Feedback control: theoretical and experimental research using a keysight DAQ

Tom Niessen

Supervisor: Sanli Faez

Universiteit Utrecht

Natuur- en Sterrenkunde
Universiteit Utrecht
NanoLINX

Abstract

Feedback is a concept that is of great influence in our lives. From the regulation of our bodies' temperature to the ice-albedo effect on glaciers, feedback is everywhere. Feedback control is used in modern engineering to control dynamical systems, where it is mainly used to stabilize otherwise instable quantities and states. I have done theoretical research on feedback control with the aim of creating my own feedback control system. The system consists of a pendulum, of which we want to control the amplitude. The amplitude of the pendulum is measured using a LED that, depending on the position of the pendulum, shines a certain amount of light on a photodiode. The signal from the photodiode is used to calculate a feedback force, that will be applied by a fan blowing against the pendulum. These calculations happen inside a controller, consisting of a PC and digitizer, for which I have succesfully written a python wrapper. We have experimentally found that the controller can run at 45Hz, causing a delay of 30ms.

Contents

Chapter 1

Introduction

The concept of feedback is of great influence in our lives. We can find examples of feedback mechanisms everywhere in nature, from the regulation of our bodies' temperature to the ice-albedo effect on glaciers. Also in physics and engineering, feedback has been of great influence. From the start of modern science, and even before that, physicists have been interested in controlling systems, and the concept of feedback has greatly helped us do that. An early example of a feedback loop is the *centrifugal governor*[1]; 'cruise-control for steam locomotives'.
As science developed, a mathematical foundation of feedback and it's big brother control theory started to form. From this point, feedback really started to evolve and we now see feedback control everywhere in engineering. From cruise control in cars to learning a robot how to walk, feedback is everywhere!
The goal of my project is to learn about feedback control by doing theoretical research and by trying to create my own feedback control system. An essential part of the feedback control system will be the digitizer, a device that can convert between analog and digital signals. My aim is to write a wrapper to control this device, using the programming language python. Finally, I will do experimental research on my system to study time-scales of the controller. My system consists of a pendulum of which I want to control the amplitude. I measure the amplitude of the pendulum using a LED that, depending on the position of the pendulum, shines a certain amount of light on a photodiode. This way the signal coming from the photodiode will be related to the position of the pendulum. Using a digitizer and a control algorithm the controller calculates a control action that will be executed by a fan blowing against the pendulum.

Chapter 2

Theory

Besides explaining the outcome of my project, one of the main goals of my thesis is introducing the reader to the subject of feedback. In this chapter I will explain some essential theory about feedback and control. Starting with some general fundamentals and examples, while discussing the question *'What is the role of feedback in our daily lives?'*. Thereafter, I will talk about how we can create a feedback loop ourselves, increasingly focusing on subjects relevant to my project, such as *PID control*.

2.1 General feedback and control theory

2.1.1 Feedback

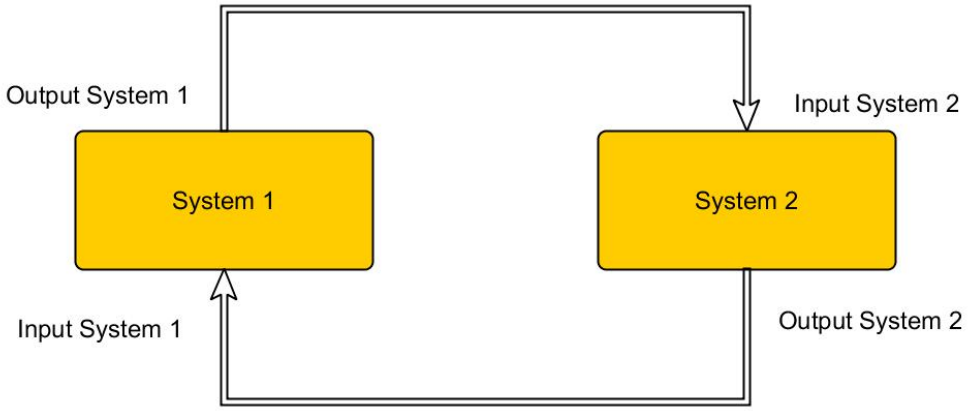


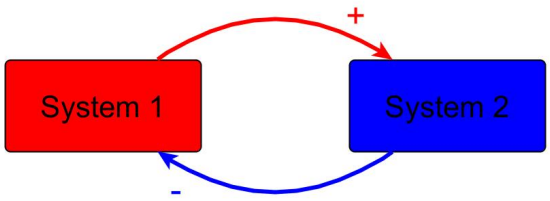
Figure 2.1: A systematic presentation of a simple two-system feedback loop.

We can speak of *Feedback* when we have a sequence of *dynamical systems* that influence each other.[2] A dynamical system is a system in which one or more physical quantities are time-dependent, as well as receptive to external factors. When a change in a dynamical quantity within a system triggers a mechanism that results in an additional change in that quantity, either positive or negative, we are dealing with a *feedback loop*.

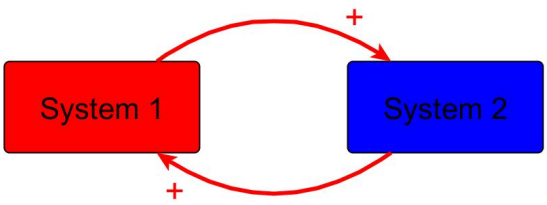
In Fig. 2.1 we see that the output of System 1, which can by any physical quantity, serves as input for System 2, and thus changes the dynamic state of System 2. System 2 on its turn influences the dynamic state of System 1 in the same way, where a certain physical quantity of System 2 serves as input for System

1. This means that, a change in System 1 eventually leads to an additional change in System 1[1]. Both systems are now said to be in a *closed feedback loop*, and they are *dynamically coupled*.

While it might seem trivial, feedback is everywhere around us. Dozens of examples of feedback can be found in various fields of physics. But also outside (especially outside!) of the bounds of physics feedback is a very common phenomenon. For instance, many examples of feedback find their origin in biology, and more specifically in ourselves, the human body, where many of the most essential processes greatly rely on feedback.



(a) Negative feedback. A positive change (of some dynamic variable) in System 1 results in a change in System 2. The change in System 2 then results in a negative change in System 1, putting it closer to it's initial state.



(b) Positive feedback. A change (of some dynamic variable) in System 1 will result in a change in System 2, which will result in an increase of the change of System 1. A positive/negative spiral.

Figure 2.2

A relatively straightforward example is how our body regulates body temperature, as seen in Fig. 2.3. If the hypothalamus senses that temperature rises below normal body temperature level, it will start a mechanism to warm our body. Several parts of our body will be instructed to act, trying to maintain (or reacquire) normal temperature: Blood vessels will constrict (vasoconstriction) so less blood will go to the skin, we will start shivering and hairs on our skin will stand up (piloerection). As a result our body's (core) temperature will stay normal. This loop also runs the other way: if temperature rises, the mechanism will work the other way, trying to cool down our body.[3]

Temperature regulation of the human body is an example of *negative feedback*. In the situation of negative feedback, a change in the system, or some dynamical quantity in the system, will cause a feedback mechanism that reverts the change and recovers the preferred state of the system. *"Negative feedback generally tries to make the system resilient against external disturbances."*[4] Fig. 2.2a shows a simple visual representation of both positive and negative feedback.

Positive feedback, while of less importance to us, appears in nature as well. A great example of positive feedback is the ice-albedo feedback effect of glaciers. When temperature falls and ice or snow is formed on a surface, the albedo[2] of the surface increases. Increased albedo results in more incoming sunlight to be reflected. Because of this temperature on the surface will decrease even further, giving the possibility of more ice to be formed. This feedback can, again, also run the other way: if large areas of ice melt because of increased temperatures, temperature will rise even further.

This example illustrates the great effect feedback can have on our world and there are many more examples of both negative and positive feedback in climate physics, such as the disturbance of the carbon cycle and the greenhouse effect. In the past decades this has become an ever-increasing field of research, helping us not only understand recent climate changes and our role in them, but also in estimating what lies ahead.

Last section has showed some examples of the importance of feedback and it's role in our daily lives. An answer to the earlier stated question (*"What is the role of feedback in our daily life?"*) could be:

[1] This can also be said of System 2: A change a System 2 eventually leads to an additional change in System 2

[2] Albedo is the **reflection coefficient** of a surface. It is a measure for the rate of (sun)light which is reflected by the surface

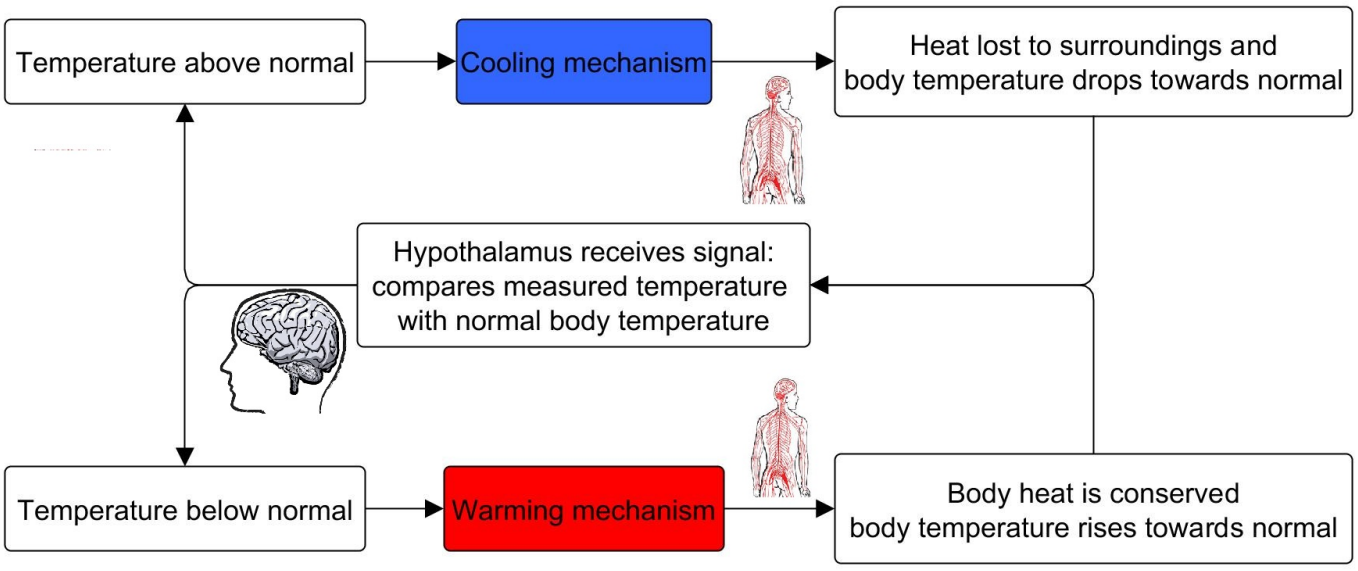


Figure 2.3: Two-way feedback loop of how our budy regulates temperature.

"Feedback does not have one particular role, it is everywhere around us. From the scale of temperature regulation in our body all the way up to the carbon cycle, feedback influences our lives to the point where imagining life without feedback is unthinkable. Examples of feedback can be found in various fields of research. We have just seen examples in biology and climate physics, but also in chemistry, astronomy, even in economics and psychology examples can be found. Feedback is everywhere!"

2.1.2 Feedback control: early use

We have talked about the importance of feedback in nature and looked at some examples of *natural feedback*. We came to the conclusion feedback can be a very powerful phenomenon which can help change or maintain the state of a system, such as temperature in our own body. Now I ask, what would be the consequence if we could make a feedback loop ourselves? We would be able to take a system and choose a quantity to maintain at a level we desire. We could even stabilize an otherwise very unstable quantity. Heck, we could even destabilize a stable quantity! This is how we define *feedback control*: to take a system and control its dynamics (or a certain physical quantity) using the concept of feedback.

When talking about the situation where a dynamic state or certain quantity needs to be maintained at a certain level, we are usually dealing with negative feedback. Most of the time this is the case in feedback control, at least in modern physics, where we mostly want to stabilize systems and/or correct for external disturbances. This is also the case in my project. *It is for this reason that from here, most of my examples and explanations will regard negative feedback.*

Around 1760, with the emergence of the industrial revolution[5], we started developing our industries. We began creating machinery for large-scale use and aimed to increase efficiency of our industries, which includes automation of many (production) processes. When I talk about the automation of a production proces, you can probably see the link with feedback control. Indeed, you are right! Industrialisation in the 19th century went hand in hand with feedback control; it was during this period we started creating our own feedback control systems.
One of the earliest examples of feedback control is the *Centrifugal Governor* [1]. This device was used in steam engines to control the motor power by regulating the amount of steam going into the engine. Note that

this early example of a feedback loop was fully without the use of computers and relied solely on mechanical action. We can see the device in Fig. 2.4. The device consists of a rotating central axis, powered by the steam engine. When the power of the engine increases the beam will start rotating faster. Attached to the beam are two round masses that will rotate with the beam. The increased rotation will increase the energy of the balls, moving them out- and upwards. The upward movement of the ball will cause the attached lever arm to pull down a thrust bearing. This results in the movement of a beam that changes the angle of a valve, decreasing the size of the steam hole. The engine will now get less steam and the power decreases. Through this feedback loop, the power of the steam engine it is connected to will remain constant; an *increase* in motor power will result in a *decrease* in motor power, and vice versa.

