Design of Modular Robotic Arms With High Model Fidelity

VIKTOR KOZMA
Design of Modular Robotic Arms With High Model Fidelity

Viktor Kozma

October 09, 2015
Abstract

Mechatronic systems, mechanical systems controlled by software embedded on electronics, are used everywhere today but designing such systems impose challenges as their complexity increases. With increasingly complex hardware the controlling electronics and their accompanied software can get to generalized for their intended use. In addition, the software also gets harder to design and test when the complexity of the hardware increases, necessitating the use of simulations.

The goal of this thesis is designing a system going against this trend in the form of a robotic arm. The system developed contains both a physical design and a modeling library to allow for simulations. Both parts are modular, allowing the user to use the subsystems provided to assemble the systems for his/her intended use.

The physical robotic arm is designed around modules, or subsystems, which the user can assemble in various ways and orders to form a robotic arm. The electronic interface designed as a distributed system makes it easy to add and control individual motorized joints fitted to the arm through a central data bus.

To accompany this mechanical system, a simulation environment has been developed. This is a library developed in Modelyze, a host language encompassing a mathematical model builder and solver. The developed library contains objects whose mathematical models corresponds to the real world hardware that the user can assemble in a similar way and serves as simulation platform for the physical arm.

After extracting the necessary parameters from the hardware tests were performed comparing the simulated responses with the real ones to ensure high fidelity of the model. The model exhibited fidelity when compared with data and various discrepancies could be explained by known limitations in Modelyze and the physical setup.

The goal of the system is to serve as a research platform where, for instance, systems are designed in the model environment and implemented on the real system. Other implementations are possible thanks to the flexibility of the system.
Sammanfattning

Mekatroniska system, mekaniska system styrd av inbyggda elektroniska system, finns idag men utveckling av sådana system medför utmaningar när deras komplexitet växer. Med ökande komplex hårdvara kan den styrande elektroniken och mjuvaran bli för generaliserade för deras uppgift. Mjuvaran blir också svårare att utveckla och testa när hårdvarans komplexitet ökar, vilket nödvändiggör användandet av simuleringar.

Målet med denna avhandling är att designa ett system med som går emot denna trend i form av en robotarm. Det utvecklade systemet innehåller en fysisk design och ett modelleringssystem som möjliggör simuleringar. Båda delar är modulära, vilket möjliggör för användaren att bygga ett system för hon/hans behov.

Den fysiska robotarmen är designad runt moduler som användaren kan sätta ihop på olika sätt för att bygga en robotarm. Det elektroniska gränssnittet är utformat som ett distribuerat system vilket gör det enkelt att lägga till och kontrollera motoriserade leder som finns på armen genom en central databuss.

En simuleringmiljö har utvecklats tillsammans med den mekaniska systemet. Detta är utvecklat i Modelyze, ett värdspråk i vilket en matematiskt modelleringssystem och simuleringer är implementerade. Det utvecklade biblioteket innehåller objekt vars matematiska modeller motsvarar den fysiska hårdvaran som användaren kan bygga ihop på liknande sett för att fungera som en simuleringer.L

Efter att ha extraherat parametrar från hårdvaran utfördes tester för att jämföra simuleringar resultat med riktiga tester för att säkerställa noggrannhet i modellen. Modellen uppvisade noggrannhet när den jämfördes med datan och eventuella felaktigheter kunde bli förklarade med begränsningar i Modelyze och fysiska installationen.

Målet med detta system är att det ska bli en forskningsplattform där, bland annat, system är designade i modelleringssystemen innan de är implementerade på det rätta systemet. Ytterligare implementationer är möjliga tack vare systemets flexibilitet.
Acknowledgements

I would like to thank my external supervisor David Broman for providing continuous feedback throughout the entire project. I also want to thank my school supervisor Baha Alhaj Hasan for always being available for feedback.
## Abbreviations

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPS</td>
<td>Cyber-Physical System</td>
</tr>
<tr>
<td>EOO</td>
<td>Equation-Based Object-Oriented Language</td>
</tr>
<tr>
<td>DAE</td>
<td>Differential algebraic equation</td>
</tr>
<tr>
<td>ODE</td>
<td>Ordinary differential equation</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>CAD</td>
<td>Computer Aided Design</td>
</tr>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>ppr</td>
<td>Pulses per Revolution</td>
</tr>
<tr>
<td>dof</td>
<td>Degrees of Freedom</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>I²C</td>
<td>Inter-Integrated Circuit</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read-Only Memory</td>
</tr>
</tbody>
</table>
## Contents

### 1 Introduction
1.1 Problem Description .................................................. 1  
1.2 Purpose .......................................................................... 1  
1.3 Methodology .................................................................... 2  
1.4 Constraints ...................................................................... 3  
1.5 Contributions ................................................................... 3  
1.6 Concept of Modularity .................................................... 3  
1.7 Thesis Overview ............................................................. 5  

### 2 Introduction to Equation Based Object Oriented Languages
2.1 Causal modeling ............................................................... 7  
2.2 Acausal modeling ............................................................. 8  
2.3 Modelyze ......................................................................... 10  

### 3 Modeling of the Modular Arm
3.1 Related Work ................................................................. 17  
3.2 Planar versus 3D modeling ................................................ 18  
3.3 Modeling the Individual Parts ......................................... 18  
  3.3.1 The Link Object ......................................................... 18  
  3.3.2 The Joint Object ........................................................ 20  
3.4 Friction Modeling ............................................................ 23  
3.5 Implementation in Modelyze ............................................. 26  
3.6 Model Verification ........................................................... 28  

### 4 Control
4.1 Related Work ................................................................. 31  
4.2 Overall Control Design and System Modeling ................. 31  
  4.2.1 Operating Point .......................................................... 32  
4.3 System Modeling ............................................................. 32  
4.4 Controller Design ............................................................ 33  
  4.4.1 Control Systems ......................................................... 33  
  4.4.2 Control Structures ...................................................... 35  
  4.4.3 Pole Placement Design ............................................... 36  
  4.4.4 Choice of Poles .......................................................... 37  
  4.4.5 Current Limiting ........................................................ 37  
4.5 Controller Implementation .............................................. 38  
  4.5.1 Discrete Time Implementation ..................................... 38  
  4.5.2 Software Performance ............................................... 40  
  4.5.3 Software Implementation ............................................ 41  
  4.5.4 Modelyze Implementation .......................................... 42  
4.6 Results and Conclusion .................................................. 42  

### 5 Physical Prototype Design
5.1 Related Work ................................................................. 45  
5.2 Design Methodology ....................................................... 45  
5.3 Requirements .................................................................. 46
5.4 Choice of Hardware ............................................. 48
  5.4.1 Motor Choice ............................................. 48
  5.4.2 Position Feedback Sensor Choice ......................... 53
5.5 The Robot Arm ............................................. 56
  5.5.1 The Links ................................................. 56
  5.5.2 The Joints ............................................... 57
  5.5.3 The Baseplate ............................................ 61
  5.5.4 Modularity ............................................... 61
  5.5.5 Manufacturing and Evaluation ............................ 63
5.6 Software Design ............................................ 64
  5.6.1 Distributed System Topology ............................... 64
  5.6.2 The I²C Protocol ........................................ 65
  5.6.3 Communication Protocols .................................. 68
  5.6.4 Motor Node Layout ....................................... 69
  5.6.5 Master Layout ........................................... 72
5.7 Electronics ............................................... 73
  5.7.1 The Electronic Platform ................................. 73
  5.7.2 Electrical Layout ....................................... 76

6 Parameter Extraction ......................................... 79
  6.1 DC-motor .................................................. 79
  6.2 Links ...................................................... 79
  6.3 Joints ..................................................... 79

7 Results and Conclusion ....................................... 83
  7.1 Testing Methodology ....................................... 83
  7.2 Testing Platform .......................................... 83
  7.3 Parameter Extraction ..................................... 85
  7.4 Modelize models .......................................... 89
  7.5 Results ................................................... 90
  7.6 Discussion ................................................ 95
  7.7 Conclusion ................................................ 97
  7.8 Future Work ............................................... 97

References ..................................................... 99

A Electric Schematic .......................................... 103
B C-code Implementation of the Controller ......................... 104
C Communication Identifiers ................................... 106
D Simulink Block Diagrams for Simulation .......................... 107
1 Introduction

Cyber-physical systems (CPS) are used widely today [1]. These systems involve physical elements controlled by embedded electronics and/or networks to fulfill certain tasks and are used everywhere with ever increasing complexity. As complexity increases, simulations become a necessary tool in order to construct these systems in a cost and time effective way.

This thesis describes the construction of a CPS with accompanied simulation tools. The CPS in this regard is a modular robotic arm with the aim of being a research platform on which various kind of experiment can be performed. The modularity stems from the fact that the arm topology is not decided beforehand, the user can use the provided modules to assemble an arm relevant for his/her intended application.

This kind of mechanical modularity requires an equally flexible approach to the mathematical modeling as the user has to have the same amount of control when constructing the computer models as he/she has when constructing the physical prototype. With the use of an acausal modeling approach and a numerical equation solver the user can use the developed library containing modeled versions of the real modules to construct the required models. The user can then use these models in order to develop and fine tune the control algorithms.

1.1 Problem Description

The research problem is defined as:

Design a physical robotic arm and its mathematical model to be both modular/reconfigurable and have high fidelity

The high fidelity requirement implies that the modeled and physical system should correspond to each other in a realistic and accurate fashion.

This can be split into two parts.

- How should the hardware be designed to be modular/reconfigurable, both mechanically and electronically
- How should the mathematical model be designed to allow for building modular systems with high fidelity

1.2 Purpose

The main purpose of the thesis is to analyze a way to design and implement a CPS using the equation-based object-oriented language, in this case a robotic arm. This modeling will be checked against a hardware robotic arm that can be reconfigured in several ways. The simulated output should correspond to the actual output of the robotic arm, i.e. have a high model fidelity.

The modeling will be implemented in Modelyze, which is a Equation-Based Object-Oriented language modeling and simulation tool developed by the supervisor [2, 3]. Modelyze is in itself not a modeling language but a host language
in which the modeling and simulation tool libraries are written. This makes it a great platform on which research in these areas can be performed.

So there is two reasons to why Modelyze been chosen for this project. One is that this thesis project is the first practical application of these modeling and simulation languages, functioning as a testing project to look for bugs and propose additional features. Also the produced libraries will serve as a tool when continuing to develop the language. For example you get the opportunity to model something that exists in real life and can thus compare the results.

The purpose of the physical system is, in addition to validating the mathematical modeling, intended to be used as a testing platform for research in modeling, simulation, machine learning and time-aware programming. A system of this type can then be used to provide the means to interact with the environment and react to external stimuli. The modularity allows for building hardware specific to the task while the Modelyze library allows for software simulation of expected results.

The final goal is an open source platform that allows anyone to take part of this work. This already includes Modelyze and its libraries and will include the hardware design as well to allow for anyone to download and build this systems themselves.

1.3 Methodology

The modeling and physical design part is performed in different ways.

For the mathematical modeling the theory is researched, both learning Modelyze and the equation-based techniques used as a framework for the entire model. The library is then developed, adding more functionality when the need arises.

The design of the physical prototype starts with collecting requirements and performing simulations to help the choice of hardware in order to fulfill the requirements. After that an iterative approach is used as the manufacturing method of choice is 3d-printing that allows for quick iterations between design and evaluation.

The modularity in the hardware sense is taken into consideration during the design phase, taking care when building the subsystems that they can be assembled in various ways. This is done for both the mechanical and electronic parts, ensuring that they fulfill the modularity criteria.

The modeling part is modular since it’s modeled with a Equation-Based approach as this type of modeling has inherent modular properties. This ensures that models constructed in this way is inherently modular by design.

The fidelity of the mathematical model is tested by comparing simulated data with real test data. This is done for different scenarios, including open-loop and closed-loop step and sinusoidal responses. Sometimes Simulink is used as an intermediary when missing functionality in the base libraries of Modelyze results in known errors. The results are visually compared to each-other to ensure that the mathematical model displays high fidelity.
1.4 Constraints

In order to keep the scope down the modeling part of this thesis is kept in two dimensions.

1.5 Contributions

There are two contributions to the area done by the author.

The first contribution is a 2D mechanical library for Modelyze. The purpose of this library is to provide functionality for building and simulating modular systems.

The second contribution is a design of a small scale modular robotic arm, or more correctly the subsystems (or modules) needed to assemble a functioning robotic arm. The modules can be assembled in any orientation, allowing for full control about the configuration.

1.6 Concept of Modularity

It is important that the meaning of modularity in this regard is understood, how this concept is applied to robotic arms.

Modularity for engineering systems can be defined as: ”Building larger systems by combining smaller subsystems”. Another related term is reconfigurability, which means that you should be able to assemble the subsystems in any order possible. Figure 1 shows different subsystems for a robotic arm.

![Figure 1: The main components used to construct a working robotic arm](image)

The subsystems in this case consists of joints containing the motors and control electronics and links that provide reach and allows for the joints to be connected together. These standardized connections also allows for additional subsystems like end effectors to be attached to the arm.
These can then be assembled in different ways with different length separation and degrees of freedom as seen in Figure 2:

![Different arm configurations](image)

As the joints do not necessarily have the same motors, some might have different gearings or of another model, different performance can be achieved depending on the order in which they’re mounted. So generally to get maximum performance more powerful motors with higher gear ratios should be placed closer to the middle. However because of the modularity the user got full control of the order in which the motors are placed to fulfill his/hers particular need.

The arm is not only locked to this type of two dimensional layout but can be configured in more individualized configuration. For example see Figure 3 where the arm is assembled in such a way that the master controller is controlling two arms simultaneously. This allows for testing algorithms that is supposed to synchronize two arms against each other.
Figure 3: A double arm configuration where the master controller controls both arms simultaneously.

The arm is not restricted to only work in two dimensions but should also be able to be constructed in three dimensions, as visualized in Figure 4:

Figure 4: A three-dimensional configuration.

So modularity in this sense is how you can construct the system out of the subsystems provided.

1.7 Thesis Overview

Section 2 explains the theoretical background for the modeling part. It presents different modeling techniques and reasoning behind their individual use and why the equation-based approach was chosen for this project. It also introduces the reader to Modelyze, the modeling language used when building and simulating the developed model.

In Section 3 the mathematical models for the robotic arm is deduced and subsequently implemented in Modelyze.

In Section 4 the control algorithm is developed. It explains the reasoning behind the choice of controller, its design and implementation.

In Section 5 the design of the physical prototype is explained. It explains the design and construction of the individual subsystems in addition to the software and electronic design. It explains how the modularity is taken into consideration and how the system is designed around it.

Section 6 handles the developed procedure of extracting parameters from the physical prototype. These parameters are then used by the mathematical model in order to simulate the physical prototype with high fidelity.
The results are presented in Section 7 where tests comparing the model to reality. This section also discusses the results, draws conclusion and proposes future work.
2 Introduction to Equation Based Object Oriented Languages

This section introduces the reader to the concept of equation based object oriented languages (EOOs). It compares different approaches of constructing models of dynamic systems [4] and how these techniques can be used to construct object oriented models.

2.1 Causal modeling

The most common used approach is Causal modeling which is described by models on the form:

\[
\begin{align*}
\dot{x} &= f(t, x, u) \\
y &= g(t, x, u)
\end{align*}
\]  

(1)

Here \( x \) represents the internal states, \( u \) the in/control signal and \( y \) the output. This is a useful formulation from a mathematical standpoint as it provides us with a direct mathematical model. The functions \( f \) and \( g \) can be used to linearize the system and extract transfer functions which allow us to easily synthesize control algorithms.

Bigger systems then consist of connections of the elements described by (1), called blocks. This provides a very intuitive and easy to visualize interface on which we easily can view the path of the signals.

However these kinds of formulations has its limitations. First you need to construct your model on the form described by (1) from a set of describing equations which takes time. This also produces other problems concerning model reusability and flexibility. If we want to change one of the equations, for example adding springs or dampening, which is usually done when improving the model, we might have to redo the entire formulation. This has a negative impact on the reusability of the model.

Another problem is that you need to know which parameters that will be interesting before you formulate the model which removes flexibility.

This approach can be visualized by doing a simple model of a DC-motor. The physical model can be described by Figure 5:

![Figure 5: The physical system of a DC-motor](image)

From this figure the describing equations of this system can be derived as:
\[ u - R \cdot i - L \cdot \frac{di}{dt} - e = 0 \]
\[ e = K \cdot \frac{d\theta}{dt} \]
\[ J \cdot \frac{d^2 \theta}{dt^2} = T - c \cdot \frac{d\theta}{dt} \]
\[ T = K \cdot i \]

Formulating these on the form described by (1) results in the following system of equations:

\[ \frac{di}{dt} = \frac{1}{L} \cdot (u - R \cdot i - K \cdot \omega) \]
\[ \frac{d\theta}{dt} = \omega \]
\[ \frac{d\omega}{dt} = \frac{1}{J} \cdot (K \cdot i - c \cdot \omega) \]

Keeping with the formulation in (1) \( x = [\theta, \omega, i] \) is the state and the applied voltage \( u \) the control signal.

A block diagram out these equations can be seen in Figure 6:

![Block diagram of a DC-motor made in Simulink](image)

Figure 6: A block diagram of a DC-motor made in Simulink

Looking at this block diagram another flaw with the causal formulation can be seen; the topology is lost. Meaning that given the block diagram in Figure 6 it is not a trivial task of discerning the actual topology of the system as seen in Figure 5.

As this kind of insight is crucial when dealing with modularity this approach is not suited for this project.

### 2.2 Acausal modeling

An acausal modeling approach on the other hand avoids some of these problems. It takes the modeling to a higher level by automatically generating the block diagrams from the provided equations. This provides a flexible approach as changing single equations does not require you to reformulate the entire system.
The result of this modeling approach is a more general form of differential equations called differential-algebraic equations (DAEs). These types of equations appear on the form:

\[ F(\dot{x}, x, y, t) = 0 \] (4)

This is a system of equations where \( x \) is a vector with dependent variables that are differentiated, \( y \) is a vector with dependent variables that isn’t differentiated (algebraic variables) and the scalar \( t \) is an independent variable, usually time[5]. For example the equations in (2) are presented in an acausal form.

