Gesture Recognition in Video Meetings

Bachelor's thesis in software engineering

Eskil Gaare Høstad

Jonathan Kjølstad

Faculty of Technology, Art and Design
Oslo Metropolitan University
May 25, 2021

Institutt for Informasjonsteknologi
Postadresse: Postboks 4 St. Olavs plass, 0130 Oslo
Besøksadresse: Holbergs plass, Oslo
Telefon: 22 45 32 00

PROSJEKT NR. 2021-15

TILGJENGELIGHET
Offentlig rapport, privat kildekode

BACHELORPROSJEKT

HOVEDPROSJEKTETS TITTEL	DATO
Gesture recognition for video meetings	25.05.2021
	ANTALL SIDER / BILAG 155

PROSJEKTDELTAKERE	INTERN VEILEDER
Eskil Gaare Høstad Jonathan Kjølstad	Jianhua Xhang

OPPDRAGSGIVER	KONTAKTPERSON
Huddly AS	Elena You, Espen Wøien Olsen

SAMMENDRAG

Studentene har utviklet en proof-of-concept løsning som gjennom maskinlæring kan gjenkjenne håndgesturer, og videre kan bruke disse gesturene til å styre funksjonalitet i et videomøte i reell-tid.

3 STIKKORD
Maskinlæring
Bildeprosessering
Object detection

1 Abstract

Hand gesture recognition is an active area of research in machine learning, which offers a wide range of possibilities for human computer interaction. In the particular circumstances of video conferencing, it is therefore an attractive candidate for facilitating a more engaging environment.

We present a proof-of-concept solution for using gesture recognition to enable interactive features in a video meeting context. Using open source deep learning solutions coupled with simple classification algorithms, we show that recent advances in this field enables the possibility of real-time gesture recognition, without the need for specialized hardware.

2 Preface

In the fall of 2020, we were presented with an interesting topic for our bachelor's project by the IT company Huddly: Exploring possibilities regarding gesture recognition in video meetings.

This paper is a presentation of our endeavor in this field, as well as a presentation of the resulting product. It is aimed at readers with a rudimentary knowledge of programming and mathematics (linear algebra), but these are not mandatory. In addition, some technical terms from fields such as data science and machine learning are used. These are covered in the appended glossary.

2.1 Acknowledgements

Throughout the process we have received a great deal of guidance and assistance.

First, we would like to thank our internal supervisor, Dr Jianhua Zhang, who provided valuable feedback through his expertise on the subject.

We would also like to thank our external supervisors from Huddly, Elena You and Espen Wøien Olsen. Through continuous support, they have facilitated an educational and rewarding development process, with interesting challenges.

Contents

3 Introduction

3.1 Parties

3.1.1 Students/Developers

Eskil Gaare Høstad, Software Engineering
Jonathan Colbjørnsen Kjølstad, Software Engineering

3.1.2 Client

Huddly AS Karenslyst Allè 51, 0270 Oslo

3.1.3 Internal Supervisor

Professor Jianhua Zhang
Deputy Head, Oslomet Artificial Intelligence Lab,
Department of Computer Science, Oslo Metropolitan University.

3.1.4 External Supervisors

Elena You, Data Scientist, Huddly AS
Espen Wøien Olsen, Data scientist, Huddly AS

4 Development Process

4.1 Acquiring the Project

In early September, we approached Huddly about the possibility of a potential bachelor project. We were fortunate to receive immediate positive feedback, and were soon presented with a few different project candidates. Of these, we decided on a project on hand gesture recognition, which seemed like a fascinating topic. A few more meetings were took place, and by early 2021 we had a fairly clear description of the goal of the project.

4.2 Project Description

This section provides an overview of the overall goal, intent, and background of the project. It is largely based on our pre-project report, with a few modifications.

4.2.1 Goal

We will design and develop software that leverages existing deep learning solutions to identify and classify human hand gestures and use them to control functions. This project is aimed to provide our client with valuable insights into new interactive gesture-based functionalities, mainly focused on video communication.

4.2.2 Conditions

Development of the gesture recognition ML model can be based on existing solutions as this is a well researched area. Further it should be noted that the students

are relatively new to the area of machine learning on images. Therefore, it is not expected that the final report will include a detailed analysis of the construction and performance of the model. Rather, the focus of this project is on the implementation of the existing models. Nevertheless, basic classification performance metrics, such as accuracy, precision, etc., should be presented in the final report.

4.2.3 About the client

Huddly is a technology company that builds team collaboration tools. Their goal is to enhance the video communication experience by bringing intelligent cameras into the meeting room. They do this by carefully combining hardware, software, and artificial intelligence solutions to produce conference cameras with intelligent features. Unlike traditional cameras, Huddly's cameras can not only provide a video feed, but by utilizing the on-board artificial intelligence engine, they are able to see, understand, process and respond to what they are seeing. This means they can automate tasks, such as controlling the camera itself, and are also able to generate and report high-quality analytics about what they see. Huddly is always looking for new smart features to implement (*Company Information*, 2020).

4.2.4 Project background

At this point, Huddly aims to gain new insights into hand gestures as a means of human-machine interaction. By using hand gestures to control functions, instead of the usual keyboard and mouse, there is the potential for a more engaging and interactive user experience. This is particularly true for video communication, where a video feed is already present and user functions are usually quite limited. By using gesture recognition to control these functions, video communication could be streamlined into a completely hands-off experience.

4.3 Work methodology

We quickly decided to adopt an agile workflow as the basis of our development process. The main reasons were the open-ended product specifications, as well as it being recommended by our external supervisors. With a small development team of only two students, an agile process would also enable a highly flexible collaboration environment.

4.3.1 Tooling

The circumstances surrounding COVID-19 meant that we could not meet in person with supervisors, or even within the group for most of the project. We knew this would be a major challenge, especially in terms of communication. To deal with this, we tried to use a range of communication tools and platforms:

- Google Meet / Zoom - Video meetings.
- Email - Formal communication with supervisors.
- Slack - This was Huddly's platform of choice for communicating with our external supervisors on an informal level. For example, if we encountered a challenging bug, we could post code blocks and stack traces and get help.
- Discord - We used Discord for VoIP and internal team communication. It allows voice communication, screen sharing, and supports code blocks as text. This was great for having an informal and easy way to communicate on a daily basis.

Another potential challenge was organizing workflow. Any collaborative project of this nature runs the risk of participants working on overlapping domains, which can lead to redundant work or conflicting file-versions (i.e., 'merge-conflicts'). To avoid

this, we wanted a flexible and efficient collaborative process where we could work simultaneously on decoupled domains. This was facilitated by the following tools:

- Git & GitHub - The obvious choice for version control.
- Trello - For organizing our task board and streamlining workflow (see section below).
- Overleaf - For collaborative writing of project report (online LateX editor with collaborative support and version control).

4.3.2 Methodology

At the core, we organized our development into one-week sprints. Each sprint generally had some main goals that was to be achieved. In addition we worked on other tasks as we saw fit.

The tasks were organized in a digital task board, inspired by a Scrum board. The board consisted of four categories:

- Backlog - An overview of work to be done, basically a To-Do list.
- Doing - When work has begun on a task, it is moved to this category and marked with current contributors.
- Review - When a task is completed, it is marked as 'ready for review', and moved to this category. Typically these items were evaluated with our supervisors during the weekly review. However, small changes were often only reviewed internally amongst the students. After being reviewed, the tasks were either moved to the 'Complete' category or back to the 'Backlog' if more work needed to be done.
- Complete - An archive of all completed and reviewed tasks.

Each sprint was recapped at the weekly review with our supervisors from Huddly. In these meetings we would discuss:

- What work was done in this sprint? (Mainly focused on the sprint goals).
- What went well, and what problems did we encounter during the sprint? Can we learn from these experiences and improve our work process?
- Are we facing any obstacles we need assistance with?
- What is the current state of the product? What should be improved? What features should be added?
- What should be our main focus for the next sprint?

To get a better sense of this process, an elaboration on one of our sprints (week 8/9) follows.

4.3.2.1 Sample Sprint (25 February - 4 March)

For this sprint we had 4 main goals: Restructure the codebase, improve thumb classification, and add two new gesture features to control the camera with. We divided these items among us and discussed possible hurdles.

It was clear that developing on one branch while restructuring the codebase on another had significant potential for merge conflicts. This was resolved by identifying a part of the codebase that did not need to be touched by the restructuring. Now Jonathan could work on the restructuring while Eskil worked on improving thumb "openness" classification.

The week then continued on as normal, with occasional online discussions between the students. Before the sprint review we found ourselves with the task board shown in figure 1.

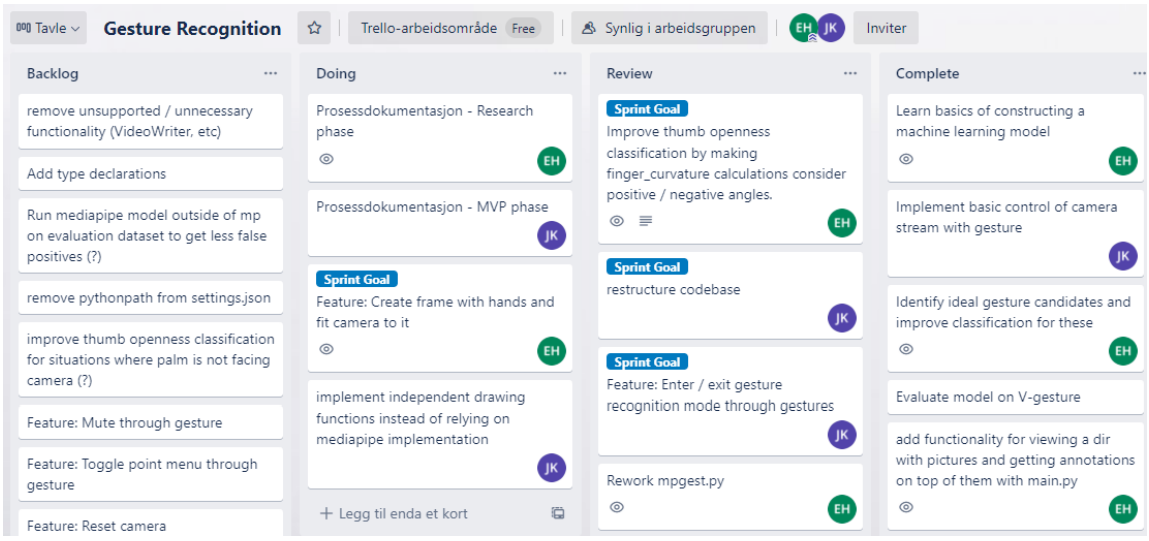


Figure 1: Snapshot of our Trello board as of 25 February 2021.

From figure 1 you can see that all but one of our sprint goals (highlighted in blue) were completed and ready for sprint review, in addition to our work on a few other tasks. Each item where work has begun are marked by the initials of one or both students (indicating which student(s) are working on that item).

At this point we presented the finished items in the weekly meeting with our supervisors. In general they were pleased with our solutions. We also discussed the unfinished sprint task. Specifically, the goal was to have a way to zoom into a frame created with two hands. This presented some challenges because the aspect ratio of the hands did not fit the original aspect ratio. Espen and Elena made a pragmatic suggestion: Mutating the hand shaped rectangle to fit some predefined ratios. This allowed us to continue development on the task without it being too much of a time sink. If we had tried to find the 'perfect' algorithm for this task, we would probably end up spending days on what really was not an integral feature.

4.4 Phases of Development

Our process consisted of four phases:

Week	Main focus
1 - 4	Research
5 - 6	Development of a MVP
7 - 15	Iterative improvement of the product
16 - 20	Finalization of the product, documentation, and writing of project report

4.4.1 Use and Impact of Product Specifications

The basis of our product specifications consisted of a document received from our Huddly supervisors and verbal instructions that were formally written down by us.[1] These initial specifications were particularly relevant during the development of the MVP, as they were in many ways a bare-bones description of the finished product. They served as a useful and independent source of information. For instance, the schematics from the specifications were an important inspiration for the architecture of the system. In addition, the specifications provided a valuable reference point when we discussed the product with our supervisors.

As development moved into the iterative phase, less and less importance was placed on these initial product specifications. Instead, the product specifications would be updated during the reviewfs. For instance, most of the gesture controlled features that are in the final product were added to the product specifications as a result of these meetings. This approach was natural as our project was in essence an

[1]For an in depth description of these, please refer to the product specification section (page 57.

open ended, investigative effort. Only after a basic implementation was developed, could we and our supervisors determine which features would be feasible to implement.

4.5 Phase I: Research

Before we could begin development, there was a need to acquire some knowledge specific to the task at hand, such as:

- Research existing gesture recognition solutions.
- Relevant tools and libraries (e.g. python, keras, tensorflow, opencv)
- Fundamentals of object detection & classification using machine learning.
- Find relevant datasets for training and/or evaluating.

Survey of existing solutions substantiated gesture recognition as a relatively mature domain. We found numerous solutions from both scientific research explorations as well as more practical projects that demonstrated and proved core concepts.

An important factor to consider was that the field of machine learning is rapidly evolving. For this reason, we focused on the most recent solutions we could find. A prominent candidate was the google produced solution MediaPipe Hands (Zhang, Bazarevsky, Vakunov, Tkachenka, & Sung, n.d.).

Using the Mediapipe Hands library we did some experiments on our local video feeds. With relatively simple vector operations we were able to perform fairly decent gesture recognition. This functionality was demonstrated to our supervisors, who were pleased with the results.

However, they also expressed a desire for us to better grasp the underlying technology of this solution. Through gaining a deeper understanding, they felt that

we would be able to make more informed decisions. In an ideal world, they wanted the machine learning model to be configured largely from scratch, to provide a more transparent solution.

