Door Simulator

Creating automated tests

Annie Ydström

2020

Master thesis in
Electrical Measurements

Supervisors: Christian Antfolk, Johan Nilsson
Anders Löfgren

ASSA ABLOY

LUND UNIVERSITY

LTH

FACULTY OF ENGINEERING

Department of Biomedical engineering

Abstract

This report describes the development of a simulator for the simulations if industrial doors. The project has been conducted at ASSA ABLOY Entrance Systems, industrial door solutions in Landskrona.

The purpose of this thesis project was to design and implement a system that can replicate the behaviour of an industrial door on a test-rig. The design should follow requirements set by ASSA ABLOY such as function, ease of use and potential for further development and should include the result from measurements on real industrial doors at ASSA ABLOY's R&D department. The development process includes determining specifications, taking measurements, generating a concept, implementation of said concept, testing and further development.

For the measurements on the doors the position of the door was recorded using a magnetic encoder while the torque was measured using a strain gauge. The simulator was a program made for a preexisting rig that utilises the same type of encoder used during measurements on the doors and a servo motor to limit the torque on the rig. Encoder values read on the simulator rig was connected to torque values from the measurements and transformed into a torque limit set on the servo. Thus the rig could emulate the same behaviour as a real door.

The thesis has resulted in a simulation program that can simulate three doors, a process for making measurements on real doors that could then be simulated and a base which can be further developed by ASSA ABLOY. The program can be extended to include other doors within ASSA ABLOY and the program can also be extended to include more types of movements on the simulated doors.

Keywords: Python, Simulator, PicoScope, Industrial door, Torque measurements, MATLAB, LabView, Instrument control

Preface

The master thesis sought to develop a door simulator for ASSA ABLOY Entrance Systems, Industrial Door Solutions in the spring of 2020. This thesis concludes the engineering studies in Engineering Physics for Annie Ydström at LTH, Lund University.

Acknowledgements

Thank you to ASSA ABLOY for the opportunity to do my master thesis at their company and a big thanks to the R&D Team at IDDS for all their help with this project. A special thanks to my supervisor at ASSA ABLOY, Anders Löfgren for all the discussions and suggestions to make this project successful.

Also thanks to Mitsubishi Electric for their support during the setup of the simulation rig.

Finally, a thank you to the supervisors Christian Antfolk and Johan Nilsson, Lars Wallman, the examiner. Also, thanks to the opponent of this master thesis for your comments on this work.

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Purpose and goal . . . 1
- 1.3 Disposition . . . 1

2 Method **3**
- 2.1 Limitations . . . 4
- 2.2 Torque Measurement . . . 4
  - 2.2.1 Theory . . . 4
  - 2.2.2 Measurements . . . 10
- 2.3 Door Simulator . . . 13
  - 2.3.1 Setup . . . 14
  - 2.3.2 Theory . . . 15
  - 2.3.3 Calibration . . . 19
  - 2.3.4 Coding . . . 19
  - 2.3.5 Feedback . . . 23

3 Result **24**
- 3.1 Torque Measurement . . . 24
  - 3.1.1 Calibration . . . 24
  - 3.1.2 Matlab results . . . 25
- 3.2 Door Simulator . . . 34
  - 3.2.1 Calibration . . . 34
  - 3.2.2 Simulation program . . . 35
  - 3.2.3 Results from feedback . . . 35

4 Discussion and Conclusions **38**
- 4.1 Torque measurements . . . 38
- 4.2 Simulation program . . . 39
- 4.3 Further development . . . 40
- 4.4 Conclusion . . . 40
- 4.5 Final thoughts . . . 41

A Code: MATLAB **43**

B Code: Door Simulator **47**
B.1 Simulation runner 47
B.2 Installation runner 48
B.3 Simulator 49
B.4 Encoder 56
B.5 Servo 60
B.6 Communication 69
B.7 Help functions 73

Chapter 1

Introduction

1.1 Background

This master thesis is written for ASSA ABLOY Industrial Door Solutions. ASSA ABLOY manufactures industrial doors, some of which uses operators/motors to open and close the doors. The operators needs to meet certain requirements and therefore the operators/motors needs to be tested for different prerequisites such as different temperatures or insufficient power supply. As it can be cumbersome to build an entire door for testing, simulating a door would be a good solution. Simulations for doors that vary in size, weight and speed is necessary since ASSA ABLOY manufactures doors of different sizes and speeds.

ASSA ABLOY has a test-rig intended to be used for door simulations. The motor/operator is mounted on an axle and a servo motor is then used to add an external torque to this axle. This torque should be dynamically set depending on the position of the ”door”, thus simulating a door opening and closing.

1.2 Purpose and goal

The purpose of this project is to create a program which will handle the torque setting and keep track of the door position. The goal is to recreate a behaviour on the test-rig that resembles that of a real door.

1.3 Disposition

In order to achieve the goal the project and the report is split into three main tasks:

- The collecting and presenting of data from real doors.
- The creation of a simulation program using results from the measurements.
- The evaluation of the test-rig and simulation program.

Chapter 2

Method

The two main parts of the project, apart from the evaluation, are the torque measurements and the construction and coding of the simulation program. The torque measurements will serve as both input to the simulation program and as a way to evaluate the outcome of the door simulator.

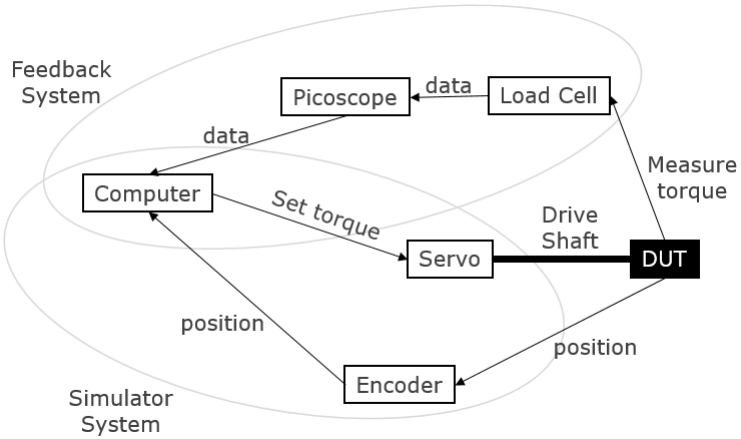


Figure 2.1: Figure shows an overview of the system.

Figure 2.1 shows a block diagram of the system with the simulator system and the feedback system and how they are connected to the DUT (device under test). Please note that the servo and the DUT are mounted on one drive shaft. The feedback system and the simulator system are not connected to each other, rather the feedback is evaluated manually.

2.1 Limitations

As the project spans over several areas some limitations will be necessary. These include:

Control of the DUT will be omitted since the program will be too complex to finish during this project.

What type of door-movements are to be included in the simulation. Some movements can be more complicated and hard to implement, such as reversing of the door and response to stop command. Therefore the type of movements that are to be included in this project are up (door opening) and down (door closing).

Enable code to run from Rasberry Pi and creating a GUI. This is not necessary for the simulator and is left as a possible improvement.

Automation of the feedback and incorporation of it into the simulator in order to improve the results. This will be left as a possible improvement for future development.

2.2 Torque Measurement

The purpose of the torque measurements is to fit a torque-value to a specific position as the door opens and closes. To read the position an encoder will be mounted on the same axle as the motor/operator and a sensor, a tension load cell, will be mounted in place of the "torque bracket", which holds the motor/operator in its place. For further information about the mounting of the sensor and the encoder see section "Mounting".

2.2.1 Theory

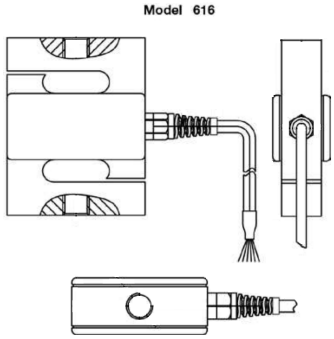
Sensors

The sensors used for torque measurements are of a strain gauge type in a tension compression load cell. It is used to measure the force the motor/operator puts on the "torque bracket" when opening and closing the door. The results can be used to calculate the torque. The gauge factor for a strain gauge k is given by equation 2.1 with resistance R end length L. This is then rewritten as relative

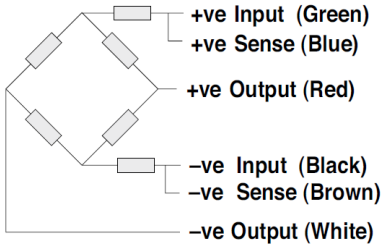
resistance r divided by the elongation [1].

$$k = \frac{\frac{\Delta R}{R}}{\frac{\Delta L}{L}} = \frac{r}{\varepsilon} \tag{2.1}$$

In general strain gauges are inclined to be affected by changes in temperature. To avoid this problem it is advised that the strain gauge is mounted in a Wheatstone-bridge to compensate for any variations in temperature [1]. The output of a strain gauge tend to be small which is why during measurements an amplifier is used. Figure 2.2a shows the load cell used. Note that two sensors are mounted on it, one on each side. The wiring of these sensors is shown in figure 2.2b which is a Wheatstone bridge composed of strain gauges and the additional sense wires, marked ”brown” and ”blue” in figure 2.2b, are installed do compensate for temperature effects and/or cable extension [2].



(a) Figure shows transducer used for measurements [2].



(b) Figure shows the Wheatstone bridge for the transducer 2.2a, [2].

Figure 2.2: These figures shows information about the transducer used for torque measurements [2].

Since the output from the sensors is measured in voltage rather than actual torque values it will be necessary to measure the output for a number of known torque values in order to calibrate the measurement arrangement and acquire a transformation constant. It is assumed that ratio between measured output from the sensors and

the torque value is linear (according to previous measurement done by ASSA ABLOY).

Encoder

The encoder used is a magnetic multi-turn absolute rotary encoder [3] and the general structure is shown in figure 2.3. Similarly to most rotary encoders the encoder used allows the shaft to rotate inside the housing where electronics are housed [4]. The magnetic encoder uses a series of magnetic poles (2 or more) to represent the encoder position to a magnetic sensor (typically magneto-resistive or Hall Effect) [3]. Being a multi-turn absolute encoder each new position read represents a number of complete turns in combination with the current angular position [5]. The magnetic sensor reads the magnetic pole positions to data values in this case the green and yellow data wires. The data wires, green and yellow, are connected to a RS485 to USB converter which converts the data using RS485 interface [6]. Values from the encoder are read through LabView (or using Python serial communication).

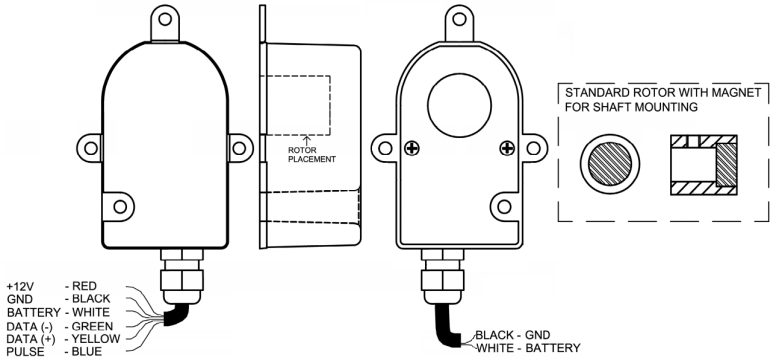


Figure 2.3: Figure shows the encoder used [7].

The functionality needed for this project is to read the current position of the encoder. According to the specification [7] this functionality is called Position request which is a byte consisting of "A0" written in hexadecimal form. The response to this request consists of 5 bytes; 0 (LSB) to 4 (MSB) represents the current position as

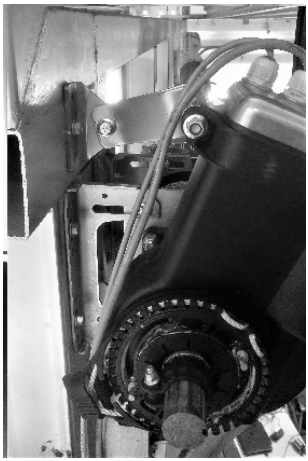
an unsigned integer (int32) written in hexadecimal form. The last byte represents the checksum of the previous bytes. A visual representation of the response can be seen below.

