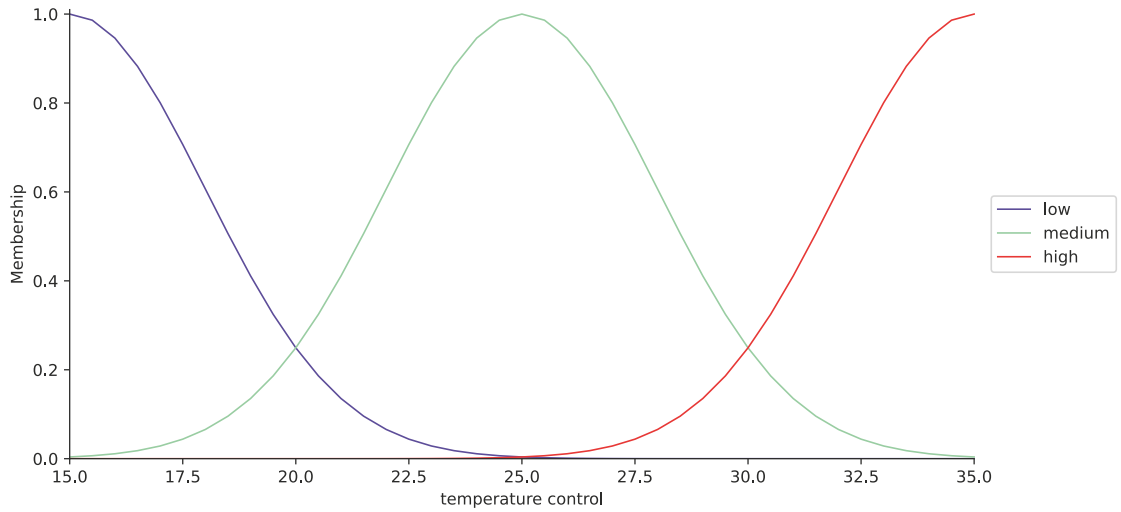


And finally, for the temperature control - the Consequent (the result) of our system, we will again use custom linguistic values and membership functions. Again, we will limit ourselves with only 3 variables, and we will make all of them to have Gaussian distribution:

```
temp_control['low'] = fuzz.gaussmf(temp_control.universe, 15, 3)
temp_control['medium'] = fuzz.gaussmf(temp_control.universe, 25, 3)
temp_control['high'] = fuzz.gaussmf(temp_control.universe, 35, 3)
```

Let's visualise that as well:

```
plt.rc('axes',
       prop_cycle=cycler(
               color=plt.get_cmap('rainbow')(np.linspace(0, 1,
                     len(clothing_qualitative_values)))))
plt.rcParams['figure.figsize'] = [10, 5]

temp_control_viz =
ctrl.visualization.FuzzyVariableVisualizer(temp_control)
fig2, ax2 = temp_control_viz.view()

ax2.legend(loc='center left', bbox_to_anchor=(1.01, 0.5))
```



The reason we choose to use normal distribution for this part of the system (and to be honest this is a good starting point for most values), is because normal distributions tend to come up

normally (pun not intended, but that is the reason it is called that way), when asking many people to give quantitative description of anything.

Once we have the inputs and the outputs defined, it is time to set the rules that the system will use to convert the inputs into outputs. As we discussed, rules are added by using the fuzzy state of the variables (using the linguistic values). For example, our first rule could be that regardless of what the temperature of the room is at the moment, if I we are 'poorly' dressed (may be not wearing any clothes all, or just a t-shirt), then we would like the temperature control system to be set to 'high'.

We can add another rule, stipulating that if the room temperature is 'hot', and we are 'warmly' dressed, then we would like the temperature control to be set to 'low'. The Boolean **AND** operand is defined with the ampersand symbol **&**. If we want to use an **OR** operand, we can use the pipe symbol **|**.

