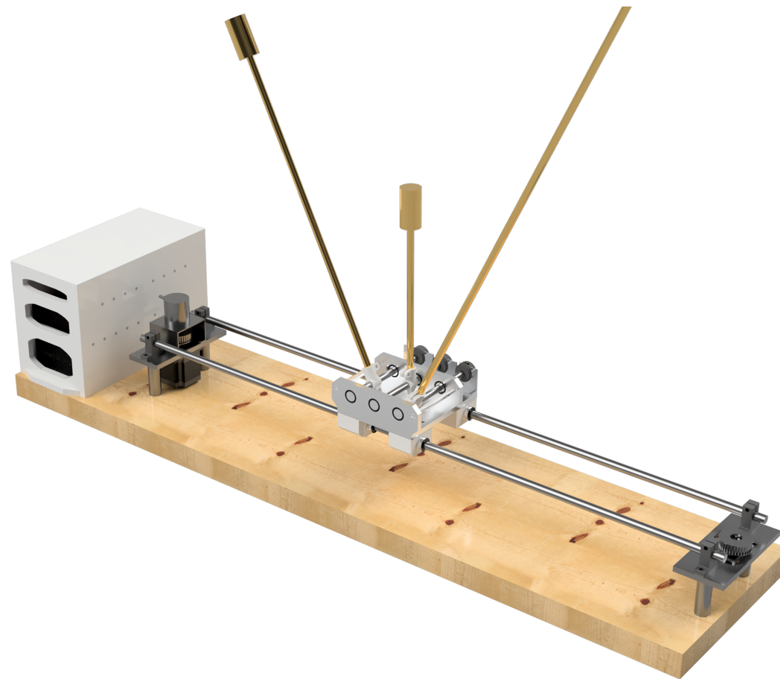
CHALMERS
UNIVERSITY OF TECHNOLOGY



A Demo Equipment for Balancing Multiple Pendulums on a Single Cart

Bachelor Thesis EENX15-18-05

NILS ASSARSSON
CARL-JOHAN HEIKER
DAVID HEIMAN
MATTIAS JUHLIN
JENS REHN
ELIAS RHODÉN

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Bachelor Thesis EENX15-18-05

A Demo Equipment for Balancing Multiple Pendulums on a Single Cart

NILS ASSARSSON
CARL-JOHAN HEIKER
DAVID HEIMAN
MATTIAS JUHLIN
JENS REHN
ELIAS RHODÉN

Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2018

A Demo Equipment for Balancing Multiple Pendulums on a Single Cart
NILS ASSARSSON
CARL-JOHAN HEIKER
DAVID HEIMAN
MATTIAS JUHLIN
JENS REHN
ELIAS RHODÉN

© NILS ASSARSSON, CARL-JOHAN HEIKER, DAVID HEIMAN, MATTIAS JUHLIN, JENS REHN, ELIAS RHODÉN, 2018.

Supervisor: Torsten Wik, Department of Electrical Engineering, Systems and Control
Examiner: Sebastien Gros, Department of Electrical Engineering

Bachelor Thesis EENX15-18-05
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: CAD-rendering of the system

Typeset in LaTeX
Gothenburg, Sweden 2018

Abstract

For students in basic control theory, the concept of inverted pendulums could potentially be more easily understood if properly demonstrated by a physical system. By studying the whole system, the student will be able to observe common weaknesses and characteristics of mechatronic systems and components which can make the difference from all theoretical concepts more obvious. For the more experienced control engineer, such a machine may serve as an experimental setup for testing different regulator concepts and pendulum configurations.

Here we report on the development of such a setup. Based on a mathematical model of up to three pendulums on one single cart driven by a motor, the design of the physical system is divided into selecting electronic components, creating a mechanical rail-and-cart system, programming and electronics design as well as developing a control system that balances up to three pendulums at the same time.

Although the case of three pendulums proved to be too hard to achieve with the constructed equipment in practice, several methods of systems control were investigated and simulated. In the end, the system was able to balance one pendulum. Overall well constructed, the machine lives up to the expectations of serving as a demonstrational equipment and experimental testbed for students and engineers. To balance more than one pendulum, for example, verifying the stepper motor model, improving the general model of the system or adjusting the construction is recommended.

Keywords: Inverted pendulums, Systems control, Mechatronics, Unstable systems, Balance, LQR

Acknowledgements

We especially want to thank our project supervisor Torsten Wik for insightful advice and discussion regarding the general advancement of the project. We also want to thank Jan Bragée and Reine Nohlborg at the Chalmers prototype lab for their valuable advice and competence regarding construction issues. A special thank you to Patrik Hansson of Expovision for the time that he put in making the CNC face milling possible.

Nils Assarsson, Carl-Johan Heiker, David Heiman,
Mattias Juhlin, Jens Rehn, Elias Rhodén
Gothenburg, May 2018

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Aim 2
1.2 Specification of issue under investigation 2
1.3 Limitations 3
1.4 Outline of report 4

2 Modeling of System **5**
2.1 One pendulum 5
2.2 Friction 8
2.3 Three pendulums 10
2.4 Stepper motor characteristics 13

3 Component Selection **15**
3.1 Microcontroller 16
3.2 Angular sensor 16
3.3 Rotary encoder 18
3.4 Motor 19
3.5 Stepper motor driver 20
3.6 Power supply 20
3.7 Other components 21

4 Simulation of System **23**
4.1 Stepper motor 23
4.1.1 Motor constant 23
4.1.2 Number of rotor pole pairs 24
4.1.3 Damping constant 25
4.1.4 Simulation of stepper driver 26
4.1.5 Simulation of microstepping 27
4.2 Parameters of cart and pendulums 28
4.2.1 Determination of average operation speed 28
4.2.2 Friction constants 29
4.2.3 Masses and lengths 29
4.3 Complete list of parameters for simulation 30

4.4 Simulation of unregulated model . . . 30
4.4.1 One pendulum on a cart . . . 31
4.4.2 Three pendulums on a cart . . . 33
4.5 Limitations due to stepper motor . . . 34

5 Construction of System **35**
5.1 Previous work . . . 35
5.2 Design of mechanical construction . . . 35
5.2.1 Construction . . . 36
5.2.2 Results & Discussion . . . 37

6 Electrical Construction & Programming **39**
6.1 Hardware design . . . 39
6.1.1 Stepper motor . . . 40
6.1.2 Stepper motor driver . . . 41
6.1.3 Rotary encoder . . . 42
6.2 Software design . . . 42
6.2.1 The design of the software . . . 44
6.2.2 Motor and stepper driver software interface . . . 45
6.2.3 Software representation of rotary encoder . . . 46
6.2.4 Angular sensor software . . . 47
6.3 Results & Discussion . . . 49

7 Control of System **51**
7.1 Simplification of system . . . 51
7.1.1 Stepper model . . . 51
7.1.2 Pendulum . . . 51
7.2 Classic control . . . 53
7.2.1 One pendulum . . . 53
7.2.2 Two pendulums . . . 54
7.3 State feedback control . . . 56
7.3.1 State-space representation of the system . . . 56
7.3.2 Linear-quadratic regulator . . . 57
7.3.3 One pendulum . . . 58
7.3.4 Three pendulums with positioning . . . 60

8 Results & Discussion **63**
8.1 Physical system . . . 63
8.2 Balancing one pendulum using LQR . . . 64
8.3 Model validation . . . 65
8.4 Future work . . . 70
8.4.1 Modeling and simulation . . . 70
8.4.2 Electrical construction . . . 70
8.4.3 Control . . . 71

9 Conclusion **73**

Bibliography **75**

A Drawings **I**

B Code **V**

C Other **XXVII**

List of Figures

1 A single pendulum on a cart. 5
2 Forces acting on pendulum. 6
3 Forces acting on cart. 7
4 Different accelerations of the cart depending on the pendulum. 9
5 Pendulum on a cart. 10
6 Three pendulums on the same cart. 12
7 Rotor and stator principle of a stepper motor. 13

8 CAD-model. 15
9 Movement interpretations of different sensors. 18
10 Speed Torque curve of stepper from its datasheet [1]. 19

11 Speed response of stepper. 25
12 Relation between step pulse input and the phase voltages. 26
13 Angle response of the implemented driver. 26
14 Voltages of microstepping. 27
15 Angle response of microstepped driver. 28
16 Input signal to simulations. 30
17 Cart position. 31
18 Cart velocity. 31
19 Pendulum angle. 32
20 Cart position. 33
21 Cart velocity. 34
22 Pendulum angles. 34

23 CAD-model. 36
24 Mechanical construction. 37

25 Hardware Principle Map. 40
26 Stepper Motor. 40
27 Stepper Driver. 41
28 Rotary encoder circuit. 42
29 Software principle map. 43
30 Photo of the panel. 44
31 Flow Chart describing the function of the software. 45
32 Step visualisation. 46
33 Rotary encoder phase pulses. 47

34 Electrical cabinet with wiring principle. 49
35 FFT-analysis of the signal with noise from an angular sensor. 50

36 Stepper described as a constant gain. 51
37 Control structure for one pendulum. 53
38 Control of a single pendulum using PI-controller. 54
39 Control structure for two pendulums. 54
40 Simplified control structure for two pendulums. 54
41 Simulation with numerically determined parameters for the second loop. 55
42 θ, one pendulum. 58
43 Motor speed, one pendulum. 59
44 Motor speed, three pendulums and positioning control activated. . . . 60
45 Pendulum angles, three pendulums and positioning control activated. 61
46 Cart position, three pendulums and positioning control activated. . . 61

47 Final physical system. 63
48 Control design for one pendulum. 64
49 Input signal used for verification. 65
50 Angle of pendulum in simulation. 66
51 Angle of pendulum in real system. 66
52 Position of cart in simulation. 67
53 Position of cart in real system. 67
54 Speed of cart in simulation. 68
55 Speed of cart in real system. 68

56 Drawing of the complete system. I
57 Wiring diagram of the complete electrical system. II
58 Wiring diagram of the circuits under the electrical cabinet. III

List of Tables

1 List of components. 15
2 Specifications of Arduino DUE. 16
3 Specifications of P2501A202, {P2501A502}. 17
4 Specifications of the rotary encoder. 18
5 Specifications of stepper motor. 19
6 Specifications of stepper motor driver. 20
7 Specifications of power supply of stepper motor driver. 20

8 Parameters from the datasheet of stepper motor [1]. 23
9 Complete parameter list for simulation. 30
10 Extended parameters for three pendulums. 33

11 Construction specifications. 36

12 Parameter values, one pendulum. 58
13 Parameter values in simulation of three pendulums and positioning control activated. 60

14 Parameter list for simulation verification. 65

15 Solution-selection matrix. XXVIII

1 Introduction

A controller is basically a device with the function of maintaining or steering a certain state of a system. Controllers have been around a long time and through the ages in different forms. One famous controller dates to 300 B.C. and is called Ctesibios' water clock. The mechanical controller held a certain water level in a tank which created a constant pressure at the bottom of the tank. Due to the constant pressure, water could flow through a hole at the bottom at a constant rate. This outlet of water then filled a cup where the water level could be measured and represent passed time. Other mechanical controllers have been invented in later years such as James Watt's controller for the angular velocity of his steam engine in 1788 [2].

During the 1920s and 1930s, the area of control systems had a flourishing era with a rapid development. Initially, the main user of controllers was electric power industries but later the technology became a significant part in other applications. They started to play an important role in stabilizing ships in rough seas and made flying aircrafts for pilots easier. At the same time, controllers started to expand their area of usage to also stabilizing unstable systems. This also made analyzing unstable systems something of interest for some well-known researchers such as Harry Nyqvist and Hendrik W. Bode, both having contributed with tools still used today [3].

The control of unstable systems can be found in many of today's applications, such as the aircraft Saab 39 Gripen [4], rockets [5] and, what is of interest in this project, inverted pendulums [2]. Inverted pendulums come in various forms and methods of stabilization such as on rail [6], wheels [7], and shaft [8]to name a few. The fascination of inverted pendulums has been commented by Ernesto Aranda-Escolástico i.a., who gave the reason that it is a *"difficult and complex system to be controlled that has been used as a benchmark to compare different control strategies"*[9]. In this project the control strategy is in focus but also the construction of an inverted pendulum system.

This project, however, partly aims to investigate the situation in where three separate pendulums are mounted on one cart and are thereby controlled by one single motor while still behaving as three separate bodies. There does not seem to be any research done on this particular problem, which makes this project relevant to investigate.

1.1 Aim

The purpose of this project was to construct a system with multiple inverted pendulums mounted on a movable cart and make it so that the pendulums would automatically stand perpendicular to the base of the system.

Firstly, as the project embraces several equally important parts (mainly control theory, programming, construction, design, and testing), a motivation for conducting the project was to get the opportunity to investigate as many of these subjects as possible while also merging them together to form a final product.

Secondly, the system to be developed should be looked upon as a type of demonstration equipment, intended to be used in teaching control theory, or as a test bed for investigating ideas concerning the control of inverted pendulums.

Lastly, while maintaining a solid and fairly under-the-hood design, the system should offer the user a certain amount of freedom when it comes to what types of experiments the user is able to conduct on the system. A modular approach to attach multiple pendulums of different lengths and sizes, along with assigning and modifying various regulators contributed to fulfilling the intentions of the project.

1.2 Specification of issue under investigation

As described in Section 1.1, several purposes set the bar for this project. Some of them can directly be translated into goals that we had to satisfy to consider ourselves finished.

Developing the mechanics to work as intended was the first goal. The system should be able to move a platform back and forth along a linear path to be able to balance one or several pendulums. This goal includes both hardware and software, which brings us to the next objective, the user interface.

The user interface should offer the user as much freedom as possible within the range of fundamental system (pendulum) control. The option to activate several different regulators (P, PI, PD and PID) along with the possibility of tweaking each regulator parameter should be available for the user, preferably without direct access to the code on the microprocessor.

Regarding the theoretical part, namely the control theory problems that are to be investigated, the goal was to be able to control at least one pendulum with the system. While control over multiple (one or two) additional pendulums is interesting, this was not considered a basic objective, but an advanced one.

Part of the investigation is about studying and investigating what methods of control work on our system, as well as shining light on some theories concerning control

of multiple pendulums.

A problem that arises mainly in the control, multiple pendulums, is robustness. How much, for example, is the angular difference in the starting position allowed to differ between two pendulums while still maintaining balance in the whole system. Robustness can of course also be evaluated on a system with only one pendulum, for example, how much difference can we allow between our model and the real system.

These issues should be possible to address with the desired system; one that can serve as a robust, valuable and versatile demo equipment for anyone interested in practical pendulum control theory.

1.3 Limitations

With 20 weeks and 5000 SEK, time and budget for this project were limited. This makes focusing on the predefined goals crucial. As a result, simplifications had to be made to the design as well as to the research and construction process to ensure the targets would be met.

The system model was made using some simplifying assumptions. The result from simulating the model is presented in graphs rather than a live virtual simulation. Furthermore, strength, solidity or other material properties have not been examined. Regarding CAD drawings, the two main purposes were measurements and design; textures and animations were of little importance here.

The material selection for the final product was based on workability, price and ease of access rather than trying to find the optimal materials by doing an in-depth research. Nor was materials chosen with appearance in mind.

As the target is to keep one or more pendulums stable, the software and design assumes that the initial state of the pendulums is standing up vertically, to make the problem easier.

1.4 Outline of report

The whole report outline will reflect the process of constructing the system. The chapters following introduction represents the different subsystems that the whole system was divided into. Within each of these chapters the process of constructing the subsystem will be described and in some, there will be a discussion about the final subsystem, including possible future improvements.

The subsystems the whole system was divided into are:

- **Modeling of System**
  The system was mathematically modelled to be able to simulate it. This chapter describes the behaviour of a single pendulum on a cart and also the behaviour of multiple pendulums on a single cart.

- **Component Selection**
  This covers the process of choosing components for the system with focus on finding parts with sufficient performance and compability while staying withing budget.

- **Simulation of System**
  This covers how the mathematical model from *Modeling of System* was implemented and simulated as well as how all system parameters were determined or approximated. How well the digital system actually simulates the real constructed system is also discussed.

- **Construction of System**
  This covers how the mechanical design and construction of the system were made. It includes early CAD-designs to the actual manufacturing of the designed system.

- **Electrical Construction & Programming**
  This covers how the electrical system and software were designed to run the chosen controller design.

- **Control of System**
  This covers how the models from *Modeling of System* were simplified and used to design the controller was made to balance the pendulums. Also how the controller later was simulated on the *digital version* of the system, that was developed in *Simulation of System*.

Subsequent Chapter *Results* covers the system as a whole with all the previous mentioned subsystems put together and evaluated in terms of how they perform together. The later discussion will also focus on the system as a whole.

2

Modeling of System

To break down the modeling process, the first task was to model a single pendulum on a moving cart. With that completed, three pendulums were modeled on a single cart.

2.1 One pendulum

The following model for a single inverted pendulum on a cart was obtained from [2]. This model will later be expanded to also describe multiple pendulums.

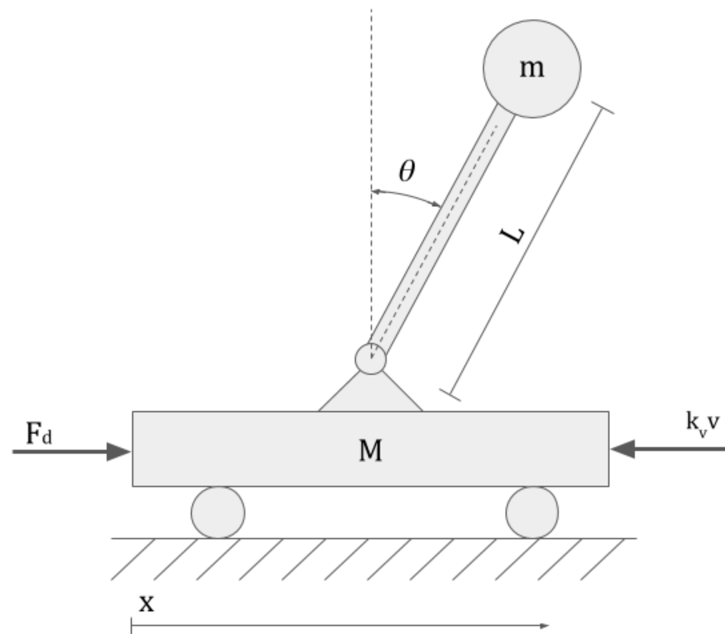


Figure 1: A single pendulum on a cart.

Given the system in Figure 1, we have defined the following variables:

- x_c - Position of the cart.
- θ - Angle of the pendulum relative to the vertical line.
- F_d - Force acting on the cart, in this case by the motor.
- F - Force acting on the pendulum
- m - Mass of the pendulum.

- L - Length to the center of mass of the pendulum.
- M - Mass of the cart.
- $k_v v$ - Horizontal friction.
- $\dot{x_c} = v$ - Speed of the cart

Analyzing the pendulum itself (m) we obtain the force equations

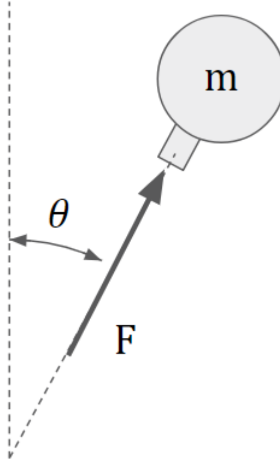


Figure 2: Forces acting on pendulum.

$$\begin{cases} \rightarrow: \sin(\theta)F = \ddot{x}_m m \\ \uparrow: \cos(\theta)F - mg = \ddot{y}_m m \end{cases}$$

$$\Rightarrow m\ddot{x}_m \cos(\theta) - m\ddot{y}_m \sin(\theta) = mg\sin(\theta) \tag{1}$$

The position of the pendulum can be described as

$$\begin{cases} x_m = x_c + \sin(\theta)L \\ y_m = \cos(\theta)L \end{cases} \tag{2}$$

Taking the second derivative gives

$$\begin{cases} \ddot{x}_m = \ddot{x} + L\frac{d}{dt}(\omega\cos(\theta)) = \ddot{x}_c + L\dot{\omega}\cos(\theta) - L\omega^2\sin(\theta) \\ \ddot{y}_m = -L\frac{d}{dt}(\omega\sin(\theta)) = -L\dot{\omega}\sin(\theta) - L\omega^2\cos(\theta) \end{cases} \tag{3}$$

where we have defined the angular velocity $\omega = \dot{\theta}$. Equation 1 and 3 combined results in

$$L\dot{\omega} + \ddot{x}_c\cos(\theta) = g\sin(\theta). \tag{4}$$

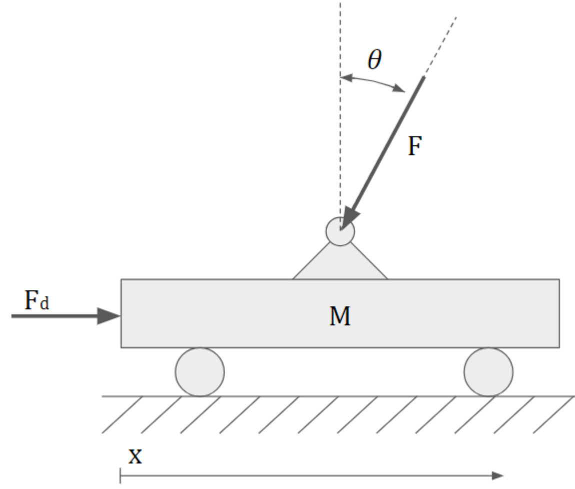


Figure 3: Forces acting on cart.

Analyzing the forces (except the friction force $k_v v$) acting on the cart horizontally gives

$$\rightarrow: F_d - F\sin(\theta) = M\ddot{x}_c = \{F\sin(\theta) = \ddot{x}_m m\} = F_d - \ddot{x}_m m,$$

$$\Rightarrow F_d = M\ddot{x}_c + m\ddot{x}_m. \tag{5}$$

Using the expression for $\ddot{x}_m$ from Equation 3 we get

$$F_d = (M+m)\ddot{x}_c + mL(\dot{\omega}\cos(\theta) - \omega^2\sin(\theta)).$$

Defining the velocity of the cart as $v = \dot{x}_c$ this can be written as

$$F_d = (M+m)\dot{v} + mL(\dot{\omega}\cos(\theta) - \omega^2\sin(\theta)). \tag{6}$$

From Equation 4 we obtain following expressions for $\dot{\omega}$:

$$\dot{\omega} = \frac{g\sin(\theta) - \dot{v}\cos(\theta)}{L}. \tag{7}$$

Using Equations 6 and 7 we obtain a new expression with $\dot{v}$, without including $\dot{\omega}$, i.e.

$$F_d = (M+m)\dot{v} + mL\left(\left[\frac{g\sin(\theta) - \dot{v}\cos(\theta)}{L}\right]\cos(\theta) - \omega^2\sin(\theta)\right).$$

This can be rewritten as

$$F_d = (M+m)\dot{v} + mg\sin(\theta)\cos(\theta) - m\dot{v}cos^2(\theta) - mL\omega^2\sin(\theta)$$

which gives

$$\dot{v} = \frac{F_d - mg\sin(\theta)\cos(\theta) + mL\omega^2\sin(\theta)}{(M+m) - mcos^2(\theta)}. \tag{8}$$

Thereby the expression for the cart acceleration in Equation 8 can be used for future simulation.

2.2 Friction

The model previously discussed in Section 2.1 does not include any friction or energy losses in the system. To include this in the model, a friction torque depending on ω is added to Equation 8 such that

$$\dot{\omega} = \frac{g\sin(\theta) - \dot{v}\cos(\theta)}{L} - k_\theta \omega \tag{9}$$

where k_θ is a friction constant.

The horizontal friction is included as an external force acting on the cart. It is proportional to the speed in which the cart is moving. This is then subtracted from the force F_d in Equation 5 such that

$$F_d = M\ddot{x}_c + m\ddot{x}_m + k_v\dot{x}. \tag{10}$$

With this expression taken into account, the cart acceleration becomes

$$\ddot{x}_c = \frac{F_d - mL(\dot{\omega}\cos(\theta) - \omega^2\sin(\theta)) - k_v\dot{x}}{(m+M)}. \tag{11}$$

To visualize how different pendulum lengths and cart masses results in different behaviour, Figure 4 shows multiple carts, each with a single pendulum and how they act different. All wagons have a force applied during a short period and pendulums start from standing vertical (0°).

