Creating a bionic hand

- as a way of exploring physical programming and inter-connectivity

Alexander Kiellerup Swystun, Áron Kuna, Azita Sofie Tadayoni, Emma Kathrine Derby Hansen

A Subject Module Project in Computer Science

Supervisor: Bo Holst-Christensen
Roskilde University Center
Monday 30th May, 2022

Abstract

This report is made to introduce, document and discuss our process of creating a bionic hand that is remotely controlled by a hand tracker. This project is meant to challenge us with various programming related obstacles, and to further our own personal ambitions in regards to our study of computer science. Therefore, this report will go over our acquired knowledge, our process and the different components of the project. We use the iterative model to structure our acquired knowledge, and use it to discuss the success of the project.

Table of Contents

1 Introduction

In this subject module project in computer science, we create a bionic hand which responds to input from a webcam. The different elements of this project will include: Building of the hand, Arduino elements to control the physical aspect and the cross-connectivity to a hand-tracker coded in python. The different elements are based on topics we would like to learn during this semester. Mainly focused on the physical computing along with learning the programming languages C and Python.
This report will mainly detail our final build, along with some description of the process over the semester. We aim for our report to be used as a guide to anyone interested in constructing a bionic hand, or at least take away our knowledge to be used in other kinds of physical computing projects. To aid with this, we will go over the different components of the project, outline some of the problems and oversights we ran into, along with things we would have liked to have been aware of prior to starting.

2 Theory

2.1 Hand-tracker

Tracking a hand means identifying coordinates of different parts of a hand, like fingers, the tip of fingers, the wrist, etc. One way to do this is to use gloves with sensors on them, however, this makes it tedious to change users, especially with different hand sizes. It is possible to make hand-tracking more feasible using a video feed.[13]
A video feed hand tracker uses images recorded by a camera. The program can identify and locate parts of the hand giving out x, y coordinates, and in some cases also z coordinates in near real-time. When displaying the video x is the horizontal coordinate, y is the vertical coordinate and z represents the distance from camera.
To describe the position of a hand and fingers, only a few parts of the hand have to be located. These points are called landmarks. In a video-based hand tracker, the position of these landmarks is given with coordinates x and y ranging from 0 to 1. Where $(0, 0)$ is the top left corner. However, when the image is flipped to give a self portrait view it becomes in the top-right corner. The opposite side of the video has the coordinate $(1, 1)$. Each landmark has an identifier that allows the practical use of the hand tracker.
To turn the pixel information provided by the camera into coordinates, a trained machine learning model is used. The model is a convolutional neural network that, when trained, can provide quick predictions.[13] However, to train a convolutional neural network, a huge amount of data is required. This data is collected partly by taking photos of hands and carefully noting the landmarks, partly by rendering high-quality synthetic hand models.[21] The training requires a long time and high processing power. The trained model can be used on an average laptop to process the video.
There are several trained hand-tracker models available to use for free. One commonly used, is the Mediapipe hand tracker by Google. It is available to use in a variety of programming languages, namely Python, JavaScript, and Android.[2]

2.2 The Cv2 library

OpenCV is short for Open Source Computer Vision Library. It is a library with more than 2500 algorithms used for open source computer vision and machine learning softwares. It was invented by Gary Brodsky in 1999 and was released a year after, in 2000[19] The algorithms that OpenCV library contains has interfaces for several programming languages, Python, Java, MATLAB and C++, where this project is making use of OpenCV-Python. CV2 is the module import name for OpenCV-Python. OpenCV-Python is used for programs written in Python and is a wrapper for an original implementation of OpenCV in C++ language[19]

2.3 Threading

In our project we make use of threading, which is common practice when doing Python programming. In Python, a thread is an execution of a module in a program, separate from other modules in the program. This means that the program will have two or more things happening at once, depending on the number of threads used. This concept of multi-threading, where multiple threads work together to achieve a common goal can be hugely beneficial to a program[17]. For our project, this is crucial for the functionality of the program, as we need the hand tracking module to run and at the same time need the program to send our UDP packets containing the tracking data. However, in most cases the different threads are not actually running separately, as that would require a separate CPU for each thread to function. Instead, when running it all on one CPU, the way to achieve the effect of multi-threading, the CPU will go through and process a piece of one thread, and then switch to another. By switching between the threads, and processing part of the code in-between the switches, it will work as if the two threads are running at the exact same time.

2.4 Electrical circuits

To make a working prototype for our project we need to make use of different electronic components, and to make them work with each-other we need to understand what an electric circuit is and how to create one. In simple terms, a circuit is a never-ending looped pathway for charge carriers. Creating a circuit means making a pathway for electric charges to flow, and consequently power different mechanisms[14]. An electric circuit includes a power source that gives energy to the charged particles constituting the current (such as a battery or a generator), one or more mechanisms that are powered by the current (such as lamps, electric motors, or computers), and the connecting wires or transmission lines that runs in-between the power source and the mechanisms[10].
An electric circuit also needs a ground connection, either a true ground connection that is actually connected to the earth, or a what is called a floating ground connection. A floating ground connection is a type of ground in which the ground doesn't have a physical connection to the earth, it simply serves as a type of 0V reference line that serves as a return path for current to the negative side of the power supply[4]. The reason why a ground connection is needed in an electric circuit, is in simple terms to add another layer of protection to a circuit. The way the ground connection works, is excess electricity will take the path of least resistance, which will be the grounding connection. When electricity flows through a circuit, it may build up to dangerous levels, which can cause short-circuiting, electrical fires, or shocking. This is why the ground is needed, to guide any excess electrical current to a safe destination through the ground connection.
When creating a circuit, there can be multiple power sources, as different elements in the circuit could require different voltages. In such a situation, the ground connection should still be the same, and connected to every element in the circuit[1].

2.5 ESP32 micro-controller

The ESP32 is a series of microcontrollers, which is a small computer consisting of one or more CPUs, generally used for processing and sending commands in electronic devices. This ranges from TV remotes to power tools and automobiles. The ESP32 is specifically used in the same vein as Arduino microcontrollers, but it also supports both WI-FI and Bluetooth connections. In its conception it was essentially used to "upgrade" some of the Arduino microcontrollers that didn't have these features. The ESP32 would be mounted on top of the Arduino itself, but since it essentially was a microcontroller, it started to be used alone. It is also sold cheaper than any Arduino controllers with these features, for example, an Arduino Uno with added WI-FI connectablitity costs roughly double that of a standard ESP32[18].

Being based off the Arduino microcontrollers, the ESP32 also runs on C, and is primarily coded in the Arduino IDE, the biggest overall changes are the need for some extra libraries, for being able to properly make use of specific components, like servo motors. This is overall a minor change, with some small adjustments in the code, but could easily be overlooked.

2.6 Connection protocols

To send and receive information across computer applications, the information must be packaged in a way that both platforms can read and write messages. There are several connection protocols available to choose from, each coming with advantages and disadvantages. Important factors are speed, reliability, security and cross-platform availability. When using the internet for transferring messages, an easily accessible solution is to use UDP.[8]
UDP or User Datagram Protocol is a low-latency, lightweight connection protocol. It works on top of IP (Internet Protocol) and is used for loss-tolerating applications.
When sending UDP messages from one computer application to another one, the communication only goes in one direction. It has no handshaking dialogues, meaning the receiver does not assure the sender

that the message arrived. There is no ordering or protection against duplicated messages. However, for each packet checksums are provided which show if a message is corrupted.
The main advantage of UDP is its speed. Since the communication works only in one direction, the sender can consistently send packets without any interruptions to multiple receivers. It is used for time-sensitive applications that can handle some packet loss, where quick information transfer is more important than precision. If an information packet is late, it gets outdated and has no importance. [3] Examples of usages are audio and video broadcasts, also commonly used for IoT applications, like controlling a drone.

2.7 The iterative model

The iterative model is a model used to implement and outline steps of implementation in a software development life cycle. It is also a model that can be useful when wanting to document and explain every step and version of a specific software. It is thought of as a cyclical process which is repeated over and over, until the result is accepted by the developer, and even to be iterated again if the developer sees it fit to develop and test the software further after time. The steps are as following:

1. Planning & Requirements
2. Analysis & Design
3. Implementation
4. Testing
5. Evaluation

The Planning and Requirements phase is where you do the planning as a base for the rest of the cycle and find out which software/hardware requirements you are working with in your software development process. Once the planning and requirements are in order the analysis and design phase begins, where the design has to correspond to the analysis made, which can be made using e.g. database models. When the design is in place, in order to fit the requirements, the implementation phase begins, which is the coding process. This is where the initial iteration is in place which will be tested and evaluated in the two last steps. Testing is used to find out if there are any bugs in the program and how it runs, while the evaluation part is where the developers themselves, or others, examine and get ideas on where the project is and if anything should be changed, which leads us to a potential second iteration from step 1 to 5 [16].

3 Analysis

We have chosen to explain and structure our analysis of the process with help from the iterative model explained in chapter 2.7. We will use the iterative model to get to terms with the steps we've been taking with the theory we use, to create this project.

3.1 Planning & Requirements

The requirements for this computer science project is to *"develop a program with a complexity that raises interesting computational issues"* [15] and document the program in a report.