Generally, for more advanced systems it isn’t possible to reduce the DAE to an ODE form as done with the with the DC-motor in (3). However there exists methods for numerically solving them in addition to symbolic methods that simplifies the models, allowing for simulation [6].

This general formulation of the system equations gives us the ability to easily construct large system by just providing the equations for the individual sub-systems and how they’re connected together. This is done through conservative equations, such as Kirchoff’s law and sum-to-zero equations.

For example, regard the electrical circuit presented in Figure 7

\[
\begin{align*}
\text{\textbf{v}}_1 & = \text{\textbf{u}}_i \\
\text{\textbf{u}}_r & = R \cdot \text{\textbf{i}} \\
C \frac{d\text{\textbf{u}}_c}{dt} & = \text{\textbf{i}}_c \\
\text{\textbf{u}}_l & = L \frac{d\text{\textbf{i}}_l}{dt}
\end{align*}
\] (5)

These are then connected together with conservative equations that takes the potential (\( v_1, v_2 \) and \( v_3 \)) from the negative side of one component is one the positive side on the next. This generates the following equations:

\[
\begin{align*}
\text{\textbf{u}}_i & = v_1 - v_3 \\
\text{\textbf{u}}_r & = v_1 - v_2 \\
\text{\textbf{u}}_c & = v_2 - v_3 \\
\text{\textbf{u}}_l & = v_2 - v_3 \\
\text{\textbf{u}}_{out} & = v_2 - v_3 \\
v_3 & = 0
\end{align*}
\] (6)

Figure 7: Example circuit for generating system equations

In this circuit we have four objects, (voltage source, resistor, capacitor and inductor) that is described by the following equations:
The last equation stating that $v_3 = 0$ comes from the fact that that potential is grounded.

The last part is to use the conservative equations to form the last equations necessary in order to be able to form the system:

$$i - i_c - i_l = 0$$

(7)

So here we have a system of equations with 11 unknowns ($u_{in}$, $u_r$, $i$, $u_c$, $i_c$, $u_l$, $i_l$, $u_{out}$, $v_1$, $v_2$, $v_3$) and 11 equations. This forms an DAE where the $x$ vector contains $u_c$ and $i_c$ (as they’re differentiated), the $y$ vector the rest of the unknowns and the scalar $t$ is the time. It can then be simulated by various numerical methods.

This way to construct these kinds of systems naturally results in the possibility to make it object oriented. An object then consists of describing equations (like the ones in (5)), parameters (like $u$, $R$, $C$, $L$, etc.) and connection information.

A great number of tools that allows you to build complex systems using this kind of object-oriented building technique. Because of this one intuitive approach of building models in a graphical way. Figure 8 shows the DC-motor model built using one of these graphical model building tools:

![Figure 8: A block diagram of a DC-motor made in OpenModelica](image)

This acausal approach to system modeling thus provides a very easy and intuitive way of building systems. However it comes with the cost of not getting a good insight in the resulting mathematical formulas. This makes it hard to use this approach for applications like designing control system where such insight is crucial in order to achieve optimal performance.

### 2.3 Modelize

One of the most common acausal modeling languages used today is Modelica [7]. Tools like OpenModelica and Dymola provides an graphical interface in which to build the models using the Modelica language, allowing the user to build and simulate models in an intuitive way. The model in Figure 8 is made in OpenModelica.
The modeling in this project is done in Modelyze [2, 3]. Modelyze is in itself not a language but a framework/meta language for implementing domain specific languages (DSLs), which in this case is a physical modeling interface and solver.

Listing 1 shows the modeling the DC-motor system governed by the equations (2) in Modelyze.

```python
def DCmotor(u : Real , R: Real , K: Real , L : Real , J : Real , c : Real ) = {
    def e , T , i , th : Real ;
    u − R∗i − L∗i ' − e = 0.0 ;
    e = K∗th ' ;
    J∗th '' = T − c∗th ' ;
    T = K∗i ;
    probe("Theta") = th ;
}
```

Listing 1: DC-motor modeling in modelyze

The probe command defines the output, what will be passed out from the solver. This can be set to any variable, allowing for easy access and high flexibility.

Modelyze also supports object oriented modeling by providing libraries of electrical and physical components that can be connected together. As an example, describing the example electrical system shown in Figure 7 can be done as seen in Listing 2.

```python
include ModelyzeEOO
include Electrical // Library containing electrical components

def Circuit(u : Real , R : Real , C : Real , L : Real ) = {
    def n1,n2,n3 : Electrical ;
    def uOut : Signal ;
    ConstantVoltage(u,n1,n3) ;
    Resistor(R,n1,n2) ;
    Capacitor(C,n2,n3) ;
    Inductor(L,n2,n3) ;
    Ground(n3) ;
    VoltageSensor(n2,n3,uOut) ;
    probe("u_out") = uOut ;
}

def main = printsim(Circuit(10.0,10.0,0.0001,0.0001),0.0001,0.01)
```

Listing 2: A Modelyze representation of the electrical system described in Figure 7 with given parameters

Each of these components are defined in the standard library electrical.moz included in the header and with the relevant elements defined in Listing 3.

```python
def ConstantVoltage(V:Real , p:Electrical , n:Electrical) = {
    def i:Current ;
    def v:Voltage ;
    Branch(i,v,p,n) ;
    v = V ;
}
def Resistor(R:Real , p:Electrical , n:Electrical) = {
```
It uses the concept of branches to construct the node-linking equations that links each component together [8]. The first element in the Branch command defines the flow variables and the second element is the potential variables. The solver then uses the defined equations to calculate the relative potential between each node for the Branch command or absolute potential for the RefBranch command in order to get a full set of equations.

The difference between the Branch command and the RefBranch command is how they calculate the potentials. The Branch command works with relative potential between two points while the RefBranch works with absolute potentials. The Ground component makes sure that this absolute potential is zero at that point in the schematic. You could technically define the Inductor in Figure 7 with the component described in Listing 5 and call it with Inductor2Ground(L,n2) in Listing 2.

A Branch can also be substituted with two RefBranch calls, one for each node, and defining the difference in potential by subtracting the absolute potential in these two points. Defining the inductor component in this way can be seen in Listing 5.
```plaintext
def Inductor(L: Real, p: Electrical, n: Electrical) = {
    def i, i_p, i_n : Current;
    def dv, v_p, v_n : Voltage;
    RefBranch(i_p, v_p, p);
    RefBranch(-i_n, v_n, n);
    i_p = i;
    i_n = i;
    dv = v_p - v_n;
    L * i' = dv;
}
```

Listing 5: Alternative formulation for the inductor in Listing 3

The minus sign in the second `RefBranch` is there to ensure that the correct sum-to-zero equations are generated.

To visualize exactly how these conservative equations are generated consult Figure 9:

```
RefBranch(i_p1,v_p1,n1)
RefBranch(-i_n1,v_n1,n2)

RefBranch(i_p2,v_p2,n2)
RefBranch(-i_n2,v_n2,n3)

RefBranch(i_p3,v_p3,n2)
RefBranch(-i_n3,v_n3,n3)

v_n1 = v_p2 = v_p3
i_p2 + i_p3 - i_n1 = 0
```

Figure 9: How the conservative equations are generated when using the `RefBranch` function

With five objects connected to the same node as described the resulting conservative equations are generated.

Mechanical systems can be described in a similar way with other flow and potential variables. There are several different ways these can be defined, like defining the linear/angular velocity as the flow and force/torque as potential. Modelica has however gone with using force/torque as the flow quantity and the position/angle as the potential quantity and the same definition is used in Modelyze. So the resulting conservative equations then makes sure that each node has the same position/angle and that the forces/torques sum to zero.

So describing a the rotational dynamics system described by Figure 10 with Modelyze can be done as Listing 6.
Figure 10: A rotational mechanical system

```python
include modelyzeEOO
include Mechanical

def mechSys() = {
def n1, n2, n3, n4, n5 : Rotational;
def s1, s2 : Signal;
Fixed (0.0, n1);
Spring (1.0, n1, n2);
Damper (0.02, n1, n2);
Inertia (0.01, n2, n5);
Spring (1.0, n3, n4);
Damper (0.02, n3, n4);
Inertia (0.01, n4, n5);
ConstantTorque (1.0, n5);
AngleSensor(n2, s1);
AngleSensor(n4, s2);
probe("th_1") = s1;
probe("th_2") = s2;
}
def main = printsim (mechSys(), 0.01, 10.0)
```

Listing 6: Describing the mechanical system in Figure 10 in Modelyze

It is built upon a similar node based structure as the electrical circuit. The relevant elements contained in the standard library mechanical.moz included in the header is listed in Listing 7.

```python
def Spring(c: Real, flangeA: Rotational, flangeB: Rotational) = {
def tau: Torque;
def relphi: Angle;
Branch(tau, relphi, flangeB, flangeA);
tau = c * relphi;
}
def Damper(d: Real, flangeA: Rotational, flangeB: Rotational) = {
def tau: Torque;
def relphi: Angle;
Branch(tau, relphi, flangeB, flangeA);
tau = d * relphi';
}
def Inertia(J: Real, flangeA: Rotational, flangeB: Rotational) = {
def tauA: Torque;
def tauB: Torque;
def phiA: Angle;
def phiB: Angle;
```

14
def phi: Angle;
    RefBranch(tauA, phiA, flangeA);
    RefBranch(-tauB, phiB, flangeB);
    phiA = phi;
    phiB = phi;
    J * phi'' = tauA - tauB;
}
def Fixed(angle: Real, flangeB: Rotational) = {
    def tau: Torque;
    RefBranch(tau, angle, flangeB);
}
def ConstantTorque(tau: Real, flangeB: Rotational) = {
    def phi: Angle;
    RefBranch(-tau, phi, flangeB);
}

Listing 7: The components used in Listing 6 from the mechanical library

There is, as in the electrical circuit, several ways you can formulate the branches in order to achieve the same results. The ones in Listing 7 is just one possible way of doing it.

Another strength with the text interface is that it becomes very trivial to construct big but repetitive systems. Say that we want to expand the mechanical system in Figure 10 to include an arbitrary number of these spring/damper/inertia combinations. This can be done by using recursion as showed in Listing 8.

```python
include modelyzeEOO
include Mechanical

def armElement(flangeA: Rotational, flangeB: Rotational) = {
    def flangeMid: Rotational;
    Spring(1.0, flangeA, flangeMid);
    Damper(0.2, flangeA, flangeMid);
    Inertia(0.01, flangeMid, flangeB);
}
def recursiveArm(n: Int, flangeA: Rotational, flangeB: Rotational) -> Equations = {
    def K = 1.0; def d = 0.02; def J = 0.01;
    if n == 1 then {
        armElement(flangeA, flangeB);
    } else {
        def flangeMid: Rotational;
        armElement(flangeA, flangeMid);
        recursiveArm(n - 1, flangeMid, flangeB);
    }
}
def mechSys() = {
    def n1, n2: Rotational;
    def s1: Signal;
    Fixed(0.0, n1);
    recursiveArm(10, n1, n2);
    ConstantTorque(1.0, n2);
    AngleSensor(n2, s1);
    probe("theta") = s1;
}
```
These techniques can be used to construct large systems with a minimal amount of effort, providing a high amount of flexibility.
3 Modeling of the Modular Arm

In order to simulate the movements of the robotic arm its dynamic model has to be derived.

There are two primary ways of deriving these equations [9]. The first is using the so called Euler-Lagrangian equations in which the total energy of the system is described. Using energy conservation arguments the system dynamics can be derived. The second method is the Newton-Euler method in which the forces acting on the elements in the system and how they interact with each other is used to derive the system equations.

Each of these methods has its pros and cons. The Euler-Lagrangian works very well when the goal is an analytical solution as it’s considering the whole system from the formulation stage. The Newton-Euler considers each element in turn before merging them together for a final formula. This makes it more complex when formulating the describing equations but simplifies numerical simulations as a recursive approach can be used. This also makes it very well suited for numerical simulation.

Due to the fact that this will be modeled using Modelyze, which works with a node based structure, the chosen method is the Newton-Euler method.

The general setup for this is similar to the approach done in Listing 6 for the one dimensional mechanical systems but with different definitions of the nodes. Because the simulation will take place in a two-dimensional plane each node needs three variables to describe it [10]. To describe it’s position we need to know it’s $x$ and $y$ coordinates in addition to the rotation $\theta$ around the $z$-axis. We also need to know the three forces acting upon it, the forces in the $x$ and $y$ direction, $f_x$ and $f_y$, and the torque around the $z$-axis, $T$.

3.1 Related Work

There’s been work done before with the aim of providing easy way to build models of mechanical systems in multiple dimensions.

The main tool for doing these sorts of model building is modelica since it provides a good simulating environment as well as an easy tool for building the models. Some libraries have been constructed with the aim of providing model building in multiple dimensions. One of the libraries is the Multibody library which allows building and simulating models in a 3D-space [11]. A simplified library that works in two dimensions with the intention of teaching modelica is also available [10].

Another approach comes from the modeling technique bond-graphs. Since bond-graphs works on the same principles as the equation-based object-oriented languages, with the concept of effort and flow between nodes based on equations, existing models can be used as inspiration [12].
3.2 Planar versus 3D modeling

As to keep the scope of this project narrow enough the Modelyze library should be developed in two-dimensions. This allows for several simplifications over the 3D-system.

One such simplification is that the rotational inertia is constant for all possible orientations while in a 3D-space an inertia tensor has to be constructed in order to allow for transformations between the coordinate system.

In addition when moving in a 3D-space you also have to take gyroscopic forces into account [13]. Being able to ignore that makes the models easier to build as well as to verify and validate.

3.3 Modeling the Individual Parts

We define the nodes to be the connections between each of the parts. In each node we define the flow to be $[f_x, f_y, T]$ and the potential to be $[x, y, \theta]$. Each part has two connections, or flanges, which each can be used to connect to other parts.

Two main components are modeled, the link object and the joint object. The link object connects two points in space together with a rigid connection and thus add reach and inertia to the system. The joint object connects two flanges together by a rotating connection powered by a dc-motor and will be the main powered unit in the system. Each joint has its own inertia and mass that corresponds to the inertia and mass of the real joint, allowing the library to include all individual hardware parameters for each individual joint.

3.3.1 The Link Object

As the main building block of the modular system is the link object, which is an object which links between two points on the planar space.

We assume it to be rigid and horizontal to the ground to not get any impact from the gravity. The free-body diagram of this system is shown in Figure 11.
The physical parameters of the arm is its mass $m$ and its rotational inertia $J$ around its center of mass.

The describing equations for this link is as follows:

Given the positions $x_A$ and $y_A$ the values of $x_m$, $y_m$, $x_B$ and $y_B$ can be kinematically derived by:

$$
x_B = x_A + L \cdot \cos \theta
$$
$$
y_B = y_A + L \cdot \sin \theta
$$
$$
x_m = x_A + L_m \cdot \cos \theta
$$
$$
y_m = y_A + L_m \cdot \sin \theta
$$

The angle of both flanges are the same:

$$
\theta = \theta_A = \theta_B
$$

The force balance equations is as follows:

$$
m \ddot{x}_m = f_{xA} - f_{xB}
$$
$$
m \ddot{y}_m = f_{yA} - f_{yB}
$$

The torque balance around the center of mass becomes:

$$
J \ddot{\theta} = T_A - T_B + f_{xA} \cdot L_m \cdot \sin \theta - f_{yA} \cdot L_m \cdot \cos \theta +
+ f_{xB} \cdot (L - L_m) \cdot \sin \theta - f_{yB} \cdot (L - L_m) \cdot \cos \theta
$$
3.3.2 The Joint Object

The joint object is an object which connects two flanges together by a rotational joint, for example the end of one link with the start of another. Between the two connecting flanges of the joint a torque can be applied in order to provide actuation.

The simplest way of building a joint between two flanges is just applying a torque between these. That is a massless joint that just provides a motor torque to the links.

Let’s say that flange A has the position coordinates $x_A$, $y_A$ and $\theta_A$ and the forces/torques $f_{xA}$, $f_{yA}$ and $T_A$ and flange B has the position coordinates $x_B$, $y_B$ and $\theta_B$ and the forces/torques $f_{xB}$, $f_{yB}$ and $T_B$. A free body diagram of the torques of this joint can be seen in Figure 12:

![Figure 12: The torques of a massless joint](image)

Because the joint only extends in the z-direction the x and y coordinates are the same for both flange A and flange B but it can rotate freely relative to each other:

\[
\begin{align*}
  x_A &= x_B \\
  y_A &= y_B \\
  \Delta \omega &= \dot{\theta}_B - \dot{\theta}_A
\end{align*}
\]

Assuming rigid connections it transfers all the forces without hindrance:

\[
\begin{align*}
  f_{xA} &= f_{xB} \\
  f_{yA} &= f_{yB}
\end{align*}
\]

With an applied torque $T_M$ and the friction $T_f$ the resulting torques on flange A and flange B becomes:

\[
\begin{align*}
  T_A - T_M + T_f &= 0 \\
  T_M - T_f - T_B &= 0
\end{align*}
\]

The friction $T_f$ depends on the relative rotational speed $\Delta \omega$. This relationship is described in Section 3.4.
If the applied torque $T_M$ originates from a DC-motor with an applied voltage $u$, back-EMF constant $K_e$, electrical resistance $R$, inductance $L$, gear ratio $n$ and efficiency $\eta$ it can be described as in (2):

\[
\begin{align*}
u - R \cdot I - L \cdot \frac{dI}{dt} - K_e \cdot n \cdot \Delta \omega &= 0 \\
T_M &= \eta \cdot K_e \cdot n \cdot I
\end{align*}
\] (15)

Using these equations a massless joint can be constructed allowing for two links to be connected.

But because that this approach ignores inertias the additional inertia introduced by the joint has to be implemented externally in order to produce accurate simulations. This means that this model can’t be used individually if you want a proper modular system.

A better way to build the joint model to include its mechanical properties so it gets this functionally and can be done in the following way. Consider the free-body diagram in Figure 13:

The joint consists of two parts, the top and bottom that are connected together through the motor and a bearing. The top part (flange B) is connected to the motor housing/stator and the bottom one (flange A) is connected to the motor shaft/rotor. Each of the parts has an individual inertia, $J_A$ for flange A and $J_B$ for flange B. The entire joint has the mass $m$. 