This was a little surprising to us. From our preliminary discussions with Huddly, we were under the impression that we were supposed to rely substantially on machine learning models, but not to go too deep into this part of the product. Nevertheless, we accepted the challenge. This being at the time of writing our pre-project report, the contained product specifications were updated to reflect this increased focus on the machine learning model.

On their advice, we began this process by attempting to create a hand-detection model. This was approached from two angles:

- Jonathan worked on creating a hand classification model from scratch using the Keras library, and tried using anchor boxing to effectively turn it into a hand detection model.
- Eskil experimented with using one of the models from the Tensorflow Object Detection API and modifying them to detect hands.

4.5.1 Building a Hand Detection Model From Scratch

A common approach to building an object detection model is to build and train a classifier that is able to score an image on the probability of it containing the object we are looking for. One can then cleverly split an input image into many smaller sub-images in such a way that the objects in the image most likely are completely encased in a sub-image.

One would then feed all the sub-images to the classifier, which would in turn give a high score to the images containing the object, since we know how we split the image in the first place, we now know the location of the object. This approach of

splitting the image is referred to as anchor boxing and employs the thought of divide-and-conquer, turning the detection problem into two smaller problems, optimizing the division of the image and building a classifier.

4.5.1.1 Anchor Box Optimization

The anchor box method proposes to divide the image space into discrete boxes. Then checking the detected probability of a hand in each box. For each box with a positive detection we compare it to nearby boxes and discard all but the box with the highest overlap per non-overlap, or rather intersection over union (IoU). We are then left with a single box around each hand in the input image, and can use these boxes to locate the hands.

The problem at hand is to find the optimal placements of the boxes, and for each placement find some boxes with different shapes and sizes that best encapsulate an arbitrary hand. We want to maximize the number of hands confined by a box while minimizing the total number of boxes. Too few boxes and the chances of detecting a hand would decrease significantly, on the contrary, too many and the detection would slow down exponentially.

Some new methods show promise by optimizing the anchor boxes with the same loss function as the detection model and thus being able to train them in parallel (Zhong, Wang, Peng, & Zhang, 2018). However, the simplest approach would be to simply learn by example, and run through the dataset finding the most common locations, sizes and aspect ratios of hands and manually tune these until the number of hands bound by a box per total box converges.

4.5.1.2 Object Classification Model

The topic of object classification on an image, is by far one of the biggest and widely researched in machine learning. It would therefore be wise to use a well-documented existing solution and adapt it to our needs, rather than reinventing the wheel.

A fully-connected binary image classifier model was proposed which consisted of three main layers. First it would flatten the color channels of the image turning input data from 3D to 2D, secondly a dense fully connected layer would extract relevant features needed to identify a hand, and lastly an activation layer turning the data into a single scalar which encodes the probability of the input image being of a hand.

This was by no means a complex model, but still training the feature extraction layer would take some time. Helpfully Keras already has some pre-trained solutions for this, and the image feature extraction network from ResNet50 (He, Zhang, Ren, & Sun, 2015) could be used, with only the last layer needing to be altered through training. This model was then compiled with binary crossentropy as the loss function and training began.

4.5.1.3 Dataset

For both training of the classifier model and optimizing of the anchor boxes, the EgoHands dataset (Bambach, Lee, Crandall, & Yu, 2015) was used. In this dataset the images were mostly captured by two head-mounted cameras on two people playing board games in different settings and lighting, with the hands being annotated as segmentations in the form of matlab shapes. We then converted this data to csv and converted the segmentation shapes to bounding boxes. The hands contained in the bounding boxes were cropped and labeled as such, and an equal amount of images not containing hands was also extracted. The two classes of the dataset were then

further split randomly into evaluation and training sets.

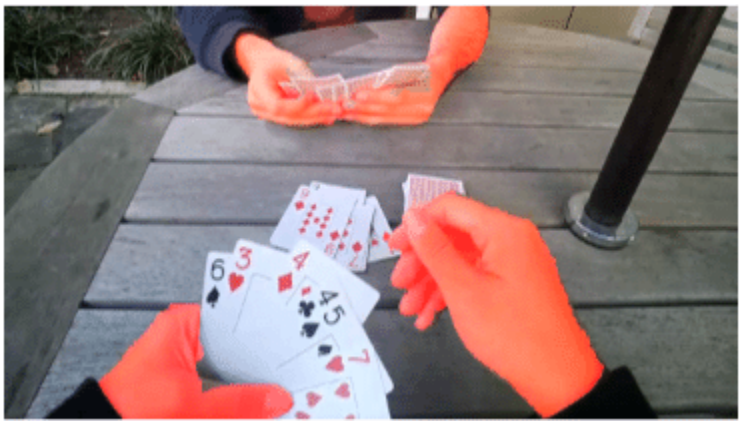


Figure 2: Image from the EgoHands dataset. The annotated hand segmentation has been indicated with a red color.

4.5.1.4 Results

Throughout the training, the results looked good, with an receiver operating characteristics (ROC) curve showing the true-positive rate and false-positive rate both tended towards one and zero, respectively. However, when the model was evaluated on the entire evaluation set, underlying architectural problems began to show. It turned out that the proposed model was more suited for detecting either one of two classes given an image that is guaranteed to contain either. This is similar, but in reality far from our problem.

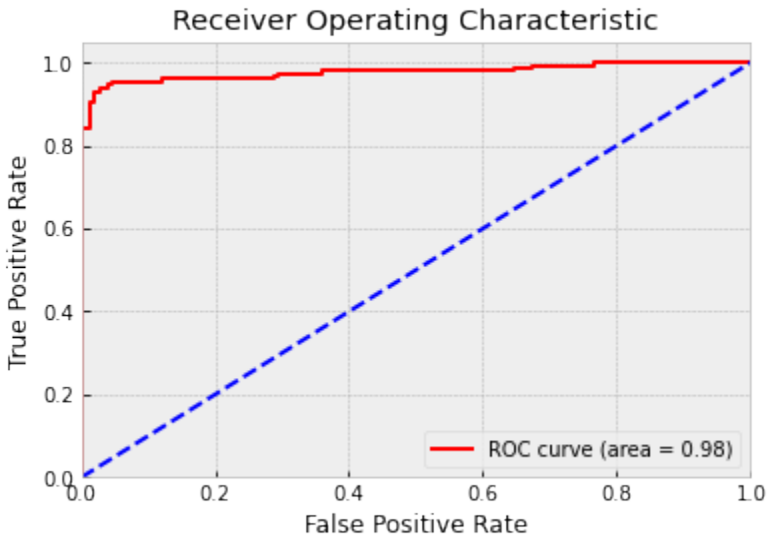


Figure 3: ROC plot showing the ROC curve in red, with a blue line representing the performance with random guessing.

Assuming that this model structure would work in our case with the two classes being "a hand" or "not a hand", it means that we would need equally good data for both hands and anything that is not a hand. Obtaining good data for hands and everything else in the universe is an impossible task. This led to the model overfitting, classifying really well on the training set, but atrociously on real world data.

In hindsight, it is easy to see where we went wrong and why we assumed this model structure would work. Most object detectors around out there use a similar structure, and it was easy to assume that it would also work in our case. The difference in our case is that we are only interested in one class, it is then more naturally solved with regression and is more closely related to an anomaly detection problem.

Even though the proposed model structure was a failure, this part of the research was still a valuable exercise in the creation and training of models. The goal of this was never to create the perfect model - we already knew we would not be able to compete with state of the art solutions, but rather it was to educate ourselves in

how an object detection model works. While this model itself might have failed, it provided a deeper insight into keras and tensorflow, and the scripts written to handle the dataset and the optimization of anchor boxes still could be used elsewhere.

4.5.2 Tuning Object Detection Models to Detect Hands

The idea here was to utilize pre-existing ML-models for object detection, and fine tune them to detect hands. For this purpose, the Tensorflow Object Detection API seemed well suited.

The approach entails using a machine learning model that has been thoroughly trained to detect multiple types of objects. In the case of the Tensorflow API, the object detection models have been trained on the COCO dataset to detect approximately 80 types of objects from around 330K images (Lin et al., 2014; Huang et al., 2016, p. 5).

The model can then be altered to fit a more specific task. In our case, detecting hands in images. To achieve this, we needed to find labeled datasets of hands which could be used to train the model. For this purpose we again used the dataset EgoHands (Bambach et al., 2015).

The next step was to configure an environment for training the model. For this, we knew we wanted to utilize a decent Graphical Processing Unit (GPU), as is customary for machine learning related to images. This ruled out using our own computers. We discussed the possibility of using one of Huddly's servers, but that would entail challenges related to security and availability. In the end, we discovered that Google cloud provides a three month trial for services which includes training on high-end GPU's.

After much tinkering with server side settings, training parameters and datasets, we were at a point where we could do preliminary training with different types of object detection models and find a suitable candidate for hand detection.

4.5.3 End of Research Phase

We were now in the beginning of February, and had some important discussions on our progress with the Huddly supervisors. The feedback was positive in regards to where we were at with the machine learning model. Nevertheless, there was plenty of work still left to do to get a model that could detect and classify gestures.

At the same time, we also had to remember that an integral part of the product was the actual demonstration. It was therefore important to balance these two tasks in a reasonable way.

In the end, Huddly felt that the most important part of the project was the demonstration of gesture controlled functionality. This was conveyed to us through an updated product specification document[2].

The new updated specifications were back inline with how we initially had understood the project. As a result, we put the development of 'our own' ML-model on ice, and looked to start a new phase of development.

4.6 Phase II: Minimum Viable Product

After receiving the updated product specification and a draft of the demonstration application, work could finally begin on constructing an application framework for what was to become the MVP. Since focus now had been shifted away from model creation to a more practical demonstration of the capabilities with an existing model, we needed to pick a robust hand tracking model to use as a foundation.

We quickly agreed on utilizing Google's MediaPipe hands model due to us both already having familiarized ourselves with it through earlier experiments and research. Considering how integral this library would become to our final product, a brief

[2]This document is later referred to as the basis of our product specification, ref page 57

overview is appropriate.

4.6.1 MediaPipe Hands

MP Hands is an open source hand and finger tracking solution developed by google. It combines two different ML models, and from this is able to infer hand landmarks from images (Zhang et al., 2020).

More than just consisting of these two models, MediaPipe Hands implements them in the MediaPipe framework, which is an "open source cross platform framework for building pipelines to process perceptual data of different modalities, such as video and audio" (V. B. Fan Zhang, 2019, para. 2). This adds a layer of abstraction to the models which make them easier to interact with. This framework drastically improves performance for real-time use-cases. An example of this is how it uses the hand landmark model to track the hand, and consequently only needs to run hand detection on frames where the hand is missing (Zhang et al., 2020, p. 1).

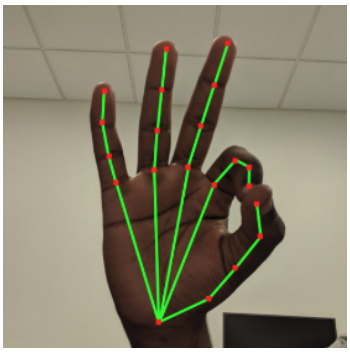


Figure 4: Landmarks given by MediaPipe Hands

The basic flow consists of passing a frame to the MP pipeline. MP Hands passes this frame to its palm detector, which returns bounding boxes for hands detected. These cropped regions are then operated on by the hand landmark model, which returns 21 3D keypoints. An illustration of this can be seen in figure 5.

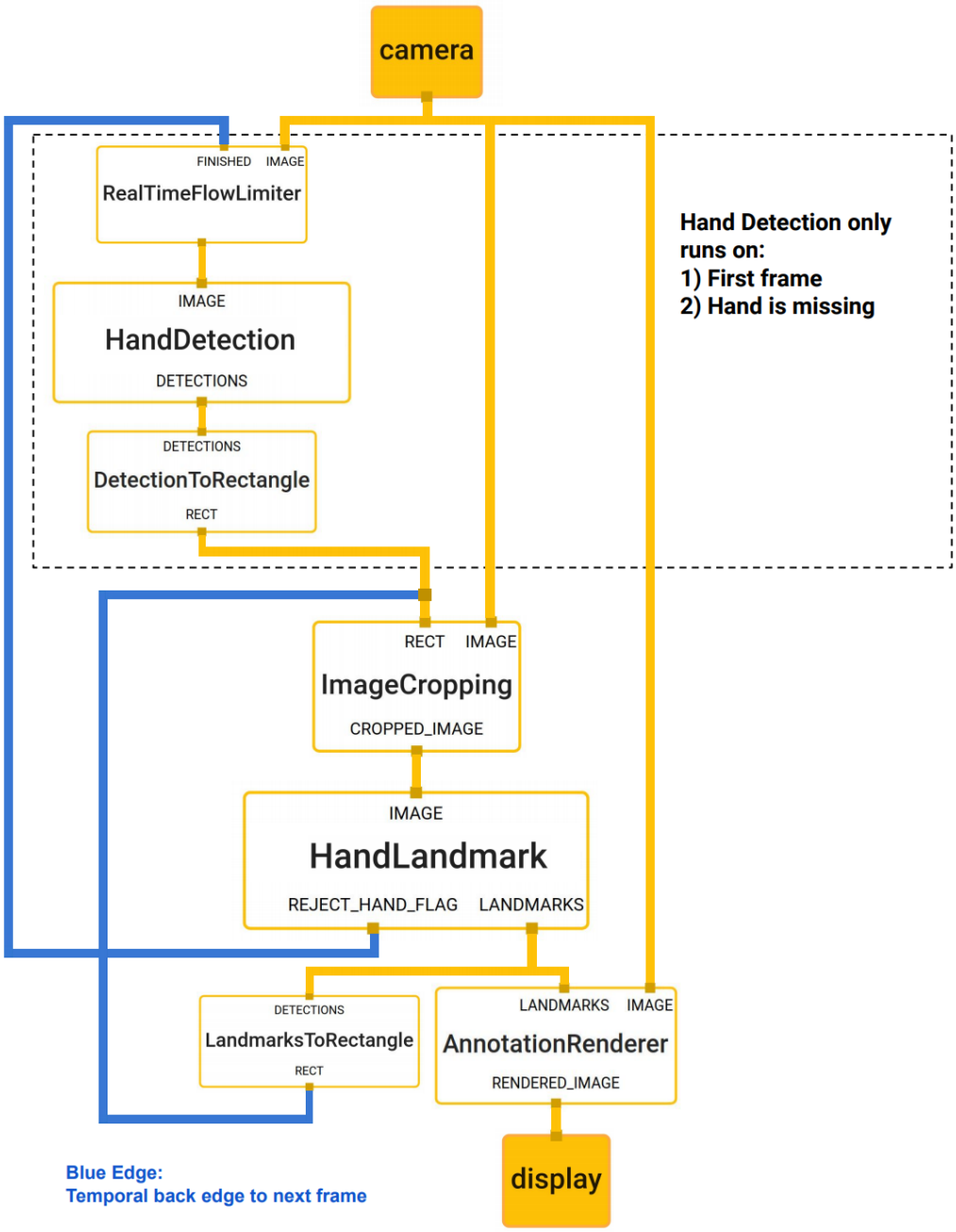


Figure 5: An illustration of the MediaPipe Hands pipeline for real-time (video) use. The figure is sourced from the article "On Device, Real-Time Hand Tracking with MediaPipe" (V. B. Fan Zhang, 2019).