Let's see what other rules we can think of:

```
rule1 = ctrl.Rule(clothing['poorly'],
                  temp_control['high'])
rule2 = ctrl.Rule(clothing['decent'],
                  temp_control['medium'])
rule3 = ctrl.Rule(room_temp['cold'] & clothing['decent'],
                  temp_control['high'])
rule4 = ctrl.Rule(room_temp['OK'],
                  temp_control['medium'])
rule5 = ctrl.Rule(room_temp['hot'],
                  temp_control['low'])
rule6 = ctrl.Rule(room_temp['hot'] & clothing['decent'],
                  temp_control['medium'])
rule7 = ctrl.Rule(room_temp['hot'] & clothing['warmly'],
                  temp_control['low'])
```

Once the rules are set, we need to register them in the control system. We do that by passing a list of all the rules to the ControlSystem class and store the resulting control system in a variable. Let's call it 'temp_ctrl_system':

```
temp_ctrl_system = ctrl.ControlSystem(
     [rule1, rule2, rule3, rule4, rule5, rule6, rule7])
```

In order to show the control system in action, we can use the ControlSystemSimulation class and pass in the already defined control system. We can store the simulation object in a variable called 'controlling':

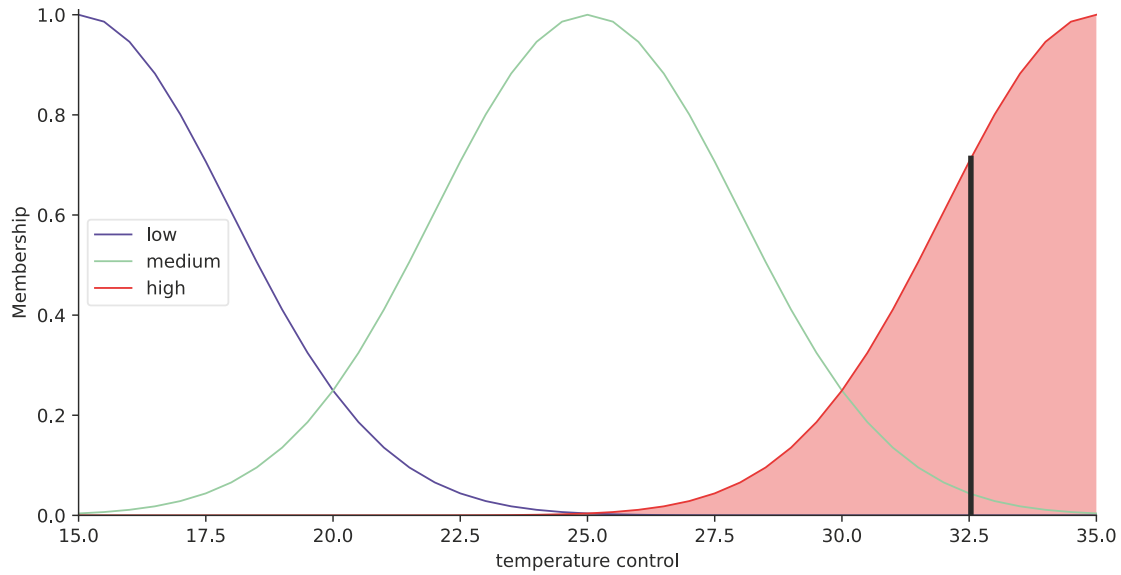
```
controlling = ctrl.ControlSystemSimulation(temp_ctrl_system)
```

Now, with that simulation object created we can start passing in (crisp) input values. We use the .input property, addressing the respective linguistic variables. Let's say that the 'room temperature' is 6 degrees Celsius, and that we have 0 'amount of clothing':

```
# Pass inputs to the ControlSystem using Antecedent labels
# with Pythonic API. For more than one input use dictionary
controlling.input['room temperature'] = 6
controlling.input['amount of clothing'] = 0
```

In order to get an output value, we have to tell the simulation to crunch the numbers (to compute the output, based on the input). We do that using the .compute( ) method. Then, we print out the .output property of the simulation, addressing the 'temperature control' variable. We can also ask for a visualisation of the solution by again calling the .view( ) method on the output variable, but this time we pass in the simulation object:

```
# Crunch the numbers
controlling.compute()
print(controlling.output['temperature control'])
temp_control.view(sim=controlling)
```



As expected, we can see that the control system has made the decision to crank up our imaginary air conditioning unit almost to the max. I encourage you to play with the inputs to see what decisions the control system will make. Think about how the rules we set are interpreted. Try to set different rules (or to add and remove rules) and see how the decision-making process will change.