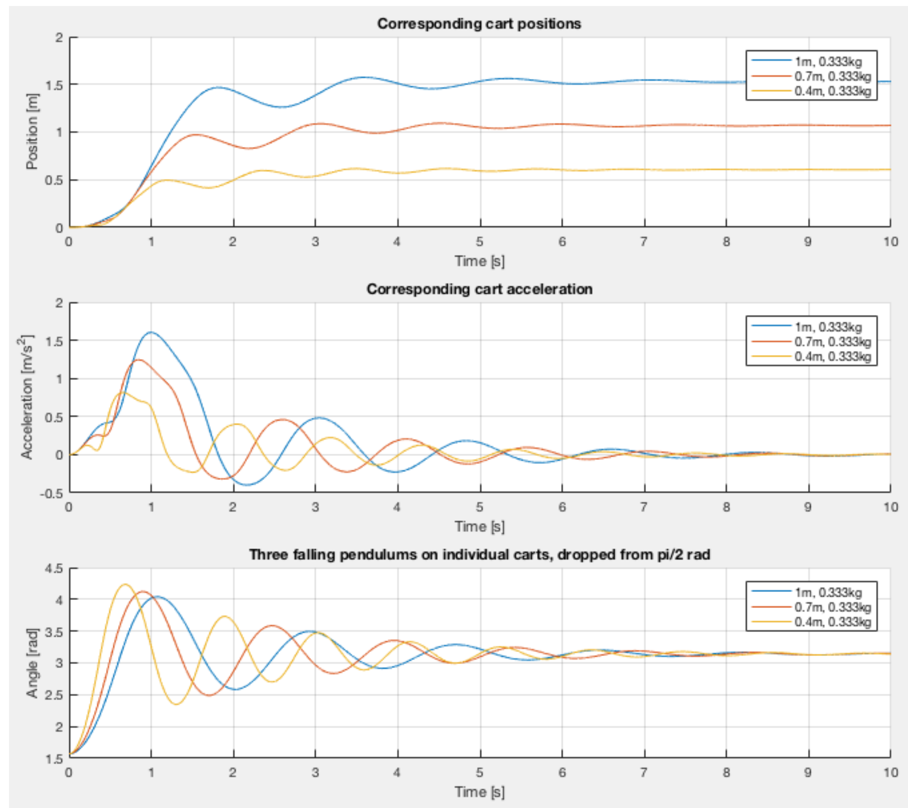


Figure 4: Different accelerations of the cart depending on the pendulum.

2.3 Three pendulums

The following section will describe how the formulas from Section 2.1 were modified to describe a cart with multiple pendulums, as illustrated in Figure.

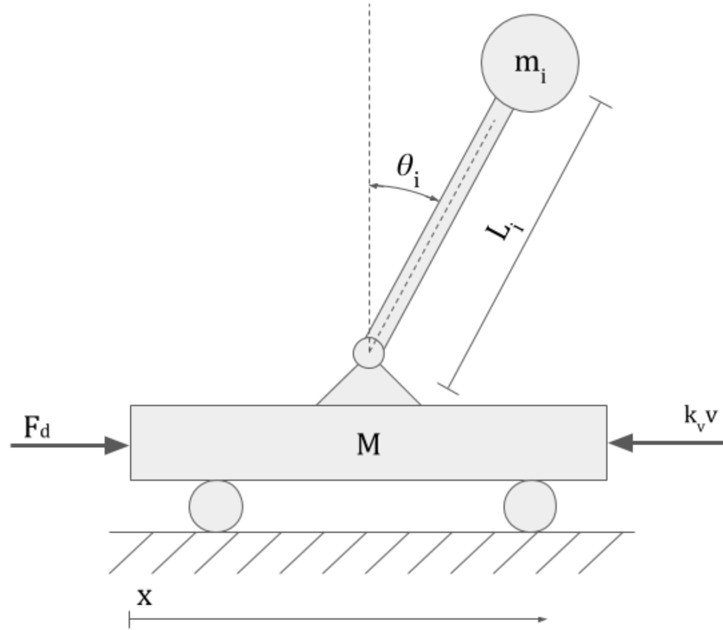


Figure 5: Pendulum on a cart.

For $i = 1, ..., N$ pendulums, each will satisfy Equation 4, i.e.

$$L_i\dot{\omega}_i + \ddot{x}_c \cos(\theta_i) = g \sin(\theta_i) \tag{12}$$

which gives the following expression for $\dot{\omega}$:

$$\dot{\omega}_i = \frac{g \sin(\theta_i) - \ddot{x}_c \cos(\theta_i)}{L_i}. \tag{13}$$

Equation 5 can be rewritten as:

$$F_d = M\ddot{x}_c + \sum_{i=1}^{N} m_i \ddot{x}_{mi}. \tag{14}$$

As in Equation 10, the horizontal friction is subtracted from the driving force, resulting in

$$F_d = M\ddot{x}_c + \sum_{i=1}^{N} m_i \ddot{x}_{mi} + k_v \dot{x}_c. \tag{15}$$

Equation 3 now becomes

$$\ddot{x}_{mi} = \ddot{x}_c + L_i\dot{\omega}_i\cos(\theta_i) - L_i\omega_i^2\sin(\theta_i). \tag{16}$$

Inserting 16 into 15 gives

$$\begin{aligned} F_d &= M\ddot{x}_c + \sum_{i=1}^{N}\left[m_i\left(\ddot{x}_c + L_i\dot{\omega}_i\cos(\theta_i) - L_i\omega_i^2\sin(\theta_i)\right)\right] + k_v\dot{x} \\ &= \ddot{x}_c(M + \sum_{i=1}^{N} m_i) + \sum_{i=1}^{N} m_iL_i\dot{\omega}_i\cos(\theta_i) - \sum_{i=1}^{N} m_iL_i\omega_i^2\sin(\theta_i) + k_v\dot{x}_c \end{aligned}$$

$$\Longleftrightarrow \ddot{x}_c(M + \sum_{i=1}^{N} m_i) = F_d - \sum_{i=1}^{N} m_iL_i\dot{\omega}_i\cos(\theta_i) + \sum_{i=1}^{N} m_iL_i\omega_i^2\sin(\theta_i) - k_v\dot{x}_c \tag{17}$$

Using the expression for $\dot{\omega}_i$ from Equation 13 in Equation 17 gives

$$\ddot{x}_c(M + \sum_{i=1}^{N} m_i) = F_d - \sum_{i=1}^{N} m_i\left[g\sin(\theta_i) - \ddot{x}\cos(\theta_i)\right]\cos(\theta_i) + \sum_{i=1}^{N} m_iL_i\omega_i^2\sin(\theta_i) - k_v\dot{x}_c$$

$$\ddot{x}_c(M + \sum_{i=1}^{N} m_i) = F_d - \sum_{i=1}^{N} m_ig\sin(\theta_i)\cos(\theta_i) + \sum_{i=1}^{N} m_i\ddot{x}_c cos^2(\theta_i) + \sum_{i=1}^{N} m_iL_i\omega_i^2\sin(\theta_i) - k_v\dot{x}_c$$

$$\ddot{x}_c(M + \sum_{i=1}^{N} m_i) - \sum_{i=1}^{N} m_i\ddot{x}_c cos^2(\theta_i) = F_d - \sum_{i=1}^{N} m_ig\sin(\theta_i)\cos(\theta_i) + \sum_{i=1}^{N} m_iL_i\omega_i^2\sin(\theta_i) - k_v\dot{x}_c$$

$$\ddot{x}_c\left(M + \sum_{i=1}^{N} m_i(1 - cos^2(\theta_i))\right) = F_d - \sum_{i=1}^{N} m_ig\sin(\theta_i)\cos(\theta_i) + \sum_{i=1}^{N} m_iL_i\omega_i^2\sin(\theta_i) - k_v\dot{x}_c$$

and eventually

$$\ddot{x}_c = \frac{F_d - \sum_{i=1}^{N} m_ig\sin(\theta_i)\cos(\theta_i) + \sum_{i=1}^{N} m_iL_i\omega_i^2\sin(\theta_i) - k_v\dot{x}_c}{M + \sum_{i=1}^{N} m_i(1 - cos^2(\theta_i))} \tag{18}$$

Thereby we have obtained the expression for the cart acceleration for a cart with N pendulums. An example simulation was done (see Figure 6), where a force is applied during a short period of time and all pendulums start from hanging down (180°).

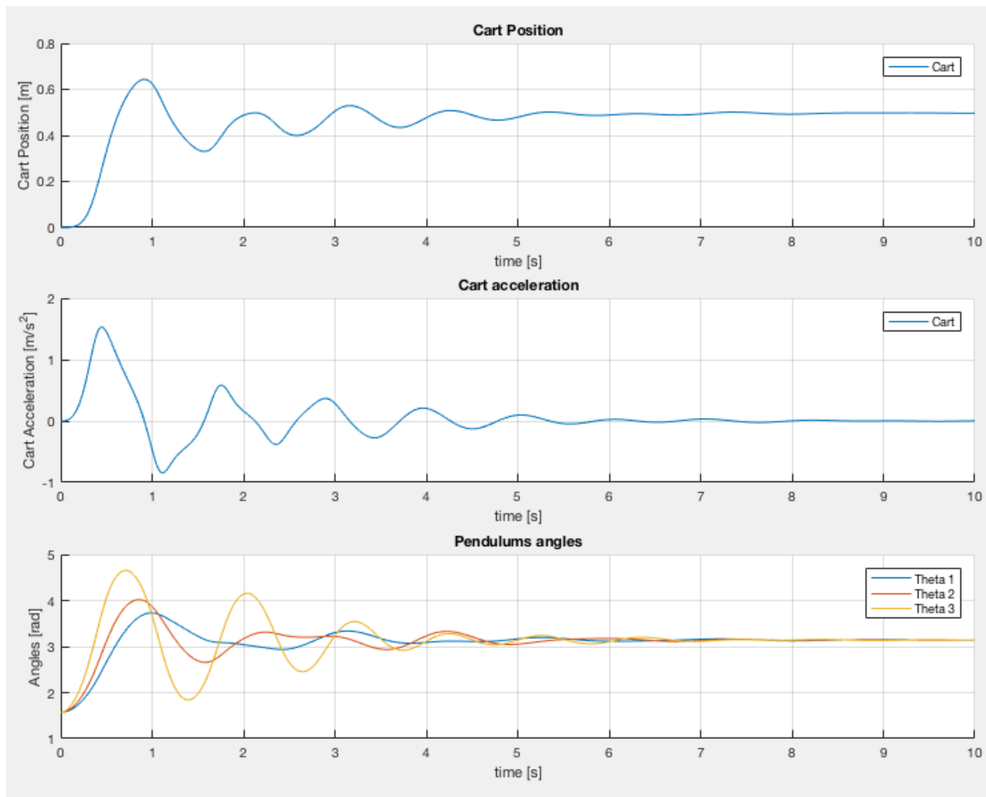


Figure 6: Three pendulums on the same cart.

2.4 Stepper motor characteristics

For enhanced control (compared to an ordinary DC motor) of the cart position, a two phase stepper motor will be used. This part will cover the basic dynamics and characteristics of the motor, along with an explanation of so-called microstepping.
A two phase stepper motor consists of a rotor and a permanent magnet with multiple poles that sits inside a stator. When a phase winding in the stator is energized, a magnetic dipole is created. By altering this energisation of the phases a rotation, or positional change, in the rotor can be produced.
The stepper motor model used was found in [10].

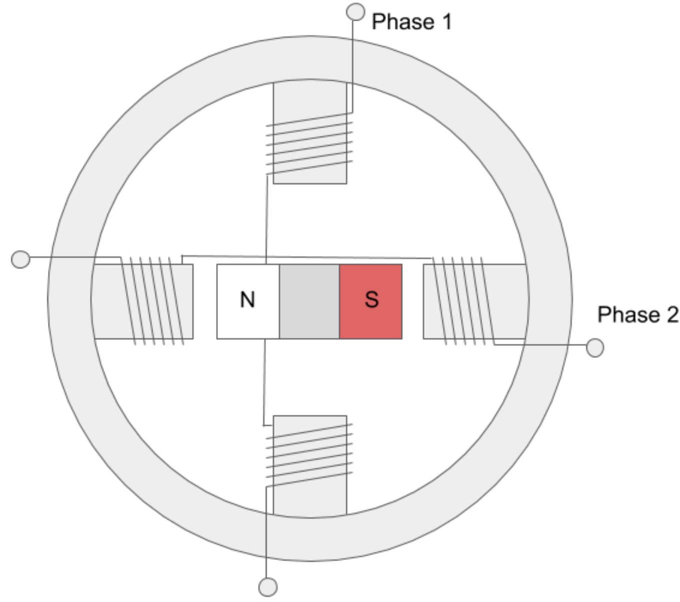


Figure 7: Rotor and stator principle of a stepper motor.

With n_s being the number of rotor pole pairs and m_s being the number of stator phases, the number of steps S per revolution can be calculated.

$$s = 2n_s m_s \tag{19}$$

Naturally, the stepping angle $\Delta\Phi$ is given by

$$\Delta\Phi = \frac{360°}{s} = \frac{360°}{2n_s m_s} \tag{20}$$

Each phase (j) generates a torque T_{Mj} that is dependent on the position of the rotor in relation to the phase coil itself, i.e.

$$T_{Mj} = k_m \sin(n_s\Phi(t) + \Phi_{0j})I_j(t) \tag{21}$$

Here, k_m is the motor constant, $\Phi(t)$ is the rotor position. Φ_{0j} is the location of the coil j in the stator and $I_j(t)$ is the coil current.
Next, the relation between current and voltage for each phase is formulated as

$$U_j = emf_j + RI(t) + L_s \frac{dI(t)}{dt} \tag{22}$$

Here, R is the resistance of the coils while L_s is the coil inductance. The induced electromotive force per phase, emf_j is expressed as

$$emf_j = k_m \sin(n_s \Phi(t) + \Phi_{0j})\omega \tag{23}$$

ω being the angular velocity of the rotor. Now, the total torque produced by the motor is the sum of the torque produced by all phases, i.e.

$$T_M = \sum_{j=1}^{m_s} T_{Mj} \tag{24}$$

Defining the inner friction in the motor as a damper, which gives a friction torque $D\omega$, the rotational torque equation for the motor can be expressed as:

$$J \frac{d\omega}{dt} = T_M - T_{load} - D\omega. \tag{25}$$

The torque that is delivered to the pendulum-cart is in this case T_{load}, which gives the final equation representing the torque from the stepper motor.

$$T_{load} = T_M - J \frac{d\omega}{dt} - D\omega \tag{26}$$

3

Component Selection

The components regarded as decisive for the performance of the final prototype are listed in Table 1 and motivated later in this section. Their position is shown in Figure 8, those unmentioned are located inside the electrical cabinet. The motivations are based on budgetary grounds and availability in terms of ordering and what the Department of Electrical Engineering could provide. Components that can be chosen more freely are motivated under *Other components* in Section 3.7.

Table 1: List of components.

Component	Pieces	Manufacturer	Article number
Angular position sensor	2	Novotechnik	P2501A202
Angular position sensor	1	Novotechnik	P2501A502
Microcontroller	1	Arduino	A000062
Stepper motor	1	Trinamic	QSH5718-56-28-126
Stepper motor driver	1	-	3128S
Rotary encoder	1	Yumo	E6B2-CWZ3E

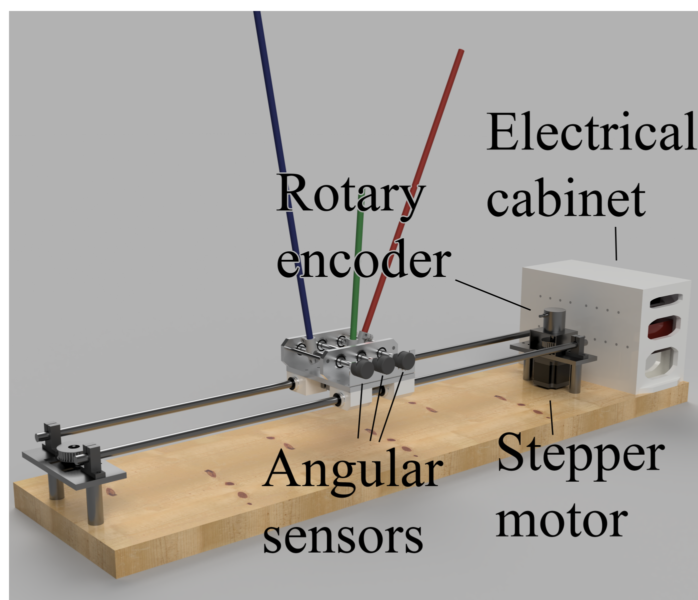


Figure 8: CAD-model.

3.1 Microcontroller

In earlier projects concerning inverted pendulums either an Arduino Mega2560 or an Arduino DUE was used as the microcontroller. Both of these microcontrollers have many digital I/O connections as well as analog inputs which make them modular and the system easy to expand. To further support fast data input the Arduino DUE, with its higher clock rate, is regarded as the best choice of microcontroller in the Arduino family. However, the Arduino DUE comes with the disadvantage of lower operation voltage, 3.3 V instead of Arduino Mega's 5 V. In practice it will affect the compatibility of the microcontroller. In this project it is possible to solve the incompatibility with a voltage divider. Henceforth the Arduino DUE will be referred to as the microcontroller. [11]

In Table 2 the most important specifications regarding the Arduino DUE (and its microcontroller *AT91SAM3X8E*) can be found. The data was retrieved from [11] and [12].

Table 2: Specifications of Arduino DUE.

Microcontroller	AT91SAM3X8E
Operating Voltage	3.3V
Input Voltage (recommended)	7-12V
Analog Resolution	10 & 12 bit
Clock Speed	84 MHz
Input leak current ADC	$\pm 0.5\mu A$

3.2 Angular sensor

The choice of angle sensor was mostly influenced by the limited budget and available components. In previous projects two high-precision potentiometers were used [13] with a repeatability of 0.01°. These sensors had a dead zone of 15° (will be described in 5.1) but this will not affect this system since the pendulums never make full rotations. The cost of one similar angular sensor is less than three new ones with lower, but still adequate tolerance. The angular sensors used in this project are listed in Table 1 and have specifications that can be found in Table 3.

However, the usage of these angular sensors used in older projects [13] is not optimal in terms of resolution. Here, a 10 bit reading is used although the internal Analog to Digital Converter (ADC) of the microcontroller has a 12 bit reading resolution [11]. No complaints were stated about the resolution of the angular sensors used in [13] even though they could not have obtained a higher resolution than 0.33° with the 10 bit reading resolution that was used.

Table 3: Specifications of P2501A202, {P2501A502}.

Range	345 ± 2 °
Resistance value	$2\ k\Omega$ $\{5\ k\Omega\}$
Repeatability	$0.003\%(0.01°)$
Recommended current	$\leq 1\mu A$

3.3 Rotary encoder

The purpose of the rotary encoder is to calculate the position of the cart, accurate position determination is important when implementing the model. The resolution of the rotary encoder left over from earlier work[6] and described in [14] is 1024 pulses per second. Based on the radius of the conveyor pulley, one rotational step of the rotary encoder corresponds to 0.12 mm. This is evidently the smallest distance that the encoder can measure.

The resolution, or repeatability, of the angular sensors described in Section 3.2 is 0.01°. Herein lies a potential conflict between the angular sensors and the rotary encoder. As illustrated in Figure 9, if the cart is hastily moved in any direction, the initially upright pendulum will stand at an angle β and the cart will have moved a distance d. The horizontal distance that the pendulum top has moved is assumed to be zero. If the angle β is equal to the smallest angle that the angular sensor can recognize, the distance d can be described by the angular sensor information if the length of the pendulum is known. If the resolution of the rotary encoder is rougher than the angular sensor, the estimation of d will be zero according to the rotary encoder, but a value above zero according to the angular sensor. Even though it can be regarded as a bottleneck, using this rotary encoder is advantageous in an economic perspective, since it was left over from earlier work [6].

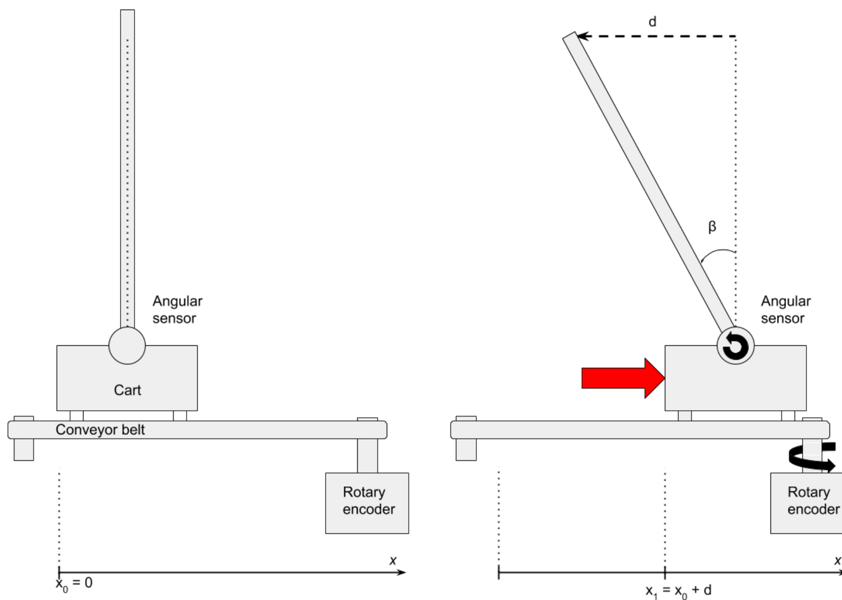


Figure 9: Movement interpretations of different sensors.

Table 4: Specifications of the rotary encoder.

Resolution	1024 P/R
Input voltage	5-12 V

3.4 Motor

An important specification of the motor is the torque. An approximation of the absolute minimum torque is to say that the force inflicted on the cart from falling pendulums is what needs to be counteracted by the motor. The acceleration of a cart with one pendulum of length 1 m is approximately 1.6 m/s^2 according to Figure 4. For pendulums of 0.7 m and 0.4 m they are approximately 1.3 m/s^2 and 0.7 m/s^2 respectively. If they were on the same cart and dropped at the same time, the acceleration would not exceed the sum of the three, i.e. $a = 3.6$ m/s^2 . With a pulley of radius $r = 0.02$ m and a cart with a mass of $M = 2$ kg the torque needed to be generated is $T = M \cdot a \cdot r = 0.14$ Nm. If this torque was inflicted on falling pendulums, the result would be that the pendulums stopped falling but they would not rise up again. this is clearly insufficient and more torque than calculated is needed. The chosen motor can provide 1.26 Nm and is considered large enough.

Table 5: Specifications of stepper motor.

Rated voltage	2.52 V
Max applicable voltage	75 V
Rated phase current	2.8 A
Holding torque	1.26 Nm
Step angle	1.8°

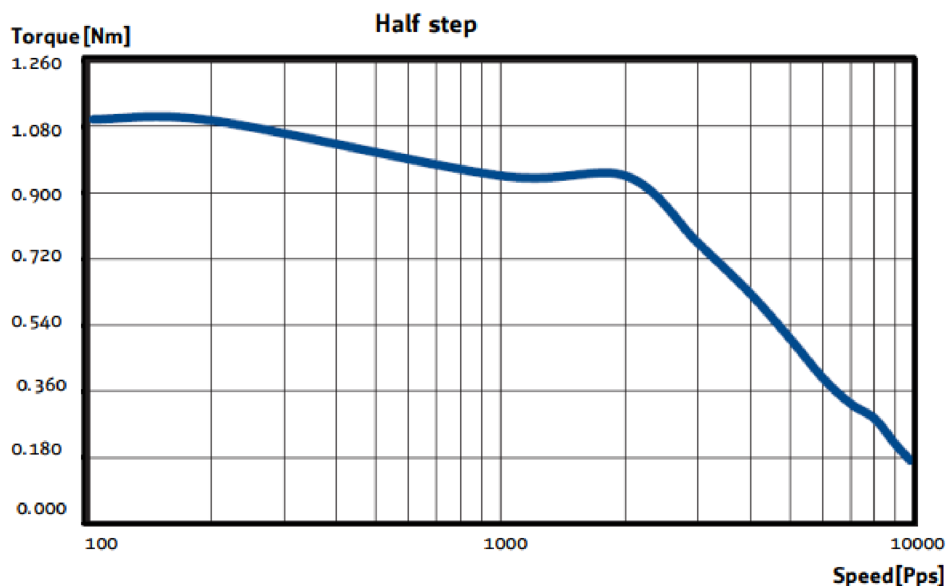


Figure 10: Speed Torque curve of stepper from its datasheet [1].

From the speed-torque curve it can be obtained that the maximum speed of the motor is approximately 10000 pulses/s. Using a half step of $\Delta\Phi = 1.8°$ and converting to rad/s gives.

$$10000\frac{\Delta\Phi}{2}\frac{\pi}{180} = 10000(0.9)\frac{\pi}{180} \approx 157\ rad/s \tag{27}$$

3.5 Stepper motor driver

The stepper motor is able to handle voltages up to 72 V although a lower voltage of 30 V is sufficient based on the speed-torque characteristics of chosen motor [1]. Even though the chosen driver has more than enough voltage to deliver, 40 V, it was best in respect to cost and supplier [15].
Another important specification is with what precision the stepper motor driver can controll the motor. The chosen motor has a step angle of 1.8° but with microstepping enabled, the motor can be instructed to move parts of those steps e.g. $1/4^{th}$ of a step. To match the rotary encoder which detects difference of $\frac{360°}{1024} = 0.35°$ the stepper motor driver have to support microstepping of at least $\frac{0.35}{1.8} \approx 1/5$. The chosen stepper motor driver supports microstepping up to 1/128

Table 6: Specifications of stepper motor driver.

Nominal voltage	40 V
Nominal current	3 A
Microstepping	Up to 1/128

3.6 Power supply

In this project, two different power supplies are used. One for powering the microcontroller and one for the stepper motor driver. For the microcontroller a regular micro-USB charger with 12 V and 0.5 mA output is used. The power supply that best suited the stepper motor driver, in terms of cost, manufacturer and specifications, is able to deliver a voltage of 36 V and current of 9.3 A. Although it is oversized for this application it could be bought from a supplier from which other components were ordered and therefore at a lower cost [16].