MediaPipe Hands has a number of qualities that made it a natural choice for our use case.

- The models have been thoroughly trained on datasets with a good span of perceived gender and skin tone, ensuring good performance over a range of subjects (G. S. Fan Zhang Valentin Bazarevsky, 2019, p. 4).
- In contrast to most ML solutions, it does not require powerful hardware to perform well, even for real-time detection on video (Zhang et al., 2020, p. 1). This was particularly relevant for our use case, since it made an eventual implementation on Huddly cameras feasible.
- It is open source, and the models are available separate from the rest of the solution. This makes using these models feasible in a production context for Huddly.

4.6.1.1 Clarifying Terms Related to MediaPipe

Going forward, this library will be referenced numerous times. A clarification of related terms is therefore in order.

- MP model - This refers to the conjoined MP Hands models, i.e. both the palm detector and the hand landmark model. In other words, this is the model that accepts image data and outputs a set of hand landmark predictions.
- BlazePalm - The MP palm detection model.
- Hand landmark model - The MP hand landmark model.

4.6.2 Dividing and Designating Tasks

Even though we wanted to push out a working demonstration as quickly as possible, we still needed to get an idea of how to structure the problem into smaller modules. Hopefully spending time on utilizing divide and conquer to split the problem at hand into smaller, more manageable problems here would spare us time and effort iteratively developing it in the future.

We decided to momentarily split up and work on separate things to avoid code conflicts, while periodically checking and reviewing each other's work progress.

- Jonathan worked on developing the groundwork for the program structure, and focused mainly on making modules work together.
- Eskil worked on researching and deciding on which gestures to use, found and converted datasets, and began evaluating our classification pipeline on them.

4.6.3 ML Module for Gesture Recognition

The goal of this module was the detection and classification of hand gestures in a real time, video feed setting. By basing this module on the MediaPipe solution, we identified three main alternatives for the ML module architecture:

1. Use the MediaPipe Hands solution as is with the full framework, and combine it with our own code in our machine learning module.
2. Separate the hand-detection and hand landmark models from the MediaPipe framework, and implement them in our own framework.
3. Write our own hand landmark classifier corresponding to the MediaPipe calculator architecture, which could then be added to a new MediaPipe graph.

Option 3 would entail using C++ for the landmark classifier. As we did not have any experience with the language, it was quickly dismissed. Option 2 did have some benefits, in addition to being more customizable, the finished machine learning module would be more transparent. However, considering that the MediaPipe framework has been developed by highly competent professionals, for one of the world's leading tech-companies, overall performance would be likely to drop. For instance, we did try out an example[3] using this approach, and the drop in frame rate compared to the models implemented in the MediaPipe pipeline was quite significant.

After conferring with Huddly, we ended up going for option 1. This was by far the easiest to implement. Realistically it would also provide the best performance. Considering that the purpose of our product was an effective demonstration, and not a solution ready for production, this was all but a natural decision. The next step would be to create a solution for taking the landmarks output from the MP model, and classifying them into a discrete set of hand gestures.

4.6.3.1 Landmark Classifier

Going from image to predicted gesture involved a few key steps:

- Develop a classifier that takes landmarks (the output from the MP model) as input, and predicts a gesture as output.
- Find and convert labeled datasets, to be used for evaluation and further improvement.
- Create code for generating performance metrics, to be used for evaluation and further improvements.

[3] `https://github.com/metalwhale/hand_tracking`

From our earlier experiments, we had a clear idea of what needed to be implemented to create a way to recognize gestures in video. The basic idea was to run the MP model frame-by-frame, and pipe the hand landmarks output to a classification algorithm (the landmark classifier).

In the research phase we had concluded that using accumulated angles in finger joints, as proposed in the MediaPipe paper (Zhang et al., 2020, p. 4), showed the most promising results[4]. Since we already had written some code for this method, most of the work consisted of refactoring it to a proper structure.

We also needed a way to evaluate the efficiency of our gesture detector. This was imperative for documenting that our approach really did work. Equally important, it would enable us to make and see effects of potential changes. Which, in turn, would enable us to make improvements to the gesture classifier later on.

First in order was finding appropriate data to evaluate against. Notably, the choice of dataset basically determined the type of gestures our ml-pipeline could classify. In an ideal world, we would design gestures that intuitively corresponded to their actual use case, and create and label data for these accordingly. For this scenario however, it was important to make a few pragmatic decisions, so we decided to use gestures that had readily available datasets.

After a bit of searching online, we found a few different labeled datasets of hand gestures. Most suited to our use was the Kinect leap dataset, used in two papers from Marin, Dominio, and Zanuttigh (2015b, 2015a). This decision was mainly based on that the images were similar to the use-case of our program, and that our classifier showed promise in discerning the gestures contained in the dataset.

[4]As opposed to distance between fingertips and palm.

Figure 6: Images from the Kinect Leap dataset.

We then created some simple programs and scripts for evaluation. Mainly this consisted of doing gesture recognition on the dataset, and then outputting performance metrics and a confusion matrix. Other important means of evaluation were also implemented, like supporting live annotations overlaid on video/images.

4.6.3.2 Gesture Recognition on a Video Stream

At this point our program could give frame-by-frame estimates for static hand gestures. Although these estimates proved quite good, issues arose when trying to use the detected gestures from the video feed to control functions. Humans tend to gesticulate when speaking, and generally move our hands about quite a lot. We needed to be able to differentiate normal hand movements from movement meant as user input, and also the ability to track gestures over time.

We implemented a solution for this problem that took time into account by creating a system that ran on every frame in the input video. Each frame it would look for every defined hand, get a gesture estimate for it, and temporarily save the estimate in a per hand queue. Entries in these queues would then either be a defined gesture or nothing. These queues then work as both a *continuity* and *intention* check. Solving both the problems with the estimates at once.

Instead of then naively using every estimate as user input, whenever a new gesture estimate was given, it would be compared to previous entries in the queue. With this system a gesture was only considered as intentional if a certain threshold of the other gestures in the queue also represents that gesture. This meant that one would need to hold a gesture a little while before it was functionally considered as user input. This effectively added some delay when using gesture controlled functions, but greatly diminished the number of unintentional triggered user functions.

4.6.3.3 Resulting ML module

We now had a working prototype of a machine learning module that could detect hand gestures in a real time, video feed setting. The underlying structure of this module proved effective, and is in fact kept pretty much intact in the final product.

4.6.4 Gesture Controlled Functionality

4.6.4.1 Zooming and Panning

The most sought-after feature, especially when using such a wide field of view camera as we are, is zooming and panning. We started with defining a gesture for zooming in and another one for zooming out. These would simply define a region of interest on the input image, crop out that region and scale it up to the input resolution. For panning we implemented a pointing gesture, that when used would move the region of interest to be centered around the index finger before cropping. This way, using said gestures one could quite easily move the frame around by simply pointing to where one wants it to be.

4.6.5 MVP — Result

At this point we had a functioning product that fulfilled the bare minimum of the specifications. It consisted of an ML module for gesture recognition, evaluation methods for said module, and some functionality controlled by gestures. We were fortunate in that the development of the MVP had gone faster than we anticipated, and moved on to the iterative phase.

4.7 Phase III: Iterative Development

4.7.1 Improvement in Module Architecture

Even though we went through planning and implementation of the program structure before finishing a working demo, all the additions in functionality and configurability in the main handler code was getting difficult to work with. Most of the smaller modules with well-defined tasks were clear enough, but the main code for handling the video streams and their processing required major changes.

It was clear that refactoring into a better structured codebase would be beneficial. We also agreed that this needed to be done as fast as possible. A program at this scale can easily get out of hand, and a well organized foundation would make our code more readable and easier to later work with.

A new program structure was then proposed with better defined modules[5]. Mainly a Stream Controller module, which handles all the input and output. And a Frame Processor that takes in an image, and just outputs the processed versions. This way the Stream Controller is basically just a video to video interface, allowing the Frame Processor to manipulate each image in the stream.

[5]The program structure implemented at this point closely resembles the one in the final product.

4.7.2 Alternative Way of Zooming

Now that a working version of zooming and panning was implemented, we also wanted to explore some alternatives. We settled on an alternative way of zooming by using both hands to define a region to crop. When both hands are showing and performing the given gesture, a point on each hand then defines opposing points in a rectangle. From these points a best fit rectangle with the same aspect ratio as the input image is created to avoid letterboxing. The same process of scaling as the other method of zooming is then applied.

4.7.3 Improving the Landmark Classifier

After implementing the initial version of the landmark classifier, we were now looking to improve it. The first step of this process was evaluating the current version. This was done in two ways. The first consisted of running the ml-pipeline on the evaluation dataset, to get statistical information on performance. The other was a more informal approach, which consisted of overlaying annotations on live video to get a feel for performance.

When evaluating against the static image dataset, there were different sources of misclassification. Without going too much into specifics: Some of them were due to the MP model not correctly predicting hand landmarks, and some of them were due to the landmark classifier misclassifying correctly inferred landmarks [6].

In this case, we were only concerned with evaluating and improving the *landmark classifier*. To do this we first filtered out every image where MediaPipe could not detect any hands. Then, we wrote a script to mark any images with faulty hand landmark projections, so they could be ignored when evaluating the classifier. At the end we were left with a pruned dataset that only consisted of correct landmark predictions.

[6] More details on this and the full evaluation method, will be covered in the results segment

The results of evaluation on the pruned Kinect Leap dataset can be seen in the confusion matrix of figure 7 and evaluation table that follows.

Note: Confusion matrices are a common way of evaluating classification models.[7] As such they are often associated and used with deep learning methods of classification. However, in *this* section, the target of evaluation is the landmark classifier, which is *not* a machine learning model, but an algorithm that uses accumulated angles to map a state of open or closed to each finger.