One final thing we can do here is to try and visualise the entire decision space. For this example, with only 3 dimensions (2 inputs and 1 output), that would be relatively easy. When dealing with more than 3 dimensions things might be trickier to visualise.

What we want to have are 2 axis containing the respective universes (the ranges of values) for the two input variables, and then for the third axis we want the computed output. We can think of it as a point cloud (or a 3D scatter plot) that contains all possible combinations of input values and the respective output. Of course, the density of the cloud will depend on how fine those value steps will be for every input dimension.

Let's start by creating just that, two variables containing an array of possible values for the room temperature and the clothing. We will use a 0.5 degree step for the room temperature. We will use Numpy's .arange( ) method to construct the arrays of values, giving a start and an end values, and the necessary step. Since the range of the clothing universe is shorter, we can select a smaller step, for example 0.1. Of course, if we assume we can only change clothing by 1 full layer, then that wouldn't make much sense practically, but it will make the visualisation of the decision surface clearer to understand, so we will go for the finer step:

```
room_temp_options = np.arange(5, 36.5, 0.5)
clothing_options = np.arange(0, 5, 0.1)
```

Our next step is to populate three lists, that will contain the coordinates for every dimension, for every input value combination. We can do that by first defining the three empty lists and then starting two nested for loops, to iterate through all the values in the temperature and clothing options.

For every temperature step we go through all clothing options. we give the current temperature and clothing layer value as inputs to the control system simulation and we compute an output. Then we append the result and the used input values in their respective lists:

```
temp_control_results = []
room_temp_results = []
clothing_results = []

for c in clothing_options:
    for r in room_temp_options:
        controlling.input['room temperature'] = r
        controlling.input['amount of clothing'] = c
        controlling.compute()

        temp_control_results.append(
                    controlling.output['temperature control'])
        room_temp_results.append(r)
        clothing_results.append(c)
```

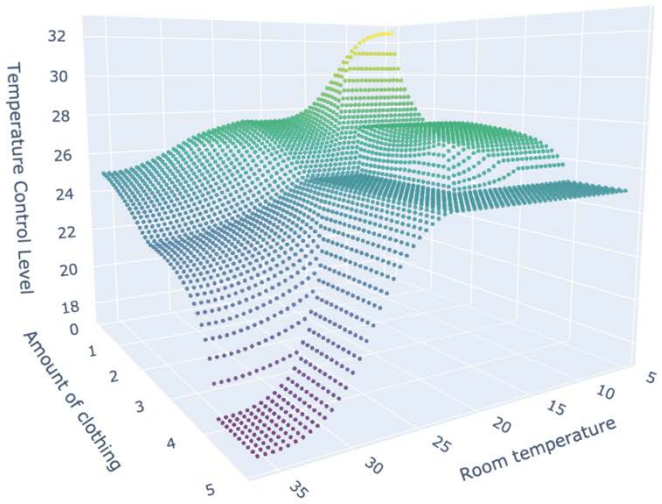
Finally, we are ready to visualise the data. This is where we will use the Plotly library. The way Plotly works is a little different than Matplotlib. In Matplotlib's pyplot we defined the figure first and then pass the data to the respective type of plot constructor, and then finally we could address the figure or the axis to apply the necessary stylistic customisations. Here we start by defining the data and the plot element with its stylistic parameters, then we define the layout and its style, and finally we construct the figure passing those two arguments.

The important thing here is we will use the three lists we created in the previous code block as the coordinates for our 3 dimensional scatter plot.