![Figure 13: Describing the forces and torques on a arm joint](image-url)
Similarly as the previous joint each flange can rotate independently of each other but has the same \( x, y \) coordinates. The attachments are offset from the center of rotation by the lengths \( L_A \) and \( L_B \) respectively.

The joint is driven by a DC-motor with an applied voltage \( u \), back-EMF constant \( K_e \), electrical resistance \( R \), inductance \( L \), gear ratio \( n \) and efficiency \( \eta \) acts upon flange B with a torque \( T_M \). Because the motor housing is connected to the flange A, an equal but opposite torque acts on flange A.

This results in the following describing equations:

The kinematic equations:

\[
\begin{align*}
x &= x_A + L_A \cdot \cos \theta_A \\
x_B &= x + L_B \cdot \cos \theta_B \\
y &= y_A + L_A \cdot \sin \theta_A \\
y_B &= y + L_B \cdot \sin \theta_B \\
\Delta \omega &= \dot{\theta}_B - \dot{\theta}_A
\end{align*}
\] (16)

The force balance:

\[
\begin{align*}
m \ddot{x} &= f_{xA} - f_{xB} \\
m \ddot{y} &= f_{yA} - f_{yB}
\end{align*}
\] (17)

The torque balance:

\[
\begin{align*}
J_A \ddot{\theta}_A &= -T_M + T_A + T_f + \\
&+ f_{xA} \cdot L_A \cdot \sin \theta_A - f_{yA} \cdot L_A \cdot \cos \theta_A \\
J_B \ddot{\theta}_B &= T_M - T_B - T_f + \\
&+ f_{xB} \cdot L_B \cdot \sin \theta_B - f_{yB} \cdot L_B \cdot \cos \theta_B
\end{align*}
\] (18)

The DC-motor equations for generating \( T_M \) is the same as in (15).

This approach makes a few assumptions. One of the major ones is that the center of mass is aligned at the center of rotation, which could be a source of inaccuracies when simulating.

Another way of constructing this model is taking the object oriented nature of the way the system is built into account. This means that we can use two arm link objects connected together with a dc-motor actuator to model a joint. See Figure 14 for visualization:
Instead of describing the entire system with a set of equations we build up the joint using already existing parts. In this case two links connected together by the massless joint described in Figure 12. This generates a much more accurate model with more flexibility since each part can be modeled individually.

This approach has a few of downside. One is that this approach generates a lot more equations, resulting in this system taking longer to calculate. Another drawback is that this results in a lot more parameters that has to be identified, such as the center of mass offset lengths $L_{mA}$ and $L_{mB}$ and the individual link masses $m_A$ and $m_B$.

### 3.4 Friction Modeling

In order to produce accurate simulations of the physical system it is important to know how the friction behaves. That is how big the friction torque is with relationship to the dynamic parameters. Most commonly the rotational speed as we can usually expect a higher friction torque at higher speeds.

The most common way of describing this type of friction is a combination of static friction, also called Coulomb friction, and viscous friction, which is speed dependent friction [14]. Visualizing this type of friction model versus the rotational speed is done in Figure 15:
A combination of Coulomb and dynamic friction

Figure 15: Linear discontinuous relation between the friction torque $T_f$ and the rotational speed $\omega$

To describe this type of friction we need two parameters, the maximum static friction $c_c$ and the dynamic friction $c_d$. Knowing these two parameters we can formulate the friction torque as:

$$T_f(\omega) = c_c \cdot \text{sgn}(\omega) + c_d \cdot \omega \quad (19)$$

The downside of this representation is that it requires discontinuous modeling to realize, as the object won’t start to rotate until a certain torque, $c_c$, has been applied. So the friction model have to apply the counter-torque equal to the applied torque until the value $c_c$ has been reached. In order to realize this discontinuous modeling tools are required.

But since Modelyze is in its early stages and miss the discontinuous, also called hybrid modeling, this type of friction can’t be implemented. So instead a continuous simplification of the friction can be used. Such a representation using an exponential function can be formulated in the following way:

$$T_f(\omega) = c_1 \cdot \text{sgn}(\omega) \left( 1 - e^{-c_2|\omega|} \right) + c_3 \cdot \omega \quad (20)$$

This is visualized in Figure 16
Friction described by an exponential function

Figure 16: Exponential relation between the friction torque $T_f$ and the rotational speed $\omega$

This function is continuously differentiable for all values of $\omega$, allowing for simulation without discontinuous functionality.

It does however lose fidelity close to zero as any torque, how small, will induce a rotation. This is however small enough to be negligible, considering that a lot of other unmodeled friction dynamics is prevalent in this range.

This is a simplified way of describing the friction, the relationships are in reality a lot more complex. On such additional complexity is the fact that the Coulomb friction constant differs when the two areas are stationary and moving relative to each other. That is two areas moving against each other usually experience less Coulomb friction than when stationary. Usually one can approximate the static constant with the dynamic one but for very high fidelity systems this difference has to be taken into account. This raises some difficulties, for example extracting these individual parameters and modeling the transition between the stationary and moving mode.

There are other ways to improve upon this model. One such is improvement is using a quadratic or even cubic representation of the relationship between friction and speed. This may improve the fit when trying to extract the friction parameters from the test data but could result in over-generalizing the model. Another improvement could also be that there exist some relationship between friction and another dynamic variables, for example acceleration, that could be taken into account.

Other types of friction that can be modeled exists, such as slip-stick friction [15]. This type of friction can be visualized as springs attached between the two areas that stick and release. This creates relationships that is hard to model and simulate properly.

Another effect that needs to be taken into account is the normal force dependence of the friction constants as basic friction theory says that the friction is proportional to the normal force between the areas [14]. This is something that has to be taken into account as the modular design philosophy does not guarantee that the friction parameters are constant between different configurations.
So the system has to take the weight of each individual component and add them together to calculate the normal forces between the different joints.

### 3.5 Implementation in Modelyze

Because of the fact that this system works in a 2D-system three different sets of conservative equations has to be constructed in order to connect the objects together. One for each of the positions and forces in the $x$ and $y$ direction and one for the angle and torque in the $\theta$ direction.

To simplify the calling of these objects so that it isn’t necessary to pass three different arguments for each flange the `MultiBranch` object is used. Listing 9 demonstrates this concept.

```python
// Using single branches
def PlanarObject(flangeAx: Translational, flangeAy: Translational, flangeAth: Rotational, flangeBx: Translational, flangeBy: Translational, flangeBth: Rotational) = {
  // Variables
  def fxA, fyA, fxB, fyB : Force;
  def TA, TB : Torque;
  def xA, yA, xB, yB : Position;
  def thA, thB : Angle;

  // Branches
  RefBranch(fxA, xA, flangeAx);
  RefBranch(fyA, yA, flangeAy);
  RefBranch(TA, thA, flangeAth);
  RefBranch(-fxB, xB, flangeBx);
  RefBranch(-fyB, yB, flangeBy);
  RefBranch(-TB, thB, flangeBth);

  /*
   * Object equations goes here
   */
}

// Equivalent object using MultiBranches
def PlanarObject(flangeA: Mechanical2D, flangeB: Mechanical2D) = {
  // Variables
  def fxA, fyA, fxB, fyB : Force;
  def TA, TB : Torque;
  def xA, yA, xB, yB : Position;
  def thA, thB : Angle;

  // Branches
  MultiRefBranch([fxA, fyA, TA], [xA, yA, thA], flangeA);
  MultiRefBranch([-fxB, -fyB, -TB], [xB, yB, thB], flangeB);

  /*
   * Object equations goes here
   */
}
```

Listing 9: How the `MultiBranch` works in Modelyze
The different equations for each type of object are then implemented in Modelyze using the MultiBranch object. Additional objects like fixing (equivalent to grounding), sensors and feedback controllers are included in addition to more experimental joints like springs and constant torque output. This provides all the tools to build a 2D-system while also providing a good base on which to implement additional objects.

The implementation of the modular arm system is implemented in three layers as shown in Figure 17:

- **User application**: User level, the user building the system with components from the modular level
- **modularrmslib.moz**: Modular level, contains parameters for each physical object and wraps the equation level
- **mechanical2d.moz**: Equation Level, contains the equations of motion

![Figure 17: Modular arms system implementation in Modelyze](image)

The equation level contains the equations of each object as described in Section 3.3.1 and 3.3.2 implemented using the MultiBranch approach that is shown in Listing 9.

The modular level contains all the single pieces that builds up the physical robotic arm. This layer contains unique objects for each of the different parts used to build the physical robotic arm. This includes one object for each joint as well as one object for the links with settable length. This means that this level contains all the physical parameters for each part.

This level also implements a system for keeping track of the normal forces each component is subject to in order to properly calculate the friction. This is done by using the same flange objects that is used by the 2d-mechanical equations and builds an additive system with it. This system increases the normal force the closer to the fixation point you get by a set amount (decided by the mass of the previous components). The downside of this is that you can’t mix between objects in this level and the equations level as this will render the normal force system incomplete.

Some example implementation of such objects can be seen in Listing 10:

```plaintext
// 20 x 20 aluminum links
def Link20x20mm(L : Real, flangeA : Mechanical2D, flangeB : Mechanical2D) = {
  def m = 0.4*L;
  def lm = L/2.0;
  def J = m*L*L/12.0; // Moment of inertia @ com
}
```
BuildNormalForce(m, flangeA, flangeB);
ArmLink(L, Lm, J, m, flangeA, flangeB);
}
// DCX26L with gear ratio 150
def JointType1(u : Signal, flangeA : Mechanical2D, flangeB : Mechanical2D) = {
def K = 0.0214; def R = 0.74+0.33; def L = 0.129e-3;
def n = 150.0; def eta = 0.75;
def m = 1.4; // Mass of the entire joint
def JA = 0.035; // Moment of inertia around flangeA
def JB = 0.041756; // Moment of inertia around flangeB
def offset_A = 0.02; def offset_B = 0.057;
def Fn : Signal;
def frictionFunction(Tf : Signal, w : Signal) -> Equations = {
  LinearFriction(Tf,w,0.041679 + 0.0002*Fn);
}; // Friction function
BuildNormalForce(m, flangeA, flangeB);
ReadNormalForce(flangeB, Fn);
DCmotorJoint(u, K, R, L, n, eta, JA, JB, m, offset_A, offset_B, frictionFunction, flangeA, flangeB);
}
Listing 10: Some example implementation of objects in the modular level

The highest level is the user level which is the code the user writes in order to build the modular system. Example configuration code can be seen in Listing 11:
def mainSys() = {
def f1, f2, f3, f4, f5 : Mechanical2D;
def sth1, sth2 : Signal;
Fix(f1, f5);
JointType1(24.0, f1, f2);
Link20x20mm(0.5, f2, f3);
JointType1(0.0, f3, f4);
Link20x20mm(0.5, f4, f5);
RotSensor(f2, sth1);
RotSensor(f4, sth2);
probe("th_1") = sth1;
probe("th_2") = sth2;
}
Listing 11: Some example implementation of a modular robotic arm

3.6 Model Verification

In order to verify that the models are plausible and not completely wrong some simple tests can be performed.

In order to test the arm links one simple way is to compare multiple connected links with an equivalent single link. So one 1 meter link should behave exactly
the same as two 0.5 meter links connected together. See Listing 12 for such an
applied verification test:

```python
def mainSys () = {
    // Single Arm
    def f1 , f2 , f3 , f4 , f5 : Mechanical2D ;
    // Modular arm
    def g1 , g2 , g3 , g4 , g5 , g6 , g7 : Mechanical2D ;
    // Signals
    def s1f , s2f , s1g , s2g : Signal ;
    // Single Arm
    ArmFixed ( f1 ) ;
    SpringJoint ( 0.0 , 0.0 , f1 , f2 ) ;
    ArmLink ( 0.5 , f2 , f3 ) ;
    ConstantTorqueJoint ( 2.0 , 0.5 , f3 , f4 ) ;
    ArmLink ( 1.0 , f4 , f5 ) ;
    RateSensor ( f2 , s1f ) ;
    RateSensor ( f4 , s2f ) ;
    // Modular Arm
    ArmFixed ( g1 ) ;
    SpringJoint ( 0.0 , 0.0 , g1 , g2 ) ;
    ArmLink ( 0.25 , g2 , g3 ) ;
    ArmLink ( 0.25 , g3 , g4 ) ;
    ConstantTorqueJoint ( 2.0 , 0.5 , g4 , g5 ) ;
    ArmLink ( 0.5 , g5 , g6 ) ;
    ArmLink ( 0.5 , g6 , g7 ) ;
    RateSensor ( g2 , s1g ) ;
    RateSensor ( g5 , s2g ) ;
    probe ( "Single_1" ) = s1f ;
    probe ( "Single_2" ) = s2f ;
    probe ( "Modular_1" ) = s1g ;
    probe ( "Modular_2" ) = s2g ;
}
```

Listing 12: Verifying the arm link object

Performing a simulation confirms that this is the case.

It’s also necessary to verify proper functionality of the friction model. To do
this a simple model that includes a crossover from positive to negative angular
velocity to check if it behaves properly.

The result of that simulation can be seen in Figure 18:
Simulation using the exponential friction model

Figure 18: Simulation to verify the exponential friction model

Plotting the friction torque versus the angular velocity (for $t > 2$) results in:

Friction model verification

Figure 19: Plotting friction torque versus angular velocity in Figure 18

Here we can see the same overall layout as in Figure 16, albeit with the parameters $c_1$ and $c_2$ in (20) set to relatively low. The reason for this is that the solver timeouts when trying to simulate this type of friction when these parameters are too high, possibly because of difference in the speed of the dynamics close to zero. This limits the usefulness of this approach and a choice of whether or not this model should be used in favor of the simple linear model has to be taken.
4 Control

This section describes the design of the controller. It starts with the related work and overall system design and continues with the design of the controller, its implementation and results.

4.1 Related Work

Underlying theory behind basic control principles is used as a starting point for the control design [16] but the main inspiration is the Dynamics an Motion Control course at KTH [17]. This includes basic control structures and pole placement techniques to maximize performance.

Alternative approaches exists, for example using some fuzzy adaptive control to automatically find the best control parameters for specific control situations [18]. It’s also possible to apply the control to the entire robotic arm instead of just a joint by joint approach [19]. This has the advantage of removing errors caused by joint flexibility, assembly and manufacturing imprecisions but requires external sensors to acquire feedback.

4.2 Overall Control Design and System Modeling

The purpose of the controller is to allow the user to position each joint of the robot arm at chosen positions in an optimal way. The purpose of the control design is to design what voltages it should apply to the actuator in order for it to move to the defined position.

The controller should exhibit three main qualities; performance, predictability and flexibility.

Since the system has timing requirements to fulfill it is important that the controller is able manage this. A suboptimal control algorithm will require more of the hardware resulting in unnecessary over-dimensional actuators.

As a big part of this system revolves around doing accurate simulations of the physical system it’s important that it is predictable. That means suppressing unmodeled effects as much as possible to allow for best possible fidelity between the simulated and real system. Such effects include computation time delays in the microcontroller which can have some random properties. Another is limits in the power supply, if the controller requests more current than the power supply can provide deviations from the ideal model occurs. Some motor control circuits also includes internal current limiters adding another layer to this problem. The controller should thus try to suppress these effects in order to make the controller as predictable as possible.

Another quality needed is flexibility because of the modular approach. As the motor joints and their accompanied controller is meant to be used in several different ways the control algorithm should not be locked into a specific configuration. To ease the handling of the system the user should be able to program
the controller without having to rewrite the firmware of the controller. This quality has to be taken into account when implementing the controller.

4.2.1 Operating Point

As to keep the complexity of the system down each joint will have constant control parameters that will be programmed into the system at startup. This approach however has negative impacts on the performance because of how a robot arm works. For example, if the controller is optimized for a situation in which the arm is fully stretched out, having a high load inertia, it will behave suboptimal in situations where it’s folded, having a low load inertia and vice versa. A visualization of these two situations is shown in Figure 20.

![Figure 20: How different orientations of the arm affects the load inertia of the innermost joint](image)

So one has to decide the operating point around which the controller is designed. If the configuration of the robot arm is known the expected load inertia can be estimated and thus providing a working point. But since the load inertia of the arm is not constant one has to determine the operating point which will provide the best trade-off between control performance and model robustness.

For this control synthesis that working point is when the arm is fully stretched out with the joints locked, the position with the highest load inertia. The reasoning behind this is that if a controller is designed around a high inertia drives a system with low inertia the closed loop response will be about the same as with driving high inertia. The suboptimal part here is that the low inertia system could have a much faster response if the system was designed around it.

However, if a system designed around a low inertia tries to drive a system with high inertia the closed loop response will include overshoots as the controller underestimates how much voltage it should supply. Demonstrations and explanation of this effect is shown in Section 4.6.

4.3 System Modeling

In order to design the controller we need to know the dynamics of the system it is supposed to control. The dynamics of the system as well as the controller is represented in a Laplace notation.

The system in question is a DC-motor with parameters $R$ and $K$, connected to a gearbox with ratio $n$ and efficiency $\eta$ connected to the load inertia $J$ with the
linear friction $c$. The angle of the arm is $\theta$, the current in the motor is $i$ and the applied voltage is $u$. The inductance is neglected as is a reasonable assumption in all electro-mechanical systems [16].

This results in the equations:

\[ T = \eta \cdot n \cdot K \cdot i \]
\[ J \cdot \ddot{\theta} = T - c \cdot \dot{\theta} \quad (21) \]
\[ u = R \cdot i + K \cdot n \cdot \dot{\theta} \]

Combining these together gives:

\[ J \cdot \ddot{\theta} = \eta \cdot n \cdot K \left( \frac{u - K \cdot n \cdot \dot{\theta}}{R} \right) - c \cdot \dot{\theta} = \eta \cdot n \cdot K \frac{u}{R} - \eta \cdot n^2 \cdot K^2 \frac{\dot{\theta}}{R} - c \cdot \dot{\theta} \quad (22) \]

Converting this to a transfer function results in:

\[ J \cdot \theta(s) \cdot s^2 = \frac{\eta \cdot n \cdot K}{R} U(s) - \frac{\eta \cdot n^2 \cdot K^2}{R} \theta(s) \cdot s - c \cdot \dot{\theta}(s) \cdot s \Rightarrow \]
\[ \theta(s) = \frac{\eta \cdot n \cdot K}{R \cdot J \cdot s^2 + (\eta \cdot n^2 \cdot K^2 + R \cdot c) s} U(s) \quad (23) \]

This model is then used for the control synthesis for the controller.