At the beginning of the project we set up our goals as the following:

What do we want to gain from the project?

- Better understanding of Python
- Better understanding of Arduino
  - The technical parts
  - Using it as part of a network
- Learn more C/C++
- Practice in developing interactive design

The focus of the project is to build a bionic hand controlled by a remote hand tracker and use the development of our bionic hand as a way to improve our knowledge in the selected topics. The optimal functionality of the hand is secondary to our learning goals.
The project is a success when:

- Every member of the group are able to improve the project further individually.
- We have built a bionic hand, controlled remotely by a hand tracker.
- If the report documents the development process well enough for an outsider to understand, and potentially function as a guide to recreate it.

What do we want the to product to do:

- We want each finger of the hand to be able to move individually or together
- We want the hand to be connected to the internet and, to be able to control it remotely.
- We want to have a hand tracker that can give an output, based on what it tracks.
- We want to connect a python program with the micro-controller remotely.

3.2 Analysis & Design

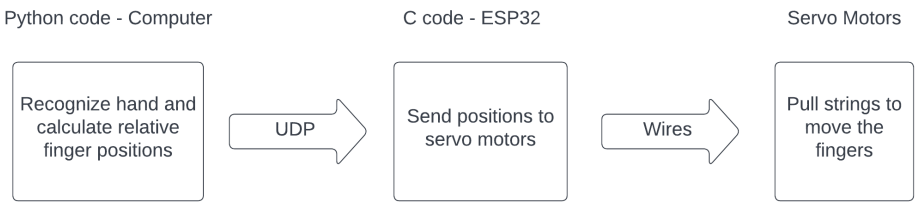

Figure 1: The components of the design and information flow between them.

The design part of the project consists of three main parts. First is the python code on a laptop, the second is the ESP32 and the C code running on it and the third part is the 3D printed bionic hand capable of human-like hand movements. The communication between the laptop and the ESP32 is wireless, using UDP packets through a wifi network. Then the ESP32 drives five servo motors which make the fingers move on the bionic hand.

Design Reasoning

This paragraph will provide reasoning for the choices we've made through the process of constructing and designing the hand and its necessary parts. As a baseline, most of the things we've chosen have been introduced to us in some way or another by our university teachers, which gave us some familiarity. To start, we've mainly chosen to utilize Python to control the MediaPipe hand tracker and to send our UDP packets. We all had an interest in learning the language itself through our semester, as it is widely used and focuses on readability for sharing our actual code. We chose to use UDP to send our information packets, as our goal is to ensure that our hand responds in real time to our hand tracker. The servomotors usually are in quite constant motion so speed was prioritised, and losing a single packet does not mean much when it constantly updates the input values. MediaPipe was our initial choice for a hand tracker, as it provides a stable framework that's easily edited to suit any needs we had, and the values are easily accessible.

Next, we decided to use the Arduino IDE to program our ESP32 that controls the micro servo motors. While other developer environments can be used, the Arduino IDE is specifically designed to upload code quick and easy to these micro-controllers, the ESP32 in our case. This model was chosen on behalf of its WI-FI connectivity, which allows it to connect to a local IP along with the sender, allowing for quick response to all 5 of our servo motors.
All of our micro servo motors are powered by a single 9-volt battery, which ensures that we are not drawing too much power through the micro-controller. These motors have a 180-degree rotation, and this is very well suited for our hand, as the hand goes from fully open to fully closed, which the motors mirror perfectly, along with making it easier to reset the motors rotation and values.

3D Model

Our model is a 3D printed hand from the website: thingyverse.com. Thingyverse is a website that offers pre-made 3D models suitable for 3D printing and 3D print inspirations available for downloading. We chose to use *"Bionic Hand with 5 individual fingers by velevdev"*[20] because it is a well-designed construction which perfectly fulfils our needs. By downloading the files of the different pieces of the hand on our computer we can start our 3D printing process. We went to FabLab RUC as it was very convient to use their available 3D printers and put our files from Thingyverse into a program called

PrusaSlicer. The model was then printed using PLA plastic filament. The construction of the hand itself is composed of the different 3D printed pieces, each corresponding to a different joint in the hand. Fishing wire is funneled through the front of the fingers, which is connected to the servo motors for contracting the fingers. Behind is elastic cord to maintain the upright position of the fingers when the motors are not engaged. Each motor requires a connection of power (5 volts), ground and one for rotation input, which has been secured with electrical tape, as the connectors can sometimes become loose.

One of our goals was to try to make the build itself as easily portable and accessible as possible, which is why we chose to power the electronics from batteries, eliminating the need to have it constantly wired to a computer or outlet.

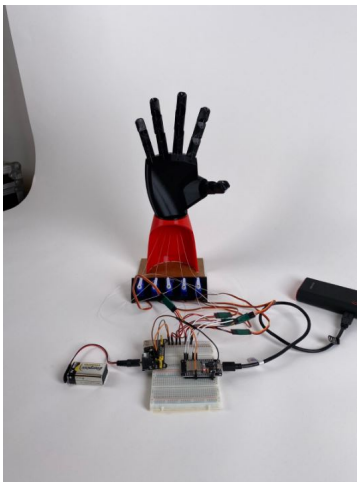

Figure 2: Full set-up, Handtracker & UDP sender not included (6th of May)

In Figure 2 is the full working setup of the hand as it were during the start of May, using a 9-volt battery to power the 5 servo motors, and a power bank to power the ESP itself, the code has already been loaded onto the ESP prior to hooking it up to a power bank.

User interaction program

The python code provides the tools to track the user's hand and send information to the ESP32 about the position of the fingers. It runs on a laptop and uses the camera of the laptop for video input. It is divided into three main parts. The hand-tracker class, the distance calculations, and the UDP package sending. To make bug-fixing and readability easier these parts are separated into different files and invoked in the file called 'main.py'.

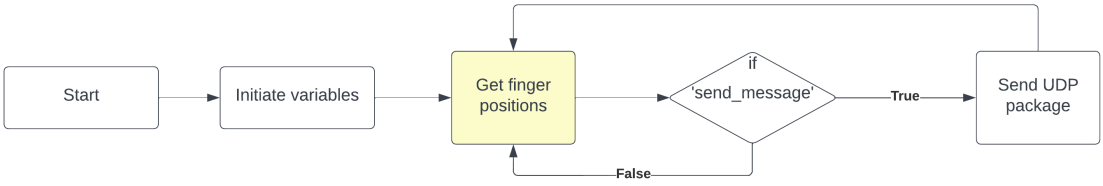

Figure 3: The flow of the python code. The Python code uses the laptop's camera and a pre-trained neural network to track the hand movement. Then relative finger positions are calculated to be sent to the ESP32.

The hand-track class

To track the movement of a hand we need a pre-trained neural network. Creating and training our own hand-tracker would be a tedious and time-consuming procedure. However, there are free trained models available on the internet. For the project, we need a hand-tracker that can track joints, usable with python, and runs on a simple laptop.
The hand-tracker provided by Mediapipe (Google) [2] fulfills all these major needs. Additionally, it has a clear user guide and code examples so that we can implement it in our code.
We have two main goals when implementing Mediapipe's hand-tracker. The first is to make it simply usable, without code repetition. Secondly, it has to run continuously while other parts of the python code transmit the UDP packets to the bionic hand. To achieve these goals our design decision was to create a separate class for the hand-tracker. Then the class variables can be updated separately without interfering with the rest of the program. This part of the project was inspired by a previous project written by Aron Kuna et al. [9]
The hand-tracker class starts the laptop's camera, using the cv2 library. Then using mediapipe's library and pre-trained model assigns the location of the needed joints to class variables. These variables are used in the distance calculation part to find the relative locations of the joints.
The update of the class variables happens under the function called 'loop'. This function then set to run continuously using a library called 'threading'. It means when a hand-tracker object is created using the hand-tracker class, the update of the variables are ongoing and independent of the rest of the program. The flow of the class is shown in Figure 5.
The class is overall designed in a reusable way. Making it ready to use in other applications with a few changes implemented.

OpenCV

OpenCV is a key library for the hand-tracker. It is commonly used for images and videos, to perform tasks like image recognition, tracking action on a video, generating 3D point clouds and much more. We have mostly used the OpenCV library to capture video and display it in a window on our computer screen, with video from the default webcam on our computer. We use the displayed video throughout the whole project to interact with the tracker from MediaPipe. We have for example, used it to display the hand tracker landmarks (Figure 4) and display error text on the video, whenever the hand tracker is not being able to be read properly. It is also used to begin and close the video display on the screen when we start the program and when we want to end it. OpenCV is therefore a very important library in our project. It helps us communicate our physical hands and fingers positions through the hand tracker, and on to the rest of the system.