LSB			MSB	Xor Checksum
??	??	??	??	??

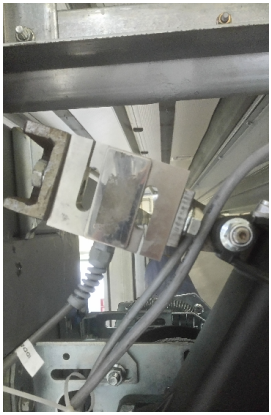
As the encoder has both a maximum speed of rotation of 6000 RPM and a maximum acceleration of 100 Rad/s^2 there is a resolution of 720 steps per turn [7]. There is also a limit to how often you can request the encoder value. According to the specification [7] the minimum time between requests is 6.4 ms which means the sample rate of the LabView program and the Door Simulator has its limits. However, this limit should be small enough not to affect the measurements as the top speed of the doors, 1 m/s or 117 RPM, is slow in comparison to the maximum RPM of the encoder.

Mounting

The strain gauge sensor is mounted as the ”torque bracket” as shown in figures 2.4a and 2.4b. The reason for mounting the sensor this way is to measure the resulting force from the motor/operator.



(a) Figure shows installation of the operator before mounting the sensor.



(b) Figure shows the complete mounting of the sensor

Figure 2.4: These figures shows the before and after of mounting the sensor.

Figure 2.5 shows the resulting forces on the load cell during opening and closing, please note that the figure only shows the resulting forces on the load cell and not the inner forces which are pointed in the opposite direction. In the figure red arrows show the resulting forces on the load cell, the black, curved arrows represents the torque of the motor and the blue arrows represents the door movement. Opening results in a negative voltage due to extension of the load cell. On the other hand the load cell will give a positive response during closing due to compression of the load cell.

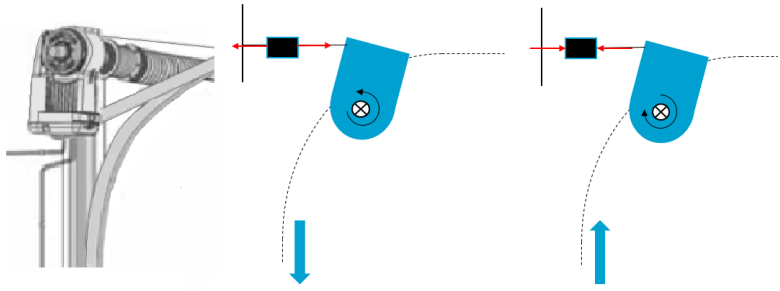


Figure 2.5: Figure shows the resulting forces on the load cell from door movements (not the inner forces). Part of the figure comes from the product sheet for overhead sectional doors [8].

The encoder is mounted on the axis as shown in figure 2.6. As the start-position is set by the user it is set to when door is in its opened position and set to zero, the position values as the door closes will be defined as negative. It is important to keep the encoder from rotating with the axle therefore a metal band is fastened to the apparatus and strapped to the structure holding the door.

Figure 2.6: Figure shows the mounting of the encoder.

2.2.2 Measurements

When performing torque measurements the equipment specified in section "Equipment" below needs to be installed as described in section 2.2.1 and then a calibration measurement is performed. Afterwards the measurement is taken in accordance with the measurement protocol described below in the section "Measurement protocol".

The doors used during torque measurements are A10, A20 and A60, all of type "Overhead Sectional Door" an example of which can be seen in figure 2.7. All of the doors are part of the test-hall for ASSA ABLOY in Landskrona. A table containing interesting data about the different doors is shown below in table 2.1.

Figure 2.7: Figure shows a general sketch of an Overhead Sectional Door [8].

Door	Size (WxH mm)	Weight (kg)	Operator
A10	8000x4500	800	HD
A20	8000x3500	400	STD
A60	5000x5000	301	MIO

Table 2.1: Information about the doors that are tested in this project.

The operators works with different speeds while opening and closing

the doors which will affect the outcome of the torque measurements. The HD (heavy duty) operator has a top speed of 0.18 m/s both up and down, STD (standard) has a top speed of 0.25-0.3 m/s up and down and the MIO operator has a top speed of 1 m/s going up and 0.7 m/s going down.

Equipment

The following is a list of all equipment used during the torque measurements.

- LabView 2015
- Dalmatic: Magnetic Multi-turn encoder with RS485 to USB converter
- Tension Compression Load Cell; Model 616; Tedea-Huntleigh
- Amplifier
- PicoScope 2205A
- PicoLog 6 Beta program
- Laptop computer

The encoder values were taken using a LabView program which utilises code from preexisting test-rigs at ASSA ABLOY. The existing VI's are used to set up the communication, write commands and read the current encoder value using LabViews inbuilt VISA VI's. Figure 2.8 shows the block diagram of the LabView program.

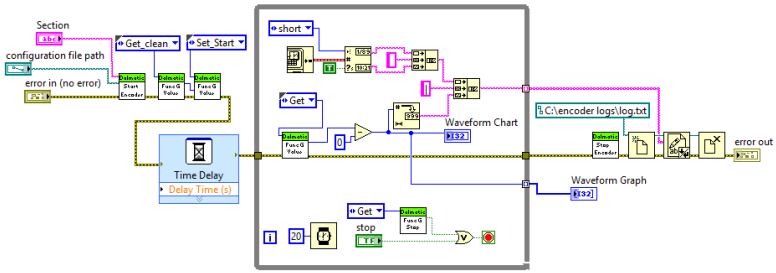


Figure 2.8: Figure shows main program from the VI which is designed for reading of encoder values.

Experiments during the first measurement will determinate the sampling rate to use for reading encoder values and reading with PicoScope. After running tests on A20 it was decided that the PicoScope should sample every 10 ms and the encoder reading program should sample every 20 ms. This is to avoid an unnecessary amount of duplicates of encoder values.

Calibration measurement

The torque measurements are given in mV and in order to present these values as torque one must measure the voltage for a couple of known torque values and then calculate the conversion factor. As the ratio is assumed to be linear, symmetric values on either side of zero will be used for the known torque values.

The measurement process is as follows: while measuring with the PicoScope a torque of known quantity is applied to the axle. The resulting voltage is noted for four different torques, at zero and two values during compression and extension respectively.

Measurement protocol

Perform the following steps when measuring the data for torque file.

- Make sure door is installed and at the top position
- Reset the amplifier
- Start recording encoder values
- Start PicoScope recording (5 min)
- Repeat the following sequence twice:
  - Close the door and hold closed position for 3 s
  - Open the door and hold opened position for 3 s
  - Close the door, at the 2m mark press STOP, hold for 3 s
  - Close the door and hold closed position for 3 s
  - Open the door, at the 2m mark press STOP, hold for 3 s
  - Open the door and hold opened position for 3 s

  - Close the door, at the 2m activate SAFETY to reverse the door

- Save the data in files marked as:
  "*door name*"_*meas*_"*measurement number*"

Creating the torque files

Measurements from the PicoScope is gathered in CSV files and the encoder values are collected in a text file. Both uses a timestamp on each value, making it possible to connect the encoder values to a torque value. In order to make vectors for each movement MatLab will be used.

During measurements it was observed that the measurements contained a lot of noise. The question asked was if an average of the torque values should be constructed and it was decided that an average from four measurements should be used.

Primarily the interest lies in the up- and down-movement therefore this part was separated from the rest. An average of four up- and down-movements was created to get a general picture and reduce noise. The values for the torque measurement was then filtered using a low pass filter due to high frequency noise on the signal. In order to select the cut-off frequency, a plot showing the FFT of the signal was made and the constants of the filter were then tweaked until the noise was reduced to an acceptable level. It was this result that was then used to evaluate results from the simulator rig. The filtered signal was then matched to an encoder value and written into a text file.

2.3 Door Simulator

In short the door simulator will work by letting the operator (DUT) run a "door" up and down at the same time as the encoder values are read to the computer. Using the encoder value and direction of movement the program looks up what torque should be set in a `torque file` and then writes a command to the servo. Every time the encoder value reaches the specified top value a counter for number of cycles should be increased by one. The user should be

able to specify the wanted number of cycles and when the simulator has reached this the program should automatically stop.

2.3.1 Setup

The mechanical setup for the door simulator consists of a rig, see figure 2.9, a general-purpose AC servo from Mitsubishi Electric [9] and an external encoder of the same type as used during the torque measurements. A laptop is used for controlling the system using a script written in Python3. The reason for writing the code in this language is to enable the program to be run this code from a `Rasberry Pi` through a `gui` similar to existing test-rigs at ASSA ABLOY.

The servo engine is controlled by sending commands with RS232C serial communication and the encoder will also be controlled through serial communication although using RS485 interface. The encoder will use code which should work similarly to the code used during torque measurements since it is the same type of encoder.

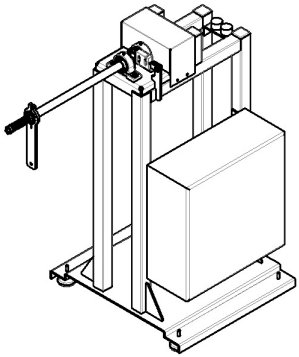
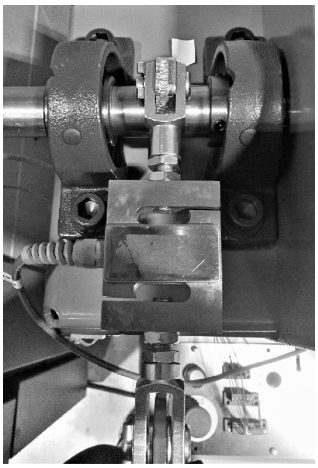


Figure 2.9: Figure shows an overview of the structure for the door simulator rig, curtesy of ASSA ABLOY Entrance Systems IDDS R&D. Other components such as the DUT and the Servo is mounted onto the structure.

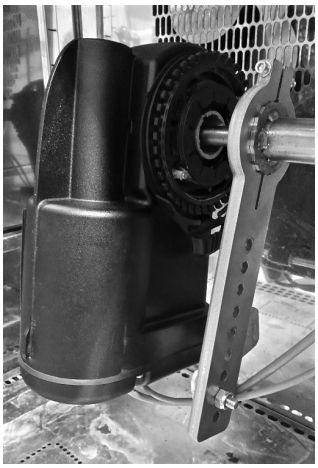
Mounting

In the following figures the mounting of important components to the door simulator system are shown. These include the DUT, the

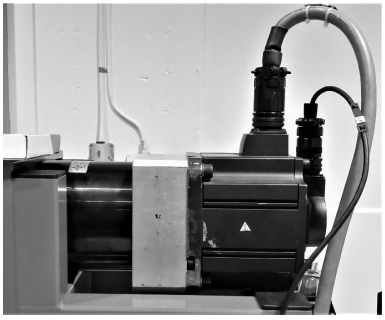
Servo and the load cell which is used for feedback.



(a) Mounting of load cell used for feedback measurements.



(b) Example of how a DUT is attached to the simulation rig.



(c) Mounting of the Servo.

Figure 2.10: Mounting of vital components on the simulation rig.

2.3.2 Theory

Serial communication

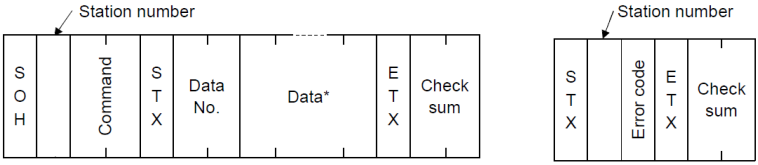
The in-built Python library for serial communication will be used for control of the servo and the encoder. Data-sheets for each instrument specifies which settings will be used when establishing contact trough serial communication. The communication with each in-

strument will be kept in separate threads using Python threading in order for the communication not to overlap each other. The data-sheets also specifies the communication protocol for each instrument, see section "Communication protocol".

Communication protocol

The protocol for communication with the encoder using RS485 is described in previous chapter 2.2.1. The code for the door simulator is written to be used in a similar fashion to the LabView code used during the torque measurements.