### 4.4 Controller Design

This section describes the design of the continuous controller.

#### 4.4.1 Control Systems

The main operating concept behind all feedback control algorithms is that you read data from a sensor, compare it to a reference point and adjust the output accordingly. The act of designing a controller is designing the way it relates the reference, sensor data and output.

The most common way of doing this is error feedback where a system $G_s(s)$ is controlled by a controller $G_c(s)$ as seen in Figure 21:

![Figure 21: Error feedback control of a generic system](image)
The way this operates is by taking the error difference $e$ between reference point $r$ and the measured output $y$ and feeds it into a controller that outputs the control signal $u$ into the system $G(s)$.

The most common way of designing the control system $G_c(s)$ is by the PID, proportional, integral, derivative, controller and is constructed the following way [16]:

$$u(e(t)) = P \cdot e(t) + I \cdot \int e(t) dt + D \cdot \frac{de(t)}{dt}$$

(24)

The act of designing this controller is deciding the parameters $P$, $I$ and $D$.

There is other ways of designing the controller but they’re all fundamentally based on this PID structure.

A more flexible control system is the output feedback which is constructed as seen in Figure 22:

![Figure 22: Output feedback control of a generic system](image)

This system is also called a two degrees of freedom implementation as you got two controllers to design, increasing the flexibility.

A mathematical explanation of the difference between these systems using control theory can be formulated as follows:

Let’s use a polynomial representation of the systems $G_s(s)$, $G_c(s)$ etc and set that $G_s(s) = \frac{B(s)}{A(s)}$ and $G_c(s) = \frac{S(s)}{R(s)}$ for the error feedback. With this representation the closed loop response becomes:

$$y = \frac{B(s)}{A(s)} u = \frac{B(s) S(s)}{A(s) R(s)} (r - y) \Rightarrow$$

$$B(s) S(s) \quad A(s) R(s) y + y = \frac{B(s) S(s)}{A(s) R(s)} \Rightarrow$$

$$y = \frac{B(s) S(s)}{1 + \frac{B(s)}{A(s)} \frac{S(s)}{R(s)}} = \frac{B(s) S(s)}{A(s) R(s) + B(s) S(s)}$$

(25)

For the output feedback we set the polynomial representation of $G_{fb}(s) = \frac{S(s)}{R(s)}$
and $G_j(s) = \frac{T(s)}{R(s)}$. The closed loop response of this becomes:

$$
y = B(s) y = \frac{B(s)}{A(s)} \left( \frac{T(s)}{R(s)} r - \frac{S(s)}{R(s)} y \right) \Rightarrow 
\frac{B(s)}{A(s)} \frac{S(s)}{R(s)} y + y = \frac{B(s)}{A(s)} \frac{T(s)}{R(s)} r \Rightarrow 
\frac{y}{A(s)} = \frac{B(s) T(s)}{A(s) R(s) + B(s) S(s)} 
$$

(26)

The advantage thus becomes that with error feedback the choice of $S(s)$ effects both the poles and the zeros of the system while with the output feedback they can be placed separately. This gives more control of the closed loop system dynamics and is thus the approach used in this project.

### 4.4.2 Control Structures

The primary choice when designing a controller is the structure of the control system $G_c$. For a PD controller this becomes $G_c(s) = P + Ds$ and for a PID controller $G_c(s) = \frac{D s^2 + Ps + I}{s}$. As the robot arm will work in parallel to the ground no steady state error caused by gravity will be present. This means that a PD controller is sufficient.

However, when dealing with control systems it’s convenient that $G_c(s)$ is proper, which means has equal or more poles than zeros. This can be achieved by adding a low-pass filter to the PD controller which results in the following, more generalized, formula:

$$
G_c(s) = \frac{S(s)}{R(s)} = \frac{s_1 s + s_0}{r_0 + s} 
$$

(27)

The polynomial representation will be used later in the control system synthesis.

Another control system can be chosen if so desired. For an PID-controller with an added low pass filter the generalized control system becomes:

$$
G_c(s) = \frac{S(s)}{R(s)} = \frac{s_2 s^2 + s_1 s + s_0}{s (r_0 + s)} 
$$

(28)

For a PI-controller the control system becomes:

$$
G_c(s) = \frac{S(s)}{R(s)} = \frac{s_1 s + s_0}{s} 
$$

(29)

Because this system already is proper no low-pass filter is necessary.

Having decided on the control structure of $G_c$ one can start designing the controller.
4.4.3 Pole Placement Design

With the control structure $\frac{S(s)}{R(s)}$ decided the overall controller is designed using a pole placement technique [17].

Needed for this is the system equation:

$$G_s(s) = B(s) = \frac{g_0}{s^2 + g_1 s}$$

(30)

This is the minimal system realization of the transfer function in (23) on a generalized form. This formulation will be used throughout this control synthesis.

The closed loop poles, from (26), with the controller from (27) and system from (30) becomes:

$$A_{cl}(s) = A(s)R(s) + B(s)S(s) = s^3 + (g_1 + r_0)s^2 + (g_1 r_0 + g_0 s_1)s + g_0 s_0$$

(31)

As this is a third order system we need to decide on three poles on order to be able to place them.

Let the desired closed loop polynomial be: $A_d(s) = A_m(s)A_0(s)$ so that $\text{deg} \ (A_d) = \text{deg} \ (A_{cl})$ and $\text{deg} \ (A_m) = \text{deg} \ (A)$ which results in $\text{deg} \ (A_0) = \text{deg} \ (A_d) - \text{deg} \ (A_m)$. In this case $\text{deg} \ (A_d) = 3$, $\text{deg} \ (A_m) = 2$ and $\text{deg} \ (A_0) = 1$. The reasoning behind this formulation will become clear later.

The chosen poles are $\omega_1$, $\omega_2$ and $\omega_3$, which means that:

$$A_m(s) = (s - \omega_1) \cdot (s - \omega_2), A_0(s) = (s - \omega_3)$$

which means $A_d$ becomes:

$$A_d(s) = s^3 + (-\omega_1 - \omega_2 - \omega_3)s^2 + (\omega_3(\omega_1 + \omega_2) + \omega_1\omega_2)s - \omega_1\omega_2\omega_3$$

(32)

Now we can solve the resulting so called Diophantine equation that results from $A_d(s) = A_{cl}(s)$ with equations (30) and (32). This means that we take the coefficients for the different powers of $s$ and put them equal to each other which will construct a linear system of equations. With three unknowns ($s_1$, $s_0$ and $r_0$) and three equations (coefficients for $s^2$, $s^1$ and $s^0$) it is solvable and the result becomes:

$$r_0 = -g_1 - \omega_1 - \omega_2 - \omega_3$$

$$s_0 = -\frac{\omega_1\omega_2\omega_3}{g_0}$$

$$s_1 = \frac{g_1\omega_1 + g_1\omega_2 + g_1\omega_3 + \omega_1\omega_2 + \omega_1\omega_3 + \omega_2\omega_3 + g_1^2}{g_0}$$

(33)

When it comes to choosing $T(s)$ the following structure is chosen:

$$T(s) = t_0A_0(s)$$

(34)

With this formulation the closed loop response defined by (26) becomes:

$$G_{cl} = \frac{B(s)T(s)}{A(s)R(s) + B(s)S(s)} = \frac{B(s)t_0A_0(s)}{A_m(s)A_0(s)} = \frac{B(s)t_0}{A_m}$$

(35)
Because of that we chose to formulate \( A_c(s) = A_m(s)A_0(s) \) we use the output feedback structure to cancel out one of the poles with a zero. This results in a order reduction of the closed loop system. The constant \( t_0 \) works as a static gain and is chosen so that the closed loop dc-gain of the system from \( r \) to \( y \) is 1. That is:

\[
t_0 = \left. \frac{A_m(s)}{B(s)} \right|_{s=0} = \frac{A_m(0)}{B(0)} = \frac{\omega_1 \omega_2}{g_0}
\]  

(36)

This technique can be used similarly on any system or controller but the linear equation systems that results from the Diophantine equations can quickly grow very big. Symbolic math tools such as Maple or MuPAD helps a great deal when solving those systems of equations.

4.4.4 Choice of Poles

The choice of poles is how the dynamics of the closed loop system is chosen. The choice usually depends on the requirements of the systems which states something like "the system should go from point A to point B in x seconds". The poles are then chosen to fulfill this requirements.

Choosing to slow poles will make the systems slower than it needs to be. However it will make the system more gentle and easier for the hardware to control it. Choosing to quick poles will make the system faster than it needs to be, which may not be a bad thing. But it will require more of the hardware in order to retain it’s performance when converted to a discrete time controller.

In this project, the requirements was that it should be able to move 1 meter within 1 second. After some testing the poles \( \omega_1 = -10 + 1i \) and \( \omega_2 = -10 - 1i \) has been found to fulfill this criteria, almost independently of system configuration.

When it comes to the choice of the pole \( \omega_3 \), which gets canceled out by a zero, it shouldn’t matter as the closed loop system doesn’t depends on it. However it will affect the system if nonlinearities are introduced, for example the current limiting and Coulomb friction. Choosing it to slow in fast systems causes the controllers \( G_{ff} \) and \( G_{fb} \) to become unstable with positive poles. One way of choosing this so that it has minimal effects on the overall system performance is setting this pole equal to the open loop non-zero pole of the system \( G_s \). This approach also results in the interesting property of making \( T(s) = R(s) \), providing some opportunities for simplifications.

4.4.5 Current Limiting

In order to provide high predictability of the system in addition to being gentle on the hardware a current limiting system has been implemented in the controller. This system limits the output voltage the controller provides in order to keep the estimated current below a set limit.

This system relies on the fact that the total winding resistance \( R \), the back-emf constant \( K \) and the gear ratio \( n \) is known. It also requires the system to know
the angular velocity the motor currently is rotating with, $\omega$. This means that the voltage $u_{\text{max}}$ needed to achieve a current $i_{\text{max}}$ according to the dc-motor equations (21). Ignoring the inductance this can be estimated to:

$$u_{\text{max}} = K \cdot n \cdot \omega + R \cdot i_{\text{max}}$$  \hspace{1cm} (37)

Since the current, voltage and angular velocity can be negative the minimum voltage that can be applied in order not to violate the current limitation is:

$$u_{\text{min}} = K \cdot n \cdot \omega - R \cdot i_{\text{max}}$$  \hspace{1cm} (38)

The controller software then makes sure that the output voltage $u$ keeps within the boundaries defined by $u_{\text{min}} \leq u \leq u_{\text{max}}$.

4.5 Controller Implementation

This section describes the implementation of the controller on the hardware and Modelyze.

4.5.1 Discrete Time Implementation

Because the hardware deals in discrete time-steps we need to describe the continuous controller derived above to discrete time.

In continuous time the Laplace operator $s$ denotes derivatives while in discrete time transfer functions the operator $z$ denotes time delays in number of samples. So the transformation between these formulation depends on how you approximate the derivative in discrete time.

Some methods of performing this transformations includes [17]:

Forward Euler

$$s \cdot x = \frac{dx(t)}{dt} \approx \frac{x(t + T_s) - x(t)}{T_s} = \frac{z - 1}{T_s} x(t)$$  \hspace{1cm} (39)

Backward Euler

$$s \cdot x = \frac{dx(t)}{dt} \approx \frac{x(t) - x(t - T_s)}{T_s} = \frac{1 - z^{-1}}{T_s} x(t) = \frac{z - 1}{zT_s} x(t)$$  \hspace{1cm} (40)

Tustin approximation / Trapezoidal method

$$s \cdot x = \frac{dx(t)}{dt} \approx \frac{2}{T_s} \frac{x(t + T_s) - 1}{x(t) + 1} = \frac{2(z - 1)}{T_s(z + 1)} x(t)$$  \hspace{1cm} (41)

So in order to convert from a continuous time transfer function to a discrete time transfer function with the Tustin method the following can be done:

$$\frac{S(s)}{R(s)} = \frac{s_1 s + s_0}{r_0 + s} \Rightarrow \frac{S(z)}{R(z)} = \frac{s_1 \left(\frac{2(z - 1)}{T_s(z + 1)}\right) + s_0}{r_0 + \left(\frac{2(z - 1)}{T_s(z + 1)}\right)} = \frac{d_1 z + d_0}{z + f_0}$$  \hspace{1cm} (42)
The same approach can be taken for the feed forward part:

\[
\frac{T(z)}{R(z)} = \ldots = \frac{c_1 z + c_0}{z + f_0}
\]

The constants in these two formulas becomes:

\[
\begin{align*}
d_1 &= \frac{(2s_1 + T_s s_0)}{\sigma_1} \\
d_0 &= \frac{(T_s s_0 - 2s_1)}{\sigma_1} \\
c_1 &= \frac{(2t_0 - T_s t_0 \omega_3)}{\sigma_1} \\
c_0 &= -\frac{(2t_0 + T_s t_0 \omega_3)}{\sigma_1} \\
f_0 &= \frac{(T_s r_0 - 2)}{\sigma_1} \\
\sigma_1 &= T_s r_0 + 2
\end{align*}
\]

The same method can be applied to any approximation technique.

There’s also another way of attaining these formulas which involves matching the poles and zeros of the continuous system with the ones in the discrete system [20]. Since the relationship between the continuous poles/zeros and the discrete ones is:

\[
z_p = e^{s_p T_s}
\]

We can convert all the zeros and poles in the continuous system to zeros and poles in the discrete system, resulting in a discrete transfer function. In addition to this a gain has to be computed so that the continuous and discrete system has the same dc-gain.

One of the most important specifications to decide on when implementing in discrete time is the sample time used when controlling the system. Having it too low will impact the performance and setting it too high will require more of the hardware but is preferable as long as the hardware can support it.

Another downside of using high sample times is that if the sensor is an encoder it will impact the resolution of its derivative. Since the encoder quantize the position signal with a fixed step length its derivate will be how many of those steps that has been taken since the last sample. This means that for faster sample times the controller will register fewer steps from the encoder which will therefore lose resolution for the derivate.

Mathematically this resolution can be described as:

\[
\omega_{res} = \frac{r_e}{T_s}
\]

Where \(\omega_{res}\) is the resolution of the rotational speed estimation, \(T_s\) the sample time and \(r_e\) is the resolution of the encoder. For a quadrature encoder with its set pulses per revolution, \(ppm\), attached directly to the motor axis the speed estimation resolution of the output shaft on the other side of a gearbox with ratio \(n\) the value of \(r_e\) becomes:

\[
r_e = \frac{2\pi}{4 \cdot ppm \cdot n}
\]

So for a example, a sample time of 0.01 seconds, an encoder resolution of 128 pulses per revolution and gear ratio of 132 gives a resolution of 0.0093 rad/s.
A common rule of thumb is choosing a sampling time is to choose one that is around 10 - 30 times faster than the fastest pole in the system. So if that pole is $|w_{\text{fastest}}| = 10$ that number becomes: 

$$\left\{ \frac{2\pi}{30 \cdot |w_{\text{fastest}}|} \right\} \approx 0.021 < T_s < \left\{ \frac{2\pi}{10 \cdot |w_{\text{fastest}}|} \right\} \approx 0.063$$

which corresponds to sampling frequencies between 16 and 48 Hz.

### 4.5.2 Software Performance

One often overlooked aspect when doing hardware implementation is the computation time needed for the controller to calculate the control signal. This means that there’ll be a time delay between sampling and actuation as seen in Figure 23.

![Figure 23: The time delay between the sampling of the sensor signal and the actuation](image)

This computation time is not necessarily constant but can vary depending on the software. For example encoder interrupts can prolong this computation time.

So in order to properly simulate this system you need to know this computation time and include it in the simulation. However this is not 100% accurate because of the inherent randomness of this property.

One way to solve this is instead of doing the actuation as the last thing in each cycle the actuation is instead postponed until the start of the next sample. This means that the computation delay is a constant value of one sample time which can be consistently simulated as seen in Figure 24:

![Figure 24: Delaying the actuation to the beginning of the next sample to improve predictability](image)

This will affect the performance of the system since we introduce an additional time delay. But since this project is about simulation of a mechanical system...
4.5.3 Software Implementation

As the formulation in (42) and (43) requires you to know the samples ahead of the current sample it needs to be rewritten in order to be able to be implemented. This can be done by extending the fraction by the inverted order of the system, in this case $z^{-1}$. This gives:

$$\frac{S(z)}{R(z)} = \frac{d_1 + d_0 z^{-1}}{1 + f_0 z^{-1}}$$

$$\frac{T(z)}{R(z)} = \frac{c_1 + c_0 z^{-1}}{1 + f_0 z^{-1}}$$

This means that for each sample we use values from the sample before that as inputs to the controller equations.

In practice, for the $n$:th sample we want to compute $u(n)$. So for the systems in (48) the equations to calculate this becomes:

$$u_{fb}(n) = d_1 \cdot y(n) + d_0 \cdot y(n-1) - f_0 \cdot u_{fb}(n-1)$$

$$u_{ff}(n) = c_1 \cdot r(n) + c_0 \cdot r(n-1) - f_0 \cdot u_{ff}(n-1)$$

$$u(n) = u_{ff}(n) - u_{fb}(n)$$

Because we need the values for the last sample $n-1$ we need to store these values between samples.

Since we want the system to be as modular and flexible we don’t want to lock the system into one specific control structure. For example in a configuration where one of motors works in the vertical we might want a controller with an integrator to remove the steady state error. Or we might want to use a speed controller instead and it should not be required to rewrite the entire controller, you should just have to load in new control parameters.

Such an implementation can be described the following way:

$$\frac{S(z)}{R(z)} = \frac{d_{(n_d-1)} + d_{(n_d-2)} z^{-1} + \ldots + d_0 z^{-(n_d-1)}}{1 + f_{(n_f-1)} z^{-1} + \ldots + f_0 z^{-n_f}}$$

$$\frac{T(z)}{R(z)} = \frac{c_{(n_c-1)} + c_{(n_c-2)} z^{-1} + \ldots + c_0 z^{-(n_c-1)}}{1 + f_{(n_f-1)} z^{-1} + \ldots + f_0 z^{-n_f}}$$

Here values $n_d$, $n_c$ and $n_f$ denotes how many control parameters $S(z)$, $T(z)$ and $R(z)$ consists of each. The control parameters are stored in vectors so $d = [d_0, \ldots, d_{(n_d-1)}]$, $c = [c_0, \ldots, c_{(n_c-1)}]$ and $f = [f_0, \ldots, f_{(n_f-1)}]$. So setting $n_d = 2$, $n_c = 2$ and $n_f = 1$ results in (48).

