Albert Danielsen
Ruben Winsjansen

Dexterous Gripper

Modular platform for robotic manipulations

Master's thesis in Cybernetics and Robotics
Supervisor: Jan Tommy Gravdahl
June 2019

NTNU
Norges teknisk-naturvitenskapelige universitet

Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

MSc thesis assignment

Name of the candidates:	Albert Danielsen and Ruben Winsjansen
Subject:	Engineering Cybernetics
Title:	Dexterous Gripper: modular platform for robotic manipulations

Background

Pick and place operations involving compliant objects is a challenging subject in robotics. Pliable materials, such as food, may be easily damaged during operations as their shape, weight and texture often differs. Historically, the approach has been to use different end effectors specialized for each gripping task, which is both time consuming and expensive. A proposed alternative is using novel sensor technology combined with a dexterous gripper to enable a wider array of tasks. Defined as the ability of hand or end effector to relocate objects in an arbitrary way according to a given task, dexterous robotics will be key in enabling such a shift.

In their dexterous robotics research, SINTEF is performing experiments with a GelSight sensor, an MIT-developed camera-based tactile sensor. To facilitate further research into the use of this sensor there is a need for a custom cost-effective robotic gripper with the capability of dexterous manipulation.

Tasks

In this assignment, the students will work along researchers at SINTEF Ocean and NMBU to design a dexterous gripper. The project's outcome should be an open source, cost-effective, platform for enabling dexterous gripping tasks using the aforementioned sensor technology. Its purpose is to serve as a platform for further research into dexterous manipulation with tactile sensing feedback.

To be handed in by: 3/6-2019

Jan Tommy Gravdahl

Jan Tommy Gravdahl
Professor, supervisor

Abstract

Dexterous manipulations is a field of research within robotics. In the development of new control algorithms and methods of making such systems, the need for a physical and electrical platform arises. This master thesis, in cooperation with SINTEF Ocean, presents a suggestion of design of a dexterous gripper. This system can prove to be a valuable platform for further research into robotic dexterity. The system comprises multiple fields of engineering, which are presented in this thesis. This includes the physical design of the mechanical system, the design of a printed circuit board, control theory for controlling the system, as well as the implementation of software for control.

The mechanical system which has been created, uses 3D printing technologies to reduce cost, together with a tendon system which enables bi-directional movement with one motor per joint. The system is fully back-drivable, and the princples used has potential to enable low elasticity actuation of finger joints.

The electronics that make up the system consist of motors, motor drivers, angle sensors and other electronics. To handle all of these components in a manageable way, and make the setup of the system more streamlined in a research setting, several printed circuit boards was designed and manufactured. An angle sensor board ensured for easier angle measurements in the system, an serial peripheral interface board made the setup of communication between controllers simpler, and a low level control board ensured for ease of control.

The successful actuation of a two degree of freedom system was made using two cascaded control loops, consisting of an inner and an outer loop. The inner loop applies the correct voltage phase in accordance to stator position to drive the motor, and thus the joints of the finger, clockwise or counterclockwise based on the commands from the outer loop. The outer loop gives commands based on angle sensor readings from the joints of the finger.

Application programming interface, graphical user interface, communication and control of the system has been implemented in software. A system capable of detecting and responding quickly to external events has been made, thanks to the use of C++, wired communication between nodes and integration of a real time operating system.

Through performance tests of the system, the conclusion is that it will be able to serve as a platform for further research into dexterous manipulations.

Sammendrag

Finmotorisk manipulasjon av objekter er et forskningsfelt innen robotikk. I den pågående utviklingen av nye teorier og kontrollalgoritmer oppstår behovet for en fysisk og elektrisk plattform å teste på. Denne masteroppgaven, i samarbeid med SINTEF Ocean, presenterer et forslag til design av en finmotorisk griper. Dette systemet kan vise seg å være en verdifull plattform for videre forskning i robotiske manipulasjonsoppgaver. Arbeidet som har inngått i oppgaven består av flere ulike fagområder. Dette inkluderer den fysiske utformingen av det mekaniske systemet, designet av et elektrisk styresystem, reguleringsteknikken bak styringen av systemet, og implementasjonen av programvare for kontroll.

Det mekaniske systemet som har blitt laget, har tatt i bruk 3D printer teknologi for å redusere kostnader. I tillegg har et menneskeinspirert senesystem muliggjort to-veis aktuering ved bruk av én motor. Systemets aktueringsprinsipp tillater reversibel kjøring, og prinsippene som er brukt i designet legger til rette for lav elastisitet.

Elektronikken som utgjør systemets motorstyresystem består av motorer, motordrivere og vinkelsensorer. For å håndtere alle disse komponentene, og gjøre oppsettet av systemet lett å bruke i eksperimenter, ble flere kretskort designet og produsert. Et vinkelsensorkort sørget for enklere vinkelmålinger i systemet, et kort ble laget for å legge til rette for master-slave kommunikasjon, og et kort for lavnivåkontroll muliggjorde kontroll av børsteløse likestrømsmotorer.

For å styre en finger, som utgjør to frihetsgrader, er kaskaderegulering implementert. Bestående av en indre og en ytre sløyfe. Den indre reguleringssløyfen sørger for at korrekt fase får riktig spenning ut ifra motorens statorposisjon, videre vil leddene i fingeren rotere.

Programmeringsgrensesnitt, brukergrensesnitt, kommunikasjons- og kontrollsystem er implementert i programvare. Til sammen utgjør dette et system som er i stand til å detektere og reagere hurtig på eksterne innvirkninger. Dette er mye takket være implementasjon av høyhastighets C++ kode, sanntidsoperativsystem og ledningsbasert kommunikasjon mellom sløyfene.

Gjennom ytelsestester konkluderes det med at systemet vil kunne tjene som en plattform for videre forskning i temaet finmotorisk manipulasjon.

Foreword

This master thesis is the final project for a masters degree in Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). This thesis is a continuation of two pre-projects from the fall of 2018: “Design of advanced robot finger” and “Low-level control of actuator for robotic finger”.

First we would like to thank our supervisor during this project, Jan Tommy Gravdahl. You have been a source of support in our meetings. We would also like to thank our supervisor for the pre-project, Øyvind Stavdahl. You helped set the course and focus, for what would be a very interesting, and challenging, project.

A mention also goes to SINTEF Ocean which deserve thanks for equipping us with an office, equipment, tools and funding. The task would have been a lot harder had it not been for free access to 3D printers, power supplies, an oscilloscope and a ventilated soldering station. These tools have We were also introduced to previous gripper designs as well as commercial alternatives used in previous research.

Our external advisor, Aleksander Lillienskiold, who gave us free reigns in paving a path for this project, also deserves thanks. You supplied us with papers, gave recommendations on which hardware and software to use and gave valuable input for how we should proceed. We hope that we have delivered on your expectations.

Last, but not least, we would like to thank our girlfriends, Yvonne and Torunn, you are truly amazing and we cannot believe you have put up with us through this project.

Contents

Acronyms

ADC	analog-to-digital converter. 50, 51
API	application programming interface. ii, 5, 60
BLDC	brushless direct current. xi, 31–34, 39, 44, 47, 56, 92
BOM	bill of materials. xiii, 45, 90–92
CAD	computer-aided design. 83
CAM	computer-aided manufacturing. 45
CLK	clock. 41
CS	chip select. 41
DC	direct current. 17, 33, 40
DIP	distal interphalangeal. 10, 91
DOF	degrees of freedom. 4, 11, 59
DPAK	decawat package. 91
EMF	electromotive force. 50
FET	field-effect transistor. 39, 49
FFC	flat flex cable. 47, 51–53, 91, 92
FOC	field oriented control. 33, 83
GPIO	general-purpose input/output. 40, 64
GUI	graphical user interface. ii, 42, 75
HAT	hardware attached on top. xii, xiii, 3, 28, 52, 83, 90–92

HTSSOP heatsink thin-shrink small-outline package. 39, 47, 91

I^{2}C inter-integrated circuit. vi, xii, 28, 40–44, 47, 52, 56, 58, 63, 64, 69, 74, 75, 83, 88

IC integrated circuit. 39, 44–47, 49, 50, 53

IDE integrated development environment. 40, 56

MCP metacarpophalangeal. 10, 11, 14–16, 21

MCU microcontroller unit. 47, 49

MISO master input slave output. 41

MOSI master output slave input. 41

OS operating system. 40

PCB printed circuit board. ii, 3, 39, 44–47, 50–53, 74, 83

PID proportional integral derivative. 38, 84

PIP proximal interphalangeal. 10, 11, 14–16, 21, 80

PLA polylactic acid. 82

PWM pulse-width modulation. 28, 33, 34, 39, 40, 42, 43, 47, 49, 52, 57, 58

QFN quad flat no-leads. 91

ROM read-only memory. 47

ROM range of movement. xi, 16, 21, 75

ROS robotic operating system. 59

RTC real-time computing. 54

RTOS real time operating system. 40, 56

SBC single board computer. 40

SCL serial clock. 42, 44, 52

SDA serial data. 42, 44, 52

SMD surface-mount device. 44–46, 49

SOIC small outline integrated circuit. 91

SOT small-outline transistor. 91

SPI serial peripheral interface. ii, xii, 28, 40–44, 47, 52, 55, 57, 64, 69, 74, 83, 88

TCP transmission control protocol. 60

Figures

List of Tables

Chapter 1

Introduction

Food production often involves some raw materials being lost or destroyed beyond commercial consumption during processing. This is a problem which may be reduced by taking more care during the handling of raw material. Normally this means training human staff to be more thoughtful towards the material or using more custom fitted tools. Though, with the advance of robotics in the food industry, there is a need for tailoring robotic systems to take care for the destructibility of materials and to maximize the production volume.

Robotics in food production have become more common since the turn of the century [IQBAL et al., 2017]. They see much use in the processes of handling, palletizing and packing food items. But they have yet to see the same boom as that in other industries like the automobile industry. Comparing the food industry and the automobile industry one could argue this is because of the materials being handled are much more pliable within the food sector, and bruised food does not sell well.

Dexterity is an aspect of robotics that engineers and scientists have been working towards for a long time, and major improvements have been seen in this field. In [Okamura et al., 2000], dexterity is defined as the ability to perform grasps and manipulations of objects in an arbitrary way given an arbitrary task. An example of a manipulation requiring a high degree of dexterity is the cooperation of several manipulators changing the configuration of an object while maintaining contact with the object throughout the task.

Robotics have historically utilized trivial control schemes in the food industry. This project, dubbed dexterous gripper, aims to build a robotic gripper with the ability to sense that it is gripping something and which has the ability to perform multiple dexterous grasps. Meaning that, given a trajectory or a point in space, it is be able to move along the planned trajectory or plan its own if it is not given. Additionally, when the end effector is exerting force on an object, it is able to sense and adjust the force applied. To facilitate all of this cybernetic methods have been used to control the system.

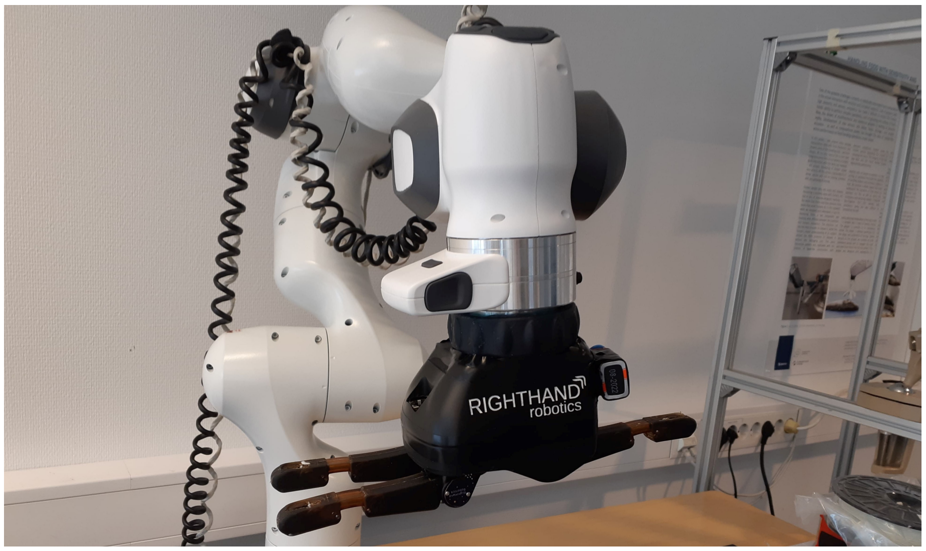


Figure 1.1: Industrial robot arm with Right Hand Robotics REFLEX gripper.

1.1 iProcess

SINTEF are working to bring more dexterous robots into the food sector. Through their projects that aim to create systems for gentler handling of caught fish, they are doing research into robots with aim to make them utilize more of the material from the animal. One of these projects is iProcess, using the idea that "flexible robotic automation technology will enable an increase in raw material utilization, reduce food loss and waste, and to cope with biological variation of raw material from fish to wheat" [Aursand, 2017].

The dexterous gripper is meant to be used within the iProcess system. This system consists of the same tools humans use for gripping: eyes, brain and hands, though in an artificial system. The eyes are replaced with cameras and computer vision technology, the brain is the central processor handling the algorithm calculating the optimal grip and the hand is an industrial robot arm with a dexterous end effector with the ability to gently handle pliable objects. One of the current industrial robot hands is seen in figure 1.1, used with a Right Hand Robotics gripper.

The dexterous end effector is where the work of this thesis comes in. In one of the projects performed at SINTEF Ocean the usage of a sensor developed at Massachusetts Institute of Technology, called GelSight, is researched. It is made up of a small camera with three-colored light that together measure deformation on a transparent gel. Through the use of vision software, the topographical map that the sensor gives out, different textures and measurements of deformation can be achieved [Li and Adelson, 2013]. Trough the use of this sensor the capability of sensing both force of deformation and the direction of force can be estimated [Li et al., 2014]. This sensor comes in a small enough package that it can be integrated onto end effectors to measure applied force.

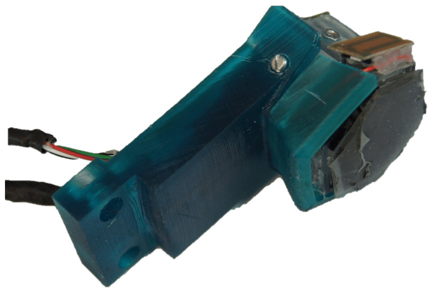


Figure 1.2: SINTEF-made GelSight sensor prototype.

1.2 Contributions

The work that this thesis encompasses has resulted in the creation of new mechanical designs, new electronics designs and software. The delivered contributions are mostly available in the project's online repository[1] and comprises the following:

- Dexterous finger: includes assembled parts, blueprints and schematics.
- Gearbox : includes assembled parts, blueprints and schematics.
- Kiyona control board: includes assembled PCBs and circuit schematics.
- Angle sensor board: includes assembled PCBs and circuit schematics.
- Raspberry Pi HAT: includes assembled PCB and circuit schematics.
- Low level control software: includes c++ scripts for low level controller.
- High level control software: includes c++ scripts for high level controller.
- Controller manager software: includes c++ scripts for software to tune and run several controllers at once.

[1]Project repository available at: `https://github.com/Bardie4/Dexterous`

Chapter 2

Specification

The purpose of the dexterous gripper is to provide a platform for further research into dexterous robotic manipulation. To create a system capable of this it is important to first specify in which way this should be done and at what performance the system should perform. Multiple explicit preferences was put forth in specifying the regarding to the general specifications of the system.

The specified need from SINTEF researchers is a gripper that consists of multiple separately controlled fingers, with multiple degrees of freedom (DOF), that is capable of grasping various objects and performing manipulations with them. Thus the system should be modular, meaning the design of one finger should allow for the scalability of assembling multiple into a gripper in an arbitrary configuration.

The system should be capable of different tasks, distancing itself from the approach of making special tools for every tasks. A gripper should have high enough level of control in each joint to provide the basis for distinct tasks. To exemplify, it should be capable of gripping as diverse objects as cabbages, tomatoes and small fish.

There are available commercial alternatives capable of dexterous manipulations, with some projects having large teams and lots of funding. Thereby some have made highly dexterous, but often expensive grippers. As a contrary this project consists of a small team with limited funding. Consequently, a focus in this project has been to keep the system simple and cost-effective.

Specific desires were expressed concerning the design of the mechanical system. There should be two actuated joints. The reasoning being that a third joint would be added later, containing the GelSight sensor. There was no desire to actuate this third joint, making it passive. So, for the purpose of performance tests that are a part of this thesis the design of the second link's form was left up to the authors.

As electrical motors generally inhibit high speed but low torque, it was agreed that some transmission system had to be made. Both to deliver a higher amount of torque to the joints of each finger link, as well as moving the motors away from each links, thus allowing for a slender design. Furthermore, this means that the transfer of power has to move through some medium. The wanted medium was a system of human inspired "tendons", or wires.

Back-driveability is the capability of a drive mechanism to not only transfer torque from motor to some output, but also to let torque transfer the other way. It was set as a priority to make the transfer of torque in the system back-driveable, as this is easily combinable with a tendon based transmission system.

There were also requests regarding how software should be designed. Most importantly it should focus on ease of use. More specifically an application programming interface was requested, to allow researchers to easily interface with the gripper from their own custom made software. Furthermore, the expressed desire for the use of ØMQ, an Ethernet communication protocol, and flatbuffers, a method of serializing data was made.

Research is being made into the correction of a robotic gripper during a slip event. If a gripper is fitted with a GelSight sensor, a software system can be built to detect that the gripper is losing its grasp on an object. Then measures can be taken to tighten the grasp, or even catch the object after it has slipped out. This, of course, sets high specifications in the speed of the algorithms. A supported frequency of 60 Hz, or 17 milliseconds response time, was expressed as a need for the system to perform under such conditions.

Another expressed is for the ability to mimic the human behavior of tapping fingers together within a certain time frame. This is simply wanted as a measure of the systems performance. If a finger could be able to change directions and drive an arbitrary angle in each direction multiple times a second it would be considered a success.

Lastly, there has been expressed a desire that the outcome of this project should be open source. That is, it should be freely available to access the drawing needed to assemble the physical system, the schematics needed to manufacture the electronic system and the required software to program the system. Thus anyone can pick up where this project leaves off.

Chapter 3

Mechanical design

The mechanical design of a robotic gripper is a large undertaking and can be a complex process. It requires a wide range of skills, that is in some cases foreign to cybernetics students. However, the process of designing the system from scratch opens up a unique opportunity. Normally, an engineer specialized in control theory is at the mercy of the engineers that designed the system. How easily the system controls may or may not have been a priority in their design. In this project, ease of control has taken priority in multiple cases.

3.1 Prior work

Some work on the mechanical part of this assignment had already been carried out prior to this thesis. [Winsjansen, 2018] investigated the dynamical impact of gears and tendons, reviewed gear alternatives, and designed a prototype for a pulley based gear system. [Danielsen, 2018] did research in gripper design, and designed the first version of a finger for the gripper, with concepts that are still in use in the final version.

3.2 Mechanical design specifications

The wanted outcome of this project was, in short, a gripper that had two actuated links per finger, was back-driveable, and easy to repair. The gripper should be able to pick up small food items like tomatoes, it should be flexible and suitable for many tasks, and not be specialist to a specific task. Furthermore, objects should be grasped with a pinching grip. Lastly, modularity of the finger design should be a main focus. So that if there was only time for one finger, the design would be scalable. With this principle, an arbitrary amount of fingers can be assembled to make up different gripper configurations.

3.3 Design choices

The specifications in section 3.2 still left room for some major design choices. For example, the question of whether to use tendons or not, or what type of gearbox to use. Many of these questions were at least partially answered in the pre-project leading up to this thesis, but not entirely set in stone. However, using tendons was decided already in the project phase because it allowed motors to be placed outside the finger, which reduces weight and enables a slimmer design.

3.3.1 Cybernetics and mechanical design

When in doubt, knowledge of control theory has been the guiding hand in design choices through this project. To get a sense of direction, it is helpful to first establish what an ideal system looks like. In an ideal system, controlling a joint would be as simple as controlling a motor. In such a system the relationship between joint and motor is static. Joint angle is proportional to the motor angle, and joint torque is proportional to motor torque.

In the pre-project leading up to this thesis, [Winsjansen, 2018] concluded that a gearbox was necessary to achieve reasonable torque capabilities without large and expensive motors. It was also established that gearboxes inevitably introduces unwanted properties, namely friction, backlash and elasticity. Choosing what type of gear to use, meant choosing between elasticity and backlash. Specifically, belt drives and harmonic drives have no backlash but non-negligible elasticity, while planetary gears and traditional spur gears have non-negligible backlash.

Tendons

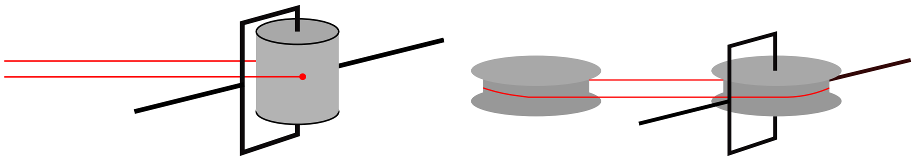

(a) Open ended configuration (b) Endless tendon.

Figure 3.1: Two basic tendon configurations.

Using the human finger as an example, it quickly becomes apparent how much the tendons themselves complicate a system. Two separate muscles are needed to enable movement in two directions. Most movements require a certain degree of cooperation between muscles. The tendons themselves are flexible, and thus introduce unwanted dynamical effects.

[Uyguroğlu and Demirel, 2006] describe two basic tendon configurations: the open ended, and the endless tendon configuration. Both concepts are shown in figure 3.1.

The endless tendon configuration has the advantage of only needing one motor for bidirectional movement. It is also inherently back-driveable, and is therefore an excellent candidate. The endless tendon configuration is in principle identical to a belt drive. According to [Höfler, 2018], belt drives are either friction locked, or uses toothed belts. In both cases, tension is required to keep the belt from slipping. The elasticity of the belt gives the belt drive a shock dampening quality, which is often advantageous. However, for this application, it was established that elasticity is a disadvantage. Efforts has been put into finding out if the belt actually needs to be elastic, but no concrete information was found. It is in the author's intuition that an elastic material is simply well suited to create traction because of its ability to deform and fit the surface of the pulley.

The open ended tendon configuration allows the wire to be properly fixed in both ends. Therefore, traction is no longer an issue, and a non elastic material can be used as a tendon. In this configuration, tension is used to prevent slack. An open ended configuration does not allow active actuation in both directions with only one motor.

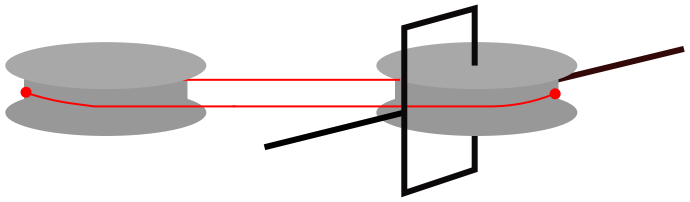


Figure 3.2: Combination of the open ended and endless tendon configuration.

During the pre-project, the decision was made to use a combination of the two configurations. An illustration of this concept can be seen in figure 3.2. Although it looks similar to the endless tendon configuration, the tendon is fixed to both pulleys, indicated by a red dot in the figure. This gives the system a limited range of motion. However, the traction problem is eliminated, which enables the use of a non elastic tendon. Similarly to the open ended system, tension is used to reduce slack in the tendon rather than to gain traction. The system is back-driveable, and can perform bidirectional movement with one motor. This configuration will be referred to as the limited loop configuration.

3.3.2 Pulley based gear system

The promising prospect of a tendon system with low elasticity and backlash inspired the idea of designing a pulley based gear system with the same principle. The simplest possible implementation of this is shown in figure 3.3, where the radius of the driving pulley is reduced to increase torque output to the link. The side effect if this change is a significantly reduced range of motion. The range can be increased by winding the string around the pulleys as shown in figure 3.4.

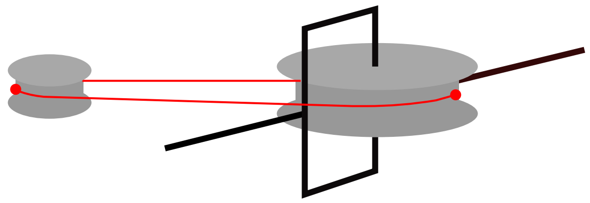


Figure 3.3: Tendon system with mechanical advantage.

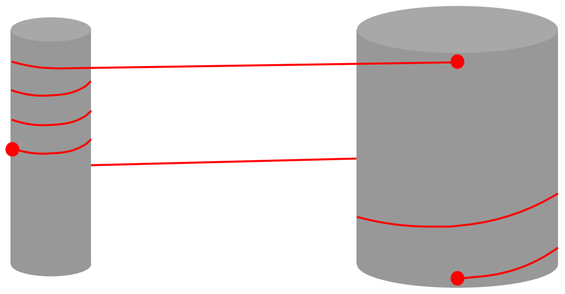


Figure 3.4: Pully based gear system with extended range.

3.3.3 Choosing a gear system

A prototype of the pulley based gearbox was designed during the pre-project phase, and plans were sent to the mechanical workshop at Gløshaugen NTNU. The design of the prototype is seen in figure 3.21. However, the prototype was not finished before the deadline of that project, so the choice of gear system was postponed. [Winsjansen, 2018] concluded that making a custom gearbox would likely be more expensive than buying a commercial one, because of the need for a mechanical workshop to build it. The system would therefore need to perform better than commercially available options to be justified, and was at that stage unlikely to be used.

However, by the start of this thesis, access was given to a 3D printer. It was discovered that its accuracy and quality of was sufficient to print the pulley system. This meant that everything except the motors, screws, electronics and bearings could be manufactured in-house with the push of a button. This was a major deciding factor. 3D printing makes the project more available, and lowers the threshold for others in the open source community to adopt the project. It also fits in to the modular design philosophy, where fingers can be added to the gripper as needed. It was therefore decided to move forward with the pulley based gearbox design. Further details on the gearbox system is found in section 3.5.

3.4 Design of finger

The human anatomy is an obvious place to draw inspiration from when designing a robotic gripper. Through countless years of evolution the human hands have proved to be a tool that has given the necessary advantage to become the top of the food chain. The human hand goes far beyond the specifications for the Dexterous gripper. It is capable of a large amount of gripping techniques, and is a proven multipurpose tool.

3.4.1 Finger phalanges

The human finger, with terminology as defined in figure 3.5, is made up of three links that protrude out of the palm. In order of increasing distance to the palm they are: the proximal phalanx, the middle phalanx and the distal phalanx. Which are connected respectively by the proximal interphalangeal (PIP) joint, the metacarpophalangeal (MCP) joint and the distal interphalangeal (DIP) joint. Each of the links are connected to tendons that when flexed or extended allow for movement and gripping. The dexterous gripper only has two joints. These will be referred to as the proximal and middle phalanx from this point on.

A focus of this project is facilitating experiments with the GelSight sensor, which is likely to be mounted on the gripper through a third non-actuated link. Therefore, the name "distal phalanx" is reserved for this hypothetical third link. Similarly the name, PIP joint, and MCP joint will be used to refer to the first and second joint of the gripper.