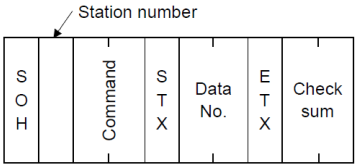
The protocol for communication with the servo uses RS232C where the computer acts as the master to the slave (the servo). There are two types of messages that can be sent to the servo, each with its own response, the transmission of data from the master to the slave and the request of data transmission from the slave to the master. Figure 2.11 shows the structure for sending data to the servo and its answer which contains an error code showing if the servo was able to process the message and/or if an error occurred. Figure 2.12 shows how the data request message and the answer are structured. Please note that the answer in figure 2.12b also contains an error code similarly to figure 2.11b.



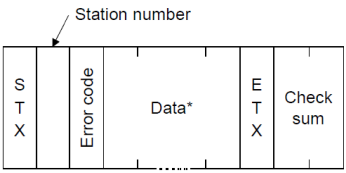
(a) Figure shows data transmission from master.

(b) Figure shows data transmission answer from slave.

Figure 2.11: These figures shows data transmission from master to slave [9].



(a) Figure shows data request from master.



(b) Figure shows answer for data request.

Figure 2.12: These figures shows data transmission from slave to master [9].

Every number/symbol correlates to a data frame in the message transmitted or received. The command and the data number is represented by two frames each whereas the transmitted data can consist of 4, 8, 12 or 16 frames depending on the command. A list of all parameters and their functionality is described in section 5.1.2 of the instruction manual [9]. Each number/symbol must be transmitted in hexadecimal (ASCII code). For example if the decimal number 155 is to be transmitted it must be converted into hexadecimal, 9B, and then all symbols must each be converted into ASCII numbers, 39 and 42. The translation will be as follows $155 \rightarrow 9B \rightarrow 39, 42$.

The method for calculating the checksum is to add up all numbers of the message (in hexadecimal form), taking the two lower digits of the sum and convert each character into its hexadecimal equivalent. See section 14.5 of the instruction manual [9].

A lot of the data transmitted from computer to servo is the setting of parameters, command [8][4] and data number [0][0] to [5][4]. The data transmitted is to be written in accordance with figure 2.13 [9]. An important thing to take into consideration is the write mode. If a parameters needs to be set more than once every hour it is recommended to write to RAM rather than EEPROM [9].

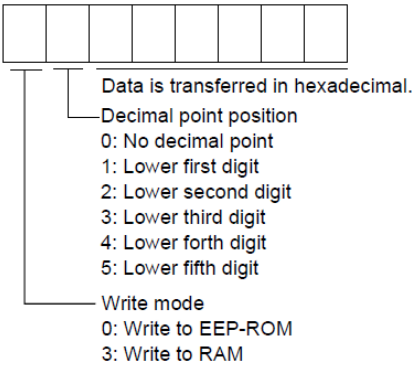


Figure 2.13: Figure shows the structure for data when writing to a parameter [9].

Using results from torque measurements

Torque values are stored in a file were a torque value is assigned to an encoder value. The script should continuously read the current position and direction of movement and use this value to collect the corresponding torque value from the torque file.

Setup on Servo

The purpose of the servo is to limit the torque on the drive shaft, in practice the servo will create a torque to counteract that of the DUT such that the total torque on the drive shaft resembles that measured on the real doors. To do so some settings needs to be made by setting the parameters 0, 19, 20 and 28 [9]. Parameter 0 sets the mode of operation which in this case is speed mode. Parameter 19 limits which parameters can be set, by default only parameters 0 to 19 can be set. Therefore the parameter needs to be set to 000C in order to enable the setting of additional parameters 20 to 49. While the servo is running and the shaft is stationary the default operation is to have the servo maintain the stop position. This is not ideal since this position may not be the current. Instead you can select parameter 20 so that the stop position is not maintained and the speed of 0 r/min is performed. Finally, in order to limit the torque parameter 28, internal torque limit is set. These settings were done according to recommendations from Mitsubishi Electric.

The torque limit is set in percentage form, 0 to 100%, were 100 % is the maximum torque and 0 means that no torque is produced, [9].

2.3.3 Calibration

As mentioned, the internal torque limit of the servo is set in percentage levels and in order to convert the measurements from the previous chapter, a calibration is required. This is done by setting an inner torque limit, in percent, and measure the torque needed to rotate the axle using a torque wrench, values given in Nm. These values were plotted and a linear equation was fitted to them.

Using this equation combined with the translation constants from the calibrations on the doors a torque value can be translated from mV to %, see equation 2.2.

$$torque\_limit = \left| \frac{torque \cdot \frac{1}{c} + b}{a} \right| \tag{2.2}$$

Here *torque* is the measured torque value in mV, a and b are the coefficients to the linear equation from the rig-calibration and c is the translation constant from the door calibrations. The torque limit is rounded of to the nearest integer due to restrictions in setting range on the servo [9].

2.3.4 Coding

The code is developed continuously throughout the duration of the project. For the purpose of debugging small unit tests will be created for each class, testing the different functionalities of each class.

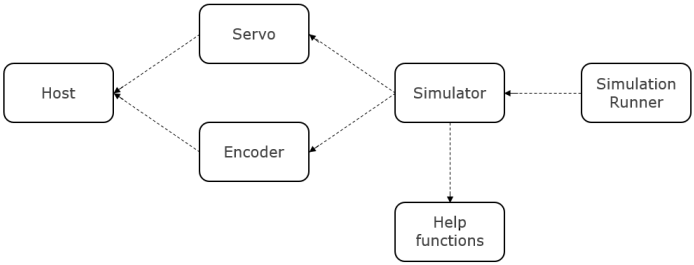


Figure 2.14: Figure shows a UML diagram of the code structure.

The simulation program consists of these five parts each written as separate Python files.

- Simulation runner
- Simulator Class
- Encoder Class
- Servo Class
- Communication Class
- Help Functions

An important part of the simulation program are the torque files which connects a torque value to a specific door position i.e. encoder value. The torque files are written as a text-file and split into two parts to represent the two directions in which a door can move; `up` and `down`.

Set the right torque value

In order to set the correct torque value the program uses the torque files created from the torque measurements. When creating the `Simulator` object a local dictionary (list) is created by running the help function `load_torque_file` which reads a torque file depending on the defined operator- and door-type. While the simulator is running, it will continuously read the current value and direction of movement from the encoder, using these values and to read the correct torque value from the created dictionary using `get_torque_val`.

Communication between computer and instruments

Since the encoder and the servo utilises RS485 and RS232C respectively the communication is better kept separate. Each instrument defines its own class, `Servo` and `Encoder`, setting up the communication on a separate thread in the `Communication` class. Both classes will contain basic functions for reading and writing to the instruments as well as some more complex functions which are meant to be used for handling exceptions, encoding messages etc.

Running a simulation

When starting a simulation the user should use the `Simulation_runner.py` file, not forgetting to set the variables `door`, `operator`, `encoder portname`, `servo portname` and `nbr_of_cycles`. The Simulator runner will create an `Encoder` object, a `Servo` object and a `Simulator` using these set variables. If the user has specified it the simulator will do an installation of the door, otherwise the simulator will clear the E24 error code (which occurs when the power to the DUT is toggled) by running the DUT to it's top position. A simulation will then be run for the specified number of cycles. The program will accept a keyboard interrupt and in that case stop the running of the simulator.

In figure 2.15 a sequence chart over the system can be seen. The process displayed is that of a user starting to run the simulator from the `Simulation_runner.py` file.

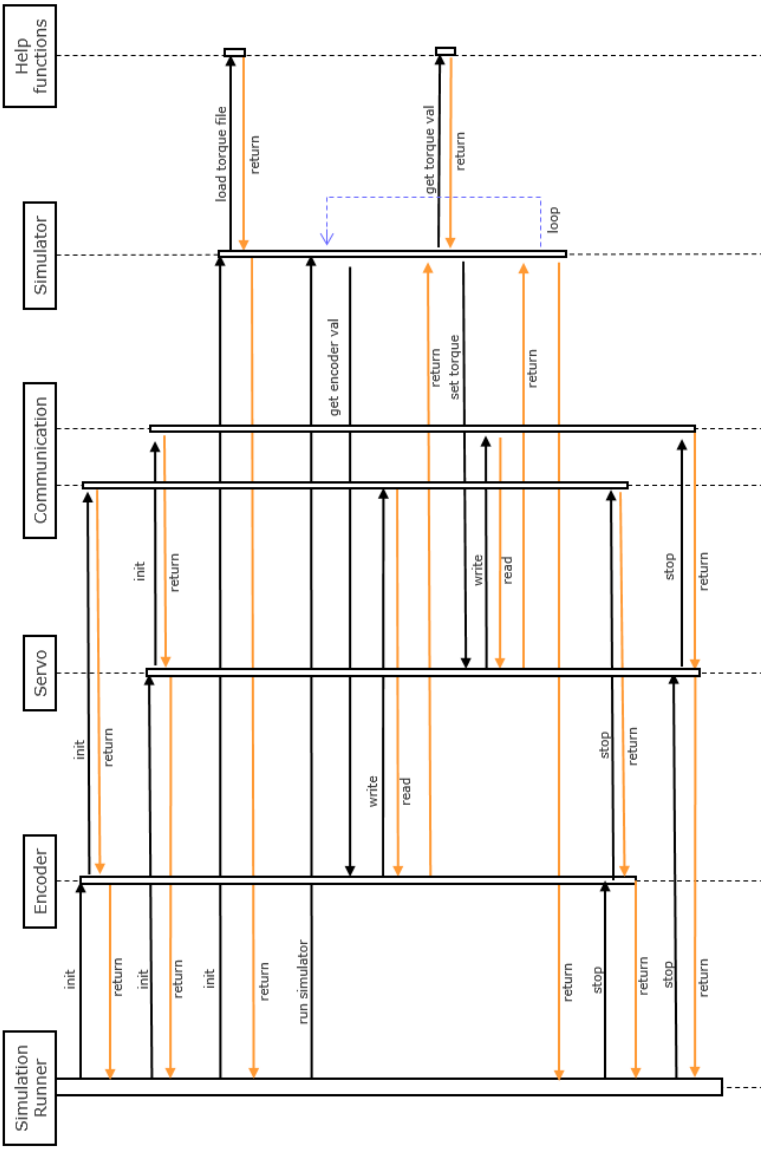


Figure 2.15: Sequence chart.

2.3.5 Feedback

In order to evaluate how well the simulator program can emulate the behaviour of an actual door, a tension load cell will be mounted on the rig and used similarly to the previous torque measurements, see figure 2.10a. The load cell will be connected to an amplifier and the output signal will be measured through a picoscope. This means it will measure the output from the simulation rig during the running of the program. The output will then be compared to the average of the movements which was calculated during the creation of the torque files.

Chapter 3

Result

3.1 Torque Measurement

The following is results from calibration of doors on which the torque was measured, the torque measurements and the handling of the measurement data.

3.1.1 Calibration

Tables 3.1 to 3.3 shows the result from calibrating the different doors. The known torque was varied due to differences in measurement results.

Torque $[Nm]$	**Voltage** $[mV]$
0	1.7 ..3.5
+25	128
-25	-153.79
+50	260.87
-50	-283

Table 3.1: Calibration measurement on A20.

Torque $[Nm]$	**Voltage** $[mV]$
0	0
+50	83.17
-50	-56.29
+100	303.64
-100	-325.16

Table 3.2: Calibration measurement on A10.

Torque $[Nm]$	**Voltage** $[mV]$
0	0
+50	107.33
-50	-200.63
+100	427.76
-100	-454.63

Table 3.3: Calibration measurement on A60.

A plot was done for each calibration measurement and with MatLab's basic fit functionality a linear fit was done. The function for the linear fit can be seen in equation 3.1. The constants to each

fitted function for each measurement are displayed in the table 3.4.

$$y = a \cdot x + b \tag{3.1}$$

Door	a	b
A10	2.7941	1.072
A20	5.4781	-9.244
A60	4.1455	-24.034

Table 3.4: Result from basic fit - linear function

When constructing the code these values for a were used in order to calculate the torque in percent, see equation 2.2.