Table 7: Specifications of power supply of stepper motor driver.

Nominal voltage	36 V
Nominal current	9.7 A

3.7 Other components

The primary focus for choosing the remaining components were to minimize the cost in order to stay within budget. Therefore, a minimum amount of orders were made in order to reduce shipping costs. Another focus was to minimize environmental impact where it was possible. This was achieved partly by reusing old components, and also by trying to construct the system with a minimal size but yet large enough to be suitable for demonstration in e.g. a lecture hall.

4

Simulation of System

The whole system model given in Section 2 was implemented in Simulink to have a digital version of the whole system to conduct tests on during the development of the control system. This chapter will describe how the first simulations were implemented from previous formulas and which parameters were used. Some parameters are from arbitrary components since the simulation is implemented before the actual system is built.

4.1 Stepper motor

The used stepper motor in the simulation is the one chosen in Section 3. All parameters will be explained how they are calculated from the parameters in the datasheet. All relevant parameters from the datasheet can be found in Table 8.

Table 8: Parameters from the datasheet of stepper motor [1].

Parameter	**Variable**	**Value**
No. of phases	m_s	2
Rated phase current	I_{max}	2.8 A
Holding torque	T_{hold}	1.65 Nm
Max applicable voltage	U_{max}	75 V
Step angle	$\Delta\Phi$	1.8°
Rotor inertia	J	300 gcm^2
Phase resistance	R	0.9 Ω
Phase inductance	L	2.5 mH

4.1.1 Motor constant

The motor constant k_m is determined from the holding torque and rated phase current. Assuming that only one phase is active, microstepping is disabled and that the rotor is static and aligned with the active phase so that maximum possible torque is generated.

If the rotor is aligned with the active phase then the sinus function equals 1. This results in the following simplified version of Equation 21.

$$T_{Mj} = k_m I_j(t)$$

Since we assumed that only one phase is active, T_{Mj} is also the total torque produced by the motor, i.e.

$$T = k_m I_j(t)$$

Assuming that the rated phase current is flowing through the phase and that the torque produced is the *holding torque* makes it possible to determine the motor constant k_m from the parameters from Table 8, i.e.

$$k_m = \frac{T_{hold}}{I_{max}} = \frac{1.65}{2.8} \approx 0.59$$

4.1.2 Number of rotor pole pairs

The number of rotor pole pairs can be calculated from the step resolution $\Delta\Phi$.

Rewriting Equation 20 as:

$$n_s = \frac{360}{2m_s\Delta\Phi}$$

Which gives the following expression for n_s:

$$n_s = \frac{360}{4(1.8)} = 50 \tag{28}$$

4.1.3 Damping constant

The damping constant was determined experimentally by studying the speed response while applying a ramp as speed input to the stepper. It was noticed that a high damping constant makes the speed collapse earlier, while a too low damping constant introduces resonances and makes the speed collapse even earlier. The optimal constant was found to be $D = 0.02$, which gives the speed response in Figure 11.

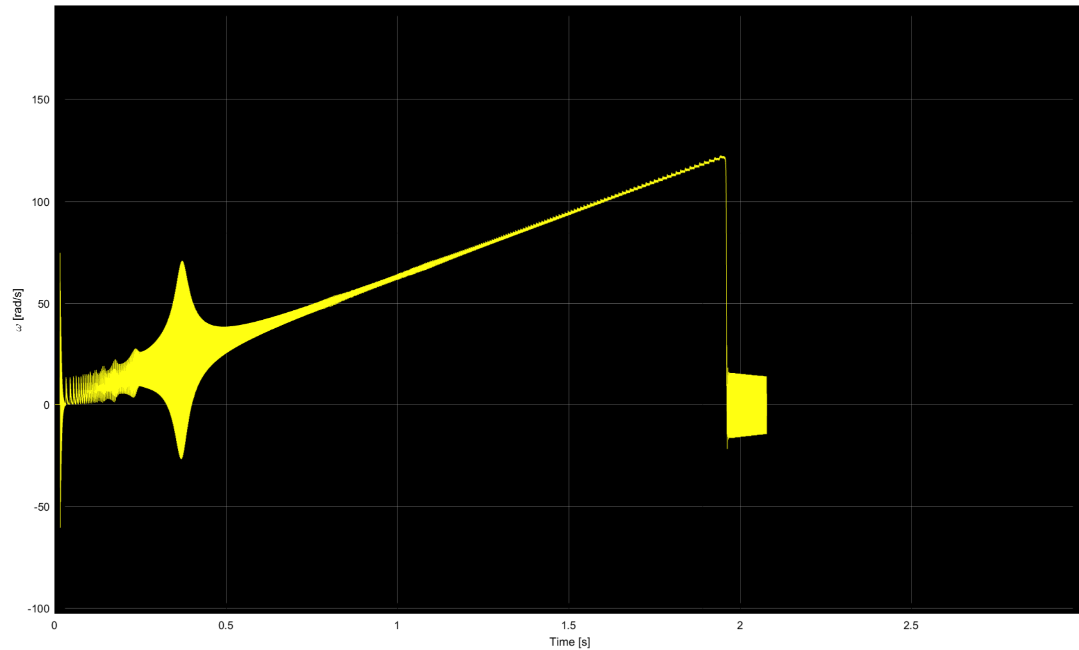


Figure 11: Speed response of stepper.

The simulation plotted in Figure 11 shows the resulting speed while applying a ramp input to the stepper driver. There occurs resonances at lower speeds and as can be seen, the speed collapses at around 120 rad/s, which is below the maximum speed of 157 rad/s, found in Equation 27.

4.1.4 Simulation of stepper driver

The model for a stepper motor previously given does only describe the motor response given the phase voltages, but a driver must be implemented to generate the phase voltages shown in Figure 12.

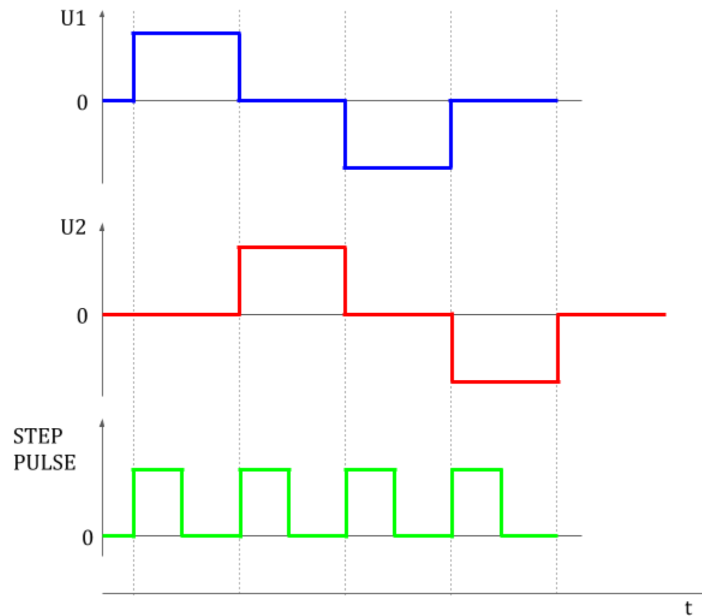


Figure 12: Relation between step pulse input and the phase voltages.

Implementing a driver following the repeating sequence shown in Figure 12 results in the following angle response of the stepper motor.

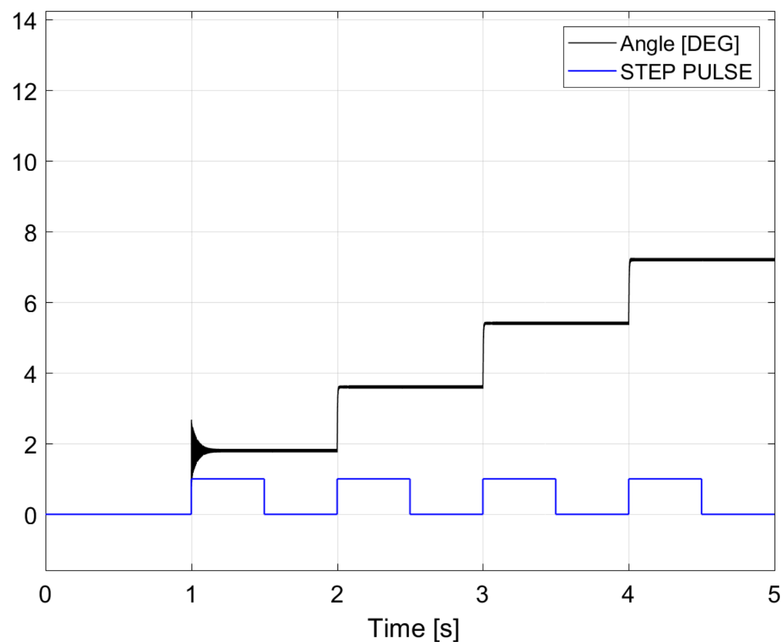


Figure 13: Angle response of the implemented driver.

As expected, the rotor moves one $\Delta\Phi$ - step at each rising edge of the *STEP PULSE* signal. For this motor $\Delta\Phi = 1.8°$ and matches the increment shown in Figure 13, thereby this simulated driver is considered good enough for further simulation use.

4.1.5 Simulation of microstepping

A function shared by many stepper drivers is *microstepping*. Applying partly full phase voltages results in that each mechanical step is divided into sub-steps. An example of how this could look is shown in Figure 14.

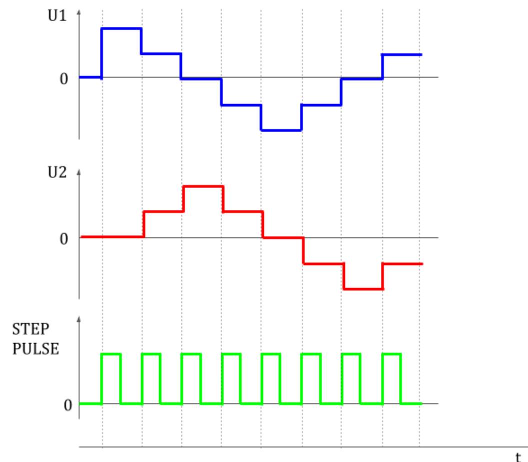


Figure 14: Voltages of microstepping.

However, to keep simplicity within the model and avoid creating a more complex driver, the number of rotor pole pairs is increased. Basically decreasing the motors mechanical step size to the chosen microstep. This could be a problem due to that this method would mean that the simulated torque could be larger than what would be produced by the real microstepped motor.

Thus, the number of rotor pole pairs n_{new} will be calculated from the real number of pole pairs n_{real} and number of microsteps Δn_{micros}, i.e.

$$n_{new} = n_{real}\Delta n_{micros} \tag{29}$$

Applying microstep of 2 ($\Delta n_{micros} = 2$) upon the previously simulated stepper from Figure 13 results in the following step response:

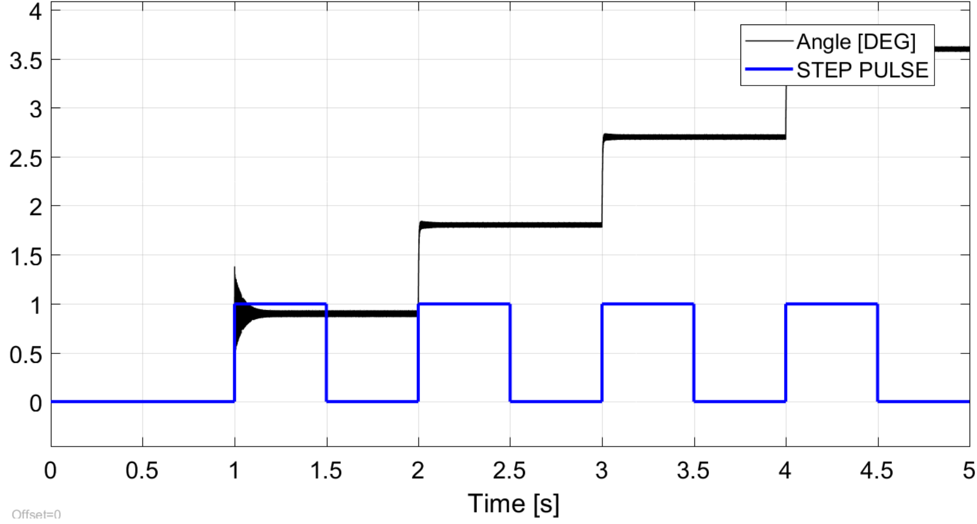


Figure 15: Angle response of microstepped driver.

As can be seen in Figure 15 the step angle is 0.9° which means that Equation 29 is correct regarding the step size.

4.2 Parameters of cart and pendulums

This section describes how the parameters for the cart and pendulums were obtained and approximated.

4.2.1 Determination of average operation speed

The friction for the cart and pendulum both have been approximated to be linear with respect to the velocity of the cart and the pendulum. To approximate this constant a typical operating speed is required to approximate these.

The typical speed is calculated from the scenario that the cart need to match the horizontal speed of a falling pendulum, that have accelerated from angle $\theta = 0°$ to $\theta_f = 10°$, and assuming the pendulum is of length 1 m ($L = 1$).

The total vertical distance to fall, is then

$$\delta y = L(1 - cos(\theta_f)) \approx 0.015 \ [m]$$

and the time it takes to fall δy from stationary can be calculated as

$$\delta y = a t_f^2 \ \Rightarrow t_f = \sqrt{\frac{\delta y}{a}},$$

where the acceleration a here is the gravitational acceleration g. Given the total fall time t_f, we can calculate the vertical speed v_y at θ_f:

$$v_y = gt_f = g\sqrt{\frac{\delta y}{g}} = \sqrt{g\delta y}$$

Given the vertical speed and the angle of the pendulum we can calculate the horizontal speed v_x as:

$$v_x = \frac{v_y}{\tan(\theta)}$$

This gives the following expression for v_x:

$$v_x = \frac{\sqrt{g\delta y}}{tan(\theta_f)} = \frac{\sqrt{9.81(0.015)}}{tan(10°)} \approx 2.18\ [m/s] \tag{30}$$

4.2.2 Friction constants

In a document from SKF [17] it is shown that linear ball bearing rails (LBBR) typically have a static friction force and a dynamic friction force. It will be assumed that the motor used in this project will overcome the static friction and because of that we use the given dynamic force to approximate the friction constant for the pendulum model.

From Table 3.8 in [17] the dynamic friction force for 12 mm diameter shaft bearing is 1.5 N. Assumed that this force is measured at this system's typical speed, Equation 30, gives the following expression:

$$\dot{x}k_v = v_x k_v = F_f \Rightarrow k_v = \frac{F_f}{v_x} = \frac{1.5}{2.18} \approx 0.69\ [N/(m/s)]$$

The friction constant k_θ for the pendulums axis is arbitrarily chosen to 0.5. This rough estimate is acceptable due to its small influence to the rest of the system.

4.2.3 Masses and lengths

The masses and lengths of the mechanical components were chosen arbitrarily.

4.3 Complete list of parameters for simulation

The previously given and calculated parameters gives the following tables of parameters used to simulate a cart with a single pendulum (Table 9) and with extended parameters for a cart with three pendulums (Table 10).

Table 9: Complete parameter list for simulation.

Parameter	Variable	Value
No. of phases	m_s	2
No. of rotor pole pairs	n_s	50
Phase voltage	U	75 V
Rotor inertia	J	300 $gcm^2 = (0.3)10^{-4}kgm^3$
Phase resistance	R	0.9 Ω
Phase inductance	L_s	2.5 mH
Damping constant stepper	D	0.02 Nm/(rad/s)
Pendulum length	L	1 m
Pendulum mass	m	1 kg
Pendulum friction constant	k_θ	0.5 Nm/(rad/s)
Cart mass	M	2 kg
Cart friction constant	k_v	0.69 N/(m/s)
Radius of belt pulley	r	0.02 m

4.4 Simulation of unregulated model

The simulations for a single pendulum and multiple pendulums, will use a similar input signal to their stepper drivers (see Figure 16).

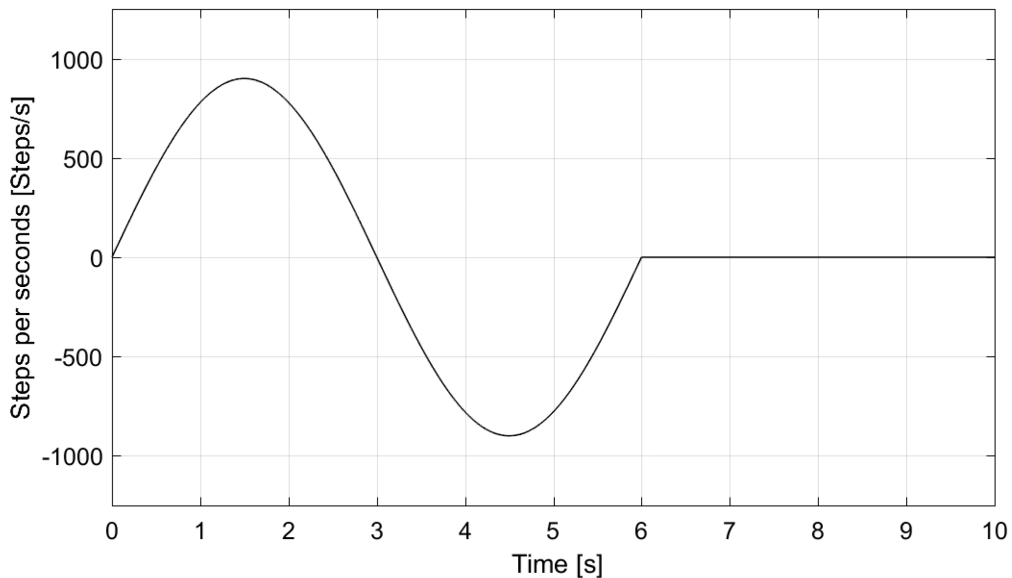


Figure 16: Input signal to simulations.

The following variables will be plotted for each simulation.

- Cart position - x_c
- Cart velocity - $\dot{x_c}$
- Pendulum angle - θ

4.4.1 One pendulum on a cart

In figures 17, 18 & 19 the resulting behaviour of the simulated system is presented. The pendulum is hanging down in its stable position at 180°, the cart is stationary and the stepper motor is given the input signal previously specified.
The plots will show how the system physically behaves and how the discrete nature of the stepper affects the pendulum and cart.

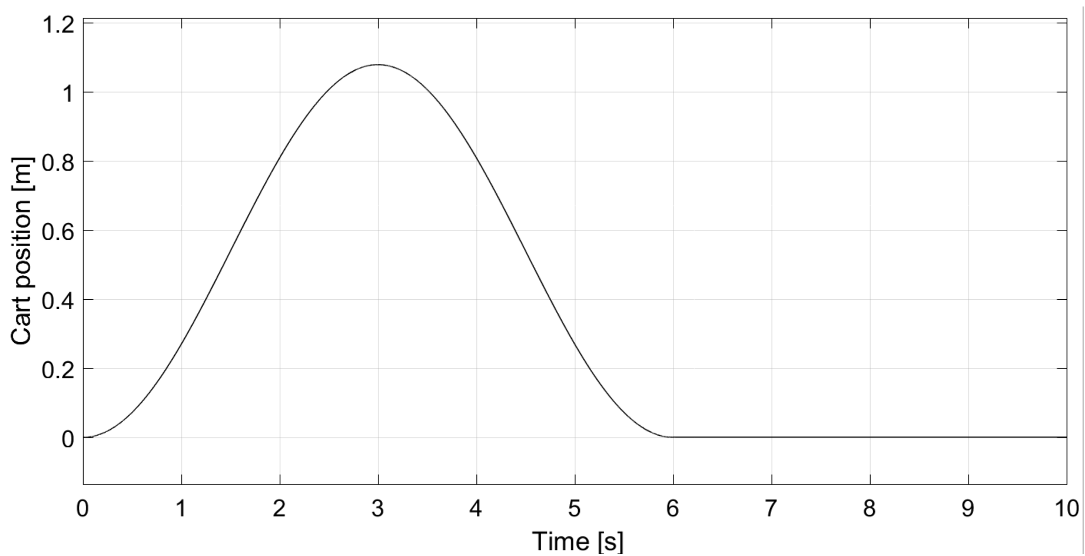


Figure 17: Cart position.

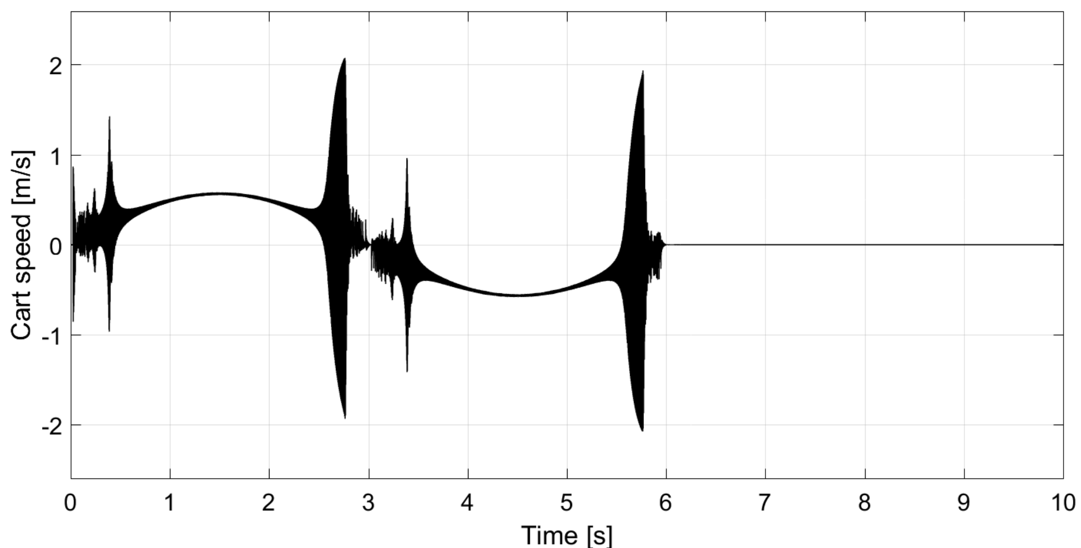


Figure 18: Cart velocity.

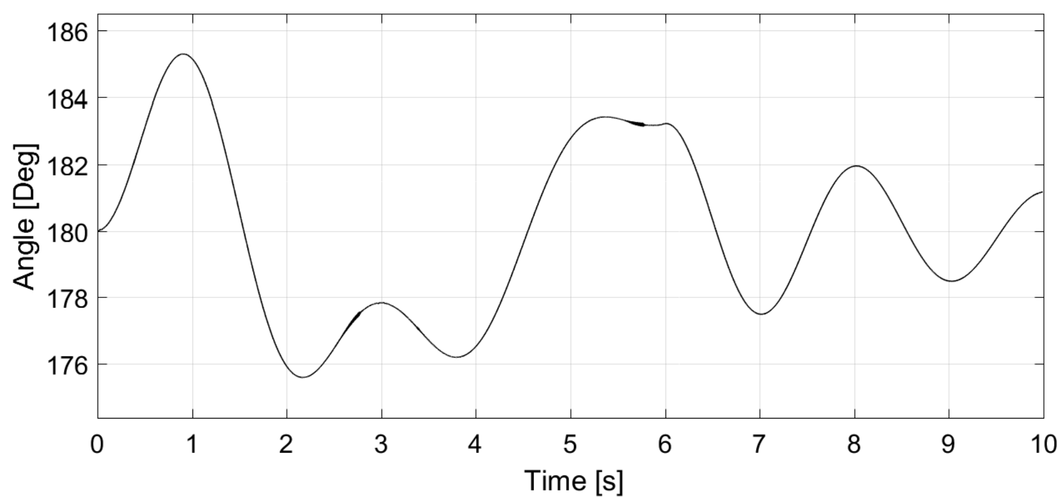


Figure 19: Pendulum angle.

4.4.2 Three pendulums on a cart

The simulations of three pendulums, the parameters can be found in Table 10. The response from the simulated system is presented in figures 20, 21 & 22.The pendulums and wagon starts stationary, and all pendulums begins at 180°.

Table 10: Extended parameters for three pendulums.

Parameter	**Variable**	**Value**
Pendulum 1 length	L_1	0.5 m
Pendulum 2 length	L_2	1 m
Pendulum 3 length	L_3	2 m
Pendulum 2 mass	m_1	0.5 kg
Pendulum 2 mass	m_2	0.5 kg
Pendulum 3 mass	m_3	0.5 kg

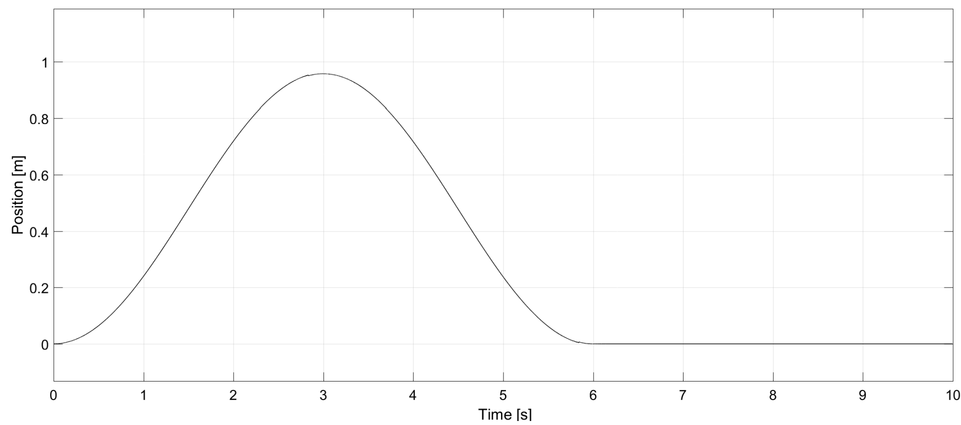


Figure 20: Cart position.