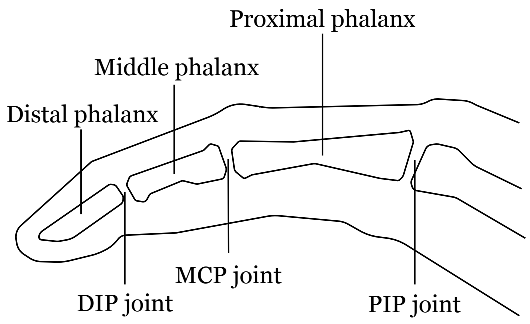


Figure 3.5: Anatomy of a human finger.

3.4.2 Evolution of design

Figures 3.6, 3.7 and 3.8 shows how the finger design progressed throughout the project. These three versions represent major design changes. There were however, many iterations in between to solve smaller problems.

The first version was an iteration of the finger design documented by [Danielsen, 2018], where an early version of the tendon system had been integrated. The middle phalanx was little more than a copy of the first phalanx at this point. The two joints added up to a length of 13 cm, with a base area of 3x3 cm, when not counting the sensors which added to the thickness at the joints. At this stage, solutions on how to attach the joints, as well as integration of the angle sensors were in place.

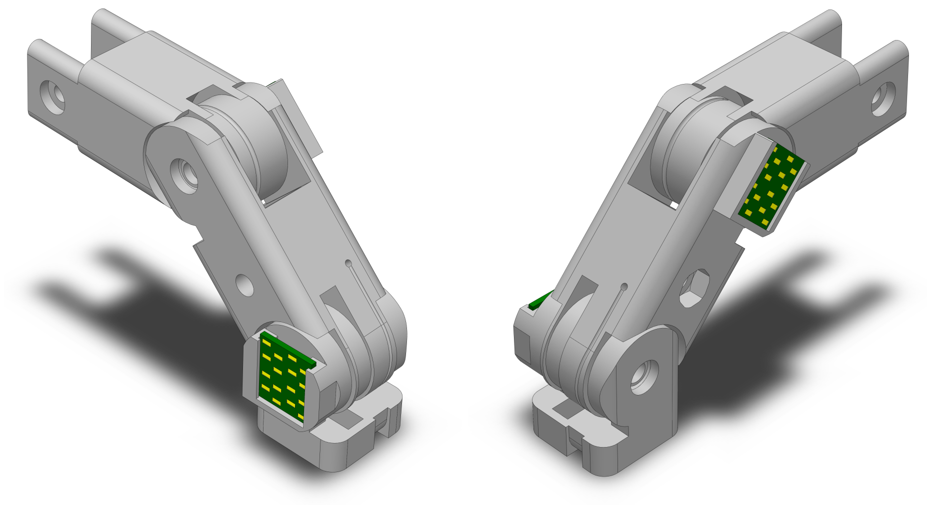


Figure 3.6: 3D render of dexterous finger revision 1.

The 3D render in figure 3.7 shows the second revision of the finger design. Compared to the previous revision this one features a more human-like design. Measurements of the authors' fingers were made to get similar proportions between joint sizes and link length. The outer diameter of the PIP joint was set to 3 cm. By comparison, the author's index finger PIP joint was measured to be roughly 2.8 cm across the PIP joint. The MCP joint diameter was based on the measured ratio between the MCP and PIP joint diameter of the authors hand, which showed that the MCP joint had roughly 2/3 the diameter of the PIP joint. Here a 20.4 mm in diameter joint was chosen. The proximal phalanx length was based on the measured ratio between the author's proximal phalanx and PIP joint diameter. The length of the middle phalanx on the other hand, was not based on the human hand. It was concluded that the lack of distal phalanx would have made it unnecessarily short.

The tip of the middle phalanx was designed to have the shape of a sphere. The reason for this is that a 2-degrees of freedom (DOF) finger can not control the orientation of the tip independently of position. With a spherical tip, the contact surface always has the same shape regardless of orientation. With this proposed design a pinching grip will have consistent quality, regardless of where the object is in relation to the fingers.

The final version had major upgrades in terms of tendon routing and tight-

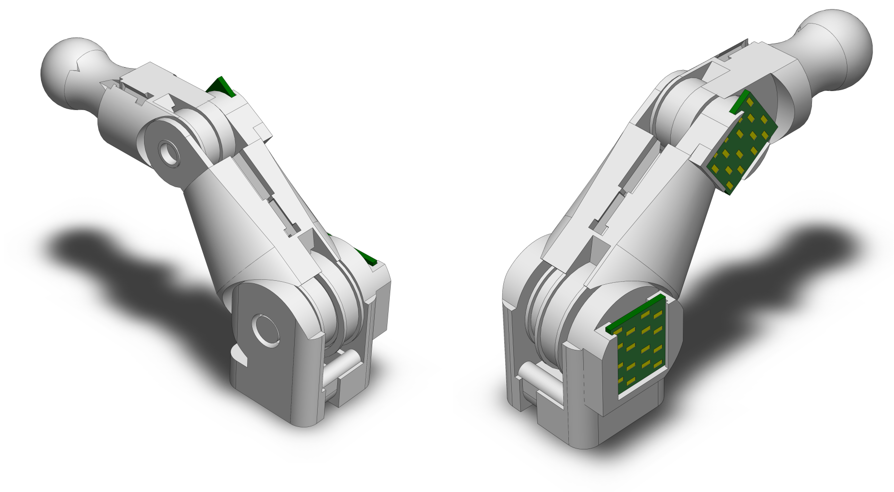


Figure 3.7: 3D render of dexterous finger revision 2.

ening mechanisms, with minor changes to appearance. The range of motion was upgraded from 110 to 135°, and moved to increase the overlapping workspace of a multi-finger configuration.

In the final version the proximal phalanx is 5.29 cm long from joint to joint. The middle phalanx is 6.75 cm from joint to tip, and 5.73 cm from joint to the center of the sphere. The sphere itself is 1.02 cm in diameter.

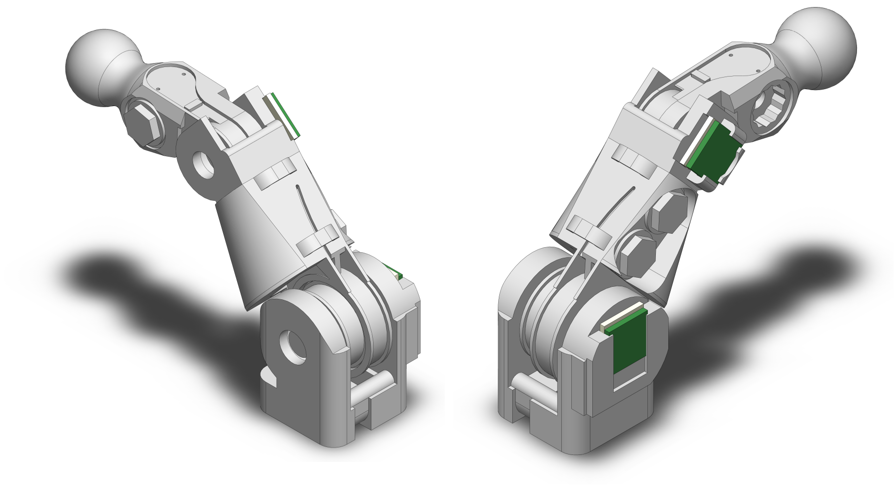


Figure 3.8: 3D render of dexterous finger revision 3.

3.4.3 Design solutions

The shape and appearance of the finger has mostly been a secondary concern. There were three major design problems that needed to be solved to realize the finger: How to attach joints and sensors, how to fasten and tighten the tendons, and how to route the tendons.

Joint attachment and angle sensors.

Figure 3.9 illustrates how a joint is attached. On each side of the joint, a plug with a square shaped tip is inserted into a fitted slot in the link, locking the plug and link together as the joint rotates. The outwards facing part of the plug is round and acts like hinges for the joint when the plug is set in place. These hinges rest on ball bearings, lowering friction between connected parts.

A magnet, illustrated in red in figure 3.9, is inserted into the plug, and is locked in rotation with the link. A sensor soldered to a small circuit board is then mounted over the joint. There is minimal amount of leeway between the circuit board and its mount, so sliding it into the mount will ensure that it is kept in place. The sensor measures the direction of the magnetic field created by the magnet to determine the angle of the link.

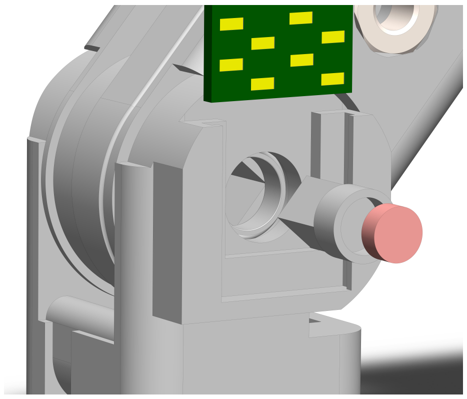


Figure 3.9: Plug for assembling finger.

Tendon tension

The first methods used to tighten the tendon was several variations of the concept seen in figure 3.10. The string would be tied around a screw with a nut attached at the end. The screw could then be tightened, which would move the nut inwards, and thereby tighten the string. This method had several problems. the string would sometimes be dragged into the nut and be subjected to sharp edges that would cut the string. However, the most important shortcoming was that the tendon could only be tightened to a certain extent before reaching the end of the bolt. This meant that the string would have to be sufficiently tight before being tied to the screw to create enough tension. Although the system worked, it made assembly unnecessarily difficult.

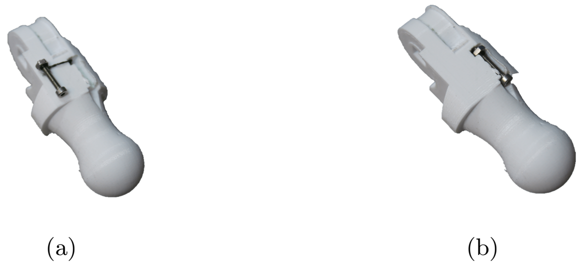

(a) (b)

Figure 3.10: First tendon tightening system.

The final concept is similar to a guitar tuning key. Figure 3.11 shows the tuning key analog used to tighten the tendon. The tendon is put through the hole in the axle, and tied in place. The tuning key is then rotated to tighten the tendon. Just below the hole in the axle, there is a star shaped pattern that locks into a slot in the finger. This lock prevents the tuning key from unwinding. The fastening method enables a high accuracy in tightening of the tendon as the fastener features 12 notches per rotations and a circumference of 15 mm, giving 1.25 mm of displacement of the tendon with each notch. Because of the hex shaped bottom, the tuning key can be tightened using a wrench. The tendon will also pull the tuning key towards the slot as it gets tighter, which prevents it from popping out on its own.

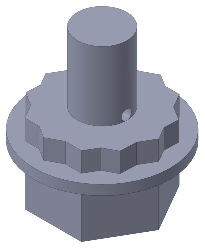
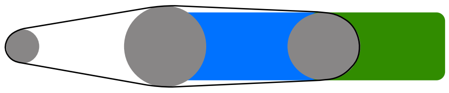


Figure 3.11: Tendon tuning key.

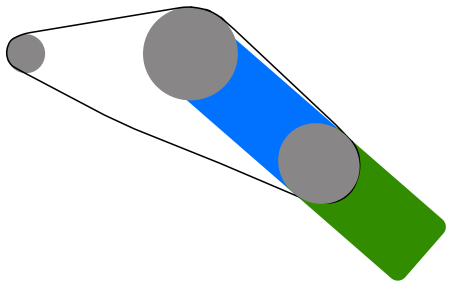
Tendon routing

The tendons are not elastic, and the set goal was to have a completely rigid link between motor and finger joints. This means that no matter what position the finger is in, the length of the tendon loop must stay the same. For the PIP joint, this is trivial, as its position is always the same relative to the gearbox.

The tendons controlling the MCP joint must pass through the PIP joint, without the movements of the PIP joint changing the total length of the tendon



(a)



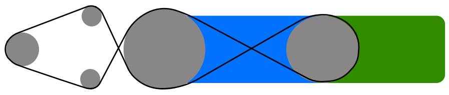
(b)

Figure 3.12: Tendon slack.

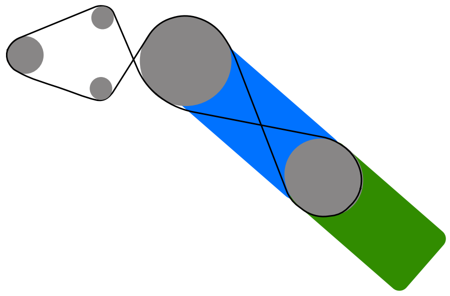
loop. This would results in tension loss or a full stop as the tendon would be unable to stretch further.

The tension slack problem is illustrated in figure 3.12. Here a pulley is used in the PIP joint to route the tendons to the MCP joint. The tendon wraps itself around the top and bottom of the pulley in the PIP joint. Small rotations of the PIP joint works perfectly well. Rotation of the PIP joint results in the tendon unwrapping on one side, and being further wrapped around the opposite side. For small movements of the PIP joint, the total loop length stays the same, and therefore the tension is constant. However, if the PIP joint is rotated to a certain extent, the tendon would unwrap itself entirely at the bottom side, to the point where it would lose contact with the pulley. The tendon loop length would effectively have been shortened, and tension would be lost.

For the endless tendon configuration, [Uyguroğlu and Demirel, 2006] uses two separate tendon loops that are both connected to the pulley in the PIP joint. However, this simple solution is made more complex when transferred to the limited loop configuration. Such a solution would require both loops to be fixed to the PIP pulley and also loop around it to ensure sufficient range.



(a)



(b)

Figure 3.13: Tendon with slack prevention measures.

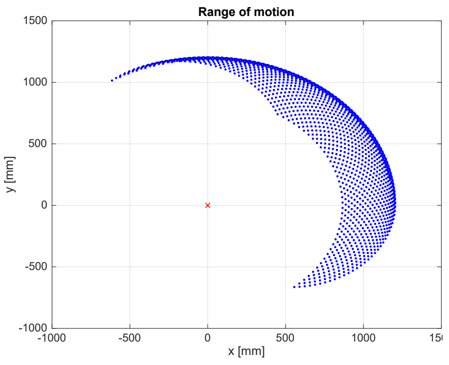
A solution more suitable to the limited loop configuration is illustrated in figure 3.13. Here the tendon is crossed between the pulley at the PIP joint, and the MCP joint. This increases the area where the tendon wraps itself around the PIP pulley, and would allow for greater range of movement (ROM) in the PIP joint without causing problems for the MCP joint. Additional pulleys are added before the PIP joint to wrap it even further around the pulley.

Range of motion

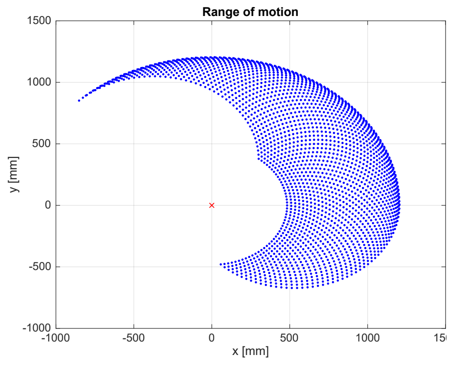
the range of movement (ROM) of a finger is defined by its link lengths and the allowed angle of the joints. Physical barriers stop the finger from bending too far backwards, or reach forward too far. To ensure that the ROM would be suitable for grasping tasks, some thought went into the mechanical design.

For dexterous manipulation, at least two fingers has to work together. To work together, it helps to have a lot of overlapping space where both fingers can operate. The fact that an object is placed between the manipulators, means that it is not an absolute requirement. Nevertheless, overlapping space offers greater flexibility.

In the early finger revisions it was noted that the 2D space that the finger could operate in was shaped as a thin crescent moon. This is illustrated in figure 3.14. Here, figure 3.14a shows both joints had a ROM of -20 to 90°, and the lengths of the links are 5.29 cm for the first and 6.75 cm for the second. 0° is defined as both fingers being extended. The resulting 2D space was poorly suited to overlap with an opposing finger. By increasing the ROM of each joint by only 15°, and moving the ROM a little, massive improvements were made. In the final version, the proximal phalanx has a -45 to 90° ROM. The middle phalanx has 0 to 135° ROM. The results are shown in figure 3.14b. Here the resulting area is much better suited to overlap with an opposing finger.



(a) ROM before adjustment.



(b) ROM after adjustment.

Figure 3.14: Plots of range of movement before and after joint angle adjustments.

3.5 Design of gear box

A general rule of thumb in using DC motors in practical applications is that they give a lot of speed, but little torque. For the control of the dexterous gripper what was wanted was a compromise between speed and torque, therefore it was found helpful to introduce a mechanical advantage to the transfer of torque from motor to the actuated joint. A gear box was created to house this mechanism, with focus on making it non-intrusive in the movement of the finger and while also compact and lightweight.

3.5.1 Evolution of design

The first gearbox in use was the prototype from the pre-project. Although it was originally made at the mechanical workshop at NTNU Gløshaugen, it was further built upon to test the strength of 3D printed parts, and to determine if the tendon needed grooves to stay in place. The gear box was large, heavy and could only actuate one link. Its main purpose was to test the principle.

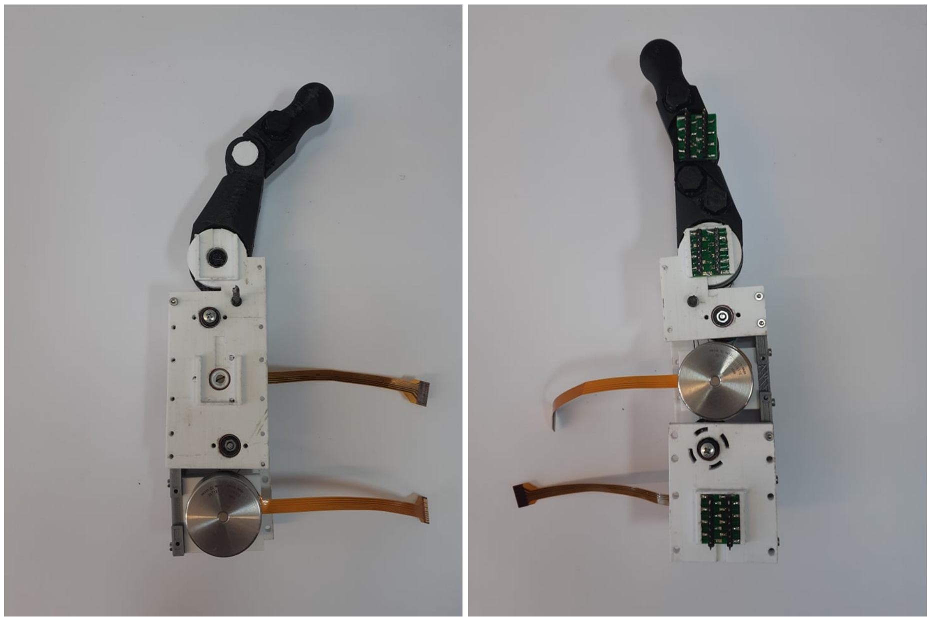


Figure 3.15: Photo of the second gearbox design

The second design was fully 3D printed, and had two separate tendon systems in order to actuate both links. An image of this gear box is shown in figure 3.15. It was small, long, slim and was used for the majority of this project. The different colours is a testament to how many parts where changed during the course of the project. Tightening mechanisms for the tendons was the main focus during this phase. From the picture the gearbox is visibly warped due to tension in the tendons, and the slim shape.

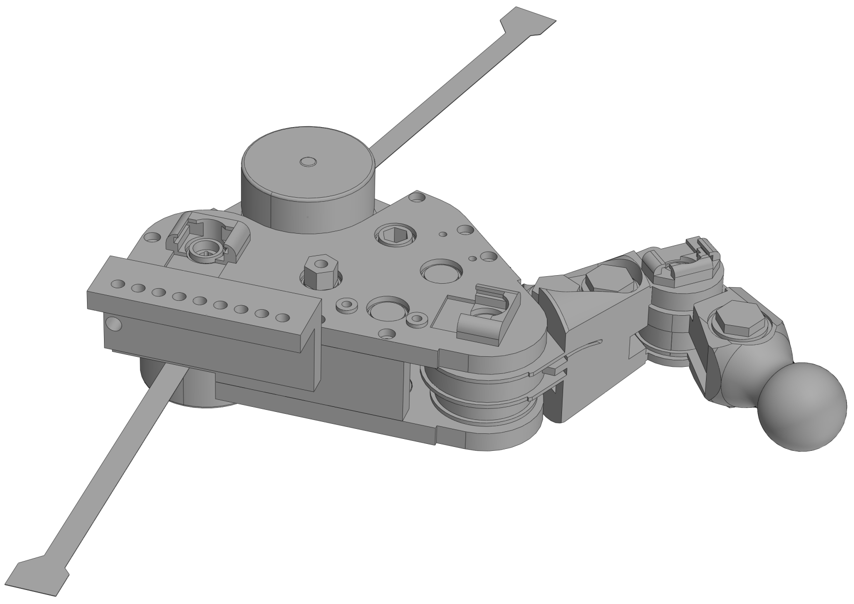


Figure 3.16: 3D render of final gearbox and finger design.

The third and final version is seen in figure 3.16. A shorter and wider design made it more robust to tension in the tendons. SINTEF requested the final version of the gearbox to be changed so that the “palm” was not sticking out. To accommodate, the gearbox was made wider on the opposite side. Sensor slots were changed as well because of new and smaller circuit boards. A mounting bracket was added for the gear box itself.

3.5.2 Design solutions

There were four major challenges to overcome when designing the gearbox: making the gearbox reasonably small while also having a reasonable amount of mechanical advantage, adding tension to the tendons, finding a good route for the tendons, and designing spiral grooves to keep the tendons in place.

Size and mechanical advantage

It was an aim to make the gearbox compact, as the modular design means that fingers can be placed in many different configurations. Therefore a large gearbox can be an obstacle for certain configurations.

Similarly to a spur gear, the ratio of the diameter of the two pulleys determines the mechanical advantage. How small the gear can be made is therefore limited by the mechanical advantage, and how small the smallest pulley can be made without breaking. This holds true for spur gears as well. However, the pulley system also needs to be tall enough to fit the string that is wound around it.

$$n_d = \frac{r_l}{r_d} \cdot n_l \tag{3.1}$$

Equation 3.1 gives an approximation of the number of turns necessary for the driving pulley to achieve a certain range of motion for the link. Here, n_l, is the range of motion wanted for the link expressed in turns. r_l is the radius of the link pulley, and r_d is the radius of the driving pulley, and n_d is the resulting turns necessary on the driving pulley. From the equation, it is seen that the number of turns needed on the driving pulley is related to the mechanical advantage. With the chosen 135° range of motion, and a mechanical advantage of 12, the result is 4.5 turns on the driving pulley. This holds true even if the gear is split up into several stages. Also, only half of the space of the driving pulley is used at any time. The minimum height of the first pulley is therefore given by equation 3.2 where h is the pulley height, d_t is the diameter of the string, and n_d is the number of turns on the driving pulley.

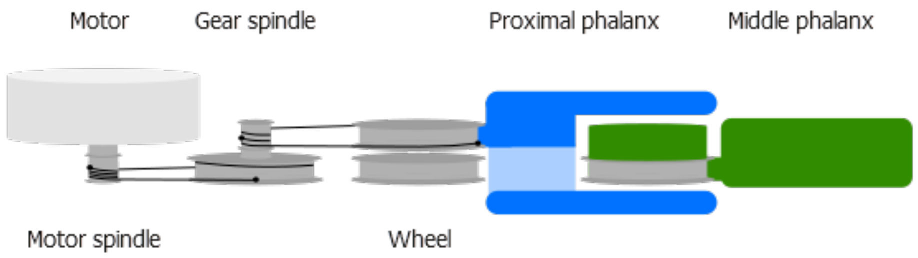


Figure 3.17: Actuation of the proximal phalanx through a two stage gear box.

$$h = 2 \cdot d_t \cdot n_d. \tag{3.2}$$

Through experimentation with 3D printing, it was concluded that a pulley diameter below 6 mm was problematic in terms of structural integrity, as tension could bend the pulley. With the outer diameter of the finger joints already established to be 30 mm and 20.4 mm, the maximum mechanical advantage for a single stage gear box is 5 and 3.4.

[Winsjansen, 2018] had calculated that, with the chosen motors, a mechanical advantage of 30 would give the system similar torque capabilities to that of a human finger. However, the resulting size of the gear box would be too large. It was instead decided to aim for a 2 stage gear box. Figure 3.17 and 3.18 are illustrations of how the gear box is used in to actuate the proximal and middle phalanx.

Table 3.1 lists the diameter of each pulley, and table 3.2 lists the resulting mechanical advantage.

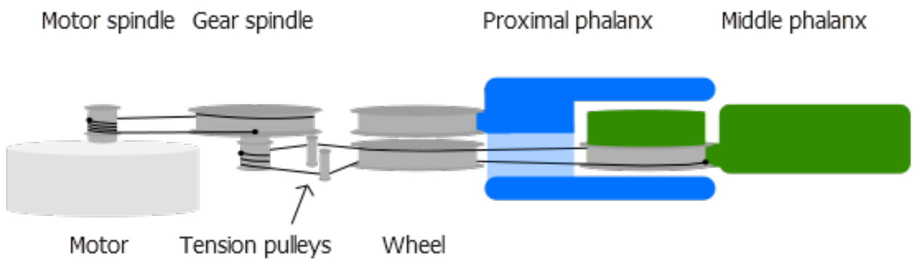


Figure 3.18: Actuation of the middle phalanx through a two stage gear box.

Pulley diameter	
Motor spindle	7.0 mm
Gear spindle large pulley	21.0 mm
Gear spindle small pulley	7.0 mm
Proximal phalanx pulley	28.0 mm
Middle phalanx pulley	18.4 mm
Wheel	23 mm

Table 3.1: Table of diameters.

Mechanical advantage			
	Stage 1	**Stage 2**	**Total**
Proximal phalanx	3	4	12
Middle phalanx	3	2.63	7.88

Table 3.2: Table of mechanical advantages.

Number of turns		
	Motor spindle	**Gear spindle (small pulley)**
Proximal phalanx	4.50	1.50
Middle phalanx	6.66	2.22

Table 3.3: Table of turn numbers.

The tendon length l_{pp} moving in and out of the PIP joint pulley is estimated in equations 3.3a-3.3b. n_{pp} is the number of turns required to achieve the wanted ROM in the joint, and r_{pp} is the radius of the PIP joint pulley. This length can be used to calculate the number of turns needed on each pulley, which can be used to extract the required height of the pulley.

$$l_{pp} = n_{pp} \cdot 2\pi r_{pp} \tag{3.3a}$$

$$l_{pp} = \frac{135^\circ}{360^\circ} \cdot 2\pi 14mm = 32,99mm \tag{3.3b}$$

Similarly, equations 3.4a-3.4b is used to estimate the length l_{mp} of a tendon moving back and forth in the last stage of the middle phalanx loop. Unlike the previous equation, the equation for this tendon has to account for movement of both joints. n_{mp} is the number of turns of the MCP link, r_{mp} is the radius of the MCP joint pulley, and r_w is the radius of the pulley referred to as the "wheel" in figure 3.17 and 3.18.

$$l_{mp} = n_{mp} \cdot 2\pi r_{mp} + n_{pp} \cdot 2\pi r_w \tag{3.4a}$$

$$l_{mp} = \frac{135^\circ}{360^\circ} \cdot 2\pi \cdot 9.2mm + \frac{135^\circ}{360^\circ} \cdot 2\pi \cdot 11.5mm = 48.77mm \tag{3.4b}$$

Table 3.3 lists the number of turns on the smallest pulley of each stage. The smallest pulleys demand the most turns, and are therefore the crucial design element in deciding the pulley height.