3.1.2 Matlab results

This section contains results from Matlab calculations of the measurements. Both raw and processed are displayed.

A20

The following results are presentation and calculations from measurements on door A20. Figure 3.1 and 3.2 shows the raw measurement data. The figure 3.4 is a mean of all sequences in 3.2 that corresponds to the door closing and opening, there are 4 of them. These values are then filtered using a FIR-filter with an order of 100, a passband-edge frequency of 100 Hz and a sampling frequency of 96 kHz. These settings where decided upon by first plotting the FFT for one of the torque measurements and then some experimentation was done until the filtered values looked alright. The magnitude response of the filter can be seen in figure 3.3, and plotted in figure 3.5.

A comparison was made for down and up movements between unfiltered and filtered measurement data. This comparison can be seen in figure 3.6. Finally a plot consisting of encoder values and torque values can be seen in figure 3.7. Please note that since the number of encoder values was twice that of the torque values for each encoder value a mean of two torque values was calculated.

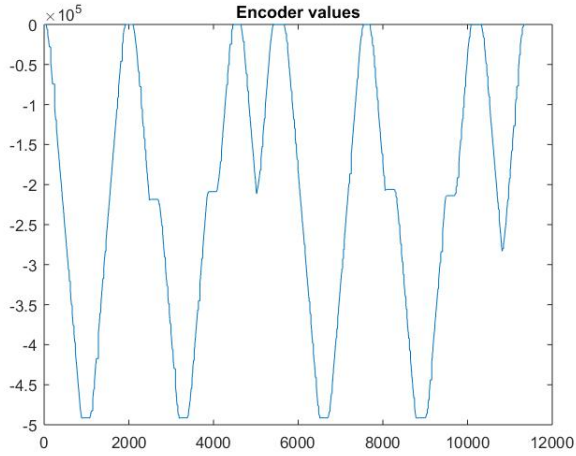


Figure 3.1: Figure shows the encoder value for each sample.

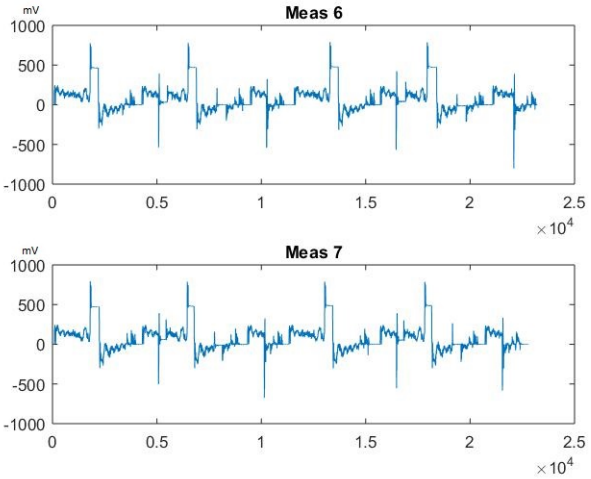


Figure 3.2: Figure shows the torque value in mV for each sample from the two measurements which followed the measurement protocol in section 2.2.1.

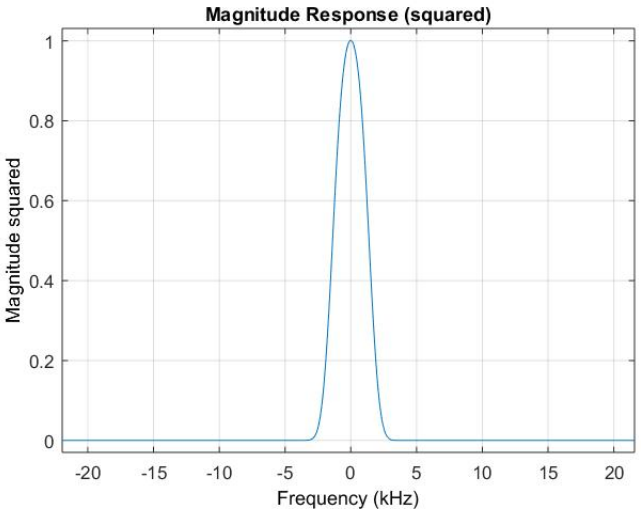


Figure 3.3: Figure shows the magnitude response for the filter used.

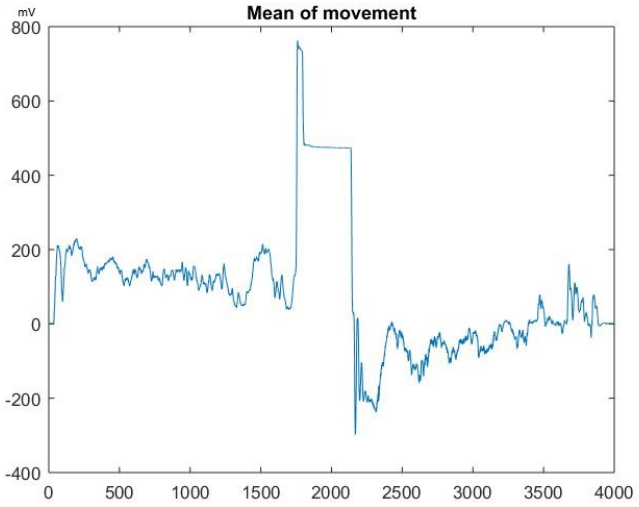


Figure 3.4: Mean of all up and down movements for each sample from measurements in figure 3.2.

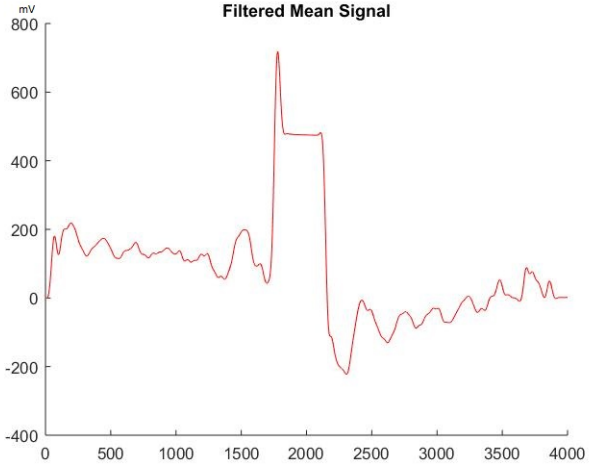


Figure 3.5: Filtered signal from figure 3.4 using the filter in figure 3.3.

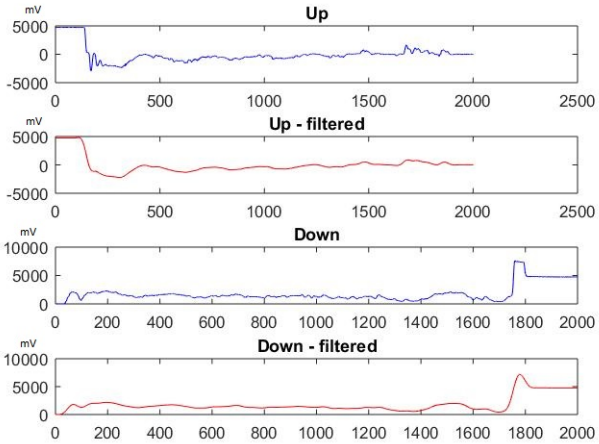


Figure 3.6: Comparison between filtered and unfiltered mean of up and down movements for every sample.

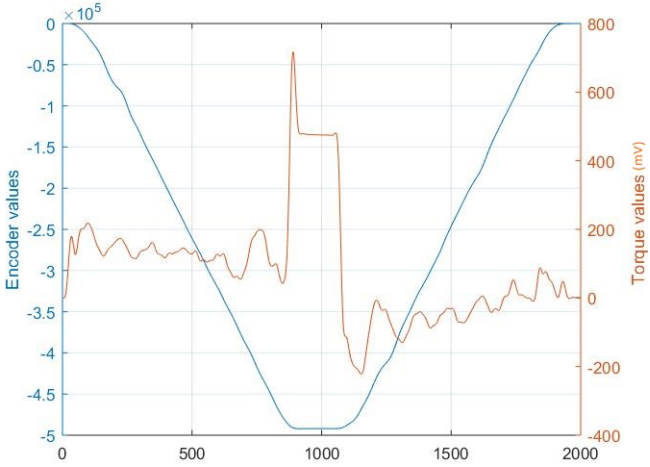


Figure 3.7: Torque versus encoder values for each sample.

A10

In figure 3.9 the mean of up and down movements from measurements on A10 can be seen. These values were filtered using the filter in figure 3.3 and the result is displayed in figure 3.10. Finally the encoder values from figure 3.8 and the torque values from figure 3.10 which were used to create the torque file for door A10 are displayed in figure 3.11.

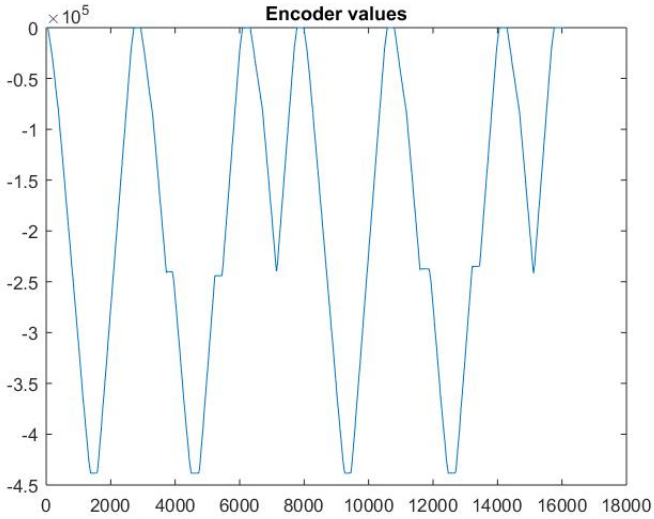


Figure 3.8: Figure shows the encoder value for each sample.

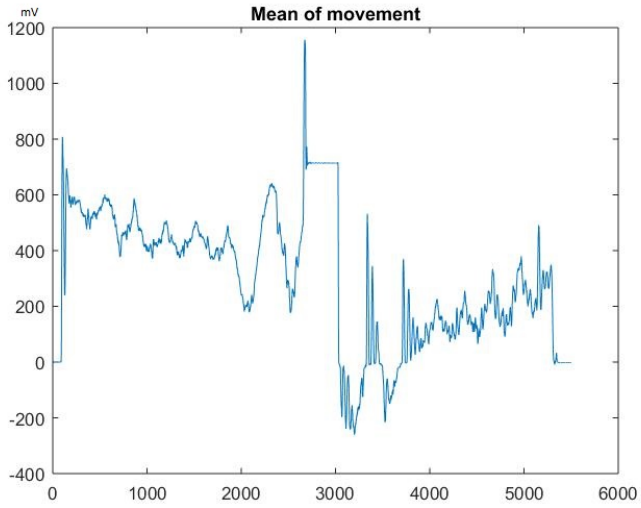


Figure 3.9: Mean of all up and down movements for each sample.

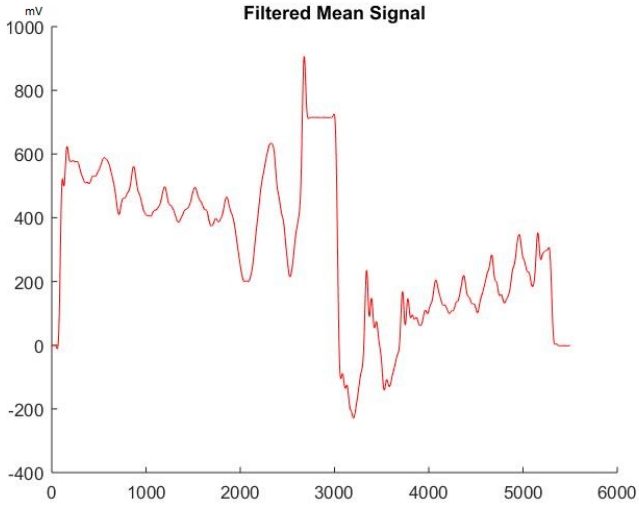


Figure 3.10: Filtered signal from figure 3.9 using the filter in figure 3.3.