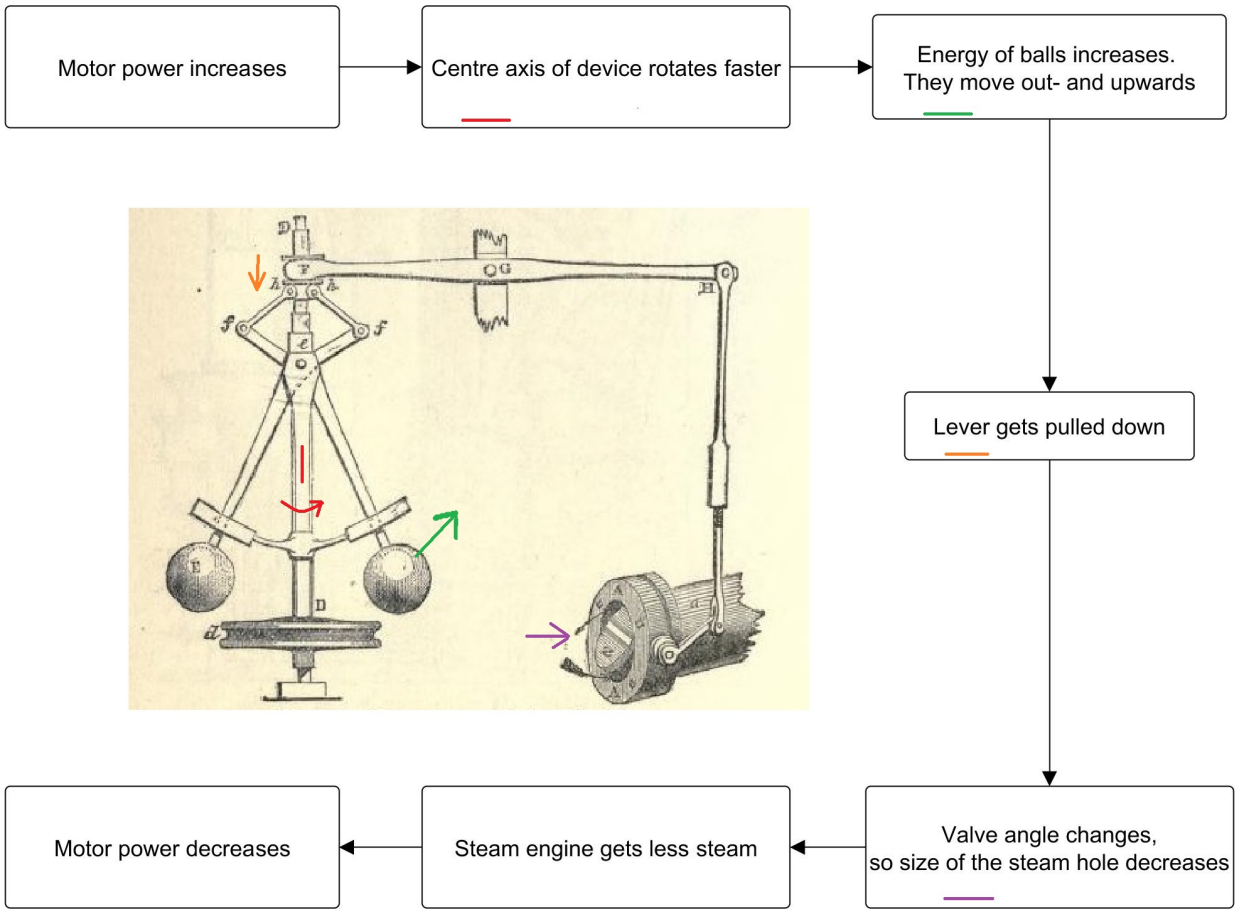


Figure 2.4: Picture of an early feedback control device: the Centrifugal Governor.[1]

The example of the centrifugal governor illustrates how we started using (mechanical) feedback control to automate processes. For instance, the central governor was used in steam locomotives to keep the speed of the train constant, it can be seen as a precursor of what we now call *cruise control*. Another great example of early feedback control was the *thermostat*.

2.1.3 Modern feedback control

With the rise of the computer during the second half of the 20th century, we started using computers to perform feedback control. This had many advantages and our feedback control systems really started to develop from here. A feedback control system no longer had to rely on mechanical components to react to a

change in the system. In a modern feedback control system we can accurately measure the quantity we want to control, called the *process variable*, and let computer algorithms calculate what corrective actions should be taken to react to a change in the system, we call the actions *control actions*. We can choose a *setpoint* and minimize the deviation of the process variable from the setpoint.
A schematic example of a modern feedback control loop can be found in Fig. 2.5. We see that the system we want to control is affected by external disturbances for which we want to correct; we want a certain physical quantity to stay on the setpoint by controlling the dynamics of the system. The state of the system is measured by *sensors*, which will send the measured signal (more than often a voltage) to the *controller*. In modern control systems, control algorithms on the sensor values will generally be done on a digital computer. Before the computer can interpret the measured values the analog signal has to be digitized, this can be done by an analog-to-digital (A/D) converter. After conversion, the computer uses a control algorithm to compute the action that has to be performed by the *actuator*. The actuator is a device that can influence the system we want to control, for instance by applying a force on it. We can control this so-called *feedback force* by sending a command or voltage to the actuator. The signal or command first has to go through the D/A converter before it can be sent to the actuator. When the signal arrives at the actuator, it will perform the action calculated by the computer.

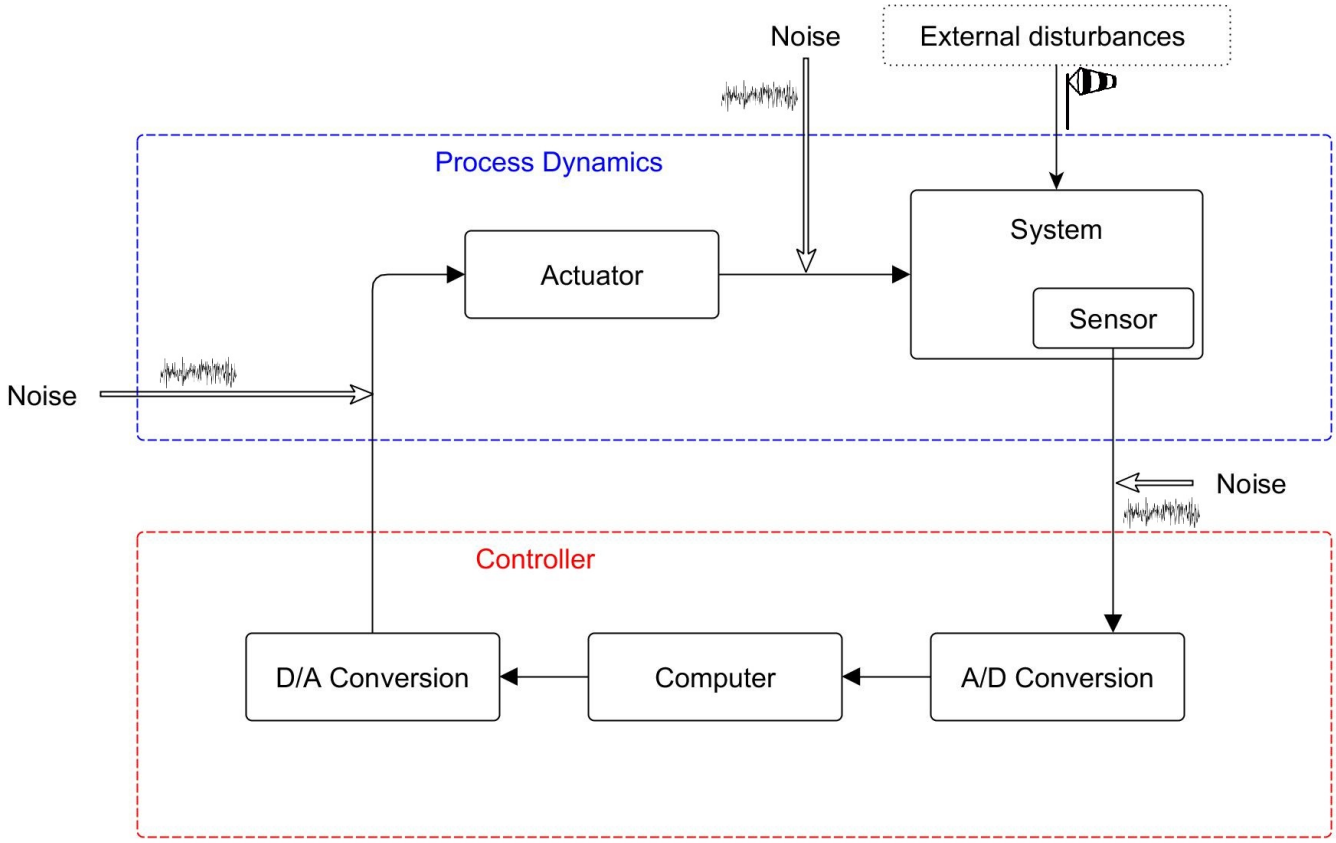


Figure 2.5: Schematic example of a modern feedback control loop

Feedback control is of great importance in modern engineering. Almost every modern device relies in some way on feedback control. Whenever a system needs to be held constant under the influence of external disturbance, you can assume feedback control is present. We have seen the centrifugal governor, '*cruise control* in steam trains', but cruise control in modern vehicles relies no less on feedback control, only this time it is computer driven feedback control. And what to think of aircrafts: constantly varying flows of air,

pressure differences and turbulence, these are all things that need to be corrected for, even when the pilot is turning the plane. Perhaps the ultimate feedback control device would be a robot. In a robot a system needs to be implemented that is capable of exhibiting highly flexible or 'intelligent' responses to changing circumstances[4, p. 211].

The final goal of feedback control will always be to control a system as good as possible. We generally want the loop to run as fast as possible; the higher the frequency of the loop, the faster the system can react to disturbances. But of course, there are many things that have to be taken into account when designing such a system. Some important factors are:

- *Noise.* When transferring and processing an analog signal, noise will most definitely enter the system
- *Measurement errors and bandwidth.* The accuracy of the measurement have large influence on the control system; bad measurements can introduce big errors. Something that is closely related to this is the bandwidth of devices used in the system. To measure accurately we want to measure as many data points as possible, but larger packets of data will probably have a consequence on the data transfer. The transfer can become slower and more noise could enter the system. On the other hand, if we measure too few points we could introduce additional measurement errors.
- *Conversion error.* When converting data from analog to digital, the output will not be exactly the same. It will be the digital representation of analog data, this results in small errors in the digital data.
- *Algorithm roundup.* When the computer does computations with the data to calculate the actuator's response, the software will most definitely perform roundup to make the program more efficient.
- *Dynamic properties of the process: response time and linearity.* What can also play a big role is how fast the actuator will response to a new command/signal, what is the time between sending a command to the actuator and the actuator actually performing the command? Also, will the actuator respond linearly to a change in a signal it gets?
  What matters next is how our system will respond to a change of the actuator. How fast will our system respond and will it respond linearly? These are all characteristics that are determined by the dynamic properties of our system.

The next question that might come to mind is, how does the computer calculate the signal that needs to be sent to the actuator? Which algorithm is the basis for these computations? Indeed, there are multiple forms of *control systems* or *feedback laws* possible. We can have Automata with Programmable Logic Controllers, on-off control, sequential control but perhaps the most well-known and commonly used controller is PID control. This is the controller I have studied and used in my project, where I have tried to implement both a continuous and a digital PID controller. In the next section I will treat PID control; I will explain it's principles and derive the mathematics.