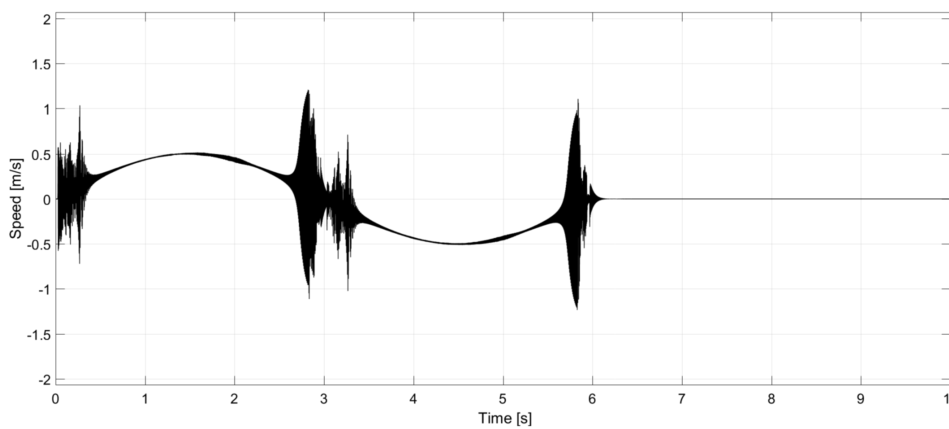


Figure 21: Cart velocity.

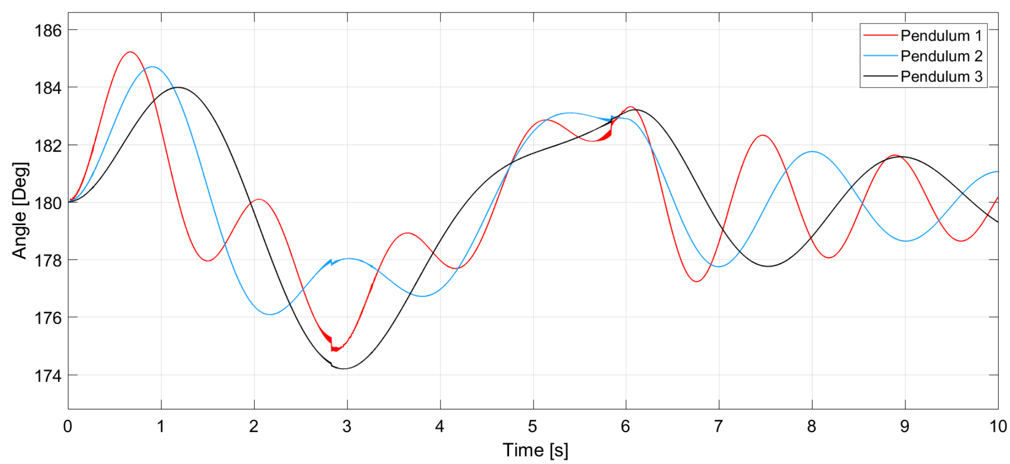


Figure 22: Pendulum angles.

4.5 Limitations due to stepper motor

As mentioned in Section 4.1.3 it was problematic to simulate the stepper motor at high speeds, where the speed collapses far to early compared to its rated speed. It is currently unknown what caused these problems but it might be the discrete nature of the stepper motor or a simulation issue with the stepper driver.

5 Construction of System

Before constructing the physical system, some literature review was conducted. After that the system was designed and constructed. Lastly, components were chosen to fulfill the system requirements.

5.1 Previous work

A mechanical construction for pendulum experiments has previously been built at Chalmers and this construction along with its components were accessible for reuse. When considering construction issues, some reports associated with this construction were examined to gain knowledge from their experiences of either constructing or using the existing system. One report stated that the linear bearings in the old construction needed replacement due to increased friction[13]. According to that report, the major problem they had was friction throughout the whole system.

Further on it is recommended that a stepper motor should be used for enhanced position detection. Also the encoder with a resolution of *1024P/R* was causing problems for the *Arduino Due* by sending data with a higher frequency than the board could handle. The timing belt for the cart was stretchable which also affected the measured position of the cart and seemed to be a problem.

Mentioned in another report is also the fact that the position of the cart was inexact and that the angular sensors had a dead zone in which no measuring occurred[6].

5.2 Design of mechanical construction

This section describes how the system was constructed mechanically. First by laying out the outline specifications of the construction and then describing how it was made along with discussing the result.

5.2.1 Construction

Initially, a specification stating the desired size of the system was created (see Table 11).

Table 11: Construction specifications.

Length	1 m
Height	irrelevant
Width	0.2 m
Weight of cart	max 2 kg
Total weight	max 10 kg
Length of a pendulum	max 1 m

A solution-selection matrix, (see Appendix C), was created to help sort out the best way to construct each component. Solutions chosen are marked in boldface and were mainly chosen with performance, time and budget in mind. The CAD model presented in Figure 23 was developed and used as a guide for what was aimed to be built.

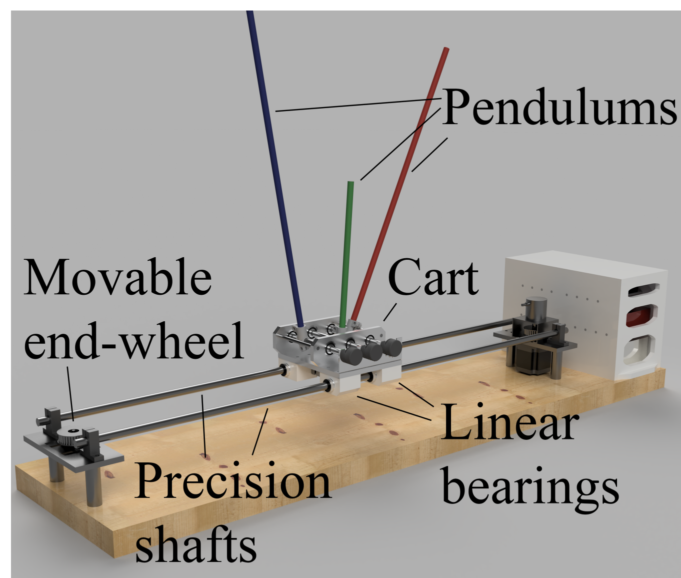


Figure 23: CAD-model.

In the bottom of the construction a base plate made of birch was created, onto which everything else would be mounted. Upon that four aluminum spacings were fastened to make room for the motor and cart movement. Two precision shafts were mounted at each side on top of aluminum blocks to allow the cart to move along with low friction using linear bearings. At the opposite side of the motor, a movable end-wheel was constructed to allow tightening of the timing belt. A rotary encoder (for cart position input) was mounted on the motor side and connected to the motor and timing pulley shaft through a bushing.

Three shafts were mounted on the cart with angular sensors (for pendulum angle measurement) connected to each one. Plastic T-connectors were used for mounting the pendulums to allow a modular design. Ball bearings were used for the shafts on the cart and for the end-wheel to minimize friction. Regarding materials, a goal was to make the system lightweight but still robust. Parts with precision requirements were constructed with high-grade aluminum, but for the base plate wood was more lightweight and robust enough. Parts with lower demands on weight and tolerance were 3D-printed in plastic to stay within budget and to keep the system lightweight.

Three pendulums were built in different lengths with weights at their very top to help differentiate the center of mass between the three for control purposes.

5.2.2 Results & Discussion

Figure 24 shows how the final construction turned out and a drawing of the whole system can be found in Appendix A. In Table 11 the specifications of the mechanical construction are defined. The measurements regarding the mechanical parts, including the aluminum spacings, plates and shafts, meet the requirements. The birch base however had to be extended in both length and width to house the electronics.

Figure 24: Mechanical construction.

The solution-selection matrix (see Appendix C) was followed thoroughly with the exception that the rubber cart stop was replaced by a software implementation making the cart stop before running out of space. Regarding functionality, the cart moved along the shafts with reasonably low friction, the timing belt was able to be tightened and the pendulums could swing in a wide enough zone with low friction. The overall build was robust, stable and did not twist or shake during operation.

The weight of the cart met the requirements. However, it should be noted that the weight of the pendulums was not included in this calculation as their weight was not specified at the time. The total weight of the system was not measured but the system was convenient enough to handle. Any pendulum could be mounted so the specification of max pendulum length (1 m) should just be seen as a safety recommendation if one would fall unexpectedly.

The movable end-wheel design was changed during construction due to alternate design choices found. However, this was not optimal as the bolts for fastening it were placed tightly below the pulley and in that way unable to be fastened. Masking tape was used as a temporary fix, but a better method would be suggested as a permanent solution.

The metal plate on which the angular sensors were mounted had some precision issues due to inaccurate sheet metal bending. Even the slightest misalignment of the angular sensor position created a springy effect on the shaft which the pendulums were attached to. Some mechanical adjustments were made to reduce the problem to an acceptable level, but a more accurate method than sheet metal bending would have been preferred.

Regarding the 3D-printed components, a problem that arose was that the actual measurements were smaller than the planned ones. To get the right size of the components, adjustments had to be made to the CAD models. Furthermore, the 3D-printed linear bearing housings suffered from the same problem. When fastened to the base plate of the cart the bearings were fixed in a slightly offset position, resulting in a considerable increase in friction when running along the shafts. This was solved by loosening the bolts slightly to give the housings some leeway. Moreover, the T-connectors were not as flexible as expected, which made the attachment of the pendulums harder. Also, this was temporarily solved using masking tape.

6

Electrical Construction & Programming

In order to achieve a demonstrative equipment and a product possible to act as a test bed for control systems, certain requirements on the electrical construction and software design need to be met. The electrical construction must both be well documented and expandable to facilitate future projects. This means drawings of electrical circuits and room for more components/wires. The software, as well, has to enable extensions and that is achieved by clearly splitting up the software in different parts. This is described more thoroughly in Section 6.2.

A demonstrative equipment requires an intuitive user interface and easy interactive design. Further reading will explain how this is handled in the implementation.

6.1 Hardware design

Although the system consists of several components, its main function is fairly simple. The, in control theory, frequently used block diagrams easily describe the various uses of the selected components. To explain the basic functions of each part, they are divided into categories based on their field of operation. As illustrated in Figure 25 for this system.

The first category, power supply, consists of a main power supply connected to the external powerline. Internally, the connection to the external powerline is divided in two; the purpose of which is to connect the smaller microcontroller 12V DC power supply adapter while still providing a higher voltage (36V DC) to the stepper driver.

The microcontroller serves as the system hub. Here, sensor feedback from the rotary encoder and the angular sensors are manipulated by the user in order to calculate the control signal. The control signal generates a series of voltage pulses in the stepper driver, essentially producing a rotation in the motor based on the sensor input.

The different parts of the system illustrated in Figure 25 and commented above, will be more thoroughly described.

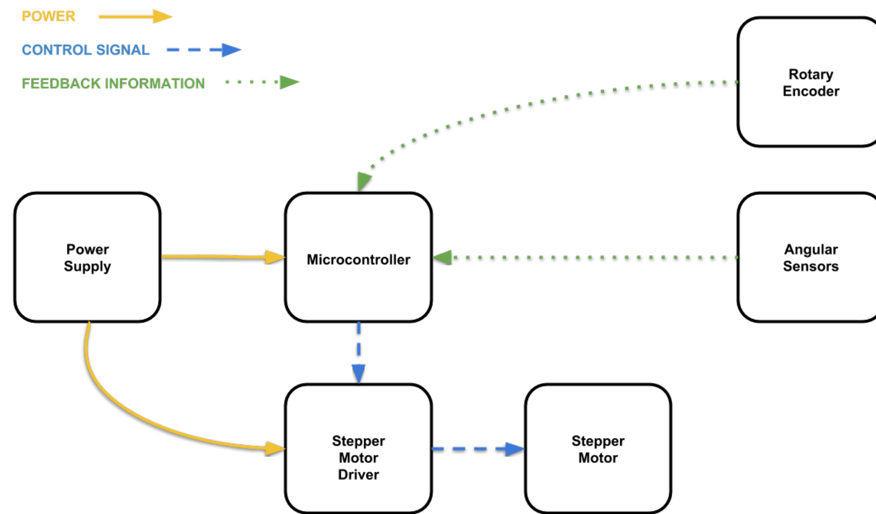


Figure 25: Hardware Principle Map.

6.1.1 Stepper motor

The detailed description concerning the chosen stepper motor is found in Section 3.4, while an in-depth study of the basic mechanism of it was given in Section 4.1. Here, the main focus will lie on how to connect it to the system.

As mentioned earlier (Section 4.1), the rotational motion of the rotor is the product of directing a current through the different stator phases, and thus creating a magnetic dipole in the stator that opposes the pole position of the rotor. By applying a voltage over each phase in series, enough electromagnetic force is produced to move the rotor one step.

This stepper motor is of the bipolar type, meaning it has two phases and thus two voltage phases need connectivity. Basically, two wires per phase are required in order to run a current through each phase. These wires are *A+*, *A-*, *B+* and *B-* for the phases *A* and *B*. An illustration of this can be seen in Figure 26.

Figure 26: Stepper Motor.

6.1.2 Stepper motor driver

Stepper motors that operate on higher voltages than a microprocessor can provide, need external power supply. The voltage also needs to be distributed over the different phases in the correct order to move one step; a stepper driver acts as an interface between the motor and the user. By only giving the driver a low voltage step signal, (*Pulse* in Figure 27) from the microprocessor, a direction signal (*Dir*) and a higher terminal voltage from the power supply (*Vcc & GND*), the driver has enough to convert the information into square wave voltage pulses that the stepper motor phases need to produce a rotational step.

The selected driver also provides the possibility to use so called microstepping. This is a method to increase the number of steps per revolution by simply modifying the pulses sent to the stepper motor phases. However, this is not given as an input from the microcontroller, but set via a DIP-switch on the actual driver hardware.

The driver also has an input labeled *Inactivate* which sets the voltage on the outgoing phases (*A+, A-, B+, B-*) to 0 V. Furthermore, the driver needs an operating voltage of 5 V DC, which is delivered from the microcontroller.

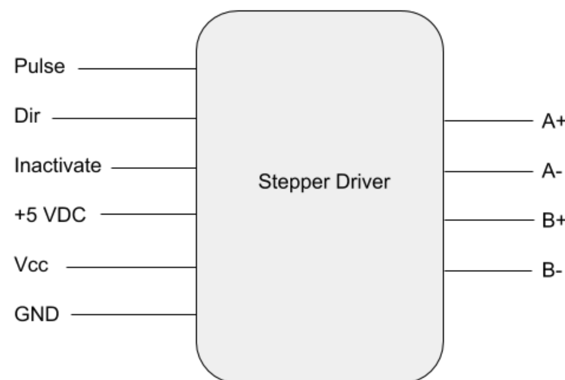


Figure 27: Stepper Driver.

6.1.3 Rotary encoder

The rotary encoder uses 5 V DC [14], which is delivered from the microcontroller, and outputs two phases A and B, (see Figure 28), and a phase Z which is not used in this project and therefore not illustrated. Both A and B send out 1024 pulses per revolution in a square wave form. By incrementing a counter with zero as initial value upon clockwise signals and decrementing it on counter clockwise pulses, the microcontroller can keep track of the position of the cart.

As described in the *Component selection* (see Chapter 3), the microcontroller operates at a lower voltage, 3.3 V in comparison to the 5 V of the rotary encoder, and can be damaged if connected directly to either of the three phases. Therefore, the voltage is lowered using a voltage divider, as shown in Figure 28.

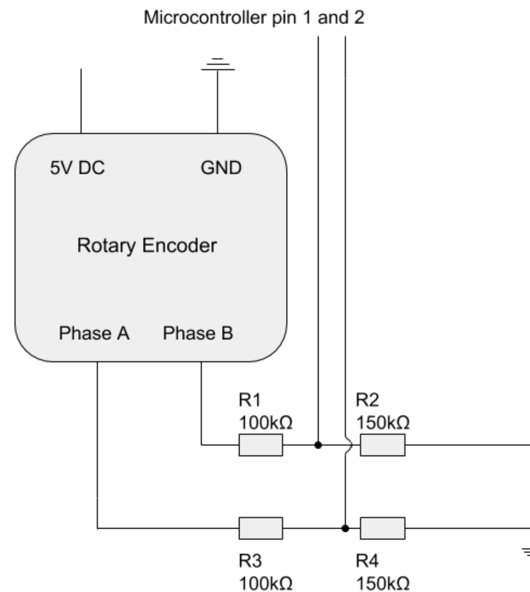


Figure 28: Rotary encoder circuit.

6.2 Software design

The software is designed with the user in mind; there is no point in building a system that cannot be understood by the general user. Simplicity and structure is also important for future project groups, should they wish to optimize or make changes to the system. Therefore, the software is designed to be as intuitive and clear as possible. One way of acheiving this is to try and mirror the physical system in the software (see Figure 29) which is easily done in an object oriented programming language such as C++, which is also advantageous, since the microprocessor runs C++.

To the largest extent possible, every physical component in the system has a software object twin. However, this is of course only true for components that play an active part in controlling and running the system; power supply units and simpler electronic components are either not included in the software system, or expressed in simpler terms than objects.

In this system, most components are one of a kind. The angular sensors are the only complex components that there are several of. This is an argument against using objects in the software, because one of the points of object orientation is to easily construct several instances of the same object. This software could perhaps be slightly more efficient if physical objects were only seen as inputs and outputs from the control loop, which is explained later. On the other hand, this would not be an intuitive and easily understood structure.

Next, every program file will be described in detail.

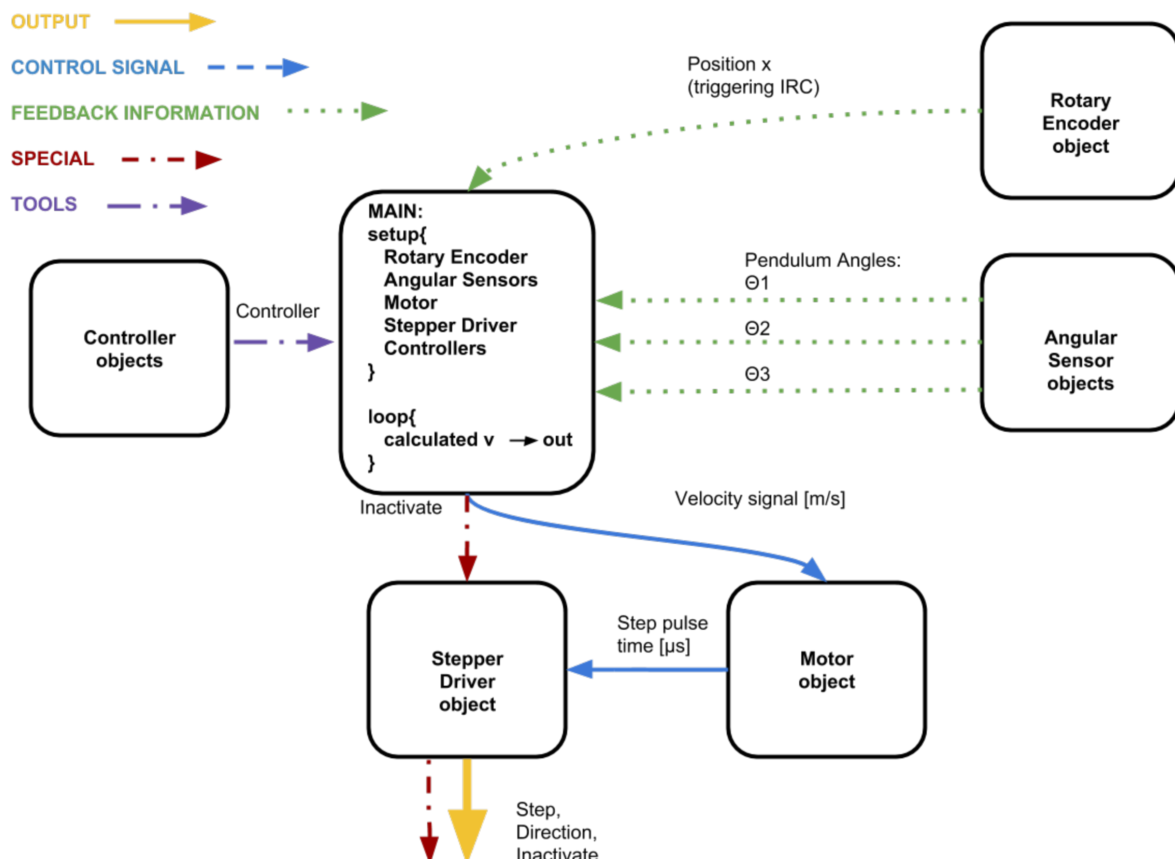


Figure 29: Software principle map.

6.2.1 The design of the software

The software, as can be seen in Appendix B, is designed to have the three states *Init*, *Calibrated*, *Running*. The system's initial state transits to the state *Calibrated* by the push of the *Calibrate button* on the panel (see Figure 30). Only when the system is calibrated the control sequence, *Running*, can start when pressing the *Start/stop switch*. However in order to make a correct calibration the cart has to be pushed as close to the electric box as possible, and the pendulums tilted in the same direction before pressing the *Calibrate button*. The reason for this is to let the system know where the boundaries of the rail are and what voltage over the angular sensors that correspond to the angle at which each pendulum is calibrated. After the system has been calibrated, the cart moves to the center of the rail.

Figure 30: Photo of the panel.

The function of the program is described in the flow chart in Figure 31. The control loop starts and runs continuously when the *Start/stop switch* is pressed from the calibrated state. It stops again when the button is pressed once more. The state can also be forced to *Calibrated* if the cart is too close to the edges of the rail as a safety precaution to protect the mechanical construction. In the state *Running* several operations are made. First, sensor data is collected from the angular sensors and the rotary encoder. The information from the rotary encoder is firstly used to determine if the cart has moved too far in one direction and if so, terminates the control sequence. If not, the sensor data is passed to the control objects which calculates the new speed of the motor. Lastly the motor speed is updated and the program jumps back to collecting sensor data. Following sections will cover how the motor is controlled and how sensor data is generated.

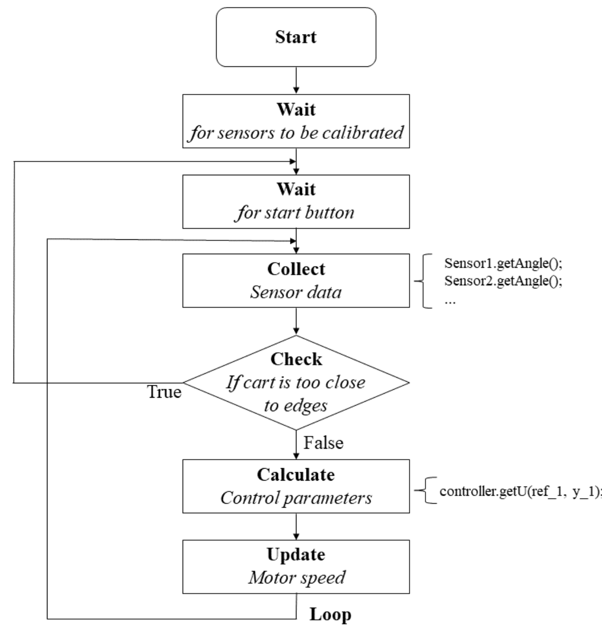


Figure 31: Flow Chart describing the function of the software.

6.2.2 Motor and stepper driver software interface

The main focus in the motor software is to create an interface between the user, who only communicates with the system via the control loop, and the system.

The motor is represented by an instance of the Motor class, and takes a velocity parameter expressed in m/s as an input parameter. It then converts this parameter to a pulse time expression which can be used by the stepper driver software. The stepper motor is designed to move a certain angle per step, not continuously rotate at a given rotational velocity. The distance the cart travels and the time it takes to perform one step is expressed as

$$\begin{cases} d_{step} = \frac{2\pi r_{pulley}}{steps/rot} \quad [m] \\ \\ t_{step} = \frac{d_{step}}{v_{cart}} \quad [m/s] \end{cases} \tag{31}$$

To make the cart travel at a specific velocity, it needs to travel a certain distance, d_{step}, in a certain amount of time, t_{step}. If the step is taken repeatedly at the rate of the step time, the resulting cart motion will correspond to the desired cart velocity, v_{cart}. The time per step value is

$$t_{step} = \frac{d_{step}}{v_{cart}} = \frac{2\pi r_{pulley}}{(steps/rot)v_{cart}} \tag{32}$$

However, there is also a need to differentiate between pulses. For a signal to go high for a specific time interval, it also needs to go low again for it to be able to repeat the cycle. The easiest way to divide this is to simply partition the cycle in two, one high part and one low (called Cooldown in Figure 32). This means that a pulse time

equals half the time of a step time. Therefore, the pulse time is calculated in the following way:

$$t_{pulse} = \frac{t_{step}}{2} = \frac{\pi r_{pulley}}{(steps/rot) v_{cart}} \tag{33}$$

In Figure 32, t_{step} is visualized as the step cycle time while the actual pulse time is described as the width of the step signal, which is equal to the cooldown time. The angle represents the rotational movement that corresponds to the described step signal.