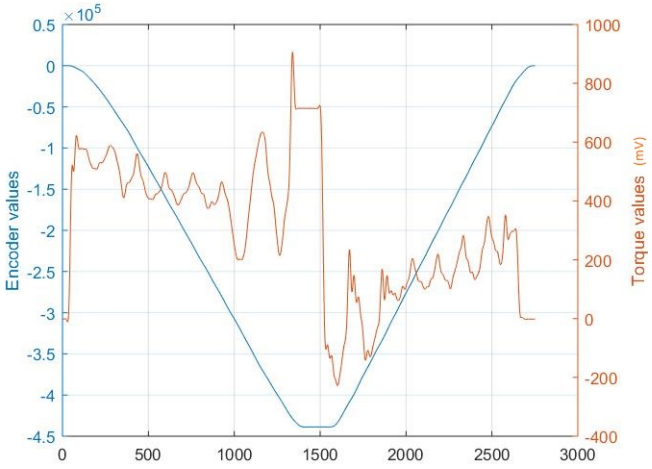


Figure 3.11: Torque versus encoder values for each sample.

A60

In figure 3.13 the mean of up and down movements from measurements on A10 can be seen. These values were filtered using the filter in figure 3.3 and the result is displayed in figure 3.14. Finally the encoder values from figure 3.12 and the torque values from figure 3.14 which were used to create the torque file for door A10 are displayed in figure 3.15.

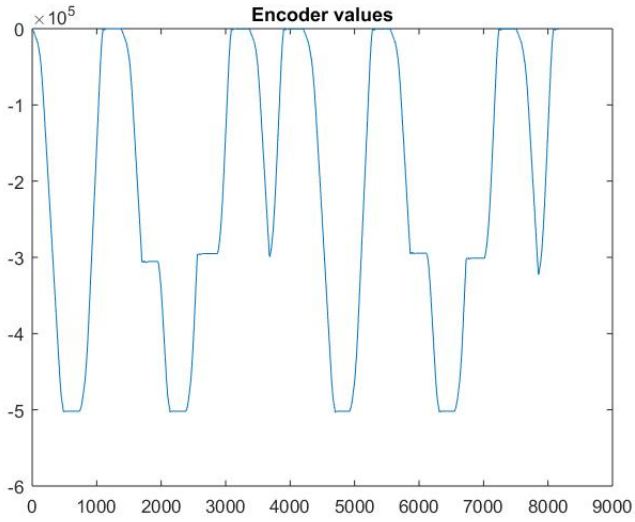


Figure 3.12: Figure shows the encoder value for each sample.

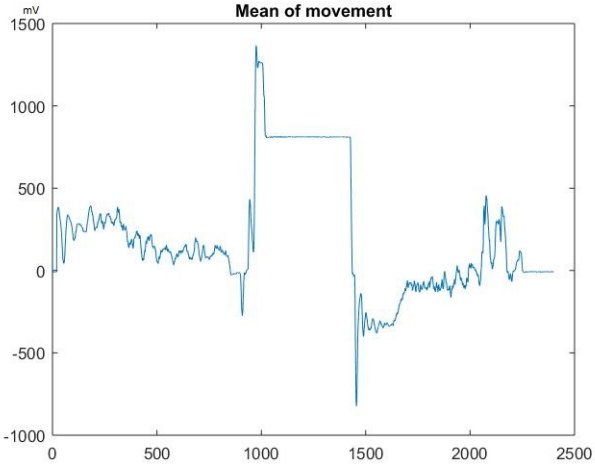


Figure 3.13: Mean of all up and down movements for each sample.

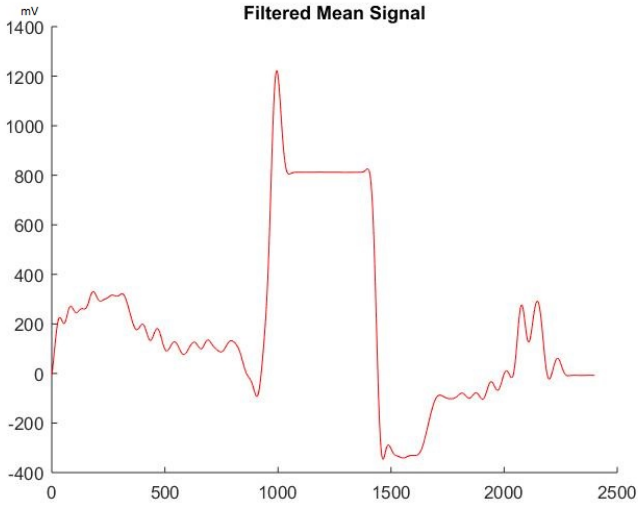


Figure 3.14: Filtered signal from figure 3.13 using the filter in figure 3.3.

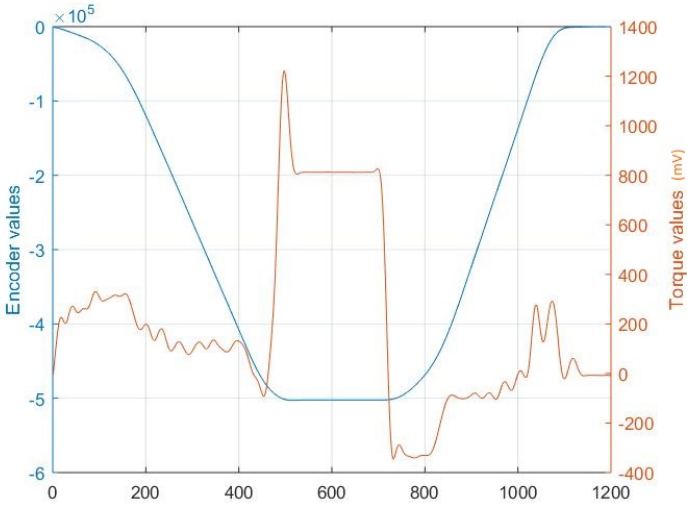


Figure 3.15: Torque versus encoder values for each sample.

3.2 Door Simulator

The following are results from calibration of the simulator rig, the result of the simulation program and also the results from the feedback. All results comes from running tests on a STD motor and therefore the torque file from measurements on A20 was used.

3.2.1 Calibration

In table 3.5 the measured values for the specified internal torque limit can be seen. These were plotted and a linear equation was fitted to the values, equation 3.2. The torque was also measured while the servo was turned off and a load voltage corresponded to 13 Nm.

Internal torque limit [%]	1	2	3	4	5	6	7	10	15
Output on torque wrench [Nm]	13.2	20	27	33	39	44	51	71	104

Table 3.5: Internal torque limit and the corresponding output on the torque wrench.

$$y_{rig} = 6.434 \cdot x + 6.7997 \tag{3.2}$$

The constants from equation 3.2 where used as a and b in equation 2.2.

3.2.2 Simulation program

The final version of the code for the door simulator can be found in appendix B. While testing the code could finish the wanted number of iterations without any false positives. Interesting information and instructions to the user are continuously written in the terminal window.

The code works as intended with some limitations such as the resolution of the set torque function on the servo and the resolution of the sampled encoder values. The latter can be partly improved by using a higher sampling rate. The resolution of the encoder values is also dependant on the DUT as different DUTs can either have a higher or a lower speed of rotation.

Initial findings when running the code showed that the DUT could have difficulties opening the door due to the torque limit being set too high when the DUT tried to open the ”door”. This problem was addressed by introducing a constant resistance for the first 20 000 encoder values from the bottom position.

3.2.3 Results from feedback

The results from the feedback system will be used to evaluate how well the simulation program emulates the behaviour of a door.

Calibration

In table 3.6 the calibration of the tension load cell used on the simulation rig for feedback can be seen.

Nm	mV
0	29
25	-813
-25	958
50	-1710
-50	1800

Table 3.6: Results from calibration of the tension load cell on the simulation rig.

Feedback measurements

Two problems occurred while trying to use the feedback system. Firstly, the amplifier intended for the feedback system had a too narrow bandwidth to be able to capture the behaviour of the simulator rig. In order to solve this the amplifier was replaced with the same one used during the torque measurements.

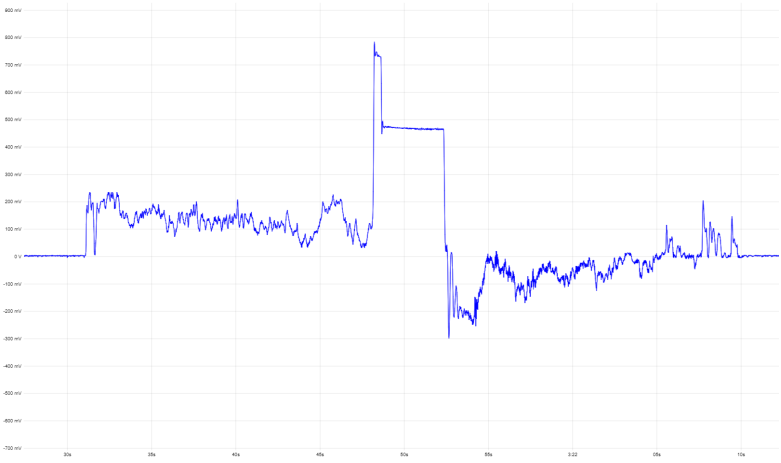


Figure 3.16: An example of a wanted output from the feedback system while running the simulator.

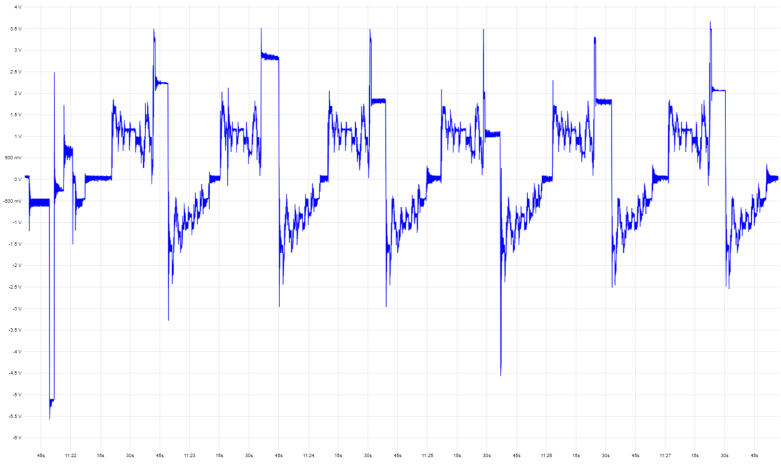


Figure 3.17: Output from feedback system while simulating door A20 with STD motor.

The results from the feedback measurements incited smaller changes to the code. A result from running a simulation of six cycles on the simulator using the final version of the code can be seen in figure 3.17. This can the be compared with the wanted signal seen in figure 3.16. The wanted signal is an example of an down and up movement from torque measurements on A20. Output from the feedback system can also be compared with the mean of movement in figure 3.4.

One big difference is the scale of the the y-axle which can be explained with the different mounting of the load cell. As to the look of the curve there are three main differences; the plateau after the the peak is at different levels, the bumps are more flat during the simulation compared to in real life and the big dip as the door starts to move upwards is not present in the real life measurements.

Chapter 4

Discussion and Conclusions

4.1 Torque measurements

When analysing the torque measurements there are some questions to keep in mind:

- What are the main differences and similarities?
- What does the torque measurements say about the behaviour of the door?
- What could be improved?

Some behaviours that can be observed for all of the torque measurements are the peaks that represents the sections of the door going through the bend of the door, compare all figures representing the mean of movement, figure 3.9, 3.4 and 3.13. As the control unit of the doors are programmed to keep a consistent torque throughout the movements the measured torque values for up and down movements are steady with some minor peaks due to the door-sections. Another similarity is the thin peak right before the plateau which represents the floor. This is the engine/motor breaking.

Looking back to the calibration measurements, the values for negative torque values (compression) are generally larger than the positive torque values (expansion). This is likely an inbuilt behaviour of the sensor.