This can then be implemented as C-code in the controller. One such implementation that also includes the current limiter is listed in Appendix B.

There are ways to improve the performance of this system. One thing that takes a lot of time is the floating point calculations. If high speed performance
is required the controller could be implemented using integers instead. This can be achieved by multiplying all variables by a big factor, say $10^6$, perform the calculations and then divide by the same number. This will sacrifice accuracy for computation time since integer operations is faster than floating point operations.

However, as long as the actuate-on-start-of-the-next-sample philosophy is used this extra computation time from the floating point approach won’t affect the system performance. Just as long as the controller can handle the update frequencies it should be okay.

### 4.5.4 Modelyze Implementation

As Modelyze does not at the moment of writing contain discontinuous functionality a continuous implementation has to be used. An implementation of the output feedback PD-controller can be seen in Listing 13:

```python
def PDcontroller(s1 : Real, s0 : Real, p1 : Real, p0 : Real, r0 : Real, ref : Signal, meas : Signal, uout : Signal) = {
    def ufb, uff : Signal;
    ufb' + r0*ufb = s1*meas' + s0*meas;
    uff' + r0*uff = p1*ref' + s0*ref;
    uout = uff - ufb;
}
```

Listing 13: Modelyze implementation of a continuous PD feedback controller

The parameters $s_1$ and $s_0$ are the parameters in the $S(s)$ polynomial, $p_1$ and $p_0$ the parameters in the $T(s)$ polynomial and $r_0$ the parameter in the $R(s)$ polynomial.

### 4.6 Results and Conclusion

So to check whether the controller work properly or not it is implemented and simulated using Simulink. This simulation model include everything described above, that is a discrete time implementation with one sample delay and the current limiting. The Simulink block models is listed in Appendix D

Something that has to be validated is the assumption that we should choose our operating point as the point with the highest inertia. So simulations is done in situations where the controller is designed around a high inertia driving a load inertia load and vice versa.

The controller is designed for a one second rise time for a step response of 1 rad/s.

For the first situation, designed around a high inertia driving a lower inertia results in the following:
Some performance is lost with higher rise times, as a result from the controller having a too high derivate part. This leads to an overdamped system which reduces performance.

For the opposite situation, designed around a low inertia and driving a high inertia the following results are achieved:

Here the opposite situation can be seen, with overshoot resulting from the system being underdamped as a result from the lower design inertia.

This behavior can also be visualized by constructing a root locus where we plot the continuous time roots of the closed loop as a function of the load inertia. This results in the plot seen in Figure 27:
When the load inertia goes below the design inertia we get fast low dampened
and a slow single root. The slow root dominates the system and keeps it from
oscillating, giving a slow step response without overshoot. For the opposite
situation, with higher load inertia, we have slow low dampened roots and one
fast. In this situation the low dampened roots dominates the system resulting
in overshoot and oscillations in the system.

So if you have to choose on how you implement the controller it is better to
design it around a high inertia. You get the same approximate rise time when
driving the system in non-ideal conditions (as seen in Figure 25b and Figure
26b) and you generally don’t want overshoot.
5 Physical Prototype Design

This section describes the design of the physical prototype. Its underlying methodology, and construction in different areas. It also describes the electronic and software design that is used to implement the controller and accompanied communication protocols.

5.1 Related Work

There has been a lot of work in the area of robotic arms.

Modularity/reconfigurability has been an active research area for some time and comes in different shapes. One type of reconfigurable robot is UBot [21] which is a cube like robot which can connect to other UBots to build bigger robots. More relevant to this project that comes in the form of a robotic arm is based on a space application [22]. This is a proper robotic arm with the intent on allowing for reconfigurability but is a tad bit more advanced than what this project is aiming for. Nevertheless its overall design has inspired this robotic arm. A more complete design process for this system is shown in [23].

Different approaches for connecting the elements together exists. In [24] such a connection is shown. But because we do not require of the robotic arm to reconfigure itself, it’s enough if the operator is able to do the reconfiguration, allowing for an easier interface.

As the target method for manufacturing the arms is 3D-printing that has to be taken into account when developing the arm. Most such implementations focus on rapid prototyping and remains relatively small [25]. This project instead focuses on designing bigger components with proper dc-motors and bearing.

The typical configuration of the robot arm resulting from this project is of the SCARA (Selective Compliance Articulated/Assembly Robot Arm) type. This means that each link is connected to the next by a planar coupling that allows for rotation in the x-y plane. A similar project with this configuration is the crowd-funded FLX.ARM [26] which aims to be a high precision low budget robotic arm. It’s a configurable SCARA type robot arm that can be used for tasks normally done by cartesian type robots with stepper motors, like 3D-printing and milling.

5.2 Design Methodology

The methodology followed is visualized in Figure 28:
The first step is doing a background study to see what has been done earlier in this area as well as collect the requirements that has to be followed. When that is done simulations are performed to see what hardware we need to successfully fulfill the requirements.

When the hardware is known the design process can start. The biggest part that needs to be designed is the joints that houses the dc-motors which are designed by using a CAD-program. These are then manufactured, in this case by using 3d-printing. The finished print is then evaluated to check whether or not it manages to fulfill more hardware dependent requirements. These include easiness of assembly and durability.

If it doesn’t fulfill any of these tests the design is changed and a new version is produced. As most of these properties are hard to simulate (for example how much tolerance is needed and how well can it handle applied forces and torques) an iterative process is needed.

When a satisfactory design is achieved the experiments can start, which include parameter extraction, model validation and implementation.

A similar approach are done with the electronics. When the hardware is decided upon electronic components that are capable of driving said hardware up to its requirements are chosen. With the overall topology decided upon the electronics are designed and evaluated, redesigning portions that doesn’t behave up to par.

For the software it is designed around the control implementation uphold by software structures based around the overall system topology. That includes taking care of the communication and reading sensors. The software is continuously worked upon during the entire design process, fixing bugs, optimizing and adding features.

### 5.3 Requirements

As usual when designing a system there are requirements to take into account. These requirements where provided:

- Be able to configure with 1 to 4 arms
- Be able to operate in 3 dimensions, but physical modeling only in 2 dimensions
- Be able to move a 1 kg object, 1 meter in less than 1 sec and 0.5 cm accuracy
• Be able to move an up to 100 g object, 1 meter in less than 0.5 sec with 0.2 cm accuracy

Each requirements affects different parts of the system.

The amount of arms that should be used depends on the hardware rigidity and the electronic interface. For example mechanical flex that occurs when connecting weight to the hardware components that can affect the performance and voltage drops over long connection wires.

The ability to be able to mount in 3 dimensions depends must be taken into consideration when designing the hardware,

The timing requirements is mainly aimed at the control algorithm as something to design against. However, the system has to be designed so that it can reach these times without violating the motor limits, thus the motor choice has to be done with this in mind.

Other more general requirements for different areas of the system that relates to the overall usability of the system can be stated as follows.

Overall hardware requirements:
• The finished robot arm should fit on a table, giving the nominal operating length around 1 m
• Different end effectors should be easy to mount on the arm and interface with the software

The modular design philosophy means that the following is aspects that has to be taken into account when designing the arm:
• The arm should be able to be reconfigured in a reasonable amount of time and effort
• It should be possible to mount the joints horizontally and vertically
• Its functionality should not depend on the degrees of freedom

To fulfill the last requirement the electronics are designed as a distributed system. This will encompass a master controller which handles the communication with a computer for loading trajectories and one slave for each motor joint on the arm that takes care of the individual motor control. The following is some requirements that needs to be taken into consideration when designing the distributed electronic system:
• Each node should be controlled by one micro-controller each
• Each micro-controller should handle motor control and position sensing
• The overall system should not depend on the number of degrees of freedom
• Each node should be able to connect to a data bus between them all
• The master controller should be able to send controller parameters and reference points on the data bus to all nodes as well as request info from the nodes
• The master controller should be able to communicate to a computer for loading trajectories and controller parameters
• The PCB design should be small enough to fit on the joints

5.4 Choice of Hardware

Before we can design the hardware we need to know what to design it around. This choice mainly revolves around what kind of actuators and sensors that should be used.

5.4.1 Motor Choice

The first choice when choosing the motors is the motor type. Here the choice landed on brushed DC-motors for several reasons. Their simplicity is one reason, they’re easy to control as they just require a DC voltage as input. They’re also easier to model with very easy describing functions. The alternative is a brushless motor which has better performance, but requires customized control circuitry to supply the alternating voltage required to drive the motors. They’re also more complex to model.

Normal procedure when choosing a motor as taught at KTH [27] is looking at the required work conditions and then chose a motor that is capable of running in these conditions. Such a work condition usually is a set torque and rotational speed or a repeating acceleration, constant speed, deceleration, rest pattern. This allows you to simulate the procedure with different motors and see which motors are capable of it without violating any restrictions. Most commonly this restriction is how much heat is generated which should be low enough not to overheat the motor during nominal operation.

However, it is hard to define such a repeating operation for a robot arm to design after, making it difficult to define the motor requirements from a performance stand point. So instead the motor choice is done against a single transient motion following one of the transient behaviors defined by the requirements. So the motor chosen is one that is able to do the required transient while not violating any maximum permissible values.

The motors considered when doing this choice is from the company Maxon Motors\(^1\) which supplies all kinds of customized motors. This means that you can pick custom motor, gearbox and encoder combinations and the manufacturer will put them together. The motors taken into consideration are from their DCX-series which focuses on robotic applications and is listed in Table 1:

\(^{1}\)http://www.maxonmotor.com/maxon/view/content/DCX-Detailsite
<table>
<thead>
<tr>
<th>Motor</th>
<th>DCX 22L</th>
<th>DCX26L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electrical Characteristics:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$R$</td>
<td>1.83 Ω</td>
<td>0.74 Ω</td>
</tr>
<tr>
<td>$L$</td>
<td>0.192 mH</td>
<td>0.129 mH</td>
</tr>
<tr>
<td>$K$</td>
<td>0.0229 Nm/A</td>
<td>0.0214 Nm/A</td>
</tr>
<tr>
<td>Max. output power:</td>
<td>$P_{\text{max}} = 47.8$ W</td>
<td>$P_{\text{max}} = 71.7$ W</td>
</tr>
<tr>
<td>Nominal current:</td>
<td>$i_n = 1.4$ A</td>
<td>$i_n = 2.76$ A</td>
</tr>
<tr>
<td>Gearbox</td>
<td>GPX22 3-stage</td>
<td>GPX32 3-stage</td>
</tr>
<tr>
<td>Gear Ratios:</td>
<td>$n = 62 - 231$</td>
<td>$n = 62 - 231$</td>
</tr>
<tr>
<td>Average backlash:</td>
<td>1.3°</td>
<td>1.7°</td>
</tr>
<tr>
<td>Max efficiency:</td>
<td>$\eta = 0.74$</td>
<td>$\eta = 0.75$</td>
</tr>
<tr>
<td>Max continuous power:</td>
<td>$P_c = 6$ W</td>
<td>$P_c = 25$ W</td>
</tr>
<tr>
<td>Max intermittent power:</td>
<td>$P_I = 8$ W</td>
<td>$P_I = 31$ W</td>
</tr>
<tr>
<td>Max continuous torque:</td>
<td>$i_c = 1.2$ Nm</td>
<td>$i_c = 5$ Nm</td>
</tr>
<tr>
<td>Max intermittent torque:</td>
<td>$i_I = 1.5$ Nm</td>
<td>$i_I = 6.25$ Nm</td>
</tr>
<tr>
<td>Encoder</td>
<td>ENX16 easy</td>
<td>ENX16 easy</td>
</tr>
<tr>
<td>Counts per turn:</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Total weight:</td>
<td>$m = 165$ g</td>
<td>$m = 412$ g</td>
</tr>
</tbody>
</table>

Table 1: The motors considered, the gear ratios comes in the following values: 62/83/103/111/138/150/172/186/231

A suitable motor would be a motor that fulfills the requirements, especially the ones where it should move an object within a set amount of time.

For the simulations, the control algorithm developed in Section 4 is used and tuned to theoretically fulfill the timing requirements. The built in current limitation is also used in order to not overload the motors.

The requirements say that it should move the payload 1 meter. The motor choice is done with the worst case scenario, this is that the center motor should move the payload 1 meter by itself. The geometry of this movement can be seen in Figure 29:

\[
\theta_{\text{step}} = 2 \cdot \arcsin \left( \frac{R_{\text{step}}}{2 \cdot L_{\text{arm}}} \right) \quad (51)
\]

For the simulation the arm the moment of inertia is estimated to consist of the
chosen type beam running the entire length of the arm, which has a weight of 0.4 kg/m. On this arm two other joints at lengths $\frac{1}{3}L_{\text{arm}}$ and $\frac{2}{3}L_{\text{arm}}$ from the center which is modeled as point masses with mass $m_j = 0.6 kg + m_{\text{motor}}$. A point mass $m_{\text{end}}$ is mounted at the end of the arm. The rotor/gearhead inertias are neglected. This results in the following value of $J$.

$$J_1 = \frac{0.4 \cdot L_{\text{arm}}}{3} + m_j \left( \frac{L_{\text{arm}}}{3} \right)^2 + m_j \left( \frac{2 \cdot L_{\text{arm}}}{3} \right)^2 + m_{\text{end}} \cdot L_{\text{arm}}^2$$

For the following simulations the length of the arm is set to be $L_{\text{arm}} = 1 m$.

The resulting simulations depict the transients for the output shaft angle as well as the torque provided by the motor. A suitable motor will be able to perform the required step without violating any maximum restrictions as seen in Table 1.

The first motor tested was the DCX22L with a gear ratio of 238. With the maximum current set to 3 amperes and sample rate to 100 Hz the following results are achieved:

![Arm angle vs. time for DCX22L](image1)

![Transmitted Power vs. time for DCX22L](image2)

![Current to the motor vs. time for DCX22L](image3)

![Generated torque vs. time for DCX22L](image4)

Figure 30: Results from the simulation for a DCX22L motor for a 1 meter movement for an 1 m long arm

As showed, it can handle the required speeds but the transmitted power greatly exceeds the maximum intermittent power the gearbox is able to handle. Same goes for the torque which is almost a magnitude higher than it has to be. Re-making the controller so it takes these restrictions into account heavily reduces the performance of the controller, violating the timing requirements.
Doing these simulations again for the DCX26L motor with a gear ratio of \( n = 150 \) results in the following results:

- **Time (seconds)**: 0, 0.5, 1, 1.5, 2
- **Angle [radians]**: 0, 0.2, 0.4, 0.6, 0.8, 1, 1.2
- **Arm angle vs. time for DCX26L**
- **Power [W]**: -15, -10, -5, 0, 5, 10, 15
- **Transmitted Power vs. time for DCX26L**
- **Current [A]**: -3, -2, -1, 0, 1, 2, 3
- **Current to the motor vs. time for DCX26L**
- **Torques [Nm]**: -8, -6, -4, -2, 0, 2, 4, 6
- **Generated torque vs. time for DCX26L**

![Graphs showing arm angle, transmitted power, current, and generated torque over time for DCX26L motor](image)

**Figure 31**: Results from the simulation for a DCX26L motor for a 1 meter movement for an 1 m long arm

Here it results in a similar response within the requirements of the system. Here both the resulting torque and power falls within the maximum ratings of the motors (although it required the maximum current to be set to 2.7 amps).

So in order to achieve the required results the DCX26L motor is a better choice. At least for the main center motor driving the bulk of the arm.

There’s also another timing requirement that states that the arm has to move a 0.1 kg object for 0.5 m. So it has to be tested to see whether or not the DCX26L motor can handle this situation as well. In this case we consider the simulated situation where the motor is situated on joint number two, so the length of the arm is reduced to \( L_{arm} = 0.6m \). Because of it’s placement it also only needs to move one other joint at the position \( \frac{1}{2}L_{arm} \) in addition to the end point mass \( m_{end} \) off 0.1 kg. The link inertias are the same as before. This results in the following load inertia:

\[
J_{1m} = \frac{0.4 \cdot L_{arm}^3}{3} + m_j \left( \frac{L_{arm}}{2} \right)^2 + m_{end} \cdot L_{arm}^2 \tag{53}
\]

Using the same DCX26L motor configuration as before but with a sampling rate of 300 Hz to account for the faster dynamics following results was achieved:
As can be seen, it handles this no problem at all.

However since the robot is supposed modular the DCX22L motor can perfectly serve as an actuator for the peripheral links. In those situations the load inertia will be small enough to not result in violation of the maximum ratings of the motor. This also has the upside of making the peripheral joints lighter, reducing the inertia the innermost motors has to move.

When it comes to accuracy there’s some points to take into account. One is the resolution of the sensor (which is calculated in Section 5.4.2). The main issue here is the backlash of the motor. To calculate the impact a set amount of backlash has on the accuracy the following equation has been derived:

\[
R_{\text{err}} = \frac{\text{backlash}}{2} \cdot \frac{\pi}{180} \cdot L_{\text{arm}}
\] (54)

The division by two comes from that the error results from the distance between the center point and end point of the backlash.

For the DCX26L this backlash is 1.7° which results in a radial play at the end of a 1 meter long beam of \( \approx 1.5 \) cm. This does not fulfill the precision requirement.

There are ways to counter this problem, it is, for example, possible to get the gearbox as a reduced backlash version, reducing the backlash to 1.0°. This results in an angular accuracy of 0.8 cm. For the 0.6 meter arm this value becomes 0.5 cm, also above the required accuracy for a 0.5 m movement.
This is still to high, but it is not a lot that can be done about it. Backlash is hard to get rid of without going over budget. There are types of drives which has next to no mechanical backlash (Harmonic Drive) that is used widely in modern robotic arms. However these are very over budget so we have to live with the backlash.