A 0.13mm thick string with a weight capacity of 8 kg was acquired for the gearbox. To have range to spare, 8 turns was used on the motor spindle. The formula in equation 3.2, reveals that the pulley height of the first loop must be at least 2.08 mm. To get some extra room, the pulleys where chosen to be 6mm tall. Instead of using smaller pulleys for the second loop, a thicker 0.8mm string string was used instead. The smallest pulley on the gear spindle could fit 3 turns of this string, which provided more range than specified in table 3.3.

Tendon tension and fastening

The problem of how to fasten the string to the pulleys, as well as how to add tension, was a much bigger challenge for the first stage of the gear box than the second. The second stage could be tightened in the finger, which had more available space for tightening mechanisms. This mechanism is covered in section 3.4.3.

Each stage consists of a large pulley and a smaller pulley. The larger pulley is the natural place for a tightening mechanism because of the extra space. The string still needs to be fixed in the middle of the small pulley. To achieve this, the string was routed through a hole in the middle of the small pulley. The hole was then filled with plastic glue. In the assembly manual, this is referred to as the locking hole.

The first tightening mechanism used for the finger showed in figure 3.10 was also tried in the gear box, with the same results. The tightening range was simply too small.

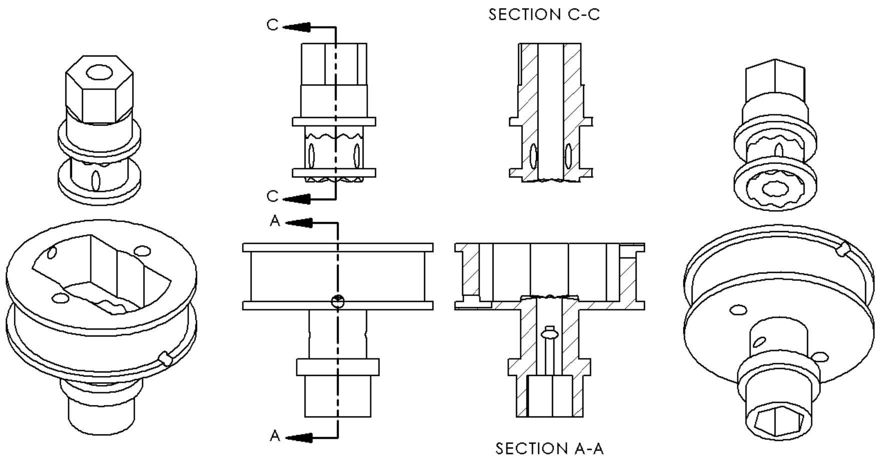


Figure 3.19: Drawing of the gear spindle.

The final mechanism is shown in figure 3.19. The top of the gear spindle acts as a tightening mechanism for the string in the first gear stage, while the bottom serves as the next stage. Both parts are joined together using a bolt through their center. Then a string is tied to the upper part, which is rotated using a wrench or similar. This tightens the string. The string is then tied to the top of the gear spindle. A bolt runs through the whole gear spindle, and can be tightened with a nut on the bottom side. The surface between the gear top, and the rest of the gear spindle has a ridged pattern, which stops it from moving as long as the bolt is sufficiently tightened.

Tendon routing

The general layout of the gearbox, and a simplified version of how the tendons are routed, is illustrated in figure 3.20. Colours indicate which plane each loop reside in. Blue tendons are in the upper plane, and red tendons are in the lower plane. In the illustration, the number of turns is left out to avoid clutter in the figure. Thus, it illustrates how it would look in an endless tendon configuration. In reality, they are wrapped around the pulleys in accordance with table 3.3.

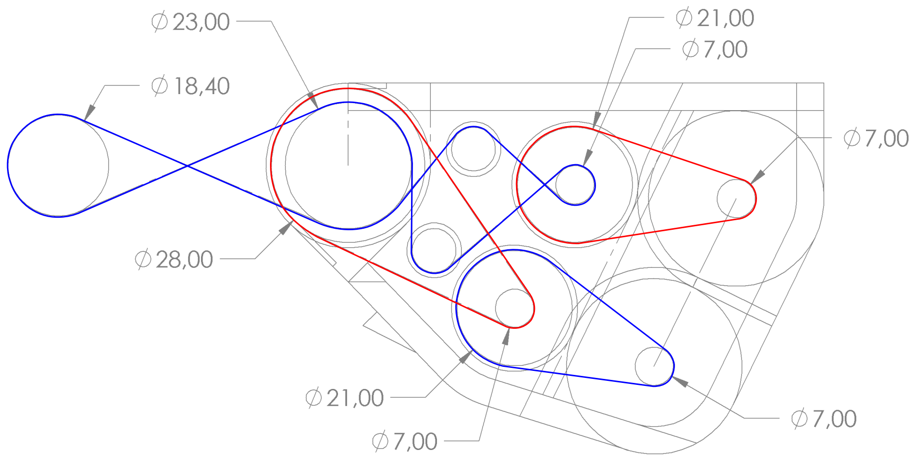


Figure 3.20: Gearbox layout and tendon routing

Spiral grooves

The first prototype design of the gear box, as seen in figure 3.21, had spiral grooves to keep the wire in place. If the wire were to cross itself, it would cause increased tension since the radius at the crossing point would increase. If the string is not configured in a perfect spiral, the exit and entry points of the pulleys might traverse the height of the pulley at different speeds, which would also cause changes in tension.

The idea behind the grooves was creating a more reliable system where the string positions were stable. With an increase in tension, the string would be pulled towards the shortest path, which would would be inside the grooves.

Vertical traversal of the pulley

Since one pulley "feeds" the other pulley with string, it is important that the exit/entry sites on each pulley move at the same speed vertically. This ensures that the string exiting one pulley always falls directly into the grooves of the other pulley. In equations 3.5a-3.5f, the conditions necessary for both spirals to traverse vertically at the same speed is calculated.

Equation 3.5a shows height gain in spiral 1 h_1, as the first spiral is rotated. Here p_1 is the "pitch", which is the height gain for one turn. $+theta_1$ is rotations in radians. Equation 3.5b is identical, but for pulley 2.

The relationship between rotation in pulley 1 and 2 is described in equation 3.5c, and is given by the mechanical advantage. Here r_1 and r_2 is the radius of pulley 1 and 2 respectively.

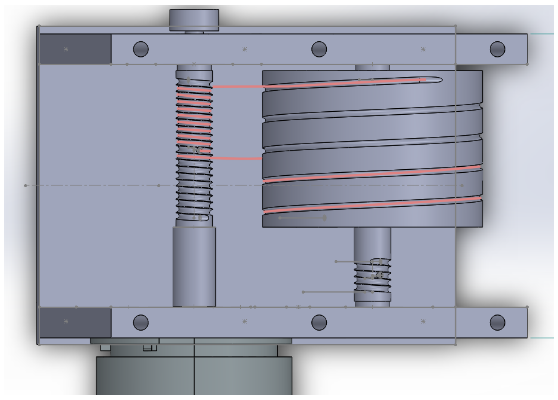


Figure 3.21: Prototype of pulley based gearbox. [Winsjansen, 2018]
.

Inserting equation 3.5c into 3.5b, yields equation 3.5d which shows the traversed height of pulley 2 in relation to rotation of pulley 1.

Equation 3.5e holds true if the pulleys traverse vertically at the same rate. The result is shown in equation 3.5f, and shows that the pitch of pulley 1 and 2 must be related through the mechanical advantage.

$$h_1(\theta_1) = \frac{p_1}{2\pi} \cdot \theta_1 \tag{3.5a}$$
$$h_2(\theta_2) = \frac{p_2}{2\pi} \cdot \theta_2 \tag{3.5b}$$
$$\theta_2 = \frac{r_1}{r_2}\theta_1 \tag{3.5c}$$
$$h_2(\theta_1) = \frac{p_2 r_1}{2\pi r_2}\theta_1 \tag{3.5d}$$
$$\frac{dh_1}{d\theta_1} = \frac{dh_2}{d\theta_1} \tag{3.5e}$$
$$p_1 = p_2\frac{r_1}{r_2} \tag{3.5f}$$

Rate of exchange

Equations 3.6a-3.6i investigates if one pulley is able to receive the same length of string that the other pulley feeds as they rotate. Equation 3.6a is used to calculate the length l of the string that fits inside one turn of the spiral.

The length of the string for an arbitrary angle of rotation is found by dividing by 2π and multiplying by the angle. Thus, equation 3.6b gives the string length of pulley 1 with respect to the angle of pulley 2.

Using the relationship between θ_1 and θ_2 from equation 3.5c, the relationship between the string length of pulley 2 and the angle of pulley 1 is found in equation 3.6c.

In equation 3.6d the string lengths are set equal. This must be true for the pulleys to be able to exchange the string without problems. Simplification of this equality through equation 3.6e-3.6i reveals the same condition as in equation 3.5f.

$$l = \cdot\sqrt{(2\pi r)^2 + p} \tag{3.6a}$$

$$l_1(\theta_1) = \theta_1 \frac{\sqrt{(2\pi r_1)^2 + p_1^2}}{2\pi} \tag{3.6b}$$

$$l_2(\theta_1) = \theta_1 \frac{r_1}{r_2} \frac{\sqrt{(2\pi r_2)^2 + p_2^2}}{2\pi} \tag{3.6c}$$

$$l_1(\theta_1) = l_2(theta_1) \tag{3.6d}$$

$$\sqrt{(2\pi r_1)^2 + p_1^2} = \frac{r_1}{r_2}\sqrt{(2\pi r_2)^2 + p_2^2} \tag{3.6e}$$

$$(2\pi r_1)^2 + p_1^2 = \frac{r_1^2}{r_2^2}[(2\pi r_2)^2 + p_2^2] \tag{3.6f}$$

$$(2\pi)^2 + \frac{p_1^2}{r_1^2} = (2\pi)^2 + \frac{p_2^2}{r_2^2} \tag{3.6g}$$

$$p_1^2 = \frac{r_1^2}{r_2^2} p_2^2 \tag{3.6h}$$

$$p_1 = \frac{r_1}{r_2} p_2 \tag{3.6i}$$

Spiral phase

Although the spirals exchange the cord at a synchronized rate, the issue of "phase" still remains. Assuming the grooves at the top of both spirals have been manually aligned, the connection at the bottom of the spiral may be out of phase. Figure 3.22 illustrates the phase shift. Here there are zero whole turns.

Suppose the horizontal line at the top is the string connecting the two helices at the top of the two pulleys. Because of the difference in radius, the other line representing the bottom connection will be phase shifted by 19.19 °. This phase shift is an additional amount of rotation that was not accounted for in equation 3.6a-3.6i. In equation 3.7a-3.7f this phase shift is added.

Equation 3.7a is the expression for the string length for pulley 1. Here, angle is replaced with turns, and whole turns are denoted as n. From figure 3.22 it is seen that both pulleys has at least half a turn $\pm$ the phase shift, and the number of whole turns. Equation 3.7b is the string length of pulley 2. Here the condition for pitch found in equation 3.5f is inserted into p_2.

Setting the string lengths equal to each other (equation 3.7d-3.7e) leaves a messy, and less useful expression. As an example, equation 3.7f gives the solution to p_1. However, when inserting the desired radius, turns, and p_2, the resulting pitch for pulley one may not be practical. Tweaking the variables to something sufficiently close to the desired values will often produce practical results. consequently, using this formula becomes an optimization problem.

A simpler solution to the phase shift problem is introducing a separate phase shifted spiral on each pulley. This configuration is displayed in figure 3.23.

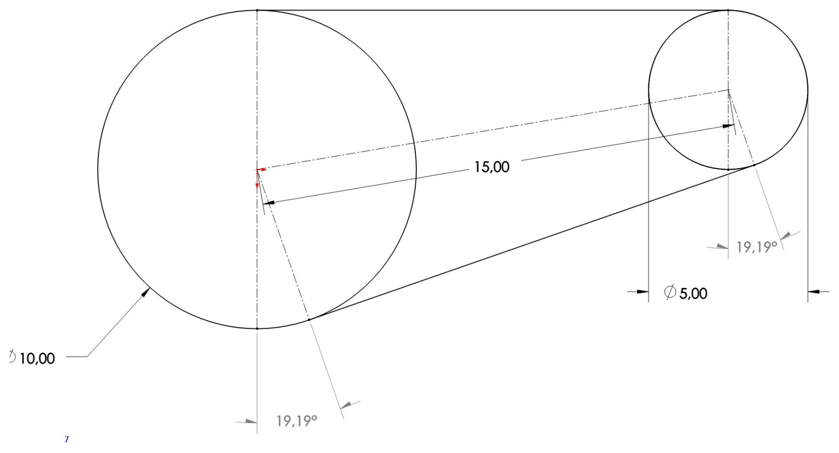


Figure 3.22: Helix phase shift diagram.

$$l_1 = (n_1 + 0.5 + \eta) \cdot \sqrt{(2\pi r_1)^2 + p_1^2} \quad (3.7a)$$

$$l_2 = (n_2 + 0.5 - \eta) \cdot \sqrt{(2\pi r_2)^2 + (p_1 \frac{r_2}{r_1})^2} \quad (3.7b)$$

$$l_1 = l_2 \quad (3.7c)$$

$$(n_1 + 0.5 + \eta) \cdot \sqrt{(2\pi r_1)^2 + p_1^2} = (n_2 + 0.5 - \eta) \cdot \sqrt{(2\pi r_2)^2 + (p_1 \frac{r_2}{r_1})^2} \quad (3.7d)$$

$$(n_1 + 0.5 + \eta)^2 \cdot ((2\pi r_1)^2 + p_1^2) = (n_2 + 0.5 - \eta)^2 \cdot ((2\pi r_2)^2 + (p_1 \frac{r_2}{r_1})^2) \quad (3.7e)$$

$$p_1 = \sqrt{\frac{(2\pi r_2)^2 (n_2 + 0.5 - \eta)^2 - (2\pi r_1)^2 (n_1 + 0.5 - \eta)^2}{(n_1 + 0.5 + \eta)^2 - \frac{r_2^2}{r_1^2}(n_2 + 0.5 - \eta)^2}} \quad (3.7f)$$

Grooves or not

Figure 3.24 illustrates the existence of a shortest path between the fixed points used in the gear box. Adding tension should therefore force the strings into a stable configuration without grooves. Whether it is “stable enough” is a different question, and will be further discussed in section 7.1.2.

The complexity added by the inclusion of grooves proved to be very time consuming, and the concept was eventually abandoned. Small changes could

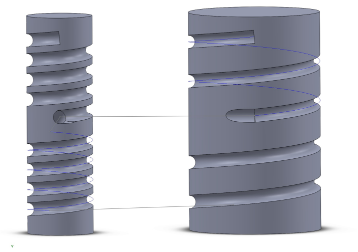


Figure 3.23: 3D render of helix phase shift solution.

mean that the grooves would need to be redesigned. For example, the thickness of the string would effectively change the radius of both pulleys. However, in terms of percentage, the change would be small for the largest pulley, but large for the smallest pulley. For example, a pulley with 10 mm diameter, with a 2 mm diameter string wrapped around, yields an effective radius of 11 mm, which is a 10% increase. If the other pulley is 20mm, the radius changes by only by 5%.The mechanical advantage would therefore be changed, and everything would need to be redesigned. Another issue that deserves mentioning is that the string takes a sloped path through the air gap between pulleys. This is apparent from figure 3.24, where the squares on the right show the cylinders rolled out on a plane defined by the trajectory of the tendon. Thus, The spiral grooves on each pulley needs a vertical offset, as well as all the previously discussed considerations, for it to work.

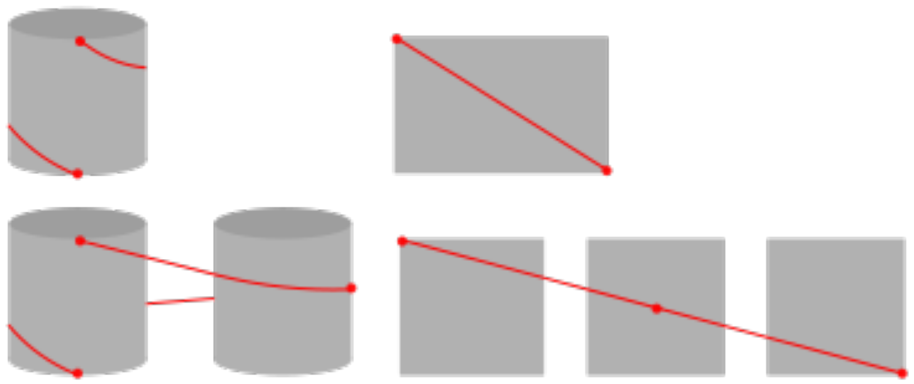


Figure 3.24: The shortest path for the string in the gear system

3.5.3 Hybrid gearbox

A hybrid solution was proposed, using a planetary gear as the first gear stage, while keeping the second intact. The principle is illustrated in figure 3.25. This would have the benefit of reducing complexity and Assembly time. Backlash

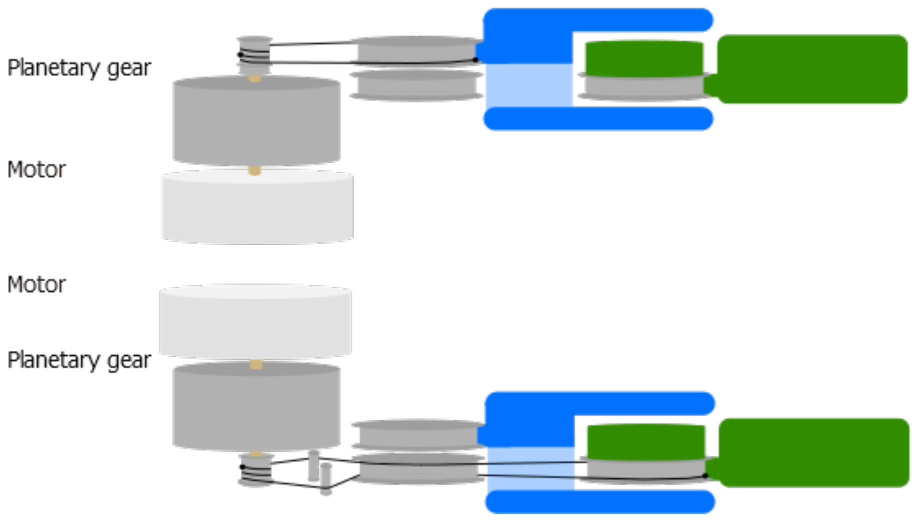


Figure 3.25: Proposed solution using planetary gear as first stage

introduced by the planetary gear would be reduced through the second gear stage. However, the project ended before such a system could be explored.

3.6 Design of rack

A rack was made to simplify the testing environment of a two finger configuration. It can house one Raspberry Pi model 3 B with a connected SPI HAT and two Kiyona control boards, meaning it can control a two finger configuration of the dexterous gripper. The Raspberry Pi and the two ESP32s are interconnected through holes in the rack that make the connection ports on all cards accessible, there needs to be room for the input of four SPI cable, two coming from each of the two fingers, to the Raspberry Pi HAT. The ESP32s each need to output six wires with the PWM signal for the motors, accept the two wire I^2C interface as well as connect to external power supply and ground. The rack also features slots for all three boards to keep them secured, and mounts for two fingers.

To mount the fingers in a configuration with the ability to perform dexterous manipulations side mounts were also designed. They each feature seven cutouts that meet the dimensions of the mounting plate of the gearbox presented in section 3.5. Thus, the operator is able to assemble the fingers in a variety of different configurations, allowing for experiments with differing overlaps of the ranges of motion of two fingers.

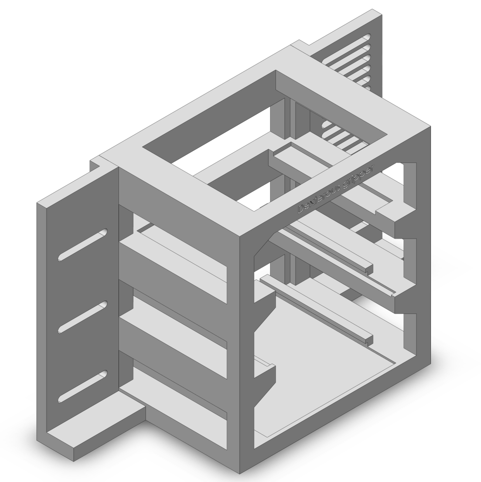


Figure 3.26: Dexterous gripper rack.

3.7 Construction

An assembly guide has been made to streamline the process of putting the finger and gearbox together, it is available on GitHub [1], together with original Solidworks files, as well as print-ready files.

3.7.1 3D-printing

All the plastic parts for a complete finger with gearbox can be printed in one run using a FlashForge Finder 2.0 printer. Figure 3.27 shows the layout of the print, which was set up in the FlashPrint software. The estimated print time is 39.5 hours with the highest resolution, 35mm/s print speed, and 15% infill. All parts where made using these settings. Using 100% infill on high stress parts like the tuning pegs and gear spindle top is recommended, but not strictly necessary.

Experimentation with the 3D-printer showed that printed parts tended to shrink a little. For the bearings which had an outer diameter of 13 mm and bore of 8 mm, the axle could be designed with 8 mm diameter and make a tight fit. The bearing slot would need to be 13.2 5mm in diameter because of shrinkage. In general, 0.25 mm was the go-to clearing when something needed to fit tightly together. It should be noted that other 3D-printers may behave differently, and that sanding the printed parts is always necessary. Orientation also plays a large part on the result. For example, holes that perforates an object horizontally will not be perfectly round. The side facing down will also

[1]The project's repository is available at: `https://github.com/Bardie4/Dexterous`

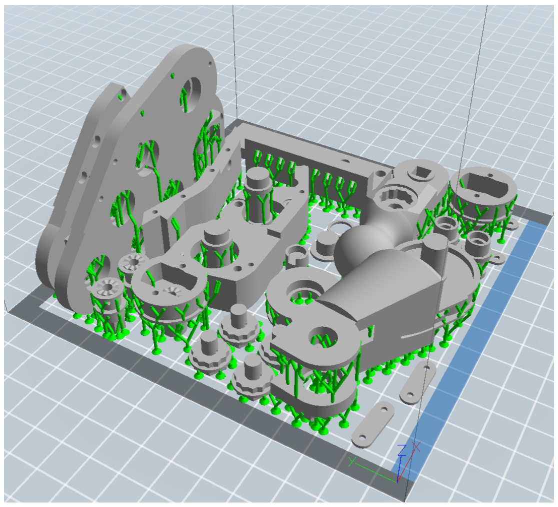


Figure 3.27: 3D print layout for finger and gearbox parts.

have inferior quality compared to the side facing up.

3.7.2 Parts

In addition to the 3D printed parts, the following items needs to be acquired:

- 2 pcs. 24V 15W EC flat motors
- 6 pcs. M3 x 6mm bolt
- 20 pcs. M2 x 10mm bolt
- 2 pcs. M3 x 30mm bolt
- 2 pcs. M3 hex nut
- 16x 8x12x3.5 bearings
- POWER PRO Microfilament braided line (diameter: 0.13 mm)
- FSE ROBLINE (diameter: 0.8mm)

Chapter 4

Controller design

The choice of control strategy is crucial in controlling the dexterous gripper and ensuring stability of the system. Therefore considerable effort has been put into designing a suitable controller.

4.1 Overview

SINTEF's vision for the gripping system was for it to be a somewhat self contained control system. This meant that control loops should run on hardware that is a part of the gripper, while an outside system sends commands for the gripper to perform.

Outside the gripper itself, cameras and other sensors, like the GelSight, would be used to locate objects and figure out appropriate control objectives for the gripper in order to interact with its environment. This system is outside the scope of this thesis, and only exists in fragments. It will be referred to as task control. The GelSight sensor would enable the task controller to verify the quality of grasps performed by the gripper. New control objectives could then be sent to the gripper in case of detected inadequate grasp quality.

[Winsjansen, 2018] had decided on BLDC motors as the actuator of choice for the gripper, and done research into control methods for the motor. For BLDC motors, as opposed to DC motors, the magnetic field has to be controlled manually. This system will be referred to as "low level control". The low level control system acts like an interface that simplifies interactions with the motor.

Between the task loop and low level control is "high level control". High level control can be many different things depending on the task at hand. Position control, trajectory control, and so on. The total structure of the control system is illustrated in figure 4.1. Where high and low level control are the main areas covered by this thesis.

From a kinematic perspective the finger of the dexterous gripper is rather simple, as illustrated in figure 4.2. It consists of two links, l_1 and l_2, with two actuated joints θ_1 and θ_2. Meaning, its workspace is only in two dimensions. As

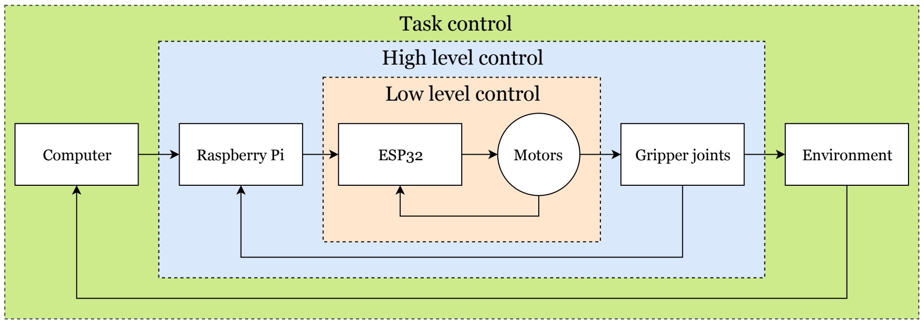


Figure 4.1: Overview of total system structure.

the GelSight sensor housing is further built upon and miniaturized by SINTEF, it will potentially be mounted on the finger in the form of a passive third link l_3.

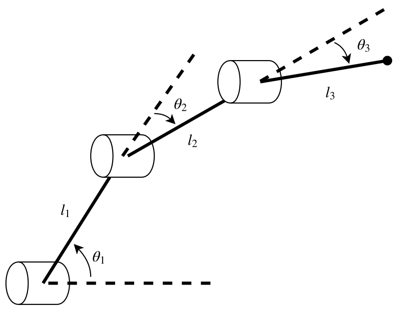


Figure 4.2: Kinematic illustration of the finger.

4.2 Low level motor control

The low level control consists, in short, of motor control using the angle sensor readings. Figure 4.3 shows the feedback control system for the control of one motor.

Robust and accurate control of each joint is needed to control the finger. To achieve this it was chosen to utilize a BLDC motor, as presented in section 5.1.2. The chosen motor, Maxon EC 32 flat, is able to operate at a wide range of speeds given an able controller. For this project it was aimed to make a controller using an external angle sensor for commutation.

The success in control of a BLDC motor wholly depends on the implementation of a commutation principle. [Winsjansen, 2018] presents several such principles, where the conclusion is a recommendation of using trapezoidal commutation in a commercial motor driver sold with the motors, an Escon module 24/2 servo controller. This proved unfeasible as it was simply too slow, which is

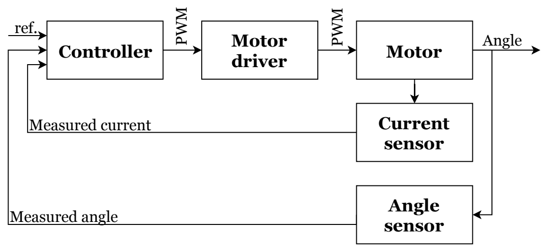


Figure 4.3: Joint control loop.

discussed further in section 5.1.1. Another control strategy was therefore chosen instead: sinusoidal commutation.