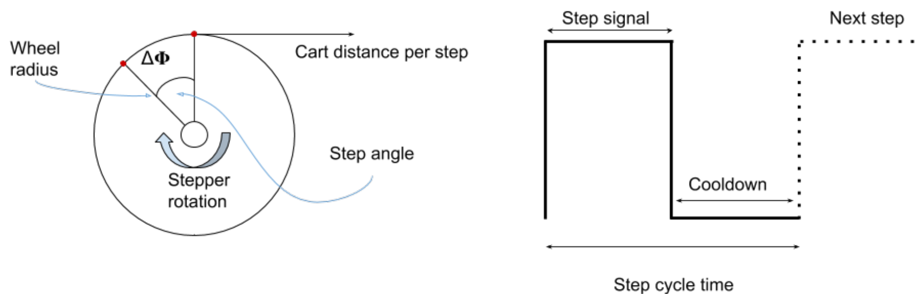


Figure 32: Step visualisation.

The software representation of the stepper driver is an object described as a class instance of the class StepperDriver, responsible for distributing the step pulses and directions to the physical stepper driver. This makes it the last part of the software communications chain.

The stepper driver object receives its pulse time and direction information from the motor object mentioned in Equation 33. When called upon from the control loop, the stepper driver object forces the microcontroller to distribute pulses to the physical stepper driver at the even time intervals of t_{pulse}.

The stepper driver object uses the internal timer of the microcontroller to determine when to send out the step pulses. The command is executed in every cycle of the control loop, but it only sends out a pulse directive if the time since the last step is equal or greater than the desired pulse time. After this, the timer is reset, setting the reference point for the next step cycle.

6.2.3 Software representation of rotary encoder

As discussed in Section 6.1.3, the rotary encoder distributes 1024 pulses per rotation, which it sends out from two separate phases that also determine in which way it is rotating. The rotary encoder always sends out these pulses, so if the system is supposed to calculate a cart position from this, every pulse needs to be accounted for.

This is done by using what is known in the Arduino library as 'attachInterrupt'. There are pins on the microcontroller that are 'interruptable', that is, pins that are programmed to make the system jump to a subroutine when given a signal.

The calculation of the position is done by comparing the two phase signals, A and B, from the rotary encoder. Only the phase A triggers interrupts, and it does this on both rising and falling flank. In Figure 33, both clockwise and counter clockwise rotation are depicted. The phase B is always offset from the phase A.

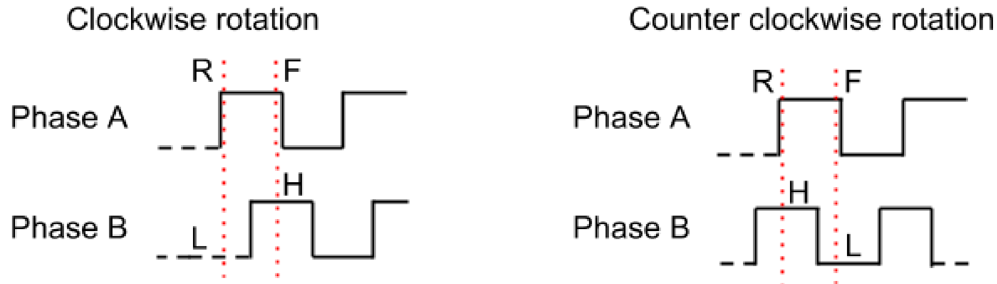


Figure 33: Rotary encoder phase pulses.

For every change in the A phase (rising, falling), there are two possible B states (high or low). Figure 33 shows that two of the four combined states of A and B are characterized as clockwise: (A rising, B low) or (A falling, B high). The opposite (A rising, B high) or (A falling, B low) are seen as a counter clockwise rotation [14]. From this, the cart position is derived in a way similar to the method in Equation 31. Counter clockwise rotation is represented by increasing the position, whereas clockwise rotation means decreasing the same variable. The position is update is accordingly

$$Pos_{cart} = \begin{cases} Pos_{cart} + \frac{2\pi r_{pulley}}{pulses\ per\ rotation} & \text{(A rising, B high) or (A falling, B low)} \\ \\ Pos_{cart} - \frac{2\pi r_{pulley}}{pulses\ per\ rotation} & \text{(A rising, B low) or (A falling, B high)} \end{cases} \tag{34}$$

There is a concern that the rotary encoder may send out too many interruptions for the processor to handle. There is currently no calculated value for how many interruptions this application can handle before the main application becomes too slow to control the pendulums. If the encoder can handle 1024 interruptions per rotation, and there are approximately eight encoder rotations for one meter that the cart travels in one direction, at a rate of for example 2 m/s, the encoder will trigger about 16500 times per second.

6.2.4 Angular sensor software

As mentioned in Section 3.2, the angular sensors are of two different kinds, one with a resistance of 5 $k\Omega$ and two with 2 $k\Omega$. In terms of programming and determining the angle of the pendulums there are no differences between the two. This is due to that it is the voltage over the angular sensors that is measured and not the resistor value. The voltage goes from 0 V to 3.3 V no matter the resistance of the sensor.

The voltage is measured by an internal ADC in the microcontroller with a resolution of 12 bits. The value that is return from the ADC is a number between 0 and 4096 ($= 2^{12}$) where 0 corresponds to 0 V and 4096 to 3.3 V.

To translate the value from the ADC, x, to degrees, x needs to be interpolated between a known angle y_0 with a corresponding voltage, x_0 and maximum angle $345° = y_1$ and maximum voltage 3.3 $V = x_1$. Any given angle $y \in [y_0, y_1]$ is then given by the following expression

$$y(x) = y_0 + (y_1 - y_0)\frac{x - x_0}{x_1 - x_0} \tag{35}$$

where x is the voltage at a given point and x_0 and y_0 are variables that have to be calibrated to obtain a correct angle.

The angular velocity is simply determined by taking the angular difference at to points and divide it by the difference in time between the two. The angle and the angular velocity are what is communicated between the Sensor object and the rest of the program.

Due to signal noise it was also of interest to be able to include digital LP-filter in the software, the structure used is

$$y[n+1] = (1 - h\omega_c)y[n] + K(h\omega_c)x[n] \tag{36}$$

where $x[n]$ is the input signal, $y[n]$ is the output, ω_c is the cutoff frequency and K is the filter gain.

6.3 Results & Discussion

When the electrical system and programming were assembled, a few adjustments from the original plan were made. The wiring from the stepper driver, marked in Figure 34, had a high enough current to generate a electromagnetic field to disturb other components. Most sensitive was the angular sensors and the rotary encoder even though they were shielded. To solve this issue the cables to these components were wired outside the electrical cabinet. The buttons on the panel (*Calibration button* and *Start/stop button*) were also severely affected by the high currents in nearby wires and was interpreted as pressed when they were not and were therefore not implemented in the program.

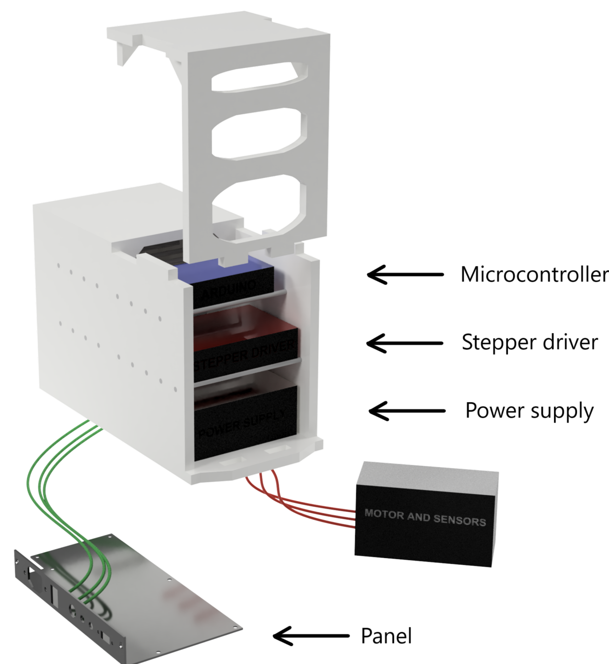


Figure 34: Electrical cabinet with wiring principle.

As the system is built the maximum performance of the angular sensors in terms of repeatability is not achieved for unknown reasons. Figure 35 represent the noise from one of the angular sensors. The noise level of the sensor is $\approx \pm 0.5°$ which is 100 times greater than specified in the data sheet of the angular sensor. To decrease the noise both an analog low pass filter and a digital low pass filter was designed to remove frequencies over 4.8 Hz, which is low enough to cut most of the noise but leave the signal representing the movement of the pendulum unaffected. The same problem existed with the other two angular sensors and solved the same way.

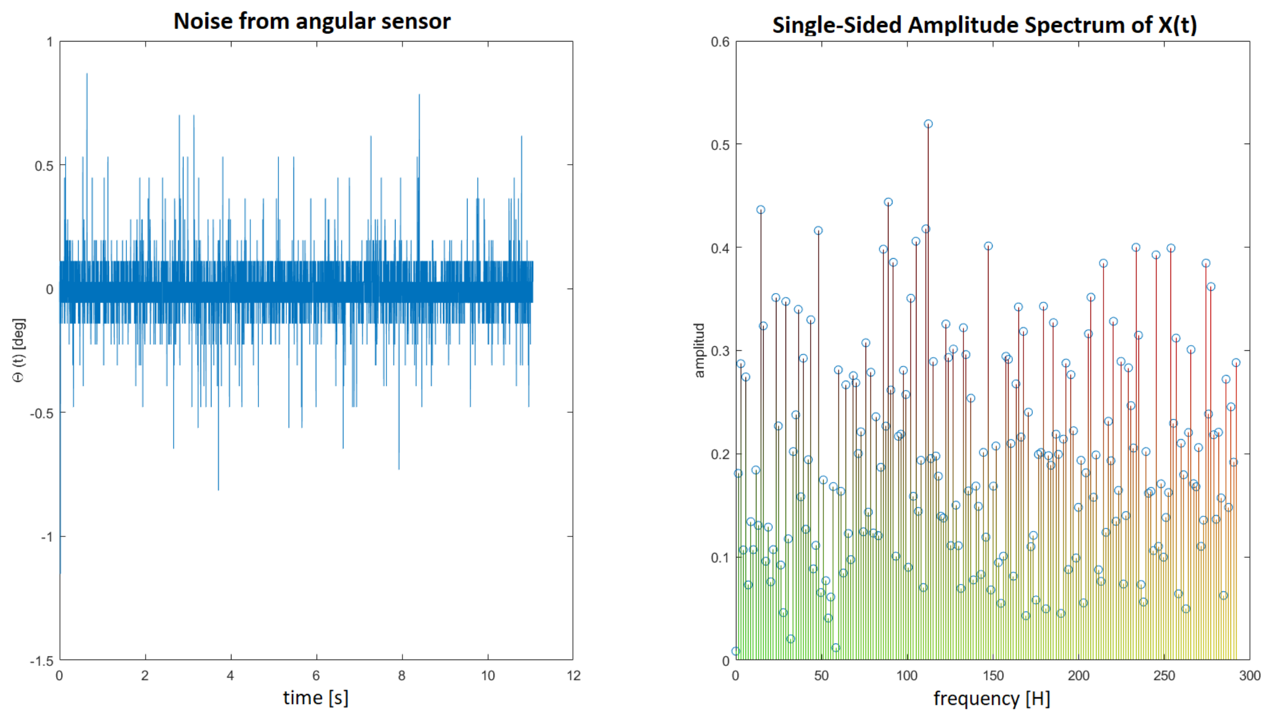


Figure 35: FFT-analysis of the signal with noise from an angular sensor.

When running the motor at the same time as reading the angular sensors, the set speed was not the same as the actual speed. The reason for this is that to rotate the motor a pulse has to be sent at a high frequency, every 78th microsecond at 1 m/s and $1/8^{th}$ in microstepping. Reading the angular sensors takes approximately 100 microseconds each [11] which means that if the time since last step is checked every cycle of the control loop, as described earlier, the speed will be severely limited. Only one pulse every 300 microseconds can be sent. Instead, a timer interrupt was implemented. In this way the speed was maintained and it was possible to plot values of the system in real time.

7

Control of System

This chapter will describe the process of choosing and designing the control system.

7.1 Simplification of system

Because some parts takes place in the Laplace domain and to ease the process of transforming the system to the Laplace domain, several assumptions are made which simplifies the equations and the system described in Section 2.

7.1.1 Stepper model

The biggest assumption is made regarding the dynamics of the stepper motor, which are completely neglected. If the load torque never exceeds the motors rated holding torque, there should never be any slip or missed steps in the motor. This assumption results in that the stepper can be described as a constant that is determined from step resolution ($\Delta\Phi$) and radius of the pulley (r).

$$G_{stepper}(s) = K_{stepper} = \Delta\Phi \frac{\pi}{180} r \tag{37}$$

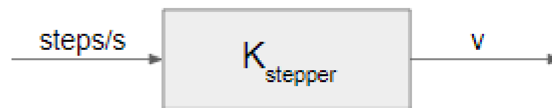


Figure 36: Stepper described as a constant gain.

To keep the controller design and simulation simple, the output of the controller is set to be the speed of the cart in [m/s].

7.1.2 Pendulum

The motion of any of the pendulums can be described by Equation 13. To be able to use Laplace transform, a linearization is done. The pendulums will operate around $\theta_i = 0$. Hence, the assumptions

$$sin(\theta_i) \approx \theta_i \tag{38}$$

and

$$cos(\theta_i) \approx 1 \tag{39}$$

are made. This results in the linearized system

$$\dot{\omega}_i = \frac{g\theta_i - \ddot{x}_c}{L_i} \,. \tag{40}$$

Taking the Laplace transform gives

$$s\Omega_i = \frac{g\Theta_i - s^2 X_c}{L_i} \,. \tag{41}$$

This equation can be rewritten to describe the process by $G_i(s)$, which is then the transfer function from the velocity of the cart V to the angle of the pendulum Θ_i. Using $\Omega_i = s\Theta_i$ and $sX = V$, we get

$$s^2\Theta_i = \frac{g\Theta_i - sV}{L_i} \Rightarrow$$

$$\Rightarrow g\Theta_i - L_i s^2 \Theta_i = sV$$

and consequently

$$G_i = \frac{\Theta_i}{V} = \frac{s}{g - L_i s^2} \,. \tag{42}$$

7.2 Classic control

This section will describe the approach taken to control the system with classical controllers such as P, PI, PD, PID. The letters stand for the three different parts of a PID-controller, i.e. the Proportional part, the Integral part and the Derivative part. The different parts treat the deviation of the system in their respective ways, either proportionally, by integrating the control error or by differentiating it. All to create different control effects.

7.2.1 One pendulum

Using the Routh-Hurwitz' stability criterium, it can be shown that the system will be stable using a PI-controller. To keep simplicity, PID-controller is not investigated.

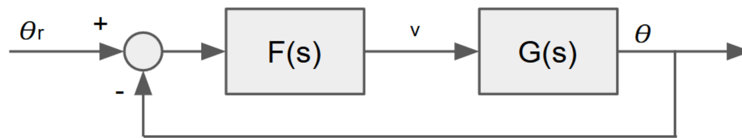


Figure 37: Control structure for one pendulum.

The open-loop transfer function is defined as:

$$L(s) = F(s)G(s) = (K_p + \frac{K_i}{s})\frac{s}{g - Ls^2} \quad (43)$$

This gives the following characteristic equation:

$$Ls^2 - K_p - (K_i + g) = 0 \quad (44)$$

Which gives the Routh-Hurwitz' matrix as:

$$\begin{bmatrix} L & -(K_i + g) \\ -K_p & 0 \\ -(K_i + g) & 0 \end{bmatrix}$$

That gives the stability margins for K_p and K_i as:

$$-K_p > 0 \Rightarrow K_p < 0 \quad -(K_i + g) > 0 \Rightarrow K_i < -g$$

Running the simplified simulation with a pendulum of length 1 m and $K_p = -20$, $K_i = -20$ gives the following response of the pendulum.

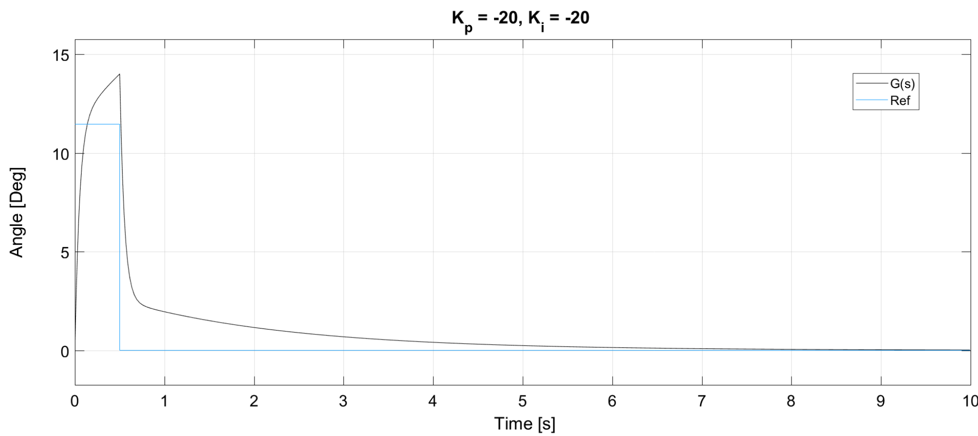


Figure 38: Control of a single pendulum using PI-controller.

7.2.2 Two pendulums

To control two pendulums a cascaded structure is used, where the second controller controls the inner loop's reference signal.

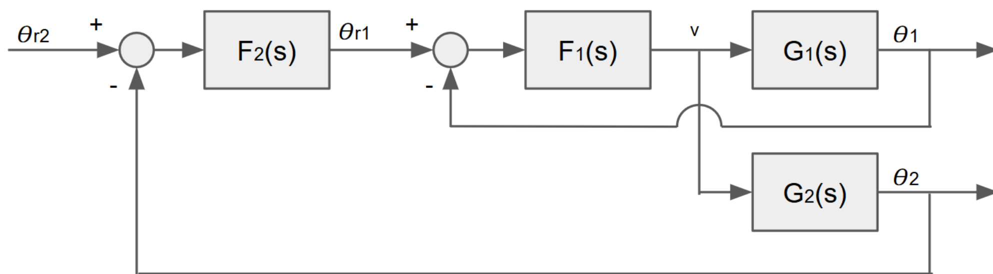


Figure 39: Control structure for two pendulums.

Ignoring the coupling between the two pendulums through the cart the inner loop's transfer function is

$$G_{loop1} = \frac{V}{\Theta_{r1}} = \frac{F_1(s)}{(1 + G_1(s)F_1(s))} .$$

Simplifying the block scheme using G_{loop1} gives the block scheme in Figure 40.

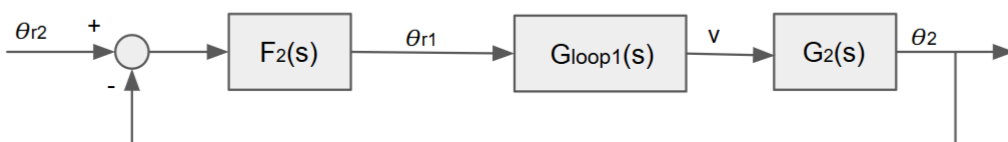


Figure 40: Simplified control structure for two pendulums.

Instead of analytically determine the parameters of the second loop, an attempt was made to determine them numerically in MATLAB, since the magnitude of the equations escalates when trying to apply the Routh-Hurwitz' criterium to the outer loop with a PID controller. However no stable combination of parameters were found. In Figure 41 one of the better results is presented and as can be seen, the system starts to stabilise but cannot keep the pendulums balanced for long.

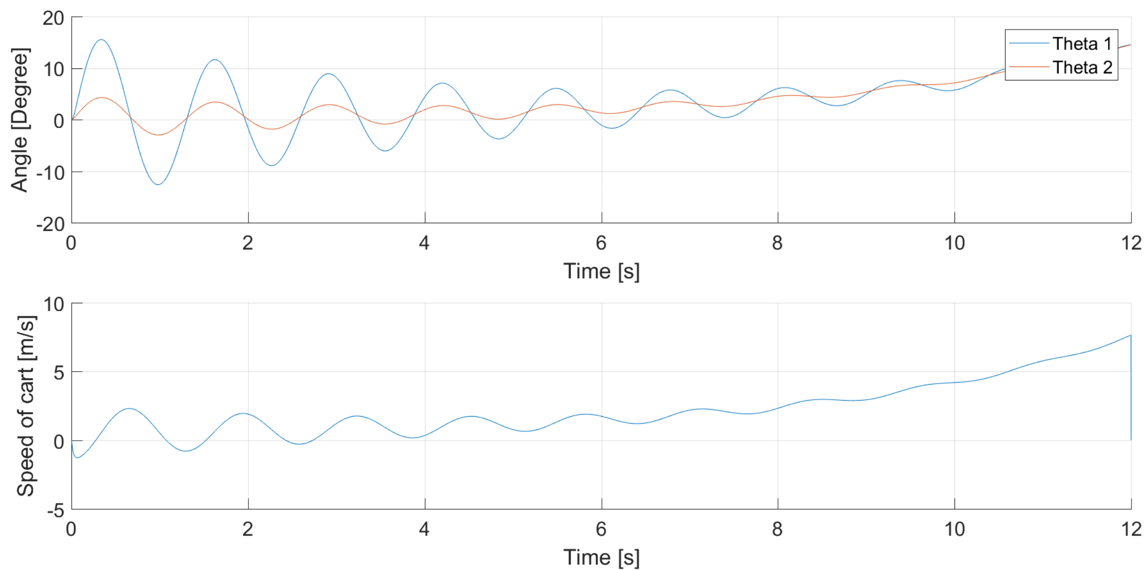


Figure 41: Simulation with numerically determined parameters for the second loop.

It may, however, still be possible to control the system this way, but it will require more advanced analysis of the system, which was not conducted in this project. Instead the possibilities of state feedback control was investigated.

7.3 State feedback control

This section shows the process of finding a state-feedback controller suited to balance the pendulums.

7.3.1 State-space representation of the system

The variables that need to be kept track of are the pendulum angles θ_i, their derivatives $\dot{\theta}_i$, and the cart position x_c. The first variable for the state vector is θ_i. The second variable is not only $\dot{\theta}_i$. Instead, given that the derivative of the angle velocity $\dot{\theta}_i$ includes the cart acceleration $\ddot{x}$ (see Equation 40) minor modifications had to be made. Consider that the control signal is the cart velocity:

$$u(t) = v(t) \tag{45}$$

Then, a state variable which includes the angle velocity that is also appropriate for this state-space model is $L\theta_i + v$. Furthermore, if position control is included, the state variable x_c is needed as well.

So to summarize the state vector:
A unique set of the first two variables is needed for each pendulum and on top of this the last state variable is the position x_c.

For example, the state-space model for a system with one pendulum and without position control is

$$\dot{x}(t) = Ax(t) + Bu(t) = \begin{bmatrix} 0 & 1/L \\ g & 0 \end{bmatrix} x(t) + \begin{bmatrix} -1/L \\ 0 \end{bmatrix} u(t) \tag{46}$$

where the state vector is

$$x = \begin{bmatrix} \theta \\ L\dot{\theta} + v \end{bmatrix} \tag{47}$$

If instead, one would like to control three pendulums with position control, the state-space model becomes

$$\dot{x}(t) = Ax(t)+Bu(t) = \begin{bmatrix} 0 & 1/L_1 & 0 & 0 & 0 & 0 & 0 \\ g & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/L_2 & 0 & 0 & 0 \\ 0 & 0 & g & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1/L_3 & 0 \\ 0 & 0 & 0 & 0 & g & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} x(t) + \begin{bmatrix} -1/L_1 \\ 0 \\ -1/L_2 \\ 0 \\ -1/L_3 \\ 0 \\ 1 \end{bmatrix} u(t) \quad (48)$$

where

$$x = \begin{bmatrix} \theta_1 \\ L_1\dot{\theta}_1 + v \\ \theta_2 \\ L_2\dot{\theta}_2 + v \\ \theta_3 \\ L_3\dot{\theta}_3 + v \\ x_c \end{bmatrix} \quad (49)$$

Both systems were verified to be controllable by calculating the ranks of their controllability matrices Ç, i.e. verifying that:

$$rank(ç) = n \quad (50)$$

where A is an $n \times n$ matrix, and the controllability matrix

$$ç = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix} \quad (51)$$

7.3.2 Linear-quadratic regulator

This method requires a linear or linearized system and has a quadratic cost function. The cost function is minimized by the feedback control law

$$u(t) = -Kx(t) \quad (52)$$

where $x(t)$ is the state vector.
To find the feedback gain K, two other design matrices needs to be defined; Q_x and Q_u. Q_x is on the form:

$$Q_x = \begin{bmatrix} q_{x1} & \dots & \dots & 0 \\ \vdots & q_{x2} & & \vdots \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & q_{xn} \end{bmatrix}$$

q_{x1} defines the cost of variations in x_1, q_{x2} the cost of variations in x_2 and so forth. Q_u works in the same way but for the control signal.