5.4.2 Position Feedback Sensor Choice

In order to provide position feedback a sensor needs to be attached to the motor that measures the rotation angle of the motor and feeds it back to the microcontroller. There is a few different sensor types where the two most common ones is potentiometers and encoders.

Potentiometers works by connecting the motor shaft to a variable resistor, allowing for the microcontroller to measure the voltage drop over it. The voltage drop will depend on shaft angle which allows it to be measured. An encoder on the other hand works by attaching a code wheel on the motor shaft, alternating between on and off depending on the shaft angle. The microcontroller then counts the amount of on-off cycles, providing an estimation off the position in encoder ticks.

Table 2 lists the pros ans cons of each sensor:

<table>
<thead>
<tr>
<th>Potentiometer</th>
<th>Pros:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Provides absolute angles</td>
</tr>
<tr>
<td></td>
<td>Cons:</td>
</tr>
<tr>
<td></td>
<td>• Susceptible to noise</td>
</tr>
<tr>
<td></td>
<td>• Will rollover when past a certain angle</td>
</tr>
<tr>
<td></td>
<td>• Hard to find good specialized sensors for motor control</td>
</tr>
<tr>
<td></td>
<td>• Introduces additional friction</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encoder</th>
<th>Pros:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Easy to find, most manufacturers provides optional encoders for their motors</td>
</tr>
<tr>
<td></td>
<td>Cons:</td>
</tr>
<tr>
<td></td>
<td>• Needs to be initialized with start angles upon startup</td>
</tr>
<tr>
<td></td>
<td>• Suffers from inherent quantization noise</td>
</tr>
<tr>
<td></td>
<td>• Requires more hardware to count the encoder ticks</td>
</tr>
</tbody>
</table>

Table 2: Listing the pros and cons of a potentiometer versus an encoder as a motor position sensor

With these points taken into account the choice of sensor type landed on an encoder which has a lot of advantages in front of a potentiometer. The most significant advantage is availability as encoders are easier to find than motor position specialized potentiometers.

When it comes to accuracy the encoder also comes out on top. Even though the encoder suffers quantization noise as a result of its inherent stepping operation
the potentiometer depends on the resolution of the analog to digital conversion. This becomes even more of an issue when taking the sensor placement into account. Commonly the sensor is placed on the motor shaft before the gearbox, providing much higher resolution than if it was to be placed on the output shaft. This can’t be done with a potentiometer as it will rollover because of its limited range. This means that an potentiometer mounted on the motor shaft will constantly rollover, removing the absolute angle measurement which is one of the big advantages the potentiometer has over the encoder.

The only major downside then is that the encoder is more hardware dependent as it is required to actively count the steps, for a potentiometer you just need to read the ADC every sample. Another drawback is that you need to initialize the start angle when starting up the board so it knows what step count corresponds to zero which adds some minor inconvenience when operating the system.

In order to properly specify the requirements this imposes on the hardware some encoder calculations has to be made.

The most common type of encoder is the quadrature type where the encoder output consists of two signals, commonly denoted as channel A and B. These are 90° out of phase with each other which allows for the microcontroller to deduce which way the encoder is turning. See Figure 33 for explanation:

![Diagram of quadrature encoder output signals CW and CCW](image)

Figure 33: The output signal of a quadrature encoder. The turning direction can be decided by determining which signal leads the other

The resolution of the encoder is measured in pulses per revolution, ppr, where one pulse is the period of one channel (as pictured in Figure 33). But the actual usable resolution is 4 times the ppr since we can count each transition on both A and B. So for an encoder with a set ppr mounted on the motor shaft, the angular resolution of the output shaft mounted on the other side of a gearbox with ratio \( n \) becomes:

\[
\theta_{res} = \frac{360^\circ}{4 \cdot \text{ppr} \cdot n}
\]

This is the smallest angle of the output shaft the encoder is able to measure.

Another interesting thing to take into account when choosing the encoder resolution is how well the hardware is able to handle the encoders. At high speeds
it is possible for the controller to miss steps as it gets flooded with encoder interrupts. So to check whether or not the controller will manage to handle the high speed scenarios the following calculation can be made:

\[ f_{\text{max}} = \frac{n_{\text{max}} \cdot 4 \cdot \text{ppr}}{60} \]  

(56)

Where \( f_{\text{max}} \) is the maximum frequency of the encoder the controller has to handle when the motor is running at its maximum rpm \( n_{\text{max}} \).

The major choice then is the resolution of the encoder, if it’s too high then it will overload the controller with interrupts, missing steps. If it’s too low the angular resolution will be too low, impacting performance.

In this application the motor controller is a 32 MHz PIC24F microcontroller, in the worst case controlling a motor with a maximum speed of 12400 rpm and gear reduction of 138. Choosing an encoder resolution of 64 ppr gives \( f_{\text{max}} \approx 52.9 \) kHz. Choosing an encoder resolution of 128 gives this \( f_{\text{max}} \approx 105.8 \) kHz.

In order to test what maximum frequency the controller can handle a function generator was connected to one of the encoder pins on the microcontroller. The output of the function generator was a square wave with the frequency \( f_{\text{max}}/2 \) (because each square wave generates two interrupts on the microcontroller (rising and falling edge)). To measure the time taken by the encoder code a timed background process was run simultaneously. The result of this test can be seen in Table 3:

<table>
<thead>
<tr>
<th>ppr</th>
<th>( f_{\text{max}} ) [kHz]</th>
<th>processor time [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (reference)</td>
<td>0</td>
<td>100%</td>
</tr>
<tr>
<td>64</td>
<td>\approx 60</td>
<td>127%</td>
</tr>
<tr>
<td>128</td>
<td>\approx 106</td>
<td>160%</td>
</tr>
</tbody>
</table>

Table 3: Calculation time for a task when under the effect of encoder interrupts at a certain frequency

So when designing the controller you have to make sure that the processor time is enough for both the control loop and encoder counting.

The problem here lies within how much leftover time we have between the timer interrupts for the control loop. Preliminary analysis of the system indicates that the sampling frequency required will not exceed 1 kHz. Running this sampling frequency on the preliminary implementation of the controller results in the time utilization of the control loop to be around 45%. With an additional encoder interrupt generation of 106 kHz this utilization increases to around 75%.

So the controller can handle a ppr of 128 without any problem. The next possible step, if increased resolution is required, is a ppr of 256. This would result in an extra processor time of slightly more than double that of the one for 128 (if following the pattern from Table 3). This may result in a possibility of overflowing the processor with tasks, degrading performance.

This is not optimal, especially because the situations that require a higher encoder resolution is the ones with high sample rate (as explained in Section 4.5.1). With this in mind the final encoder resolution is set to 128.
When it comes to the accuracy it’s a good idea to look at the encoder resolution. With the ppr set to 128 and gear ratio of 150 the angular accuracy becomes \( \approx 0.005^\circ \) which for a 2 meter long arm corresponds to \( 0.005 \frac{\pi}{180} \approx 0.16 \text{ mm} \) radial accuracy. This lies well inside the required boundaries provided by the requirements.

5.5 The Robot Arm

The robotic arm consists of three parts; the base plate, the links and the joints. The base plate serves as the anchor on which the arm is connected. The links are the arm parts of the robotic arm. The joints connect the links together and provides actuation.

As a main goal with this project these joints and links should be reconfigurable, that is be able to mount in any way possible. The same goes for the baseplate, it should be possible to mount all links on the baseplate.

A example of an robotic arm configuration can be seen in Figure 34:

![Figure 34: An example configuration of the robotic arm, here with 3 degrees of freedom](image)

In this configuration it has three degrees of freedom, three motors in three joints. In addition to this is should also be possible to attach an end effector to the end of the arm to allow interaction with the surrounding environment.

5.5.1 The Links

The main part of the links is to provide reach to the arm. They provide the connections between the joints.

The links does not need to be especially advanced in order to provide the desired functionality. They need to be stiff enough to not flex, light enough to not unnecessarily load the joints to much and easy to interface with the joints.

The choice landed on aluminum struts from Bosch-Rexroth which are sturdy and relatively light aluminum rod that are easy interface with. The cross section of this strut is shown in Figure 35.
To fasten the links in the joints a 8 mm hole is drilled through the link, allowing it to be screwed to the joint. In addition the grooves on the links allow for additional equipment, such as electronics and hardware for the end effector, to be slided on and fastened on the link.

The length of the links comes within the range 0.1 - 0.5 meters as seen in Figure 36. This allows for constructing arms with the final length in the desired area with the possibility of constructing longer or shorter arms if desired.

As to keep the flexibility high only this type of links is used. But for example in applications where maximum performance is required thinner links can be used on the peripheral links on the arm to keep the inertia down. However as the focus on this system is to be modular the same links are used everywhere.

5.5.2 The Joints

The mechanical purpose behind the joints is provide actuation between two connectors. This means that they’re the component that houses the dc-motors where one connector is attached to the motor house and the other to the motor shaft. Bearings are also used to allow the two parts to rotate against each other and take the forces generated.

The main requirement behind the joints stems from their intended use, that they should be operable in any orientation. Other obvious requirements are overall durability, ease of assembly and be able to produce easily.
So to be able to handle this requirement we need bearings that are able to take both radial and axial forces. The types of bearings that can handle both these forces are Angular Contact Ball Bearings and Tapered Roller Bearings, see Figure 37:

![Figure 37: Two types of bearings that can take both axial and radial forces, angular contact ball bearings (left) and tapered roller bearings (right). Taken from the SKF website](image)

Performance wise the tapered roller bearing can handle higher loads while the angular contact ball bearing manage higher rotational speeds. But since the design criteria here is decided by the size of the motor (and the rotational speeds will never be near enough to the 13000 rpm limit for the tapered roller bearing) the bearings will be over-dimensioned for their intended use. So the choice of bearing instead depends on the size and availability of each.

However since these types of bearings can only take forces in one direction and we want the joint to work in any orientation it is necessary to use two of them in one joint.

The design developed for this joint can be seen in Figure 38:
The design philosophy behind the bearings is so that there’s always one bearing capable of absorbing the forces in every possible mounted direction. For example a downward force is applied to the lower part with the upper part fixed the upper bearing will take the axial forces and an upward force on the lower part results in the lower bearing taking them. If a downward force is applied to the upper part with the lower part fixed the lower bearing will take the axial forces and an upward force causes the upper bearing to take them.

Both bearing will take radial forces and torques.

A similar design is used for the housing of the DCX26L motor but scaled up and housing bigger bearings, a comparison of size between the two motor joints can be seen in Figure 39:
Figure 39: A size comparison of size between a joint equipped with a DCX26L dc-motor and a joint equipped with a DCX22L dc-motor

The assembly process of these joints can be visualized in Figure 40:

Figure 40: The assembly process of a DCX22L joint from different angles, going from right to left, the pcb holder is glued on
5.5.3 The Baseplate

The purpose of the baseplate is to provide something physical to connect the starting joint of the arm to. It should be a sturdy base to allow for proper operation of the arm.

As the intended use of the links provided by Bosch-Rexroth is to assemble workplace equipment they present the perfect building material for constructing the baseplate. Several different assemblies are possible where one is depicted in Figure 41:

![Figure 41: One example of a baseplate built using the aluminum link system](image)

The baseplate is fastened to the underlying surface through by any possible means, for example clamping or weights.

The sturdiness criteria depends on how this baseplate is built by the user, the assembly shown in Figure 41 is one possibility. However this assembly is not optimal as it’s somewhat weak, it can’t take the weight of multiple joints attached. Adding diagonal links can improve this but require relevant hardware.

5.5.4 Modularity

The modularity in this robotic arm configuration stems from the standardized connectors fitted on the joints. These connectors are designed around the 20x20 mm links with both horizontal and vertical holes in which the links can be fastened. This means that the start joint and end joint can be mounted rotated relative to each other in 90° steps as visualized in Figure 42 and Figure 43:
Figure 42: One possible configuration option where both joints work in the same plane.

Figure 43: One possible configuration option where both joints work in planes 90° relative to each other.

In addition to this, the length of the connecting link can be varied, allowing for a high amount of available configurations.
5.5.5 Manufacturing and Evaluation

The chosen method of manufacturing these joints is 3D-printing. This method allows for quick iterations where you can quickly go between design to having actual hardware that you can evaluate and then improve the design further.

Figure 44: The 3D-printer used for printing the joints of type Ultimaker²

Figure 45 shows a manufactured and assembled joint.

Figure 45: An assembled 3D-printed joint (without the attached control electronics)

When evaluating the hardware design a few things are considered. One important factor that aids assembly is tolerances, making elements that is in contact
with mechanical elements (like bearings and motors) slightly bigger/smaller. This makes sure they fit properly since the 3D-printer aren’t 100% exact and it should not require any excessive force to assemble the elements. So to get the best balance between ease of assembly and overall fit some iterations needs to be performed.

When testing the overall robustness of the construction some tests are performed on the assembled hardware. These includes applying weights to the joint, both directly on them at the end of attached links. This is done in several different orientations, simulating different possible configurations. If something breaks a new design is developed to counter these problems and the tests are redone.

An alternative approach is using some kind of FEM (Finite Element Method) to simulate forces applied on the joint. This makes the evaluation more accurate but requires more work to set up. Manual experimentations was thus chosen as it was deemed enough, if something brakes it’s still a trivial task to print a new component.

### 5.6 Software Design

The software has to fit together in the same way as the hardware fits together. That means work together over a central data bus.

Overall there’s two main parts that need their own software, the motor control chips and the master node.

In addition to these additional parts like end effectors and sensors should be able to be present in the system. However the software for those parts won’t be mentioned here.

#### 5.6.1 Distributed System Topology

The system is built as a distributed system with one central data bus. The main controller on the system is the master node which sends out instructions and receives data from the other nodes on the bus. The master controller also communicates with the operator whom can provide instructions for the robot arm to carry out.

There should also be possible to attach end effectors and sensors to the system for the master controller to interact with. This allows for interaction with the environment.

The overall system can be visualized in Figure 46:
When it comes to the communication protocol for the data bus there are a few options and requirements. It should not depend on the number of nodes, if a new node is connected it should not interfere with the already existing nodes. This usually means that the bus works on some sort of addressing system. Each node has an unique address on which the master can reach them.

The choice landed on I²C (Inter-Integrated Circuit). The normal operation of a I²C system includes one microcontroller which acts as a master and up to 127 nodes if it works in 7-bit address mode. Only the master can initiate communication with the nodes and can both write and request data from them.

Another option is CAN (Controller Area Network) which is the main protocol used in big distributed systems today. This acts more like a network on which the nodes can send information that other nodes can use. The reason I²C was chosen in favor of CAN is because I²C is supported on a wider range of microcontrollers, making it easier add components to the system that should interface with this bus. This is because most microcontrollers comes equipped with hardware able to work with I²C, making them very easy to include in the system.

### 5.6.2 The I²C Protocol

The I²C is explained in UM10204 [28].

For I²C the most common layout of the system is to have one master who controls the bus and several slaves which only responds to the master. Only the master controller can initiate communication and write or request data from an addressable slave.

Electrically the bus consists of two wires, one for data and one for the clock. When the master wants to initiate communication it pulls the data wire low to signal to the slaves that it is transmitting, this is called a start condition. It then sends the address of the device it wants to communicate with and a read/write bit. If the unit with that address exists it will acknowledge this by pulling the data high on a ninth clock cycle. If the address was acknowledged
the master then sends out the data, 9-bits at a time (8-bits for data, 1-bit for acknowledgement) by pulling the data high/low while pulsing the clock. When it’s done it issues a stop condition which is pulling the data high when the clock is high. A timing diagram of this can be seen in Figure 47:

![Timing Diagram](image)

Figure 47: The timing diagram of an I²C operation taken from the I²C-bus specification UM10204

The unit which is pulling these signal varies depending on if it’s a read or write operation.

For a write operation the master starts with sending out the address and the read/write bit set to 0. If the slave with that address acknowledge this the master will start to send out data which the slave will continue to acknowledge. This ends with a stop condition, see Figure 48 for visualization.

![Write Operation Diagram](image)

Figure 48: I²C write operation taken from the I²C-bus specification UM10204

For a read operation it starts with sending out the address with the read/write bit set to 1. If the slave exists it will acknowledge and start pushing data to the bus which the master will acknowledge. When the master stops to acknowledge the slave will stop transmitting and the master pulls the stop condition. See Figure 49 for visualization.

![Read Operation Diagram](image)

Figure 49: I²C read operation taken from the I²C-bus specification UM10204
It is also possible to combine these two by issuing a so called repeated start condition. This allows for the master to, for example, tell the slave what data it wants to read by transmitting to it first or read multiple slaves in quick succession. See Figure 50 for visualization:

![diagram](image)

Figure 50: I\textsuperscript{2}C combined operation using the repeated start condition taken from the I\textsuperscript{2}C-bus specification UM10204

These basics are then later used to set up communication protocols for the master to deliver and request data from the slaves, which here corresponds to the motor control nodes, end effectors and sensors.

The default mode of I\textsuperscript{2}C works with 7-bit addresses, this is to accommodate it to fit into one byte together with the read/write bit. There are extended 10-bit address protocol for situations that require a large number of systems on the bus. However, to keep a high compatibility to external sensors the 7-bit address protocol was chosen. This allows theoretically for 127 unique addresses but 16 of these are reserved (0000XXX & 1111XXX) for general calls etc, which results in 111 available addresses.

The common way to setup these addresses is to divide the binary address into two parts. The first four bits is the so called control code and is different for different type of devices. The last three bytes is the identifier or chip select bits which allows up to eight of the same device on the same bus. For example an I\textsuperscript{2}C EEPROM has the control code 1010 and the identifier bits can be chosen freely by connecting the three chip select pins to ground or logic high.

So when choosing the addresses for the nodes we don’t want them to accidentally collide with other sensors we might use on the end effector. A search for devices that might find use on different end-effectors reveals the control codes: 1101 for inertial measurement units such as gyroscopes and accelerometers, 1110 for thermometers and 1010 for EEPROM’s.

With this in mind the control code for the motor nodes is chosen to be 0010 which provides 8 different unique identifiers for the motor nodes. If needed these can be increased with an additional 8 by using the control code 0011. If a collision with an exotic sensor occurs it isn’t to difficult to change these codes as it just requires a reflash of the chip with different definitions.
5.6.3 Communication Protocols

In order to reliably communicate with each other a communication protocol needs to be defined. This allows the master send commands to the motor nodes reliably. Possible messages include setting the reference point of the motor nodes, turn them on or turn off, request motor angles, programming new controller variables etc. For the most critical messages a checksum is implemented to ensure that the message has been received properly.