For the layout we will define the title of the figure and the descriptions of the three axis, as well as set the initial camera angle (remember the plot is going to be interactive, and allow us to rotate camera to look at it from different angles):

```
control_data = go.Scatter3d(
    x=room_temp_results,
    y=clothing_results,
    z=temp_control_results,
    mode='markers',
    marker=dict(
        size=2,
        color=temp_control_results,
        colorscale='Viridis',
        opacity=0.8))

layout = go.Layout(
    title='Decision surface',
    scene=dict(
```

```
        xaxis=dict(title='Room temperature'),
        yaxis=dict(title='Amount of clothing'),
        zaxis=dict(title='Temperature Control Level'),
        aspectmode='manual',
        aspectratio=dict(x=1.2, y=1.2, z=1),
        camera=dict(eye=dict(x=1.5, y=1, z=1.5))),
    margin=dict(l=0, r=0, b=0, t=0))

fig = go.Figure(data=control_data, layout=layout)
fig.show()
```



We should be able to clearly see how the various rules we defined have affected this decision surface. Again, I encourage you to experiment with the rule definitions and see how this surface changes.

Chapter XI.
Conclusion

The application of machine learning (ML) in industrial control systems is an exciting and rapidly evolving field, with substantial potential to revolutionize the way industries operate. The boom of “AI” related tools in recent years, and their rapid adoption by the business is a great indicator. Throughout this book, we have explored key ML techniques such as forecasting, classification, clustering, and optimization – all fundamental areas that are integral to modern industrial control applications. These techniques, when applied effectively, can enhance decision-making processes, improve system performance, and increase efficiency across various industrial domains.

As we look to the future, it is evident that machine learning's role in industrial control systems will continue to grow. This progression is driven not only by advancements in what is publicly regarded as “AI” and all the ML algorithms behind it, but also by significant improvements in computational hardware. Modern processors both desktop and mobile are starting to also include specialized modules for ML tasks. Datacenters and workstations employ accelerators like GPUs and TPUs to streamline ML model training. The development of embedded systems capable of efficiently running ML models are enabling the real-time application of increasingly complex control systems that were once impractical or even impossible.

However, this book only scratches the surface of what is possible with machine learning. As you continue your studies, I encourage you to explore further into the vast array of algorithms and techniques available within the fields we have touched upon. While this book has provided a foundation, there are many more approaches, methods, and models that can offer unique insights and solutions to complex industrial challenges. I urge you to try applying what you learned here to a wider range of problems. Scour the internet for interesting challenges, look in the dataset repositories for real-world problems, to see how the techniques we used will translate in an environment where things are not as well defined. The world of machine learning is continuously evolving, and staying curious and proactive in learning will empower you to contribute to the next generation of innovations in industrial control.

By deepening your knowledge and experimenting with different algorithms, you will be better equipped to tackle the challenges of tomorrow’s industries. The intersection of machine learning and industrial control offers limitless opportunities, and your understanding of this field will be a key asset as you move forward in your academic and professional careers.

XI.1. Works Cited

[1] New Castle Library, “Newcastle City Library energy consumption,” 2020. [Online]. Available: https://data.world/datagov-uk/44aca5bc-733b-492c-aada-f16462be9f38.

[2] Pixabay, “Pixabay Developer API,” 07 2022. [Online]. Available: https://pixabay.com/service/about/api/.

[3] Pixabay, “Pixabay API Documentation,” 07 2022. [Online]. Available: https://pixabay.com/api/docs/.

[4] Mozilla Corporation, “HTTP response status codes,” 07 2022. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status.

[5] C. M. Douglas Crockford, “ECMA-404 The JSON Data Interchange Standard.,” [Online]. Available: https://www.json.org/json-en.html.

[6] Randomuser, “Randomuser,” 07 2022. [Online]. Available: https://randomuser.me.

[7] C. Holt, “Forecasting Seasonals and Trends by Exponentially Weighted Moving Averages.,” *ONR Memorandum,* vol. 52, 1957.