When looking at the differences between measurements on different doors one can make a link to the behaviour of the door itself. Comparing figures displaying the encoder, figure 3.8, 3.1 and 3.12, the length of time it takes for the door to open and close is implied. Since the sample rate is the same, the more values there are, the movement takes longer to terminate. The maximal velocity of the door could also affect the amount of disturbances and the smoothness of the transitions between moving and standing still.

As to the improvements that could be made, firstly the LabView program used could be remade using the created encoder class from the simulator program to do the encoder/position measurement instead. This because ASSA ABLOY wants to use Python instead of LabView. Potentially the resolution of the time-stamps could be improved with coding through Python rather than LabView.

4.2 Simulation program

The three main questions asked when evaluating the simulation program are;

- Is the program easy to use?
- How well is it working?
- What can be improved?

In answer to the first question; the Simulation runner is kept simple so the user can set up a simulation with the wanted variables. Throughout the simulation instructions and useful information is written in the terminal window. This makes the program fairly easy to use to those who has some programming experience. An executable file would make it easier for those who are not familiar with using either a terminal window or Visual Studio Code (the editor of choice at ASSA ABLOY).

The second question relates to the reliability of the program; meaning if the code produces consistent results. This is important as the user will want to know if the simulator actually emulates the same behaviour every time. In order to answer this question one must look at the result in figure 3.17 and compare it to figure 3.16. What can be seen is the graph is similar although the simulator fails to deliver the exact same behaviour every simulation. This is likely due to limitations in both reading the encoder values and, more importantly, the limitations in the setting of the torque on the servo. However the simulator makes a good approximation of a door movement.

The improvements that could be made include, apart from those already mentioned above, enabling the setting of parameters such as the waiting time between each setting of the torque, refer to

run_simulator in the Simulator class.

4.3 Further development

There are several things that can be done to improve and expand the simulation program. The first thing that should be done is to automate the control of the DUT to enable simulations that can be run throughout the night. Secondly the ease of use could be improved, a GUI could be created and the code could be adapted to run on a Raspberry Pi. This would also make it possible to include simulations of STOP and SE cycles.

Several interesting data points could be collected and logged to a database. This could include opening and closing times, the feedback values, total number of cycles, what door type has been simulated and temperature of the DUT (if temperature sensors are mounted). With the code running on a Raspberry Pi and logging results on to a database these results could be displayed on Grafana (like other tests at ASSA ABLOY).

A possible area for expansion is the inclusion of torque files which have been measured on the same doors but for different setups of these doors. This could include differences in balancing of the door, torque setting of the operator/motor and other changes the engineers find interesting.

4.4 Conclusion

The door simulation program can approximate the characteristic behaviour of a door. Changes in the code could improve the approximation and the introduction of a servo and encoder with better resolutions would improve the results further.

This master thesis project has also introduced a protocol for making torque measurements on doors with results that can be used to simulate the door.

4.5 Final thoughts

Notes on what has been done and how it was done were made throughout this project which helped with the documentation process and the planning of the project. The documentation of this project is necessary for further development and continued use. It also helped with establishing what needed to be done, and also the order in which to do all the steps. All in all this project has helped develop skills around planning, construction and testing of a system for running automated tests.

Bibliography

[1] Grahm, L. Jubrink, H. Lauber, A. *Modern Inustriell Mätteknik; Givare* Studentlittteratur AB, 1996, 2007

[2] Tension Compression Load Cells, Model 615 and Model 616, Tedea-Huntleigh, Document No.: 12066, Revision: 25-Mar-2018

[3] Wikipedia. *Rotary Encoder*. Last update: 2020-03-06 `https://en.wikipedia.org/wiki/Rotary_encoder` [2020-04-28]

[4] MachineDesign. E. Eitel, *Basics of Rotary Encoders: Overview and New Technologies*, Written: APR 12, 2007 `https://www.machinedesign.com/automation-iiot/sensors/article/21831757/basics-of-rotary-encoders-overview-and-new-technologies` [2020-07-30]

[5] MachineDesign. R. Repas, *Multiturn absolute encoders*, Written: MAY 07, 2014 `https://www.machinedesign.com/archive/article/21813207/multiturn-absolute-encoders` [2020-07-30]

[6] Wikipedia *RS-485*. Last update: 2020-05-18 `https://en.wikipedia.org/wiki/RS-485` [2020-05-26]

[7] MTM-E-V.0 MULTITURN MAGNETIC ABSOLUTE ENCODER, DALMATIC TNV, 2017

[8] *Produktdatablad Takskjutport ASSA ABLOY OH1042P*, ASSA ABLOY Entrance Systems, Last revision: 2017

[9] General-Purpose AC Servo, MR-J2S-A Instruction manual, Mitsubishi Electric, Last revision: Dec. 2007

Appendix A

Code: MATLAB

```
close all
clear all

%% ---- Fetch values ------
delimiterIn = '|';
encoder_file = 'a60_meas_1.txt';
E = importdata(encoder_file, delimiterIn);

E_v = E.data(:, 1);
figure(); plot(E_v); title('Encoder values')

%read data: Import columns as column vectors
meas_file = 'A60_meas_1.csv';
T = readtable(meas_file);
T_v = T(2:end, 2);
T_v = table2array(T_v);
T_v = T_v.*-1;

meas_file = 'A60_meas_2.csv';
T = readtable(meas_file);
T_v2 = T(2:end, 2);
T_v2 = table2array(T_v2);
T_v2 = T_v2(1:end);
T_v2 = T_v2.*-1;

figure, subplot(211), plot(T_v); title('Meas 1')
subplot(212), plot(T_v2); title('Meas 2')

%% ---- Filter -------
n = length(T_v);
T_v_fft = fft(T_v);
```

```
power = abs(T_v_fft).^2/n;    % power of the DFT

figure;
plot(power);
xlabel('Frequency');
ylabel('Power');
%%
N   = 100;         % FIR filter order
Fp  = 100;        % passband-edge frequency
Fs  = 96e3;        % 96 kHz sampling frequency
Rp  = 0.00057565; % Corresponds to 0.01 dB
                    peak-to-peak ripple
Rst = 1e-4;        % Corresponds to 80 dB
                    stopband attenuation

% eqnum = vec of coeffs
eqnum = firceqrip(N,Fp/(Fs/2),[Rp Rst],'passedge');
% Visualize filter
fvtool(eqnum,'Fs',Fs,'Color','White')
%%
figure;
hold on
output = filtfilt(eqnum,1,T_v);

subplot(211); plot(T_v,'b'); title('Original Signal')
subplot(212); plot(output,'r'); title('Filtered Signal')

figure;
output_fft = fft(output);
power2 = abs(output_fft).^2/n;    % power of the DFT

plot(power2);
xlabel('Frequency');
ylabel('Power');

%% ---- Split -------
L = length(T_v)/2;
T_11 = T_v(1:L);
T_12 = T_v((L):end);
```

```
L = length(T_v2)/2;
T_21 = T_v2(1:L);
T_22 = T_v2((L):end);

figure();
hold on
nbr_of_samples = 2400;
plot(T_21(1:nbr_of_samples), 'b')
plot(T_22(1:nbr_of_samples), 'r')
plot(T_11(1:nbr_of_samples), 'g')
plot(T_12(1:nbr_of_samples), 'm')

T_mean = zeros(nbr_of_samples:1);
for ii = 1:nbr_of_samples
    T_mean(ii) = (T_11(ii) + T_12(ii) +
                T_21(ii) + T_22(ii))/4;
end

T_mean = circshift(T_mean, 20);

figure(); plot(T_mean); title('Mean of movement')
%% -------- Compare ------------
figure;
hold on
output = filtfilt(eqnum,1,T_mean);

plot(output,'r'); title('Filtered Mean Signal')

T_down_1 = T_mean(1:(nbr_of_samples/2));
T_down_2 = output(1:(nbr_of_samples/2));
T_up_1 = T_mean((nbr_of_samples/2):nbr_of_samples);
T_up_2 = output((nbr_of_samples/2):nbr_of_samples);

figure()
subplot(411); plot(T_up_1,'b'); title('Up')
subplot(412); plot(T_up_2,'r'); title('Up-filtered')
subplot(413); plot(T_down_1,'b'); title('Down')
subplot(414); plot(T_down_2,'r'); title('Down-filtered')

%% ---- Match -------
```

```
len = nbr_of_samples/2;
e_u_d = E_v(1:len);
e_u_d2 = E_v(4200:(4200+len-1));

for ii = 1:(len-1)
   e_u_d(ii) = (e_u_d(ii) + e_u_d2(ii))/2;
end

e_u_d = filtfilt(eqnum,1,e_u_d);
figure();
plot(e_u_d);

matrix = zeros(2, len);
matrix(1, :) = e_u_d;
iter = 1;
for ii = 1:(len-1)
    matrix(1, ii) = round(matrix(1, ii));
    matrix(2, ii) = (output(iter) + output(iter+1)) /2;
    iter = iter + 2;
end
C = matrix;

figure(); %hold on
grid on;
yyaxis left;
plot(C(1,:));
ylabel('Encoder values');
yyaxis right;
plot(C(2,:));
ylabel('Torque values')

%% ---- Write to file -------
%D = C';
fileID = fopen('a60_mio.txt', 'w');
fprintf(fileID, '%6s :\n', 'down');
fprintf(fileID, '%6.2f : %12.8f \n', C(:, 1:600));
fprintf(fileID, '%6s :\n', 'up');
fprintf(fileID, '%6.2f : %12.8f \n', C(:, 601:end));
fclose(fileID);
```

Appendix B

Code: Door Simulator

B.1 Simulation runner

```
#!/usr/bin/env python3
'''
Created on 17 mars 2020

@author: annyds
'''
from help_functions import load_torque_files
from Servo import Servo
from Encoder import Encoder
from Simulator import Simulator

import time

#Setup of variables
door = 'a20'
operator = 'std'
nbr_of_cycles = 6
run_installtion = False
encoder_dict = {'portname':'COM6'}
servo_dict = {'portname':'COM5', 'door':door}

#Main
e = Encoder(encoder_dict)
s = Servo(servo_dict)

try:
    simulatot_dict = {'operator_type':operator,
                      'door_type': door,
                      'nbr_of_cycles':nbr_of_cycles}
    simulator = Simulator(simulatot_dict)
```

```
    if run_installtion:
        simulator.run_installation(e,s)
        input('Installation done, press any key to continue')
    else:
        simulator.clear_e24(e,s)
        print('---------------------------------------')
        input('E24 cleared, press any key to continue')
        print('---------------------------------------')
        s.set_torque(0)
    simulator.run_simulator(e,s)
except KeyboardInterrupt:
    simulator.stop_simulator()
    simulator.print_statistics()
    s.set_torque(0)
finally:
    e.stop()
    s.stop()
print('Simulation stoped/finished')
```

B.2 Installation runner

```
    #!/usr/bin/env python3
'''
Created on 29 June 2020

@author: annyds
'''
from help_functions import load_torque_files
from Servo import Servo
from Encoder import Encoder
from Simulator import Simulator

import time

#Setup of variables
door = 'a20'
operator = 'std'
encoder_dict = {'portname':'COM6'}
servo_dict = {'portname':'COM5', 'door':door}
```

```
nbr_of_cycles = 1

#Main
e = Encoder(encoder_dict)
s = Servo(servo_dict)

try:
    simulatot_dict = {'operator_type':operator,
                      'door_type': door,
                      'nbr_of_cycles':nbr_of_cycles}
    simulator = Simulator(simulatot_dict)
    simulator.run_installation(e,s)
except KeyboardInterrupt:
    simulator.stop_simulator()
    s.set_torque(0)
finally:
    e.stop()
    s.stop()
print('Door {} with motor type {} installed
        on DUT'.format(door, operator))
```