2.2 PID control

If we want to control a system using a PID control algorithm, we need to choose a *setpoint* s, the value we want our *process variable* to have. We can now calculate the error $e(t)$ by taking the difference between our setpoint s and the (measured) process variable $MV(t)$:

$$e(t) = s - MV(t). \tag{2.1}$$

From the error we eventually want to calculate the controller output or *control variable* $u(t)$, which will be sent as a voltage. The controller output can not have every value, as the actuator can only put out a specific domain of voltages (in our case $V_{out} \in [4, 5]$). If the voltage is too low the actuator will not work

and above a certain value the actuator power will no longer increase. We call these boundaries U_{min} and U_{max}. This results in the following formula:

$$u = \begin{cases} U_{max}, & e(t) \geq e_{min} \\ f(e(t)), & e_{min} < e(t) < e_{max} \\ U_{min}, & e(t) \leq e_{max} \end{cases}, \tag{2.2}$$

with $f(e(t))$ a function that gives the controller output as a function of the error, $e_{min} = f^{-1}(U_{max})$ and $e_{max} = f^{-1}(U_{min})$.

In PID control $f(e(t))$ is defined so that:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau)d\tau + k_d \frac{de(t)}{dt}. \tag{2.3}$$

We see that the controller output $u(t)$ depends on three terms: respectively the Proportional term, the Integral term and the Derivative term, hence the name PID control. [4]

The Proportional term

The proportional term $k_p e(t)$, depends on the *current* control error $e(t)$ and the *proportional gain* k_p. The proportional gain is used as a tuning parameter; the control loop will optimally function at a certain value of k_p. The proportional term will generate a control action proportional to the error. As you may suspect this doesn't always lead to the wanted behavior of the loop. A control action proportional to the error means that if the error is zero, there will be no control action; the controller will not act if there is no error. A common consequence of this property is that the process variable will constantly overshoot, causing it to oscillate around the setpoint. The P-term also tends to amplify noise; when it tries to correct for a disturbance it will overshoot and cause the process variable to start oscillating. There are control loops that work fine with only the Proportional term, mainly first order processes with single energy storage, we call these *P-controllers*.[6]

The Integral term

To correct for the drawback of the P-term (the overshooting) we introduce the Integral term. The integral term $k_i \int_0^t e(\tau)d\tau$, depends on the *integral gain* k_i and the integral of the error from the starting point t_0 untill the current time t. This means the integral term is mainly influenced by the past error, so where the P-term mainly focusses on the current error, *the I-term is mainly defined by how long there has been an error*. The integral gives the accumulated error of the system, this is multiplied by the integral gain to set the contribution of the I-term to the control action.
In many cases, the I-term can eliminate the drawback of the P-controller. Now when there is no error, there can still be a control action, defined by the past errors. Controllers that only depend on the P- and I-terms are called *PI-controllers*. While the I-term helps the controller against overshooting, it is not an improvement in terms of the speed of the loop. PI-controllers *eventually* reach the setpoint, but take a lot of time doing so.[6, 7]

The Derivative term

Lastly we have the Derivative term $k_d \frac{de(t)}{dt}$. This term is calculated by taking the derivative of the current error and multiplying this by the *Derivative gain* k_d. The D-term gives an estimate for the future error of the process variable.
Controllers that only depend on the P- and D-term are called *PD-controllers*. The major setback of these controllers is their sensitivity to noise. The derivative of noise is more noise, so the D-term will create incorrect terms.

By far the most used controller is the PID-controller, which depends on all three terms. We have seen it's control action formula in Eq. (2.3). Another common form of this equation is

$$u(t) = k_p\Big(e(t) + \frac{1}{T_i}\int_0^t e(\tau)d\tau + T_d\frac{de(t)}{dt}\Big), \tag{2.4}$$

with again *Proportional* gain k_p and the time constants T_i and T_d, respectively the *integral time* and *derivative time*. The constants from Eq. (2.3) and (2.4) are related like:

$$k_i = \frac{k_p}{T_i}$$
$$k_d = k_p * T_d.$$

In these equations the proportional term $P = k_p e(t)$ can be changed to $P = k_p e(t) + u_0$ so the control action will be u_0 when there is no error. It can be chosen so that we get the desired steady-state value.[4]

2.2.1 PID tuning and The Ziechler-Nichols method

Perhaps the most important part of PID-control is estimating the three gain terms: k_p, k_i and k_d (or k_p, T_i and T_d). There are several ways to get these terms, both experimentally and mathematically. Mathematical methods are often based on the dynamic details of the system, such as the z-transform and the transfer function. The method I have chosen to tune my loop with is *the Ziechler-Nichols' closed loop method*. This method is very common and accessible method to experimentally determine the optimal gain terms without having to know a lot about the (mathematical) dynamics of the system. Before turning to explaining the method, let's define what we are trying to achieve and what influence each gain term has. The major charachteristics of our feedback control loop are[8]:

- *Rise Time.* This is the time it takes for the process variable within the system to rise above 90% of the desired level after applying a control action.
- *Overshoot.* How much the process variable overshoot the desired level. This will be the first peak of the signal.
- *Settling Time.* After putting out a control action, like a constant voltage, how long does the system take to reach a steady state?
- *Steady state error.* The difference between the control action (a voltage) and the measured output of the system (also a voltage).

In Table 2.1 we can see the influence of an **increase** of each gain term on the major charachteristics of the control loop. We can use this table to manually tune our loop.

Table 2.1: Influence of an **increase** of each gain term on the major charachteristics of the control loop.[8]

Increase of	**Rise Time**	**Overshoot**	**Settling Time**	**Steady State Error**
$\mathbf{k_p}$	Decreases	Increases	No reaction	Decreases
$\mathbf{k_i}$	Decreases	Increases	Increases	Eliminate
$\mathbf{k_d}$	No reaction	Decreases	Decreases	No reaction

The Ziegler-Nichols' closed loop method was created and published by Ziegler and Nichols in 1942. Like I said before, it is a way to experimentally tune the gain terms so the controller functions optimally. The method can be applied to P-, PD- ,PI- and PID-controllers. It consists of a set of steps that have to be taken after eachother[8, 9, 10]:

1. Determine the starting point of your tuning procedure. We can do this by adjusting u_0. Also determine if k_p should be positive or negative, by looking at what happens to the process variable when manually changing the control variable u.

2. Make the controller a P-controller by setting $T_i = \infty$ and $T_d = 0$. For the moment, also set $k_p = 0$. Also, choose a setpoint

3. Slowly increase k_p (or decrease if $k_p was determined to be negative$), while keeping an eye on the controller output. Stop when the controller output starts periodically oscillating and note the value of k_p. This value is called the *ultimate gain* k_u.

4. We can now calculate the *ultimate period* T_u by measuring the period of the oscillations.

5. Set the gain terms of the loop according to Table 2.2.

Table 2.2: Values of the gain terms according to the Ziegler-Nichols' method[9, 10]:

Controller type	$\mathbf{k_p}$	$\mathbf{T_i}$	$\mathbf{T_d}$
P-controller	$k_u/2$	-	-
PI-controller	$k_u/2.2$	$T_u/1.2$	-
PID-controller	$k_u/1.7$	$T_u/2$	$T_u/8$

Chapter 3

Setup

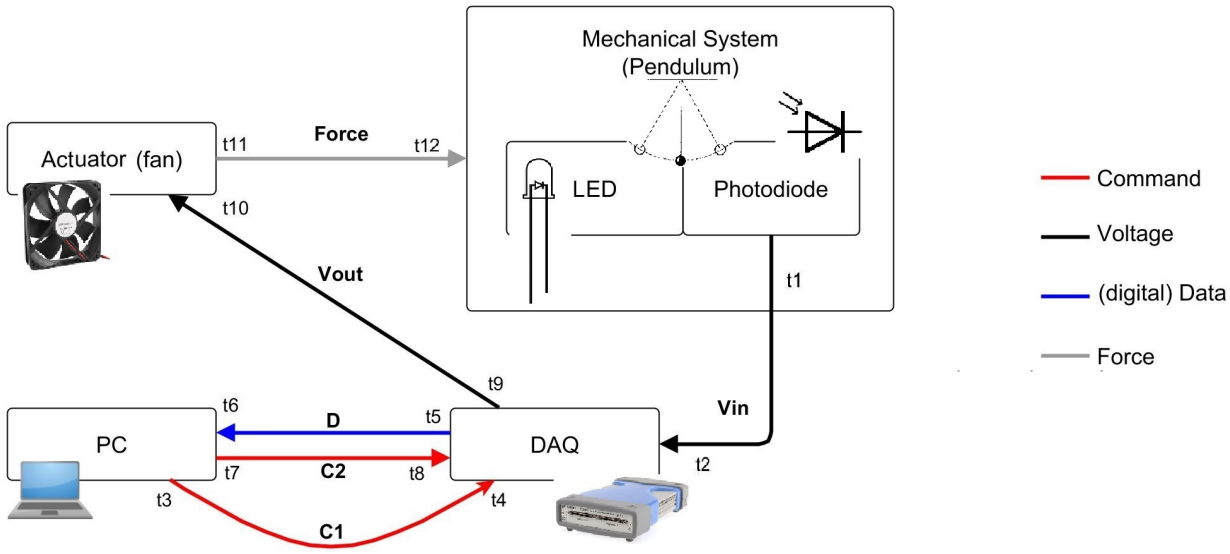


Figure 3.1: Schematic image of the feedback control loop.

In Fig. 3.1 we see a schematic representation of the feedback loop I have created. I will shortly guide you through the parts of the loop.

The Mechanical system.

The system I want to control in my feedback control loop is a simple *pendulum*. The quantity I want to control is the position, or rather the amplitude, of the pendulum. To measure the position of the pendulum I have attached a transparent paper with a gradient on top of the pendulum. On both sides of the gradient and opposite to eachother I have placed a photodiode and a led. The amount of light that arrives at the photodiode is dependent on the angle of the pendulum, which means that the voltage the photodiode generates is dependent on the position of the pendulum, the denoted by V_{in} in Fig. 3.1.
The actuator is the device that applies the control action on the system in the form of a feedback force. In our case the actuator is a DC-powered fan that blows against a surface attached to the pendulum. The more air the fan blows, the greater the amplitude of the pendulum will be. The intensity of the fan and the amplitude of the pendulum are, however, not linearly related.

DAQ (Data Acquisition)

My DAQ-card is the device I use to measure and digitize the voltage coming from the photodiode, it is a *Keysight u2331a*.Besides that, the card also puts out a voltage to the actuator, so it is both the D/A- and the A/D-convertor of my loop.

PC

The PC manages the controller part of the loop. It sends commands to the DAQ; commands to set the configuration of the DAQ, to measure and digitize data and to put out a control action (voltage). Besides managing the DAQ, the PC also calculates the controller output ($u(t)$) from the (digitized) measurement data. This is done in a PID-control program, like explained before. Lastly the PC keeps track of time; the actions in the loop can be clocked by the PC.
The PC does all this work based on a program written in *python*. The program contains:

- A wrapper for the DAQ-card. This is software that enables us to control the actions of the DAQ-card by sending commands to the device.
- The PID-algorithm to calculate the control output.
- Built in clocking, to manage and measure the timing of the loop.

In the next sections I will discuss every part of the feedback loop in more detail.