[8] P. R. Winters, “Forecasting sales by exponentially weighted moving averages,” *Management Science,* vol. 6, p. 324–342, 1960.

[9] A. B. K. J. K. O. R. D. S. R. J. H. F. V.-A. Phillip G. Gould, “Forecasting time series with multiple seasonal patterns,” *European Journal of Operational Research,* vol. 191, no. 1, pp. 207-222, 2008.

[10] J. Couch, “Marketing Budget and Actual Sales Dataset,” September 2022. [Online]. Available: https://www.kaggle.com/datasets/jacouchs/marketing-budget-and-actual-sales-dataset.

[11] P. B. J. C. Z. C. A. D. J. D. M. D. S. G. G. I. M. I. M. K. J. L. R. M. S. M. D. G. M. B. S. P. T. V. Martín Abadi, “TensorFlow: A system for large-scale machine learning,” *Computer Science,* 27 May 2016.

[12] Keras, “Keras 3 API documentation,” [Online]. Available: https://keras.io/api. [Accessed 2025].

[13] MatplotLib, “Colormap reference,” [Online]. Available: https://matplotlib.org/stable/gallery/color/colormap_reference.html.

[14] P. e. al., “Scikit-learn: Machine Learning in Python,” *Journal of Machine Learning Research,* vol. 12, pp. 2825-2830, 2011.

[15] D. L. D. S. Broomhead, “Multivariable Functional Interpolation and Adaptive Networks,” *Complex Systems,* vol. 2, pp. 321-355, 1988.

[16] S. Learn, “RBF SVM parameters,” [Online]. Available: https://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html. [Accessed May 2023].

[17] T. Lin, “Labelimg - popular image annotation tool,” [Online]. Available: https://github.com/heartexlabs/labelImg. [Accessed 2022].

[18] Google, “MediaPipe,” [Online]. Available: https://google.github.io/mediapipe/solutions/hands.html. [Accessed 2023].

[19] Google, “Google OR-Tools: Route. Schedule. Plan. Assign. Pack. Solve,” [Online]. Available: https://developers.google.com/optimization. [Accessed May 2023].

[20] GeoPandas, “GeoPandas,” [Online]. Available: https://geopandas.org/en/stable/. [Accessed May 2023].

[21] The OpenStreetMap Foundation (OSMF), “Open Street Map,” [Online]. Available: https://www.openstreetmap.org. [Accessed 2023].

[22] B. Rouberol, “Haversine,” [Online]. Available: https://pypi.org/project/haversine/. [Accessed 2023].

[23] P. T. N. R. F. I. I. E. F. A. Dwi Arman Prasetya, “Resolving the Shortest Path Problem using the Haversine Algorithm,” *Journal of Critical Reviews,* vol. 7, no. 1, pp. 62-64, 2020.

[24] Open Data Soft, “Geonames - All Cities with a population > 1000,” [Online]. Available: https://public.opendatasoft.com/explore/dataset/geonames-all-cities-with-a-population-1000/table/?disjunctive.cou_name_en&sort=population&refine.timezone=Europe. [Accessed May 2023].

[25] EPSG Geomatics Committee, “EPSG Geodetic Parameter Dataset,” [Online]. Available: https://epsg.org/. [Accessed 2023].

[26] L. A. Zadeh, “The Concept of a Linguistic Variable and its Application to Approximate Reasoning, Part I,” *Information Sciences,* vol. 8, no. 3, pp. 199-249, 1975.

[27] SciKit Fuzzy, “Membership Function Generators,” [Online]. Available: https://pythonhosted.org/scikit-fuzzy/api/skfuzzy.membership.html. [Accessed MAY 2023].

[28] L. Zhao, “Traffic Flow Forecasting,” 2022. [Online]. Available: https://doi.org/10.1145/3339823.

[29] V. B. A. V. A. T. G. S. C.-L. C. M. G. Fan Zhang, “MediaPipe Hands: On-device Real-time Hand Tracking,” in *CVPR Workshop on Computer Vision for Augmented and Virtual Reality*, Seattle, 2020.