B.3 Simulator

```
    #!/usr/bin/env python3
'''
Created on 4 mars 2020

@author: annyds
'''
import sys
#import pydevd
import time
import logging

from help_functions import get_torque_val
from help_functions import load_torque_files

class Simulator(object):
    '''
```

```
classdocs
'''

def __init__(self, in_dict):
    '''
    Constructor
    '''
    self.operator_type = in_dict['operator_type']
                         if('operator_type' in in_dict)
                         else None
    self.door_type = in_dict['door_type']
                         if('door_type' in in_dict)
                         else None
    self.nbr_of_cycles = in_dict['nbr_of_cycles']
                         if('nbr_of_cycles' in in_dict)
                         else 0
    self.torque_file = load_torque_files(self.operator_type,
                                         self.door_type)
    self.stoped = False
    self.door_opened = False
    self.door_closed = False
    self.nbr_of_iter = 0

def run_simulator(self, encoder, servo):
    '''
    Run the simulator for set operator- and door-type

    args:
        *encoder: the encoder to fetch values from.
                  Controlls the torque value
        *servo: the servo on which to set the torque value
    '''
    minimum_up = min(self.torque_file['up'])
    minimum_down = min(self.torque_file['down'])
    bottom_val = min(minimum_up, minimum_down)
    print('bottom val = {}'.format(bottom_val))

    print('-----------------------------')
    print('Run the door down and up once')
    input('Press any key to continue script')
```

```
print('-----------------------------')

self.stoped = False
self.nbr_of_iter = 0
finished_iterations = False
encoder.set_start_pos()
old_ev = 0
been_at_bottom = False


while not self.stoped and not finished_iterations:
    ev = encoder.get_encoder_val()
    direction = encoder.get_direction()
    print('Moving {}'.format(direction))
    print('encoder value: {}'.format(ev))
    #debugprint
    if ev != old_ev and direction != 'still':
        tv = get_torque_val(self.torque_file,
                            direction, ev)
        #print('torque value: {}'.format(tv))
        #debugprint
        if been_at_bottom and ev<(bottom_val+20000)
           and direction == 'up':
            #TODO should be dynamically set
            #dependent on the door and motor type
            servo.set_torque(200)
        else:
            servo.set_torque(tv)
    if abs(ev)<=300 and direction=='up'
       and been_at_bottom:
        self.nbr_of_iter += 1
        print('-------------------------------')
        print('Number of cycles {}'.format(
                                self.nbr_of_iter))
        print('-------------------------------')
        been_at_bottom = False
    if self.nbr_of_iter == self.nbr_of_cycles:
        finished_iterations = True
    if ev <= (bottom_val + 10000)
    and direction == 'still' and not been_at_bottom:
```

```
                been_at_bottom = True
                print('-------------------------------')
                print('Bottom reached')
                print('-------------------------------')
            #TODO make the time between samples
            #a variable that can be set by the user
            time.sleep(0.2)
        #end while

        self.print_statistics()

    def clear_e24(self, encoder, servo):
        '''
        Clear E24 on DUT,
        please follow instructions printed in terminal

        args:
            *encoder: the encoder to fetch values from.
                      Controls the torque value
            *servo: the servo on which to set
                    the torque value
        returns:
            None
        '''
        self.stoped = False
        minimum_up = min(self.torque_file['up'])
        minimum_down = min(self.torque_file['down'])
        minimum = min(minimum_up, minimum_down)
        start_ev = 0.5 * minimum
        encoder.set_start_pos()
        servo.set_torque(0)
        ev = 1

        print('-----------------------------')
        print('Press UP button to clear E24')
        input('Press any key to continue script')
        print('-----------------------------')

        while not self.stoped and ev != 0:
            ev = abs(encoder.get_encoder_val()) + start_ev
```

```
            direction = encoder.get_direction()
            print('Current position: {}'.format(ev))
            if abs(ev) <= 5000:
                servo.set_torque(1600.00) #mV
            if direction == 'still':
                self.stop_simulator()
            #TODO make the time between samples a
            #variable that can be set by the user
            time.sleep(0.1)

    def run_installation(self, encoder, servo):
        '''
        Install wanted door on DUT,
        please follow instructions printed in terminal

        args:
            *encoder: the encoder to fetch values from.
                      Controls the torque value
            *servo: the servo on which to set
                    the torque value
        returns:
            None
        '''
        minimum_up = min(self.torque_file['up'])
        minimum_down = min(self.torque_file['down'])
        minimum = min(minimum_up, minimum_down)
        percentage_1m = 80
        passed_1m = False
        percentage_bend = 70
        passed_bend = False

        self.clear_e24(encoder, servo)
        print('-----------------------------')
        print('Activate SW1 to start installation')
        input('Press any key to continue')
        print('-----------------------------')

        self.stoped = False
        encoder.set_start_pos()
        servo.set_torque(0)
```

```
ev = 1

print('-----------------------------')
print('Press and hold Up-button')
input('Press any key to continue')
print('-----------------------------')
while not self.stoped and ev != 0:
    ev = encoder.get_encoder_val()
    direction = encoder.get_direction()
    print('moving {}, position: {}'.format(direction,
                                           ev))
    if ev <= (percentage_1m/100)*minimum
       and not passed_1m:
        print('-----------------------------')
        print('Release Up-button')
        print('Press SE')
        print('Press Down-button')
        print('-----------------------------')
        input('Press any key to continue')
        passed_1m = True
    if direction == 'up'
    and ev >= (percentage_bend/100)*minimum
    and not passed_bend and passed_1m:
        print('-----------------------------')
        print('Press STOP')
        print('Press Up-button')
        print('-----------------------------')
        passed_bend = True
        input('Press any key to continue')
    if direction == 'down' and ev <= (minimum+100):
        servo.set_torque(1600.00)
        print('-----------------------------')
        print('Press SE')
        print('Bottom position reached')
        print('-----------------------------')
        time.sleep(0.2)
        servo.set_torque(0)
        input('Press any key to continue')
    if direction == 'up' and abs(ev) <= 20000
    and passed_bend:
```

```
                if abs(ev) <= 400:
                    servo.set_torque(1600.00)
                    print('----------------------------')
                    print('Top position reached')
                    print('----------------------------')
                    input('Press any key to continue')
                    time.sleep(2)
                    servo.set_torque(0)
                elif abs(ev) <= 1000:
                    servo.set_torque(600.00)
                elif abs(ev) <= 5000:
                    servo.set_torque(300.00)
                elif abs(ev) <= 10000:
                    servo.set_torque(250.00)
                else:
                    servo.set_torque(200.00)
            if direction == 'still' and abs(ev) <= 100
            and passed_1m:
                self.stop_simulator()
            #TODO make the time between samples a
            #variable that can be set by the user
            time.sleep(0.02)
        time.sleep(3)
        servo.set_torque(0)
        print('----------------------------')
        print('Deactivate SW1')
        print('----------------------------')

    def stop_simulator(self):
        '''
        Stop the running of the simulator

        args:
            None
        returns:
            None
        '''
        self.stoped = True
        return 1
```

```
    def print_statistics(self):
        '''
        Print statistics from the simulation

        args:
            None
        returns:
            None
        '''
        msg1 = ' Goal: {} cycles \n Ran: {} cycles
               \n Stoped:{}'.format(self.nbr_of_cycles,
                          self.nbr_of_iter, self.stoped)
        print('-----------------------------')
        print('-----------------------------')
        print(msg1)
        print('-----------------------------')
        print('-----------------------------')
        return None
```

B.4 Encoder

```
    #!/usr/bin/env python3
'''
Created on 4 mars 2020

@author: annyds
'''
import sys
import time
import logging
from Communication import Host

class Encoder(object):
    '''
    classdocs
    '''

    def __init__(self, in_dict):
        '''
```

```
    Constructor
    '''
    self.last_ev = 0
    self.new_ev = 0
    self.start_val = 0
    self.portname = in_dict['portname']
                    if('portname' in in_dict) else None
    self.host = Host(self.portname, 'rs485')
    self.host.start()
    self.clear()
    self.start_val = self.set_start_pos()

def get_encoder_val(self):
    '''
    Get the current encoder value

    args:
        None

    returns:
        *encoder_val: (double) the
                    current encoder value
    '''
    message = b'\xA0' #hex byte
    self.host.write(message)
    time.sleep(0.05)
    out = self.host.read()
    self.clear()
    e_string = ''
    if len(out) == 5:
        encoder_val  = int.from_bytes(map(ord,out[:4]),
                       byteorder='little',  signed=True)
        encoder_val = -1*abs(encoder_val-self.start_val)
        if abs(encoder_val - self.new_ev) < 100000:
            self.last_ev = self.new_ev
            self.new_ev = encoder_val
        else:
            if (self.new_ev < self.last_ev):
                a = -10
            else:
```

```
                a = 10
            self.last_ev = self.new_ev
            encoder_val = -1*abs(self.new_ev + a)
            self.new_ev = encoder_val

    else:
        encoder_val = 1
        print('you are here')
    return encoder_val

def get_direction(self):
    '''
    Check the direction of movement

    args:
        None

    returns:
        *direction: (string) up/down
    '''
    new_ev = self.new_ev
    if new_ev > self.last_ev:
        direction = 'up'
    elif new_ev < self.last_ev:
        direction = 'down'
    elif new_ev == self.last_ev:
        direction = 'still'
    return direction

def set_start_pos(self):
    '''
    Check the direction of movement

    args:
        None

    returns:
        *start_pos: (double) start value
    '''
    message = b'\xA0' #hex byte
```

```
        self.host.write(message)
        time.sleep(0.1)
        out = self.host.read()
        self.clear()
        e_string = ''
        if len(out) == 5:
            encoder_val  = int.from_bytes(map(ord,out[:4]),
                           byteorder='little', signed=True)
            self.start_val = encoder_val
            self.new_ev = 0

        return self.start_val

    def stop(self):
        '''
        Stop the running of the encoder

        args:
            None

        returns:
            None
        '''
        self.host.stop()
        return 0

    def clear(self):
       '''
        Clear the input and output buffer of the device

        args:
            None

        returns:
            None
        '''
       self.host.reset()
       return 0
```