3.1 The dynamic elements of the loop

3.1.1 The system: a Pendulum

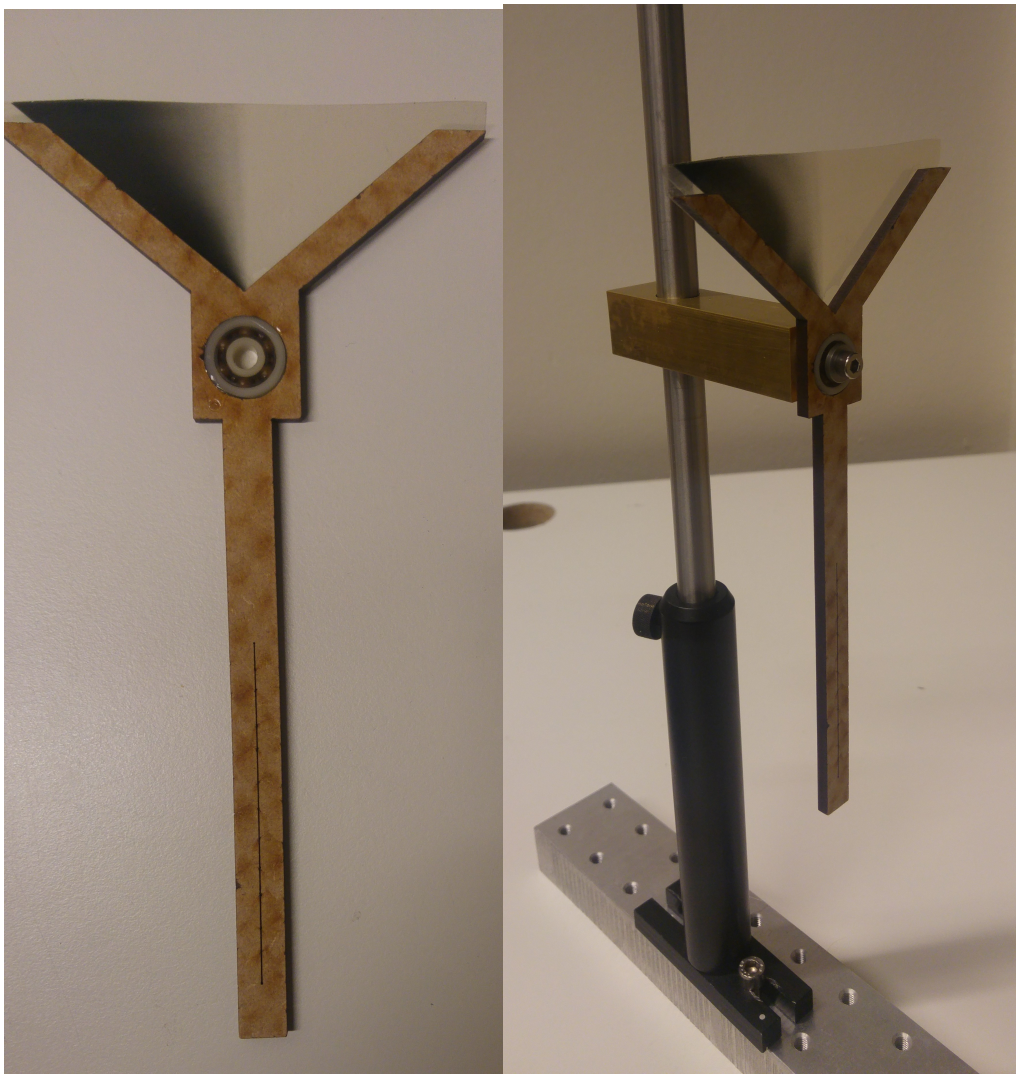
The pendulum is the centre of my control loop, this is the system of which I want to control the dynamics, or at least some part of it. The pendulum can be seen in Fig. 3.2a. It is made from $4mm$ thick plywood and was laser-cut from a self-made design. The important characteristics of the pendulum are

- The V-shape upper part, made so that a transparent-to-black gradient can be attached on top of the pendulum. As you see in Fig. 3.2a, the gradient only starts at half of the V-shape. This is because we can only control the position of the pendulum in one direction, since the fan can only blow the pendulum to one direction. The mathematica code used to create the gradient can be found in Appendix A.
- In the middle of the pendulum we see a hole in which a ball bearing is places. This way we can attach the pendulum to a pole while it can swing with a low amount of resistance. In Fig. 3.2b we can see the pendulum attached to a pole with a specially designed block.
- On the bottom side we can see a small slit. Through this slit we can place a piece of paper, to create a surface the fan can blow on.

3.1.2 Measurement: LED and PD

To measure the amplitude of the pendulum I use a photodiode and a LED. The amount of light from the LED that reaches the photodiode is dependent on the amplitude of the pendulum. The current generated by the photodiode is thus related to the position of the pendulum. To make it possible to place a photodiode and LED on each side of the pendulum's gradient, I designed[1] a block that could be placed around the pendulum, with two holes to place the photodiode and LED in, this way they are held in the right position. Fig. 3.3 shows the designed block.

[1]Of course, this couldn't have been done without the help of Paul and Dante



(a) The plywood pendulum, including gradient and ball bearing.

(b) The pendulum attached to a pole.

Figure 3.2

I use a simple LED that can produce high-intensity white light to limit the influence of light from the outside. The Photodiode generates a current dependent of how much light falls on it. It can produce a current up to 100mA and a very small voltage. Unfortunately, I cannot directly use this signal as the voltage is too small to measure very precisely: the signal needs to be amplified. To do this we could put in a resistor, the circuit would look something like Fig. 3.4a, where we have:

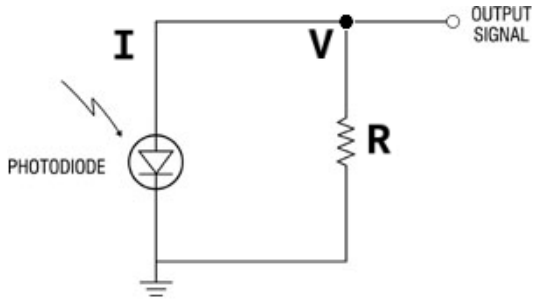
$$V = I * R \tag{3.1}$$

$$I = 100mA. \tag{3.2}$$

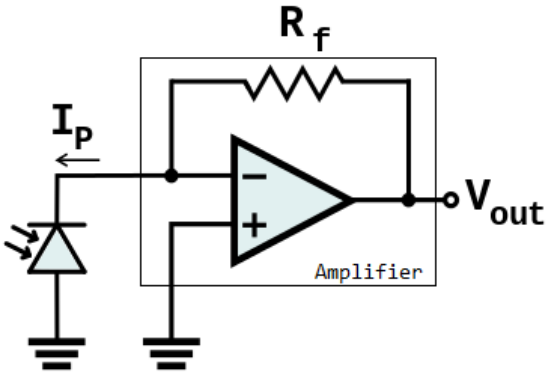
If we would now put a resistor of, $1M\Omega$, we would end up with only $0.1V$, so using a resistor wouldn't solve our problem. Something that does solve our problem is a *transimpedance amplifier*, this is an amplifier that converts our current into a voltage. It has *transimpedace* R_f. We can see that the photodiode is now inverted, so that $V_out = -(I_p * R_f)$, this solves the trick! We now have a voltage we can use. The circuit

Figure 3.3: Designed holder for the LED and photodiode

with the amplifier can be seen in Fig. 3.4b.



(a) Circuit with photodiode amplified by a transimpedance amplifier.



(b) Circuit of photodiode with amplifier.

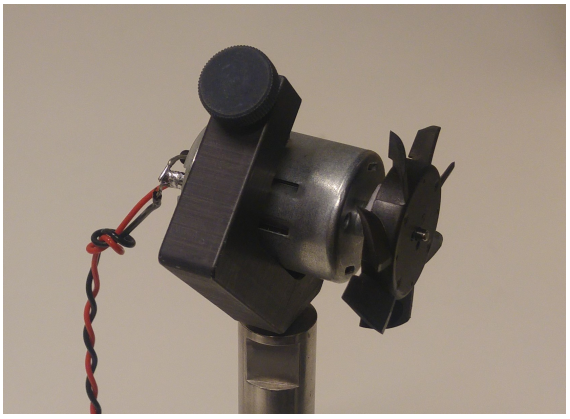
Figure 3.4

3.1.3 Actuator: Fan

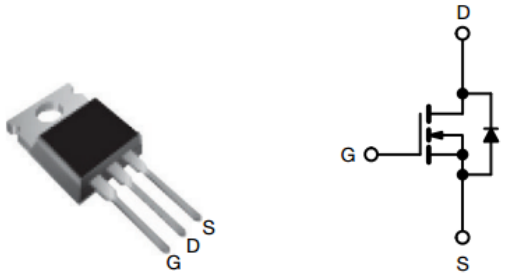
The device that is used to apply the control output to the system is a fan. The fan is made by attaching a set of blades to a '3-to-6 V' DC motor; this means the motor will need at least 3V to start and reaches maximum power at 6V. The fan is shown in Fig. 3.5a. The fan will exert a force on the pendulum by blowing against the paper that is placed throught the slit of the pendulum.

The motor is powered by a voltage coming from the DAQ-card, we can increase the power of the motor

by increasing the voltage going to the fan. However, the DAQ-card can only supply a current of $5mA$, this isn't remotely enough to power the motor. To solve this problem I add a *MOSFET* to the system. A MOSFET is a transistor with three pins: the source, gate and drain. To the source we send a current from a power supply and to the gate we connect the output from the DAQ-card. Now, dependent on the voltage on the gate, the MOSFET puts out part of the voltage on the drain, plus the current on the source. This way we can add a current to the voltage coming from the DAQ-card, before sending it to the motor. If we set the voltage coming from the power supply to about 6-7V, the result is that the motor will run at minimum and maximum power at respectively 4V and 5V coming from the DAQ-card. Fig. 3.5b shows a picture of a MOSFET, with its electric circuit.



(a) The fan (attached to a pole).



(b) Picture and circuit of a MOSFET. G, D and S are respectively the Gain, Drain and Source pins.

Figure 3.5

3.2 The controller

The controller part of the loop is the part that on one end receives an analog signal, then does some control algorithms and puts out a control action. It consists of the DAQ-card, PC and Oscilloscope.

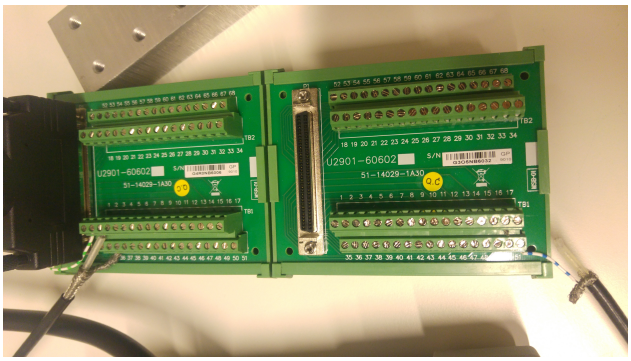
3.2.1 DAQ-card

DAQ-card stands for Data Acquisition card. The device serves as both the A/D- and the D/A-converter of my control loop. This means the card can receive an analog voltage, convert it to a digital signal and then send it to the PC, so the PC has measurement values it can use in its control algorithm. After the PC has done its calculations and computed the control ouput it commands the DAQ to output a certain voltage.

The device that I use is the *Keysight u2331a*, it is a *"USB Modular Multifunction Data Acquisition Device"*[11], it can be seen in Fig. 3.6a. Although in the end I use only a very limited amount of functions of the device in my control program, I have spent quite some time learning about the card. As the name might reveil, it has dozens of functionalities and configurations, learning about all of these functions and how to command them in python code can be quite the challenge. A few of the most important functionalities of the card are[11, 12, 13]:

- Analog input. The device has 64 analog input channels on which it can receive a voltage. It can take up to $3 \cdot 10^6$ samples per second when using one channel. When using multiple channels to receive signals at the same time it can take $1 \cdot 10^6$ samples per second. The domain of voltages that can be measured is from -10V until 10V. There are three functions that the card can use to measure and digitize a voltage

(a) The Keysight DAQ card



(b) The two connectors from the keysight DAQ card

Figure 3.6

(*Measure:Voltage*, *Run* and *Digitize*). In Chapter 4 I will compare these three methods and determin the one that is most efficient for me.

- Analog output. The device can also output a voltage, again varying from -10V to 10V with a current of 5mA. There are multiple ways to generate a voltage: it can generate a constant voltage, use one of it's predefined functions (sine, sawtooth, square and noise) or output a user-defined voltage from its buffer.

- Data storing. The device can store data from the analog input to its buffer, this is what the functions Run and Dignitize use. It is also possible to put a data into the buffer yourself, the device could then output that data as a voltage. The buffer can hold up to $8 \cdot 10^6$ samples. If we read data from the buffer we will get it in a 16-bit Data format, to convert this to an actual float number, we need the following formula:

$$Value = (\frac{2 * Int16_{value}}{2^{resolution}}) * Range, \tag{3.3}$$

  where the resolution of our device is 16 bits and the range is any domain we choose between -10V and 10V. The converted value will then be of type float.

- Configuration. The card has many different configurations, based on the function that is used. There are basis configurations like the polarity, range and choosing a channel. But when we use analog input or output we can also configure things like *sample rate* $[Sa/s]$, the number of samples and the size of a datablock in the buffer.

These were the functions I mainly use in my program, to give an idea of what the device can do. Of course there are many other functions, like adding triggers, clocking, digital in- and output and measuring things like frequency and amplitude.

3.2.2 Oscilloscope

While the oscilloscope is not directly used in my control loop, it has been essential in the development of the system. I used a *Keysight DSOX2024 Oscilloscope*. The two most important functionalities of the device were that I can show up to 4 input signals and that it has a built-in wave generator. This ables me to monitor the loop while it is running and helps me do time-research on different parts of the loop. For example, I could generate a wave on the oscilloscope, then split the signal into two: I would lead one signal directly to

the Oscilloscope's input and one signal via the PC and DAQ to the Oscilloscope. This way I could see the delay and noise caused by the controller of my system.

3.2.3 Software

Large part of my project consisted of learning how the DAQ-card works and what functionalities is has. Perhaps the most time consuming was writing a wrapper for the device using the language *Python*. A wrapper is program, or piece of code, that makes it possible to connect to a device and control its actions. So, not only did I need to figure out what functions the DAQ has, I also needed to find a way to connect to the device and controlling it by sending commands in python. The python-package that enabled me to connect to the device was *PyVisa*, a python wrapper for the Virtual Instrument Software Architechture library. This way I could easily connect to the from my PC and send commands to it using the programming language *SCPI*.