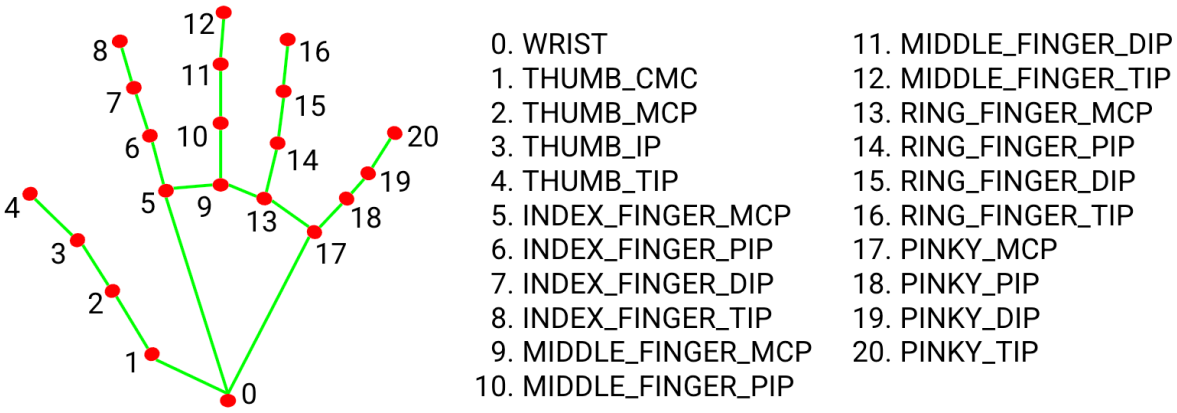

Figure 4: Landmark names for the Mediapipe hand-tracker. The landmarks are used in the program to identify joints and return their location values.

Source: Google *Mediapipe* [2]

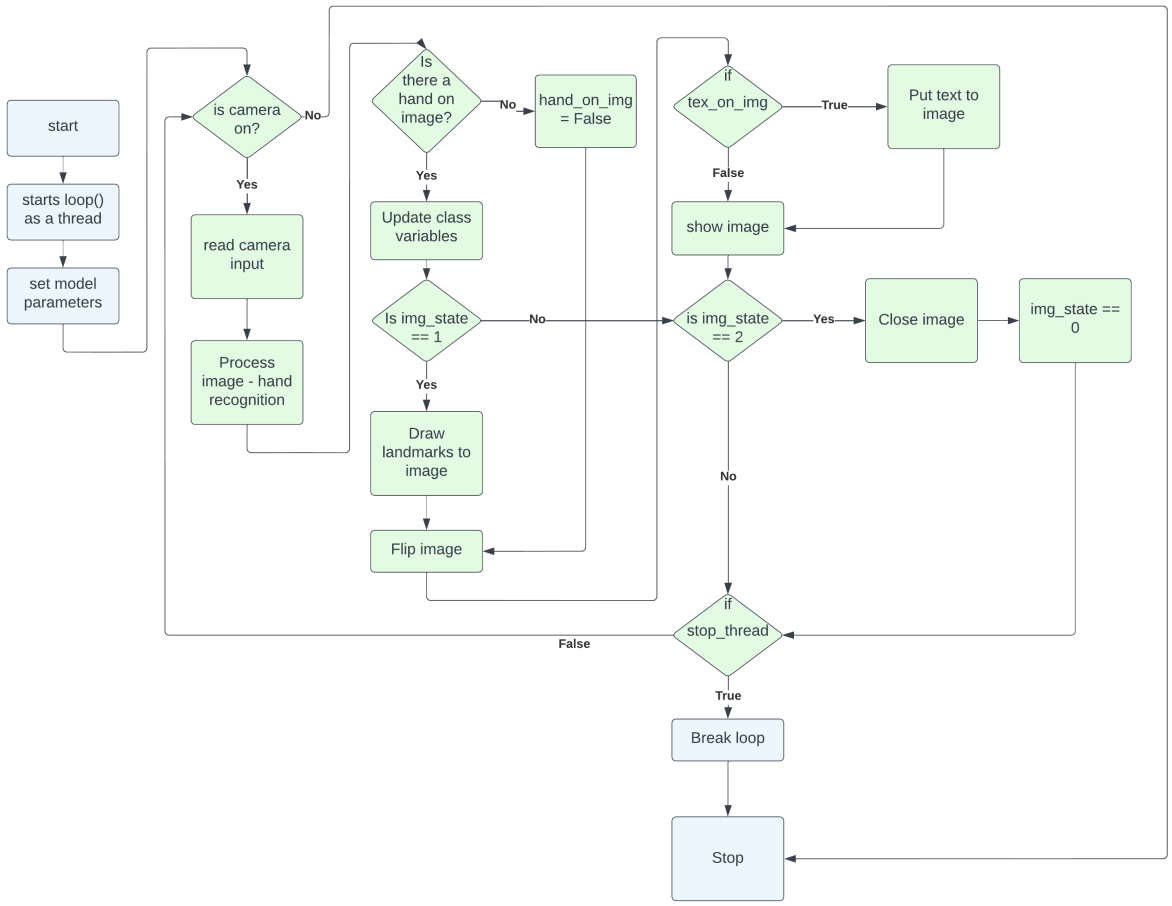

Figure 5: The flow of the main loop of the hand-track class. After a hand-tracker object is created this loop is started with the 'start' class function. This loop is constantly running in the background using the 'threading' library of python.

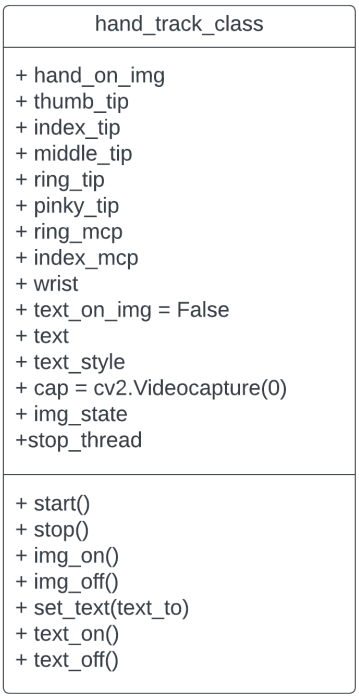

Figure 6: UML diagram of the hand-tracker class.

3.3 Fine tuning

Distance calculations

Even though the hand-tracker provided by Mediapipe provides three coordinates we decided not to use the z axis. It represents the distance from the camera, however based on our testing it is inconsistent. To make the product usable in different environments we decided to work only with the x and y axis as those are consistent.
The 'finger_positions.py' file includes all calculations to get the relative finger positions. Meaning when a finger is closed the assigned value is 0, when open it is 180, and in-between values are spaced out evenly. It also contains a function to check if the hand is in correct position or not.

To calculate the relative position of the fingers first a comparison point is needed. For most of the fingers, this comparison point is the wrist. Here we measure the distance between the wrist and the fingertips changes evenly as one closes the fingers. The only exception is the thumb as it closes in a different direction. For the thumb, the best comparison point is the bottom joint of the ring finger, called 'ring mcp' in the hand-tracker class.
By the hand-tracker object, the locations of the used joints are provided, from which the absolute distance from fingertip to comparison point is a simple calculation, in this case, a python function. However, this distance is changing based on the size of the hand or how close the hand is to the camera. To counteract this effect the absolute distance is divided by the size of the palm. For the thumb it is the width and for other fingers it is the height of the palm that is used. These distances are called vertical (the height) and horizontal (the width) distances Figure 7. This calculation gives the distance of the fingertips relative to the comparison point. Theoretically, the calculated number does not change if the hand is closer or further away from the camera.

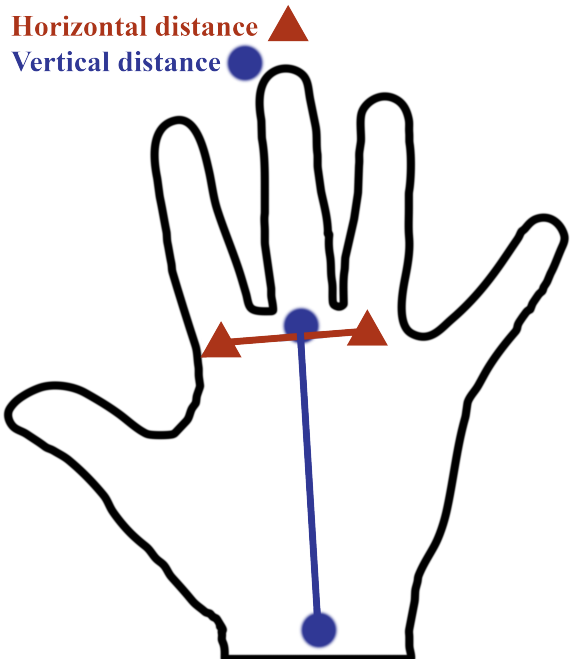

Figure 7: The vertical and horizontal distance. It uses the points on the hand which moves the least amount as the fingers are moving.

The goal is to get a number ranging from 0 to 180. That is because the ESP32 uses degrees to drive the servo motors. It is achieved by first getting a number between 0 and 1 and then multiplying by 180.
To get a number in this range, first, the minimum and maximum values of the relative finger locations have to be found. To find them we conducted several tests for each finger. By regarding the relative finger positions given by the program as lists we used a search algorithm to find the maximum and minimum values. These values are unique for each finger and stored in a dictionary.
Then by the following formula a number between 0 and 1 is given where 1 represents the open finger and 0 represents the closed one.

$$\frac{D_r - D_{min}}{D_{max} - D_{min}}$$

Where
D_r = Relative distance of the finger tip from comparison point.
D_{min} = Relative distance value when the finger is closed.
D_{max} = Relative distance value when the finger is open.