The typical message begins with the address of the node followed by an identifier for the type of message and then the message content if needed. When requesting data a combined message is used, the first message is a write operation which defines the type of data that the master wants to receive. This is then followed by a read operation where the slave will provide the requested data.

The message identifiers is listed in Appendix C.

The most common messages, like disable/enable the motors consists of two bytes, the address (with the write bit set) and the identifier. The node can then take appropriate action depending on the message. An example of such a message can be seen in Figure 51:

```
Address+W ID
1 1
```

Figure 51: A basic message containing the address to the node and the identifier as listed in Table 4

For commands that require additional content, like setting the reference point, requires additional content (usually a 32-bit floating point number). Such a message can look like as listed in Figure 52:

```
Address+W ID Content
1 1 sizeof(float)
```

Figure 52: A basic message containing the address to the node and the identifier as listed in Table 4

When the master wants to requests data from a node it will first send a read mode command to the node so it knows what the master wants. Then the master will perform a read operation where the node will return the data defined by the read mode command. So to ensure that the correct parameter is read the master
should always write the requested identifier to the node before requesting the data.

For the more advanced commands, such as controller parameter programming, a special communication protocol has to be used. This protocol can be seen in Figure 53:

![Figure 53: The protocol for programming the controller with new control parameters](image)

The parameters passed by this message is the control parameters found in (50) as well as the sampling frequency $F_s$. The number $n_d$, $n_c$ and $n_f$ denotes the order of the the controller, allowing for any type of controller to be programmed to the node. This allows for flexibility as the node is not locked to a specific control structure.

To ensure that the node has received these control parameters correctly an 8-bit checksum is also included. This is a function of the entire message that the receiving and sending node calculate individually from the sent/received data. The receiving node compares its own calculated checksum value with the value sent from the master and if they are the same it should have been received correctly.

The checksum algorithm used here is inspired by the one used in the TCP protocol. This checksum calculating method for TCP is defined as [29]: "The checksum field is the 16-bit one’s complement of the one’s complement sum of all 16-bit words in the header and text." In this application however, we will only use an 8-bit checksum. Also, the only thing included in the checksum is the data part as we don’t have any significant header to talk about. The only thing resembling a header in a I²C package is the address and communication identifiers and if either of those are wrong the package won’t arrive at all.

The implementation of this is straight forward, first every byte sent is arithmetically summed together. If the resulting sum is bigger than 8-bits it folds repeatedly by $\text{sum} = (\text{sum} >> 8) + (\text{sum} & 0xFF)$ until the sum has an 8-bit length. The final checksum is then the NOT value of this, $\text{checksum} = \sim \text{sum}$.

The beauty of this approach is that it is very fast to validate the checksum. The receiver can just calculate the same checksum while also including the received value of the checksum. If the resulting value then is zero the message was received correctly.

5.6.4 Motor Node Layout

The motor node has three main tasks it needs to perform. The main task is the motor control, calculating the required voltage to the motors. The second task
is the encoder counting required to actively measure the motor angle. The third one is the communication, handling incoming and outgoing messages to/from the master controller.

All of these works with interrupts, which means that the relevant code executes only when certain events occurs. For the encoder interrupt this event is the logic level change of any of the encoder input pins. For the control loop interrupt it is a set periodic timer event and the communication interrupt occur at the end of each received/transmitted byte. A key design choice is then how to prioritize these interrupts. That is which task should be performed first in a situation where two interrupts occur simultaneously.

One way to assign these priorities is to grant the most important and time critical task the highest priority, which in this case this is the encoder. If the encoder is not allowed to handle its task immediately it might miss interrupts, skipping counts. After that the choice is then whether to give the control loop or the communications precedence over the other.

Because the timing of control loop is important it would make sense to give that priority over the communication interrupt. Visualizing the tasks with that priority can be done with a timing diagram as seen in Figure 54:

As one can see, with these priorities the control interrupts postpone the communication interrupts, delaying the communication. The slave then has to keep the clock line low in order to keep the master from sending more data (a procedure that’s called clock stretching). This means that the messages take longer time to transmit, lowering the rate which the master can send out commands. This may be a problem when it has to quickly send out commands to multiple nodes on the system.

But because of how the control algorithm is constructed it is not time dependent (since it waits until the start of the next sample to update the voltage). This means that it won’t affect the control performance to introduce any additional delay resulting from giving the communication interrupt higher priority than the control interrupt. At least as long as they’re reasonably short so the control loop has time to finish its calculations until the next sample. A timing diagram with these priorities can be seen in Figure 55.
In this case the I2C bus is (almost) fully utilized, being only slightly prolonged by the encoder interrupts. This ensures maximum bus utilization for maximum messages per second. The downside with this approach however is that if the communication interrupt takes too much time the performance of the system will be affected. And if something happens and the device hangs in communication interrupt it won’t return to the control loop, causing system malfunction.

Even though these are some downsides this second approach is chosen as it generates a better overall system performance. And in order to reduce the impact of eventual problems generated by this approach a watchdog reset timer is used to automatically reset the unit in the case of the code hanging.

Another issue with these nodes is that you need to be able to differentiate them from each other. They need individual unique addresses so that the master can direct messages to specific nodes. We also want to keep the simplicity in handling the nodes down. Which means that we don’t want to have node specific code containing different initialization values but just one piece of code the all nodes run on.

The solution here is then to use the internal EEPROM (Electrically Erasable Programmable Read Only Memory) of the microcontrollers to store node specific values. The EEPROM is some additional non-volatile memory of the same type as the program memory that can be used to store data between shutdowns. So it is possible to store individual data on the chip while using the same code for all the nodes. The only downside of this approach is that before you load the control code on the node you have to first load the identification parameters unto the chip.

The data stored in the EEPROM are primarily the identification code used to construct the I2C address. In addition to this the motor parameters are also stored there as the motor might differ between the different nodes. The default controller parameters are also stored there that the unit uses when nothing else is provided. A validation number is also stored that the controller reads first of all to ensure that the EEPROM is actually contains data.
5.6.5 Master Layout

The task of the master node is to send commands to the nodes and thereby control the arm. How it controls the arm is up to the user to decide and depends on the current task that the arm is supposed to perform. This is done through software libraries that the master includes in their project which allows it to connect to the data bus and send commands to the nodes.

A problem that arises when operating the system is for the master to know the id-numbers of the slaves and in which order they’re located. This is necessary as the master then knows to which I²C address it’s gonna send its commands to if it wants to reach the first joint etc.

For example, one such configuration can look like Figure 56:

![Figure 56: An example configuration of a master controlling joints with different identification numbers](image)

This is because the system should be as modular as possible. You should be able to assemble out of existing joints with set identification numbers.

The master node can deduce which id-numbers are available at the I²C bus by pinging the bus with messages with all known possible I²C addresses and see who answers. The problem then becomes to know in which order they’re located.

One way to do it is to make it software dependent so the user has to provide that information when programming the master. This means that the master has to be reprogrammed each time the configuration changes (which it’s likely to be anyway).

However, there are ways to automate this acquisition process of the node order. For example, adding an additional connection between the nodes with the connection as seen in Figure 57 can give such information:

![Figure 57: Using an additional connection between the nodes to automate the acquisition of the node order](image)

The master starts with polling all known id’s on the communication bus, giving
it knowledge about which nodes are present on the robot. It then asks the nodes for the voltage value of this extra connection acquired through the nodes ADC. The first node will feel a voltage of $V_1 = V$, the second one $V_2 = \frac{R_2}{R_2 + R_1}V_1$, the third one $V_3 = \frac{R_2}{R_2 + R_1}V_2$ etc. This allows the master to extract the order of the nodes by examining the order of their respective voltages.

However, in order to keep the complexity of the hardware down it was decided to keep this entirely in software.

5.7 Electronics

The purpose of the electronics is to act as the platform on which the system is controlled. It should also provide means for an operator to interact with the system through an interface.

The overall design of the electronic system should take the modularity of the system into consideration. A distributed system fits this criteria as this means that each part comes equipped with its own control electronics. This means that adding additional modules will not slow down the system, which would happen if one controller controlled everything.

The choice then rests on how to design these nodes, which should take care of position control of the motors. One possibility is using already existing motor controllers provided by the motor manufacturer. The problem with that is that these controllers are often bulky and will fit badly on the relatively compact joints.

However as this project requires absolute knowledge of what the controller is doing in order to simulate the system properly it was decided to develop our own motor control board. This includes both the electronics and the software on which the controller is implemented. These chips will be placed on each joint and connected to their respective motor in order to control their position.

5.7.1 The Electronic Platform

In order to get proper performance out of the system components has to be chosen so that it meets the requirements.

A big requirement in this case is that the control chip for the motor nodes has to be compact in order to fit on the joints. This means that most components should be surface mounted as this results in very compact systems. Another more practical requirement has to do with supply and company procurements.

For the nodes the main microcontroller needs to be both compact and have high performance. The choice landed on a PIC24F32KA301 microcontroller, a 16-bit controller in a compact 20-pin SSOP-20 package. With its compact size (7.4 x 5.2 mm) and clock rates of up to 32 MHz it fulfills both the size and performance requirements of the nodes.

For driving the motors an already existing chip was chosen in favor of designing an H-bridge on its own. This allows for a compact design as everything is already
assembled onto one chip. The choice here landed on the LMD18200 H-bridge motor driver chip. This chip can deliver 3 amperes continuously and has a built in current sensing capability in addition to a high temperature warning and automatic shutdown. Alternatives to this circuit would be the L298N circuit which is a dual H-bridge driver that can drive 2 motors with 2 ampere each or be connected together to drive a single motor with 4 amperes. It has the same package as the LMD18200 driver but more pins and require external free wheeling diodes which affects compactness.

In addition to this each motor node chip is also equipped with a connection to the encoder and output to the motors in order to drive them. It’s also equipped with connections to the data bus, power and a programming port.

The master controller in this system is a Uno32 development board by microchip. It is equipped with a 32-bit PIC microcontroller mounted on board with an Arduino form factor. It is possible to program this board with Arduino code in addition to proper code through the the microchip development environment MPLAB. This platform will be able to communicate with the motor nodes and any external sensor through the PC interface.
For connection between the different parts of the system a standardized cabling needs to be defined. Each node on the network is connected to the other nodes through six wires. Three wires supply different voltage levels (24 V, 5 V and 3.3 V) in order to remove the need for voltage regulators on the nodes. One wire is ground and the last two are data and clock lines for the data bus that works on a 3.3 V logic voltage.

The general layout of the cabling can be seen in Figure 60:

![Figure 60](image_url)

This layout of the cabling is chosen as it generates the highest amount of flexibility with the least amount of hassle with the cables. This is because when adding or removing nodes the additional cables can be added and removed as well, never running into the situation with loose unconnected cables.

However this has the downside that the next node in the chain relies on the previous node in order to have a connection to the master. So if one node breaks down the entire system might collapse. Additionally the performance can degrade since each connection is not perfect and induce losses, meaning that the outer nodes can notice voltage drops during high loads.

These downsides is neglected however under the assumption that the final system won’t reach a size where these losses will impact the system substantially.

The cable itself is a crimp connection with its design visualized in Figure 61:
These connectors should withstand up to 7 amperes of current, which will be enough for nominal operation.

5.7.2 Electrical Layout

The electrical circuit used can be seen in Appendix A.

The pcb layout, for this system is visualized in Figure 62.

One consideration done when designing the pcb is that the 24V, ground and motor power traces should be able to handle the currents the system will be subject to. As the traces can’t be too wide (not enough room) they are instead covered
with solder, artificially making them thicker so they can handle the required current. Additionally some wires can be added to reduce some resistance.

An assembled pcb is shown in Figure 63:

![An assembled motor node pcb](image)

Figure 63: An assembled motor node pcb

77
6 Parameter Extraction

In order to achieve high fidelity during simulation it is crucial to get as accurate parameters as possible. This section describes how the major parameters are extracted for use in simulation and the control system synthesis.

The acquired parameters is a combination of values from datasheets and experimental values. An alternative to this approach is to automatically estimating these parameters using estimation methods such as Kalman filters[30].

6.1 DC-motor

As the motor provider provides extensive motor data for their motors not a lot needs to be done in the way of experimental acquisition of their parameters. However is needs to be taken into account that the motor is not the only component in the circuit, the motor control chip also adds some resistance that has to be taken into account (around 0.33 Ω for the LMD18200 chip).

6.2 Links

The links are estimated to be thin rod rotating around its center of mass. The moment of inertia around the center of mass for such an object is: \( J = \frac{mL^2}{12} \) kgm\(^2\) [14]. According the the data sheet of the links their weight is \( m = 0.4 \cdot L \) kg. Combining this gives:

\[
J_{\text{link}} = \frac{0.4 \cdot L^3}{12} = \frac{L^3}{30}
\]

This isn’t 100% accurate but gives a good estimation. For high accuracy the width/height of the link has to be taken into account as these provide a (small) portion of the links moment of inertia. The screw holes also remove some moment of inertia from the links.

However, for this thesis these effects are ignored, but could be included in the future if accurate parameters are required.

6.3 Joints

With the main dc-motor parameters known it is of interest to know the physical parameters of the joints. The main parameters here is the friction parameters and the inertia. These depends entirely on the design of the joints and must thus be found out experimentally.

The friction torque at a certain input voltage \( u_{in} \) is extracted by observing the resulting rotational speed \( \omega_{max} \) (after all transient behaviors has died out). With the motor parameters known the friction torque \( T_f \) becomes:

\[
T_f = \frac{1}{R} (\eta \cdot n \cdot K \cdot u_{in} - \eta \cdot n^2 \cdot K^2 \cdot \omega_{max})
\]
Then in order to find the relationship between friction torque and rotational speed (58) is applied on several voltages. An automated way of doing this is depicted in Figure 64:

![Stair plot with increasing applied voltage](image)

Figure 64: Different voltages applied continuously and their resulting rotational speeds

Doing this results in the relationship between rotational speed and friction torque shown in Figure 65:

![The friction torque w.r.t. rotational speed](image)

Figure 65: Friction versus rotational speed

The friction models (19) (20) is then fitted to these values as depicted in Figure 66:
For the linear relationship a simple least square approach is used to get the best fit between (19) and the experimental data. But as (20) is nonlinear a numerical method has to be used in order to find the optimal fit. For this purpose the MATLAB function `lsqcurvefit` is used.

Another approach is to use the values obtained from the linear fit for the exponential fit (as \(c_c\) and \(c_d\) in (19) has the same functionality as \(c_1\) and \(c_3\) respectively in (20)) and set \(c_2\) to a relatively high value. This has the upside of neglecting inaccurate experimental data during slow speeds.

The friction is assumed symmetric, that is that the absolute friction force does not depend on the direction of motion.

In order to find the relationship between normal forces known weights are attached to the joints in order to increase the normal force. A relationship is then found between the normal forces and the friction in order to find the relationship between the it and the static and dynamic friction coefficients (as well as finding the "clean" normal force the joint has).

The next important property to extract is the inertia. This is something that’s hard to decide analytically as the bearings has multiple rotating parts.

The main idea behind the inertia extraction is trying to find the inertia that when simulated with best fits the transient behavior of the motor during the acceleration phase. Or mathematically the inertia which results in the smallest least-squares between the simulated and real system. This assumes every parameter about the motor is known (including the friction) to properly simulate it.

As some motors has inherently high torques the transients becomes so fast that it is hard to accurately fit the simulation. In order to slow it down some extra resistance is added in series with the motor which decreases the torque generated by the motor, resulting in slower acceleration. This allows for a more accurate fit since more samples are available from the acceleration phase.

A result of this is shown in Figure 67:
In order to reduce the effects of potential errors this is repeated for different voltages and the mean inertia is selected.

This is done two times, once with the top part fixed and the bottom part rotating and once with the bottom part fixed and the top part rotating to record the inertia of both parts.

Figure 67: Simulating with the optimal inertia vs. the recorded inertia
7 Results and Conclusion

7.1 Testing Methodology

To properly validate the 2D-library in Modelyze performs correctly it needs to be compared to the real world. But because some DSLs are incomplete, most notoriously the nonlinear friction modeling, the real system can’t be properly simulated which makes this hard to test.

The solution decided upon is to use Simulink as an middle platform since the Simulink toolbox allows for nonlinear modeling. So the method here is to first compare our acquired data from the real system with Simulink using the nonlinear models. This means that we can see how accurate that model is. Then we compare the Modelyze simulation with the Simulink simulation using linear models. If those simulations are the same we can deem the Modelyze model to work correctly.

This can only be used to verify models with only one degree of freedom, such as systems with a single joint, because it would be to much work and high chance of errors if implemented in Simulink. So to verify that the multi joint functionality works somewhat high fidelity a comparison was made between a physical platform with two joints, one motorized and one free, and a Modelyze simulation.

The input to this test is a sinusoid voltage with an amplitude and frequency which results in a periodic motion for both first link and the free running one. It should also be non-chaotic, that means that multiple physical runs should result in the same general output.

As the simulation is using a simplified model it will almost be guaranteed not to be very close to the actual acquired test data. However conclusions can be by looking at the relative amplitudes between them and the phase angle between the angles of the first and second link. If these lie close to each other the multi-joint model can be deemed to have high fidelity.

7.2 Testing Platform

As by the moment of testing only one motor had been delivered with it’s accompanied joint assembled no advanced multiple joint arm could be assembled and tested. The joint used for testing can be seen in Figure 68.
This is an early iteration of the hardware that was available at the time of testing.

To slow down the dynamics a bit a 18Ω resistor is connected and simulated in series with the motor. This makes it easier to extract the dynamics from the test runs as well as reducing the impact of unmodeled resistance in the power lines.

To test the accuracy of the 2d mechanical Modelyze library as simple multi-joint with two links connected together with a screw as depicted in Figure 69:

To measure the angle of the second link a gyro was used, mostly because it was the only sensor at hand but also that it demonstrates how you can incorporate external sensors on the I²C bus. The gyro measures the rotational speed and its signal can then be integrated to get the total rotational angle (which corresponds to the angle of the first link plus the angle of the second link).