To get the feedback vector K for the chosen Q-matrices the MATLAB-command: K=lqr(A,B,Qx,Qu) was used. The resulting gain then minimizes the cost function

$$J = \int_0^\infty (x^T Q_x x + u^T Q_u u) dt \,. \tag{53}$$

7.3.3 One pendulum

The system was simulated in Simulink with the specifications given in Table 12. Note that the weights for the different variables are equal in order to keep it simple.

Table 12: Parameter values, one pendulum.

Parameter	Value
q_{x1}	1
q_{x2}	1
Q_u	1
L	0.4 m
$K_{stepper}$	$2\pi\ 10^{-4}$

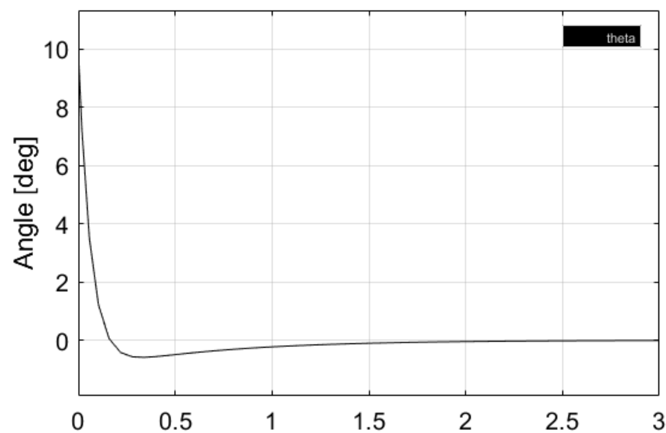


Figure 42: θ, one pendulum.

The resulting angle θ in Figure 42 shows that the pendulum is balanced from a 10° starting position without complications.

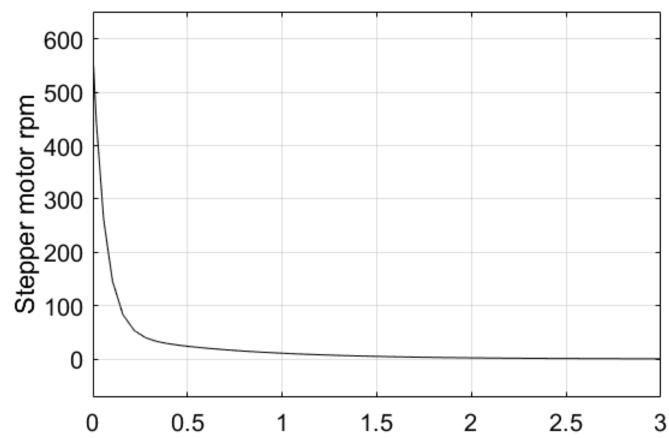


Figure 43: Motor speed, one pendulum.

The speed of the motor, (see Figure 43), does not exceed ~ 600 rpm which is well below the specified value for the motor and, therefore, the simulation shows promise for the control of one pendulum.

7.3.4 Three pendulums with positioning

Simulating three pendulums with parameter values in Table 13.

Table 13: Parameter values in simulation of three pendulums and positioning control activated.

Parameter	Value
q_{x1}	1000
q_{x2}	1
q_{x3}	1
q_{x4}	1
q_{x5}	1
q_{x6}	1
q_{x7}	10
Q_u	100
L_1	0.4 m
L_2	1 m
L_3	1.5 m
$K_{stepper}$	$2\pi\ 10^{-4}$

In order to minimize motor speed, pendulum angles, and cart position their respective weights q_{x1}, Q_u, and q_{x7} were increased until the further improvements were minor.

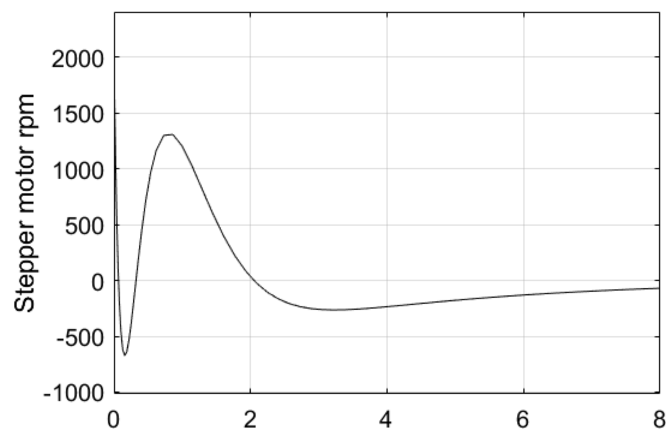


Figure 44: Motor speed, three pendulums and positioning control activated.

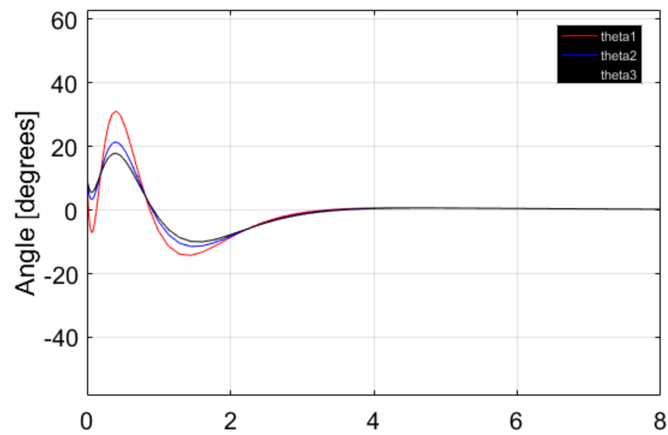


Figure 45: Pendulum angles, three pendulums and positioning control activated.

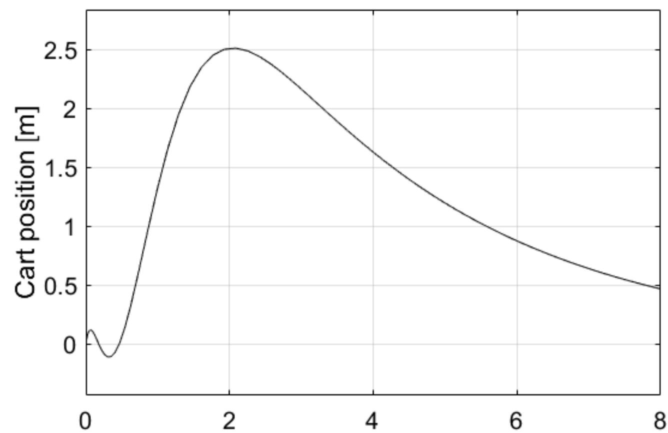


Figure 46: Cart position, three pendulums and positioning control activated.

To compare the results with the simulation of one pendulum using an LQ-regulator, the starting angles were set to 10° here as well. The results show that balancing three pendulums from a certain angle requires a lot more from the system (see figures 44-46). This with regard to motor speed, longer path of the cart, and time to converge to upright position.

8

Results & Discussion

This chapter will discuss the final system, reflect on its performance and its functionality as well as what was successfully implemented and what was not.

8.1 Physical system

The final physical system that was built is presented in Figure 47. The firmware of the Arduino DUE have made it possible to: control the motor, use the encoder as position measurement and use the potentiometers as angular sensors. Thereby all the basic functionality of the system has been fulfilled. More detailed specifications and drawings can be found in Appendix A.

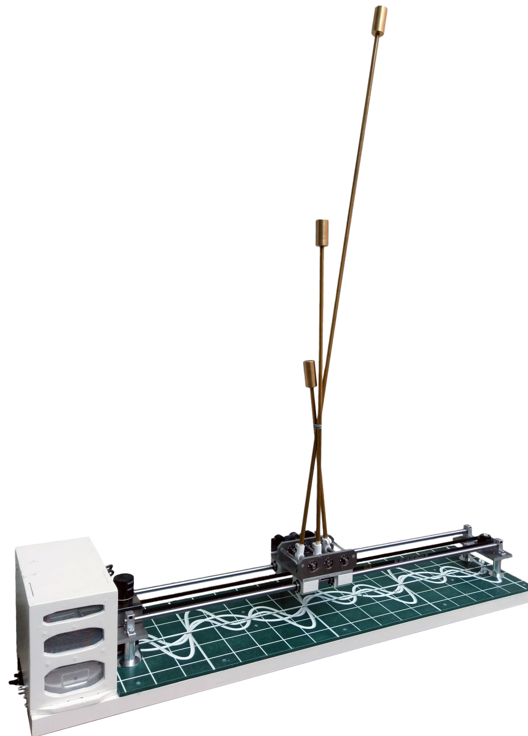


Figure 47: Final physical system.

8.2 Balancing one pendulum using LQR

One pendulum was successfully balanced with an LQR controller implemented. It uses a similar state-space model as in Section 7.3.3. However, the position was added to achieve positioning control, i.e. the following state space model was used:

$$\frac{d}{dt}\begin{bmatrix}\theta \\ (L\dot{\theta}+v) \\ x_c\end{bmatrix} = \begin{bmatrix}0 & 1/L & 0 \\ g & 0 & 0 \\ 0 & 0 & 0\end{bmatrix} x + \begin{bmatrix}-1/L \\ 0 \\ 1\end{bmatrix} u \tag{54}$$

Using the following Q-matrices:

$$Q_x = \begin{bmatrix}1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 10000\end{bmatrix} \quad \text{and} \quad Q_u = 1000000 \tag{55}$$

results in the following feedback gain:

$$K = \begin{bmatrix}-4.887 \\ -2.049 \\ -0.100\end{bmatrix} \tag{56}$$

Due to signal noise in the measurment signals, severe filtering was necessary. A physical LP-filter was therefore created and several internal digital LP-filters were also created in the software to filter the input signals. Another LP-filter was applied to the control signal to smoothen the dynamics and avoid slipping of the stepper motor. The successful control structure for one pendulum is presented in Figure 48.

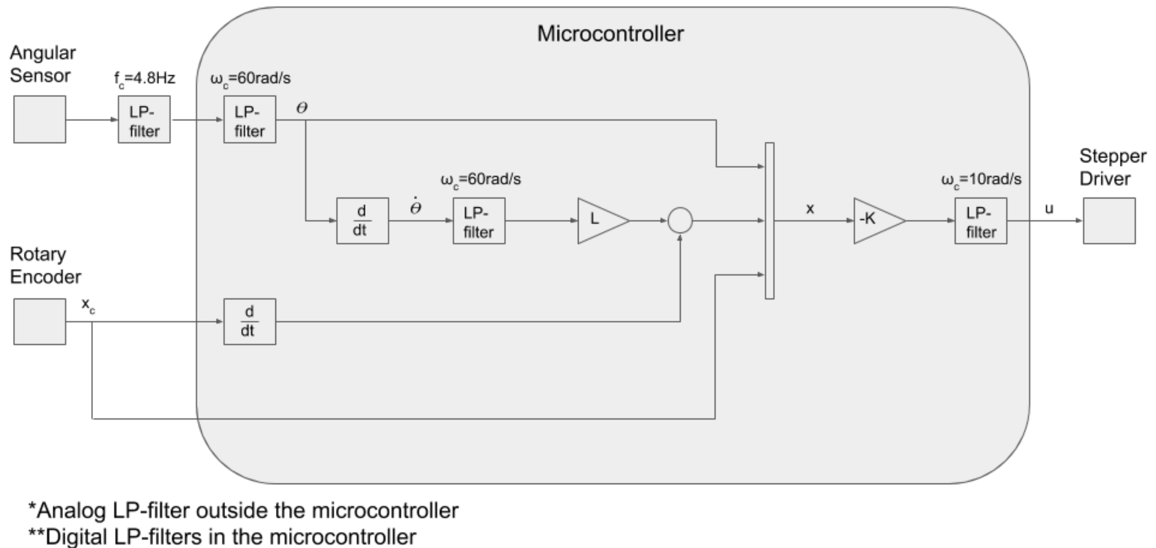


Figure 48: Control design for one pendulum.

8.3 Model validation

To verify the simulation and the model, an experiment was conducted where a given input signal was given to both the physical system and the simulated. The input signal is presented in Figure 49. Parameters measured from the real system and used in the simulation are presented in Table 14. Note that some parameters has been changed from Table 9.

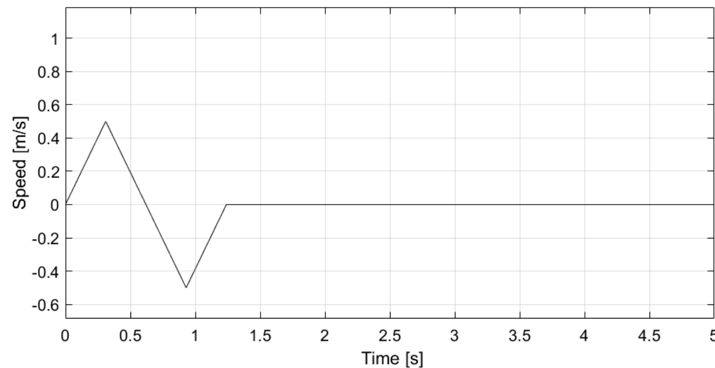


Figure 49: Input signal used for verification.

Table 14: Parameter list for simulation verification.

Parameter	**Variable**	**Value**
No. of phases	m_s	2
No. of rotor pole pairs	n_s	50
Phase voltage	U	75 V
Rotor inertia	J	300 $gcm^2 = 0.3 \cdot 10^{-4} kgm^3$
Phase resistance	R	0.9 Ω
Phase inductance	L_s	2.5 mH
Damping constant stepper	D	0.3 Nm/(rad/s)
Pendulum length	L	0.35 m
Pendulum mass	m	0.3 kg
Pendulum friction constant	k_θ	0.2 Nm/(rad/s)
Cart mass	M	1.6 kg
Cart friction constant	k_v	0.1 N/(m/s)
Radius of belt pulley	r	0.02 m

It is worth noticing that due to problems within the simulation not every parameter matches what was used in the real system. The phase voltage was 40 V and the friction constants had to be changed to avoid problems with the simulated stepper motor.

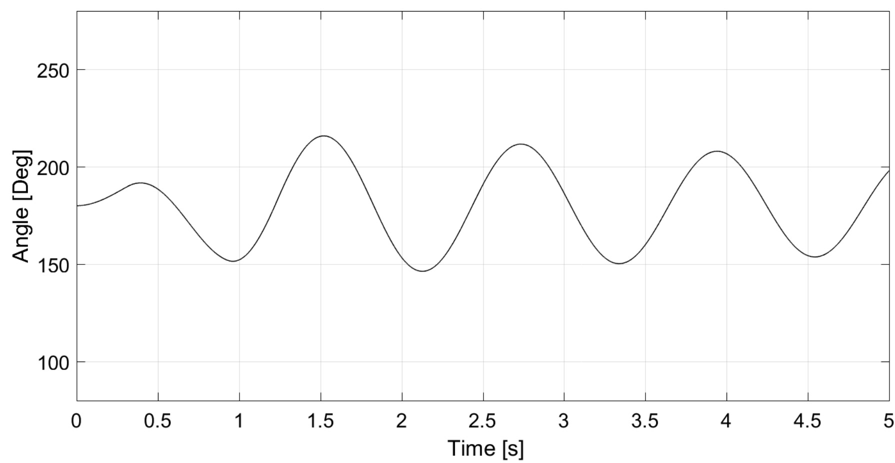


Figure 50: Angle of pendulum in simulation.

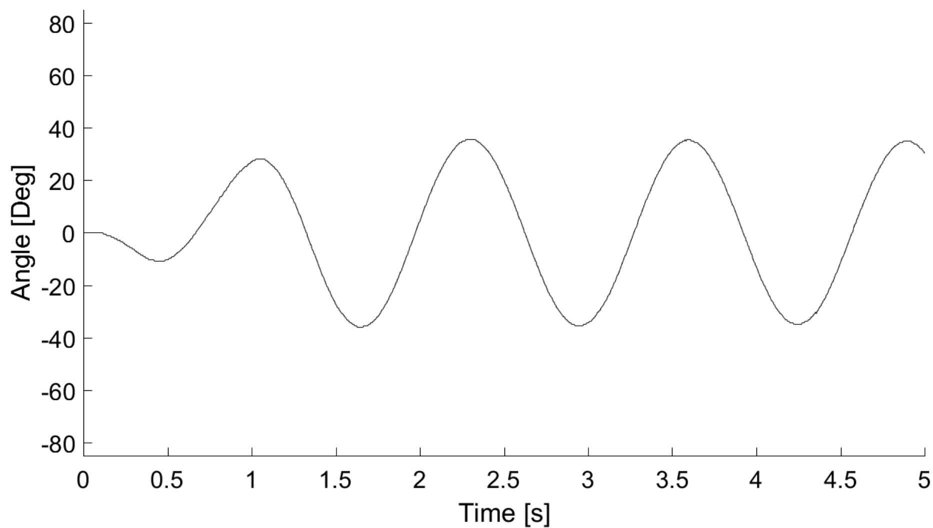


Figure 51: Angle of pendulum in real system.

It is important to note that both the simulated (Figure 50) and the real system (Figure 51) plot of the pendulum angle are initiated at 180° but the real system was calibrated so it starts from 0°. As can be seen, both has an amplitude of approximately 40°, however their frequencies does not match and neither does the initial behaviour. Furthermore the simulated system seems to have a larger friction constant than the real system since its amplitude decays faster. The major differences are probably cased by a bad approximation of the pendulum length and friction constant.

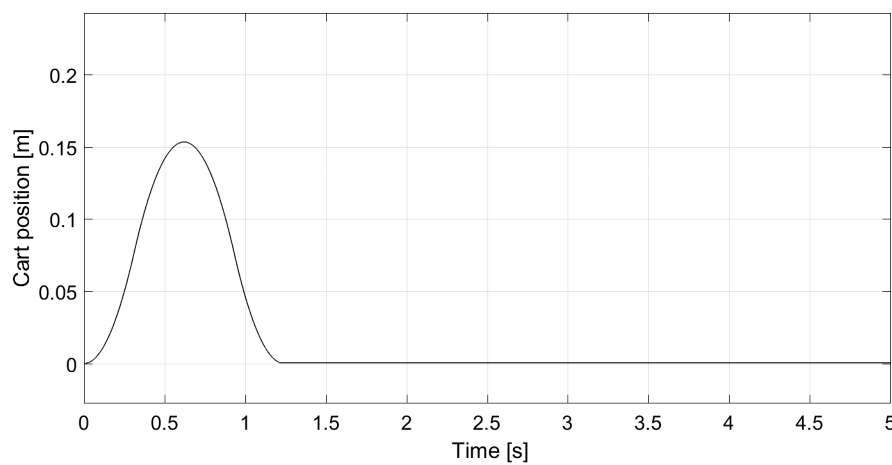


Figure 52: Position of cart in simulation.

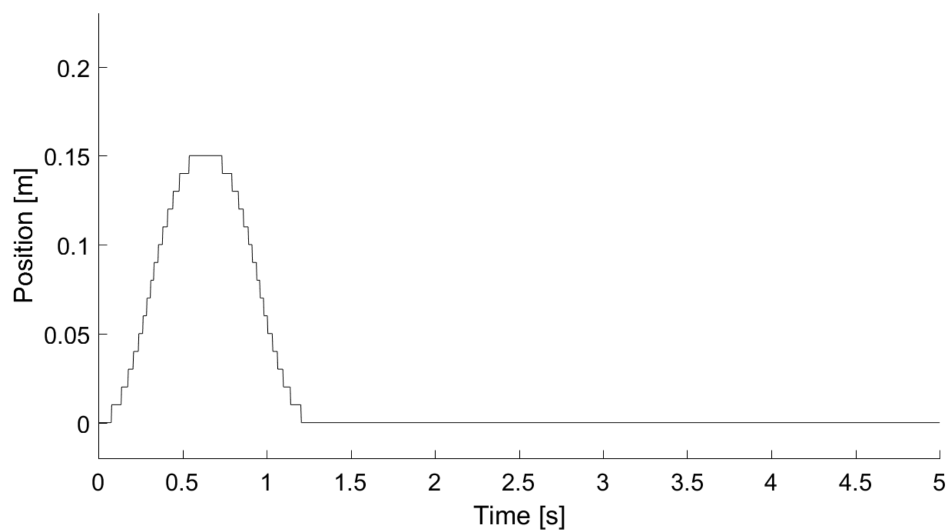


Figure 53: Position of cart in real system.

The position plots from the simulated (see Figure 52) and real (see Figure 53) systems seem to match pretty well. Both have the same shape and approximately the same maximum $0.15m$. The main difference is the discrete characteristics of the real system due to the rotary encoder not being included in the simulation. Except for the rotary encoder the model is considered accurate enough in this aspect.

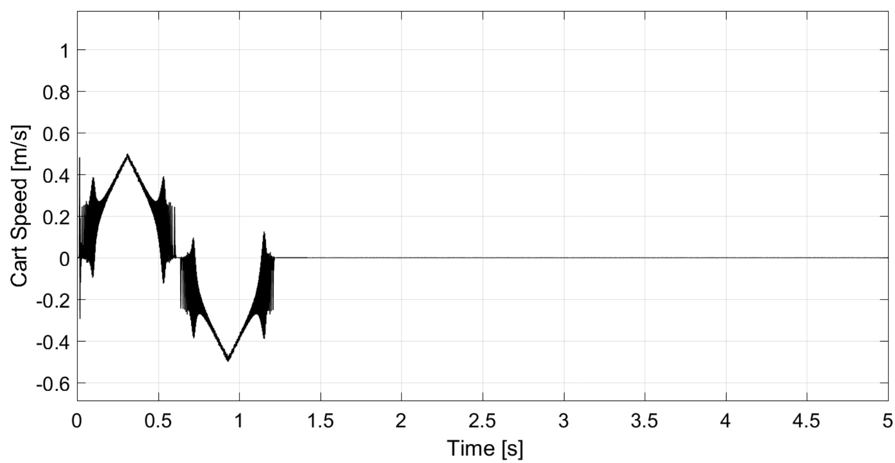


Figure 54: Speed of cart in simulation.

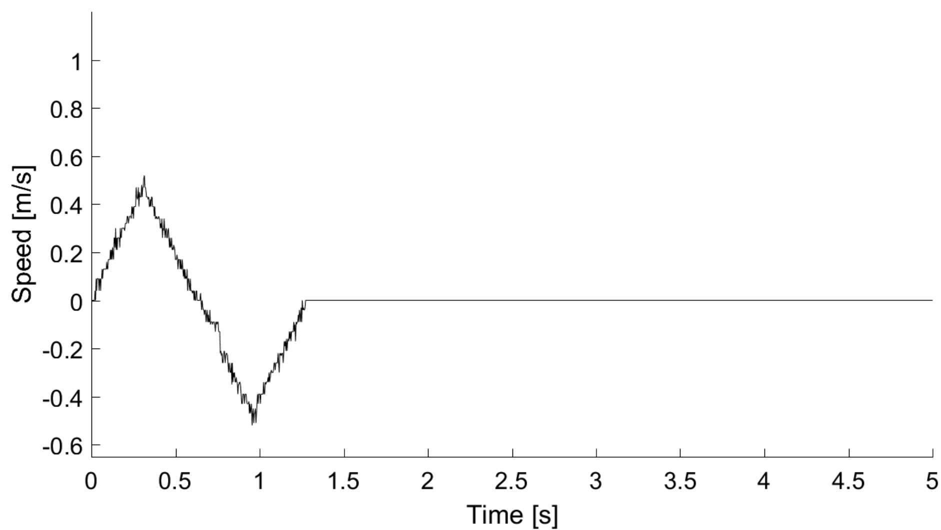


Figure 55: Speed of cart in real system.

The speed plots from the simulated (see Figure 54) and real (see Figure 55) systems seems at first very different, however the overall shape and amplitude matches. The plot from the real system has more noise, which is likely to be caused by the differentiation of the discrete positioning signal. The noise in the simulated system at lower speeds is the main difference and could possibly be caused by not including the timing belt's properties in the model, which probably would have cancelled out some of the vibrations. For this aspect, the accuracy of the model is not considered accurate, mainly because of the large spikes in the simulated system.

Concluding the verification, the simulation is in its current state not to be considered trustworthy, mainly because of problems when the stepper motor is simulated at greater speeds. Since time was limited there has not been much work done on debugging and refining the simulation.

8.4 Future work

This section will mainly discuss possible future work of the system and what remains to be implemented and analyzed to achieve the project's original goals.

8.4.1 Modeling and simulation

The simulation of the model needs general refinement and validation. Some of these issues are:

- Verify stepper model. It would be beneficial to do some verification tests for the stepper motor alone, mainly to be able to better approximate the parameters not given in the datasheet and to verify the expected behaviour.

- Implement a better stepper driver in the simulation. As described in Section 4.1.5 the driver used in the simulation increases the number of pole pairs in the rotor instead of changing the voltage sequence for the phases. A better choice could be to implement a driver that acts like the real system driver, that generates similar voltage sequences as those presented in Section 4.1.5.