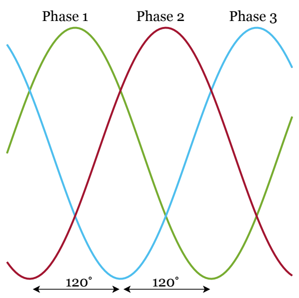
What is common for all DC motors is that they are actuated through applying current to their windings, causing the induction of a magnetic field. This magnetic field will, when overlapping with another inversely charged magnetic field cause an attracting force. Generally these fields are induced by permanent magnets in the DC motor. For a single pole pair motor the maximum force is achieved when the permanent magnets and the magnetic field are at a 90° angle of one another.

Put shortly, the control of a brushless direct current motor differs from that of a brushed DC motor by its stator and rotor's function being swapped. In a BLDC motor the rotor is a permanent magnet, while the stator holds the coils which induce the magnetic fields needed to drive the motor. The motor used in this project is a three phase motor, meaning three pairs of coils need to be magnetized to drive the rotor.

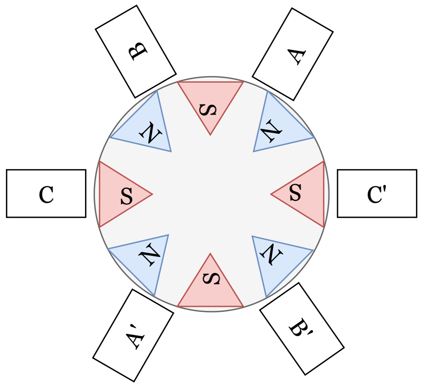
The three most widely used ways to control a BLDC motor are: trapezoidal, sinusoidal and field oriented control (FOC) commutation [Lee et al., 2009]. For this project sinusoidal commutation was chosen, as it provides better performance than trapezoidal, while not being as complex and expensive as FOC. It would provide the overall best control, but most of the performance gain is in high speed control, which is not as important in this project as controlling torque accurately.

If a pure sine wave current is delivered to the motor-windings, the magnetic field strength will also be sinusoidal. In a three phase motor the sum of the sinusoids will be a constant value as the motor turns. The optimal combination of voltages applied to the windings resemble that of three sine waves with 120° phase shift, as in figure 4.4a. This commutation principle can be implemented with widely available microcontrollers with the ability to drive a PWM-signal at adequate frequencies. As a bonus the motor will run silently given a high enough frequency.

To achieve optimal torque control the angle of the rotor is measured, as discussed in section 5.1.4. To give the angle sensor a reference value the motors are, on startup, driven to their endpoint in a calibration routine. The endpoint is then set as a zero point in software. Using this as a reference; a combination of PWM signals can be calculated that will give optimal torque. Optimal torque is achieved when the magnetic field of the stator is set 22.5°clockwise (or counter-clockwise) from the current angle. This way, the torque vector is pointing tangentially along the periphery of the motor giving the maximal torque possible.



(a) Three phase sine wave.



(b) Four pole-pair, three-phase BLDC configuration.

Figure 4.4: BLDC principles.

4.3 High level control/Outer control loop

There are several useful objectives to control in the outer loop, and it depends on the task the robot is trying to achieve. Some tasks require the robot to follow a trajectory, other tasks could require it to go to a certain position, or reach a certain speed. There can be an optimization problem to be solved, and certain conditions to avoid. For grasping tasks, or dexterous manipulation in general, the force applied to objects is important because it is through these forces the object is manipulated.

Principally, there is no controller that does everything. Because of this, considerable efforts were put into making a controller manager system. The system in question is further detailed in section 6.3. It enables changing controllers, tuning controllers, calibrating sensors, broadcasting sensor data on the local network, and running different controllers for different fingers at the same time.

This left little time for implementing the actual controllers themselves. However, two controllers were made to control the joint angle; one with joint angle set point as input, and the other with Cartesian coordinates as input.

4.3.1 Independent joint control

Independent joint control is the simplest form of control in robotics. As the name suggests, each joint is controlled independently, and any interaction between links is treated as a disturbance.

Joint space input

A controller with joint space inputs is used when inverse kinematics is calculated outside the gripper. If the robot, that the gripper is theoretically mounted on, also has its inverse kinematics calculated on a separate computer, it might make sense to do the same with the gripper. Thus, gathering everything in the same place. In this configuration, robot and gripper movements can be planned together, which opens up the opportunity to use advanced features like collision avoidance.

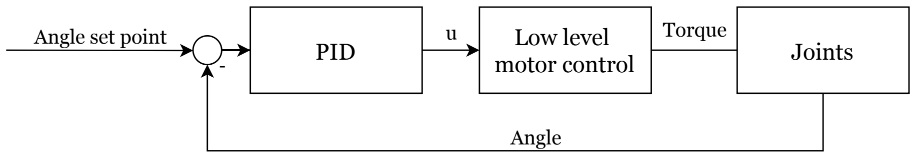


Figure 4.5: Independent joint controller with joint space inputs

Cartesian input

Instead of planning the movement of the robot and gripper together, the robot can simply move the gripper within reach of an object. Then, fingers can then be moved using Cartesian coordinates. From figure 4.6 it is seen that the only difference to the joint space controller is the inverse kinematics calculation performed on the input.

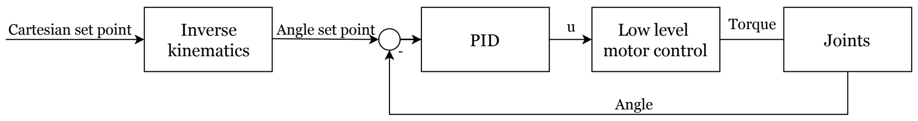


Figure 4.6: Independent joint controller with Cartesian inputs

The inverse kinematics is calculated using equation 4.1a-4.1e, where the $\pm$

sign chosen in equation 4.1a determines the elbow direction for the solution.

$$\theta_2 = atan2(\pm\sqrt{1-(\frac{x^2+y^2-l_1^2-l_2^2}{2l_1l_2})^2}, \frac{x^2+y^2-l_1^2-l_2^2}{2l_1l_2}) \tag{4.1a}$$
$$k_1 = l_1 + l_2cos(\theta_2) \tag{4.1b}$$
$$k_2 = l_2sin(\theta_2) \tag{4.1c}$$
$$\gamma = atan2(k_2, k_1) \tag{4.1d}$$
$$\theta_1 = atan2(y, x) - \gamma \tag{4.1e}$$

4.3.2 MCP joint and motor coordinates

Figure 4.7 illustrates how the MCP joint angle is a function of both the motor angle and the angle of the PIP joint. In this figure, the concept is simplified by not including any gear stage and pretending that the motor pulley has equal radius as the MCP joint pulley. Thus, the angle of the motor is not a real angle, but an illustrative coordinate.

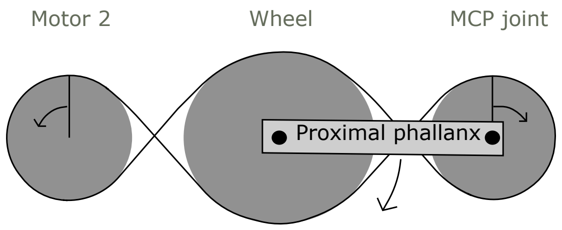


Figure 4.7: Connection between motor angle, PIP angle and MCP angle

When the proximal phalanx rotates in the direction of the arrow, the tendon of the MCP joint wraps it self around the upper part of the "wheel" at the PIP joint. Note that the wheel rotates freely, and is not fixed to the rotation of the PIP joint. The length of tendon wrapped around the upper part l_w is given by equation 4.2a, where θ_{pip} is the angle of the PIP joint and r_w is the radius of the wheel. This length must be taken from either the motor pulley or MCP joint pulley, causing a them to rotate as illustrated by the arrows in figure 4.7. This can be represented by the equality in equation 4.2b . Here l_{mcp} is the length of tendon coming from the bottom side of the MCP joint, and l_{m2} is the length of tendon coming from the bottom side of the imaginary motor pulley.

Inserting the same formula used in equation 4.2a, into equation 4.2b, yields equation .

$$l_w = \frac{\theta_{pip}}{2\pi} 2\pi r_w = \theta_{pip} r_w \tag{4.2a}$$

$$l_{pip} = l_{mcp} + l_{m2} \tag{4.2b}$$

$$\theta_{pip} r_w = \theta_{mcp} r_{mcp} - \theta_{m2} r_{mcp} \tag{4.2c}$$

$$\theta_{m2} = \frac{\theta_{mcp} r_{mcp} - \theta_{pip} r_w}{r_{mcp}} \tag{4.2d}$$

The motor coordinate can be a useful coordinate for controlling the MCP joint. As an example, lets say that the set point of the PIP joint changes, while the MCP joint should stay where it is. Then, according to equation 4.2c, the motor of the MCP joint still has to move in order for the MCP joint to stand still.

When θ_{mcp} is controlled directly, the movements in the PIP joint will be experienced as a disturbance on the MCP joint. However, when controlling θ_{mcp} through the motor coordinate θ_{m2}, it is apparent that the motor was already in the wrong position from the start.

Chapter 5

Electronic design

To control a gripper driven by electrically actuated motors there is a need for an electronic sensor and actuation system. The gripper consists of motors and sensors which need to cooperate for the finger to follow any desired trajectory. To create such a system an interconnection of hardware is needed. As a requirement for making the system usable by a person without deep knowledge of how it is programmed, or how it is put together, user friendly electronic design should be made.

5.1 Hardware

When choosing hardware for the dexterous gripper the emphasis has been put on choosing integrated circuits that are easy to work with and which provide the necessary processing power and speed to be able to perform dexterous movements. This project's specifications put high demands in reaction time for the gripper to be able to sense and automatically correct its movements.

5.1.1 Escon module 24/2 servo controller

The Escon module 24/2 servo controller was acquired during the project leading up to this thesis. It was marketed as a velocity controller, but could also be used for torque control, and was relatively cheap. The module was included in a cascade control scheme, to control the angular position of the joint of a finger. Here, it acted as the inner loop, controlling torque. The outer loop was a PID angle controller running on an Arduino Uno Rev3, which fed the Escon module inputs based on angle error.

The setup proved to have stability problems, and was simplified in order to investigate the source of the problem. Eventually, the setup consisted of only a motor with nothing attached other than the angle sensor. The outer loop was simplified to a proportional controller. However, the problem persisted. The motor would either be to weak to move when the proportional gain was low, or oscillate when the gain was increased. It was concluded that the controller was not suited for our application. Most likely, it was not suited to control torque in while rapidly changing directions, or the commutation method (trapezoidal commutation) was too inaccurate for torque control at low speeds. The Escon module was therefore rejected.

5.1.2 Motor

A Maxon EC 32 flat BLDC motor was chosen to drive the joints of the system. It is a fast and responsiv three phase, four pole pair, brushless direct current (BLDC) motor, in a small package. Four pole means there are eight permanent magnets with alternating polarity arranged along the stator of the motor. The speed and real-time concerns of this project means that a powerful and fast motor was needed. The fact that the motor comes in a very compact package (32 mm in diameter by 16 mm in height), coupled with its adequately low power usage at 24V 15W makes it suitable for integration into a robotic finger. Furthermore utilizing a BLDC motor enables the implementation of sinusoidal commutation, a high performance control strategy. The choice of motor was done in the pre-project stage and is further described by [Winsjansen, 2018].

5.1.3 Motor driver

A Texas Instruments DRV8313 2.5-A triple 1/2-H bridge driver was chosen to drive the motor. When controlling a 15W BLDC motor using a 3.3V logic microcontroller there is a need to step up the PWM signal. A voltage of 3.3V and current in the milliampere range does not induce a large enough magnetic field to rotate the rotor. Instead, the PWM signal voltage controls a 24V PWM signal that the motor is rated for through each field-effect transistor in the motor driver.

The selection of motor was based on the fact that it provides the necessary speed, accepts PWM input signals and supports the power level of 24 V 0.5 A which matches the power ratings of the chosen motor. The driver comes in a HTSSOP-28 package with dimensions of 9.70 mm x 4.40 mm, meaning it is small, and therefore suitable for placement onto a printed circuit board which can be kept close to, or even integrated ted into, the gripper. Furthermore, this driver enables motor control with sinusoidal commutation, the chosen motor control method in this project, further discussed in section 4.2.

5.1.4 Angle sensor

In this project the accurate control of joint and motor angles was critical, therefore a substantial amount of time was spent finding an optimal solution for fast and accurate angle feedback. The choice landed on a MA302 12-bit contactless angle sensor, a small surface mounted IC. The main reason for this choice was its compact package of 3 mm x 3 mm and the fact that it employs no mechanical parts which would lead to wear and tear. This sensor works by sensing the direction of magnetic fields, meaning angles can be read by placing a magnet in the center of a moving joint.

5.1.5 Current sensor

To control the motor torque with a feedback controller a torque measurement is needed. This can be implemented rather easily using accurate current sensor technology, as current is proportional to torque in most direct current motor applications [Amin and Rehmani, 2015]. A Texas Instruments INA240A4 current sense amplifier was chosen as a consequence of its low interference with the motor commutation circuit. It measures the voltage drop over a shunt resistor with a voltage gain of 200 V/V. Thus, a small resistor can be placed in series with the motor windings with only a small effective loss.

5.1.6 Low level controller

An Espressif DOIT ESP32 DevKit V1 microcontroller, or ESP32 for short, was chosen as the low level controller, or slave node. Its tasks is to control the motor through PWM and handle the low level control of the system. It was chosen mainly because of its 240 MHz clock speed and it possessing 25 general-purpose input/output (GPIO), making it capable of controlling two finger joints. Furthermore it has two fast cores which allow for multiprocessing and use of multiple pins simultaneously, and it supports serial peripheral interface (SPI), inter-integrated circuit (I^2C) and Bluetooth, thus providing multiple connection options for the system.

This microcontroller can be used with the Arduino IDE with the option to run FreeRTOS, a real time operating system. Thus, it allows for the use of concurrency patterns, and implementation of a scheduler. More on this in section 6.1.3

5.1.7 High level controller

For the implementation of the high level controller, or master node, the choice landed on the Raspberry Pi model 3 B. The task of this node is to execute the calculations needed in planning paths and kinematic tasks, therefore a need for more processing power than a microcontroller presented itself. The Raspberry Pi is a single board computer (SBC), meaning it is a fully fledged computer with a quad core 1.2 GHz processor able to run the linux operating system raspbian, which makes it capable of performing the real-time demands of this project. The compilation of C++ code, for example, has a strong community on this SBC.

The Raspberry Pi also comes with 28 ready to use GPIO pins, which allow for multiple communication protocols. This is important as one of the tasks of the master node is to communicate with the slave node. Both SPI and I^2C communication can be implemented using available open-source libraries.

5.2 Communication

The electronics that go into the dexterous gripper is made up of several components with different means of communication. Due to the desired reaction time of the gripper being very quick, the approach has been to choose components that utilize reliable and speedy communication protocols. Some knowledge about the different protocols is necessary to make all components cooperate effectively.

5.2.1 Protocols

The protocols used most excessively in this project are: SPI, I^{2}C and ØMQ.

SPI

serial peripheral interface (SPI) is a four wire communication protocol. It requires four pins to communicate back and forth: master output slave input (MOSI), which outputs data on this pin if it is master and reads data on this pin if it is slave; master input slave output (MISO), which outputs data on this pin if it is slave and reads data on this pin if it is master; clock (CLK), the clock signal which sets the speed of the transfer of data; and chip select (CS), which gets pulled low by the master to signal a communication to another node. A request and reception of data is exemplified in figure 5.1.

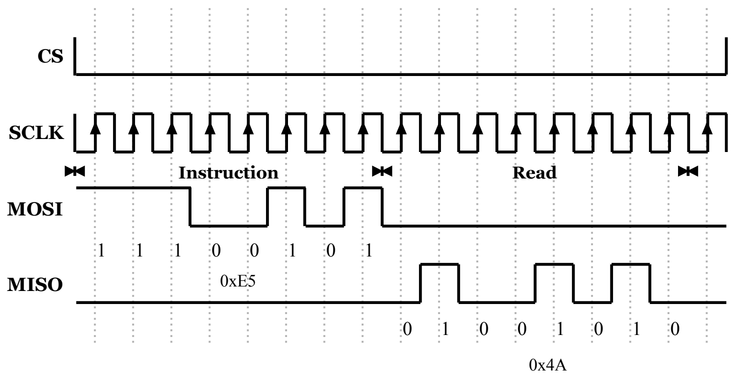


Figure 5.1: SPI transfer example.

Each finger requires two angle sensors in the joints, which each need their own chip select port on the master node, each addition of a finger means that two additional CS pins are needed. The rest of the SPI pins are shared with the other fingers in the system, consequently the time spent communicating will increase and thus the timeslot for other tasks will decrease. The two motors integrated into a finger each have a dedicated SPI port on the slave node.

I^{2}C

I^{2}C is a two wire interface which enables multiple connections through the same ports. Its ports are named SDA for serial data, and SCL for serial clock. The SDA port transfers the data, while the SCL port sets the timing of data transfer. This communication protocol integrates addressing, so all slave nodes in the system need to be defined with an address. The sequencing of data transfer is exemplified in figure 5.2. Each slave node reads only the data addressed to it by the master node, and ignores all others.

Compared to other protocols used in this project, like ØMQ and SPI, this protocol has a low frequency. As the Raspberry Pi only supports a frequency up to 400 kHz. The outer loop becomes slower than the inner loop, though in control theory this is common.

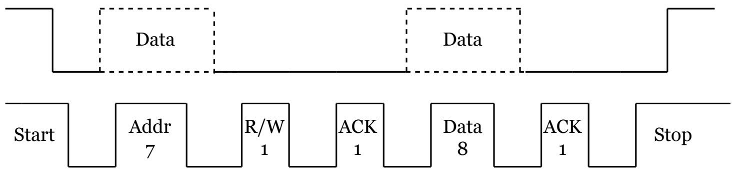


Figure 5.2: Simplified I^{2}C transfer example.

PWM

Pulse-width modulation are signals which are controlled by frequency and duty cycle, as seen in figure 5.3. The frequency defines the length of a cycle, while the duty cycle determines for what proportion of a period the signal is high. PWM is an alternative to analog values in control of electrical motors, as their inertia is inherently slower than the switching of a pwm signal. In microcontrollers PWM signals are generated using clock timers and chopping the signal up into discrete values, causing the average value to be proportional to the duty cycle.

ØMQ

ØMQ, or ZeroMQ, is a library written in C++ which enables high-speed asynchronous communication through several communication protocols (e.g. IPC, TCP, TIPC). When connecting through an Ethernet connection, extremely low latency can be achieved. This protocol can be used in between the GUI PC and master to send position commands and receive info on the state of the system. Section

5.2.2 Communication overview

In figure 5.4 the required components needed to control one joint is shown. A one-joint system is made up of an ESP32 microcontroller, a DRV8313 motor

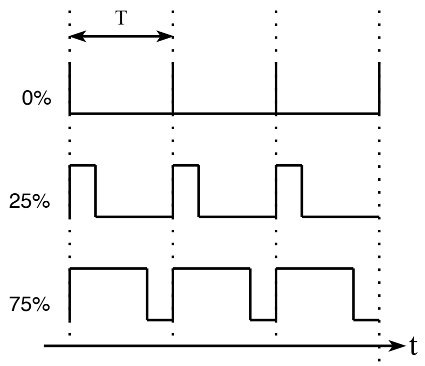


Figure 5.3: PWM transfer example.

driver, a MA302 angle sensor and an INA240 current sensor. The microcontroller sends PWM signals to the motor driver which are converted to a higher power signal with the same frequency and duty signal. The rotation of the motor is sensed by the angle sensor which sends the latest angle value to the microcontroller upon request using the SPI communication protocol. Lastly, the current sensor measures the current through the motor and reads back an analog voltage signal to the microcontroller.

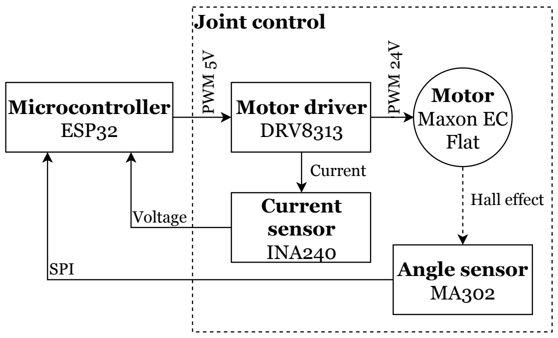


Figure 5.4: Joint control communication diagram.

To control a finger the same setup as in figure 5.4 is reused twice, as seen in figure 5.5. Additionally, a master controller is added to sense the angles and create an outer control loop. Communication between nodes is setup using I^2C as protocol. As with the slave node, the master node also uses SPI to read the angle sensor values.

The specific I^2C connection for a two slave setup, allowing for the control of two fingers, can be seen in figure 5.6. Normally such a connection would require pull-up resistors to bring the signal high. However, both the Raspberry Pi and ESP32 module allow for internal pull-up on the pins used for I^2C communica-

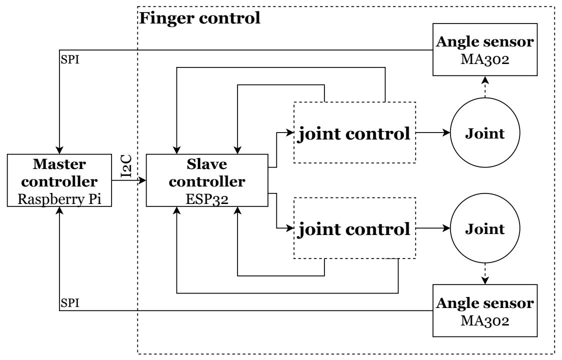


Figure 5.5: Finger control communication diagram.

tion. In this setup the Raspberry Pi has its SDA and SCL pins pulled up with internal 1.8 kΩ resistors, while the pull-ups in the slave nodes are disabled.

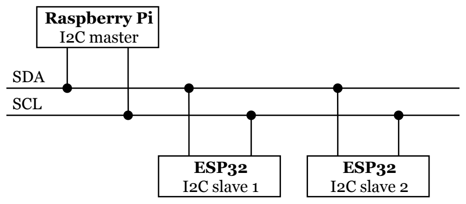


Figure 5.6: I^2C connection method.

5.3 Circuit design

The process of enabling the electronic components to communicate and work well together has led to the creation of three different printed circuit board s: one main control board, dubbed Kiyona, with the purpose of integrating a microcontroller, motor control and communication; a board for the Raspberry Pi, allowing for easier access to the microcomputer's SPI and I^2C ports; and a magnet sensor board, allowing for integration of angle measurements in the dexterous gripper.

5.3.1 PCB design

The ICs MA302, DRV8313, INA240 and MAX6520, as well as resistors and capacitors, all come in surface-mount device (SMD) packages, which means they have to be surface soldered on to the circuit board. AUTODESK EAGLE was used to create the circuit designs, as well as to place the layout of the board.

The reasoning behind creating custom PCBs is that the only commercially available hardware to be found, able of running two BLDC motors with accurate torque and position control were very expensive. Therefore the choice landed on creating a tailored solution that would not necessarily achieve the same level of performance, but would certainly be a great deal cheaper. Furthermore, the creation of a tailored PCB, made the whole system easier to work with, as before its creation the prototype setup consisted of multiple breadboards and a complex interconnection of wires.

Some of the ICs did not have available libraries for placing pads on the board with EAGLE. Therefore, custom libraries had to be created. A quick workflow for achieving this is to simply capture a screenshot from the particular IC's datasheet showing the trace and take note of the spacing between pins. Using the screenshot and the knowledge of the spacing between pins one can overlay the image in the board layout and create a matching trace pad following the outline of the image.

Once a PCB was finished they were all ordered from the manufacturer SEEED. To ensure that the boards were successfully fabricated, computer-aided manufacturing (CAM) files had to be generated from the schematic and board files in software, then uploaded for manufacture. These files include data on where to place traces, the conducive material on a PCB; the placement of holes drilled into the board; and the dimension of the board.

For ordered assembly of PCBs, which was purchased for the last revision of the main control board, the placement of components also have to be defined and linked to a bill of materials (BOM). A BOM is a table containing the desired parts, which is needed to relay what specific components should be soldered on to the board. Using EAGLE, coordinates of the components are specified in a mount SMD file. Meanwhile, the specification of which component belongs to which coordinate is specified in a pdf, showing only the relevant layers of the PCB. A PCB design consists of multiple layers, a way of categorizing the aspects that go into the full design.

Soldering

Solder is applied to the PCB using a stencil. A stencil is a sheet with holes cut out for each surface-mount device trace on a PCB. The soldering job is simplified by aligning the holes of the custom-made stencil and the PCBs pads (the exposed conductive material). When solder is applied onto a flat edge and dragged across the stencil, solder will be applied evenly. The stencil was delivered by the PCB manufacturer SEEED and offered an easier method of applying solder than the alternative of doing it manually with a solder paste syringe or the standard tin solder.

Once solder is applied evenly it needs to be heated up. A good alternative is to use a solder heat gun. By holding the heat gun at a little distance from the component that is to be soldered overheating of it can be avoided, if the heat gun outlet is then brought closer the solder will turn metallic as a sign it has melted. Once the tip is pulled away the solder will quickly cool, leaving the component soldered to the board.

5.3.2 Kiyona, low level control board

To ensure that the circuit would be reliable and compact, a suited printed circuit board was designed. Named Kiyona, the Japanese word for dexterity.

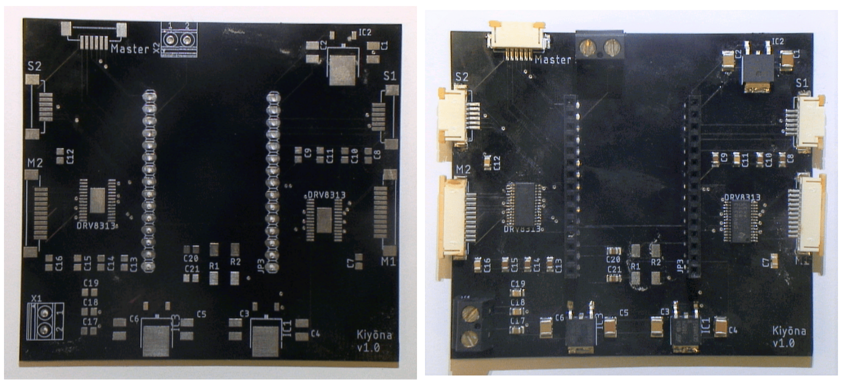


Figure 5.7: Kiyona V1.0.

The design seen in figure 5.7 was the initial design, which upon delivery was found to have critical design errors. Some oversights had been made: the through hole connections made for a female socket on the right side of the board was placed upside down. This would have been no problem if that was the case for the pads on the left side also, but unfortunately that is not the case. This error in design meant that the board was unusable.

As the first design was not usable a second revision was made. The second design also seemed faulty at first. During troubleshooting it was discovered that a crucial trace had not been drawn; the trace between supply voltage from the motor and the input to the motor driver IC. The reason for the error had to do with confusion regarding the naming of the pins in the circuit schematic. Seeing as the circuit diagram is basically one circuit on the left side and a mirrored circuit on the right side it was thought a good idea to create only the first circuit and mirror it. This would have worked well, had it not been for the mislabeling of some of the pins. The traces that were made from the two power supply inputs on the motor driver were lead to the same point but not further to the actual motor supply power on the board. So, hook up wire was soldered onto the board between the two points, that were accessible on the board via two SMD and electrolytic capacitors. This quick fix had the board up and running, without problem.