B.5 Servo

```
    #!/usr/bin/env python3
'''
Created on 4 mars 2020

@author: annyds
'''
import sys
import time
import logging
from Communication import Host

class Servo(object):
    '''
    classdocs
    '''

    def __init__(self, in_dict):
        '''
        Constructor
        '''
        self.portname = in_dict['portname']
                        if('portname' in in_dict) else None
        self.door = in_dict['door']
                    if('door' in in_dict) else 'a20'
        self.door = self.door.lower()
        self.host = Host(self.portname, 'rs232')
        self.host.start()
        self.reset()

# ------------ Basic functionality -------------
    def reset(self):
        '''
        Reset the Servo and clear the alarm history
        args:
            None
        reurns:
            None
        '''
```

```
        self.host.reset()

    def stop(self):
        '''
        Stop the running of the servo
        args:
            None
        returns:
            None
        '''
        self.host.stop()
        return 0

    def write_msg(self, msg):
        '''
        Write a message to the servo
        args:
            msg: (str) command string to write to Servo
        returns:
            None
        '''
        #print(msg) #debugprint
        self.host.write(msg)
        return 0

    def read_msg(self):
        '''
        Read a message from the servo
        args:
            None
        returns:
            result: result from reading from  the servo
        '''
        result = self.host.read()
        return result

# ------------- Set-functions -------------
    def set_i_devices(self, in_dict):
        '''
        Enable/disable input devices
```

```
    (in test operation mode)
    args:
        in_dict: (list) list with bit:1/0 => {12:1}
                  to enable bit 12
    returns:
        None
    '''
    data = ['30', '30', '30', '30', '30', '30', '30', '30',
            '30', '30', '30', '30', '30', '30', '30', '30',
            '30', '30', '30', '30', '30', '30', '30', '30',
            '30', '30', '30', '30', '30', '30', '30', '30']
    for x, y in in_dict.items():
        data[x] = hex(y).replace('0x', '')

    data.reverse()
    cmd_dict = {'cmd':[39, 32],
                'data_nbr':[30, 30],
                'data':data}
    cmd = self.encode_cmd(cmd_dict)
    self.write_msg(cmd)
    time.sleep(0.1)
    return self.check_error(self.read_msg())


def set_operation_mode(self, mode='T'):
    '''
    Set the operation mode for the servo
    args:
        mode: (string) mode of operation,
              DEFAULT = T (torque)
    returns:
        None
    '''
    raise Exception('Placeholder, function not implemented')
    mode_dict = {'T':'0004', 'P':'0000',
                 'PS':'0001', 'S':'0002',
                 'ST':'0003', 'TP':'0005'}
    if (mode.upper() in mode_dict):
        mode_str = mode_dict[mode.upper()]
    else:
```

```
            raise Exception('Invalid choice of
                            operation mode: {}'.format(mode))

        self.write_msg(mode_str)
        return 0

    def set_torque(self, torque=100):
        '''
        Set the torque value
        args:
            torque: (double) the torque value
        returns:
            None
        '''
        torque = self.transform_torque(torque)

        data = ['33', '30', '30', '30', '30', '30', '30', '30']
        torque = hex(torque)
        torque = torque.replace('0x', '')

        ii = 0
        for letter in torque:
            tv = hex(ord(torque[ii].upper()))
            tv = tv.replace('0x','')
            data[len(data) - len(torque) + ii] = tv
            ii+=1
        param = 28
        return self.write_param(param, data)

# -------------- Read-functions --------------
    def read_error(self):
        '''
        Read the current alarm/error
        args:
            None
        returns:
            alarm: (str) the current error on the servo
        '''
        in_dict = {'cmd':[30, 32], 'data_nbr':[30, 30]}
        cmd = self.encode_cmd(in_dict)
```

```
        self.write_msg(cmd)
        time.sleep(0.1)
        error = self.read_msg()
        self.check_error(error)
        error_code = error[2].decode(encoding='UTF-8')
        if error_code == 'a':
            alarm = error[5].decode(encoding='UTF-8') +
                    error[6].decode(encoding='UTF-8')
        else:
            alarm = 'No alarm'
        return alarm

# ------------- Message handling ---------------------
    def write_param(self, param, data):
        '''
        Write data to the parameter param
        args:
            param: (int) the parameter
            data: (list of str) the data,
                   needs to be written correctly
        returns:
            0 if successful
        '''
        data_nbr = []
        param = hex(param)
        param = param.replace('0x', '')
        ii = 0

        if len(param) == 1:
            param = '0' + param
        for letter in param:
            p_hex = hex(ord(param[ii].upper()))
            p_hex = p_hex.replace('0x','')
            data_nbr.append(p_hex)
            ii +=1
        in_dict = {'cmd':[38, 34],
                   'data_nbr':data_nbr,
                   'data':data}
        cmd = self.encode_cmd(in_dict)
        self.write_msg(cmd)
```

```
        time.sleep(0.1)
        return self.check_error(self.read_msg())

    def read_param(self, param):
        '''
        Read the param value
        args:
            param: (int) param number to read
        return:
            value: (int) param value
        '''
        data_nbr = []
        param = hex(param)
        param = param.replace('0x', '')
        ii = 0

        if len(param) == 1:
            param = '0' + param
        for letter in param:
            p_hex = hex(ord(param[ii].upper()))
            p_hex = p_hex.replace('0x','')
            data_nbr.append(p_hex)
            ii +=1
        in_dict = {'cmd':[30, 35], 'data_nbr':data_nbr}
        cmd = self.encode_cmd(in_dict)
        self.write_msg(cmd)
        time.sleep(0.1)
        response = self.read_msg()
        self.check_error(response)
        return response

    def encode_cmd(self, in_dict):
        '''
        Encode a command
        args:
            in_dict: (dict) should contain cmd,
                       data_nbr, data
        returns:
            msg: () encoded command
        '''
```

```
        cmd = in_dict['cmd']
              if('cmd' in in_dict) else [30, 32]
        data_nbr = in_dict['data_nbr']
                   if('data_nbr' in in_dict) else [30, 30]
        data = in_dict['data']
               if('data' in in_dict) else None

        basecmd = ['01', '30', '00',
                   '00', '02', '00',
                   '00', '03']
        cmd_list = ['01', '30', '00',
                    '00', '02', '00',
                    '00']
        cmd_string = ''

        for ii in range(len(basecmd)):
            if ii in [2, 3]:
                cmd_string = cmd_string +
                             str(cmd[ii - 2]) + ' '
                cmd_list[ii] = str(cmd[ii - 2])
            elif ii in [5, 6]:
                cmd_string = cmd_string +
                             str(data_nbr[ii - 5]) + ' '
                cmd_list[ii] = str(data_nbr[ii - 5])
                if ii == 6 and data != None:
                    for iter in range(len(data)):
                        cmd_string = cmd_string +
                                     data[iter] + ' '
                        cmd_list.append(data[iter])
            else:
                cmd_string = cmd_string + basecmd[ii] + ' '
        cmd_list.append('03')

        checksum = self.calc_checksum(cmd_list)
        cmd_string = cmd_string + checksum

        msg = bytearray.fromhex(cmd_string)
        return msg

    def calc_checksum(self, msg):
```

```
        '''
        Calculates checksum for msg
        '''
        sum = 0
        for ii in range(len(msg)):
            if ii == 0:
                continue
            else:
                sum += int(('0x' + msg[ii]), 16)

        sum = hex(sum)
        sum = sum.replace('0x','')
        if len(sum) > 2:
            part1 = hex(ord(sum[len(sum)-2].upper()))
            part2 = hex(ord(sum[len(sum)-1].upper()))
        else:
            part1 = hex(ord(sum[0].upper()))
            part2 = hex(ord(sum[1].upper()))
        checksum = part1.replace('0x','') + ' ' +
                   part2.replace('0x','')
        return checksum

    def check_error(self, msg_str):
        '''
        Check if the msg_str reports any error,
        used for checking if a message was
        transmitted properly
        args:
            msg_str: (str) the string to be checked
        returns:
            None
        raises:
            Exception if error occured
        '''
        msg = msg_str[2].decode(encoding='UTF-8')
        error = None
        if msg.lower() == 'a':
            pass
        elif msg.lower() == 'b':
            error = 'Parity error'
```

```
        elif msg.lower() == 'c':
            error = 'Checksum error'
        elif msg.lower() == 'd':
            error = 'Character error'
        elif msg.lower() == 'e':
            error = 'Command error'
        elif msg.lower() == 'f':
            error = 'Data No. error'

        if error:
            if msg.islower():
                raise Exception('{} caused an
                                 alarm on the
                                 Servo'.format(error))
            else:
                raise Exception('{}, no alarm
                                  on Servo'.format(error))
        else:
            return 0

    def transform_torque(self, torque):
        '''
        Transform a torque value from mV to percent
        '''
        a = 6.434
        b = -6.7997
        if self.door == 'a10':
            c = 2.7941
        elif self.door == 'a20':
            c = 5.4781
        elif self.door == 'a60':
            c  = 4.1455
        elif self.door == 'olle':
            raise Exception('Placeholder olle
                             door is not implemented')
        else:
            raise Exception('Selected door {} is an
                             unknown type'.format(self.door))
        t = round(1*(((torque * (1/c)) + b) / a))
        t = abs(t)
```

```
        #print('Percentage: {}'.format(t)) #debugprint
        return t
```

B.6 Communication

```
    #!/usr/bin/env python3
'''
Created on 4 mars 2020

@author: annyds
'''
from threading import Thread
import serial, time, binascii
import serial.rs485

class Host(object):
    state_dict = {1:"Idling ", 2:"Accepting ",
                  4:"Escrowed ", 8:"Stacking ",
                  16:"Stacked ", 32:"Returning",
                  64:"Returned", 17:"Stacked Idling ",
                  65:"Returned Idling "}
    event_dict = {0:"", 1:"Cheated ", 2:"Rejected ",
                  4:"Jammed ", 8:"Full "}

    def __init__(self, portname, com_type):
        self.portname = portname
        self.com_type = com_type.lower()
        # Set to False to kill
        self.running = True
        self.bill_count = bytearray([0, 0, 0, 0,
                                     0, 0, 0, 0])
        # Background worker thread

        self._serial_thread = None
        self.ser = None

    def start(self):
        """
        Start Host in a non-daemon thread
```

```
        Args:
            None
        Returns:
            None
        """
        self._serial_thread = Thread(args=(self.portname,))
        self._serial_thread.daemon = False
        self._serial_thread.start()
        if self.com_type == 'rs232':
            self.create_connection_RS232()
        elif self.com_type == 'rs485':
            self.create_connection_RS485()
        else:
           raise ValueError('Communication type not set \
                            in {}'.format(__class__.__name__))
        #self.ser.open()

    def create_connection_RS232(self):
        """

        Args:
            None
        Returns:
            None
        """
        self.ser = serial.Serial(
            port=self.portname,
            baudrate=9600,
            bytesize=serial.EIGHTBITS,
            parity=serial.PARITY_EVEN,
            stopbits=serial.STOPBITS_ONE
        )

    def create_connection_RS485(self):
        """

        Args:
            None
        Returns:
            None
```

```
        """
        self.ser = serial.Serial(
            port=self.portname,
            baudrate=19200,
            bytesize=serial.EIGHTBITS,
            parity=serial.PARITY_NONE,
            stopbits=serial.STOPBITS_ONE
            #termination char = xA
        )

    def write(self, msg):
        """

        Args:
            *msg: (string)
        Returns:
            None
        """
        if self.ser.isOpen() and self.running:
            try:
                result = self.ser.write(msg)
                return result
            except serial.SerialException as e:
                if e.args == (5,"WriteFile",
                              "Access is denied."):
                    # This occurs on win32 when a USB
                    #serial port is unplugged and
                    # replugged.  It should be fixed by
                    # closing and reopening the port,
                    #which should happen in the
                    # error handling of our caller.
                    raise IOError(errno.ENOENT,
                                  "Serial port \
                                  disappeared.",
                                   self.ser.portstr)
                else:
                    raise Exception('Error {} \
                                    occured in \
                                    {}'.format(e,
                                       self.__class__))
```

```
def read(self):
    """

    Args:
        None
    Returns:
        None
    """
    if self.ser.isOpen() and self.running:
        out = []
        while self.ser.inWaiting() > 0:
            val = self.ser.read(1)
            out.append(val)
        return out
    elif  not self.ser.isOpen():
        raise Exception('Serial communication is closed')
    else:
        print('Communication is not running')
        return None

def stop(self):
    """
    Blocks until Host can safely be stopped
    Args:
        None
    Returns:
        None
    """
    self.running = False
    self._serial_thread.join()
    self.reset()
    self.ser.close()

def reset(self):
    """
    Reset the input and output buffer
    Args:
        None
    Returns:
```

```
            None
        """
        self.ser.flushInput()
        self.ser.flushOutput()
        return 1
```

B.7 Help functions

```
    #!/usr/bin/env python3
'''
Created on 4 mars 2020

@author: annyds
'''
import yaml
#import pydevd
import os
import time

def load_torque_files(operator_type, door_type):
    '''
    Loads the torque file specified by
    the operator type and the door type

    args:
        *operator type: (string) Type of operator
        *door_type: (string) Type of door

    returns:
        *torque_file: (list) A list consisting of keys:
                      encoder values and
                      values: torque value
    '''
    path = 'Program/torque_files/'
    file_name = door_type.lower() + '_' +
                operator_type.lower() + '.yml'
    file_path = os.path.join(path, file_name)
    print('--------------------------------')
    print('Loading torque file: \n {}'.format(file_path))
```

```
    print('--------------------------------')
    with (open(file_path, 'r')) as torque_stream:
        loading = yaml.load(torque_stream,
                            Loader=yaml.FullLoader)
        torque_file = {**loading}
    print('Torque file successfully loaded')
    print('--------------------------------')
    return torque_file

def get_torque_val(torque_file, direction, encoder_val):
    '''
    Loads the torque file specified by the
    operator type and the door type

    args:
        *torque_file: (list)
        *direction: (string)
        *encoder_val: (double)

    returns:
        *torque_value: (double) A The torque value to
                        the specified encoder value
    '''
    try:
        val = torque_file[direction][encoder_val]
    except Exception as e:
        torque_list = torque_file[direction]
        closest_key = min(torque_list,
                      key=lambda x:abs(x-encoder_val))
        val = torque_file[direction][closest_key]
    return val
```