[7]For readers who are not familiar with confusion matrices and related performance metrics, this article provides a decent overview

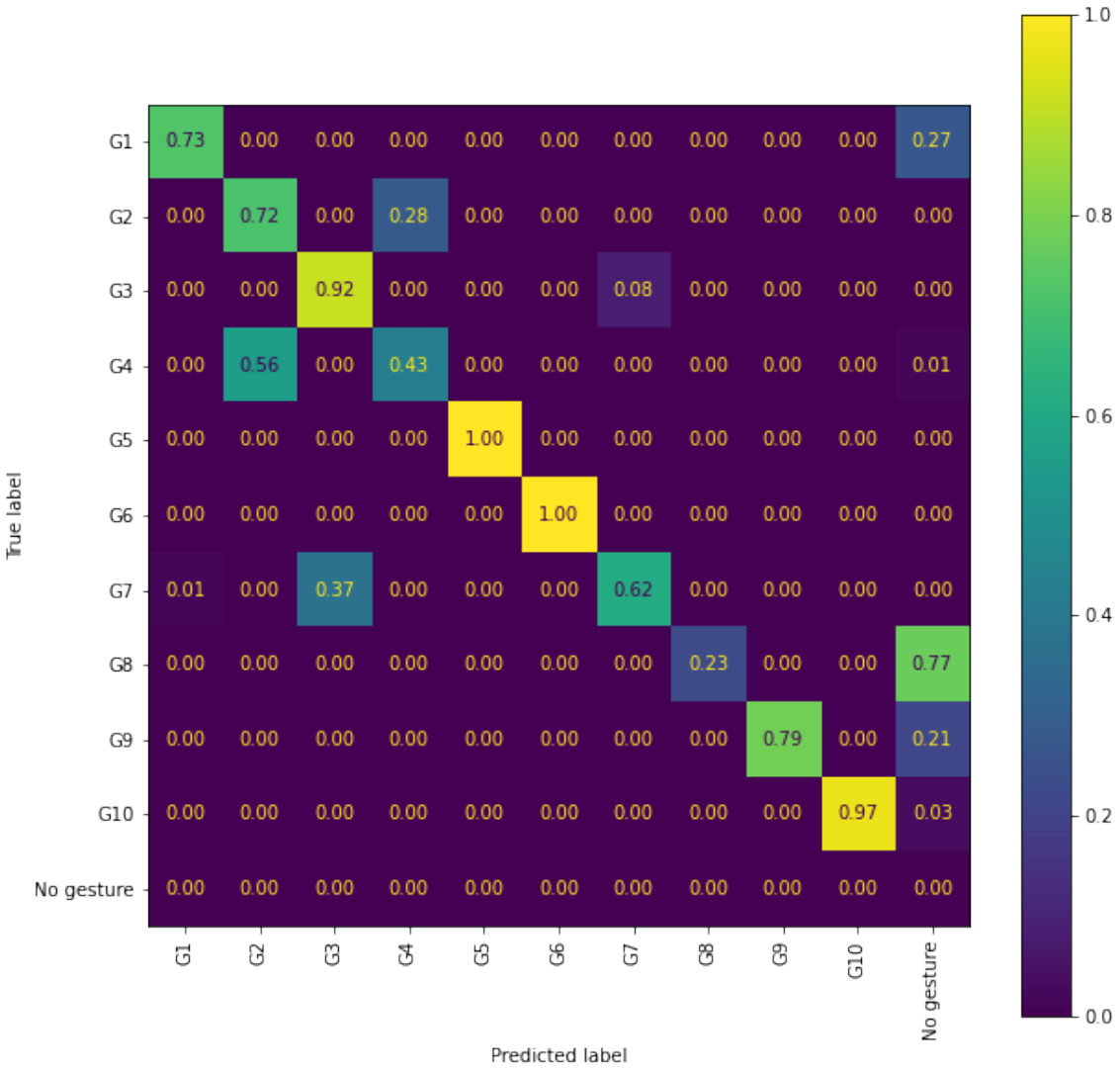


Figure 7: Resulting confusion matrix from the first version of the landmark classifier. The classifier has been run on a pruned version of the Kinect dataset, which only contains correct landmark projections.

Table 1: Evaluation table for landmark classifier v1 (first version)[8]

Class	Precision	Recall	F1-score	Support
G1	0.98	0.73	0.84	77.00
G2	0.50	0.72	0.59	60.00
G3	0.72	0.92	0.81	83.00
G4	0.66	0.43	0.52	77.00
G5	1.00	1.00	1.00	63.00
G6	1.00	1.00	1.00	58.00
G7	0.87	0.62	0.72	78.00
G8	1.00	0.23	0.37	70.00
G9	1.00	0.79	0.88	80.00
G10	1.00	0.97	0.98	64.00
No gesture	0.00	0.00	0.00	0.00
accuracy			0.73	710
macro avg	0.79	0.67	0.70	710.00
weighted avg	0.87	0.73	0.77	710.00

Looking at the results of the static image evaluation combined with live evaluation through overlaid annotations on video, we identified a major source of inaccuracy: Our classifier was struggling to estimate whether the thumb was open or closed. An example of this can be seen in the confusion matrix in figure 7, where gesture G4 and G2 have been mistaken for one another numerous times. In the approach of our classifier, the only thing separating these two gestures was the thumb either being open or closed. An example of this mislabeling can be seen in figure 8.

The next step was figuring out why thumb openness frequently was mislabeled. Considering the anatomy of the thumb and its landmarks, it was apparent that it

[8]These tables have been generated with the help of the scikit learn metrics library. For an explanation of the terms 'Precision', 'Recall' and 'F1-score', we suggest this article. 'Support' refers to how many data points (i.e. images) each row is based on.

differed from the other fingers in a few ways. For one, when the thumb is open, the landmarks generally does *not* form a straight line, as with the other fingers. Another important distinction was the angles formed between the thumb joints. For a normal finger, these angles are generally only orientated towards the wrist. The thumb however, can often be in a hyper-extended position, where some of the joint-angles are directed away from the palm.

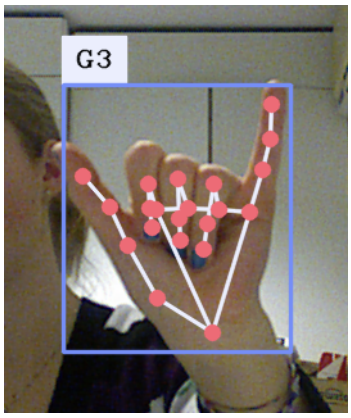


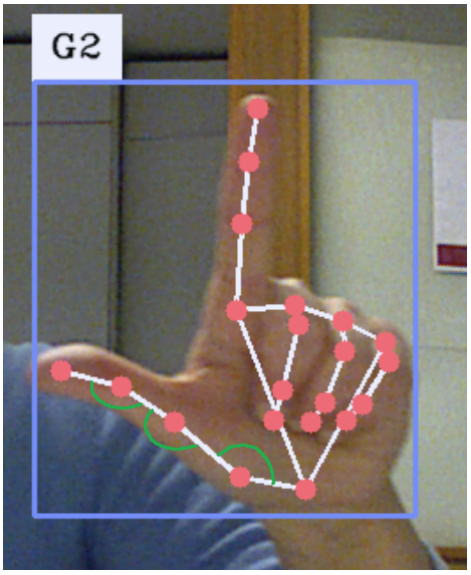
Figure 8: An example of misclassification: G7 has been mislabeled as G3, due to the thumb being incorrectly assessed as 'Closed'.

We saw two main strategies for dealing with the problems. The first was to discard the angle-based approach for estimating thumb openness, and simply look at distance from thumb tip to palm. Approach number two involved refining the way we calculated angles for the thumb, by introducing positive and negative angles.

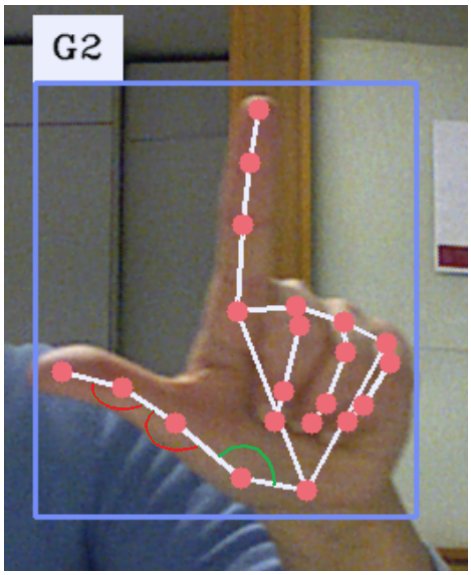
Of these, the distance based approach was by far the easiest to implement. We therefore started with some experiments to see if it had any merit. It did not take long to find out that this approach did not really lead to any better performance, so we decided to try the second alternative.

4.7.3.1 Classifying Thumb Openness Through Signed Angles

The goal of this approach was to distinguish between the angles in the thumb joints that were 1. Orientated towards the palm and 2. Orientated away from the palm. An illustration of this can be seen in figure 9.



(a) Thumb angles calculated with the first landmark classifier. Green color indicates positive values. All angles here are considered positive, regardless if they are orientated towards, or away from the palm.



(b) The desired way of calculating thumb joint angles. Angles orientated away from the palm are considered negative (red), while angles towards the palm are considered positive (green).

Figure 9: Illustrations of thumb angles calculated with the first landmark classifier, and the desired outcome of using positive / negative angles.

To correctly identify these angles, we needed some kind of a reference for the orientation. In short, we needed a way to estimate which way the palm was facing, and to use that to create a palm normal vector, which could be used as an orientation for the thumb joint angles. After trying some different solutions, the best results were obtained from using fixed landmarks on the palm to create this normal vector. An illustration of the approximated vector can be seen in figure 10.

With this palm normal approximation, we changed the way thumb joint angles were calculated to consider both positive and negative angles, as previously described. Furthermore we had to set a separate threshold for the accumulated thumb joint angles. Evaluating this new version of the classifier yielded the following results.

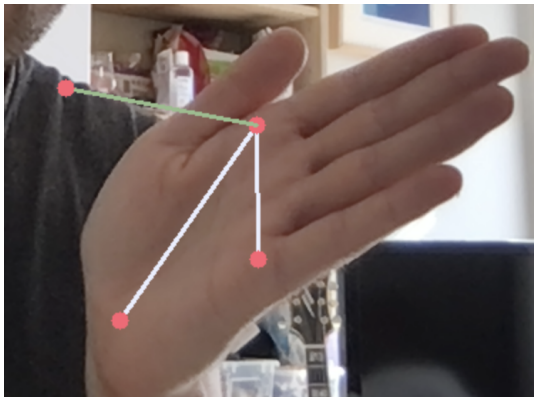


Figure 10: The palm normal vector calculated from hand landmarks. The white lines indicate the vectors which were used to approximate a normal vector for the palm. The green line is the resulting palm normal vector, projected from the index mcp joint.

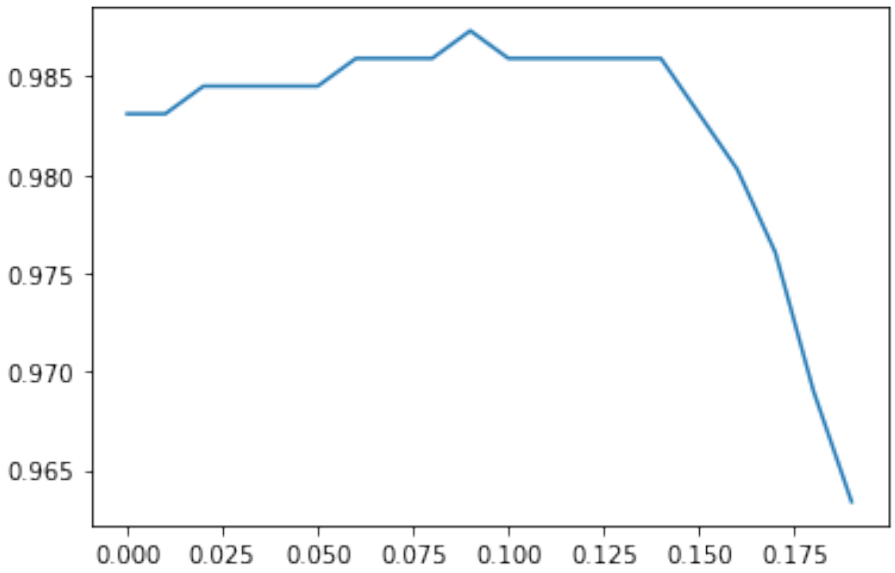


Figure 11: Tuning the thumb angle threshold. The x-axis is the accumulated thumb angle threshold in radians, y-axis is overall accuracy on the evaluation dataset.

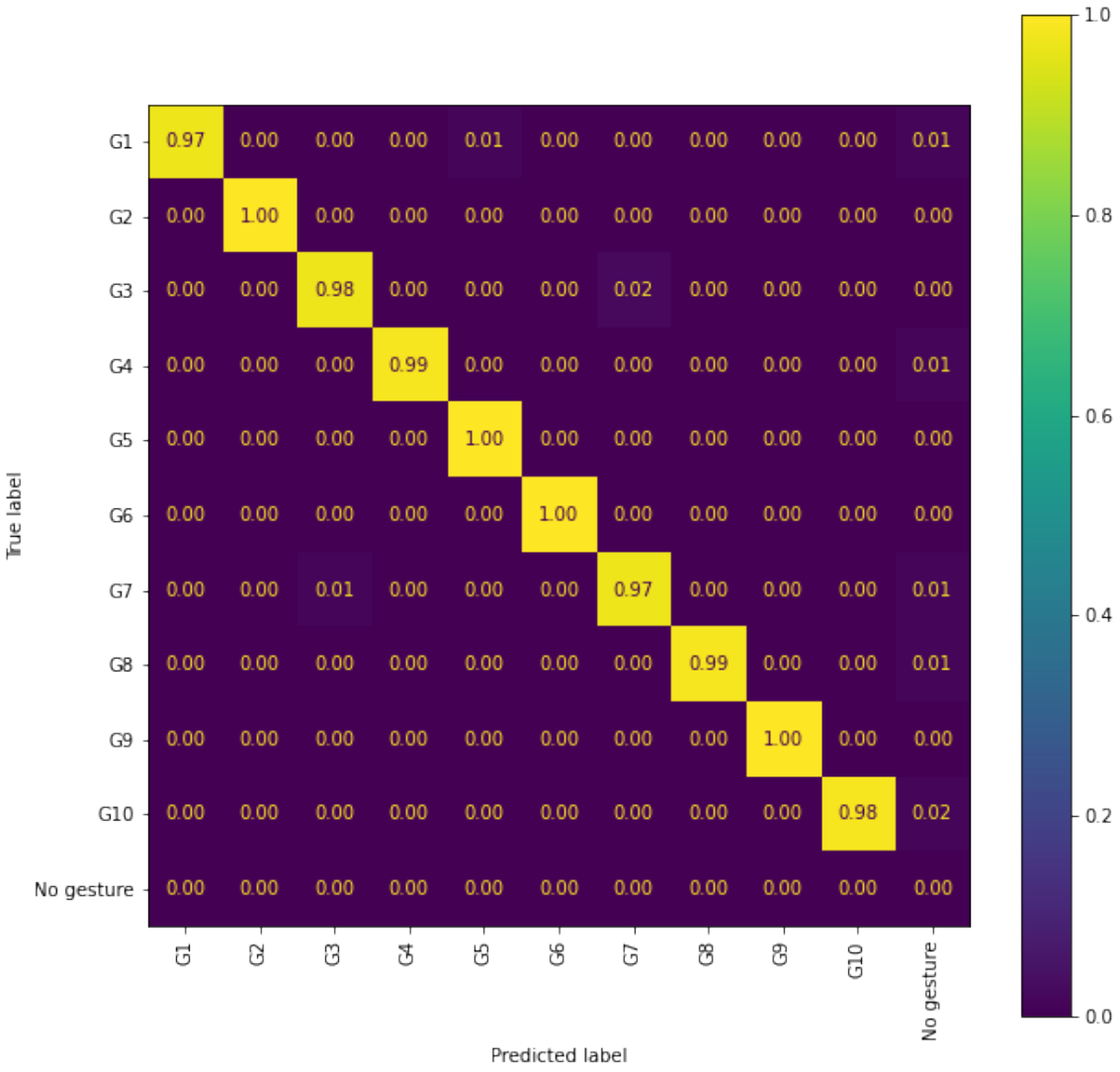


Figure 12: Resulting confusion matrix from the second version of the landmark classifier. The classifier has been run on landmarks generated from the evaluation dataset.