SCPI

SCPI stands for *Standard Commands for Programmable Instruments*. As the name might reveal, we can use this language to control our DAQ-card. SCPI provides a very clear, ordened and hierarchical set of commands. The commands are ordened in a tree-like structure; a data-structure we see a lot in programming. On the toplevel there are some *subsystems* we can choose from. Lets choose the subsystem 'OUTPut', this subsystem contains all kinds of functions that allow us to configure things that have to do with the analog output of the device. 'OUTPut', in its turn, also has some subsystem, we now choose 'WAVeform' and then 'FREQuency'. We now have the command *'OUTPut:WAVeform:FREQuency value'*, which is used to set the frequency of the analog output signal. In python it would look like this:

```
|#Open an instance of ResourceManager
|rm = visa.ResourceManager()
|#Connect to device:
|device = rm.open_resource('USB0::0x0957::0x1518::TW56100007::0::INSTR')
|#Set the output frequency to 1kHz
|device.write("OUTPut:WAVeform:FREQuency 1000")
```

Of course, we only have to connect to the device once, at the start of the program. As we can see in the example the keywords have some capitals and some lower case, this is because the each keywoard has a short and a long form. In short form the lowercase can be left out. Our command would then look like *'OUTP:WAV:FREQ value'*. The most important SCPI functions that I use in my wrapper are:

- Measuring and digitizing a voltage. The device has three functions available that do this: MEASure:Voltage, RUN, and DIGitize. The first one only reads one voltage at a time and can send it directly to the PC. The latter two can measure continuously. This should be lot faster, but the data gets stored in the buffer which,unfortunately, requires an additional command to be send to the DAQ. That brings us to the next command:

- The command to retrieve data stored in the buffer:

  ```
  device.query_binary_values('WAV:DATA?', datatype='h', is_big_endian=False).
  ```

  This command gives us the data from the buffer in an 16bit Int format, which we then have to convert to float.

- Output a voltage, this is done with the command: SOURce:VOLTage. We can specify the voltage and channel to which it should be sent. Sending 5V to channel 202 would then look like this:

  ```
  device.write('SOUR:VOLT 5, @202').
  ```

The python code for doing experiments to do time-research in my loop can be found in Appendix C. The python code with the PID-algorithm can be found in Appendix D.[13]

Chapter 4

Measurements and results

I have done several measurements to determine the charasteristics of my feedback loop, focussing mainly on the frequencies and delay within the controller part of the loop (the DAQ and PC).

4.1 Voltage input measurement

We measured the time between the PC sending a command to the DAQ and receiving data from the DAQ. In Fig. 3.1 this would match with t3 to t6. To do this we used the oscilloscope to generate a voltage and sent it to the DAQ. We then used python code to send the right commands to the DAQ and receive the data from the DAQ's buffer. However, as mentioned before, there are three commands that can be used to achieve this: MEAS:VOLT, RUN and DIG. Which one of these is the fastest and most consistent?

The python code can be found in Appendix C. For each of the three functions the code contains a loop that measures a voltage 1000 times.

4.1.1 Results

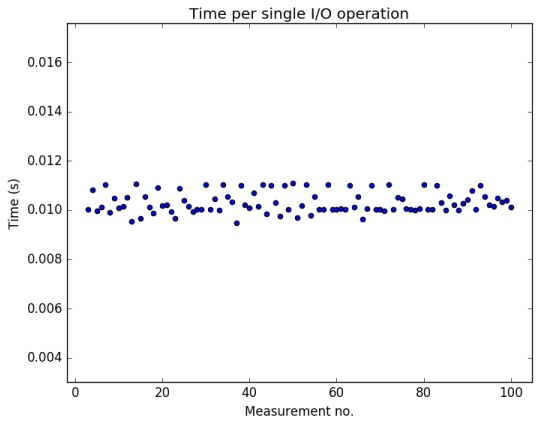
Table 4.1 shows the measured average frequency of digitizing and sending the voltage. While the DAQ can run at speeds up to 3MHz, we only reach 96.64Hz, the result are kind of dissapointing. This delay is probably caused by communication between the PC and the DAQ; the PC has to send a command which the DAQ has to convert to an action. What we can also conclude is that the function 'DIGitize' is the fastest way of getting

Table 4.1: Average frequency of the DAQ digitizing and sending a signal to the PC.
Each value is the mean of 1000 measurements.

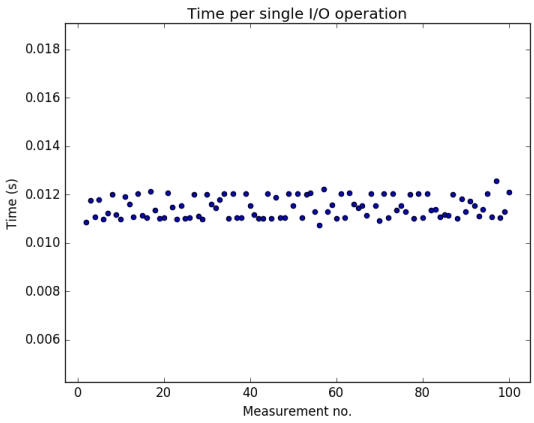
	Frequency	Standarddevation
Measure:Voltage	87.22 Hz	0.28Hz
Run	95.74 Hz	0.22Hz
Dig	96.64 Hz	0.23Hz

Figures 4.1a, 4.1b and 4.1c shows for 50 measurements how long it took to complete it. We see that for each method the times lie in a range of about 0.002s, we can conclude that in terms of consistency not one jumps out positively.

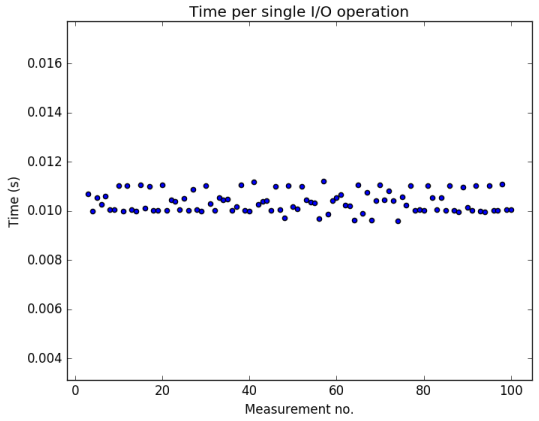
We conclude that the function DIG is most useful to us.



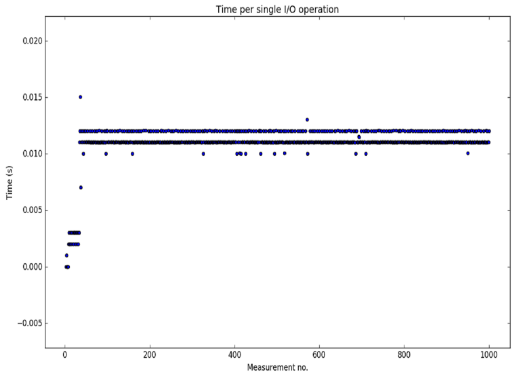
(a) DIG



(b) MEAS:VOLT



(c) RUN



(d) SOUR:VOLT

Figure 4.1: Figures showing the time it takes to measure a voltage and send it to the PC using: **a)** DIG, **b)** MEAS:VOLT and **c)** RUN, for 100 measurements. **D)** shows the time it takes to output a set voltage, commanded by the PC. The python code for this measurement can be found in Appendix C, under "Create graphs with time of each action"

4.2 Voltage output measurement

Besides measuring the timing of voltage input we also measured the timing of voltage output, this matches the time from t7 to t9 in 3.1. As in the last measurement we send a command to the DAQ 1000 times and measure the frequency of the loop. However, we also want to see if the frequency of the voltage-output loop is dependent on the value of the output voltage. We do this by commanding the DAQ to put out two different voltages (V1 and V2) right after eachother, is the frequency dependent on the difference between V1 and V2? We will also measure the avarage frequency of a voltage-output loop for different values of the output voltage, is the frequency dependent on the output voltage?

The python code of the measurements can be found in Appendix C, under "Time-research on voltage output using the command 'SOURce:VOLTage'"

4.2.1 Results

In Table 4.2 we see that the frequency of the output of a voltage is about 89.18 Hz. We also see that the frequency of measuring a voltage and then outputting that voltage is highest for the DIG function. This again confirms DIG will be the function most suitable for is.

Table 4.2: Average frequency of the loop in which the PC sends command(s) to the DAQ and the DAQ performs these actions. Each value is the mean of 1000 measurements.

Command(s) sent	Frequency
SOUR:VOL	89.18 Hz
Measure:Voltage and SOUR:VOLT	44.43 Hz
Run and SOUR:VOL	45.68 Hz
Dig and SOUR:VOL	45.93 Hz

Table 4.3 shows the average frequence of commanding the DAQ to output 2 voltages right after each other. We can see that there does not seem to be a relation between the frequency of the loop and dV (the difference between the two voltages). On the other hand, it looks like there could be a relation between the the frequency and the signs of the voltages. When both voltages are positive the frequency is around 44.90Hz, when one of the voltages is negative the frequency drops to around 44.50Hz and when both voltages are negative, the frequency drops furter, to about 44.10Hz.

Table 4.3: Average frequence of commanding the DAQ to output 2 voltages right after each other. V1 is the low voltage value, V2 is the high voltage value, dV is the difference between V1 and V2. Each value is the mean of 1000 measurements.

V1 (V)	V2 (V)	dV (V)	f (Hz)
1	1	0	44.90
1	9	8	44.91
-1	-1	0	44.10
-1	-9	8	44.11
-1	1	2	44.50
-3	3	6	44.51
-5	5	10	44.69
-9	9	18	44.52
-10	10	20	43.99

Fig. 4.2 confirms the presumption we just made. The figure shows the average frequency of commanding the DAQ to output a voltage versus the voltage of the output signal. We can clearly see there is a relation between the frequency and the output voltage. The frequency of outputting a negative voltage lay about 1.7Hz lower than when outputting positive voltages. We also see that the frequency drops near the edge of the domain, at -10V and 10V. But also at the edge of the domain the positive voltage has a significantly higher frequency than the negative one.

4.3 Measurement of the controller delay

After measurements about the communication between the PC and DAQ, we would like to look at the delay the controller as a whole introduces in our loop. That is, the delay between t2 and t10 in Fig. 3.1. We do this by generating a wave on the oscilloscope, we then split the signal into two: one signal is lead directly to the oscilloscope's input, so the delay is minimal. The other signal is sent to the DAQ, which reads the voltage

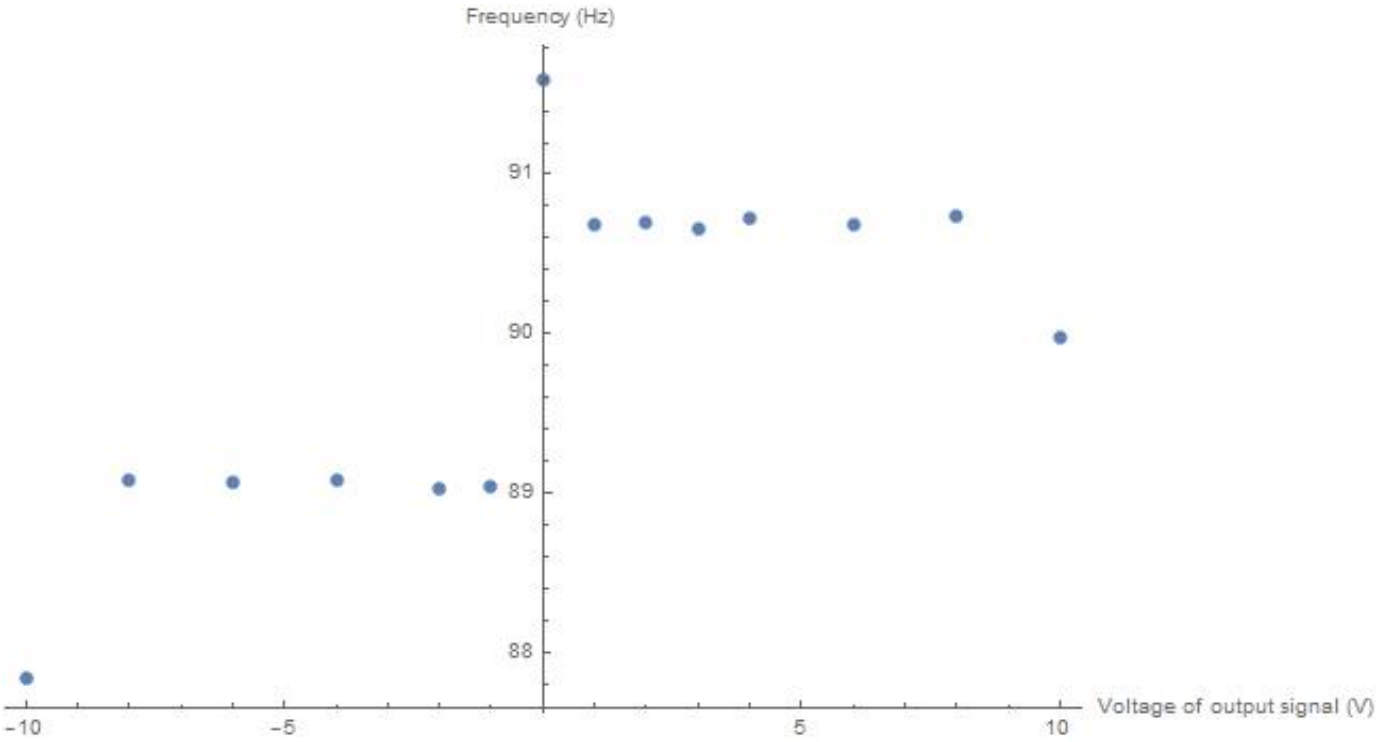


Figure 4.2: Average frequency of commanding the DAQ to output a voltage versus the voltage of the output signal. Each value is the mean of 1000 measurements.

and sends it to the PC, the PC then commands the DAQ to output that same voltage to the oscilloscope. We now have two signals on the Oscilloscope, one signal coming directly from the generator, the *Source signal* and the other coming via the controller of our feedback loop. We can now compare these two signals, at what frequency of the source wave does the noise and delay coming from the controller become too high? We can also quantify the delay of the controller, does it depend on the source wave's amplitude?