The problem with this solution is that if the hand is rotated in a plane not parallel to the plane of the camera the size of the palm significantly changes resulting in errors in the calculations. To counteract this effect a limit is set for the user on how much it is allowed to turn their hand.
The rotation of the hand is recognized by dividing the vertical distance with the horizontal one. When the hand is turned in one direction, the result of this division changes. We made several runs to determine the accepted range. This approach is not perfect since if the hand is rotated on several axis at the same time it can result in an unrecognized turn. However the goal is to aid the user in correct usage and notify them when the calculations are skewed. Not to catch intentional misuse. When the hand is closed the program allows bigger rotations since a close hand is more precisely recognized. When the hand is in an incorrect position for more than a second, the user is notified on the laptop

and the data transmission to the ESP32 is stopped.

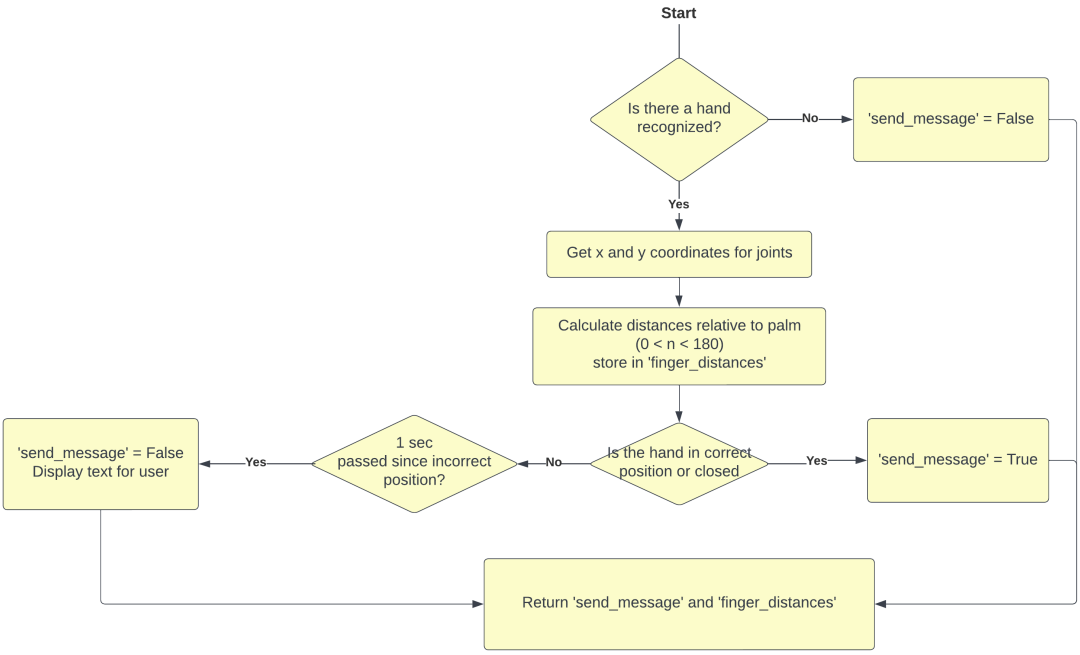

Figure 8: The flow of the 'get_positions' function. It uses a hand-tracker object to get absolute landmark locations. From that it calculates the relative finger locations. It also checks if the hand is in correct position.

UDP package sender

The final part is to send the UDP message. The relative finger tip positions are transmitted to the ESP32. First the information has to be formatted in a specific way for the ESP32 to be able to decode it. All the locations are conjugated into one string which is then sent out using the socket library in Python.

3.4 Implementation

Arduino/C code

Controlling the actual servo motors is the code on our ESP32, written in the Arduino IDE, which uses the C/C++ language. The basics of this code is taken from our lectures, made by lecturer Mads Hobye, this basic functionality is mostly connecting the ESP32 to a local WI-FI network. The connection protocol is made with the Wi-Fi library, as it just takes a single line of code with the name and password for the connection, with some added readability for the user printed to the console in the different stages of connecting (before, during, after). For our purpose we utilized a phone with data sharing, as using the WI-FI at Roskilde University would result in being unable to access the UDP port. This is because the WI-FI at Roskilde University requires a user login instead of a simple WI-FI password. Once a connection has been established, the console in the IDE prints out the local IP address that the UDP sender needs to send through, the local UDP port may also need changing, depending on which are utilized on the network. The upper part of the code setup includes both the packet for the WI-FI connection, along with the packet for connecting our Servo motors to the ESP32.

Here we also define which pins each of the different motors/fingers is connected to. We've also included the list of possible connections that the servo motors can have to the ESP32, so each user may create their own optimized setup.

After some more correction in the setup, such as defining the period Hertz of each motor and actually "attaching" them in the code, the loop begins where the ESP32 will pull all the UDP packets being sent out from the port. Due to how UDP packets are sent, it arrives to the ESP32 as a *char array*, rather than an *integer* that the servos can utilize, therefore it is first split up into different segments of their respective 3 numbers. This is done by the message being split up by percentage signs (%), which we can then ask the program to divide the array by, after which they are parsed to *integers*, which are then sent to their individual servo motors.

3.5 Physical product components

For the project, we have had to work with a lot of physical electronic components. This included making certain design choices for the circuit and the different components when we created the physical hand, to ensure that it was able to work with relative ease. Therefore, we will also be diving deeper into the different components, their capabilities, and how we implemented them.

ESP-32 DevKit v1 board

So far, we have already mentioned our chosen type of micro-controller, an ESP32. However, there is different variants of the ESP32, all with different circuitry and pin-setup. The specific model of ESP32 we make use of, is the DOIT ESP-32 DevKit v1 micro-controller. It features a secure boot and a secure reset button, a CPU consisting of a Xtensa dual-core (or single-core) 32-bit LX6 microprocessor (operating at 160 or 240 MHz and performing at up to 600 DMIPS) and a memory of 520 KiB SRAM. This gives this board the calculation and memory capabilities fit for rather simple to taxing or complex protocols. It also includes both motor and LED pulse-width modulation (PWM), making it fit for controlling our required motors[12].

Figure 9: The ESP-32 DevKit v1 microcontroller
Source: Elektronik Lavpris *https://elektronik-lavpris.dk/p148359/sbc-nodemcu-esp32-nodemcu-esp32/*.

Power to the DOIT ESP-32 DevKit v1 is supplied via an on-board Micro USB-B connector or directly via the "VIN" pin. The power source is selected automatically. The board can operate on an external supply of 6 to 20 volts. However, if using more than 12V the voltage regulator may overheat and damage the device. The recommended range is 7 to 12 volts [7].

Our implementation of this board lies in the fact that it functions as the "brain" of the electrical system. With physical connection to everything in the system, it's job is to direct current charges based on coded input. In the system it is powered by a power-bank, with an output of 5 volts, and is connected to a common ground for the system. Besides that, it also has 5 output connections (defined as such in the code), located at pin D18, D21, D22, D23 and A2(aka. D32). Below is a pin output diagram of the micro-controller.

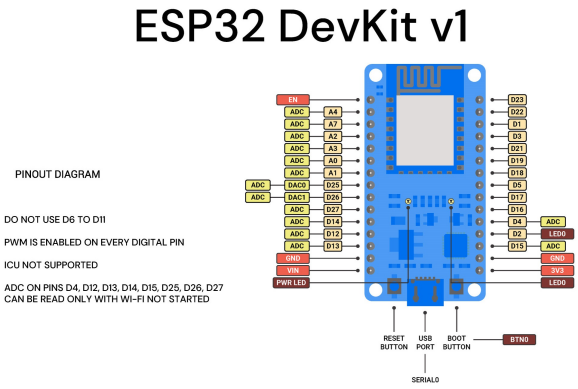

Figure 10: The pin overview of the specific micro-controller in use.
Source: Zerynth *https://olddocs.zerynth.com/r2.6.2/official/board.zerynth.doit_esp32/docs/index.html*.

As described earlier in the report, this board has the ability to connect to the internet via Wi-Fi and can therefore be coded to both receive and send data-packets. This, as well executing protocols based on the data received, is the micro-controller's function in the electrical system.

Tower Pro Micro Servo 9g motors

For our motors, we chose to use 5 Tower Pro Micro Servo 9g motors. These motors are a lightweight easily programmable model, with 180-degree rotation. It is versatile, as it can be operated by any kind of servo code, hardware or library made for controlling servo motors. It also comes with three different attachable arms.
They have a stall torque of 1.8kg, with a rotation speed of 0.1s/60 degrees. This makes them more than fit to pull at the fishing wire of the hand's fingers, with an appropriate time delay and motion speed. However, it also means that we are not able to manipulate the speed of the rotation. The motors can operate when powered by 3.0V-7.2V [11].
The motor has three wires attached, one for ground connection, one for power connection, and one for control input. Both the ground and the power supply wires are connected to the corresponding charge lanes on the breadboard, with the input wire connected to one of the appointed output pins of the micro-controller. The input-wire is in charge of giving directions to the servo, about which way the motor should turn, and for how long it should be powered. The charge of the input wire determines the direction of the rotation, and the length of time it is powered determines the number of degrees in the rotation.
The power wire of all 5 motors is connected to the same power output with a charge of 5V, comfortably fitting in the space between the motor max and min voltage for operation.