- Include timing belt characteristics in model. The large resonances in the speed curves are thought to be caused by the lack of vibration dampening in the model. In the real system this would mainly be done by the timing belt so it would seem beneficial to include it in the simulation.

- Change friction model for cart. In this model the friction for the linear bearings is assumed to be linear to the speed of the cart and any static friction was completely neglected. It was found that the static friction possibly was the biggest contribution to friction in the linear bearings, as the dynamic friction was much lower than the static.

8.4.2 Electrical construction

Possible future changes to the electrical wiring configuration include:

- Better isolation and shielding of wires from sensors and switches to the microcontroller. The impact that high current conductors has on these connections is obvious and should have been taken into consideration already from the start.

- Establishing maximum distance between wires of high and low current. An example of this would be placing the power supply, stepper motor and stepper driver on one side of the machine and wire the sensors from the other side, with the microcontroller in the center. This way, the risk of wires interfering with each other would be minimized.

- Applying well planned filters such as the low pass filters, both digital and analog, on the angular sensors in order to reduce disturbances.

8.4.3 Control

As previously mentioned, only one pendulum was successfully controlled, due to lack of time it was not possible to control multiple pendulums. Possible measures to take into consideration are:

- A Kalman filter could be implemented to mitigate the effects of disturbances on the calculated state variable $L\dot{\theta}+v$. Both $\dot{\theta}$ and v are affected by this since both variables are discrete derivatives of sensor input signals.

- Change to a DC-motor. Since the state-space model used to describe the inverted pendulum system were based on velocity as the input, as opposed to acceleration, a ramp function had to be made in the software. Without this ramp function the cart had lower maximum speeds. Due to the fact that this software was not part of the state-space model it could not be simulated beforehand, thereby worsening the agreement between simulation and reality. Meanwhile, by modeling a DC-motor it is possible to see the spectrum of input signals, i.e. currents, that are possible already in the simulation stage. What this means is that some simulations of balancing the inverted pendulums using an LQ-regulator makes it look like the LQ-regulator could actually balance the actual system. Still, the control signal given by this regulator might be too high for the motor, causing it to slip. So if the DC-motor option could prove to be viable it would take care of these problems.

9
Conclusion

Concluding the project while reviewing the original goals set in Section 1.2, some of them have been achieved and unfortunately, some have not. Regarding the mechanics all the basic functionality works and as described in Section 8.1 and Chapter 5 the cart can be programmed to move and all input sensors can be read through software.

The user interface goal was partly reached. It is possible to change controller parameters trough the code but in no other way. However, the system is ready to be used for demonstration purposes in its current state if the user has knowledge about this field.

Regarding the control the simplified state-space simulations in Chapter 7 were successful, where an LQR controller was able to balance three pendulums simultaneously with position control. However, this was never successfully realised in the real system or in the simulation developed in Chapter 4 during the time frame of the project.

The investigation of controllers to be used to control the system will hopefully shed some light on the control problem of multiple inverted pendulums. The investigation has, however, not been as substantial as initially intended but did nonetheless show that for this system there was no success in controlling it using *classic control* as explained in Chapter 7, where it was found that it was at best only possible to control a single pendulum using a PI-controller, but not multiple pendulums.

Unfortunately, the properties of the system have not been investigated to the extent originally desired. For example, the questions "how much must the pendulum lengths differ to be able to control the system?" and "what effect does the choice of system parameters have on the component selection" has not been fully investigated due to the limited time frame of the project.

As a final word, the entire project can be seen as a semi-success. A mechanically complete system with all the basic functionality has been created which could act as a possible platform to build upon for future work, which can investigate further issues. All software functionality have been successfully implemented and it was shown by simulation that it is indeed possible to control three pendulums simultaneously on the same cart using an LQR controller. Also, one pendulum was successfully balanced with an LQR controller.

Bibliography

[1] TRINAMIC Motion Control GmbH & Co. KG, "QMOT QSH5718 MANUAL," 2011, available at https://www.trinamic.com/fileadmin/assets/Products/Motors_Documents/QSH5718_manual.pdf.

[2] B. Lennartsson, *Reglerteknikens grunder.* Reading, Massachusetts: Studentlitteratur, 2000.

[3] S. Bennett, *A history of control engineering 1930-1955.* Hertfordshire, UK: Peter Peregrinus Hitchin, 1993.

[4] A. Procházka, J. Uhlíř, P. W. J. Rayner, and N. Kingsbury, *Signal Analysis and Prediction.* Reading, Massachusetts: Birkhäuser Basel, 1998.

[5] Y. C. Lee, M. R. Gore, and C. C. Ross, "Stability and Control of Liquid Propellant Rocket Systems," *Journal of the American Rocket Society*, vol. 23, no. 2, pp. 75–81, Mar-Apr 1953.

[6] J. Klintberg, "Inverterad dubbelpendel - Design och simulering av regleralgoritmer," Bachelor's thesis, Chalmers University of Technology, 2016, Department of Signals and Systems.

[7] B. Carlsson and P. Örbäck, "Mobile inverted pendulum - Control of an unstable process using open source real-time operating system," Master's thesis, Chalmers University of Technology, 2009, Department of Signals and Systems.

[8] D. Wall, "Modeling and control of a three dimensional inverted pendulum," Master's thesis, Chalmers University of Technology, 2017, Department of Electrical Engineering.

[9] E. Aranda-Escolástico, M. Guinaldo, F. Gordillo, and S. Dormido, "A novel approach to periodic event-triggered control: Design and application to the inverted pendulum," *ISA Transactions*, vol. 65, pp. 327–338, Nov 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0019057816301811

[10] A. Morar, "Stepper Motor Model for Dynamic Simulation," *ACTA ELECTROTEHNICA*, vol. 44, pp. 117–122, 2003, available at https://ie.utcluj.ro/files/acta/2003/Number2/Paper08{_}Morar.pdf.

[11] Arduino , "Arduino DUE Tech Specs," 2018, available at https://store.arduino.cc/usa/arduino-due.

[12] Atmel, "SAM3X/SAM3A Series Datasheet," 2015, available at http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11057-32-bit-Cortex-M3-Microcontroller-SAM3X-SAM3A_Datasheet.pdf.

[13] R. Söderström Olsson, J. Warnborg, and F. Zeidler, "Inverterad dubbelpendel - Modellering, reglering och balansering av en inverterad dubbelpendel," Bach-

elor's thesis, Chalmers University of Technology, 2016, Department of Signals and Systems.
[14] YUMO, "YUMO E6B2-CWZ3E Datasheet," available at https://cdn.sparkfun.com/datasheets/Robotics/E6B2Encoders.pdf.
[15] JBCNC, "3128S Information," available at https://www.jbcnc.se/en/stepper-drivers-c-34/digital-stepperdriver-10-40v-3a-p-583.
[16] Mouser Electronics, "S-350-36 Datasheet," available at https://www.mouser.com/ds/2/260/s-350-spec-1179912.pdf.
[17] SKF Group, "Linear bearings and units," pp. 1–60, 2011, available at http://www.skf.com/binary/49-55843/SKF_6402_EN_11_10_19_GB_high.pdf.

A

Drawings

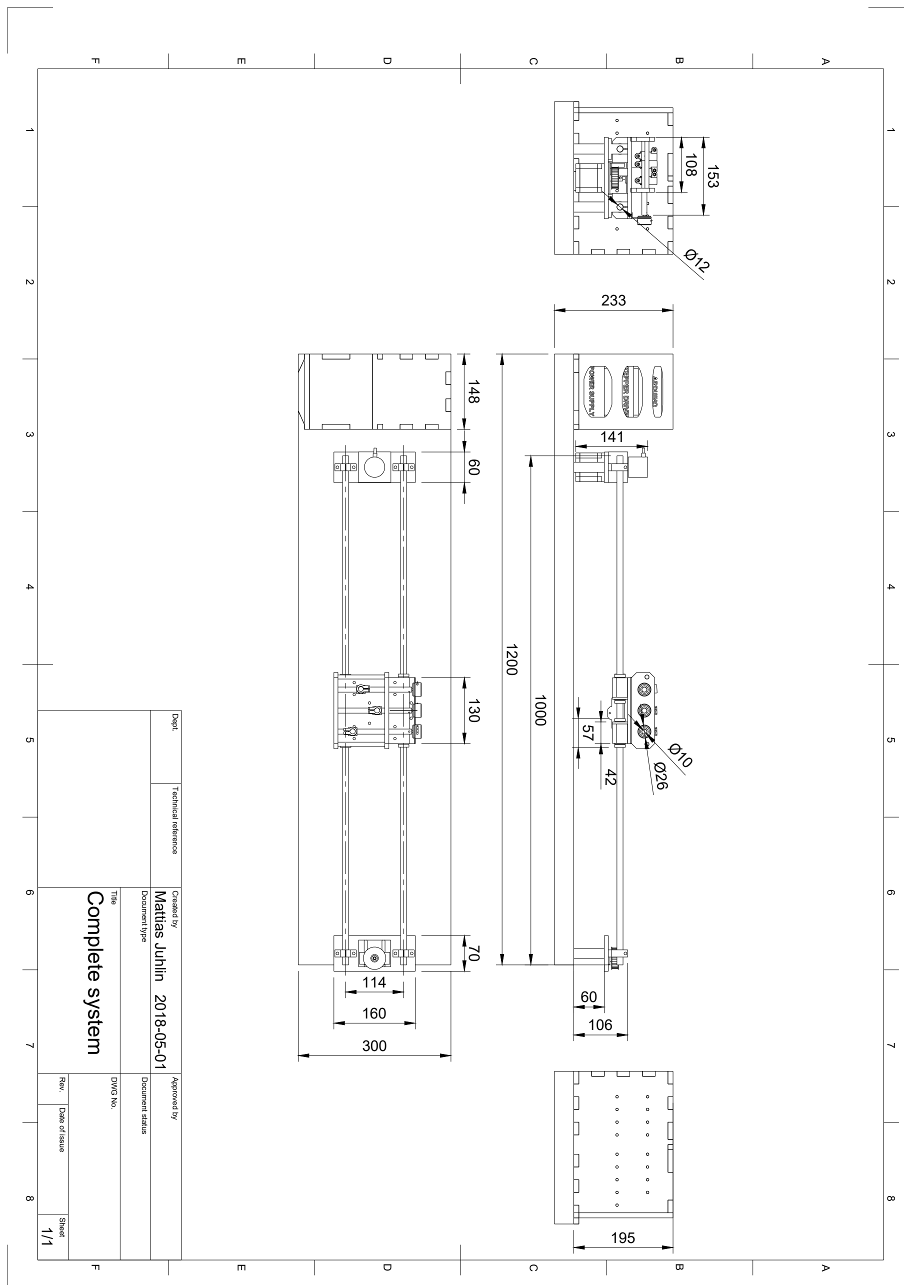


Figure 56: Drawing of the complete system.

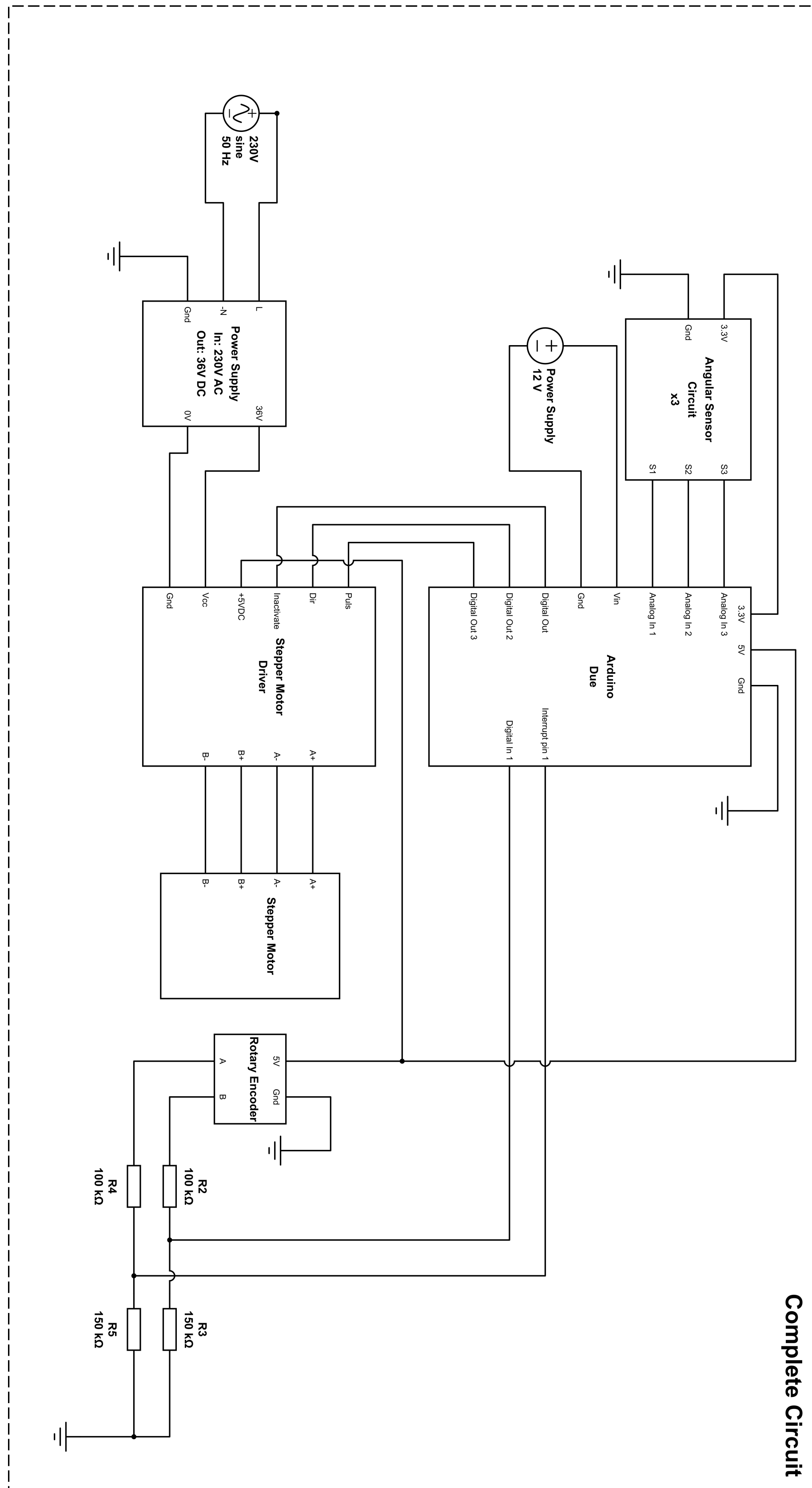


Figure 57: Wiring diagram of the complete electrical system.

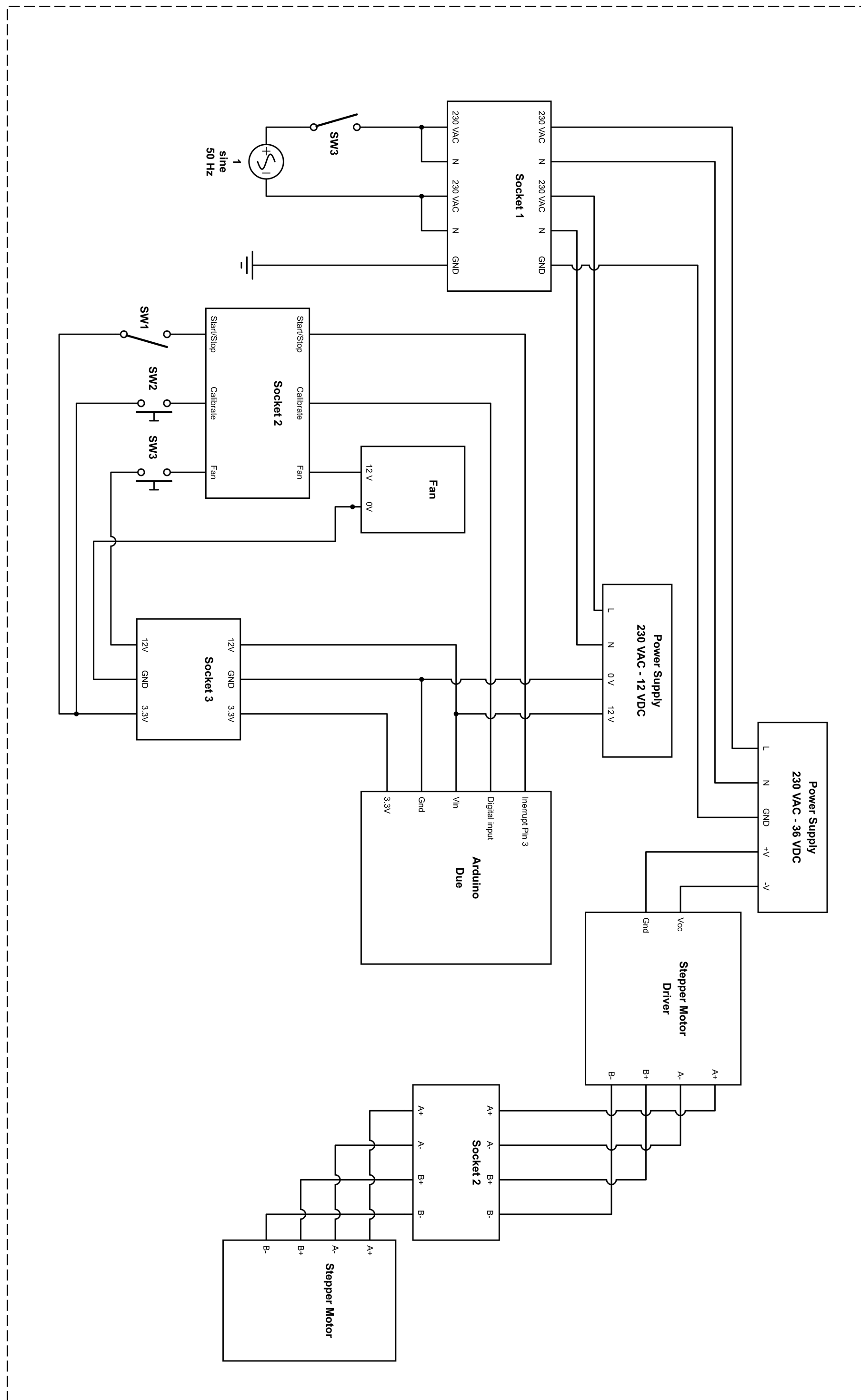


Figure 58: Wiring diagram of the circuits under the electrical cabinet.

B

Code

Following files are included:

- main.ino
- IOConfig.h
- LGR.h
- LQR.cpp
- rotaryEncoder.h
- rotaryEncoder.cpp
- sensor.h
- sensor.cpp
- DueTimer.h
- DueTimer.cpp

```
1 /*main.ino*/
2 /*       Include files       */
3 #include "sensor.h"
4 #include "IOConfig.h"
5 #include "rotaryEncoder.h"
6 #include "LQR.h"
7 #include "DueTimer.h"
8
9 /*
10 pin7 Port C 23
11 pin6 Port C 24
12
13 PIO_SODR
14 PIO_CODR
15 */
16
17 #define factor 6.3611*pow(10,-5)
18
19 #define S_TO_MS 1000000
20 #define PORT PIOC
21
22 #define DIR_PIN 5
23 #define STEP_PIN 7
24
25 #define MAX_PERIOD 8388480
26 #define PULLY_RADIUS 0.02
27 #define STEP_RES 1.8
28 #define NO_OF_MICRO_STEPS 8
29
30 #define MAX_ROTATION 345    //degrees
31 #define CAL_ANGLE_DEG 10    //degrees
32 #define MAX_VOLTAGE 3.3     //Volts
33
34 #define MAX_POS 100
35 #define MIN_POS 0
36
37 #define Kp -7
38 #define Ki -15
39
40 #define Kp2 0.08
41 #define Ki2 0.3
42
43
44
45 #define K1 -4.8873
46 #define K2 -2.0491
47 #define K3 -0.1000
48
49 /*
50 #define K1 -4.8080
51 #define K2 -2.0159
52 #define K3 -0.0316
53 */
54 long startTime = 0;
55 bool stepActive = false;
56 bool timerStopped = false;
57
58 double lastU = 0;
59 double lastXR = 0;
60 long lastCall = 0;
61
62 double i_part = 0;
63 double i_part2 = 0;
64
65 /*       Declaring Controllers       */
66 LQR sture = LQR(K1,K2,K3);
67
68 /*       Declaring Sensors       */
69 Sensor left_sensor(1,345,5,0);
70
71 /*       Declaring Stop/Start       */
72 bool go = false;
73
74 /*       Declaring Rotary Encoder       */
75 rotaryEncoder cart = rotaryEncoder();
```

```
76
77 void setup() {
78     Serial.begin(115200);
79     attachInterrupt(digitalPinToInterrupt(PHASE_A), updatePos, RISING); //deklarera interruptet i main
       (innan reglerloopen)
80
81     Serial.println("Calibrating in 5s");
82
83     for(int i = 0;i<5000;i++){
84       left_sensor.update();
85       delay(1);
86     }
87
88     left_sensor.calibrate();
89     cart.calibrate();
90
91     Serial.println("Calibrated!");
92     Timer1.attachInterrupt(timerIsr).setPeriod(8000000).start();
93     Serial.println("TSTART");
94     delay(1000);
95     pinMode(DIR_PIN, OUTPUT);
96     pinMode(STEP_PIN, OUTPUT);
97
98     Serial.println("Init completed!");
99     startTime = millis();
100    lastCall = micros();
101 }
102
103 void loop() {
104   control_Loop_LQR();
105 }
106
107 void testP2(){
108   setSpeed(0.1);
109   delay(1000);
110   setSpeed(0);
111   Serial.println(cart.getPos(),5);
112   while(1){
113     Serial.println("DONE");
114     delay(10000);
115   }
116 }
117
118 //Called when Encpoder increments
119 void updatePos(){
120   cart.calculateDistance();
121 }
122
123
124 void control_Loop_PI(){
125
126   left_sensor.update();
127   //State vector
128   double theta = left_sensor.getAngleRad();
129   double pos = cart.getPos();
130   double speed = cart.getSpeed();
131
132   double h = (micros() - lastCall)/1000000.0;
133   lastCall = micros();
134
135   double ref = -pos*Kp2 + (i_part2*Ki2);
136
137   double error = ref - theta;
138   double uC = Kp*error + Ki*i_part;
139
140   i_part += h*error;
141   i_part2 += h*(-pos);
142   Serial.print(theta);
143   Serial.print("\t");
144   Serial.print(ref,4);
145   Serial.print("\t");
146   Serial.print(pos);
147   Serial.println("");
148
149   if(millis() > startTime + 1500){
```

```
150     //Serial.println(uC);
151     setSpeed(uC);
152   }
153
154   if(cart.tooFar){
155     setSpeed(0);
156     Serial.println("CART TOO FAR!");
157     delay(1000);
158   }
159   //delay(1);
160
161 }
162
163 /*
164 pin7 Port C 23
165 pin6 Port C 24
166
167 PIO_SODR
168 PIO_CODR
169
170 #define PORT PIOC
171
172 */
173 double lpFilter2(double oldX, double oldY, double h, double cutOff){
174   return (1-h*cutOff)*oldY + h*cutOff*oldX;
175 }
176
177 /*
178     Called peridicaly by the timer interupt (perido set by 'Timer1.setPeriod()')
179 */
180 void timerIsr(){
181   //Serial.println("*");
182
183     if(stepActive){
184     PIOC->PIO_CODR |= (1<<23);
185     //digitalWrite(6,LOW);
186     }else{
187     PIOC->PIO_SODR |= (1<<23);
188     //digitalWrite(6,HIGH);
189     }
190     stepActive = !stepActive;
191
192 }
193 //Testprogramm
194 void maxAcc(){
195 double maxSpeed = 0.5;
196     int axDel = 10;
197     double inc = 0.03;
198
199     for(double i  = 0;i<maxSpeed;i+=inc){
200         setSpeed(i);
201         delay(axDel);
202     }
203     for(double i  = maxSpeed;i>0;i-=inc){
204         setSpeed(i);
205         delay(axDel);
206     }
207     for(double i  = 0;i<maxSpeed;i+=inc){
208         setSpeed(-i);
209         delay(axDel);
210     }
211     for(double i  = maxSpeed;i>0;i-=inc){
212         setSpeed(-i);
213         delay(axDel);
214     }
215     setSpeed(0);
216
217     /*
218     setSpeed(5);
219     delay(5000);
220     setSpeed(-5);
221     delay(5000);
222     setSpeed(0);
223     */
224     while(1){
```

```
225         Serial.println("DONE");
226     }
227 }
228
229 void setSpeed(double speed){
230   long stepsPerSec = convertToStepsPerSec(speed);
231   setSteps(stepsPerSec);
232 }
233
234 /*
235     Set nr of steps per second to pulse the stepper, maximum is ~ 10^6 steps per sec.
236 */
237 void setSteps(double stepsPerSec){
238     if (stepsPerSec == 0){
239         Timer1.stop();
240     }else{
241     Timer1.stop();
242         bool dir = 0 < stepsPerSec;
243         unsigned long reqDelay = (1.0/abs(stepsPerSec))*1000000.0;
244         reqDelay = reqDelay/2.0;
245
246         if(reqDelay > MAX_PERIOD){
247             reqDelay = MAX_PERIOD;
248         }
249
250         if(dir){
251         PIOC->PIO_SODR |= (1<<25);
252       }else{
253         PIOC->PIO_CODR |= (1<<25);
254       }
255
256         Timer1.setPeriod(reqDelay).start();
257     }
258 }
259
260 void testStep(){
261   while(1){
262     Serial.println("High speed");
263     setSpeed(0.3);
264     delay(5000);
265     Serial.println("Reverse low speed");
266     setSpeed(-0.1);
267     delay(5000);
268   }
269 }
270
271 double convertToStepsPerSec(double speedM_S){
272   return speedM_S/((factor/1024.0)* 200 ) * NO_OF_MICRO_STEPS  * (1.0/54) ;
273
274 }
275
276
277 void control_Loop_LQR(){
278
279   left_sensor.update();
280   //State vector
281   double theta = left_sensor.getAngleRad();
282   double omega = left_sensor.getAngularVelocity();
283
284   double pos = cart.getPos();
285   double speed = cart.getSpeed();
286
287   double h = (micros() - lastCall)/1000000.0;
288   lastU = lpFilter2(lastXR,lastU,h,10);
289   lastCall = micros();
290   lastXR = -sture.getU(pos,speed,theta,omega);
291
292   Serial.print(theta);
293   Serial.print("\t");
294   Serial.print(omega);
295   //Serial.print("\t");
296   //Serial.print(pos);
297   //Serial.print("\t");
298   //Serial.print(speed);
299   //Serial.print("\t");
```

```
300   //Serial.print(lastU);
301   Serial.println("");
302
303   if(millis() > startTime + 1500){
304     setSpeed(lastU);
305   }
306
307   if(cart.tooFar){
308     setSpeed(0);
309     Serial.println("CART TOO FAR!");
310     delay(1000);
311   }
312   //delay(1);
313
314 }
315
```

```
/*IOConfig.h*/
#ifndef _IOConfig_h
#define _IOConfig_h
  /*
   * These Channels are connected to the angular sensors via a LP-filter at sepecified IO-port
   */
  #define CHANNEL_1 10
  #define CHANNEL_2 2
  #define CHANNEL_3 4

    /*
     * These pins are connected to calibrate button and start/stop button on the panel
     */

  #define START_BUTTON_PIN 2
  #define CALIBRATE_PIN  4

    /*
     * Dir pin and step pin are connected to the stepper driver.
     */

  #define DIR_PIN 31
  #define STEP_PIN 33

    /*
     * Phase A,B,Z are connected to the rotary encoder via a voltage divider
     */

  #define PHASE_A 9 //change if necessary
  #define PHASE_B 11
  #define PHASE_Z 3
#endif