The mathematica code for these measurements can be found in Appendix B.

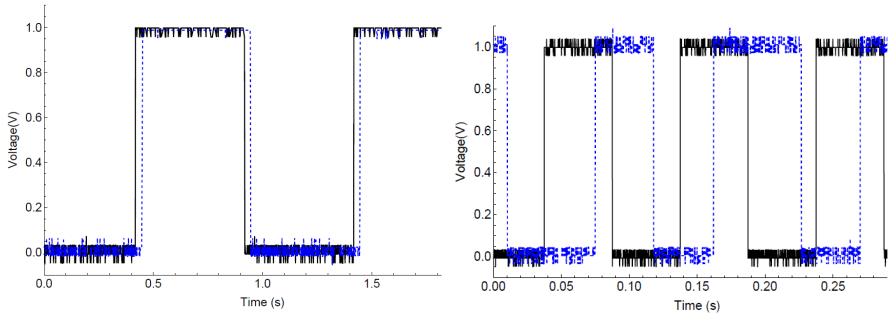
4.3.1 Results

Fig. 4.3 shows the result of sending a source wave directly to the oscilloscope and via the controller. We can clearly see the delay in the signal coming from the controller (dashed blue). At low frequency (1Hz) the delay is relatively small, bet when we increase the frequency of the source wave we see the delay get (relatively) bigger.

In Fig. 4.4 we can see the effects we saw in Fig. 4.3 even clearer. At 1Hz the controller's wave is a nice signal with relatively low delay, but when we increase the frequency of the source signal the differences between the controller output and the source signal become very noticable. Not only does the control output have a delay, at higher frequencies the sine vastly loses shape.

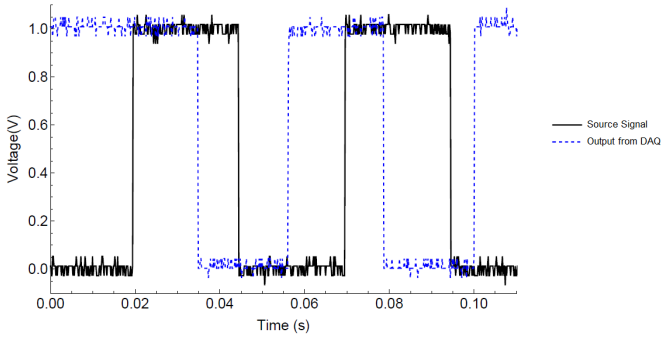
Fig. 4.5 shows the delay between the Source signal and the Controller output of 50 waves of the start and end of each peak. We can see the delay flunctuates a lot, for both the start and end of the peaks. This can probably be solved by changing the frequency of the source wave so that the frequencies of the Source wave and controller output align better. We can quantify the delay by taking the mean of every value in Fig. 4.5. If we do this for different amplitudes of the source wave we end up with Fig. 4.6.

From Fig. 4.6 we can conclude the delay of the controller is indepentent of the amplitude of the incoming wave, and that the delay can be quantified at $29 \pm 6ms$.4



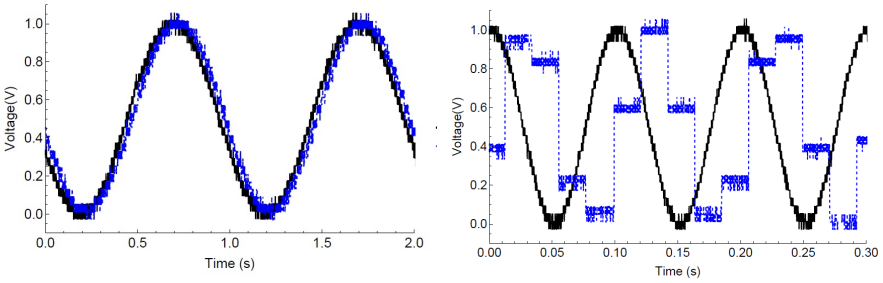
(a) Source wave of 1Hz

(b) Source wave of 10Hz



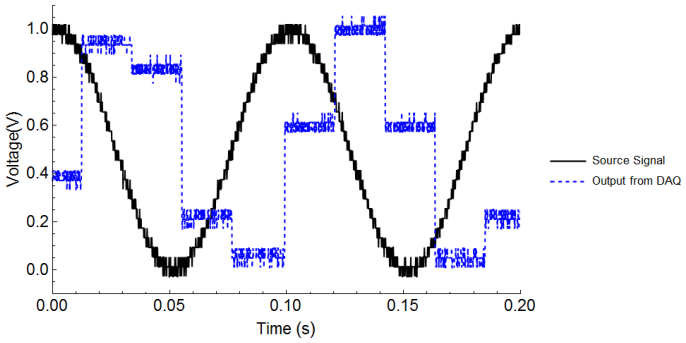
(c) Source wave of 20Hz

Figure 4.3: Figures showing a Square wave of amplitude $1V$, for 3 different frequencies of the source wave. One signal comes directly from the Function Generator (black) to minimize delay, the other is led through the controller(Dashed blue).



(a) Source wave of 1Hz

(b) Source wave of 10Hz



(c) Source wave of 20Hz

Figure 4.4: Figures showing a Sine wave of amplitude $1V$, for 3 different frequencies of the source wave. One signal comes directly from the Function Generator (black) to minimize delay, the other is led through the controller(Dashed blue).

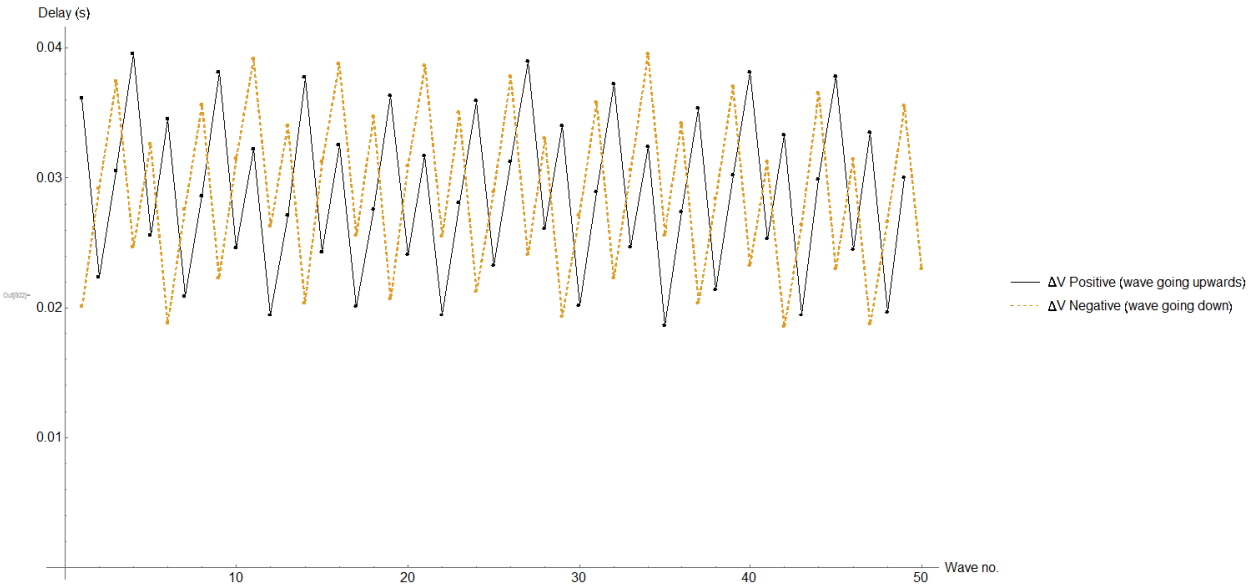


Figure 4.5: A figure showing the delay between the Source signal and the Controller output of 50 waves of: 1) The start of each peak (Black), 2) The end of each peak (Dashed Yellow)

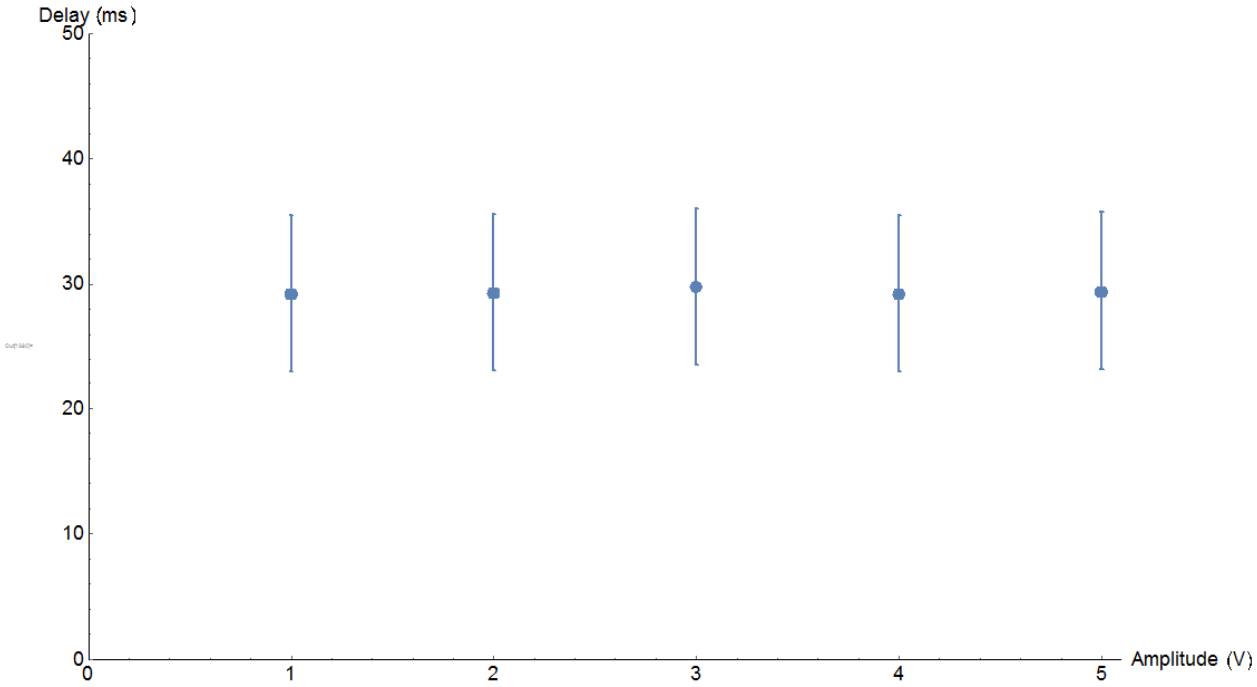


Figure 4.6: The average delay of the controller output with respect to the source signal, as a function of the amplitude of the source signal. The low voltage of the source wave is always 0V, so the high voltage equals the amplitude.