A second configuration with the free joint first and the motorized joint second was considered but resulted in very high friction in the free joint because of the weight of the motorized joint. This caused the system to have very uninteresting dynamics and was thus discarded.
The power supply is a Laboratory DC Power Supply capable of providing up to 6 ampere of current at 24V. 5V and 3.3V is supplied by two switched power supplies mounted on an arduino shield mounted on the uno32 master controller. The entire base station is visualized in Figure 70:

![Figure 70: Main power supply station and master controller](image)

Additionally to this a breadboard mounted potentiometer is used to give the user some easy control over the system that can be used when testing the system.

### 7.3 Parameter Extraction

This section describes the results from the parameter extraction for the unknown parameters for the joint. This includes the inertia and the friction in addition to the frictions dependency of any added extra weight. This is done as described in Section 6.3

Because we need to know the friction to accurately determine the inertia it is determined first.

In order to provide data to fit the friction function to the joint was driven at several different voltages and the speed was recorded. The friction was calculated using (58) and resulted in the data shown in Figure 71:
In order to get a better fit a second order function as described by (59) is also fitted to the data. This allows a more accurate fit between data and estimation.

\[ T_f(\omega) = k_1 \cdot \text{sgn}(\omega) + k_2 \cdot \omega + k_3 \cdot \text{sgn}(\omega) \cdot \omega^2 \]  

(59)

Both the linear and second order fit is depicted in Figure 72:

As the second order fit best describes the friction it is that one which is used.

To determine the relationship between normal force and friction several data acquisitions with different weights attached was made. The way the weights where attached is depicted in Figure 73:
Figure 73: Adding weight to the joint in order to increase the normal force

The results of fitting the friction to measurements done with different weights attached is shown in Figure 74:

Figure 74: Different measured frictions and their respective linear fit for different added weight

Here we can observe that there’s a small dependency between the applied normal force and friction force, however it is small. The data also becomes more inaccurate, mostly because the extra mass in the setup in Figure 73 is hard to get lined up with the center of rotation, causing oscillations as it rotates. This
is why high speed data was ignored for these tests as they just became to noisy and unreliable

Because of its insignificance and noisy reading this relationship was thus ignored for the purpose of these validation tests.

However, the system supports modeling this relationship so there’s no technical limitations. But for implementing it proper more data needs to be collected with a better setup that works better.

However since Modelyze doesn’t currently support implementation of Coulomb friction and the exponential model causes timeouts since it can’t handle its stiff properties we need to find some pure dynamic friction to use as a placeholder until such functionality is implemented. This is done in two steps; first we fit the a linear curve to the input voltage versus rotational speed curve and then use (58) to get a linear curve in the rotational speed vs. friction torque. The slope of that function is found and that is used as the pure dynamic friction. The results of this can be seen in Figure 75:

![Input voltage vs angular velocity](a)

![Rotational speed vs friction torque](b)

Figure 75: Pure dynamic friction extracted from the data above

With the friction known the inertia can be determined. This is done with the technique described in Section 6.3 using several different input voltages to provide some different transients to fit the data to. The final inertia is a mean between all the inertias calculated for different voltages with eventual outliers ignored. The result of this extraction can be seen in Figure 76
This provides the inertia used for the lower part of the joint.

The same procedure is used on the upper part with a few modifications. Because the wires are connected to the upper part it isn’t possible for it to rotate around fully without tangling itself. So the transients has to be kept short and fast, slightly reducing the resolution of the final inertia value.

Another parameter that needs to be identified is the friction of the free screw link used in the multi-joint test.

To get data to fit the friction against the free running beam is rotated by hand and then let go, recording the transient behavior as it slows down. Then friction values are fitted by an least square algorithm and the friction chosen is the mean between all the fits.

The friction model in this case is assumed linear as it is that model type that will be used in Modelyze when simulating. However a Coulomb type friction can also be determined which generally resulted in a much better fit.

### 7.4 Modelyze models

The Modelyze model used when simulating the multi joint system shown in Figure 69 is shown in Listing 14:

```python
#include modulararmslib

def mainSys () =

def f1, f2, f3, f4, f5 : Mechanical2D;
def sth1, sth2 : Signal;
def u : Signal;

Fix(f1, f5);
JointType1(u, f1, f2);
Link20x20mm (0.3, f2, f3);
ScrewJoint(f3, f4);
Link20x20mm (0.2, f4, f5);
SineVoltage (12.0, 0.6, u);
```
RotSensor(f2, sth1);
RotSensor(f4, sth2);
probe("th_2") = sth2;
probe("th_1") = sth1;
}
def main = printsim(mainSys(), 0.01, 10.0)

Listing 14: Modelyze implementation of the multijoint system

The Modelyze code used for testing the control implementation is shown in Listing 15:

```python
include modulararmslib
def PI = 3.14
def mainSys() = {
def f1, f2, f3 : Mechanical2D;
def sth1, ref, u : Signal;

init ref (PI/2.0);
Fix(f1, f3);
JointType1(u, f1, f2);
Link20x20mm(0.3, f2, f3);
PDcontroller(0.822, 119.592, 0.822, 119.592, 12.0, ref, sth1, u);
ref = PI/2.0;
RotSensor(f2, sth1);
probe("th_1") = sth1;
probe("u") = u;
}
def main = printsim(mainSys(), 0.01, 2.0)
```

Listing 15: Modelyze implementation of the controlled system

If the model requires additional resistance the resistance of the motor object has to be manually changed in the modular level (that is in the DCX26L joint in Listing 10).

7.5 Results

To verify the inertia extraction algorithm a 0.3 meter link was attached to the end of the bottom joint and the expected inertia was compared to the measured one. To calculate the added inertia the system was assumed to look like Figure 77:

```
Figure 77: Calculating the extra inertia when adding a link to the joint
```
The inertia of the link, $J_{\text{link}}$ is assumed to be that of an rod rotating around it’s center of mass so that $J_{\text{link}} = \frac{mL^2}{12}$. The mass of the link is stated to be 0.4kg/m (assuming the loss of mass from the screw holes is negligible). This results in the following equation:

$$J = \frac{L_{\text{link}}^3}{30} + 0.4 \cdot L_{\text{link}} \left( x_{\text{offset}} + \frac{L_{\text{link}}}{2} \right)^2 \quad (60)$$

With $x_{\text{offset}}$ measured to 0.057m and with $L_{\text{link}} = 0.3$m this gives $J \approx 0.006$kgm$^2$.

Doing experimental inertia extraction provides the results shown in Figure 78:

![Figure 78: Fitted inertia to test data with an 0.3m link attached](image)

The difference between the calculated inertias in Figure 78 and 76 is $J_{\text{diff}} = 0.047655 - 0.041756 \approx 0.0059$kgm$^2$, which is very close to the expected value.

To check whether the fidelity of the Modelize model the real data is first compared to Simulink using the correct non-linear models as seen in Figure 79a. To verify that the Modelize simulations are correctly modeled the Simulink simulations using the less accurate linear model used by Modelize is compared to the Modelize simulation as seen in Figure 79b.
Figure 79: Comparing data to each other to verify that all the models work

As we can see, **Modelyze** and Simulink has a perfect fit, verifying that the **Modelyze** model is in fact working correctly.

The results of performing the run with the multi joint platform and comparing it to **Modelyze** simulations can be seen in Figure 80:

Figure 80: Comparing a simulated sinusoid motion with one free running joint with real data

The sine wave voltage input used is one with a amplitude of 12V and frequency of 0.6Hz. This was chosen as it was periodic and non-chaotic, which is what is needed from a measurement such as this.

As we can see the amplitudes of the signals is around the same as well as the phase between them.
Comparing the implemented controller with the Simulink simulations provides the results seen in Figure 81:

![Figure 81: Comparison of a step response with a slow controller between the real hardware platform and a Simulink simulation](image1)

As we can see that the fit between the two are good, confirming that the controller is working as expected.

The controller parameters used here are intentionally slow as to not trigger the current limitation logic. As this removes one of the biggest discontinuous elements a comparison between reality and Modelyze can be done accurately. The results from this comparison can be seen in Figure 82:

![Figure 82: Comparison of a step response with a slow controller between the real hardware platform and a Modelyze simulation](image2)
Here we can also see a good fit between them, meaning that the continuous implementation used by Modalyze gives a reasonable fidelity, at least with slow controllers.

Using faster controller to evaluate the effectiveness of the current limiter results in results in the following comparison figure as seen in Figure 83:

![Simulated vs. real control position](image)

Figure 83: Comparison of a step response with a fast controller between the real hardware platform and a Simulink simulation

Now the fit between the real and simulated response no longer correlates as well. As the only difference is that the current limiter should trigger during this step response it could be the cause of this deficiency.

A bit more info can be extracted when comparing the voltages between these two situations, which can be seen in Figure 84:
We can see a difference of the slope between the two, indicating that the implementation of the current limiter needs some work.

7.6 Discussion

The comparison in Figure 80 may not appear to fit very good and there’s a few reasons why. The biggest one is of course the simplified friction model used by Modelyze that results in inaccuracies in low speed situations. This can visualized just at the first half second in Figure 80 where $\theta_2$ momentarily dips down more in the simulation than it did in reality as it underestimates the friction.

Another source of error is that the friction in the screw joint may not be constant during the course of the test. As it is a screw it might tighten or loosen as it rotates, increasing and reducing the friction. This could be the explanation of the the difference between $\theta_2$ for the different data recordings. Another source of error that was observed was friction between the cables as the system rotates where one direction of rotation might experience higher friction than the other. The system was oriented to minimize this effect but it might still be present.

Also the mass and inertia of the gyro and its accompanied cables is not included in the Modelyze model as these was presumed to be negligible in comparison to other expected sources of error.

But as that test was just a proof of concept that demonstrates the capabilities of the multi joint simulation by Modelyze.

Some known sources of error is present in the Modelyze model, some are easy to fix, some are a bit harder. One is the weight and inertia of the links. Their values are taken from their data sheet but hasn’t been verified. The effect of the screw holes on weight and inertia are also not present in the model. The screws
meant to fasten the links to the joints are also not modeled, however they can easily be included in the inertia of the joint. They weren’t included in the tests as the tight fit of the connector didn’t allow for a screw to be inserted anyway.

A more complex inaccuracy involves the fact that inertia of an entire joint may not be the equal to the inertia of the top and bottom added together. This is because when these two parts rotate independently the balls bearings are rolling internally but when rotating as a unit the balls stay still. This theorized discrepancy has not been possible to test in practice as two motorized joints are required. One joint that drives a second joint that has its upper and lower parts fixed to each other which allows us to extract the total inertia of the second joint. This inertia is then compared to the second joints upper and lower part inertia added together and any difference is included as a separated internal inertia.

When it comes to the hardware the design has progressed from the one used in testing and is close to a final working prototype. However hardware tests involving strength and fatigue testing has not yet been performed so additional iterations might still be performed.

However, a design issue with the hardware is the connection between motor output axis and the lower part of the joint as it has two conflicting requirements. It needs to have a good fit that results in very little play as well as long term reliability as the plastic can start to flex after a while due to fatigue. It also needs to be possible to assemble without having to practice witchcraft. This excludes solutions where the connection is strengthened with a screw as it isn’t possible to reach the screw and tighten it when everything is assembled.

One solution that is considered is acquiring an already made metal connector, fasten it to the output axis with a screw connection and then attach the entire metal connector to the joint by vertical screw holes. This solves this problem but introduces some difficulties during assembly and design as well as finding a provider for the metal connector. Some different solutions needs to be experimented with in the future.

For the electrical system some additional improvements can be made. One problem experienced is noise on the voltage lines when driving the motors. This can in some cases have the result of resetting the microcontroller when turning on the motor (which could also be because the power supply can’t provide power fast enough). It can also result in the microcontroller picking up phantom encoder pulses, which of course is not good since it causes the encoder to drift.

This noise can be somewhat reduced by using a very high PWM-frequency (125kHz in this case) but that in turn reduces the resolution of the duty cycle (results in about 0.1 volt voltage resolution with a 24V supply voltage). So some research into how to reduce noise in these types of circuits needs to be done since adding multiple motor circuits to this system might impact the performance to much.
7.7 Conclusion

The hardware has been designed around the concept of modularity. This concept ensures flexibility in its use and a good platform for adding functionality. This is done both mechanically, through standardized connections, and electronically, through a distributed system. A control algorithm has also been developed and implemented allowing for feedback control of the arm.

The mathematical model allows for a similar modular approach that allows the user to build and simulate systems. Two separate libraries, one containing the equations of motion and one containing the parameters for the individual hardware joints that represents the real world objects.

From experiments it us concluded through visual comparison between data and simulation that the model corresponds to the reality in a realistic and accurate fashion with various discrepancies explainable. The fidelity will increase when the incomplete DSLs in Modelyze are completed and proper hardware joints that behaves in a predictable way are developed.

7.8 Future Work

There are a lot of ways to continue the work of this system.

A lot of things could be added in the DSL for describing physical systems in Modelyze. Improving the hybrid modeling DSL is a must if one wants the proper fidelity in the model. This allows it to properly model discontinuous events like friction, sampled controllers and backlash. Another possible implementation here is implementing subprocesses in c-code, allowing implemented controllers to be tested out. This also allows the user to test out algorithms and their implementation before using them in the real system, allowing him/her to test out their design.

Extending the Modelyze library to 3 dimensions is a logical next step. This does improve the ability to construct and simulate models but does require some sort of vector and matrix calculation DSL to be written in order to be implemented in a good manner. One challenge here is defining the semantics so this library becomes easy to use, for example defining directions in which joints can rotate. So it’s not as straight forward as it is in the case of the 2D library, it becomes more complex both to construct and to use.

Another interesting idea is using Modelyze in order to extract system parameters like inertia in order to automatically generate optimal controller parameters for each of the motors in the joint. This allows the user to quickly obtain optimized controllers for their application, improving usability. Technically, this can be done in two ways. You could look at the Modelyze script file (which describes the system configuration) and deduce the connected elements and their order from that. However this requires the extractor to know the individual elements and their respective values. Or you could try to deduce these parameters from the generated equations resulting from building the system of equations when compiling the script. This is not a trivial task and would require a bit of work to realize.
As the hardware system is designed around to be as modular as possible additional components are naturally easy to include. One such component that would enhance the usability of the system is a prismatic joint. This is a joint that deals in linear motion, allowing for example to links to move against each other. This adds some additional functionality to the mechanical system as well as providing an interesting component to model in **Modelyze**.

One improvement on the joints is adding one additional way to configure them not considered here. That is adding connectors to the top and bottom of a joint, allowing it to be used to rotate two links axially against each other. This increases the amount of possible configuration options.

The controller itself could use some additional functionality to allow for different usage situations. One such functionality is configuring it to work as a speed controller, allowing the master to set the speed it desires the individual joints to rotate at. This is useful for situations in which the master tries to control an outside control problem (like an inverted pendulum) by moving the arms around. In some such situations controlling the joint speeds is more useful than controlling their positions.
References


A Electric Schematic
This allows for any control structure of any order up to the generic set limit of six to be implemented. The control parameter and motor data structures looks like the following:
Listing 17: The control parameter and motor data structures

The motor data structure is unique for each node and adapted to the specific motor that node is equipped with. The same goes for the control parameters but with the difference that new parameters can be programmed into the system during runtime.
## Communication Identifiers

<table>
<thead>
<tr>
<th>ID</th>
<th>Message Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disable the motor</td>
</tr>
<tr>
<td>1</td>
<td>Enable the motor with the already set reference</td>
</tr>
<tr>
<td>2</td>
<td>Enable the motor and set the reference to its current position</td>
</tr>
<tr>
<td>3</td>
<td>Enable the motor and set the reference to 0 degrees</td>
</tr>
<tr>
<td>7</td>
<td>Enable the brake of the motor if disabled, does nothing if motor is enabled</td>
</tr>
<tr>
<td>8</td>
<td>The next 4 written bytes (float) sets the angular reference point</td>
</tr>
<tr>
<td>9</td>
<td>Set reference point to zero</td>
</tr>
<tr>
<td>14</td>
<td>Set direct drive voltage</td>
</tr>
<tr>
<td>16</td>
<td>Calibrates the encoder to the absolute angle 0 degrees</td>
</tr>
<tr>
<td>17</td>
<td>Calibrates the encoder at the angle provided by the following 4 bytes (float)</td>
</tr>
<tr>
<td>31</td>
<td>Sets the node status to absolute encoder position unknown</td>
</tr>
<tr>
<td>32</td>
<td>Programs the node with new controller parameters</td>
</tr>
<tr>
<td>33</td>
<td>Instructs the node to use its default controller parameters</td>
</tr>
<tr>
<td>128</td>
<td>The next read operations requests the motor angle</td>
</tr>
<tr>
<td>129</td>
<td>The next read operations requests the reference point of the motor</td>
</tr>
<tr>
<td>130</td>
<td>The next read operations request the current status of the node, possible returns are:</td>
</tr>
<tr>
<td></td>
<td>1 Enabled with no problems</td>
</tr>
<tr>
<td></td>
<td>2 Disabled with no problems</td>
</tr>
<tr>
<td></td>
<td>3 Encoder calibration needed, motor disabled</td>
</tr>
<tr>
<td>131</td>
<td>The next read operations requests the current status of the controller parameter programming. Possible returns are:</td>
</tr>
<tr>
<td></td>
<td>1 Nothing has been programmed</td>
</tr>
<tr>
<td></td>
<td>2 Latest controller programming successful, using the new settings</td>
</tr>
<tr>
<td></td>
<td>3 Latest controller programming failed, using old settings</td>
</tr>
<tr>
<td>132</td>
<td>The next read operation requests the applied voltage on the motor</td>
</tr>
<tr>
<td>133</td>
<td>The next read operation requests the current through the motor</td>
</tr>
<tr>
<td>136</td>
<td>WHO_AM_I , returns the current 7 bit address of the node</td>
</tr>
</tbody>
</table>

Table 4: The communication identifiers used by the devices on the I²C bus
D Simulink Block Diagrams for Simulation

Figure 85: Main

Figure 86: Main/DCmotor

Figure 87: Main/DCmotor/Friction
Figure 88: Main/DCmotor/Friction/Non linear karnop friction

Figure 89: Main/Discrete controller

Figure 90: Main/Discrete controller/Current limiter