Comparing the evaluation results from the first and second version of the landmark classifier shows pretty dramatic improvements. Overall accuracy (F1-score) was now at a whopping 99%. Testing the new version in the demonstration setting also

Table 2: Evaluation table for landmark classifier v2 with positive & negative angles for thumb joints.

Class	Precision	Recall	F1-score	Support
G1	1.00	0.97	0.99	77.00
G2	1.00	1.00	1.00	60.00
G3	0.99	0.98	0.98	83.00
G4	1.00	0.99	0.99	77.00
G5	0.98	1.00	0.99	63.00
G6	1.00	1.00	1.00	58.00
G7	0.97	0.97	0.97	78.00
G8	1.00	0.99	0.99	70.00
G9	1.00	1.00	1.00	80.00
G10	1.00	0.98	0.99	64.00
No gesture	0.00	0.00	0.00	0.00
accuracy			0.99	710
macro avg	0.90	0.90	0.90	710.00
weighted avg	0.99	0.99	0.99	710.00

confirmed that it performed much better in a live video context. That being said, the evaluation results from the pruned dataset did feel somewhat generous when compared to perceived performance during live testing. It should also be noted that the size of the dataset was basically halved (from 1400 images to 710) after pruning for correct landmarks. Still, these results show that given the correct landmarks, the classifier was doing an excellent job. This meant that when evaluating the pipeline of

$$\text{image} \rightarrow \text{MP model} \rightarrow \text{hand landmarks} \rightarrow \text{landmark classifier} \rightarrow \text{predicted gesture}$$

misclassification stemmed from the MediaPipe model, not the landmark classifier.

4.7.4 Webcam Emulation

Our clients camera features are mostly implemented on the camera itself, however in this project which focus mostly on research and demonstration of possible features

they are not. We have implemented everything running outside the camera, on the computer itself.

We therefore at this point only output the video feed and optionally the annotations to either a collection of images, to a video file or to a video feed in a separate window. This worked well for the development and testing of features, but it would have been great to be able to demonstrate the program features in a real setting.

To do this, we wanted to be able to use the video output as a regular webcam. Thankfully certain operative systems make it quite easy to set up a virtual webcam interface, for which we implemented a system that streams the video output to that interface, emulating a webcam. This allowed us to use the program's output as if it was a regular USB-connected webcam on any video communication platform.

4.7.5 Iterative Development — Finishing Phase

We were now at a point where the product was satisfactory to our supervisors, and fulfilled the product specifications. It was natural that we delegated more effort into the project report as time went on, but we still had some capacity for further development. Meeting with our Huddly supervisors, they gave us suggestions for features to look into. Ultimately though, we were given free rein to experiment with whatever seemed most interesting, and ended up looking into a few different areas.

4.7.6 Bézier Interpolation for Smoother User Controlled Motion

While zooming the camera through gestures essentially worked as intended, it did feel somewhat unnatural. We were tipped by our supervisors that this might stem from the fact that the zoom was applied linearly with time, and that implementing some acceleration might create a better user experience.

With no idea which acceleration curves would yield the best feel, we wanted to

implement a system for this in which we easily could configure different curves. Bézier curves have been proven to be perfect for this, and have been used widely in computer graphics ever since the beginning of graphical aided design (Hazewinkel, 1997, p. 119).

Basing our implementation on Bézier curves would allow us to support all sorts of zooming speed variations, and enable an easy way of testing our way to an optimal smoothing curve. We can achieve full control over a acceleration by using a third degree Bézier curves. Interpolating on such curves involves solving for the roots of a cubic polynomial. Through research we found that most implementations of this interpolation was done numerically, most notably and widespread, CSS' implementation that uses Newton-Raphson iteration (*Mozilla SMIL*, 2011). We can prove that in our case it is trivial to prove that the polynomials are solvable analytically [9].

With this knowledge we implemented a cubic polynomial solver that would find the roots of any quadratic by first depressing it into a general cubic polynomial and applying either Cardanos's or the trigonometric cubic formula (Zucker, 2016). An interpolator was then implemented which would be defined by the intermediate control points, and could interpolate given linear zoom-values onto the curve by using the cubic solver. This in theory gave us infinitely fine control over the zoom speed over time, and would also allow for easing in and out any other movement in the program.

We unfortunately would only be able to ease in zooming, as easing out requires knowing in advance when the user stops showing a gesture, which is practically impossible. We ended up testing some different curves and found a favorite that would slowly accelerate the zooming over a span of about one and a half seconds. Due to the problems of not knowing when to stop, we had no other choice than to immediately stop zooming whenever the user stops showing the gesture. This was originally thought of as a problem, but through functional testing was shown to be very usable, and as an afterthought maybe the most intuitive.

[9] A full proof and elaboration of the analytic solution at page 92.

4.7.7 Dynamic Gestures

The gestures we had been working with so far were all static, meaning they were not contextualized in time or place, and could be represented by a single image. At our supervisor's suggestion, we decided to look into recognizing dynamic gestures, as this could broaden the program application.

With dynamic gestures posing a significantly more challenging problem, we decided to start with the simplest sub-problem we could think of: Hand waving.

Our attempted solution consisted of creating a queue consisting of hand landmarks from the latest 20-30 frames, amounting to about a second of real time. By reducing the coordinates for the hand landmarks to a single point on the hand, we then naively checked the distance between the greatest and smallest x value in the queue. If this distance was above a certain threshold, we classified it as a wave. Determining the orientation (left/right) of the wave was then a simple case of identifying the queue positions of the greatest/smallest x-values.

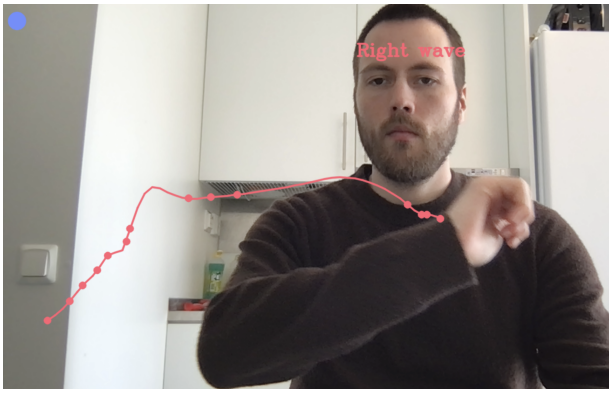


Figure 13: An example of how the program responds to a recognized hand wave. The red dots are the actual points from previous frames that were used to infer a hand wave. The red curve passing through them is the result of a quadratic spline interpolation through said points.

This approach yielded a very primitive wave recognizer. While it could do detection

in some simple cases, the main takeaway was the disadvantages this method had for dynamic gesture recognition. First of all, the amount of frames for which a hand had been detected were proportionally low. Only about 10 to 40 percent did contain a successfully detected hand. Furthermore, for hand waves to be recognized, they had to be performed at a slow pace. This was due to the fact that a quickly moving hand would not yield many frame references and would cause problems for the MediaPipe hand detection model.

Perhaps the biggest issue was finding a general way to define dynamic gestures. Only looking at horizontal distance was a very limited approach, and difficult to quantify into a greater set of dynamic gestures. We thought about defining dynamic gestures by way of mathematical curves, and matching the data points from the queue against them. However, the project was soon reaching its end, and we decided to suspend the dynamic gesture experiments.

4.7.8 Investigating Possible Performance Optimizations

Program performance is usually a desired metric, and it was particularly relevant considering Huddly frequently implements ML solutions on light weight hardware. One of the things we wanted to investigate was potential performance improvements of using GPU powered partial parallel computing.

After a fair bit of configuration, we were able to run tests investigating these benefits over some different hardware. From these results we were also able to map out a time-profile of function calls.

The overall conclusion from this was that to really utilize possible benefits of running on GPU, the program would most likely have to undergo dramatic changes. The profile of function calls also revealed that our own modules did not have any serious bottlenecks, and as such did not merit further optimization. An in depth description of these testing methods and results is available in the Test documentation

on page 65.

4.7.9 User Testing

By this point we were applying finishing touches. It was important that we delivered a product which, together with its user manual, was easy to use, install and understand. The logical way to ensure this was by way of user testing. We wrote the first draft to a user manual, and had a few users test the program. They then gave feedback by way of a google form.

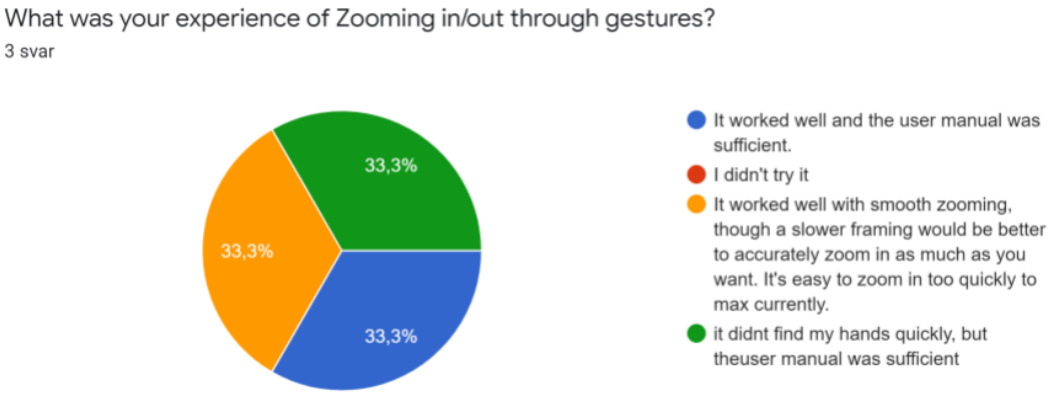


Figure 14: Answers to one of the questions from the user testing form.

The results of the user testing were positive, with a few suggestions for both the user manual and product itself. The feedback resulted in adding a gesture for resetting the program, improving the user manual, and some other minor changes.

4.8 Results

4.8.1 Gesture Recognition Module

The machine learning module for gesture recognition in the finished product consists of three parts. 1. The MediaPipe model, 2. The hand landmark classifier, and 3. The intent binary classifier.[10]

As mentioned, statistical evaluation was configured for segment 1. and 2. of this pipeline. For the classification of user intent, evaluation consisted of live testing with the product.

4.8.1.1 Dataset

The following results are based on the Kinect Leap dataset, sourced from Marin et al. (2015b, 2015a). It consists of 14 participants performing 10 different gestures. Each participant performs each gesture a total of 10 times, amounting to 1400 images. The participants are made up of two perceived genders, with some variance in skin tone.

4.8.1.2 Method

Evaluation of gesture recognition and classification has been performed on these static images in the following way.

First the MP model is used to detect hands with corresponding hand landmarks on the image. For the MP model configuration, we have used settings that correspond to the ones most similar and/or prevalent to the ones utilized in our program. Specifically

[10] An in depth explanation of the ML module is available in the product documentation (page 74).

this involves a minimum confidence value of 0.5, and max number of hands to be set to 1.[11]

The hand landmarks from the MP model are then passed on to the landmark classifier, which uses them to predict a gesture.

The possible predictions consists of 12 discrete classes. 10 of them are the gestures defined in the dataset. The rest are two classes representing negatives: 'No hands' corresponding to MP model not detecting any hands on the image, and 'No gesture' corresponding to the landmark classifier predicting a set of open/closed fingers that are not part of the defined gestures[12].

[11](see the MediaPipe Hands page for reference).

[12]The landmark classifier predicts a gesture based on an open/closed state calculated for each finger. Some of these possible configurations are not part of the defined gestures. For instance: A closed thumb, but all other fingers open will be predicted as "coooo" by the landmark classifier. For more on this, see the GestureClassifier section of the product documentation.