The third revision adds the traces missing from the previous revision, as well as adding a ground plane. The addition of a ground plane to the board adds shorter ground paths for all ground pads and lessen general noise on the board. Still, the largest change was the addition of current measurement. By utilizing a current measurement IC on the board a higher accuracy in torque control can be achieved.

This revision also saw the usage of both the top and bottom side of the board. By placement of half of the parts on the bottom, the size of the board is reduced by about 25% compared to the previous revision.

Previous versions had 11 pin flat flex cable (FFC) connectors meant for the motors. It was deemed a better option to just use terminal blocks instead, as only three of the connections are needed for connection to the three phase motor. Also removed was the 6 pin SPI FFC connector to the master, as it was instead decided to go with the I^2C protocol for communication between the slave and master node.

This last revision of the board was manufactured and assembled by SEEED, using components available in their warehouse. Parts which could not be found in their catalogue were bought and soldered separately. Specifically, SEEED assembled capacitors, resistors and the motor drivers. The components that were left were voltage regulators, a voltage reference IC, current sensors and the external connection components: FFC cables, terminal blocks and female pin headers.

Microcontroller

The microcontroller integrated onto the Kiyona control board mainly has four external connections: communicating with the master node, sensing the angle of the motor, sensing the current through the motor windings and driving the motor.

On the PCB the microcontroller is incorporated using two row of female headers matching the spacing of the DOIT ESP32 Devkit board. In doing so all the features of the board are kept, meaning that one is still able to connect to the microcontroller and program it. The 3.3V regulator on the board is still operational and the benefit of the on-board flash memory and read-only memory (ROM) is also kept. Overall the reason for the microcontroller being integrated this way, as opposed to using the microcontroller IC that this board is based upon, is that the outcome of this project is aimed at research into dexterous manipulation. The idea is that the researcher experimenting with the dexterous gripper should find no difficulty in flashing the low level controller with new code.

Motor driver

Two motor drivers are situated on the board, each with pins on one side connected to the MCU and the other side connected to the windings of the BLDC motor, visible in figure 5.8 as the HTSSOP package with 28 pins. The side connected to the microcontroller receives three PWM signals, one for each phase of the motor, as seen in figure 5.9 as IN1, IN2 and IN3. It also receives three enable inputs, EN1, EN2 and EN3, each for enabling their respective PWM outputs to the motor. Also connected to the microcontroller are NSLEEP, NRESET and NFAULT, which respectively function to disable the driver, reset the driver and reading out a fault signal if there is a thermal or voltage error.

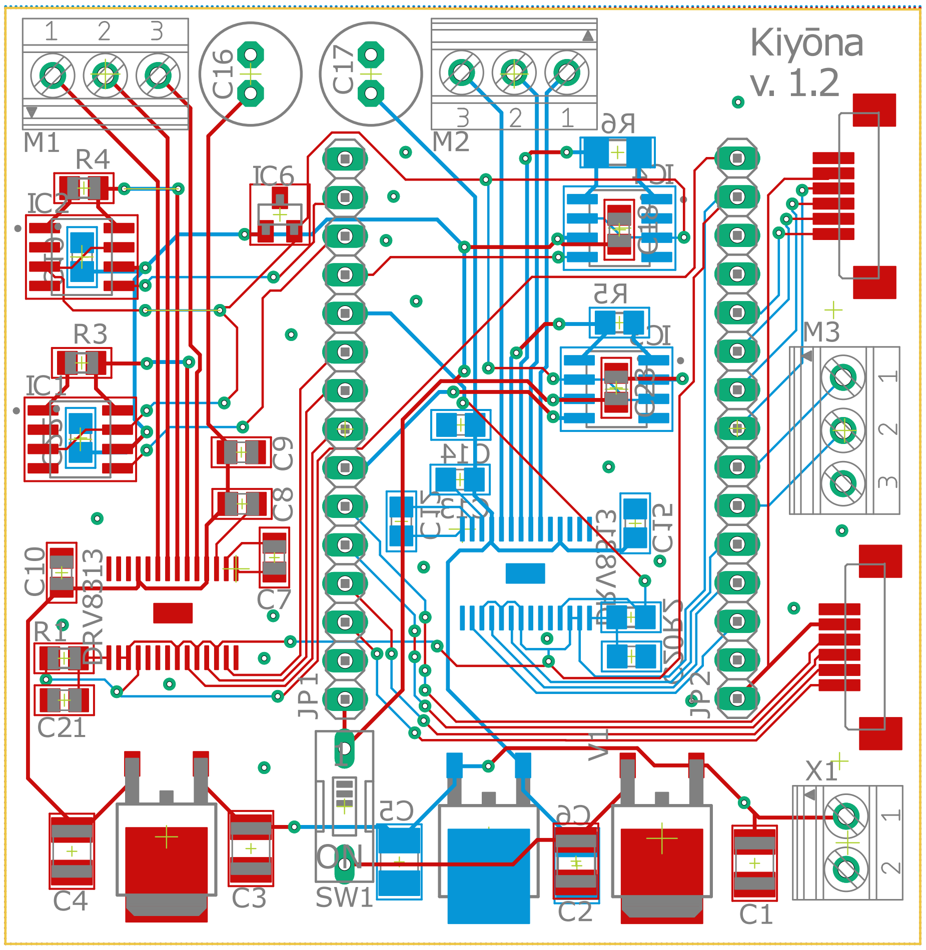


Figure 5.8: Kiyona board, third revision.

The PWM output from the driver is connected to a three pin terminal block with the outputs meant for the motor. A wide assortment of capacitors are also placed close to the driver. The IC needs to be connected to a 0.01 µF capacitor, C7, between its CPH and CPL pin, seen in figure 5.9, acting as a charge pump. C8, connected between motor voltage also acts as a charge pump. Charge pumps are needed as this is a H-bridge driver, and when the centre of a half bridge needs to be brought high the capacitor discharges to bring the voltage higher than the supply voltage on the high side of the FET. The 100 µF electrolytic capacitor, C16, is connected between motor supply voltage and ground and acts as a bulk capacitor. meaning it holds a large charge in the case that the supply voltage drops low and can keep supplying power for a couple milliseconds of time. This capacitor is connected in parallel with another, C10, with a value of 0.1 µF, which acts as a decoupling capacitor. Its job is to keep the voltage up during shorter losses of power. Mixing capacitors with different values is a common practice to mitigate voltage drops that may lead to brown- or blackouts. This is all recommended in the driver's datasheet.[1]

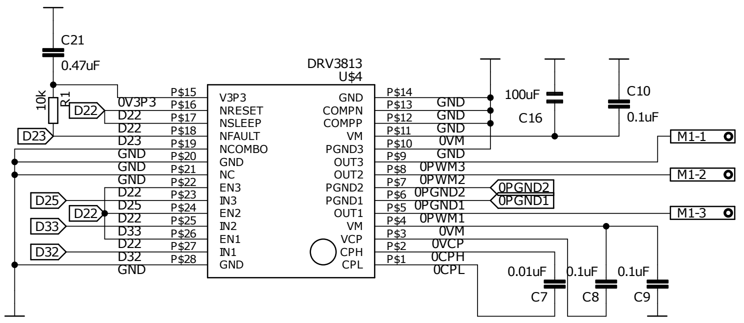


Figure 5.9: Motor driver circuit schematic.

Power supply

Feeding the MCU, motor drivers and angle sensors with the adequate power is important for the operation of the board's components. Three different voltage regulators are therefore mounted on the board, one for powering the MCU and one for powering each of the two motors. The voltage regulator powering the microcontroller, the 7805, delivers 9V/500mA, while the regulators for the motors, two 7824s, output 24V/500mA. All of the voltage regulators get their voltage input from the same power source, meaning one finger and its board only need one off-board power supply for which a 24V/2A supply is recommended. The voltage regulators have ceramic SMD capacitors between input and ground, as well as between output and ground, to reduce transient voltage spikes, as recommended in their datasheet [2].

[1]DRV8313 datasheet available at `http://www.ti.com/lit/ds/symlink/drv8313.pdf`

[2]78xx datasheet available at `https://www.st.com/resource/en/datasheet/l78.pdf`

Heat is a big issues when working with electronics in small packages supplying power to components with high power demands. The main generators of heat on the Kiyona control board are the voltage regulators, as linear voltage regulators work by converting power to waste heat. To avoid this problem the solution has been to simply not run experiments with the gripper for long periods of time.

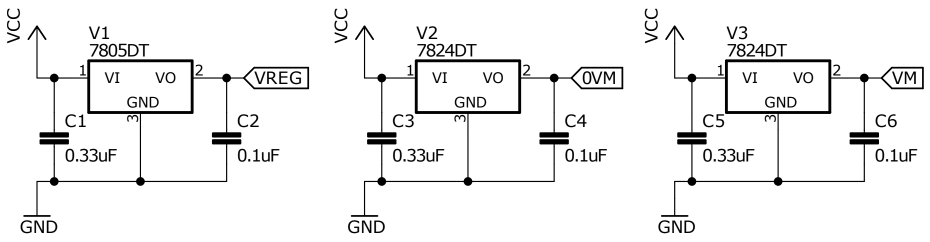


Figure 5.10: Voltage regulator circuit schematic.

Current measurement

Two current measurement ICs is added to the third revision of the PCB, letting the controller sense the current through two of the motor windings. The DRV8313 motor driver is designed for easy addition of current monitoring, as each phase has its own ground output pin. By placing a INA240 Bi-directional current sense amplifier on these pins, the back EMF from the motors can be measured. Two current sensors are attached to each motor, giving four current sensors in total. By measuring the back EMF on two of the phases the third current can be calculated. As Kirchoff's current law states:

$$I_1 + I_2 + I_3 = 0 \tag{5.1}$$

Current is measured by placement of the INA240 IC in parallel with a shunt resistor which the current passes through. For this a 0.01Ω resistor was used, as the current passing through it will be in the range of ± 0-0.5A, giving a voltage range of ± 0-0.005. The INA240 version used is specifically the INA240A4, which provides a 200 V/V gain. Meaning the output voltage from the IC has a range of ±0-1V. The ESP32 is not able to read negative voltages on its ADC, therefore the current sensor IC is given a higher than ground reference, pulling the voltage signal up from ground. By utilizing a MAX6520 voltage reference IC from MAXON the current sensor is supplied a 1.2V voltage reference. Meaning the voltage to be read by the ESP32 is in the range of 0.2 to 2.2V.

The placement of the resistor, IC and traces on the PCB is important to pay attention to. The resistor and IC is placed in a Kelvin connection as recommended in the IC's datasheet[3], meaning they are configured so that current has to pass through the shunt resistor before passing through the current

[3]INA240 datasheet available at: `http://www.ti.com/lit/ds/symlink/ina240.pdf`

sensor. A cutout is made of the PCB's ground plane to ensure that the current is measured through only the resistor before reaching ground, thus allowing for more accurate readings. This is important because of the low value of resistance of 0.01Ω, meaning every millimetre of measurement area has an effect on the reading.

The ESP32 microcontroller features multiple analog-to-digital converter (ADC) pins. Different voltage ranges can be defined in ADC readings, with the standard implementation being readings between 0 and 1V, for the readings on the board it was set to a range between 0 and 3.6V. This gives adequate range for the current sensor output in the range of 0.2-2.2V. The microcontroller can measure with up to 12-bit accuracy, giving 4096 unique values, where 2275 of them is in the effective area. This gives about 1mV/bit. Still, the ADC is not perfectly linear, especially in the area around 0 or 3.6V, which is why these areas are kept out of the effective area. In other words, the current monitor system is not very accurate, but its purpose is only to give a feedback reading to the motor controller.

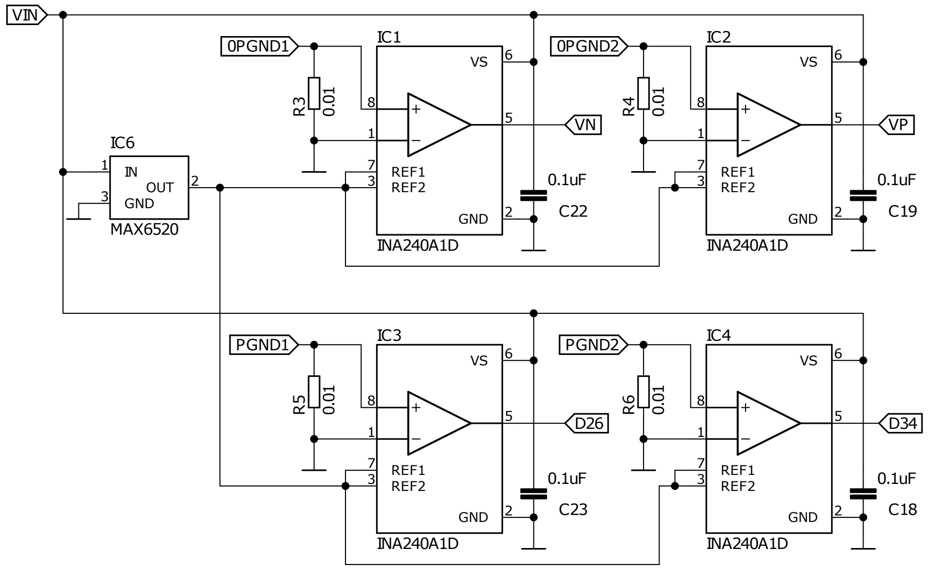


Figure 5.11: Current sensor circuit schematic.

External connections

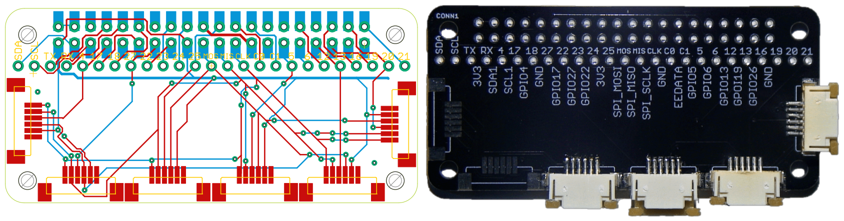
Each microcontroller needs to communicate with five devices that are not mounted on the board. These are the angle sensors that are situated in the joint of the fingers, the motors which are mounted on the gearbox and the master node. The connection with the MA302 contactless angle sensor is added to the board using a six position 1.00mm pitch FFC connector. The connection from the motor driver to the motor is not direct as the motors are placed off the board.

Power to the windings is therefore passed to a terminal block with three connections, allowing for the connection to the finger via any suitable three lead cable. To communicate with the master node I^2C is used. This is integrated on the Kiyona control board using a terminal block with three connections, one for SDA, one for SCL and the last for connecting a shared ground reference.

5.3.3 Raspberry Pi HAT

A raspberry Pi PCB HAT was designed and manufactured for the purpose of easier access to the master node's SPI port, which is seen in figure 5.12 with four out of six SPI ports soldered on. HAT is short for hardware attached on top and denotes a type of board which is placed on top of another piece of hardware to provide some additional functionality. In this case, as the control of the system relies on angle sensor readings, it was deemed necessary to streamline the process of accessing the angle reading from the master node. The design is based upon the Adafruit 16-Channel PWM/Servo HAT schematics and board outline.[4]

The board breaks out the one SPI port to six 6 position FFC connectors, where each uses an exclusive chip select pin, thus enabling the communication with six different SPI sources. I^2C, on the other hand, is incorporated using standard headers placed in the holes in the very left of figure 5.12, SDA and SCL. Seeing as this boards can read six angle readings as well as communicate with the low level controller, each of these boards can support the control of up to three fingers.



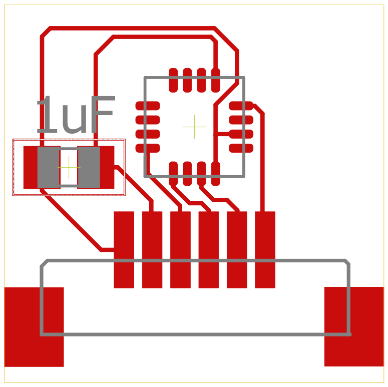
(a) Board outline. (b) Manufactured board.

Figure 5.12: Raspberry Pi HAT.

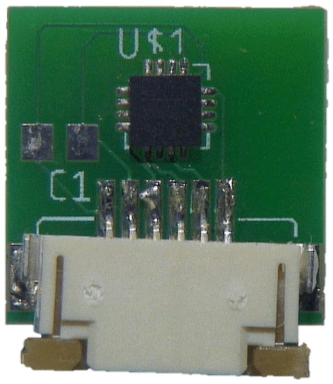
5.3.4 Angle sensor board

Seeing as each finger relies on four of these contactless angle sensors, and as no commercial alternative was available, a custom board for the sensors was made. As can be seen in figure 5.13 it has a small form factor allowing for easy mounting onto the finger and gearbox. As is seen in figure 5.13 only 7 of the pads are connected. They are the four pins needed for SPI, as discussed in

[4]Adafruit 16-Channel PWM/Servo HAT files available at: `https://learn.adafruit.com/adafruit-16-channel-pwm-servo-hat-for-raspberry-pi/downloads`



(a) Board outline.



(b) Physical board.

Figure 5.13: Angle sensor board.

section 5.2.1, as well as power supply and two ground pins. The signals from the IC lead to a 6 pin FFC connector and uses a capacitor pad for soldering on a 1 µF capacitor for reducing noise between supply voltage and the ground reference. The board is made to a dimension of 13.4mm by 13.4mm.

The angle sensor board is placed onto the finger, as discussed in section 3.4.3. After assembly onto the finger and gearbox it was discovered that the design could be improved by creating a two layer design. By moving the capacitor and the connector to the bottom side of the PCB even more space can be saved and the FFC connector would be more accessible.

Chapter 6

Software implementation

This chapter contains a description of the implementation of the control theory from chapter 4 as well as the communication protocols discussed from an electronics perspective in chapter 5. The programming language of choice was C++, which proved to be fast and reliable both in the low level slave controller nodes and the higher level master node.

6.1 Real-time computing

The dexterous gripper is a real-time computing (RTC) system, meaning its is tasked with fulfilling deadlines set by external events. Reacting quickly to its external environment is one of the key elements in making a robotic system dexterous. Through the use of real-time software frameworks such as FreeRTOS and concurrency tools such as mutexes and scheduling a high-performing system can be constructed.

Each finger relies on the control of two motors, one for each joint. The accurate control of two motors requires a fast core to periodically check for the change in reference position or joints as a consequence of external movements. In addition the processing unit needs to communicate with the other nodes in the system. All this creates high demands in the structure of the program. As another option to a fast core taking care of all aspects of controlling two motors, the program can be split across two cores to ensure more to happen simultaneously.

Execution of a program is divided into processes, which are further divided into threads. Processes are the part of a program which own the necessary resources and data. Threads are the part that follows an execution path. A multithreading system offers the ability to run multiple threads within a process. Threads are generally faster than processes and switching between threads within a process is fast. Threads within the same process inherently share resources.

6.1.1 Inter-task communication

The dexterous gripper system, which interacts with the physical world, can be said to be required to produce certain results within a given deadline, which is the definition of a real-time system. It explicitly needs to be able to respond to external interactions within a time-frame for the dexterous gripper project to be considered a success. This creates a need for an operating structure that is flexible and can run a scheduler for periodic actions like reading sensor data, outputting motor drive signals, communicating with other nodes (fingers) in the system, while also being optimized to respond to external actions. An external action can be the sudden jerk of a joint exerted by external forces.

6.1.2 Data race

The program relies on mutex synchronization to avoid the potential for data races. A data race occurs when the output of a task depends on the ordering of events. In the case of the dexterous gripper it is important that the same data registers are not attempted manipulated at the same time. The writing and reading of SPI communication to peripherals are especially exposed to this. Mutexes help avoid data races by locking access to a resource. It is a special data type that holds a value of either zero or one, a the thread that requests to use a data race-exposed resource decrements the related mutex, thus signaling to other threads that this resource is unavailable. Now, when another thread tries to access the resource while it is locked by a mutex, it is forced to wait for the previous thread to finish before the mutex is released and the resource is made available.

When implementing mutexes extra care is taken to avoid that the different threads do not end up in a deadlock. A deadlock occurs when multiple threads are waiting for other threads to release their resource, but they are not releasing their resource before other threads release theirs.[Coulouris, 2012] Thus, the threads are locked in a circular state of waiting. This is visualized in figure 6.1, where the threads, A and B, are waiting for each other to release a resource needed by each respective thread.

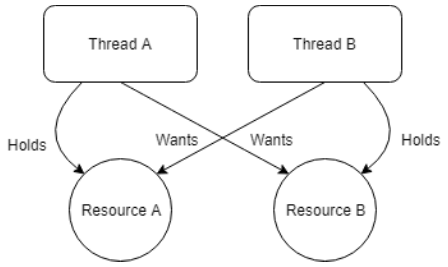


Figure 6.1: Deadlock

6.1.3 FreeRTOS

In addition to supporting the use of the Arduino IDE for programming, the ESP32 chip also has the ability to utilize the tools of the free Real Time Operating System, free RTOS. This operating system is widely used in applications which require multi-core or multithreading utilization of cores. Something which is also very useful when creating a system with real time concerns.

The benefit of using FreeRTOS is that the system can run a scheduler with multiple priorities for different parts of the system, thus allowing for the most critical parts of the system to interrupt less important parts of the system.

As an alternative to mutex locking of threads accessing the same resource simultaneously one can opt to use the FreeRTOS implementation of queues. By using this principle two threads can pass messages between each other to share resources. In the low-level torque control of the motors on the ESP32 one thread is tasked with handling motor control, while another polls the I^2C buffer for new messages. To pass new messages to the motor control thread without interrupting the timing, a queue is set up between these two threads. Messages are passed to the motor control thread when new data is input from the master.

6.2 Slave node: Low level control

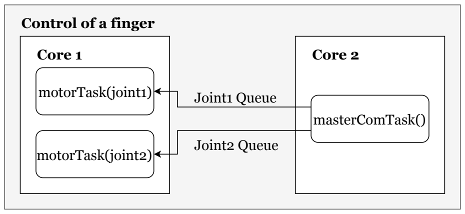


Figure 6.2: Dual core program structure for control of a finger.

The control of the system has been implemented on the system architecture presented in chapter 5. Which, in short, consists of a Raspberry Pi taking commands from a master PC and angle sensor readings from the physical system, while converting master commands into commands for the slave node which actuates the joints via BLDC motors.

As figure 6.2 shows, the low level control software is split between two cores.

6.2.1 Angle sensor reading

Implementation started with the reading of the angle sensor, these commands were inspired by the arduino library written by Monolithic Power[1], the manufacturer of the MA302 angle sensors. Software for this had to be built from

[1]Monolithic Power MagAlpha library available at: `https://github.com/monolithicpower/MagAlpha-Arduino-Library`

scratch seeing as the open library from the manufacturer was not compatible with the ESP32, as it is not a standard Arduino board. Still, the reading of the angle sensor was trivial with the open library as inspiration.

To read the angle of the sensor we need to pass it a command of 8 bits reading `0x00` in hexadecimal or `00000000` in binary over the SPI port. It then reads back 8 bits representing the angle read by the sensor, where `0` represents 0° and `255` is 360°. An alternative is to send 16 bits of only zeroes, whereupon the sensor will read back the angle reading in a 12 bit representation, giving more accuracy.

```
hspi->beginTransaction(SPISettings(spiClk, MSBFIRST, SPI_MODE0));
digitalWrite(H_CS, LOW);
uiAngle = hspi->transfer(0x00);
digitalWrite(H_CS, HIGH);
hspi->endTransaction();
```

Listing 6.1: Implementation of SPI angle reading.

Listing 6.1 shows the implementation of angle reading done in the low level controller. A transaction is begun with a call to the `beginTransaction` member of the SPI class instance, then the chip select pin is pulled low to make the sensor prepare to output an angle reading. Then by sending `0x00`, which is the read angle command, the angle is sent back to the controller. This function blocks and waits for the value to be read back. At last the chip select pin is pulled high and the transaction is ended with a call to the `endTransaction` function.

6.2.2 Motor control

Control of both motors controlling one finger is implemented entirely in software on the low level controller. Based on the angle sensor reading a desired torque is calculated, resulting in a PWM duty cycle output that sets the torque of the motor.

Each of the two motors connected to each microcontroller is setup with three PWM pins, one enable pin and one fault reading pin. The microcontroller's PWM is initialized with a call to `ledcSetup()`, passing the wanted channel, PWM frequency and resolution as arguments. The frequency `PWM_FRQ` is defined as 300 000 Hz, while the resolution is set to 8-bits. The microcontroller supports up to 12-bits of resolution but this comes at the expense of the reduction of maximum frequency. Setup of PWM drivers is defined in the manufacturer's documentation.[2]

```
  ledcSetup(joint->CHN1, PWM_FRQ, PWM_RES);
  ledcSetup(joint->CHN2, PWM_FRQ, PWM_RES);
  ledcSetup(joint->CHN3, PWM_FRQ, PWM_RES);

  ledcAttachPin(joint->PWMU, joint->CHN1);
  ledcAttachPin(joint->PWMV, joint->CHN2);
  ledcAttachPin(joint->PWMW, joint->CHN3);
```

[2]ESP-IDF Programming Guide available at `https://docs.espressif.com/projects/esp-idf/en/latest/api-reference/peripherals/ledc.html`

Listing 6.2: Initialization of PWM.

The angle error reading is detailed in section 6.2.1. It is received as a 8 bit value and further used to calculate the needed PWM signal for optimal control of the motor.

The communicated I^2C reading from the master is read into the motor control function using queues, to avoid any data race condition. The master sends torque commands for both motors, as well as a direction for each of the motors. The read velocity is passed from master to slave as a byte, then scaled to a value between 0.0 and 1.0 upon receival.

```
    // Read master commands
    xQueuePeek(joint->q_scale, &velocity_raw, 0);
    xQueuePeek(joint->q_dir,   &dir,          0);
    velocity = (double)((velocity_raw / 255.0) * 1.0);
```

Listing 6.3: Implementation of PWM output.

As the motor rotates the optimal combination of voltage applied to the windings resemble that of three sine waves with 120° phase shift, as mentioned in section 4.2. This is integrated into the low level controller using a sine lookup table. Fetching data from memory leads to greatly reduced processing time. For fastest possible commutation, implementation of a sine wave was calculated as a 256 byte array in MATLAB. By shifting through this table the microcontroller is capable of speedily calculating the needed sine value for a given change in commutation.

```
    //Output
    pwm_U = (uint8_t)((double)pwmSin[current_step_U] * velocity);
    pwm_V = (uint8_t)((double)pwmSin[current_step_V] * velocity);
    pwm_W = (uint8_t)((double)pwmSin[current_step_W] * velocity);

    ledcWrite(joint->CHN1, pwm_U);
    ledcWrite(joint->CHN2, pwm_V);
    ledcWrite(joint->CHN3, pwm_W);
```

Listing 6.4: Implementation of master queue reading.