Figure 11: The Tower Pro Micro Servo 9g motor, with all standard attachments.
Source: Ardustore *https://ardustore.dk/produkt/tiny-micro-nano-servo-motor-9g-sg90.*

Breadboard power supply module
While we power the ESP32 with a power-bank, we also need to use a stronger power source for the 5 motors, without going over their maximum operating charge. For this we make use of a breadboard power supply module.

This module is an addition to the breadboard setup for our project. In simple terms, it is able to attach a DC power source of anywhere in between 6.5V-12V and then redirect the current to its different output pins. The board has two independent channels of power output for breadboards. These power channels can be independently configured for 3.3V, 0V, and 5V operations, meaning we are able to attach different elements to this board even if they require different voltages to function [5]. The board also offers a push switch to turn OFF and ON the entire power supply module, with an LED showing whether the power connection is on or not.
However, the capabilities for the breadboard power supply module to control all power for our system, also relies on the available compatible batteries or power sources. If one power source alone is not powerful enough to supply enough voltage to every component, there is a need for a second power source with no connection to the module.
In our electrical system, this module functions as a middleman between the servo motor and the actual power supply, a 9V battery. This gives us the ability to keep the 5V charge for the servo motors at a consistent current, with any spare energy running back to the negative side of the battery. This therefore also functions as the floating ground in the system. While we could power the ESP32 at a separate 5V output, we would need to have a battery attached to the breadboard power supply module of at least 10V, if not a little more. As we were not able to construct a battery connector to fit this module for that charge, we chose to power the ESP32 separately with a common ground.

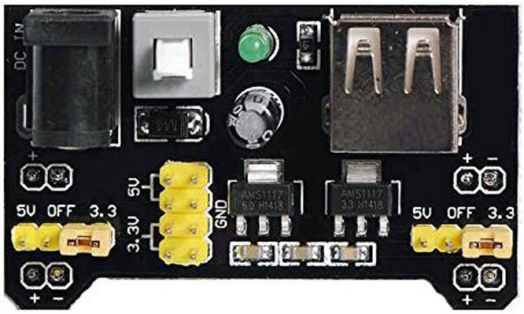

Figure 12: The breadboard power supply module used in the project
Source: Components 101 *https://components101.com/modules/5v-mb102-breadboard-power-supply-module*.

Connections
The last crucial part of the electrical system is of course the different connections carrying the different charges, and what they consist of. We have previously mentioned the use of a breadboard and wires. While a wire is a flexible metallic conductor, usually insulated, and used to carry electric current in a circuit, a bread board is a bit more complex. A breadboard is designed to let users create circuits without the need for soldering, effectively simplifying the process of making electrical connections.

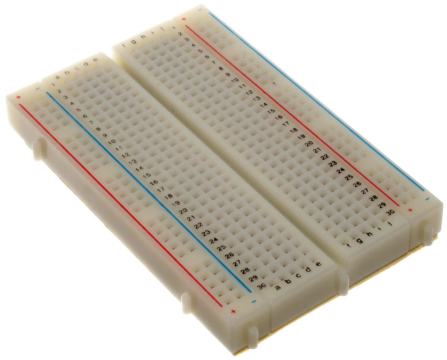

Figure 13: A standard breadboard found in a hobby electronics kit
Source: Wikipedia *https://da.wikipedia.org/wiki/Eksperimentalplade#/media/Fil:400_points_breadboard.jpg*.

This works as the breadboard have several rail connections within, typically some made for standard connection, and some made for power connection. These two types of rail connections run in different directions on the board. Each rail then has several connection ports, where the user will put their wire in until it touches the rail. Figure 14 displays an overview of how the different rail connections on a breadboard typically works[6].

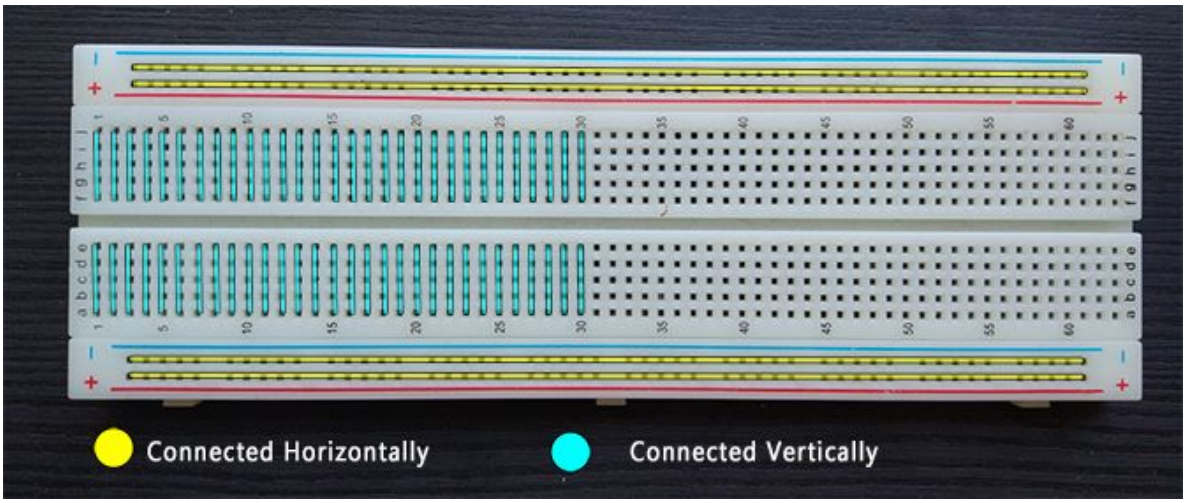

Figure 14: The current connections and their direction, in a standard breadboard
Source: MUO *https://www.makeuseof.com/tag/what-is-breadboard/*.

3.6 Evaluation

As the last step of the iterative model, we evaluate on the prior steps in our project. The evaluation is worked with in the Discussion (Section 5) which will lead to a conclusion of the whole project.

4 User guide

User guide

For our robotic hand to work as intended, there are some things that the user needs to be aware of. This user guide will go through the requirements for the programs and how to use the hand.

Before starting the program

Before starting any program, the user will need to make sure that all needed pip packages, containing necessary libraries, has been installed. Below is shown the utilised libraries for the python files, which controls the hand.

```
15  import cv2 # pip install opencv-python
16  import mediapipe as mp # pip install mediapipe
17  import threading
```

Figure 15: Importing cv2, mediapipe and threading libraries in the Python code.

```
10  import hand_track_class
11  import time
12  import math
13  import socket
```

Figure 16: Importing time, math and socket libraries in the Python code. the hand_track_class is a file created by us.

The last thing to do before starting any program, is to ensure there is power to the hand. This is just to check that the light is on, both on the power source, breadboard, as well as on the ESP32.

Internet connection

With that taken care of, the fundamentals for the program to run is in place. However, the user may also need to edit specific values in both the python code as well as the arduino code. Theses values are the IP- address for the ESP32 and the name and password for the internet connection used by the ESP32.
For the python program to be able to send data packets with instructions to the hand, both the computer used and the ESP32 micro-controller, will need to be connected to the same network. This works best with a 2,4GHz connection, and we recommend using a cell-phone hotspot as you can easily see which devices are connected at all times. Furthermore, the user may also need to change the value representing the IP-address of the micro-controller, as this has been known to change. In Figure 17 and Figure 18 pictured the parts of the different codes in charge of these editable values.

```
66      # SENDING UPD MESSAGE
67      MESSAGE = "{:0>3d}:{:0>3d}:{:0>3d}:{:0>3d}:{:0>3d}"
68
69      UDP_IP = "192.168.69.216"
70      UDP_PORT = 4210
```

Figure 17: Setting the IP address of the micro-controller in the Python code.

```
24  // Replace with your network credentials
25
26  const char* ssid = "Galaxy A4083EC";
27
28  const char* password = "mwlj7716";
29
```

Figure 18: Setting the necessary connection values in the Arduino code. The ESP32 connects to the hotspot shared by a phone.

To find the correct IP-address for replacement, the user can run the Arduino code on the ESP32 (connected to a computer), and open the serial port. When a connection to the network is established, it will show the local IP address of the micro-controller. Alternatively, if a cell-phone hotspot is used for the network, the user should be able to see the IP-address of all connected devices.

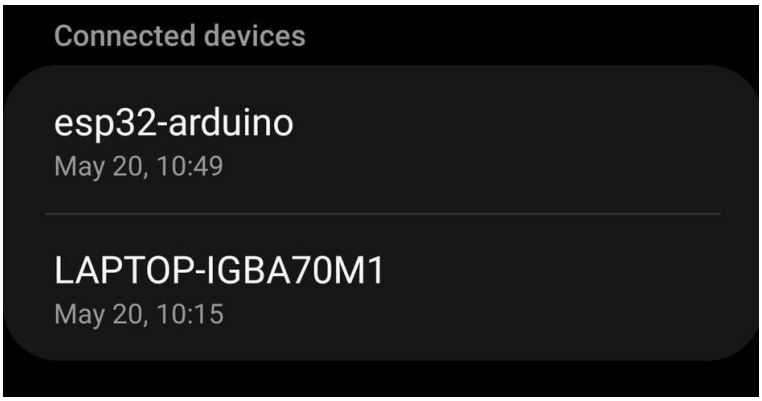

Figure 19: The Android phone used as a hotspot. It is then lists out the connected devices which are the laptop and the ESP32.