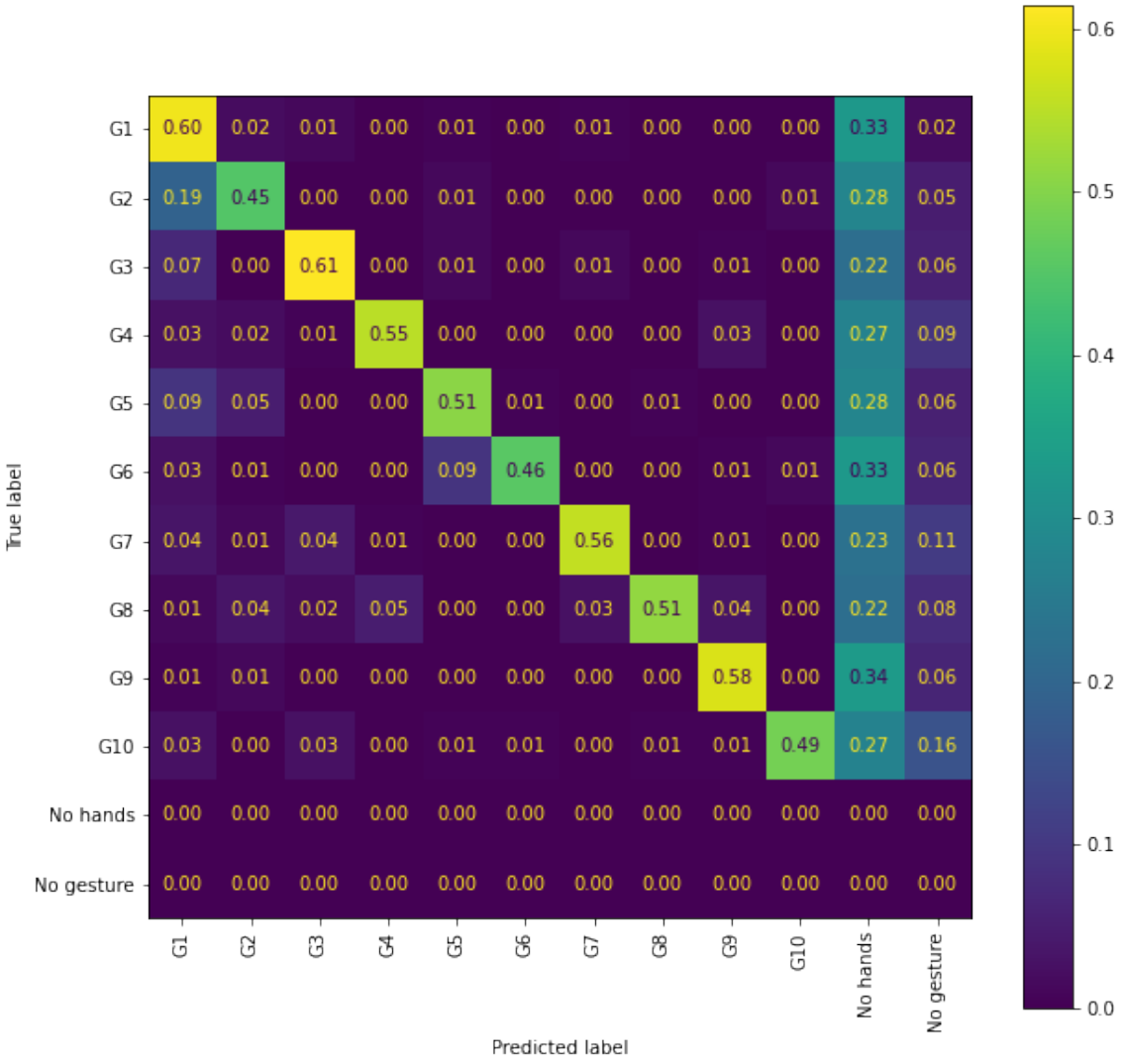


Figure 15: Resulting confusion matrix from the conjoined MP model and landmark classifier, evaluated against the Kinect Leap dataset.

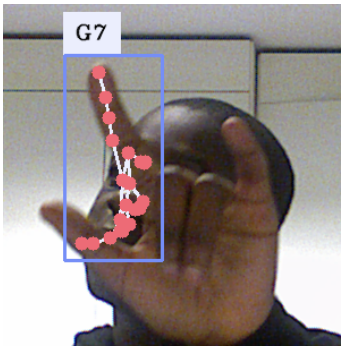
There are three sources of misclassification:

1. The MediaPipe model did not detect any hands in the image, resulting in the output "No hands".

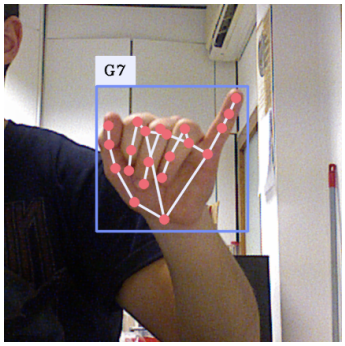
2. The MediaPipe model detected hands in the image, but the inferred landmarks were incorrect. As such the landmarks passed to the landmark classifier would not be of the actual gesture in the image.

3. The MediaPipe model detected hands and inferred the correct landmarks on the image, but the landmark classifier gave the wrong gesture class.

(a) Case 1, no hands detected.



(b) Case 2, incorrectly inferred landmarks.



(c) Case 3, misclassification of correct landmarks (G3 has been mislabeled as G7).

Figure 16: Sources of misclassification.

As shown previously,[13] the landmark classifier has a very high accuracy given correct input data. From this we can conclude that only case 1 and 2 are legitimate sources of misclassification. Consequently, to pursue further improvements one would have to improve the MP model itself.

As an alternative, it may be possible to change the landmark classification algorithm to be more forgiving of the input data it takes. As this algorithm works now, the output it produces is categorical: Each finger is either considered to be 'Open' or 'Closed'. If this instead was changed to a probability of finger openness, the classifier can produce a probabilistic output (that is, a probability distribution for the gesture set). A classifier of this type would offer a higher degree of flexibility.

[13] See the results of the improved landmark classifier (page 42.)

One idea is to observe the MP model output, to see if there are any tendencies for misprojections of landmarks. These could then be given less weight in the classification algorithm.

Table 3: Evaluation table for the conjoined MP model and landmark classifier, evaluated against the Kinect Leap dataset.

Class	Precision	Recall	F1-score	Support
G1	0.55	0.60	0.57	140.00
G2	0.73	0.45	0.56	140.00
G3	0.84	0.61	0.71	140.00
G4	0.91	0.55	0.68	140.00
G5	0.79	0.51	0.62	140.00
G6	0.97	0.46	0.62	140.00
G7	0.92	0.56	0.69	140.00
G8	0.97	0.51	0.67	140.00
G9	0.86	0.58	0.69	140.00
G10	0.94	0.49	0.64	140.00
No hands	0.00	0.00	0.00	0.00
No gesture	0.00	0.00	0.00	0.00
accuracy			0.53	1400
macro avg	0.71	0.44	0.54	1400.00
weighted avg	0.85	0.53	0.65	1400.00

4.8.1.3 Conceived Performance of Full Module

When evaluating the machine learning module as a whole, the only metric consists of user interaction with the finished product. From our own experience, as well as feedback gained from user testing, the overall impression is that gesture recognition works well. This is especially true if the user adheres to the guidelines of the user

manual[14], such as having one's palms face the camera.

4.8.1.4 Flaws and Possible Improvements

Gesture recognition works best when the max number of hands of the MP model configuration is set to 1. As all but one of our gesture controlled functionalities only require one hand to be operated, this is not a substantial problem in and of itself. That being said, it reveals one flaw of the ML module, namely the absence of a tracking solution. The way the MP model gives it's output, there is no tracking id related to each set of hand landmarks. Consequently, when put into our ML pipeline, this sometimes leads to hands being 'mixed up' with one another. As a result, if there are multiple hands present in a video setting, our ML module can get confused and jump between them.

We were a little bit late to reflect on these challenges. As such the final product is recommended to be run with a max hands setting of 1. Had we thought of this problem earlier in the process, it would be interesting to try to implement some way of tracking each hand. In addition to being a fascinating problem in and of itself, this would probably lead to better performance when using multiple hands.

4.9 Conclusion

4.9.1 Self-Evaluation

The final product is one that satisfies both the specifications and intent of the project. We believe it is an effective demonstration of the possibilities that lie in using machine learning powered gesture recognition, for interactive functionalities.

[14]See the user manual appendix on page 85

One challenge was to find interactive uses of gestures that actually were useful, and not just gimmicky features. While the usefulness of all of the different functionalities in the finished product can be debated, at least some of them felt truly beneficial in a video conference setting. Take for instance zooming and panning the camera through gestures, which both were frequently used by ourselves during several video meetings.

4.9.2 Educational Value

We consider ourselves very fortunate in having the privilege to work with such an interesting problem. As software engineering students the project has been particularly rewarding, due to the fact that we were able to use not only our development experience, but also our knowledge of mathematics and statistics.

As a stand-alone proof-of-concept product, our development process has been entirely separate from any other systems. This has enabled a fast and flexible development process, with room for lots of experimentation. The downside is that we did not get to experience the development process in a larger enterprise, with integration to other systems and a more formal development structure. However, such a process would require significantly more follow-up from our Huddly supervisors. This is particularly true when considering the circumstances surrounding COVID-19, which has forced us to work remotely for the entire project. An integrated development process would therefore inevitably have a negative impact on progress and flexibility. Our assessment is that this trade-off would probably not have been worth it.

In the end of the research phase, a decision was made to focus on the demonstration program, and disregarding developing our own ML model. This led to us being able to produce a demo that showcases what we believe are interesting features, that feels relevant to the intent of the project. In an ideal world, we would have liked to spend more time on development of our own ML model, to educate ourselves deeper into the field of machine learning. Still, we feel that this decision was correct. If we had decided on training and configuring our own model, this would likely have taken over

as the main focus of the project. The demonstration would then certainly be much simpler, and probably not as interesting for Huddly.

4.9.3 Client Impact

At the time of writing, the feedback we have received from the Huddly supervisors, and a few other select employees, have been positive. They feel, like us, that the finished product has achieved what we were set out to do: Provide a practical demonstration of gesture controlled functionality for a conference setting.

Concretely, they mentioned that the product demonstrated a "more interactive video meeting experience", and that it will hopefully inspire further research into the domain. To that end, we are scheduled to perform a company wide demonstration in the beginning of June.

When it comes to development, the road from inception to production is long. If Huddly are to pursue this concept further, they will most certainly start from scratch and create their own implementation. After all, a production ready application will have significantly more requirements and obstacles to overcome. Nevertheless, the intent of this project was to investigate and demonstrate the possibilities of this technology. The feedback so far indicates that this has been accomplished, and that the product will stimulate and encourage further innovation in the area.

5 Product Specification

5.1 Background

Huddly is a technology company that creates tools for team collaboration. By combining hardware, software and AI to create intelligent vision products, they offer innovative camera solutions.

With machine learning being a rapidly developing field, Huddly are always on the lookout for new ways to utilize these technological advancements to enhance user experience. The recognition of hand gestures is a related topic that has been widely researched for many years. At this point, Huddly is interested in the possibilities that lie in utilizing recent advancements in this field, to recognize and enable gesture-based functionality for their cameras. The main goal of the specified product should therefore be to document and demonstrate the use of machine learning based hand gesture recognition, to power relevant functionalities for remote collaboration through camera.

5.2 Preface

The software requirements have been developed with the purpose of guiding the development process. As with most IT development projects, the product specifications have undergone numerous changes throughout the process. This is only natural considering the iterative work methodology our group has adhered to.

Most notable is perhaps the shift in focus that happened at the end of the research phase. As explained in the process documentation, this involved that the product specification was altered quite significantly from the one featured in our pre-project

report. The updated specification received at this time[15] would form the basis of product development, particularly in regards to developing an MVP. As such we have chosen to use this updated version as the basis of the document, and forgo most of the specifications from the pre-project report.

5.3 Document Structure

The software requirements are made up of three parts. The first two, functional and non-functional requirements, make up what were the initial specifications for the (minimum-viable-)product. The third section elaborates on changes and additions that were made during the iterative development phase.

5.4 Functional Requirements

5.4.1 Goal Overview

Identify and classify gesture recognition using existing deep learning models and machine learning techniques. Use results and develop framework software to communicate with Huddly IQ[16] interactively.

5.4.2 Detailed Description

The students should explore different model types and architectures on a high level to familiarise with the basic foundation of what is demanded of a machine learning software product. Huddly recommends frameworks such as Keras, Tensorflow (TF) or other open-source pre-trained and existing models built on these. It is not expected

[15] See the 'End of research' section on page 25

[16] The Huddly IQ is a 150-degree wide angle camera with AI capabilities.

that the students need deep technical knowledge on deep learning architecture. A rudimentary and fundamental understanding on choosing and working with models in software projects built on machine learning is however expected.

The solution should comprise three parts. Deep learning module, python end to end library and a Command Line Interface (CLI) module.

5.4.3 Deep Learning Module

The students should find an existing model framework designed to classify and recognize hand gestures, or modify existing models to fit their purpose. Eg. PoseNet can be used as a basis for a model that focuses on hand and gesture recognition.