6.3 Master node: Modular controller manager for high level control

Although the math behind control algorithms can get very complex, the actual code is typically short and straightforward. The output is simply calculated based on user and sensor inputs. However, to get to this point, sensors must be read, and in the case of this project, the system needs to be able to take commands through a local network connection. Sometimes the gripper will also need more than one controller to be able to complete a task. For example to grab an object, a simple position controller could be utilized to extend the fingers around the object, and then gently grasp the object with a more advanced control strategy. Furthermore, different controllers need vastly different inputs. While a position controller needs a set point for each link, a trajectory controller needs a whole vector of set points. Implementing all these functionalities takes a lot more work than the actual control algorithm itself, but it also increases the re-usability of the end product, and makes the gripper into a platform rather than a static tool. These functions will be referred to as the controller manager. The source code for the controller manager is avaiable on Github [3].

Robotic operating system (ROS) is an open source framework that is capable of all the tasks mentioned above and much more. In fact, the name "controller manager" is taken from a library in ROS with similar functionality. The powerful tools provided by ROS allows to user to focus on a specific problem rather then all the details that gets you there. ROS is popular in academia, and the authors of this thesis has used it before. ROS is however not used as frequently in industry. The common answer to why it is not, is that it lacks real-time support, and that companies often want to make their own lean and specialized software. The prospect of using ROS was discussed with the project's advisor at SINTEF. He preferred not to use it because prior experience with ROS had showed that pure C++ scripts performed far better in applications that required fast reaction time.

6.3.1 Functionalities

The following functionalities is included in the controller manager:

- Automatic calibration routine at start up that sets the finger extended position to 0°.
- Modularity: Run up to 7 2-DOF fingers simultaneously, with separate controllers
- Change controllers at run time.
- User interface over network with two different message types.
  - A "simple instruction message" contains up to 10 float variables (5 states per link) that is used as user input to a controller.
  - A "Trajectory message" contains 10 floats and 5 float vectors with 100 rows each.
- Network broadcast: States of active fingers are broadcast on the network and can be monitored.
- Controller engine: Simplifies adding custom controllers
  - Gives a custom controller access to network and sensor inputs, and passes on the outputs.

[3]Controller manager source code: `https://github.com/Bardie4/Dexterous/tree/master/RaspberryPi`

– Up to 20 variables of a controller can be tuned at run time over network.
– Lets the user focus on control theory.

6.3.2 Network communication

The controller manager receives user inputs, and broadcasts the states of any active fingers on the local network. To implement this, a library called ØMQ was used to send and receive messages, as was recommended by he projects external advisor. The "Ø" in the name is supposed to represent a zero. It is therefore often also referred to as ZMQ or ZeroMQ. Listing 6.5 and 6.6 is a simple example of ØMQ server and client from the official ØMQ website[4].

Line 1 in both scripts begins with including the `zhelpers` library which contains some helping functions for the ØMQ library. In this case "*send*()" and "*send_more*()". `zhelpers.hpp` also includes the `zmq.hpp` library, which is a C++ wrapper for the original C library `zmq.h`.

A connection is initialized by making a context and socket on line 5 and 6. The ØMQ API documentation [5] states that a context must be initialized before using any of the library functions. The argument "1" means that one thread will be dedicated to handle input output operations. The socket object is described in the following way by [Sustrik and Lucina, 2017]:

> "ØMQ sockets present an abstraction of a asynchronous message queue, with the exact queuing semantics depending on the socket type in use."

The socket needs a context, and a socket type. Here a socket of type "publisher" is chosen by using "ZMQ_PUB" as the argument. The are also a number of other socket types to choose from. The choice landed on using a publisher subscriber pattern, which is a "one to many" type of communication. The reason for this is that it's a simple pattern since information only flows one way, and since the publisher does not care if anyone is listening. On line 7 the socket is connected to an endpoint. The argument of `bind()` is a string on the form "`transport//:address.`". In this example, TCP is used as the transport protocol on an ethernet device on port 5563. Note that the name of the Ethernet device is not "eth0" on every computer. `s_sendmore()` is similar to `s_send()`, but includes a flag that tells the receiver that the rest of the message is coming next. Thus, two different messages are sent inside the loop.

```
1  #include "zhelpers.hpp"
2  
3  int main () {
4      //  Prepare our context and publisher
5      zmq::context_t context(1);
6      zmq::socket_t publisher(context, ZMQ_PUB);
7      publisher.bind("tcp://eth0:5563");
```

[4]ØMQ examples found at: `http://zguide.zeromq.org/`
[5]ØMQ Api documentation: `http://api.zeromq.org/2-1:zmq`

```

    while (1) {
        //  Write two messages, each with an envelope and content
        s_sendmore (publisher, "A");
        s_send (publisher, "We don't want to see this");
        s_sendmore (publisher, "B");
        s_send (publisher, "We would like to see this");
        sleep (1);
    }
    return 0;
\
```

Listing 6.5: ZMQ Server

In the client script, the socket type is chosen to be `ØMQ_SUB`, which makes it into a subscriber. This client script uses `.connect()` instead of `.bind()`. The argument is the IP-address of the server, and port number.

The `setsockopt()` function lets the user set options. A message filter is set so that only messages that start with B can be received. The filter size is 1 byte, which is the third argument.

Function `s_recv()` takes an incoming message and turns it into a string. It is a blocking call and will wait until a message arrives. Since B is the filter it will not return on the messages `‘‘A’’` or `‘‘We_don’t_want_to_see_this’’` and keep blocking. The message `‘‘B’’` will arrive, and also `‘‘We_would_like_to_see_this’’` since it is a part of the same message as `‘‘B’’`. They are however captured with separate calls to `s_recv()`.

```
#include "zhelpers.hpp"

int main () {
    //  Prepare our context and subscriber
    zmq::context_t context(1);
    zmq::socket_t subscriber (context, ZMQ_SUB);
    subscriber.connect("tcp://192.168.1.40:5563");
    subscriber.setsockopt( ZMQ_SUBSCRIBE, "B", 1);

    while (1) {

        //  Read envelope with address
        std::string address = s_recv (subscriber);
        //  Read message contents
        std::string contents = s_recv (subscriber);

        std::cout << "[" << address << "] " << contents << std::endl;
    }
    return 0;
}
```

Listing 6.6: ZMQ Client

The information sent in this example was a string. Strings are essentially vectors of characters, and each character represent an 8 bit value, or a byte. This means that ØMQ helps transmit a series of bytes across the network. The variables that are needed for this project are not necessarily 8 bit values. A method to turn any variable into a series of bytes, and then translating back to its original form is therefore required. This process is called serialization []. The SINTEF project advisor recommended a library called flatbuffers.

6.3.3 Flatbuffer schemas

The first step in utilizing the flatbuffers library, is creating a schema. A schema acts as a blueprint for a data structure that can be serialized. The schema is then compiled, which produces a header file that contains functions to build the data structure, serialize it, and deserialize it. Details on the process of compiling schemas can be found in the official flatbuffers tutorial[6].

The schema for a message is shown in listing 6.7. It is called `SimpleInstructionMsg`, because it is the smallest message used in this project. On line 1, a name space is chosen. All functions generated by compiling the schema will use this name space. Next, a table is created containing 12 variables. On line 18, it is specified that `SimpleInstructionMsg` is the root type. The root type is the variable that is serialized. The root type must therefore contain all the data fields specified in the schema. In this example there is only one table. Thus, it must be the root type. Lastly, a file identifier is included. A file identifier makes it possible to identify this message type before de-serializing it.

```
1  namespace my_schemas;
2  
3  table SimpleInstructionMsg {
4      finger_select:short;
5       controller_select:short;
6       data1:float;
7       data2:float;
8       data3:float;
9       data4:float;
10      data5:float;
11      data6:float;
12      data7:float;
13      data8:float;
14      data9:float;
15      data10:float;
16 }
17 
18 root_type SimpleInstructionMsg;
19 file_identifier "INST";
```

Listing 6.7: ZMQ Client

6.4 Controller manager code overview

The controller manager is a multithreaded application that is mostly made up of four different classes. The `Finger`, `ControllerEngine`, `ZmqSubscriber` and

[6]Flatbuffers C++ tutorial available at: `https://google.github.io/flatbuffers/flatbuffers_guide_tutorial.html`

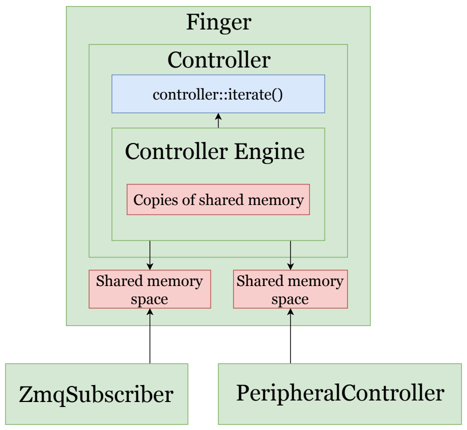


Figure 6.3: `Controller_Engine`

`PeripheralsController` class. The diagram in figure 6.3 gives an overview of the program structure. In this figure, class objects are colored green, memory space is colored red, and functions are blue. Black arrows represent pointers. There can be multiple instances of the `Finger` class, but only one `ZmqSubscriber` and `PeripheralsController`. Every controller contains its own instance of a `ControllerEngine`, and a `Finger` object can contain many controllers. The `Finger`, `ZmqSubscriber` and `PeripheralsController` all have a member function named `run()`, that runs on a separate thread. `Finger::run()` threads can be spawned and stopped based on user input. The`ZmqSubscriber::run()` and `PeripheralsController::run()` threads operate continuously while the program is active. Communication between threads happens over memory shared by the threads. The shared memory is a member of the `Finger` class, and is protected with mutex locks. From Figure 6.3 it can be seen that the `ZmqSubscriber`, `PeripheralsController` and `ControllerEngine` has access to this memory through pointers.

The `ZmqSubscriber` object listens for commands on the local network. Every message contains the identity of the `Finger` object which the message is meant for, the selected controller, and the controller input, or "payload". The `ZmqSubscriber` directs these messages to the correct finger. It is also responsible for spawning the `Finger::run()` threads, and does so if the finger is given a command while it's not already running. The `PeripheralsController` supplies the fingers with sensor information, and relays the controller outputs to the ESP32 over I^2C. It also contains a ØMQ publisher so that sensor data can be monitored on the local network. Another important task of the `PeripheralsController` is to set the pace of the controllers. Controllers have to wait for sensors to be read before each iteration, and measurements are done at a reasonably steady rate. The purpose of the `ControllerEngine` is to simplify the process of adding new controllers. From figure 6.3 it can be seen that a controller class contains a function called `iterate()`, and an instance of `ControllerEngine`.

The `iterate()` function performs one controller iteration. The `Controller-Engine` has access to this function through a pointer, and calls it when it is appropriate. It also provides the controller with updated copies of information from the shared memory space that is safe to use in the `iterate()` function.

6.4.1 The `Finger` Class

The definitions of the `Finger` class is shown in listing 6.8. A finger object uses a number from 0 to 6 as an identity. All fingers must have a unique identity. The identity enables the `ZmqSubscriber` to send commands to the correct finger. The identity is also associated with physical SPI chip select pins, and I^2C addresses. It also helps the user identify sensor data broadcasted on the network. The reason for having a maximum of 7 fingers was the amount of available GPIO pins that could be used as chip select for SPI. Communication with the ESP32 was initially planned as an SPI connection, and each finger needed a total of 3 chip select pins. After moving ESP communication to I^2C, there were no hardware limitations to adding a few more. However, it was concluded that having pins available for other sensors would most likely be more useful than adding another finger. The identity is set during initiation of a `Finger` object, and can be seen on line 12 as the argument of the `Finger` constructor.

The shared memory that fingers use to communicate with `ZmqSubscriber` and `PeripheralsController` is standardized by using two different structs called `ZmqSubFingerMem`, and `PeripheralFingerMem`. They are initiated as a class member on line 5 and 6. On line 8-9, controller objects are added. Controller objects contain member functions that perform the actual calculations. Controller objects also has their own identities which is given by the `bindController()` function. This function is called in the constructor of the finger, and gives controllers access to the shared memory that provides them with sensor data and user inputs. Further details on controller classes will be discussed in section 6.4.4.

```
1  class Finger{
2      public:
3          short id;
4          short controllerSelect;
5          ZmqSubFingerMem zmqSubSharedMem;
6          PeripheralFingerMem periphSharedMem;
7
8          //Controllers
9          JointSpacePosController jsPosCntrllr;
10         CartesianSpacePosController cPosCntrllr;
11
12         void bindController(ControllerEngine* handle, short controller_id);
13         Finger(short identity);
14         void calibration();
15         void shutdown();
16         void* run();
17 }
```

Listing 6.8: Finger class definition

The member function `.run()` is shown in listing 6.9. Unless it is already running, this function is initiated on a new thread by the `ZmqSubscriber` in the event of a new message directed at this particular Finger object. On start up, the calibration routine is initiated (line 2). The calibration routine drives the finger joints to the end position, to relate the sensor readings to the physical position of the finger. During calibration, the finger communicates directly with sensors without the help of the `PeripheralsController`. At the end of the calibration routine, it sends a flag to the `PeripheralController` that lets it know that it can proceed to communicate with the sensor, and relay the information to the finger via `periphSharedMem`.

On line 3-5, a private copy of the the `controllerSelect` variable is made from the shared memory. As the name suggests, this variable is used to tell the finger which controller to use. If it is 0, it means that the finger should exit the main loop and shut down. The `.shutdown()` function tells the `PeripheralController` that it can stop communicating with the sensors associated with this particular finger. Also, it tells the ZmqSubscriber that this thread is no longer active, so that it knows that the `.run()` function needs to be restarted if a new command arrives.

Inside the loop, the controllers themselves will check if `controllerSelect` corresponds to their identity, and if not, they will exit and move on to the next one. At the end of the loop, a 300µs pause is added. This prevents excessive use of `pthread_mutex_lock` in the event of a non existing controller being selected.

```
1  Finger::run(){
2   calibration();
3 
4   pthread_mutex_lock(&zmqSubLock);
5   controllerSelect = zmqSubSharedMem.controllerSelect;
6   pthread_mutex_unlock(&zmqSubLock);
7 
8   while( !(controllerSelect == 0) ){
9    jsPosCntrllr.controllerEngine.run();
10   cPosCntrllr.controllerEngine.run();
11   pthread_mutex_lock(&zmqSubLock);
12   controllerSelect = zmqSubSharedMem.controllerSelect;
13   pthread_mutex_unlock(&zmqSubLock);
14   usleep(200);
15  }
16  shutdown();
17 }
```

Listing 6.9: Finger class definition

6.4.2 The `ZmqSubscriber` class

The purpose of the `ZmqSubscriber` class is to receive commands via network, and send the information to `Finger` objects. Figure 6.4 illustrates the interactions between a `Finger` object and a `ZmqSubscriber` object, and how the shared memory space is utilized. In this figure, class objects are colored green, memory space is red, functions are blue, and scripts are yellow. Black arrows represent the ability to read/write to a memory space. Blue arrows represent the ability to call a function. Stifled arrows represents a network message.

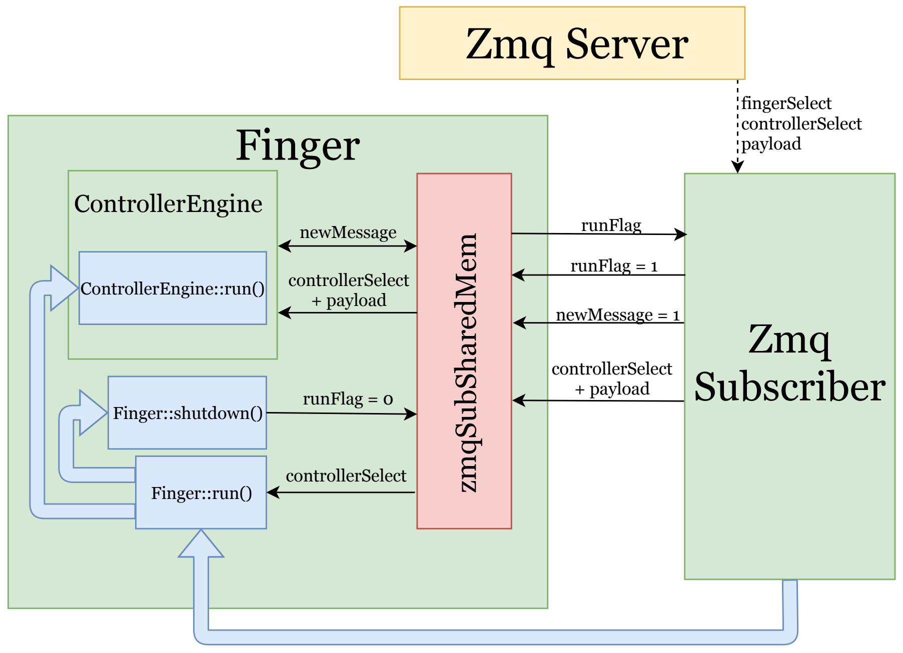


Figure 6.4: Interactions between the `ZmqSubscriber` and Finger class object.

The `ZmqSubscriber` object receives a message from the server containing a payload of controller inputs addressed to a specific controller (`controller-Select`), in a specific `Finger` (`fingerSelect`). The ZmqSubscriber will then read the `runFlag` variable of the selected `Finger`, set it to one, set `newMessage` to one, and write `controllerSelect` and payload to the shared memory.

If `runFlag` was set to zero before it was overwritten, it means that there is no active thread running the `Finger::run()` function, and therefore no one to receive to message. In this case, the `ZmqSubscriber` will call the Finger::run() function on a new thread.

The `Finger` object itself will only read the `controllerSelect` variable. If it is 0, no controllers is selected, and the `Finger::shutdown()` function is called to terminate the thread and set `runFlag` to 0. When `controllerSelect` is not 0, `Finger::run()` will cycle through controllers one by one. Controllers will exit on their own if they are not selected.

The `newMessage` variable is set to zero whenever a controller reads the shared memory. This is done to stop controllers from reading large trajectory messages on every iteration. If a controller reads a message and finds out it is intended for another controller, the `newMessage` variable is set to one again, to make sure that other controllers will read it.

```
 1 class ZmqSubscriber{
 2
 3   private:
 4   ZmqSubFingerMem* fingerMemPtr[7];
 5   ZmqSubFingerMem fingerMem;
 6   Finger* fingerPtrs[7];
 7
 8   bool oldRunFlag;
 9
10   //ZMQ
11   zmq::context_t context;
12   zmq::socket_t subscriber;
13   char* address[5];
14
15  public:
16   ZmqSubscriber()
17   void bindFinger(Finger* finger)
18   void passOnSimpleInstructions(zmq::message_t* buffer)
19   void passOnTrajectoryMsg(zmq::message_t* buffer)
20   static void *initFinger(void *finger_object)
21   void* run()
22 }
```

Listing 6.10: ZmqSubscriber class definition

The class definition of the `ZmqSubscriber` class is shown in listing 6.10. On line 4, a vector containing 7 pointers to data structures of type `ZmqSubFingerMem` is defined. These will point to the shared memory of the `Finger` objects, that was shown in figure 6.4. The function `.bindFinger()` on line 15 will set these pointers. When a `Finger` is "bound", the identity of the `Finger` will determine the position of the pointer in the vector. For example, a `Finger` object with identity 3 will have its pointer placed in `fingerMemPtr[3]`.

The `fingerMem` variable (line 5) is where the incoming data is stored before it is sent to a `Finger`. On line 6, a vector of 7 pointers to class objects of type `Finger` is defined. These pointers are needed to gain access to the function `Finger::run()`, so that the `ZmqSubscriber` is able to activate a `Finger`. These pointers are also set by `.bindFinger()`. The `.initFinger()` function (line 20) is the function that calls `Finger::run()`.

```
ZmqSubscriber.run(){
 while(1){
 zmq::message_t address;
 zmq::message_t buffer;
 subscriber.recv(&address);
 subscriber.recv(&buffer);

 if ( SimpleInstructionMsgBufferHasIdentifier( buffer.data() ) ){
  passOnSimpleInstructions(&buffer);
 }else if ( TrajectoryMsgBufferHasIdentifier( buffer.data() )) {
  passOnTrajectoryMsg(&buffer);
 }
}
```

Listing 6.11: ZmqSubscriber::run()

The `ZmqSubscriber::run()` function shown in listing 6.11 will run continuously on its own thread while the controller manager is active.

The function starts out similar to the ØMQ subscriber example in listing 6.6. However, the `s_recv()` function from the `zhelpers.h` library is not used. The `zmq.hpp` library is used directly instead. This is because the `s_recv()` function includes the unnecessary step of turning the message into the data type `std::string`. Aside from that, line 3-6 in listing 6.11 is identical to calling `s_recv()` twice.

In section 6.3.3 it was mentioned that a `file_identifier` was included in the flatbuffer schema to make it possible to identify a message type before de-serializing it. The function `SimpleInstructionMsgBufferHasIdentifier()` does exactly that, and returns 1 if the incoming message is of the type "`SimpleInstructionMsg`". This function is one of many functions that is generated by compiling a flatbuffer schema. The names of the generated functions always starts out with the name of the root type. In Line 7-10, the member function `passOnSimpleInstructions()` or `passOnTrajectoryMsg` is called based on what type of message is received. These functions de-serialize the message and pass it on to the correct `Finger` object.

```

ZmqSubscriber::passOnSimpleInstructions(zmq::message_t* buffer){
 auto messageObj = GetSimpleInstructionMsg(buffer->data());

 fingerMem.fingerSelect = messageObj->finger_select();
 if ( (fingerMem.fingerSelect < 0)  (fingerMem.fingerSelect > 6) ){
  return;
 }
 if (fingerMemPtr[fingerMem.fingerSelect] == NULL){
  return;
 }

 fingerMem.controllerSelect = messageObj->controller_select();;
 fingerMem.data1 = messageObj->data1();
 ...
 fingerMem.data10 = messageObj->data10();

 pthread_mutex_lock(&zmqSubLock);
 oldRunFlag = fingerMemPtr[fingerMem.fingerSelect]->runFlag;
 *fingerMemPtr[fingerMem.fingerSelect] = fingerMem;

 if (oldRunFlag == 0){
  pthread_create(&(tid[2+fingerSelect]), NULL, &initFinger, fingerPtrs[fingerMem.
     fingerSelect]);
 }
 pthread_mutex_unlock(&zmqSubLock);
}
```

Listing 6.12: ZmqSubscriber::passOnSimpleInstructions()

The code of `passOnSimpleInstructions()` is shown in listing 6.12. Note that the only difference between this function and `passOnTrajectoryMsg()`, is the payload that is transferred.

The message is first de-serialized using the generated function GetSimple-InstructionMsg(). Before loading the whole message, the fingerSelect variable is verified (line 4-10). If it is not between 0-6, or it refers to a Finger object that is not yet bound to the ZmqSubscriber, the message is discarded. If the Finger identity is valid, the message is loaded into the private variable fingerMem.

The next part (line 24-31) involves the Finger objects shared memory space, and a mutex lock is required. The .runFlag is read from the shared memory before it is overwritten. If it is 1, it means that the finger is active. If it is 0, it needs to be activated. Regardless of whether it is active or not, the message is passed to the finger on line 26. Then, if the Finger object was inactive, the initFinger() function is called on a new thread, with the address of the Finger object as the argument. This function will then run the member Finger::run() function to active the Finger object.

6.4.3 The PeripheralsController class

The purpose of the PeripheralsController class is to supply controllers with sensor information, relay the calculated output of controllers to the ESP32, and set the pace of controller iterations. Figure 6.5 illustrates the interactions between a Finger object and a PerpipheralsController object, and how the shared memory space is utilized. The class objects are colored green, memory space is red, functions are blue, and scripts are yellow. Black arrows represent the ability to read/write to a memory space. Yellow arrows represent I^2C communication. Red arrows represent SPI communication. The purple arrow represents a condition signal from the pthreads library. Condition signals can be used to create a conditional block on other threads. Here, the Peripherals-Controller can remove the block. Details on this functionality can be found online [7].

The first thing to note from figure 6.5, is that the shared memory space is not the same used to communicate with the ZmqSubscriber. By extension, the runFlag, is not the same variable either. The PerpipheralsController will check the runFlag of each Finger, and proceed to communicate with the sensors of those with runFlag set to 1.

As mentioned earlier, the calibration routine Finger::calibration() is executed first whenever the Finger::run() function is called by the Zmq-Subscriber object. It is only after calibration that runFlag is set to 1. The reason behind this is that the calibration routine communicates directly with the ESP32, and angle sensors. Any other communication with the sensors during calibration will disturb the process. The communication with peripherals, is also the bottleneck of the controller manager. Unused sensors should therefore not be read. The Finger::shutdown() function which is called before the Finger::run() thread terminates will also set the runFlag to zero to stop further measurements and communication with the ESP32.

[7]Documentation of conditional broadcast available at: https://pubs.opengroup.org/onlinepubs/009695399/functions/pthread_cond_broadcast.html

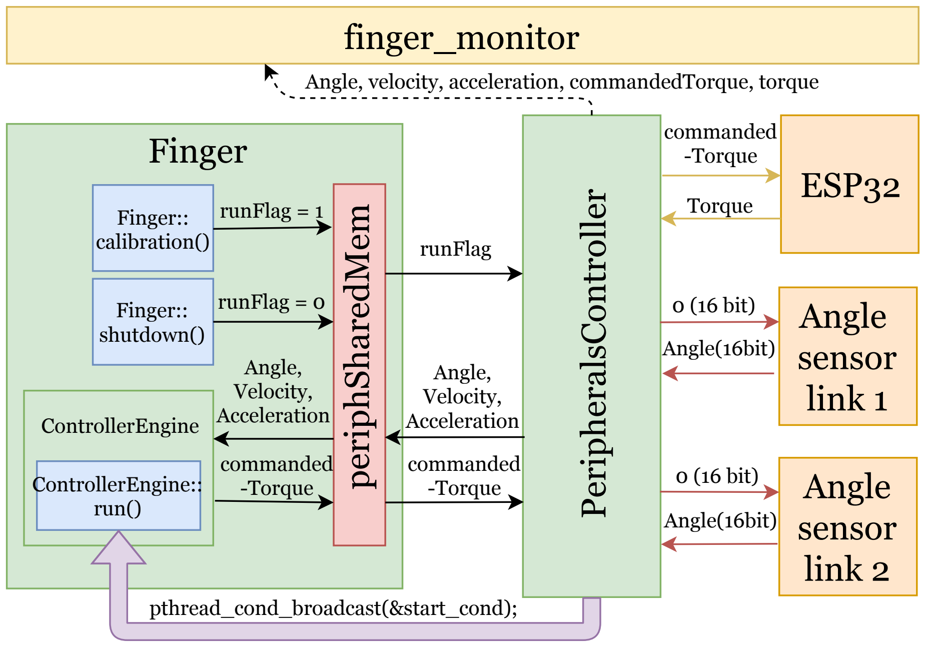


Figure 6.5: Interactions between the `PeripheralsController` and a `Finger` class object.

Angular velocity and angular acceleration is calculated simply by differentiating the angle inputs. This method is known amplify noise [Olfa et al., 2016]. Given more time, an observer would have been implemented instead.

The `ControllerEngine` takes care of the difficult parts of integrating a controller into the controller manager. In relation to the `PeripheralsController`, this means accessing the shared memory and performing iterations at the right pace. Figure 6.6 illustrates the interactions between the `PeripheralsController` and `ControllerEngine`. The `ControllerEngine::run()` function is blocked before sensor information is read from the shared memory space. The block is released by the `PeripheralsController` after sensors are read, with a call to `pthread_cond_broadcast(&start_cond)`. The condition variable `&start_cond` is a global variable used by all controllers. All controllers is therefore released at the same time.