Starting use
When the power to the hands elements, and internet connection is established to both the hand and the computer, the user is free to start the Python program. It will take a few seconds to open the interface, but from then on the hand will be usable. When using the program, it is important to keep the users hand at a correct angle, for the most precise control of the hand. If the user tilts their hand into a so called "illegal position", a message will be displayed on the web-cam interface. The user is free to change from one hand to another, and different users can change control of the hand with no difficulty, even with different sized hands or from different distances.

5 Discussion

5.1 Thoughts on the project

In the beginning of this project we set goals that we wanted to achieve throughout the project (See Section 3.1. In this section of the discussion we want to discuss how those set goals were treated in the process. We also want to display other thoughts we had on the project in retrospect.

As we started working on our project our goal was to be able to use a hand tracker to control a bionic hand. We wanted to be able to communicate a program to a physical computed bionic hand as an art installation. We had similar interest in languages we wished to learn or get familiar with during the project. Those interests were about learning how to program an Arduino, and learning how to code in Python language, where we ended up working with an ESP32 and using Python as our main language. We set the goal of learning how to connect two different program languages as we worked with C and Python, where looking back on the process, we feel like we have been able to live up to those goals.

We are satisfied with the product we achieved to construct. We were able to accomplish our goals by making an electrical circuit that made it possible for the hand tracker to communicate with the ESP32, which would communicate the message on to the servomotors, and in the end, our 3D printed hand. The process of making it was smooth and we were not stuck more than a day without being able to move forward in one way or the other. We started the process early enough to get done with the physical computing elements and code in mid May. The communication and delegation of tasks between the group, and the different strengths of the group members was balanced and very organically organized. We used Microsoft Teams for most of our process and delegated tasks virtually as well as in person.

One of the things we could have done better was documenting. During our process we did not prioritize taking pictures and didn't document the steps taken, in real time, to achieve our finished product. We feel that this is something to be done better in a future project, as the report is going to miss some elements of learning. With this, we also wished to have started the report writing earlier. We started to do the physical computing as the first thing in the project. Then we wrote some pages before our midterm presentation, but began writing it more continuously in the start of May. Therefore a lot of the points, as stated above, was not documented in real time and therefore is written with a point of view that looks back relying on theory and memory.

5.2 Relevancy of the project

There are multiple ways to interpret the question; "is the project relevant?". Is it relevant for the individual group members to study? Is it relevant according to the project prompt from the university? Is it relevant to society? For this project, there are also different ways in which it is a relevant project, for different reasons.
Firstly, of course the project is relevant to the prompt given to us at the start of the semester, but the nature of the project has also been relevant to our education and our group members in other ways. The project involves areas of programming which the group members all agreed to include, as we all wished to have more experience in, and subsequently learn more about, these specific ways of programming. Especially programming of physical components is relevant to our situation as students of Roskilde University, as we have access to proper facilities allowing us the opportunity to work with it confidently. With this, we are specifically referring to FabLab RUC and their facilities, which we chose to take advantage of to create a physical programming project.

Secondly, the use and experience with newer, open-source technology, such as the pre-trained hand tracker from Mediapipe, is also quite relevant for us as university students to get to work with and

learn to adapt to our own needs. Especially since, while it is important to understand why and how elements of your project work, it is not always practical to create every element from scratch when working on a bigger project. We find that the ability to work with and adapt existing technology to your own purpose, is a quite relevant skill to train. This is true both for personal projects but also for professional work, as innovation will always be built on the backs of older innovation.
It is with this sentiment that we also find the use of inter connectivity in-between devices, to be a very relevant subject to include in our project, as it is a very prominent area of modern technology and system development, especially with the rise of IoT (Internet of Things).

5.3 Problems and solutions

We faced several unexpected issues during the development. In this section, we discuss the major challenges and the lessons these obstacles taught us. The major difficulties arose when working with physical components, when we connected different programming languages and when we translated real-world hand movement to digital information.
To make the hand move we used servo motors which are essential for the product. However the behaviour of the servo motors did not always align with our expectations. When we experienced unexpected behaviour it was hard to pinpoint where the error originated. As the error could be in the python code, the C code or servo motor/hardware failure. To determine the area of failure we used rigorous testing methods. We learned to test the components of the physical product separately on simple systems which we know are working. We also printed the expected behaviour to the terminal after each phase which helped eliminate system errors. Additionally we noted to keep the testing in mind during the design phase.
For the physical programming we used C but for the hand-tracker, the Python language. Connecting the two demanded carefully specified programming. For quick and reliable message transfer through UDP the information was organized into one string which has to be processed on the ESP32. The communication between the two languages turned out to be less than fluent. We had to figure out the specific ways for information transfer. We concluded that for complicated projects the least possible different programming languages are preferred. It leaves less space for errors, leads to quicker development and makes it easier to change things or to further develop the project. As an example in a similar project we would consider to use Raspberry Pi over Arduino which would allow us to program everything in Python.
As described in the analysis, since the z-axis, which determines the distance of the hand-tracker is inconsistent, we decide to only use the x and y coordinates. It gave us exciting challenges on how to get consistent results for different hand sizes, and hand positions. And how to detect incorrect hand positions. This process evoked our creativity and lead to several solution ideas.
However, even when we found a solution it was hard to make sure it is a general solution. We faced questions like; what if someone has very different hand proportions? How different can it be? What would it do to our program? We had to try to exploit our own system and see how can a user turn their hand into incorrect positions to eliminate as many errors as possible. We had to think of unexpected user interactions. After all we decided to keep in mind the goal of the program and align the development accordingly. We realized that for a bulletproof result, testing is needed in a lot of diverse environments.

5.4 Possible further improvements

If we were provided with more time and resources, we would have considered certain ways in which we could have improved our bionic hand. Seeing as currently it has no real practical applications, most of our improvements would revolve around improving it past its current simple nature. The first step to make the bionic hand act more natural would be to improve its range of motion. This would include rotation in the wrist, individual hand joints rather than only finger joints, and potentially an arm to cover a wider area. Most of this could be accomplished with another hand design & motor setup, as

the rotation in the wrist would put stress on the strings that close the fingers in the current setup.

The next improvement would be to incorporate the essential functionality of a hand, to be able to grab onto objects. This would require some sensors to be able to sense when something is already held by the bionic hand, as forcing the motors to close the bionic hand fully could damage whatever is being held. The forces needed to properly hold onto different objects and surfaces does vary, and would require calibration for these surfaces. Additionally the texture of the artificial hand could be changed to more closely match the grip of its biological counterpart.

During the last month we also began to remedy the problem in which the output from the hand tracker to the ESP32 would be skewed if the hand was rotated in any direction. Our solution was to set a timer if the hand is detected as rotated, and then stop the output if it stays in that position for a set amount of time. The hand can be rotated in many ways, which is quite difficult for the hand tracker to recognize and account for. While correcting is possible, it is also time consuming for our current method, which brings up the limits of the webcam hand tracker. Ideally, we would be able to construct a glove to track the user's hand movements, which would be more responsive and easily allow us to incorporate rotation and grip. The gloves could additionally be supplied with resistances for when the user is grabbing something with the hand, to give a clear indication to the user that they are grabbing something.

6 Conclusion

The goals that we set for the project turned out to be realistic. They were all focused on the software and hardware components which led to a project heavily focused on the actualization of our product. This means that we focused on the project's physical aspects, almost up until the last month, and did not give ourselves the possibility of writing the project while we were doing the steps to achieve our goals. Even though our project does not come with an aim of creating technological innovation, the project has turned out to be a very relevant project for us to understand computer science better, and test different possibilities in the field. The project ended up giving us the chance to learn relevant aspects of physical computing, as well as a chance to learn different programming languages and have them communicate with each other. We faced several unexpected obstacles during the development that made us learn more about the technologies we worked with. The challenges gave us insight into the different environments of computer science we wrote the project within, like the ones mentioned above but also how to turn real life properties, like the location of a hand, into virtual information. While the function we applied worked as we planned, there is room for further improvement which would include giving it more practical uses, and maybe work with the hand in a more problem oriented frame work.

7 Bibliography

References

[1] Why does a circuit always have to have ground? URL http://www.learningaboutelectronics.com/Articles/Why-does-a-circuit-always-have-to-have-ground. Accessed 27/05/2022.

[2] Hands-mediapipe. URL https://google.github.io/mediapipe/solutions/hands.html. Accessed 27/05/2022.

[3] Ecstuff4u for electronics engineer. URL https://www.ecstuff4u.com/2019/08/udp-advantage-disadvantage.html. Accessed 27/05/2022.

[4] Why do electrical circuits need to be grounded?, September 2019. URL https://earlybirdelectricians.com/blog/electrical-circuits/. Accessed 27/05/2022.

[5] C. 101. Breadboard power supply module, June 2020. URL https://components101.com/modules/5v-mb102-breadboard-power-supply-module. Accessed 27/05/2022.