5.4.4 Python e2e Library

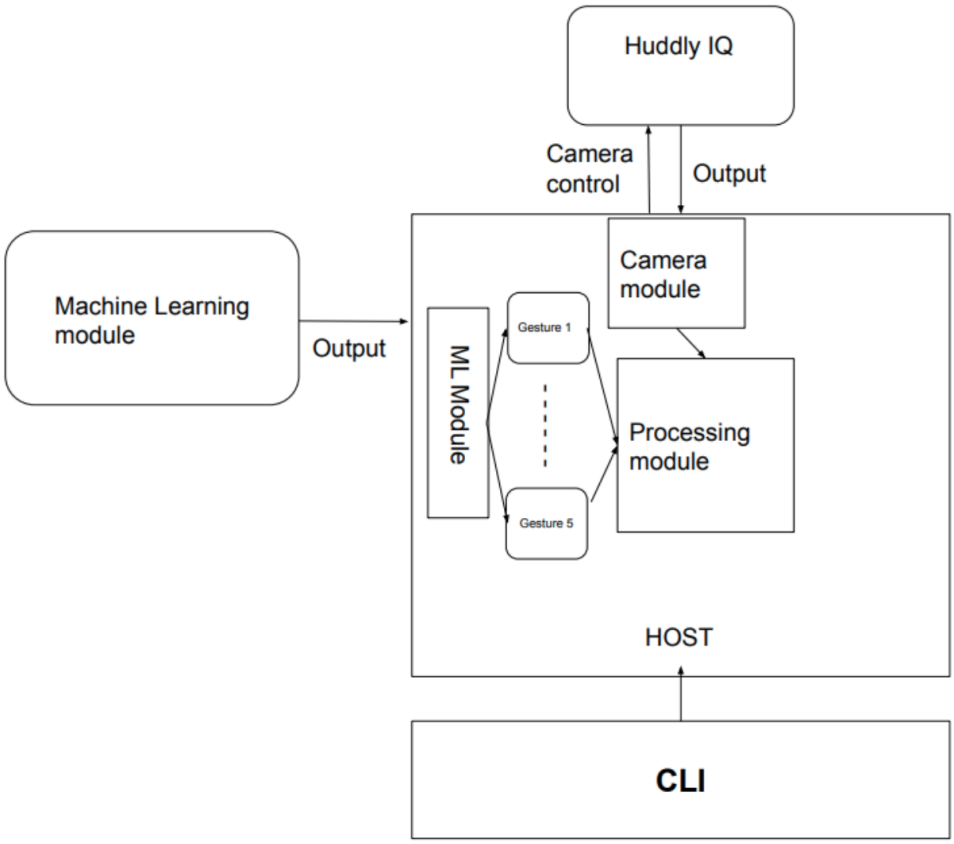
Camera platform software written in an appropriate programming language (Python 3.5 + is recommended) to support an end to end solution. This will require a framework for reading output of the machine learning model, camera software to communicate with Huddly IQ and a processing library to match and label camera output with model output. The machine learning framework should provide scalable properties and a structured basis for handling data. The camera software should be able initiate and stop camera sessions, fetch single and multiple images and sample data correctly. The processing library should be able to handle camera/image data and model output data to synchronize and align the two endpoints.

5.4.5 Command Line Interface (CLI)

The system can be controlled using a CLI. This module can be created in either bash or python. The module can support tasks such as starting/stopping sessions and save

camera output with and without bounding boxes.

5.4.6 Schematics



5.5 Non-Functional Requirements

The main intention of the product is to investigate and demonstrate the use of machine learning to power gesture-based functionalities in a conference setting. As such the product should be developed as a stand-alone proof-of-concept. Typical

non-functional conditions, such as safety, interoperability, elasticity, etc., are therefore not present.

5.5.1 Deep Learning Module

The performance of the deep learning module should be documented in a way that corresponds with basic academic standards. Consequently, the finished product should have reports on how the module performs in regards to gesture recognition. Evaluation should be performed on relevant datasets, and produce relevant performance metrics such as confusion matrices, accuracy, precision, recall, etc.

The final product demonstration should predominantly consist of gestures in which the performance of the deep learning module has been documented in this manner.

5.5.2 Useability

The product should be designed for users with basic programming and software development experience. Users with this background should therefore be able to install and run the product, without complications.

5.6 Changes Made During the Iterative Process

As explained in the process documentation, our work methodology involved frequent deliveries and updated specifications of the product. Whereas the formal system requirements formed the basis of product development, this section aims to document changes and additions to the system requirements. These modifications stem from the iterative phase of development,[17] and were composed jointly between students

[17]See page 34.

and supervisors, during the weekly reviews.[18]

5.6.1 Modifications to Requirements

The initial software requirements were fairly broad by design, and the need for modification and/or removal of parts of these requirements were low. Nevertheless some notable changes were made.

- The functional requirements specify functionality for writing video output to file. This was deemed unnecessary and therefore disregarded.
- Support for operating on single/multiple images (as opposed to a video stream) is mentioned in the functional requirements. Some of this functionality does exist in various scripts and evaluation methods, but were considered not relevant for the demonstration part of the product. As such, this functionality was discarded from the end requirements.

5.6.2 Gesture Controlled Features

The following are live-video features intended to be initiated and controlled by hand gestures.

- Ability to enter / exit a gesture recognition camera state.
- Functionality for zooming in- and out from an intended point.
- Ability to pan a zoomed frame.
- Ability to fit the camera to a hand created frame.

[18] See page 15.

- Functionality for setting a dynamic value on a given range.
- Control camera brightness.
- Control sound input or output.

6 Test Documentation

6.1 User Testing

The main purpose of user testing was to ensure that intended users were able to install and run the program, using only the user manual as instructions. A secondary purpose was to identify possible improvements/changes that could and should be made to the program.

Testing was done in the following way: A user would be given access to the program code and the accompanying user manual. The user would then install and run the program using said instructions. Afterwards, they would answer a questionnaire about their experience.

The user tests were performed between 19.04 to 30.04, with different increments of the program and user manual. The test users consisted of four subjects. In accordance with the intended user specified in the product specifications, they all had, at minimum, a basic experience level with programming.

6.1.1 Results

The overall feedback was positive. Every participant had been able to install and run the program with relative ease. They were also enthusiastic about the program itself.

We received some suggestions for tweaks and changes to make the program easier to navigate, like creating a separate gesture for resetting the program session. The feedback also revealed that some of the functionalities were a bit confusing for the subjects.

For a full overview of the results, see the user testing appendix (page 80).

6.1.2 Impact

As a consequence of the test feedback, we added a few sections to the user manual. They were designated to help the users understand how to best position their hands for proper gesture detection, and give them a better understanding of how to best interact with the different features.

We also made tweaks that were suggested by the users, like lowering the speed with which the camera zoom operated.

6.2 Testing Support for Partial Parallel Computing Inference

Parallel processing power has been one of the main factors in accelerating machine learning into the mainstream, but up to this point we had only used serial computing to run our program. Mediapipe supplies two versions of every model, one optimized for serial run on Central Processing Unit (CPU), and one that enables parallel computing with Graphical Processing Unit (GPU) inference. Unfortunately the models with GPU inference require a lot of configuration and building both the tensorflow and mediapipe packages from scratch. This is therefore not something we would be able to officially support, but was nonetheless interesting to test the possible performance benefits.

We built the tensorflow package from source with cuda support, and changed some compile options in mediapipe to allow for GPU inference. The machine learning model itself also needed slight additions. To be able to run the original model on the GPU, we needed a layer in the model that converted the images from the python code into another color format, and furthermore into GPU buffers. The conversion had to take place on the CPU, the model could therefore only partially be run on the GPU while being compatible with the rest of our code. Once the model was altered

and both the packages were compiled correctly, the original code worked seamlessly with GPU inference.

6.3 Benchmarking and Investigating Bottlenecks

Even though we had been able to see the live frame rate when using the program, we wanted to gain some deeper insight into the performance of the program, especially now with parallel computing inference being supported. This would not only give us performance metrics across a few different devices, but would also let us see what part of the program was using the most processing power, which allowed for insight into possible bottlenecks and areas for potential optimization.

6.3.1 Performance Across Different Hardware

For this we created two scripts. Firstly one that would record the source video to file while using the program. We recorded two videos in which we tested out all supported functionality of the program in both (1280 x 720 pixels) (SD) and (1920 x 1080 pixels) (HD). Secondly we created a benchmarking script that ran the program on a few different devices with these recorded videos and outputted some metrics.

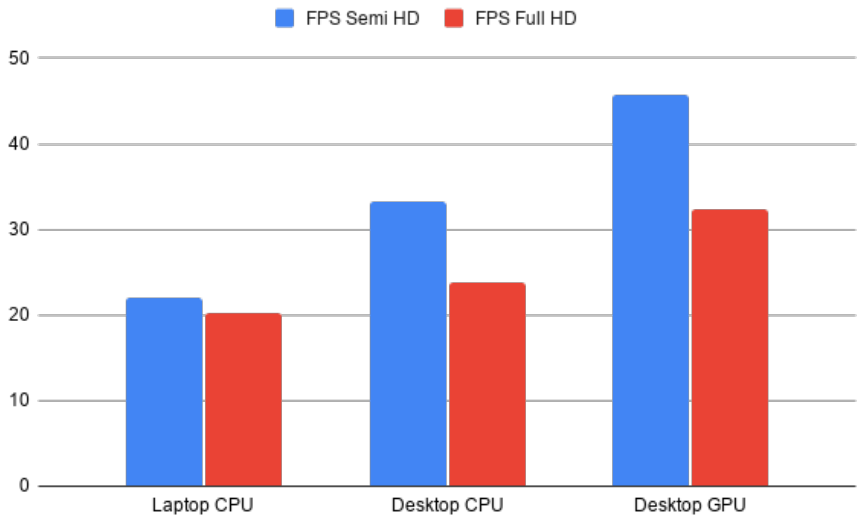


Figure 17: Average frame rate in frames per second across different hardware (Intel i5-8259U vs Intel i5-7500 vs Intel i5-7500 and Quadro RTX 4000) in (1280 x 720 pixels) (SD) and full (1920 x 1080 pixels) (HD)

Given that most webcams record at around 30 frames per second, the chart of figure 17 tells us that the program can comfortably run in real time on a decent desktop CPU in SD, and on a GPU in full HD.

The average frame rate on both a laptop and desktop CPU was as expected, and was similar to the frame rates we would see running it live. The benchmark run on CPU with GPU inference was however quite disappointing, and only showed marginal improvements. While running the benchmark, the GPU utilization never exceeded 20%. This told us that parallel processing power was not the greatest limiting factor for execution speed.

6.3.2 Finding the Bottlenecks

Python has inbuilt functionality that can count and time all function calls. By utilizing this on the benchmark, we could get an understanding of what was halting performance.

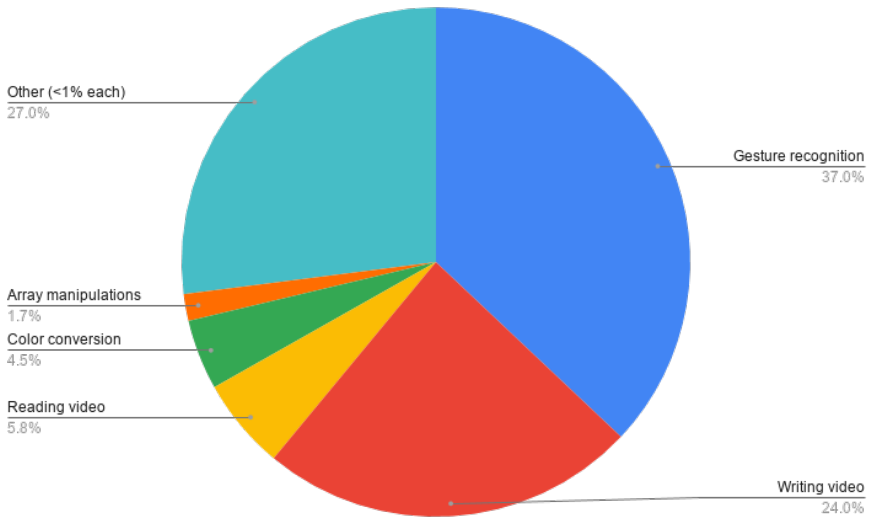


Figure 18: Fraction of runtime spent processing most processing intensive functions. Showing individually every function taking up over one percent of runtime, rest accumulated in others.

The above times are exaggerated due to the nature of timing functions this way, but the ratios between the times will remain the same. We see that while the mediapipe solution process is taking up the most processing time, cumulatively reading and writing the frames is in reality the greatest limiting factor. One would expect the situation to be different with GPU inference enabled, but we only see slightly faster time in the solution process. This is because the CPU now has to do the intensive task of converting each frame to a GPU buffer individually.

We can conclude that using GPU inference only improves performance by a

fraction, due to the model and the entire basis of our program treating the video feeds as a series of separate frames. Improvements could be made here by instead treating the whole or sections of the input video as a GPU buffer; this however, will require drastic changes to both our program and the mediapipe model. This is by far too big of a challenge to tackle at this stage in the project, but is still an interesting discovery nonetheless.

6.4 Unit Testing

The finished product contains a fairly brief amount of unit tests. Considering the scope of the project as a proof-of-concept, it was decided that spending time on extensive test coverage was unfruitful, since the finished code would not be subject to further development. Nevertheless, unit tests were written and used as a valuable tool for development. Particularly in cases where the desired output was easy to define in advance of the actual implementation, facilitating test driven development.

As is customary, for every potential change done to the repository, every unit test was run, and would have to pass for changes to be allowed.

A full documentation of every unit test would result in a long (and uninteresting) section, so we've instead decided to include some select examples:

```
from src.processing.cam_control import CamController
import numpy as np
import pytest


@pytest.fixture
def blank_image():
    image = np.zeros([100, 100, 3], dtype=np.uint8)
    image.fill(255)
    return image
```

```
def test_process_with_default_settings_should_not_change_image(blank_image):
    res = blank_image.shape[:2]
    cc = CamController(res)
    processed_img = cc.process(blank_image)
    np.testing.assert_array_equal(blank_image, processed_img)
```

This test is, as the name implies, for ensuring that the CamController.process() method does not mutate an image when the CamController settings are default values. It is useful for avoiding unintentional bugs when developing the CamController class.

```
from src.util import math_utils as mu
import pytest
import numpy as np

def test_normalized_to_pixel_coordinates():
    side_length = 100
    x0, y0 = mu.normalized_to_pixel_coordinates(
        0, 0, width=side_length, height=side_length
    )
    assert x0 == y0 == 0
    x1, y1 = mu.normalized_to_pixel_coordinates(
        0.5, 0.5, width=side_length, height=side_length
    )
    assert x1 == y1 == side_length // 2
```

This tests that the function for converting from normalized to pixel coordinates works as intended.