6.4.4 Adding a controller

To help users implement their own controllers, a template for a controller class was developed. This enables the user to create a new controller simply by renaming a few pointers, adding their own variables and writing a controller function.

The class definition is showed in listing 6.13. The only non optional lines in

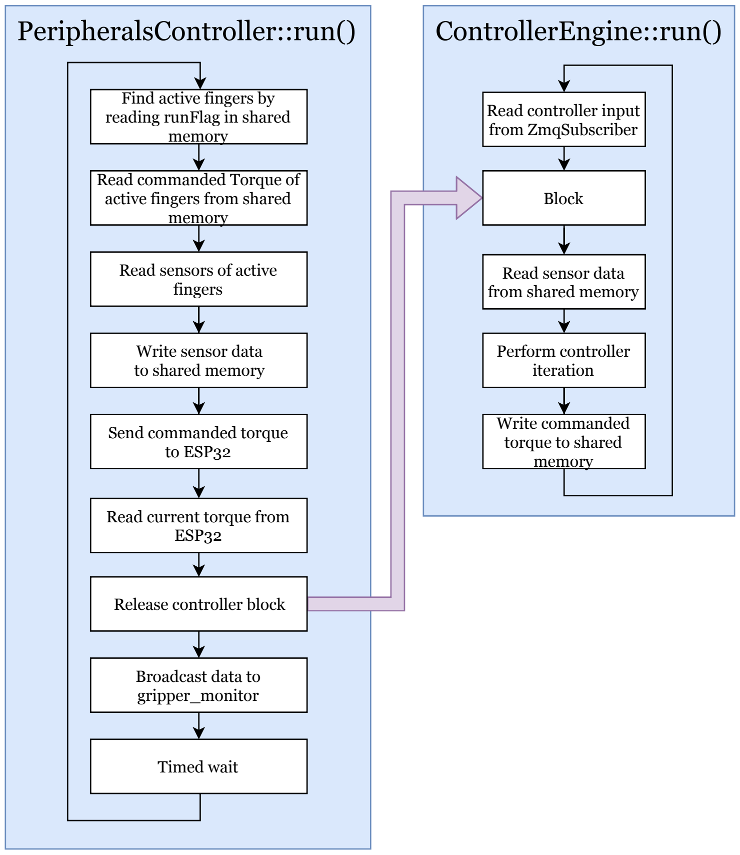

PeripheralsController::run()
Find active fingers by reading runFlag in shared memory
Read commanded Torque of active fingers from shared memory
Read sensors of active fingers
Write sensor data to shared memory
Send commanded torque to ESP32
Read current torque from ESP32
Release controller block
Broadcast data to gripper_monitor
Timed wait
ControllerEngine::run()
Read controller input from ZmqSubscriber
Block
Read sensor data from shared memory
Perform controller iteration
Write commanded torque to shared memory

Figure 6.6: Interactions between the `PeripheralsController::run()` and `ControllerEngine::run()` function

this class definition is the `ControllerEngine` (line 4), and the member functions (line 25-28). Everything else is pointers that will point to the memory of the `ControllerEngine`. The sole purpose of these pointers is only to rename variables. The variables held by the `ControllerEngine` can in fact be used directly without pointers if that is desirable. The reason a name change might be relevant, is that incoming network data has different meaning for different controllers. For example, the `SimpleInstructionMsg` shown in listing 6.7 contains 10 float variables named "data1-10". To one controller the variable `data1` can contain an angle set point. To another controller `data1` may contain a Cartesian coordinate. The `ControllerEngine` also has 20 member variables that can be changed during run time. These variables are suitable for controller parameters like gains, and offsets.

```
1
2  class ControllerTemplate {
3    public:
4      ControllerEngine controllerEngine;
5
6      //ZmqSubscriber data
7      float *data1, *data2, *data3, *data4, *data5, *data6, *data7, *data8, *data9,
         *data10;
8      float *var1, *var2, *var3, *var4, *var5, *var6, *var7, *var8, *var9, *var10,
        *var11, *var12, *var13, *var14, *var15, *var16, *var17, *var18, *var19, *
        var20;
9      int *trajSize;
10     float* trajTimeStamp;
11     float* trajPosition;
12     float* trajVelocity;
13     float* trajAcceleration;
14
15     //PeripheralsController data
16     float *jointAngle1;
17     float *jointAngle2;
18     float *angularVel1;
19     float *angularVel2;
20     float *angularAcc1;
21     float *angularAcc2;
22     float *commandedTorque1;
23     float *commandedTorque2;
24
25     static void iterateStatic(void *controller_object);
26     ControllerTemplate();
27     ControllerEngine* getHandle();
28     void iterate();
29 };
```

Listing 6.13: Template for a controller class

The pointers defined from line 9 to 23 in listing 6.13, are less likely to need a name change. However, they are useful in the template because they reveal what information is available in the `ControllerEngine`.

The 4 member functions are shown in listing 6.14. Line 6-59 is the constructor of the template controller class. Here, all the optional pointers are assigned. But more importantly, the variables available in `ControllerEngine` is revealed for the user without having to look up the class definition of the `ControllerEngine`. Line 7-10 is the only non optional part of the constructor. On line 7, the `ControllerEngine` is initialized. On line 8-9, the `ControllerEngine` gains access to the `iterateStatic()`, function, and a pointer to this specific instance of the class. Line 10 specifies that the controller will not be receiving trajectory messages. For a trajectory controller, it is set to one.

The `getHandle()` function (line 66-68) simply returns the address of the `ControllerEngine`. It is used as an argument to the `Finger::bindController()` function, and should remain unchanged.

The `iterate()` function (line 61-64) will contain the actual controller code. In the template, the output set equal to the input.

It is the `ControllerEngine` that decides when `iterate()` is called. However, The `ControllerEngine` can't store a pointer to this function, because it would need to know the definition of this specific controller class. To solve this problem, a static function called `iterateStatic()` (line) is used instead. Static member functions are just like any other functions, and doesn't have access to any class members. The `ControllerEngine` is therefore able to store a pointer of this function and call it when needed. Since the definition of the controller class is available to the `iterateStatic()` function, it is able to call the `iterate()` function as long as it's given a pointer to the specific instance of the class.

```
1  void ControllerTemplate::iterateStatic(void *controller_object){
2    return ((ControllerTemplate*)controller_object)->iterate();
3  }
4
5  ControllerTemplate::ControllerTemplate()
6    :controllerEngine(){
7    controllerEngine.controllerObject = this;
8    controllerEngine.iterate = &JointSpacePosController::iterateStatic;
9    controllerEngine.trajectoryMessage = 0;
10
11   //ZmqSub inputs
12   data1 = &controllerEngine.data1;
...
22   data10 = &controllerEngine.data10;
23   trajSize = &controllerEngine.trajSize;
24   trajTimeStamp = controllerEngine.trajTimeStamp;
25   trajPosition = controllerEngine.trajPosition;
26   trajVelocity = controllerEngine.trajVelocity;
27   trajAcceleration = controllerEngine.trajAcceleration;
28   //Peripheral inputs
29   jointAngle1 = &controllerEngine.jointAngle1;
30   jointAngle2 = &controllerEngine.jointAngle2;
31   angularVel1 = &controllerEngine.angularVel1;
32   angularVel2 = &controllerEngine.angularVel2;
33   angularAcc1 = &controllerEngine.angularAcc1;
34   angularAcc2 = &controllerEngine.angularAcc2;
35   //Controller output
36   commandedTorque1 = &controllerEngine.commandedTorque1;
37   commandedTorque2 = &controllerEngine.commandedTorque2;
38   //Run time adjustable variables (example: Kp, Ki and so on..)
39   var1 = &controllerEngine.var1;
...
```

```
  var20 = &controllerEngine.var20;
}

void ControllerTemplate::iterate(){
  *commandedTorque1 = *data1;
  *commandedTorque2 = *data2;
}

ControllerEngine* ControllerTemplate::getHandle(){
  return &controllerEngine;
}
```

Listing 6.14: Member functions of template controller

To make the controller available to the `Finger` class, it must be added as a class member. An example of this was shown in listing 6.8 line 9-10, where two controllers were added as members. Listing 6.15 shows the constructor of the `Finger` class. Here controller objects are initiated on line 2, before they are bound on line 4 and 5. The `bindController()` function assigns an identity to a controller, and gives it access to the shared memory.

```
1 Finger(int identity)
2   :jsPosCntrllr(), ctPosCntrllr(){
3   id= identity;
4   bindController(jsPosCntrllr.getHandle(), 2);
5   bindController(ctPosCntrllr.getHandle(), 3);
6   zmqSubSharedMem.runFlag=0;
7   periphSharedMem.runFlag=0;
8 }
```

Listing 6.15: Member functions of template controller

Finally, the `ControllerEngine::run()` function is added to loop in `Finger-::run()`. How the existing controllers were added to the loop can be seen in listing 6.9 line 9-10.

6.5 Master-slave I^2C communication

Communication had to be set up between the master node (Raspberry Pi) and the slave node (ESP32). Being able to send commands from master to slave is crucial in control of the dexterous gripper system. With the Raspberry Pi and the ESP32 multiple choices were available, among them are I^2C, SPI, serial communication, Bluetooth and WiFi. Of these I^2C and Bluetooth are implemented, with SPI having an available port on the Kiyona PCB.

As presented in section 5.2.1, I^2C is a two wire protocol with the ability for addressing data. On the ESP32 side the built-in esp-idf `i2c-driver` library is used with the microcontroller configured as slave as seen in listing 6.16. Using the I^2C protocol requires pull-up resistors on each of the two wires, but luckily the ESP32 comes with built in pull up resistors which can be toggled in code using the `GPIO_PULLUP_ENABLE` keyword. After all configuration parameters are set in the `conf_slave` object, the I^{2}C driver is configured using `i2c_param_config`, and lastly the driver is installed using `i2c_driver_install`.

```
i2c_slave::i2c_slave(){
  i2c_port_t i2c_slave_port = I2C_EXAMPLE_SLAVE_NUM;
  i2c_config_t conf_slave;
  conf_slave.sda_io_num = I2C_EXAMPLE_SLAVE_SDA_IO;
  conf_slave.sda_pullup_en = GPIO_PULLUP_ENABLE;
  conf_slave.scl_io_num = I2C_EXAMPLE_SLAVE_SCL_IO;
  conf_slave.scl_pullup_en = GPIO_PULLUP_ENABLE;
  conf_slave.mode = I2C_MODE_SLAVE;
  conf_slave.slave.addr_10bit_en = 0;
  conf_slave.slave.slave_addr = ESP_SLAVE_ADDR;
  i2c_param_config(i2c_slave_port, &conf_slave);
  i2c_driver_install(i2c_slave_port,
                     conf_slave.mode,
                     I2C_EXAMPLE_SLAVE_RX_BUF_LEN,
                     I2C_EXAMPLE_SLAVE_TX_BUF_LEN,
                     I2C_NUM_1);
}
```

Listing 6.16: I^{2}C configuration.

On the Raspberry pi side it is implemented using the C library `pigpio`[8]. The master side of the communication controls the speed of transfer, set at 400kHz.

6.6 ØMQ publisher with GTK GUI

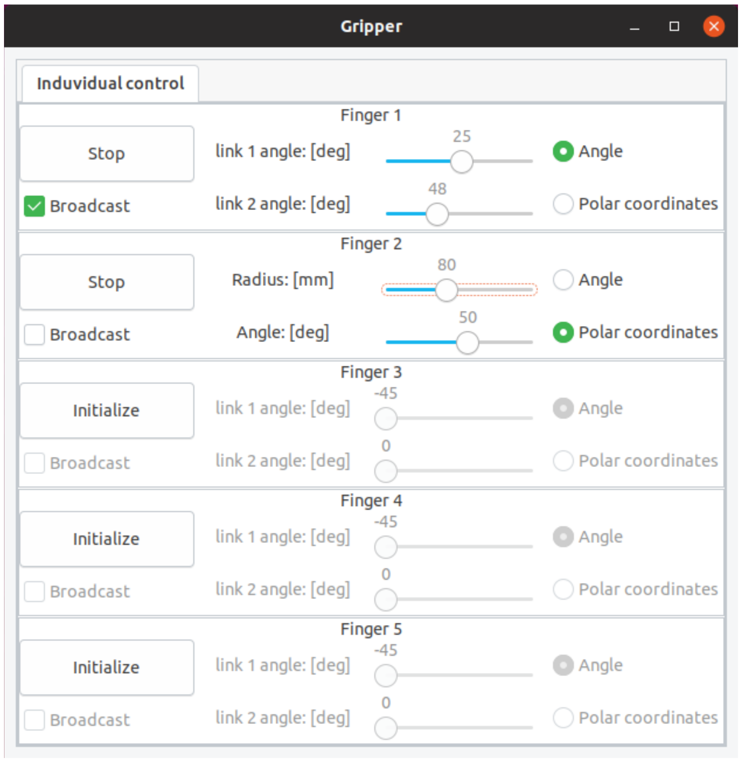
An application with graphical interface were made to test the master node. The GUI is displayed in figure 6.7. The application has 5 rows of identical fields for controlling 5 different fingers.

Pressing 'initialize" sends a `SimpleInstructionMsg`, telling the master to run controller 1. This is the calibration routine, where angles are set to the correct position. The scales on the right side can be used to send a new angle set point. Messages are only sent when the broadcast check box is active. Thus, messages can be sent continuously by using the scales, or one at the time by changing the scales first before activating broadcast. When "Angle" is selected, the messages with data from the scales is sent with controller 2 selected. This is the joint space controller.

The row for controlling finger 2 has "Polar coordinates" selected, instead of "Angle". This means that `controllerSelect` is set to 3, which is the controller with Cartesian coordinates as input. In fact, the application converts the polar coordinates from the scales to Cartesian coordinates before sending them. The reason for using polar coordinates in the GUI is that it is easier to stay within the ROM with polar coordinates given its shape.

The GUI was made using the GTK library. It is an object oriented library, where buttons and windows are "GtKWidget" objects that can be placed inside one another to create the layout. Each row for controlling a finger has 22

[8]pigpio library and documentation available at `http://abyz.me.uk/rpi/pigpio/`



Finger 1
[✓ Broadcast]
link 1 angle: [deg] 25
link 2 angle: [deg] 48
[✓ Angle] Polar coordinates

Finger 2
Broadcast
Radius: [mm] 80
Angle: [deg] 50
Angle [✓ Polar coordinates]

Finger 3
Broadcast
link 1 angle: [deg] -45
link 2 angle: [deg] 0
Angle Polar coordinates

Finger 4
Broadcast
link 1 angle: [deg] -45
link 2 angle: [deg] 0
Angle Polar coordinates

Finger 5
Broadcast
link 1 angle: [deg] -45
link 2 angle: [deg] 0
Angle Polar coordinates

Figure 6.7: GUI for sending commands to master

such widgets. The rows were created as a class to avoid repeating code, and additional rows can easily be added by creating new instances of the class. Still, the application consists of around 400 lines of code. The source code is available on Github [9].

[9]Source code for ØMQ publisher with GTK GUI: `https://github.com/Bardie4/Dexterous/tree/master/ZMQ_server`

Chapter 7

Results and discussion

Throughout this project multiple theoretical principles have been realized in physical designs. Efforts have been made to finalize a practical mechanical design, an electronic design allowing for streamlined control implementation and a stable control scheme for verification of system controllability. The final state of the system is presented in this chapter, as well as performance tests to gauge the capability of the system.

Creating a controllable system from only an idea may be a daunting task. As a famous astronomer once said:

> “If you wish to make an apple pie from scratch, you must first invent the universe.” - Carl Sagan

7.1 Mechanical

The final state of the produced and assembled mechanical system is a single finger. This includes two independently controllable joints, and custom-made gearbox.

7.1.1 Assembly

Building a complete finger with gearbox took roughly ten hours, but can be done in seven if one has put the system together once or twice before. As part of assembly, sanding down all the parts took two hours and thirty minutes. The main focus of the sanding process is smoothing out the bearing slots and the axles that goes into the bearings, basically any part that requires a tight fit. Extra tension from a large axle or to tight bearing slot would cause noticeable increase in friction in the bearing. The bearings needs to be tested once they are in place. Fine tuning the dimensions of these parts to fit the accuracy of the 3D printer should reduce the time needed to perform this task by at least one hour. Figure 7.1 shows a photo of all the 3D printed parts sanded down.

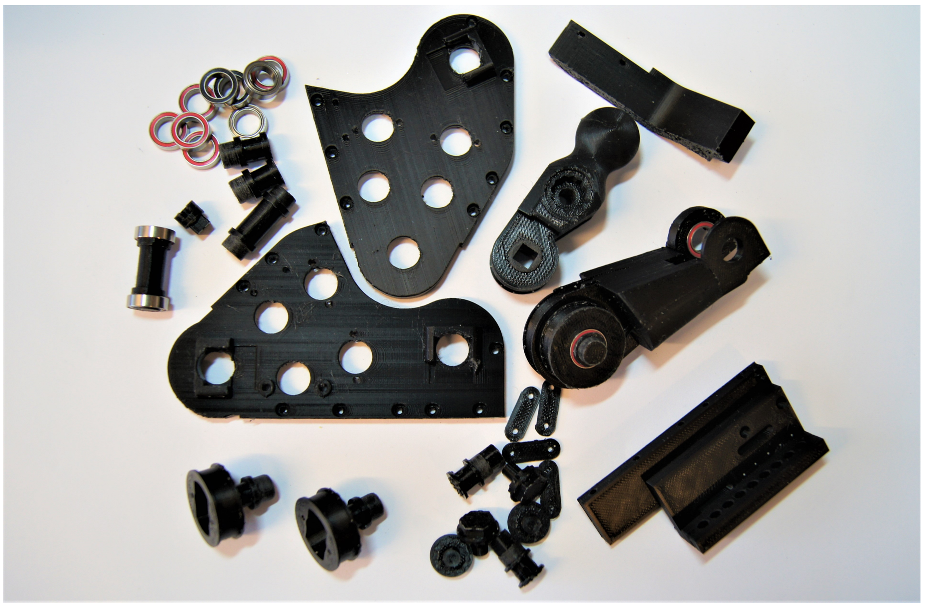


Figure 7.1: Dexterous finger, assembly set.

For the actual assembly, the main difficulty is working with the first gear stage. The string is thin, and the number of turns is high. A successful attempt at assembling the first gear stage took thirty minutes, which makes a total of one hour for both motors. However, the most important part is keeping it in place while the second stage is assembled. If the gear spindle pops out of place and unwinds the string, both stages has to be done all over. Tape can be used to keep it in place during assembly.

The second stage is relatively easy to work with. The number turns is low on the gear spindle. On the joint pulleys there is less than one turn, which makes it a lot easier. The tightening mechanism is practical and large.

The finished finger is seen in figure 7.2.

7.1.2 Tendon stability

Since the tendon configuration should be suited for non elastic tendon materials, the path taken by the tendon must always be of the same length to not cause slack, or get the system stuck. Chapter showed that considerable efforts were made into investigating the use of grooves to make a stable position for the tendons to rest. The idea was then discarded because of the large amount of effort it took to realize the concepts. Because of the principle of one path always being the shortest, it was assumed that the tendon would have a stable position anyway.

The result of not using grooves was mixed. The first gear stage with the

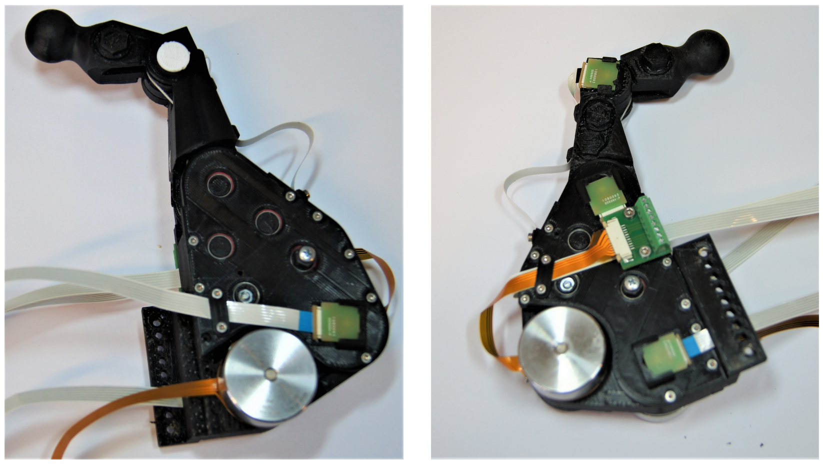


Figure 7.2: Dexterous finger, assembled.

most turns and thinnest tendon, was troublesome. The second stage, which could only physically fit 3 turns because of the string size, behaved nicely.

The shortest path for the tendon, is the path which cause the least tension. Thus, it is more inclined to move in this position. The challenge was therefore to get it in the perfect configuration, and tightening it before it moved. This turned out to be quite hard. The tendon would be stable, but not in the position of the shortest path. The tendon on the first stage was inclined to cross itself, which resulted in irregularities in tension, which was felt as irregular friction as the motor was turned.

Similar issues were not encountered with the second stage. The thickness of the string meant a significant amount of slack was needed in order for it to cross itself. The pitch of the spiral was higher. Tightening the second stage was also easier as it was done in the finger, and the process did not disturb the tendon configuration as much as it did in the first stage.

7.1.3 Elasticity

The data for the plot in figure 7.3 is gathered through setting the PIP joint to its end position and letting the motor continue to try and rotate it further in the direction of the physical stop. The plot shows how the angle of the motor moves, given an increasing voltage, even as the joint it is driving is standing still. This elasticity is found to be caused by the tendon that transfers torque between motor and joint not being perfectly stiff. The system's linearity highly relies on wires being fastened tightly. The tendon in the second stage of the gear transmission is believed to be the cause of the problem, as it is a somewhat flexible braided wire. Still, the flexibility experienced here is not seen as an inherent problem to the system as another wire material can be chosen for the tendon in the future.

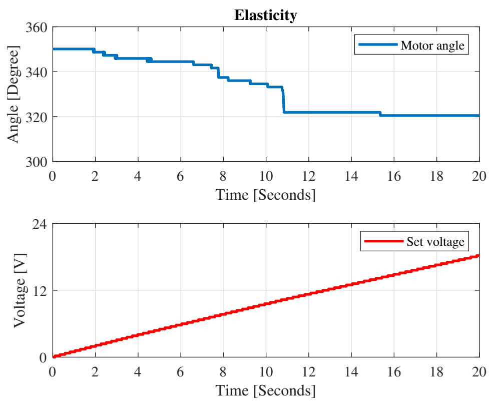


Figure 7.3: Elasticity vs. voltage.

7.1.4 Friction

From the plots that are presented in section 7.3.1, it is apparent that that angle control results in steady state errors caused by friction. It is likely that the tension in the system is too much for the bearings. The bearings are cheap, and marketed towards the radio controlled cars market. No data sheet was available from the supplier, and maximum load is unknown. The first gear stage is also known to be a source of friction. For the mechanical system, reducing friction is the best improvement that can be made at this point, because friction reduces precision.

7.1.5 Finger

Through several revisions the dexterous gripper has evolved into a sleek design with a range of movements that enable dexterous manipulations. Through the gearing of the motor adequate torques are achieved in the distal phalanx, a tendon system allow the motors to be mounted a distance away from the joints and for each of the joints to be driven in two directions. The second link has some friction issues, where it will make a creaking sound and become slow if the tendon tension is high enough. Higher quality bearings and a more solid joint would probably improve results.

The finger and gearbox design has been made with back-driveability in mind. Using the limited loop configuration, presented in 3.3, has allowed the finger joints to actuate both ways by the motors. It also allows for the motors to be rotated by rotation of the fingers. The principle was tested by forcibly knocking the finger back, documented on video.

7.1.6 Weight and size

The finger and gearbox are light as they are made out of PLA plastic. The motors mounted on the gearbox is what adds to the weight of the gearbox, but it is still light enough to be mounted on an industrial robot arm. Its durability also seems to prove adequate for gripping tasks.

When it comes to size, the finger, including gearbox, is comparable to the fingers on Right Hand Robotics' gripper, REFLEX. As can be drawn from figure 7.4, a three finger configuration would fit in the same volume as this gripper. Though, some smart adjustments would have to be done to also fit all the needed electronics inside.

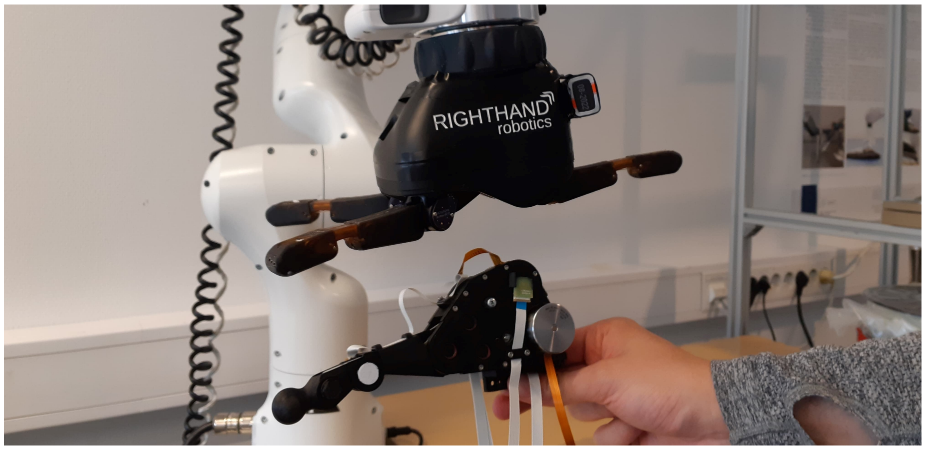


Figure 7.4: Right Hand Robotics' REFLEX and dexterous finger size comparison.

7.1.7 Overall result

The mechanical system is modular in the sense that a finger with gearbox is self contained. Several fingers can therefore be added to a gripper in arbitrary configurations. It is a little too time consuming to assemble, which drives up the price if paid technicians are tasked with assembling it. However, for hobbyists it is extremely cheap. Friction needs to be reduced, and tendon materials needs to be explored to reduce elasticity. The tendon configuration itself worked well for actuation of the finger in general, but grew complicated when applied as a gearing system. The configuration suggested in chapter 3.5.3, which replaced the first stage with a planetary gear is likely to help with the greatest shortcomings of the system, which is assembly time and friction. At the same time, planetary gear does introduce a backside, namely backlash, which is what the presented system aimed to eliminate. Lastly, bearings of higher quality would help the overall performance of the system.

7.2 Electronics

As an outcome of the work in creating custom electronics, multiple PCBs have been manufactured. The delivered electronics are: Kiyona control boards, in three different revisions; angle sensor boards, numbering 8 total; and a Raspberry Pi HAT, of which only one has been assembled.

The whole electronic system has been designed to enable easy use of the Dexterous gripper. Through using standardized cables and connectors, the electronics are easily setup, saving time in setups of experiments. Furthermore, this allows for a modular design, as an arbitrary number of connections can be made to the I^2C bus to control the same number of slave nodes. All that is needed to facilitate this is to give each Kiyona control board its own I^2C slave address in software. On the master side the limiting factor for the number of connected fingers is the Raspberry Pi HAT, it serves six SPI connections, enabling the access to the angle sensors of three fingers.