[6] I. Buckly. What is a breadboard and how does it work? a quick crash course, Jun2 2018. URL makeuseof.com/tag/what-is-breadboard/. Accessed 27/05/2022.

[7] Grobotonics. Esp32 development board - devkit v1. URL https://grobotronics.com/esp32-development-board-devkit-v1.html?sl=en#:~:text=Power. Accessed 27/05/2022.

[8] P. J. User datagram protocol, August 1980. URL https://www.ietf.org/rfc/rfc768.txt. DOI 10.17487/RFC0768.

[9] A. K., L. S. Harder, E. D. Galsgaard, K. Heiseldal, and J. Meyer. Magic mirror interface, December 2021. URL https://github.com/esbendg/IDS-HandIn/blob/master/hand_track_class.py. Accessed 27/05/2022.

[10] T. R. Kuphaldt. Vol. i - direct current (dc), 1996. URL https://www.allaboutcircuits.com/textbook/direct-current/chpt-1/electric-circuits/. Accessed 27/05/2022.

[11] I. C. London. Servo motor sg90 data sheet. URL http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf. Accessed 27/05/2022.

[12] R. Mischianti. Doit esp32 dev kit v1 high resolution pinout and specs, December 2021. URL https://www.mischianti.org/2021/02/17/doit-esp32-dev-kit-v1-high-resolution-pinout-and-specs/. Accessed 27/05/2022.

[13] S. J. Nowlan and J. C. Platt. A convolutional neural network hand tracker. *Advances in neural information processing systems*, pages 901–908, 1995.

[14] T. E. of Encyclopaedia Britannica. electric circuit, 2018. URL https://www.britannica.com/technology/electric-circuit.

[15] RUC. Subject module project in computer science, February 2021. URL https://study.ruc.dk/class/view/25264. Accessed 27/05/2022.

[16] SDLC. Iterative model: What is it and when should you use it?, December 2016. URL https://airbrake.io/blog/sdlc/iterative-model. Accessed 27/05/2022.

[17] A. Sharma. Definitive guide: Threading in python tutorial, May 2020. URL https://www.datacamp.com/tutorial/threading-in-python. Accessed 27/05/2022.

[18] C. Tan. Esp32 vs arduino, 2021. URL https://all3dp.com/2/esp32-vs-arduino-differences/. Accessed 27/05/2022.

[19] G. L. Team. Opencv tutorial in python, August 2021. URL https://www.mygreatlearning.com/blog/opencv-tutorial-in-python/. Accessed 27/05/2022.

[20] velvedev. Bionic hand with 5 individual fingers, March 2021. URL https://www.thingiverse.com/thing:4807141/files. Accessed 27/05/2022.

[21] F. Zhang, V. Bazarevsky, A. Vakunov, A. Tkachenka, G. Sung, C. Chang, and M. Grundmann. Mediapipe hands: On-device real-time hand tracking. June, 2020.

A Appendix

Link to GitHub page where the Python and C code can be reached:
`https://github.com/Exxa1/bionic_hand_SP4/`

Link to the Overleaf document where to code of the project report can be observed:
`https://www.overleaf.com/read/yfppvyfrxcdt`

```python
"""
HAND TRACKER CLASS
Subject module project in Computer Science
Semester: F2022
Authors: Azita Sofie Tadayoni, 68888
Emma Kathrine Derby Hansen, 71433
Alexander Kiellerup Swystun 72048
Áron Kuna 70492

This code is inspired by the
Interactive Digital Systems exam turn in assignment from 2021 autumn semester by Aron Kuna
Exam code: U25233
Github page of the code used: https://github.com/esbendg/IDS-HandIn/blob/master/hand_track_class.py
"""

import cv2 # pip install opencv-python
import mediapipe as mp # pip install mediapipe
import threading
mp_drawing = mp.solutions.drawing_utils
mp_drawing_styles = mp.solutions.drawing_styles
mp_hands = mp.solutions.hands

class Hand_track:

    #Initialize object
    def __init__(self):
        self.hand_on_img = False

        self.thumb_tip = None
        self.index_tip = None
        self.middle_tip = None
        self.ring_tip = None
        self.pinky_tip = None
        self.ring_mcp = None
        self.index_mcp = None
        self.wrist = None

        self.text_on_img = False
        self.text = None
        self.text_style = {
            "font"                   : cv2.FONT_HERSHEY_SIMPLEX,
            "bottom_left_corner" : (100,100),
            "font_scale"              : 1,
            "font_color"              : (0,0,255),
            "thickness"              : 3,
            "line_type"               : 2
        }

        self.cap = cv2.VideoCapture(0)
        self.img_state = 0                # 0 is off, 1 is on, 2 is closing window
        self.stop_thread = False


    def start(self):
        def loop():
            with mp_hands.Hands(
                static_image_mode=False,
```

```
                max_num_hands=1,
                model_complexity=0,
                min_detection_confidence=0.5,
                min_tracking_confidence=0.5) as hands:
                while self.cap.isOpened():
                    success, image = self.cap.read()
                    if not success:
                        print("Ignoring empty camera frame.")
                        continue

                    image.flags.writeable = False
                    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
                    results = hands.process(image)
                    if results.multi_hand_landmarks:
                        self.hand_on_img = True
                        self.index_tip = results.multi_hand_landmarks[0].landmark[mp_hands.HandLandmark.INDEX_FINGER_TIP]
                        self.middle_tip = results.multi_hand_landmarks[0].landmark[mp_hands.HandLandmark.MIDDLE_FINGER_TIP]
                        self.ring_tip = results.multi_hand_landmarks[0].landmark[mp_hands.HandLandmark.RING_FINGER_TIP]
                        self.pinky_tip = results.multi_hand_landmarks[0].landmark[mp_hands.HandLandmark.PINKY_TIP]
                        self.thumb_tip = results.multi_hand_landmarks[0].landmark[mp_hands.HandLandmark.THUMB_TIP]
                        self.ring_mcp = results.multi_hand_landmarks[0].landmark[mp_hands.HandLandmark.RING_FINGER_MCP]
                        self.wrist = results.multi_hand_landmarks[0].landmark[mp_hands.HandLandmark.WRIST]
                        self.index_mcp = results.multi_hand_landmarks[0].landmark[mp_hands.HandLandmark.INDEX_FINGER_MCP]
                    else: self.hand_on_img = False

                    if self.img_state == 1:
                        # Draw the hand annotations on the image.
                        image.flags.writeable = True
                        image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)
                        if results.multi_hand_landmarks:
                            for hand_landmarks in results.multi_hand_landmarks:
                                mp_drawing.draw_landmarks(
                                    image,
                                    hand_landmarks,
                                    mp_hands.HAND_CONNECTIONS,
 mp_drawing_styles.get_default_hand_landmarks_style(),
 mp_drawing_styles.get_default_hand_connections_style())
                        # Flip the image horizontally for a selfie-view display.
                        image = cv2.flip(image, 1)

                        #Text on image
                        if self.text_on_img:
                            cv2.putText(image, self.text,
                                self.text_style["bottom_left_corner"],
                                self.text_style["font"],
                                self.text_style["font_scale"],
                                self.text_style["font_color"],
                                self.text_style["thickness"],
                                self.text_style["line_type"])