Chapter 5

Conclusion

The goal was to learn about feedback control by doing theoretical research and by trying to create my own feedback system. While I have not (yet) created a fully functional feedback control system, I am very, very close, a picture of the loop can be seen in Fig. 5.1. I succeeded in writing a wrapper for the digitizer and every other part of the system is ready, I just need to find the proper gain terms using the Ziegler-Nichols method.
I have been able to do experimental time research on the controller and found some important properties of the loop. I found that the 'DIG' function of the DAQ is the most efficient function to use and that the output frequency of the DAQ depends on the sign of the voltage. Lastly I found that my controller can run at around 45Hz, causing a delay of aproximately 30ms.

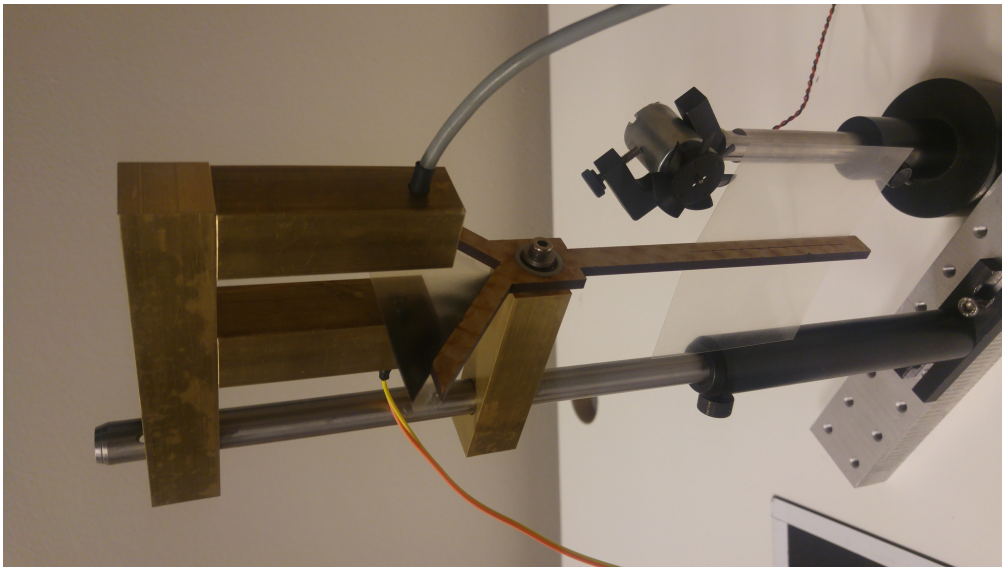


Figure 5.1: A photo of the mechanical part of the system.

Appendices

Appendix A

Mathematica code for creating gradient

```
(*The gradient will go from angle1 to angle2 *)
angle1 = 40;
angle2 = 140;

(*A function that is rises linearly with the angle*)
f[x_, y_] := Tanh[x/y]*180/Pi + 90

(*Alter the function so it runs from \[Theta]1 to \[Theta]2*)
f2[x_, y_] := Which[angle1 < f[x, y] < angle2, f[x, y], True, 140]

(*Plot the function, with a chosen colorfunction*)
DensityPlot[f2[x, y], {x, -3, 3}, {y, 0, 2},
ColorFunction -> "PigeonTones", AspectRatio -> 2/6]
```

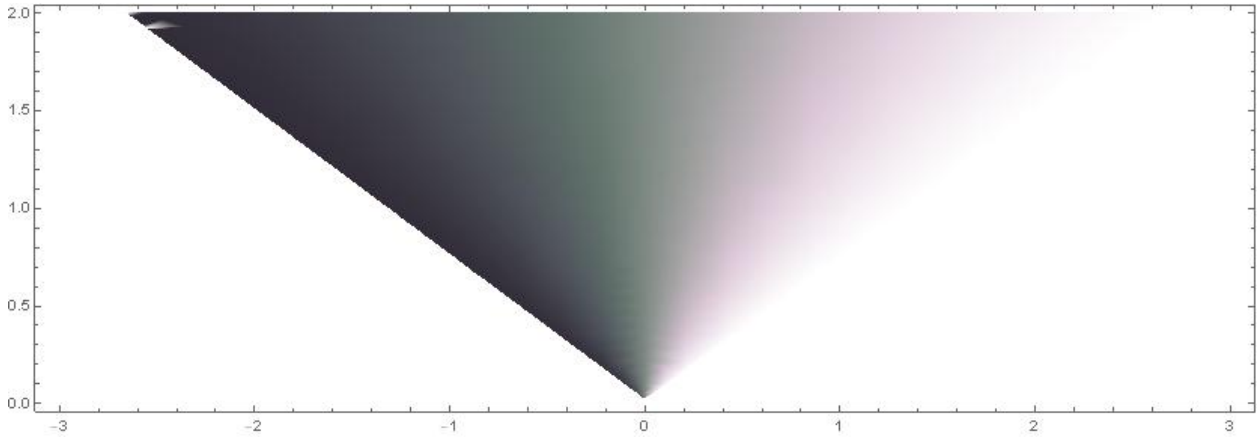


Figure A.1: Example of a gradient created by the code shown above
$angle1 = 40$, $angle2 = 140$

Appendix B

Mathematica code for processing data from the oscilloscope

```
(*Import the csv-file that contains the Oscilloscope's data. In this \
case we import two square waves with amplitude 1*)
lijst = Import[
  "C:\\Users\\Tom\\Google Drive\\Studie\\Scriptie\\Metingen \
14-1-17\\TimeDelay\\Delay (0 1).csv", "Table", "NumberPoint" -> ",",
  "FieldSeparators" -> ";"]; (*"FieldSeparators"\[Rule]";"*)

(*Drop the first 22 rows of data, these contain unrelevant data about \
the device. *)
values = Drop[lijst, 22];
 (*Delete the rows where one of the values is a string, these are \
rows that contain false data.*)
values = DeleteCases[values, {_, _, _, _String}];
values = DeleteCases[values, {_, _, _String, _}];
(*From the data, extract the times, source signal (directly from the \
Function generator) and the new signal (via the DAQ and PC)*)
time = values[[All, 2]];
time = time -
  time[[1]];  (*This is done to make sure time starts at 0*)
source = values[[All, 3]];
new = values[[All, 4]];

(*Plot the 2 signals in one plot, so they can be compared*)
ListLinePlot[{Transpose[{time, source}], Transpose[{time, new}]},
 PlotLegends -> {"Source Signal", "Output from DAQ"},
 PlotRange -> {{0, 6}, {-0.1, 1.2}},
 AxesLabel -> {"Time (s)", "Voltage(V)"},
 AxesStyle -> Directive[Black, 12]]

(*These are two loops that construct a list with the increase of \
voltage at each datapoint. Done for the source and new signal.*)
diffSource = ConstantArray[0, Length[source]];
n = 1;
While[n < Length[source] - 1,
```

```
  diffSource[[n + 1]] = source[[n + 1]] - source[[n]]; n++];

diffNew = ConstantArray[0, Length[source]];
n = 1;
While[n < Length[source] - 1,
  diffNew[[n + 1]] = new[[n + 1]] - new[[n]]; n++];

(*For both signals construct a list that contains the index of each \
element that is greater than 0.5. A voltage increase of more than \
0.5V means we are dealing with the start of a peak in the signal.*)
positionsSource = Flatten[Position[diffSource, _?(# > 0.5 &)]];
positionsNew = Flatten[Position[diffNew, _?(# > 0.5 &)]];

(*Sometimes measurements dont start at the start of a wave. So it can \
happen that the first peak of the source signal does not belong to \
the first peak of the new signal. Same for the last peak
Here, this is not the case, so I commented out the section
*)
(*positionsSource=Drop[positionsSource,-1];
Length[positionsSource]
positionsNew=Drop[positionsNew,1];
Length[positionsNew]
positionsNew-positionsSource*)

(*Now that we have the index of the start of each peak for each \
signal, we can get the time of each peak.
If we then substract the times we get a list of the delay between the \
two signals for every peak.
*)
delaylist = ConstantArray[0, Length[positionsNew]];
n = 0;
While[n < Length[positionsSource],
 delaylist[[n + 1]] =
  time[[positionsNew[[n + 1]]]] - time[[positionsSource[[n + 1]]]]; n++]
(*Plot the delay data*)
ListPlot[delaylist, Joined -> True,
 PlotMarkers -> Graphics@{Disk[{0, 0}, Scaled@0.015]},
 AxesLabel -> {"Wave no.", "Delay (s)"}, LabelStyle -> {18, Black}]

(*This section is the same as the above section, but now for the end \
of each peak (or: the start of each dip *)
diffSource1 = ConstantArray[0, Length[source]];
n = 1;
While[n < Length[source] - 1,
  diffSource1[[n + 1]] = source[[n + 1]] - source[[n]]; n++];

diffNew1 = ConstantArray[0, Length[source]];
n = 1;
While[n < Length[source] - 1,
  diffNew1[[n + 1]] = new[[n + 1]] - new[[n]]; n++];
```

```
positionsSource1 = Flatten[Position[diffSource1, _?(# < -0.5 &)]];
positionsNew1 = Flatten[Position[diffNew1, _?(# < -0.5 &)]];

(*In this case the first source dip does not belong to the first dip \
in the new signal, so we delete the first value of the source*)
positionsSource1 = Drop[positionsSource1, -1];
(*Length[positionsSource1]
positionsNew1=Drop[positionsNew1,1];
Length[positionsNew1]*)

delaylist1 = ConstantArray[0, Length[positionsNew1]];
n = 0;
While[n < Length[positionsSource1],
 delaylist1[[n + 1]] =
  time[[positionsNew1[[n + 1]]]] - time[[positionsSource1[[n + 1]]]];
 n++]
(*Again, plot the delay data, now for the delay in the end of each \
peak*)
ListPlot[Join[delaylist, delaylist1], Joined -> True,
 PlotMarkers -> Graphics@{Disk[{0, 0}, Scaled@0.015]},
 AxesLabel -> {"Wave no.", "Delay (s)"}, LabelStyle -> {18, Black}]

(*Plot the two graphs together*)
ListPlot[{delaylist, delaylist1}, Joined -> True,
 PlotMarkers -> Graphics@{Disk[{0, 0}, Scaled@0.015]},
 AxesLabel -> {"Wave no.", "Delay (s)"}, LabelStyle -> {18, Black},
 PlotLegends -> {"\[CapitalDelta]V Positive (wave going upwards)",
   "\[CapitalDelta]V Negative (wave going down)"},
 PlotStyle -> {Black, Dashed}]

(*Sum the peak- and dip-delay for each wave and plot it*)
min = Min[{Length[delaylist], Length[delaylist1]}]
delaylistB = Take[delaylist, min]
delaylist1B = Take[delaylist1, min]
ListPlot[delaylistB + delaylist1B, Joined -> True,
 PlotMarkers -> Graphics@{Disk[{0, 0}, Scaled@0.015]},
 AxesLabel -> {"Wave no.", "Delay (s)"}, LabelStyle -> {18, Black},
 PlotStyle -> {Black}]

(*Calculate the mean delay, standarddeviation included.*)
Mean[Join[delaylist, delaylist1]]
StandardDeviation[Join[delaylist, delaylist1]]
```