The electronics all performed as wanted. The Raspberry Pi HAT eased the use of the system and allowed for a master slave communication pattern, as well as angle readings. The angle sensor boards allowed for integration into the mechanical finger and gave accurate, though somewhat noisy, readings. Lastly, the Kiyona control board allowed for connections of angle sensor readings, driving motors and accepting commands from the master. A picture of the Kiyona control board, revision 2, can be seen in figure 7.5, this was used to produce the results in this chapter. Though the linear voltage regulators could be swapped out to allow for the higher inrush current needed to drive the motor from a standstill.

The third, and final, revision of the Kiyona control board, which adds current sensor measurements for motor control, remains untested. They were ordered, but time concerns led to this not being assembled nor tested. CAD files are included in the project's online repository[1]. Through the implementation of current sensing, field oriented control can be utilized in motor control, which is likely to give more accurate control.

The angle sensors are mounted on the finger, using mechanical sliders. The same concept could have been employed to also mount the Kiyona control board onto the physical gripper. To really create an accessible, and easy to use, platform for manipulation tasks this could be done in the future, but it was left out because of time concerns.

7.3 Control

As mentioned in section 4.3, there was little time left for high level control. There was instead a higher focus the controller manager, which provided modularity in the sense that fingers could be added at will, and controllers could be changed at run time. The controller tested here is simple and not very in-

[1]Project repository available at: `https://github.com/Bardie4/Dexterous`

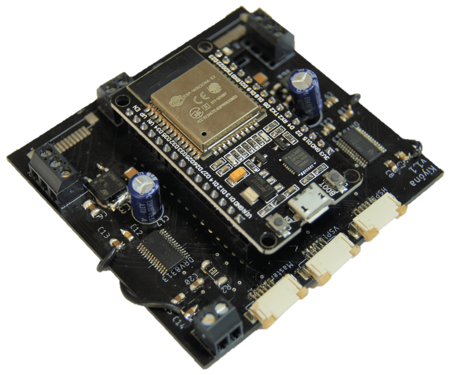


Figure 7.5: Kiyona control board v.1.1, with ESP32 microcontroller.

teresting in itself, but serves the purpose of gauging the performance of the mechanical system, and the capabilities of the system as a whole.

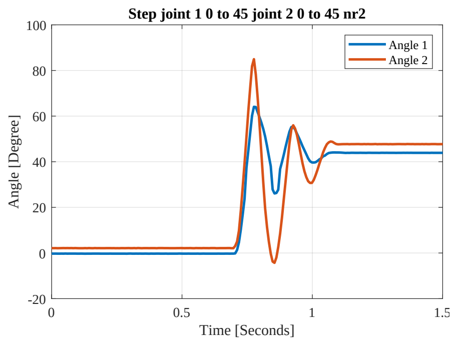
Through testing, it was found that a proportional controller yielded best results. Because of the lack of an estimator, the derivative term in the PID-controller was noisy and would reduce stability. Besides, the response was already very fast with a proportional controller, so the derivative gain was set to zero. The friction in the system meant that integral action resulted in "stick-slip" cycles as was expected in cases were friction was sufficiently high, covered by [Winsjansen, 2018]. Thus, integral action was not used either. All plots in this section is therefore done with a proportional controller.

7.3.1 System dynamics

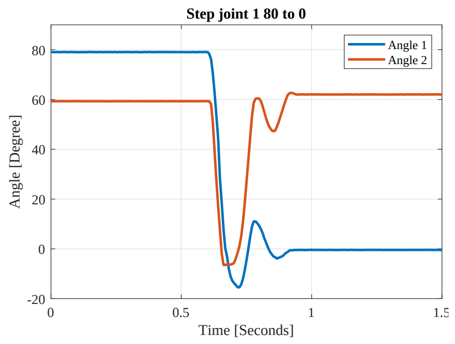
Different inputs to the system are used to get a sense of the system dynamics. A step response from changing the set point of the proximal phalanx, or link 1, is seen in figure 7.6, while another step response in the middle phalanx, or link 2, is seen in figure 7.7. Changing the setpoint of both links simultaneously yielded the result in figure 7.8 and 7.9. Common to all plots is the steady state error caused by friction, and the fact that there is no integral action. The response of the system is fast, and a stable state can be reached within between a quarter of and half a second.

Most notably in figures 7.6a and 7.6b, is how much movement of link 1 affects link 2. This phenomena was expected, as explained in section 4.3.2. The joint angle θ_2 is a function of θ_1 and the motor angle θ_{m2}. Since θ_2 is already in the wanted position, the motor only compensates when errors arise from movement in θ_1. Even though the θ_2 is in the right position from the start, the motor angle θ_{m2} is not. It is therefore likely that using θ_{m2} as the control objective would yield a better result.

The response from changing the angle set point of link 2 is seen in figure 7.7a and 7.7b. Here it is seen that movement of the second link barely affects

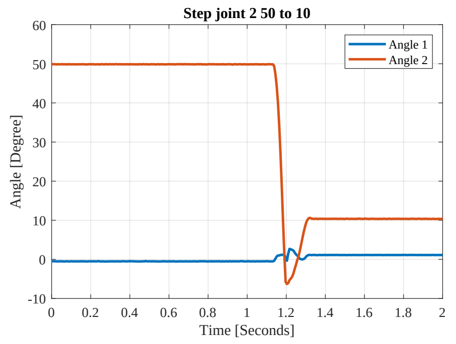


(a)

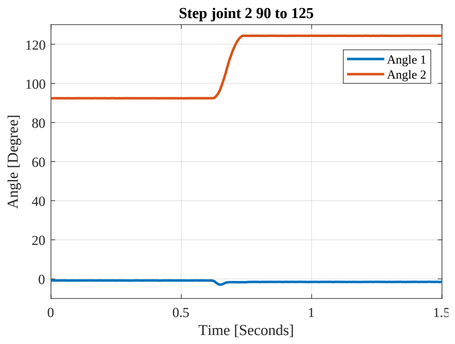


(b)

Figure 7.6: Step responses from changing angle set point of link 1.



(a)



(b)

Figure 7.7: Step responses from changing angle set point of link 2.

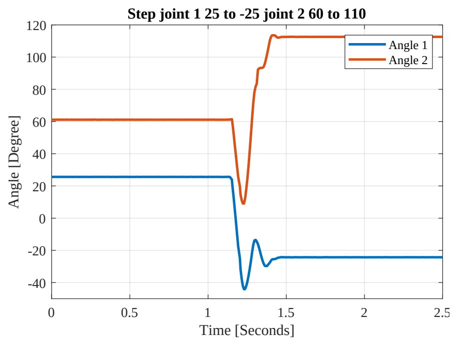
link 1, only giving it a minor error offset.

The step response from changing both angles at once and in the opposite direction is seen in figure 7.9b. Same directions is seen in figure 7.8a. The response time is less than half a second, and settles in approximately the same time as when only link 1 is moved. The reason for this is that both motors has to move regardless if only the set point of link 1 is changed.

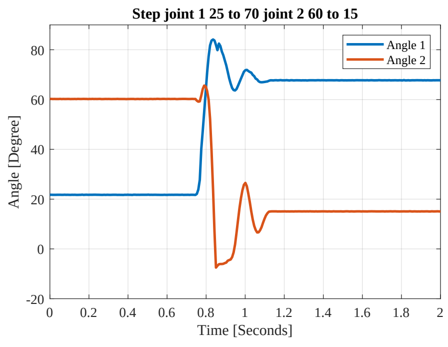
The specifications defines two fingers being able to be tapped together multiple times a second. The final system is capable of accommodating this, as the finger is able to move in one direction, reach stability, move in the opposite direction and again reach stability in under

Feed forward

The dynamics of the two joints that make up a finger are interlinked. The rotation of one joint causes movement in the other joint. This makes the precision of angle control inaccurate for each joint, when the other joint is moving. The

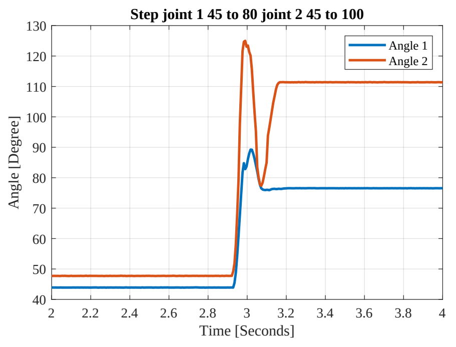


(a)

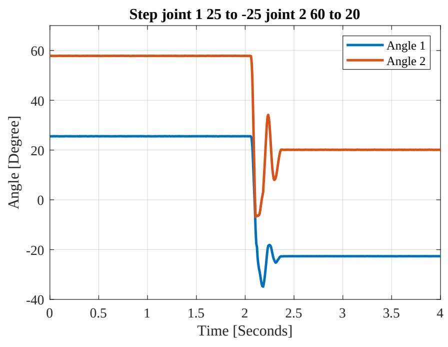


(b)

Figure 7.8: Step response from changing angle set point of both joints in opposite direction

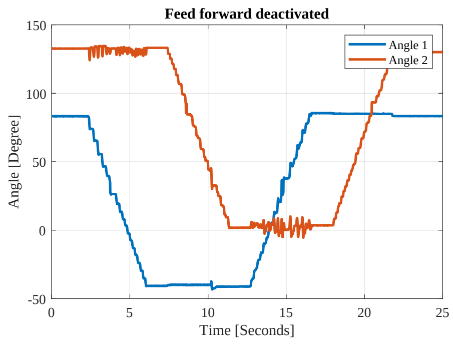


(a)

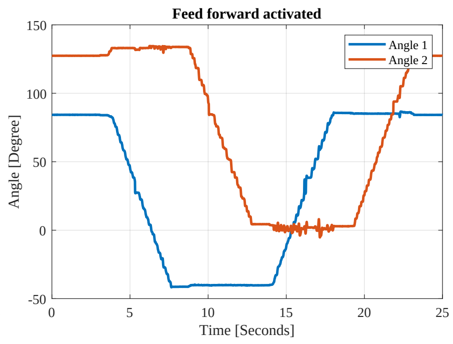


(b)

Figure 7.9: Step response from changing angle set point of both joints in the same direction.



(a)



(b)

Figure 7.10: No feed forward vs. feed forward.

dynamics were tested, and the results can be seen in figure 7.10. The accuracy in response of each joint to slopes in the other joint, was noticeably increased. Through further tuning of the feed forward implementation, it is believed that the response can be further improved.

7.3.2 Overall result

Further developments of controllers is needed.

7.4 Software

7.4.1 Slave node

Through tests, the response time of two simultaneously running motor control threads were found. This test included the simultaneous run of angle sensor readings, byte calculations and commutation of the motors, as detailed in section 6.2. The measured average was found at 2.5 milliseconds, for which the data can be seen in figure 7.11. The worst times were measured at 2.9 milliseconds. The average gives a frequency of 400 Hz under normal conditions.

7.4.2 Master node

The controller manager running on the master is designed to be modular, and can control up to 7 different fingers at the same time. Controllers can be changed during run time, which enables a gripper to perform complex tasks. A controller template was development to ease the process of adding new controllers to the system. Sensor information is broadcasted on the local network, and was used to make the plots in this chapter.

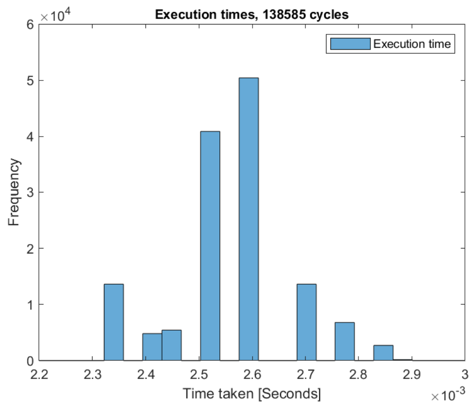
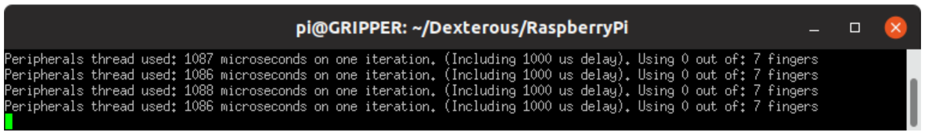


Figure 7.11: Slave node code execution time.

Performance

Figure 7.12 shows the iteration time of the `peripheralsController` when no fingers are active. No communication is done with the salve node or sensors when this is the case. The result is 1087µs, which includes a 1000µs sleep period. The sleeping period serves the purpose of leaving some resources open to the operating system, so that the iteration time is as stable as possible. It also prevents the master node from sending more information to the slave nodes than they can chew. It can be adjusted to much lower levels without any problems. 100*The* results shows that only 87µs is used by `peripheralsController` to perform one iteration. Thus, the master node is mostly sleeping in a state when no fingers are controlled.

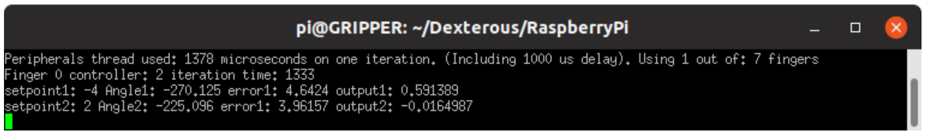


```
pi@GRIPPER: ~/Dexterous/RaspberryPi
Peripherals thread used: 1087 microseconds on one iteration. (Including 1000 us delay). Using 0 out of: 7 fingers
Peripherals thread used: 1086 microseconds on one iteration. (Including 1000 us delay). Using 0 out of: 7 fingers
Peripherals thread used: 1088 microseconds on one iteration. (Including 1000 us delay). Using 0 out of: 7 fingers
Peripherals thread used: 1086 microseconds on one iteration. (Including 1000 us delay). Using 0 out of: 7 fingers
```

Figure 7.12: `PeripheralsController` iteration time. No fingers active.

Figure 7.13, shows the the iteration time when one finger is active. The result is a 291µs increase in iteration time. Note that this is the time needed to communicate with sensors and slave node of one finger, not the time needed for controller calculations. The 3 byte message sent to the slave node using I^2C with a baud rate of 400kbit/s, should theoretically take around 100µs. Which leaves 190µs for SPI communication and broadcasting of sensor data over local network.

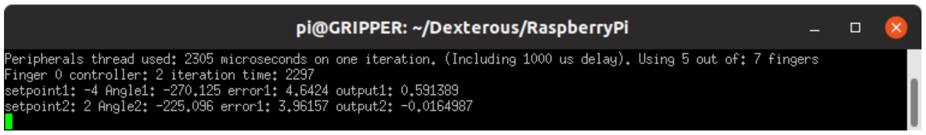
Figure 7.14 shows the iteration time when 5 fingers are active. With an elapsed time of 2305µs and an increase of 1218µs from the inactive state. This corresponds to 243.6µs used per finger, and is not much different from what was



```
pi@GRIPPER: ~/Dexterous/RaspberryPi
Peripherals thread used: 1378 microseconds on one iteration. (Including 1000 us delay). Using 1 out of: 7 fingers
Finger 0 controller: 2 iteration time: 1333
setpoint1: -4 Angle1: -270.125 error1: 4.6424 output1: 0.591389
setpoint2: 2 Angle2: -225.096 error1: 3.96157 output2: -0.0164987
```

Figure 7.13: `PeripheralsController` iteration time. One finger active.

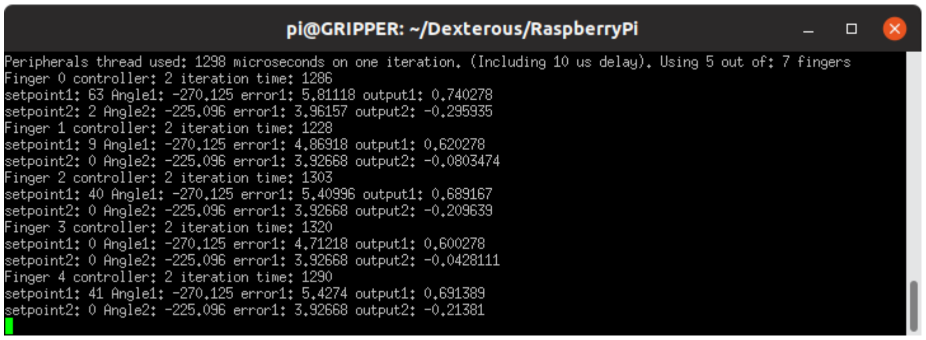
used in figure 7.13. Thus, the iteration time grows linearly with the number of active fingers. The results also show that the master node is able to communicate with sensors and slave nodes of at least 3 fingers in under one millisecond.



```
pi@GRIPPER: ~/Dexterous/RaspberryPi
Peripherals thread used: 2305 microseconds on one iteration. (Including 1000 us delay). Using 5 out of: 7 fingers
Finger 0 controller: 2 iteration time: 2297
setpoint1: -4 Angle1: -270.125 error1: 4.6424 output1: 0.591389
setpoint2: 2 Angle2: -225.096 error1: 3.96157 output2: -0.0164987
```

Figure 7.14: `PeripheralsController` iteration time. Five fingers active.

By reducing the sleeping period, the controllers get less dedicated time to perform calculations. However, since the application is multithreaded, they can work along side the `peripheralsController`. Figure 7.15 shows the result of reducing the sleeping period to 10µs with 5 fingers active. The `peripherals-Controller` used 1298µs on one iteration. The controllers only get 10µs dedicated time alone, but has 1288µs time to work along side the the `peripherals-Controller` thread. It is seen from figure 7.15 that they used approximately the same time as the `peripheralsController`. This is because they need a condition signal from the `peripheralsController` to finish one iteration. If they do not finish in time, they would have used twice the time since they would have to wait for the next condition signal to complete one whole iteration. This result shows that the communication with slave nodes, sensors and the process of broadcasting sensor data on the local network is the bottleneck of the system. More complex controllers should not cause any issue.



```
pi@GRIPPER: ~/Dexterous/RaspberryPi
Peripherals thread used: 1298 microseconds on one iteration. (Including 10 us delay). Using 5 out of: 7 fingers
Finger 0 controller: 2 iteration time: 1286
setpoint1: 63 Angle1: -270.125 error1: 5.81118 output1: 0.740278
setpoint2: 2 Angle2: -225.096 error1: 3.96157 output2: -0.295935
Finger 1 controller: 2 iteration time: 1228
setpoint1: 9 Angle1: -270.125 error1: 4.86918 output1: 0.620278
setpoint2: 0 Angle2: -225.096 error1: 3.92668 output2: -0.0803474
Finger 2 controller: 2 iteration time: 1303
setpoint1: 40 Angle1: -270.125 error1: 5.40996 output1: 0.689167
setpoint2: 0 Angle2: -225.096 error1: 3.92668 output2: -0.209639
Finger 3 controller: 2 iteration time: 1320
setpoint1: 0 Angle1: -270.125 error1: 4.71218 output1: 0.600278
setpoint2: 0 Angle2: -225.096 error1: 3.92668 output2: -0.0428111
Finger 4 controller: 2 iteration time: 1290
setpoint1: 41 Angle1: -270.125 error1: 5.4274 output1: 0.691389
setpoint2: 0 Angle2: -225.096 error1: 3.92668 output2: -0.21381
```

Figure 7.15: `PeripheralsController` iteration time. Five fingers active.

Description	Qty/Amt.	Total
M2 10 mm bolt	23	10.00
M3 6 mm bolt	6	10.00
M3 30 mm bolt	2	10.00
M3 hex nut	2	10.00
Bearing 8x12x3.5 mm	16	239.00
Microfilament braided line	2 m	40.00
Yachting rope	2 m	40.00
Plastic	200 g	50.00
	Total	409.00

Table 7.1: Total assembly bill of materials.

API

Pre-made message types in the form of flatbuffers schemas makes up a standardized method of interfacing with the master node, promoting ease of use. In addition to this, the GUI application, covered in 6.6, enables quick testing, and the code used in the application can be used as a guide.

7.5 Cost-effectiveness

In ensuring that the dexterous gripper comes at a low cost, emphasis has been put on using low cost parts, when compared to other commercially available grippers. This section presents the cost of components and manufacture. This is all disregarding the work that is needed to solder the boards, assemble the gearbox and finger, and set everything up in a configuration suitable for an experiment.

The Kiyona control board makes up the brain of the low level control, and also houses the costliest electronics. One board was assembled with the components detailed in table 7.2 (all prices in NOK). The total price of the board was NOK 270.20. Adding the price of manufacture of ten boards (the minimal amount allowed with manufacturer SEEED), NOK 44.3, and the price of a stencil, NOK 78.50, this adds up to NOK 393.

Creating the physical finger requires some financial investment. Table 7.1 shows a rough estimate of the money spent to create one finger. For a finger and needed electronics this adds up to

One angle sensor board added up to NOK 65.99 in component costs, detailed in table 7.3. Ten boards come at a cost of NOK 42.90, bringing the total up to NOK 108.89.

Each Raspberry Pi HAT is made up of the parts in table 7.4, totaling NOK 57.25. With the price of manufacture, of NOK 42.90, the price comes up to NOK 100.15.

The total cost of the electronics and boards for control of one finger is NOK 2822.22.

Description	**Package**	**Qty**	**Price**	**Total**
DRV8313 motor driver	HTSSOP-28	2	25.57	51.14
FFC connector	6 pos 1 mm pitch	2	6.73	13.46
FFC cable	5” 6 pos 1 mm pitch	2	13.59	27.18
Linear volt. reg 24V 500mA	DPAK	2	10.50	21.00
Linear volt. reg 9V 500mA	DPAK	1	5.25	5.25
Terminal block	3 pos 0.1” pitch	3	4.19	12.57
Switch	2 pos DIP	1	7.74	7.74
Female header	15 pos 0.1” pitch	2	16.86	33.72
INA240A4 current sensor	8SOIC	4	23.82	23.82
Max6520 1.2V reference	SOT23-3	1	28.12	28.12
100µF capacitor	Radial electrolytic	2	1.36	2.72
0.1µF capacitor	0805	7	0.37	2.59
0.01µF capacitor	0805	2	0.37	0.74
0.47µF capacitor	0805	2	0.77	1.54
0.1µF capacitor	1210	2	1.73	3.46
0.33µF capacitor	1210	3	5.35	16.05
10kΩ resistor	2010	2	1.59	3.18
0.01Ω resistor	1206	4	3.98	15.92
			Total	270.20

Table 7.2: Kiyona bill of materials.

Description	**Package**	**Qty**	**Price**	**Total**
MA302 angle sensor	QFN-16	1	65.62	65.62
0.1µF capacitor	0805	1	0.37	0.37
			Total	65.99

Table 7.3: Angle sensor board bill of materials.

Description	**Package**	**Qty**	**Price**	**Total**
FFC connector	6 pos 1 mm pitch	6	6.73	40.38
Female header	15 pos 0.1” pitch	1	16.86	16.86
			Total	57.24

Table 7.4: Raspberry Pi HAT bill of materials.

Description	**Qty**	**Price**	**Total**
Maxon EC 32 Flat BLDC motor	2	704.45	1408.90
11 pole flexprint motor connector	2	139.91	279.82
ESP 32 microcontroller	1	62.90	62.90
Raspberry Pi 3 Model B	1	339.00	339.00
8" FFC cable	2	28.90	57.80
12" FFC cable	2	41.20	82.40
Kiyona board w/components	1	270.20	270.20
Raspberry HAT w/components	1	57.24	57.24
Angle sensor board w/components	4	65.99	263.96
		Total	2822.22

Table 7.5: Total electronics bill of materials.

"Now its up to the researchers at SINTEF to create the apple pie."
- Ruben Winsjansen

Chapter 8

Conclusion

The “limited loop” tendon configuration proved to be successful, and made the fingers back-driveable. It also allowed a lightweight and sleek finger design, in addition to multi-direction actuation with each motor. As a gearing mechanism the configuration grew complicated, which increased assembly time. A high reliability 3D printer was useful to reduce the overall cost of the system. Assembly of the system is too difficult and time consuming, though changing the first gear stage into a planetary gear would probably reduce both assembly time and friction. Bearings of higher quality are needed to increase precision, as the friction in the current version is too high. A less elastic tendon material is needed to reduce elasticity.

The electronic system, consisting of several custom printed circuit boards, performed to specification. It provides a means of communication, and an interface for control of the broader system. The system architecture supported the project’s specification for time concerns. Moreover, through sinusoidal commutation motor control has been implemented successfully, which contributes to a responsive system. Though not tested practically, current sensor based, motor control was designed. This is likely to improve the motor control further once implemented in software.

The master node software was able to run controllers at 1000Hz, on up to three separate fingers at the same time. The high speed calculations and communication of the Raspberry Pi, together with the modular nature of the software enables a total of 7 fingers to be controlled simultaneously. Development of a template controller has made integration of custom controllers user friendly.

For high level control, two PID controllers were made. They were used to verify that the system as a whole, with everything from sensor readings to network communication, was operating successfully together. For further development of the system, an estimator is needed to increase the quality of the velocity measurements. The system is also in need of additional and more advanced controllers before being set to use in gripping tasks.

Bibliography

[Amin and Rehmani, 2015] Amin, M. and Rehmani, M. (2015). Operation, construction, and functionality of direct current machines. *Operation, Construction, and Functionality of Direct Current Machines*, page 190.

[Aursand, 2017] Aursand, M. (2017). iProcess newsletter June 2017. `http://iprocessproject.com/wp-content/uploads/2017/09/iProcess-Newsletter-June-2017-2.pdf`.

[Coulouris, 2012] Coulouris, G. (2012). *Distributed Systems Concepts and Design.* Pearson.

[Danielsen, 2018] Danielsen, A. (2018). Prosjektoppgave: Konstruksjon av finger til avansert robothånd.

[Höfler, 2018] Höfler, A. (2018). Belt drive: Basics. `https://www.tec-science.com/category/mechanical-power-transmission/belt-drive/`. Accessed: 2019-01-23.

[IQBAL et al., 2017] IQBAL, J., KHAN, Z. H., and KHALID, A. (2017). Prospects of robotics in food industry. *Food Science and Technology*, 37:159 – 165.

[Lee et al., 2009] Lee, S., Lemley, T., and Keohane, G. (2009). A comparison study of the commutation methods for the three-phase permanent magnet brushless dc motor. In *Electrical Manufacturing Technical Conference 2009: Electrical Manufacturing and Coil Winding Expo*, pages 49–55.

[Li and Adelson, 2013] Li, R. and Adelson, E. H. (2013). Sensing and recognizing surface textures using a GelSight sensor. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1241–1247.

[Li et al., 2014] Li, R., Platt, R., Yuan, W., ten Pas, A., Roscup, N., Srinivasan, M. A., and Adelson, E. (2014). Localization and manipulation of small parts using GelSight tactile sensing. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3988–3993.

[Okamura et al., 2000] Okamura, A. M., Smaby, N., and Cutkosky, M. R. (2000). An overview of dexterous manipulation. In *Proceedings 2000 ICRA.*

Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065), volume 1, pages 255–262 vol.1.

[Olfa et al., 2016] Olfa, D., Sidhom, L., Chihi, I., and Abdelkrim, A. (2016). On the numerical differentiation problem of noisy signal.

[Sustrik and Lucina, 2017] Sustrik, M. and Lucina, M. (2017). ØMQ API. `http://api.zeromq.org/2-1:zmq`. Accessed: 2019-02-25.

[Uyguroğlu and Demirel, 2006] Uyguroğlu, M. and Demirel, H. (2006). Kinematic analysis of tendon-driven robotic mechanisms using oriented graphs. *Acta Mechanica*, 182:265–277.

[Winsjansen, 2018] Winsjansen, R. (2018). Pre-project report: Low-level control of actuator for robotic finger.

Fornavn Etternavn

NTNU
Norwegian University of
Science and Technology