```

```
                        #Displaying image
                        cv2.imshow('MediaPipe Hands', image)
                        (cv2.waitKey(5) & 0xFF == 27) # This line is neccessary for
the image to show up

                    elif self.img_state == 2:
#Close the window then set showImg to 0 (off)
                            try:
                                cv2.destroyWindow('MediaPipe Hands')
                            except: pass
                            self.img_state = 0
                    if self.stop_thread:
                        break

        self.thread = threading.Thread(target = loop)
        self.thread.start()

    def stop(self):
        self.stop_thread = True
        self.thread.join()
        self.cap.release()

    def img_on(self):
            self.img_state = 1
    def img_off(self):
            self.img_state = 2

    def set_text(self, text_to):
        self.text = text_to
    def text_on(self):
        self.text_on_img = True
    def text_off(self):
        self.text_on_img = False
```

```python
"""
MAIN
Subject module project in Computer Science
Semester: F2022
Authors: Azita Sofie Tadayoni, 68888
Emma Kathrine Derby Hansen, 71433
Alexander Kiellerup Swystun 72048
Áron Kuna 70492
"""
# Setting up libraries
from finger_positions import get_positions, stop_hand_tracker
from send_udp import send
import keyboard

# Main loop
while True:
# Calculate relative finger distances
        finger_distances, send_message = get_positions()

# SENDING UPD MESSAGE
        if send_message:
                send(finger_distances)

        if keyboard.is_pressed("q") or keyboard.is_pressed("x") or keyboard.is_pressed("Esc"):
                print("You pressed q")
                stop_hand_tracker()
                break
```

```
"""
CALCULATE FINGER POSITIONS
Subject module project in Computer Science
Semester: F2022
Authors: Azita Sofie Tadayoni, 68888
Emma Kathrine Derby Hansen, 71433
Alexander Kiellerup Swystun 72048
Áron Kuna 70492
"""

import hand_track_class
import time
import math

tracker_object = hand_track_class.Hand_track()
tracker_object.start() # Start handtracker
tracker_object.img_on() # Display image for user
tracker_object.set_text("Incorrect hand position")

position_time = time.time()
send_message = True

limits = {
    "min": {
        "thumb": 0.6,
        "index": 0.7,
        "middle": 0.6,
        "ring": 0.6,
        "pinky": 0.6,
        "rotate": 2.4
    },
    "max": {
        "thumb": 2.8,
        "index": 1.9,
        "middle": 2,
        "ring": 2,
        "pinky": 1.5,
        "rotate": 3.1
    }
}

def rel_distance(finger, compare_to, coord_dict, orientation):
    result = (math.dist(coord_dict[finger], coord_dict[compare_to])/orientation - limits["min"][finger]) / (limits["max"][finger] - limits["min"][finger])
    result *= 180
    if result > 180:
        result = 180
    elif result < 0:
        result = 0
    return int(result)

def get_positions():
    global position_time, send_message

    if tracker_object.hand_on_img: # When a hand is visible on image
            coords = {
                "thumb":(tracker_object.thumb_tip.x, tracker_object.thumb_tip.y),
                "index": (tracker_object.index_tip.x, tracker_object.index_tip.y),
```

```
                "index_mcp": (tracker_object.index_mcp.x, tracker_object.index_mcp.y),
                "middle": (tracker_object.middle_tip.x, tracker_object.middle_tip.y),
                "ring": (tracker_object.ring_tip.x, tracker_object.ring_tip.y),
                "ring_mcp": (tracker_object.ring_mcp.x, tracker_object.ring_mcp.y),
                "pinky": (tracker_object.pinky_tip.x, tracker_object.pinky_tip.y),
                "wrist": (tracker_object.wrist.x, tracker_object.wrist.y)
            }

            horizontal_dist = math.dist(coords["ring_mcp"], coords["index_mcp"])
            vertical_dist = math.dist(coords["wrist"],coords["ring_mcp"])

            finger_distances = { # setting distances between 0 and 1 independent of the distance from camera
                "thumb-rmcp": rel_distance("thumb", "ring_mcp", coords, horizontal_dist),
                "index-w": rel_distance("index", "wrist", coords, vertical_dist),
                "middle-w": rel_distance("middle", "wrist", coords, vertical_dist),
                "ring-w": rel_distance("ring", "wrist", coords, vertical_dist),
                "pinky-w": rel_distance("pinky", "wrist", coords, vertical_dist)
            }

            # See when hand is closed (when pinky and ring tip is lower than ring mcp) - currently not in use
            is_partly_closed = coords["pinky"][1] > coords["ring_mcp"][1] and coords["ring"][1] > coords["ring_mcp"][1]
            # See when hand is fully closed
            is_fully_closed = is_partly_closed and coords["middle"][1] > coords["ring_mcp"][1] and coords["index"][1] > coords["ring_mcp"][1]

            # Check if the hand position is correct
            if limits["min"]["rotate"] < (vertical_dist/horizontal_dist) < limits["max"]["rotate"] or is_fully_closed:
                # open - good to go
                send_message = True
                tracker_object.text_off()
                position_time = time.time()
            else:
                # incorrect hand position
                if time.time() - position_time > 1:
                    tracker_object.text_on()
                    send_message = False # Do not send message
    else:
        finger_distances = None
        send_message = False
    return finger_distances, send_message

def stop_hand_tracker():
    tracker_object.stop()
```

```
"""
SEND UDP
Subject module project in Computer Science
Semester: F2022
Authors: Azita Sofie Tadayoni, 68888
Emma Kathrine Derby Hansen, 71433
Alexander Kiellerup Swystun 72048
Áron Kuna 70492
"""

import socket


UDP_IP = "192.168.101.216" # Changes every time!
UDP_PORT = 4210

sock = socket.socket(socket.AF_INET, # Internet
            socket.SOCK_DGRAM) # UDP

def send(finger_distances, sock = sock):
    # Formatting message for the arduino code
    MESSAGE = "{:0>3d}:{:0>3d}:{:0>3d}:{:0>3d}:{:0>3d}".format(finger_distances["index-w"],
                                                              finger_distances["middle-w"],
                                                              finger_distances["ring-w"],
                                                              finger_distances["pinky-w"],
                                                              finger_distances["thumb-rmcp"])
    print ("message:", MESSAGE)
    sock.sendto(bytes(MESSAGE, "utf-8"), (UDP_IP, UDP_PORT))
```

```
// ARDUINO CODE
// Subject module project in Computer Science
// Semester: F2022
// Authors: Azita Sofie Tadayoni, 68888
// Emma Kathrine Derby Hansen, 71433
// Alexander Kiellerup Swystun 72048
// Áron Kuna 70492


// Load Wi-Fi library

#include <WiFi.h>
//------------------------------
#include <ESP32Servo.h>

Servo servoIndex;  // create servo object to control a servo
Servo servoMiddle;
Servo servoRing;
Servo servoPinky;
Servo servoThumb;
// 16 servo objects can be created on the ESP32

int pos = 0;    // variable to store the servo position
// Recommended PWM GPIO pins on the ESP32 include 2,4,12-19,21-23,25-27,32-33

int pinIndex = 18;
int pinThumb = 21;
int pinMiddle = 22;
int pinRing = 23;
int pinPinky = 32;
//------------------------------

// Replace with your network credentials

const char* ssid = "Galaxy A4083EC";

const char* password = "mwlj7716";

//The udp library class

WiFiUDP udp;

unsigned int localUdpPort = 4210;  // local port to listen on

char incomingPacket[255];  // buffer for incoming packets

 char *  replyPacket = "Hi there! Got the message :-)";  // a reply string to send back

 char * broadcastPacket = "I am here";



void setup() {

  Serial.begin(9600);



```

```
  // Connect to Wi-Fi network with SSID and password

  Serial.print("Connecting to ");

  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {

    delay(500);

    Serial.print(".");

  }

  // Print local IP address

  Serial.println("");

  Serial.println("WiFi connected.");

  Serial.println("IP address: ");

  Serial.println(WiFi.localIP());



  udp.begin(localUdpPort);

  Serial.printf("Now listening at IP %s, UDP port %d\n",
WiFi.localIP().toString().c_str(), localUdpPort);

//-----------------------------------------
  // Allow allocation of all timers
  ESP32PWM::allocateTimer(0);
  ESP32PWM::allocateTimer(1);
  ESP32PWM::allocateTimer(2);
  ESP32PWM::allocateTimer(3);
  servoIndex.setPeriodHertz(50);    // standard 50 hz servo
  servoMiddle.setPeriodHertz(50);    // standard 50 hz servo
  servoRing.setPeriodHertz(50);    // standard 50 hz servo
  servoPinky.setPeriodHertz(50);    // standard 50 hz servo
  servoThumb.setPeriodHertz(50);    // standard 50 hz servo
  servoIndex.attach(pinIndex); // attaches the servo on pin 18 to the servo object
  servoMiddle.attach(pinMiddle); // attaches the servo on pin 18 to the servo object
  servoRing.attach(pinRing); // attaches the servo on pin 18 to the servo object
  servoPinky.attach(pinPinky); // attaches the servo on pin 18 to the servo object
  servoThumb.attach(pinThumb); // attaches the servo on pin 18 to the servo object
  // using default min/max of 1000us and 2000us
  // different servos may require different min/max settings
  // for an accurate 0 to 180 sweep
//-----------------------------------------

}



unsigned long timer = 0;

```

```
118
119
120 void loop() {
121
122
123
124
125
126   int packetSize = udp.parsePacket();
127
128   if (packetSize)
129
130   {
131
132     // receive incoming UDP packets
133
134 //    Serial.printf("Received %d bytes from %s, port %d\n", packetSize,
    udp.remoteIP().toString().c_str(), udp.remotePort();
135 //      Serial.print(incomingPacket);
136
137       // WORKSPACE
138 //      for (int i = 0; i <= incomingPacket.length; i++) {
139 //
140 //      }
141
142       int index, middle, ring, pinky, thumb;
143       sscanf(incomingPacket, "%d:%d:%d:%d:%d", &index, &middle, &ring, &pinky,
    &thumb);
144
145
146       Serial.printf("%d, %d, %d, %d, %d", index, middle, ring, pinky, thumb);
147
148       Serial.print("\n");
149
150       //-------------------------------------
151         servoIndex.write(index);
152         servoMiddle.write(middle);
153         servoRing.write(ring);
154         servoPinky.write(pinky);
155         servoThumb.write(thumb);
156       //-----------------------------------------
157
158       // WORKSPACE END
159
160     int len = udp.read(incomingPacket, 255);
161
162     if (len > 0)
163
164     {
165
166       incomingPacket[len] = 0;
167
168     }
169
170 //    Serial.printf("UDP packet contents: %s\n", incomingPacket);
171
172
173
174     // send back a reply, to the IP address and port we got the packet from
175
```

```
176     udp.beginPacket(udp.remoteIP(), udp.remotePort());
177
178     udp.print(replyPacket);
179
180     udp.endPacket();
181
182   }
183
184 }
185
```

```
Collection of required libraries for the program to run correctly.

Python
    required pip libraries:
        - socket
        - time
        - math
        - cv2 (opencv-python)
        - mediapipe
        - threading
        - keyboard

ESP 32
    required Arduino libraries:
        - ESP32Servo.h
        - WiFi.h

```