Appendix C

Python code to do time-research on DAQ-PQ communication

```
#=============================================================================
# Import libraries
#=============================================================================
import visa
import timeit   #This method does several runs and avarages to get time of part
import time          #This is more simple (just does one run), but also less exact
import numpy as np  #For doing mathematical work
import matplotlib.pyplot as plt      #For plotting data
from scipy.optimize import leastsq  #
import pylab as plt2     #Another package for plotting data

#=============================================================================
# Initialize device connection
#=============================================================================

#Open connection
rm = visa.ResourceManager()

#Connect to device
device = rm.open_resource('USB0::0x0957::0x1518::TW56100007::0::INSTR')

#=============================================================================
# Settings
#=============================================================================

#These are the setting which we wish to use:
device.timeout = 500000          #Set Timeout duration (maximum duration of an action)
nPoints      = 1                 #Number of points to be measured
channel_In   = 116            #The channel used for the measurement
range_In     = 10             #The expected voltage range of the signal
polarity_In = "BIPolar"      #The polarity of the signal

channel_Out = 202                          #Channel used to put out the voltage
#The range of the voltage out, we set this to match the input range:
```

```
range_Out=range_In
#We also set the output polarity to match the input polarity:
polarity_Out=polarity_In

#Warm-up measurements, the first measurement is always very slow.
device.query("MEASure:VOLTage? (@%d)" % (channel_In))
device.query("MEASure:VOLTage? (@%d)" % (channel_In))
#Send command to start measurement:
device.write("DIG")
#Send command to read data from buffer:
device.query_binary_values('WAV:DATA?', datatype='h', is_big_endian=False)

#Some additional settings, after the warmup measurements
nPoints         = 1                     #Number of points to be measured
        #Sampling rate of measurement (hz) (default 1000hz, using max):
rate            = 3000000

#Apply the settings by sending the right SCPI commands
device.write("SENSe:VOLTage:RANGe %d,(@%d)" % (range_In, channel_In))
device.write("SENSe:VOLTage:POLarity %s,(@%d)" % (polarity_In, channel_In))
device.write("ROUT:SCAN (@%d)" % (channel_In))
device.write("ROUT:CHAN:RANG %d,(@%d)" % (range_In, channel_In))
device.write("ROUT:CHAN:POL %s,(@%d)" % (polarity_In, channel_In))
device.write("ACQ:SRAT %d" % (rate))
device.write("ACQ:POIN %d"% (nPoints))

#=============================================================
# Calculate frequency of measuring input and/or output voltage
#=============================================================
#The three functions that can be used to measure the frequency are:
#RUN, DIGitize and MEASure:VOLTage
#The following loops send one of these commands to the card, as fast as possible
#       while we measure the time of each action.
#Each loop also contains a commented SOUR:VOLT command, this can be enabled to
#       get the frequency of measureing a voltage and then output that Voltage

        #Measure using the MEAS:VOLT command
def g(it):
        n=0
        while n<it:
                #Read with measure:voltage:
                val=float(device.query("MEASure:VOLTage? (@%d)" % (channel_In)))
                #Output the measured value
                #device.write('SOUR:VOLT %f, (@%d)' % (val, channel_Out))
                n=n+1

a=time.time()
g(1000)
b=time.time()
print("MEAS:VOLT f(Hz) =", 1000/(b-a))
```

```
        #Measure using the DIG-command
def g(it):
        n=0
        while n<it:
                #Send command to start measurement
                device.write("DIG")
                #Send command to read data from buffer:
                meas = device.query_binary_values('WAV:DATA?', datatype='h',
                                is_big_endian=False)
                #Convert value from 16bit Int to float:
                val = (2*meas[0]/(2**16))*range_In
                #Write the measured value:
                #device.write('SOUR:VOLT %f, (@%d)' % (val, channel_Out))
                n=n+1

a=time.time()
g(1000)
b=time.time()
print("DIG f(Hz) = ", 1000/(b-a))

        #Measure using the RUN-command
def g(it):
        n=0
        while n<it:
                #Send command to start measurement:
                device.write("RUN")
                #Send command to read data from buffer:
                meas = device.query_binary_values('WAV:DATA?', datatype='h',
                                is_big_endian=False)
                #Convert value from 16bit Int to float
                val = (2*meas[0]/(2**16))*range_In
                #Output the measured value:
                #device.write('SOUR:VOLT %f, (@%d)' % (val, channel_Out))
                n=n+1
#=============================================================================
# Time-research on voltage output using the command 'SOURce:VOLTage'
#=============================================================================
#Putting out a single voltage
def g(it,val):
        n=0
        while n<it:
                #Output the measured value
                device.write('SOUR:VOLT %f,(@%d)' % (val, channel_Out))
                n=n+1

a=time.time()
g(1000,1)
b=time.time()
print("Output voltage f(Hz) = ", 1000/(b-a))

#Output of two voltages right after each other:
```

```
def g(it, val1, val2):
        n=0
        while n<it:
                #Output val1:
                device.write('SOUR:VOLT %f, (@%d)' % (val1, channel_Out))
                #Output val2:
                device.write('SOUR:VOLT %f, (@%d)' % (val2, channel_Out))
                n=n+1

a=time.time()
g(1000,1,2)
b=time.time()
print("Output two voltages f(Hz) =" % (val), 1000/(b-a))
#=============================================================
# Create graphs with time of each action
#=============================================================
#This section contains I make a graph of the time each measurement takes.
#I do this for Run, Dig, MEAS:VOLT and SOUR:VOLT

        #---------------------
#RUN:
        #Set the amount of measurements to be in the graph:
nPoints=1000
n=0             #The initial n
H = np.zeros((nPoints,2))       #Generate empty list we are going to fill

#A loop that performs the action (RUN, in this case) nPoints times.
#For each iteration the loop fill a list with
#A) The number of the measurement
#B) The time on which the measurement to place
for n in range(nPoints):
        H[n,0]=time.time()
        device.write("RUN")
        #Send command to read data from buffer:
        meas = device.query_binary_values('WAV:DATA?', datatype='h',
                        is_big_endian=False)
        #Calculate average and convert to float:
        average = sum(meas) / float(len(meas))
        val = (2*average/(2**16))*range_In
        H[n,1]=n+1

#substract each time with the time of its predecessor,
                #to get the time each action took
H[:,0]=H[:,0]-H[0,0]
#Generate the plot and
#substract the time of each measurement with the time of its predecessor,
                #to get the time each action took
plt.figure(1)
plt.scatter(H[2:nPoints,1],H[2:nPoints,0]-H[1:nPoints-1,0])
plt.autoscale(enable=True, axis='both', tight=True)
plt.title("Time per single I/O operation")
```

```
plt.xlabel("Measurement no.")
plt.ylabel("Time (s)")

        #----------------------
#MEAS:VOLT
nPoints=1000
n=0
J = np.zeros((nPoints,2))
start=time.time()
for n in range(nPoints):
        J[n,0]=time.time()
        val=float(device.query("MEASure:VOLTage? (@%d)" % (channel_In)))
        J[n,1]=n+1
end=time.time()
J[:,0]=J[:,0]-J[0,0]
plt.figure(2)
plt.scatter(J[2:nPoints,1],J[2:nPoints,0]-J[1:nPoints-1,0])
plt.autoscale(enable=True, axis='both', tight=True)
plt.title("Time per single I/O operation")
plt.xlabel("Measurement no.")
plt.ylabel("Time (s)")

        #----------------------
#DIG
nPoints=1000
n=0
K = np.zeros((nPoints,2))
start=time.time()
for n in range(nPoints):
        K[n,0]=time.time()
        device.write("DIG")
        meas = device.query_binary_values('WAV:DATA?', datatype='h',
                        is_big_endian=False)
        #Send command to read data from buffer
        average = sum(meas) / float(len(meas))
        val = (2*average/(2**16))*range_In
        K[n,1]=n+1
end=time.time()
K[:,0]=K[:,0]-K[0,0]

plt.figure(3)
plt.scatter(K[2:nPoints,1],K[2:nPoints,0]-K[1:nPoints-1,0])
plt.autoscale(enable=True, axis='both', tight=True)
plt.title("Time per single I/O operation")
plt.xlabel("Measurement no.")
plt.ylabel("Time (s)")
plt.show()

#=============================================================
# Measure the frequency of DIG as a function of the packet size
#=============================================================
```

```
def g(it):
        n=0
        while n<it:
                #Send command to start measurement
                device.write("DIG")
                #Read data from the buffer
                #Multiple times when the amount of points get big,
                #because the data then gets stored in blocks of 10.000 samples
                a=device.query_binary_values('WAV:DATA?', datatype='h',
                                is_big_endian=False)
                b=device.query_binary_values('WAV:DATA?', datatype='h',
                                is_big_endian=False)
                c=device.query_binary_values('WAV:DATA?', datatype='h',
                                is_big_endian=False)
                n=n+1

a=time.time()
g(10000)
b=time.time()
print("DIG f(Hz) = ", 10000/(b-a))

#=============================================================================
# Errors+Reset+Close connection
#=============================================================================
print(device.query("SYST:ERR?"))
device.write("*RST")
device.close()
print("instrument connection closed")
```

Appendix D

Python code of the PID-control loop

```
# -*- coding: utf-8 -*-

#=============================================================================
# Import libraries
#=============================================================================
import visa
import timeit   #This method does several runs and avarages to get time of part
import time             #This is more simple (just does one run),
                                    #but also less exact
import numpy as np      #For doing mathematical work
import matplotlib.pyplot as plt         #For plotting data
from scipy.optimize import leastsq
import pylab as plt2    #Another package for plotting data

#=============================================================================
# Initialize device connection
#=============================================================================

#Open connection
rm = visa.ResourceManager()

#Connect to device
device = rm.open_resource('USB0::0x0957::0x1518::TW56100007::0::INSTR')

#=============================================================================
# Configuration of channel and measurement
#=============================================================================

device.timeout = 5000   #Set Timeout duration (maximum duration of an action)
channel_In  = 116               #The channel used for the measurement000
range_In        = 10                    #The expected voltage range of the signal
polarity_In = "BIPolar"         #The polarity of the signal

nPoints         = 1                     #Number of points to be measured
#Sampling rate of measurement (hz) (default 1000hz):
rate            = 3000000
```

```
device.write("ROUT:SCAN (@%d)" % (channel_In))
device.write("ROUT:CHAN:RANG %d,(@%d)" % (range_In, channel_In))
device.write("ROUT:CHAN:POL %s,(@%d)" % (polarity_In, channel_In))
device.write("ACQ:SRAT %d" % (rate))
device.write("ACQ:POIN %d"% (nPoints))

#========Warm-up measurement==========================================
        #Somehow the first instance of DIG/Measure:Voltage is very slow than the
        #subsequent instances.
device.write("DIG")
completed = 'NO'
while (completed == 'NO'):
        completed = (device.query("WAV:COMP?")).strip()

#=====================================================================
# The PID-loop
#=====================================================================

#Choose the setpoint: the value we want our process variable to have
setpoint=0.117
#choose the gain terms P, I and D
P=0
I=0
D=0

#k1,k2 and k3 are constants that come up when discretizing the PID-algorithm
k1=P+I+D
k2=-P-2*D
k3=D

#Choose the initial values of the control loop, because our PID algorithm
#looks to the past two values of the error, we need to start at n=2,
#we also need to choose the values the error had at the start of the loop.
n=2
values=[0,0]
errors=[setpoint,setpoint]
u=0
Umin=4
Umax=5

#The motor needs at least 4V to run, but to start it, it needs a little more
#Thats why we start the motor with a 5V pulse before initiating the control loop
device.write('SOUR:VOLT 5, (@201)')
print(4.4)
time.sleep(2)

while True:
        #Measure a voltage
        device.write("DIG")
        #Get the data from the buffer
```

```
        meas = device.query_binary_values('WAV:DATA?', datatype='h',
                is_big_endian=False)
        #convert the 16bit Int to float
        average = sum(meas) / float(len(meas))
        value = (2*average/(2**16))*range_In
        #calculate the error
        error=setpoint-value

        #add the error and system variable to their lists
        values.append(value)
        errors.append(error)

        #The actual discrete PID-function that calculates the control action
        delta_u=k1*(errors[n])+k2*(errors[n-1])+k3*(errors[n-2])
        u=u+delta_u

        #Set the boundary values of the control output
        if u>Umax:
                u=Umax
        if u<Umin:
                u=Umin

        #command the DAQ to apply the control output
        device.write('SOUR:VOLT %d, (@201)' % (u))

        n=n+1

#=============================================================================
# Errors+Reset+Close connection
#=============================================================================
print(device.query("SYST:ERR?"))
device.write("*RST")
device.close()
print("instrument connection closed")
```

Bibliography

[1] R. Routledge, *Discoveries and inventions of the nineteenth century.* Routledge, 2.10b ed., 1881.

[2] J. Bechhoefer, "Feedback for physicists: A tutorial essay on control," *REVIEWS OF MODERN PHYSICS*, vol. 77, 2005.

[3] T. H. Benzinger, "Heat regulation: homeostasis of central temperature in man.," *Physiol. Rev., 49*, 1969.

[4] K. J. Astrm and R. M. Murray, *Feedback Systems.* Princeton University Press, 2.10b ed., 2009.

[5] C. More, *Understanding the Industrial Revolution.* London ; New York : G. Routledge, 1 ed., 2000.

[6] P. Albertos and I. Mareels, *Feedback and Control for Everyone.* Springer, 1 ed., 2010.

[7] S. Y. S. Temel and S. Gren, "Discrete time control systems," *Recitation report 4*, 2013.

[8] J. Zhong, "Pid controller tuning: A short tutorial." Lecture notes, 2006.

[9] F. Haugen, *PID Control.* Tapir Academic Press, 2004. Chapter 4: Experimental tuning of PID controllers.

[10] T. Co, "Ziegler nichols tuning method." Lecture notes, 2000.

[11] *U2300A Series USB Modular Multifunction Data Acquisition Devices - Data Sheet*, 2014.

[12] *U2300A Series USB Modular Multifunction Data Acquisition Devices - Users Guide*, 4 ed., 2009.

[13] *U2300A Series USB Modular Multifunction Data Acquisition Devices - Programmer's Reference*, 7 ed., 2014.