```

```
1  /*LQR.h*/
2  #ifndef LQR_H
3  #define LQR_H
4
5
6  #define L1 0.35
7  #define L2 0.58
8  #define L3 0.91
9
10 class LQR {
11
12   public:
13     LQR(double par1, double par2, double par3);
14     double getU(double pos, double velocity, double angle_1, double omega_1);
15
16   private:
17     double k1;
18     double k2;
19     double k3;
20     double k4;
21     double k5;
22     double k6;
23     double k7;
24   };
25 #endif
26
```

```
1  /*LQR.cpp*/
2  #include "LQR.h"
3
4  LQR::LQR(double par1, double par2, double par3){
5    this->k1 = par1;
6    this->k2 = par2;
7    this->k3 = par3;
8  }
9
10 /*
11 * getU returns the controlsignal based on current states of cart and pendulum
12 */
13
14 double LQR::getU(double pos, double velocity, double angle_1, double omega_1){
15   double u1 = this->k1 * angle_1;
16   double u2 = this->k2 * (omega_1 * L2 + velocity);
17   double u3 = this->k7 * pos;
18   return (u1 + u2 + u3);
19 }
20
21
```

```
/*rotaryEncoder.h*/
#ifndef ROTARYENCODER_H
#define ROTARYENCODER_H


#define WHEEL_RADIUS 0.02688 //m
#define STEPS_PER_REV_PHASE_A 1024.0
#define factor 6.451612903225806451612903225806 5*pow(10,-5)*2
//calibrated factor, corrsponds to number of meters of one pulse from the rotary encoder

/*Rotary Encoder software representation.
 *Most parameters and methods need public access.
 *In the MAIN file, an interrupt is defined, attatched to the A Phase of the encoder.
 *When rotating one step, the encoder sends a pulse through phase A.
 *This triggers the Interrupt Subroutine (ISR), a void wrapper function in main
 *that ultimately triggers the class method claculateDistance in the
 *rotary encoder object.
 *calculateDistance calculates the linear distance that the cart has travelled
 *and adds or subtracts it to the paramter "pos" [m].
 *pos is given an initial value upon calibration.
*/


class rotaryEncoder{

  public:
  rotaryEncoder();
  void calibrate();
  void calculateDistance();
  double getPos();
  double getSpeed();
  bool tooFar = false;

  private:
  double pos;
  double oldPos;
  double lastCall = 0;



};


#endif

```

```
/*rotaryEncoder.cpp*/
#include "rotaryEncoder.h"
#include "Arduino.h"
#include "IOConfig.h"



/*Constructor for the rotary encoder object.
 * Defines physical pins as inputs; pulses will be sent to these
 * from the physical rotary encoder.
*/
rotaryEncoder::rotaryEncoder() { //set pins as input for the two phases of the rotary encoder
pinMode(PHASE_A, INPUT);
pinMode(PHASE_B, INPUT);
}

/*this will be triggered from a void method in main. When a pulse is detected in Phase A,
 *the position variable will increase or decrease depending on the other phase (B).
 */
void rotaryEncoder::calculateDistance(){

      if (digitalRead(PHASE_A))
        digitalRead(PHASE_B) ? pos = pos + factor : pos = pos - factor;
      else
        digitalRead(PHASE_B) ? pos = pos - factor : pos = pos + factor;

  tooFar = abs(pos) > 0.35;

}



/*
 *Calibration method on ground level.
 *Given that the cart is in the intended calibration position (any end),
 *the reference position value will be set here.
*/
void rotaryEncoder::calibrate(){
  pos = 0; //center
  }

/*common getter too keep the pos variable private.
*/
double rotaryEncoder::getPos(){
  return pos;
}

double rotaryEncoder::getSpeed(){
  double currentCall = micros();
  double h = (currentCall - lastCall)/(1000000.0);
  double speed = (pos - oldPos)/h;
  oldPos = pos;
  lastCall = currentCall;
  return speed;
}





```

```
/*sensor.h*/
#ifndef _sensor_h
#define _sensor_h

#include "Arduino.h"
#include "IOConfig.h"

class Sensor {

  public:

    Sensor(char channel, int max_angle_rotation, char max_voltage, int cal_angle);
    void update();                  //updates the value to retrun by bypassing it through a LP-filter
    double getAngleDeg();           //return current angle in degrees
    double getAngleRad();           //return current angle in rad
    void calibrate();               //Calibrates sensor
    double getAngularVelocity();    //return angular velocity in rad/s
    bool isCalibrated;              //True if calibrated, false as default

    /*
     * Below are private variable used internaly in the Sensor class.
     */
  private:
    char channel;
    int lBound_angle;
    int uBound_angle;
    int lBound_voltage;
    int uBound_voltage;
    int IO_pin;
    const static char numberOfSamples = 10;
    double angleRaw[numberOfSamples];
    double angleRawData;
    char pointer = 0;
    double angleRad;
    double angleDeg;
    long lastCall = 0;


};

#endif /* sensor.h */

```

```
1 /*sensor.cpp*/
2 #include "sensor.h"
3
4 /*
5  * defines used in file, change if needed
6  */
7
8 #define RESOLUTION 16
9 #define CONTINOUS_MODE 1
10 #define ONE-SHOOT_MODE 0
11 #define GAIN 1
12
13 //#ifdef ADC_IN_USE
14 //MCP3428* sensor = NULL;    //Creats a communication with the Analog-Digital Converter
15 //#endif
16
17 double oldVal = 0;
18 double currentVal = 0;
19 double currentRaw = 0;
20 double lastX = 0;
21 unsigned long lastCall_lp = 0;;
22 double m = 0;
23 double speed = 0;
24 double nonFilteredSpeed = 0;
25
26 Sensor::Sensor(char channel, int max_angle_rotation, char max_voltage, int cal_angle) {
27
28   this->channel = channel;             //Stores channel sensor is connected to (1..4)
29   lBound_angle = cal_angle;            //Stores at which angle the sensor is calibrated along
30   uBound_angle = max_angle_rotation;   //Stores the maximum rotation possible, for P2501A502 and P2501A202↙
      its 345 deg.
31   uBound_voltage = max_voltage;        //Stores voltage sent from Arduino, preferably 3.3V
32   lBound_voltage = 0;                  //default, use 'calibrate()' for correct value
33   isCalibrated = false;
34
35   analogReadResolution(12);
36
37     switch(channel){
38     case 1:
39       IO_pin = CHANNEL_1;
40       break;
41     case 2:
42       IO_pin = CHANNEL_2;
43       break;
44     case 3:
45       IO_pin = CHANNEL_3;
46       break;
47   }
48 }
49
50
51
52 double getRadFromReading(double x){
53   return (((1/16.45)*x)+m)*(PI/180);
54 }
55
56 double getRadFromReadingNoOffs(double x){
57   return (((1/16.45)*x))*(PI/180);
58 }
59
60 void Sensor::calibrate() {
61   isCalibrated = true;
62   double sum = 0;
63   int noOfSamples = 70;
64   int nr = 0;
65   for(int i = 0;i<noOfSamples;i++){
66     update();
67     delay(20);
68     if(i>20){
69       sum += currentVal;
70       nr++;
71     }
72
73   }
74   angleRawData = sum/nr;
```

```
75   m = -(angleRawData/16.45);
76 }
77
78
79 double Sensor::getAngleDeg(){
80     return getRadFromReading(currentVal) * 180/PI;
81 }
82 double Sensor::getAngleRad(){
83     return getRadFromReading(currentVal);
84 }
85
86 double Sensor::getRaw(){
87     return currentRaw;
88 }
89
90 double Sensor::getAngularVelocity(){
91   return speed;
92 }
93
94 double lpFilter(double oldX, double oldY, double h, double cutOff){
95   return (1-h*cutOff)*oldY + h*cutOff*oldX;
96 }
97
98 //Read sensor and update internal variables
99 void Sensor::update(){
100   double w = 10;
101   double h = (micros() - (double)lastCall_lp)/1000000.0;
102   lastCall_lp = micros();
103   oldVal = currentVal;
104     currentVal = lpFilter(lastX,currentVal,h,60);
105     currentRaw = analogRead(IO_pin);
106     lastX = currentRaw;
107
108   speed = lpFilter(nonFilteredSpeed, speed,h,60);
109   nonFilteredSpeed = getRadFromReadingNoOffs((currentVal - oldVal)/h);
110 }
111
```

```
/*
  DueTimer.h - DueTimer header file, definition of methods and attributes...
  For instructions, go to https://github.com/ivanseidel/DueTimer

  Created by Ivan Seidel Gomes, March, 2013.
  Modified by Philipp Klaus, June 2013.
  Released into the public domain.
*/

#ifdef __arm__

#ifndef DueTimer_h
#define DueTimer_h

#include "Arduino.h"

#include <inttypes.h>

/*
    This fixes compatibility for Arduono Servo Library.
    Uncomment to make it compatible.

    Note that:
        + Timers: 0,2,3,4,5 WILL NOT WORK, and will
                  neither be accessible by Timer0,...
*/
// #define USING_SERVO_LIB  true

#ifdef USING_SERVO_LIB
    #warning "HEY! You have set flag USING_SERVO_LIB. Timer0, 2,3,4 and 5 are not available"
#endif


#define NUM_TIMERS  9

class DueTimer
{
protected:

    // Represents the timer id (index for the array of Timer structs)
    const unsigned short timer;

    // Stores the object timer frequency
    // (allows to access current timer period and frequency):
    static double _frequency[NUM_TIMERS];

    // Picks the best clock to lower the error
    static uint8_t bestClock(double frequency, uint32_t& retRC);

  // Make Interrupt handlers friends, so they can use callbacks
  friend void TC0_Handler(void);
  friend void TC1_Handler(void);
  friend void TC2_Handler(void);
  friend void TC3_Handler(void);
  friend void TC4_Handler(void);
  friend void TC5_Handler(void);
  friend void TC6_Handler(void);
  friend void TC7_Handler(void);
  friend void TC8_Handler(void);

    static void (*callbacks[NUM_TIMERS])();

    struct Timer
    {
        Tc *tc;
        uint32_t channel;
        IRQn_Type irq;
    };

    // Store timer configuration (static, as it's fixed for every object)
    static const Timer Timers[NUM_TIMERS];

public:

    static DueTimer getAvailable(void);
```

```
76
77     DueTimer(unsigned short _timer);
78     DueTimer& attachInterrupt(void (*isr)());
79     DueTimer& detachInterrupt(void);
80     DueTimer& start(long microseconds = -1);
81     DueTimer& stop(void);
82     DueTimer& setFrequency(double frequency);
83     DueTimer& setPeriod(unsigned long microseconds);
84
85     double getFrequency(void) const;
86     long getPeriod(void) const;
87 };
88
89 // Just to call Timer.getAvailable instead of Timer::getAvailable() :
90 extern DueTimer Timer;
91
92 extern DueTimer Timer1;
93 // Fix for compatibility with Servo library
94 #ifndef USING_SERVO_LIB
95     extern DueTimer Timer0;
96     extern DueTimer Timer2;
97     extern DueTimer Timer3;
98     extern DueTimer Timer4;
99     extern DueTimer Timer5;
100 #endif
101 extern DueTimer Timer6;
102 extern DueTimer Timer7;
103 extern DueTimer Timer8;
104
105 #endif
106
107 #else
108     #error Oops! Trying to include DueTimer on another device?
109 #endif
110
```

```
1  /*
2   DueTimer.cpp - Implementation of Timers defined on DueTimer.h
3   For instructions, go to https://github.com/ivanseidel/DueTimer
4
5   Created by Ivan Seidel Gomes, March, 2013.
6   Modified by Philipp Klaus, June 2013.
7   Thanks to stimmer (from Arduino forum), for coding the "timer soul" (Register stuff)
8   Released into the public domain.
9  */
10
11 #include <Arduino.h>
12 #if defined(_SAM3XA_)
13 #include "DueTimer.h"
14
15 const DueTimer::Timer DueTimer::Timers[NUM_TIMERS] = {
16     {TC0,0,TC0_IRQn},
17     {TC0,1,TC1_IRQn},
18     {TC0,2,TC2_IRQn},
19     {TC1,0,TC3_IRQn},
20     {TC1,1,TC4_IRQn},
21     {TC1,2,TC5_IRQn},
22     {TC2,0,TC6_IRQn},
23     {TC2,1,TC7_IRQn},
24     {TC2,2,TC8_IRQn},
25 };
26
27 // Fix for compatibility with Servo library
28 #ifdef USING_SERVO_LIB
29     // Set callbacks as used, allowing DueTimer::getAvailable() to work
30     void (*DueTimer::callbacks[NUM_TIMERS])() = {
31         (void (*)()) 1, // Timer 0 - Occupied
32         (void (*)()) 0, // Timer 1
33         (void (*)()) 1, // Timer 2 - Occupied
34         (void (*)()) 1, // Timer 3 - Occupied
35         (void (*)()) 1, // Timer 4 - Occupied
36         (void (*)()) 1, // Timer 5 - Occupied
37         (void (*)()) 0, // Timer 6
38         (void (*)()) 0, // Timer 7
39         (void (*)()) 0  // Timer 8
40     };
41 #else
42     void (*DueTimer::callbacks[NUM_TIMERS])() = {};
43 #endif
44 double DueTimer::_frequency[NUM_TIMERS] = {-1,-1,-1,-1,-1,-1,-1,-1,-1};
45
46 /*
47     Initializing all timers, so you can use them like this: Timer0.start();
48 */
49 DueTimer Timer(0);
50
51 DueTimer Timer1(1);
52 // Fix for compatibility with Servo library
53 #ifndef USING_SERVO_LIB
54     DueTimer Timer0(0);
55     DueTimer Timer2(2);
56     DueTimer Timer3(3);
57     DueTimer Timer4(4);
58     DueTimer Timer5(5);
59 #endif
60 DueTimer Timer6(6);
61 DueTimer Timer7(7);
62 DueTimer Timer8(8);
63
64 DueTimer::DueTimer(unsigned short _timer) : timer(_timer){
65     /*
66         The constructor of the class DueTimer
67     */
68 }
69
70 DueTimer DueTimer::getAvailable(void){
71     /*
72         Return the first timer with no callback set
73     */
74
75     for(int i = 0; i < NUM_TIMERS; i++){
```

```
76          if(!callbacks[i])
77              return DueTimer(i);
78      }
79      // Default, return Timer0;
80      return DueTimer(0);
81  }
82
83  DueTimer& DueTimer::attachInterrupt(void (*isr)()){
84      /*
85          Links the function passed as argument to the timer of the object
86      */
87
88      callbacks[timer] = isr;
89
90      return *this;
91  }
92
93  DueTimer& DueTimer::detachInterrupt(void){
94      /*
95          Links the function passed as argument to the timer of the object
96      */
97
98      stop(); // Stop the currently running timer
99
100     callbacks[timer] = NULL;
101
102     return *this;
103 }
104
105 DueTimer& DueTimer::start(long microseconds){
106     /*
107         Start the timer
108         If a period is set, then sets the period and start the timer
109     */
110
111     if(microseconds > 0)
112         setPeriod(microseconds);
113
114     if(_frequency[timer] <= 0)
115         setFrequency(1);
116
117     NVIC_ClearPendingIRQ(Timers[timer].irq);
118     NVIC_EnableIRQ(Timers[timer].irq);
119
120     TC_Start(Timers[timer].tc, Timers[timer].channel);
121
122     return *this;
123 }
124
125 DueTimer& DueTimer::stop(void){
126     /*
127         Stop the timer
128     */
129
130     NVIC_DisableIRQ(Timers[timer].irq);
131
132     TC_Stop(Timers[timer].tc, Timers[timer].channel);
133
134     return *this;
135 }
136
137 uint8_t DueTimer::bestClock(double frequency, uint32_t& retRC){
138     /*
139         Pick the best Clock, thanks to Ogle Basil Hall!
140
141         Timer       Definition
142         TIMER_CLOCK1    MCK /  2
143         TIMER_CLOCK2    MCK /  8
144         TIMER_CLOCK3    MCK / 32
145         TIMER_CLOCK4    MCK /128
146     */
147     const struct {
148         uint8_t flag;
149         uint8_t divisor;
150     } clockConfig[] = {
```

```
151         { TC_CMR_TCCLKS_TIMER_CLOCK1,   2 },
152         { TC_CMR_TCCLKS_TIMER_CLOCK2,   8 },
153         { TC_CMR_TCCLKS_TIMER_CLOCK3,  32 },
154         { TC_CMR_TCCLKS_TIMER_CLOCK4, 128 }
155     };
156     float ticks;
157     float error;
158     int clkId = 3;
159     int bestClock = 3;
160     float bestError = 9.999e99;
161     do
162     {
163         ticks = (float) VARIANT_MCK / frequency / (float) clockConfig[clkId].divisor;
164         // error = abs(ticks - round(ticks));
165         error = clockConfig[clkId].divisor * abs(ticks - round(ticks)); // Error comparison needs scaling
166         if (error < bestError)
167         {
168             bestClock = clkId;
169             bestError = error;
170         }
171     } while (clkId-- > 0);
172     ticks = (float) VARIANT_MCK / frequency / (float) clockConfig[bestClock].divisor;
173     retRC = (uint32_t) round(ticks);
174     return clockConfig[bestClock].flag;
175 }
176
177
178 DueTimer& DueTimer::setFrequency(double frequency){
179     /*
180         Set the timer frequency (in Hz)
181     */
182
183     // Prevent negative frequencies
184     if(frequency <= 0) { frequency = 1; }
185
186     // Remember the frequency — see below how the exact frequency is reported instead
187     //_frequency[timer] = frequency;
188
189     // Get current timer configuration
190     Timer t = Timers[timer];
191
192     uint32_t rc = 0;
193     uint8_t clock;
194
195     // Tell the Power Management Controller to disable
196     // the write protection of the (Timer/Counter) registers:
197     pmc_set_writeprotect(false);
198
199     // Enable clock for the timer
200     pmc_enable_periph_clk((uint32_t)t.irq);
201
202     // Find the best clock for the wanted frequency
203     clock = bestClock(frequency, rc);
204
205     switch (clock) {
206       case TC_CMR_TCCLKS_TIMER_CLOCK1:
207         _frequency[timer] = (double)VARIANT_MCK / 2.0 / (double)rc;
208         break;
209       case TC_CMR_TCCLKS_TIMER_CLOCK2:
210         _frequency[timer] = (double)VARIANT_MCK / 8.0 / (double)rc;
211         break;
212       case TC_CMR_TCCLKS_TIMER_CLOCK3:
213         _frequency[timer] = (double)VARIANT_MCK / 32.0 / (double)rc;
214         break;
215       default: // TC_CMR_TCCLKS_TIMER_CLOCK4
216         _frequency[timer] = (double)VARIANT_MCK / 128.0 / (double)rc;
217         break;
218     }
219
220     // Set up the Timer in waveform mode which creates a PWM
221     // in UP mode with automatic trigger on RC Compare
222     // and sets it up with the determined internal clock as clock input.
223     TC_Configure(t.tc, t.channel, TC_CMR_WAVE | TC_CMR_WAVSEL_UP_RC | clock);
224     // Reset counter and fire interrupt when RC value is matched:
225     TC_SetRC(t.tc, t.channel, rc);
```

```
226     // Enable the RC Compare Interrupt...
227     t.tc->TC_CHANNEL[t.channel].TC_IER=TC_IER_CPCS;
228     // ... and disable all others.
229     t.tc->TC_CHANNEL[t.channel].TC_IDR=~TC_IER_CPCS;
230 
231     return *this;
232 }
233 
234 DueTimer& DueTimer::setPeriod(unsigned long microseconds){
235     /*
236         Set the period of the timer (in microseconds)
237     */
238 
239     // Convert period in microseconds to frequency in Hz
240     double frequency = 1000000.0 / microseconds;
241     setFrequency(frequency);
242     return *this;
243 }
244 
245 double DueTimer::getFrequency(void) const {
246     /*
247         Get current time frequency
248     */
249 
250     return _frequency[timer];
251 }
252 
253 long DueTimer::getPeriod(void) const {
254     /*
255         Get current time period
256     */
257 
258     return 1.0/getFrequency()*1000000;
259 }
260 
261 
262 /*
263     Implementation of the timer callbacks defined in
264     arduino-1.5.2/hardware/arduino/sam/system/CMSIS/Device/ATMEL/sam3xa/include/sam3x8e.h
265 */
266 // Fix for compatibility with Servo library
267 #ifndef USING_SERVO_LIB
268 void TC0_Handler(void){
269     TC_GetStatus(TC0, 0);
270     DueTimer::callbacks[0]();
271 }
272 #endif
273 void TC1_Handler(void){
274     TC_GetStatus(TC0, 1);
275     DueTimer::callbacks[1]();
276 }
277 // Fix for compatibility with Servo library
278 #ifndef USING_SERVO_LIB
279 void TC2_Handler(void){
280     TC_GetStatus(TC0, 2);
281     DueTimer::callbacks[2]();
282 }
283 void TC3_Handler(void){
284     TC_GetStatus(TC1, 0);
285     DueTimer::callbacks[3]();
286 }
287 void TC4_Handler(void){
288     TC_GetStatus(TC1, 1);
289     DueTimer::callbacks[4]();
290 }
291 void TC5_Handler(void){
292     TC_GetStatus(TC1, 2);
293     DueTimer::callbacks[5]();
294 }
295 #endif
296 void TC6_Handler(void){
297     TC_GetStatus(TC2, 0);
298     DueTimer::callbacks[6]();
299 }
300 void TC7_Handler(void){
```

```
301     TC_GetStatus(TC2, 1);
302     DueTimer::callbacks[7]();
303 }
304 void TC8_Handler(void){
305     TC_GetStatus(TC2, 2);
306     DueTimer::callbacks[8]();
307 }
308 #endif
309
```

C
Other

Table 15: Solution-selection matrix.

Function					
Drive line	Line	**Belt**	Wire	Chain	
Driveline tightener	**Movable end-wheel**	Turnbuckle			
Cart stop	**Rubber**	Spring	Foam	Nothing	
Cart moving	Wheels	**Linear bearings on shafts**	Linear bearings on rails		
Positional Sensor	Magnetical	Stepper motor	**Rotational encoder**	Ultra sound	Laser
Angular Sensor	Magnetical	Gyroscope	**Potentiometer**	Optical sensor	Rate gyro
Pendulum Material	Aluminum	Wood	Plastic	Steel	**Any**
Pendulum Attachment	Magnetical	Threads	Burdock	**Screw**	
Base Plate	Sheet Metal	**Wooden**	Metal Framework	None	