7 Product Documentation

7.1 Preface

The product documentation is a description of the proof-of-concept demonstration, it's capabilities, and program structure. It assumes the reader has basic programming knowledge. Some proficiency in math (linear algebra) is also beneficial, although this is mostly needed for the in-depth documentation of specific program methods/functions.

Note that evaluation of the gesture classification module is not part of this document. For this, we refer to the results section of the process documentation (page 49).

For the structure of this document, we've decided to first give a general overview of the product, with descriptions of the program architecture and flow, using text, diagrams and figures. For the particularly interested reader, the full project documentation is available as an appendix.

7.2 Program Description

This program is a proof-of-concept demonstration of using machine learning powered gesture recognition, to interactively engage with a camera and other systems in real time. It consists of a camera session, in which the user can engage with the session through hand gestures. Examples of this include camera manipulation such as zooming and panning, and also adjustment of other system settings such as microphone volume.

7.2.1 Adherence to Product Specifications

The end product is one that corresponds closely to the initial specifications. Most importantly, it fulfills the intent of demonstrating possibilities of ML powered gesture recognition, to engage with a camera session. Architecturally, the program is also similar to the initial schematics. For instance, the initial specifications make mention of an ML module and a processing module, which both are present in the final product.

7.3 Program Diagrams

7.3.1 Video Streaming

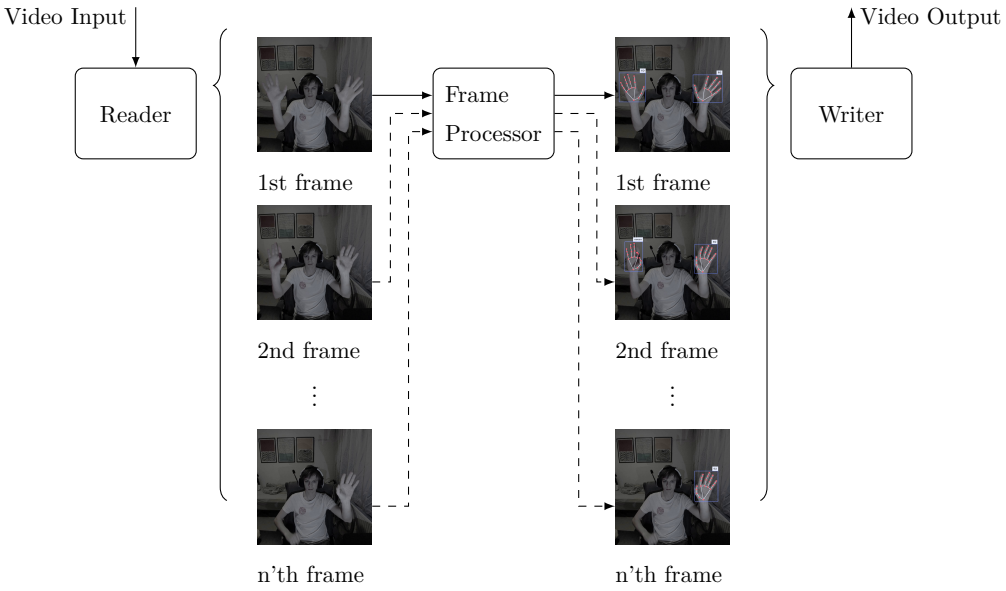


Figure 19: Program flow from video input to video output.

Video input in the form of either a video file or a webcam stream is split into individual frames by the reader. Frames are processed singularly by the frame processor and written to a video buffer by the video writer. Once a frame is written, a new one is read and processed, this happens in series because of the nature of live video. The writer can be any of the video writers, it can write to a window, to file or to a virtual webcam.

7.3.2 Recognition (ML Module)

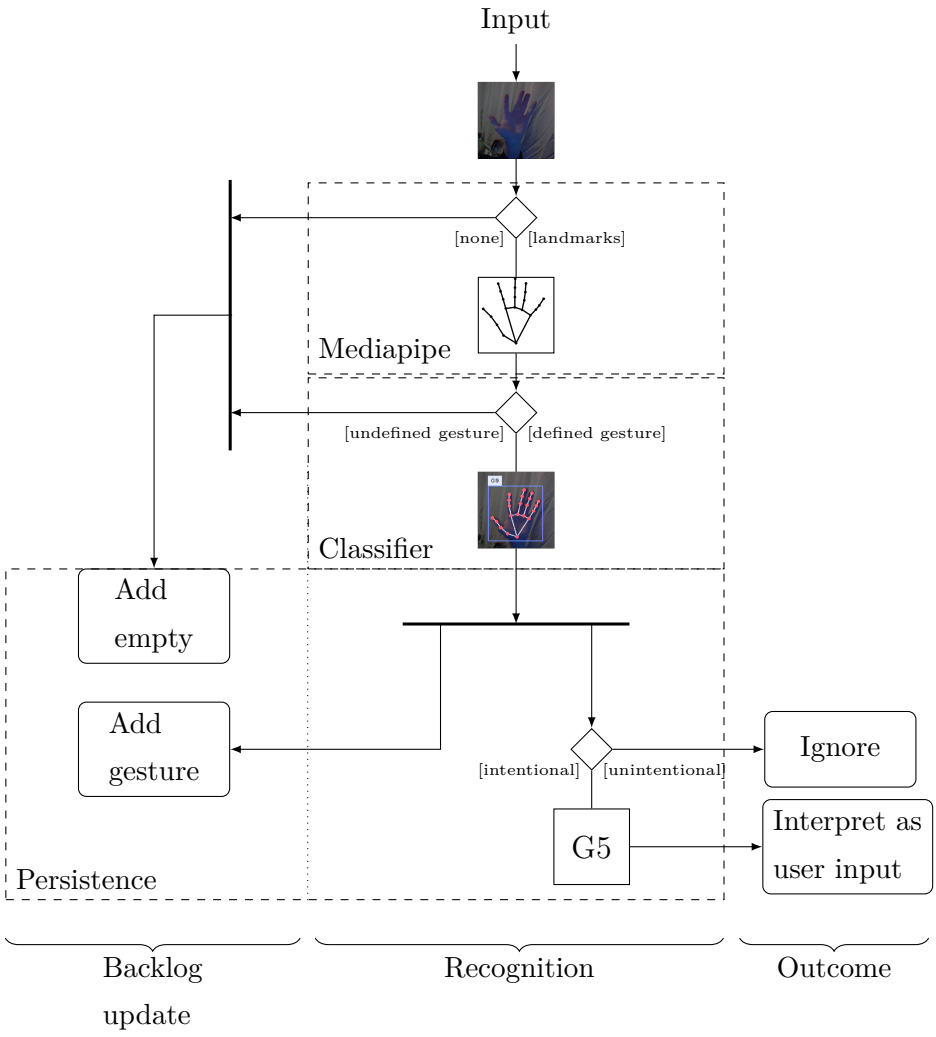


Figure 20: Illustration of the flow from input image to interpreting it as user input.

The diagram of figure 20 illustrates the flow from image input to interpreting it as user input (here with gesture 5, the open hand, as a graphical example). Three modules are responsible for the overall gesture recognition: MP hands, our gesture

classifier and persistence, the intent classifier. Persistence uses a backlog of the previous gestures to tell hand movements apart from intended user input. Crucially the backlog is also filled with non-gestures in the cases where no gesture can be recognized.

1. Firstly the image is fed to mediapipe hands, in which it either returns nothing or the landmarks of the hand. If it yields nothing we add an empty element to the backlog. Otherwise the landmarks are passed on to the classifier.

2. The classifier will in all cases yield some gesture estimate. Here we choose to ignore all gestures not bound to some user action, and also treat these as non-gestures in the backlog. The defined gestures are passed on to the persistence checker.

3. Persistence compares the gestures estimate with previously classified gestures in the backlog. If persistence deems a gesture intentional we interpret it as user input, otherwise we just ignore it.

Frame Processor

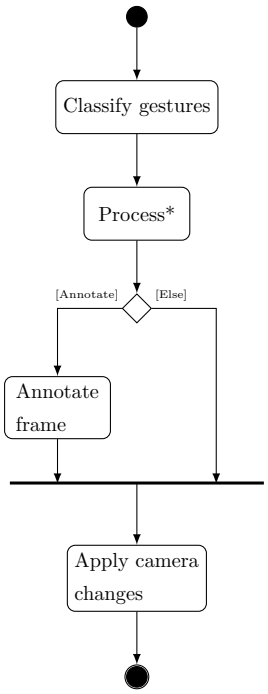


Figure 21: The general flow of the frame processor.

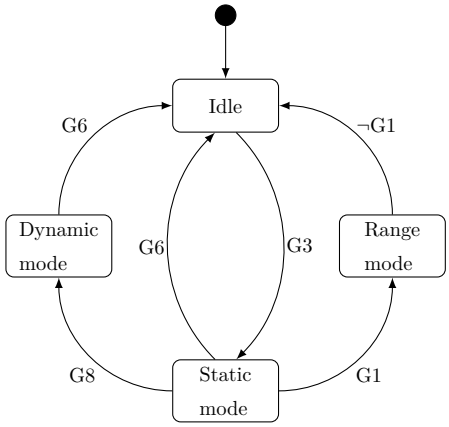


Figure 22: Illustration of how detected gestures transitions the state of the frame processor

The FrameProcessor class is essentially the heart of the application. The diagrams of figure 21 and 22 illustrate the activity and state of the frame processor after receiving a frame to process. First of all, the frame is sent through the recognition pipeline, possibly yielding some gesture. The process method is then run with the image, this is mostly where the logic and changes to the frame is made. Camera settings are applied here and will be repeated for multiple hands. When the process is done running we annotate the classified gestures and landmarks, and lastly apply the

camera settings to the image. The frame processor will then return the manipulated image to the stream controller, which then can write it to a video buffer.

The 'Process' method seen in figure 21 behaves conditionally according to frame processor state. These state dependant implementations are illustrated with the activity diagrams of figures 23, 24, 25 and 26.[19]

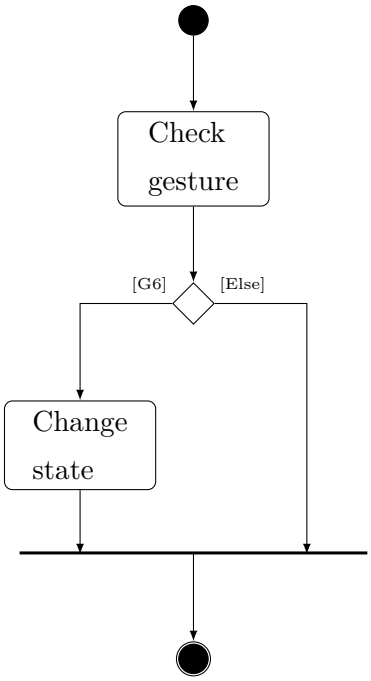


Figure 23: Idle state

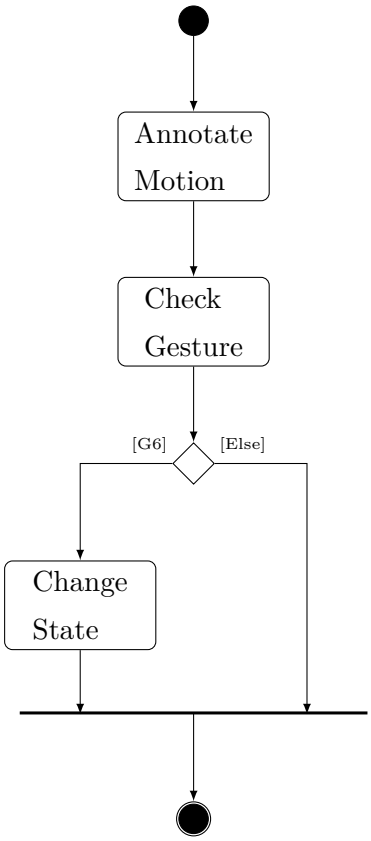


Figure 24: Dynamic gesture state

[19]An in depth description of this state dependent behaviour is available in the full product documentation appendix.

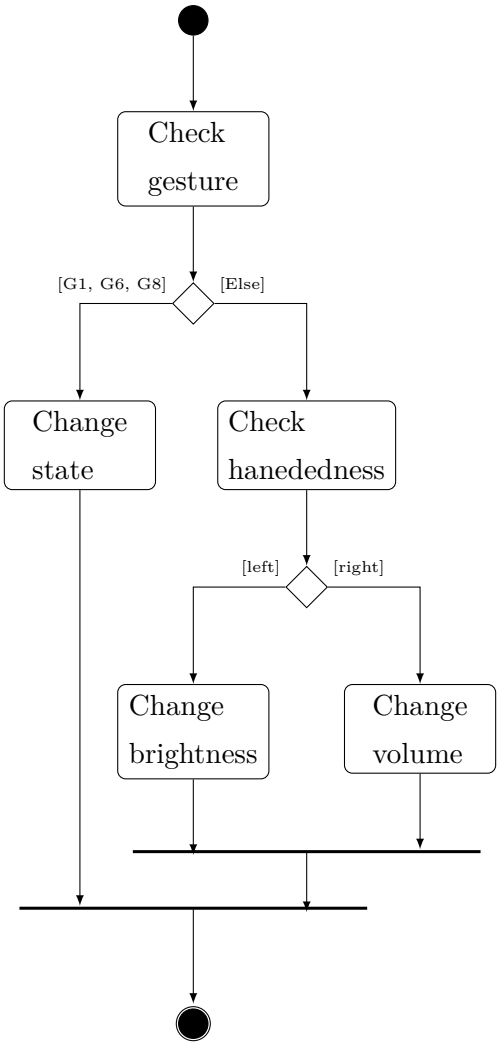


Figure 25: Range gesture state

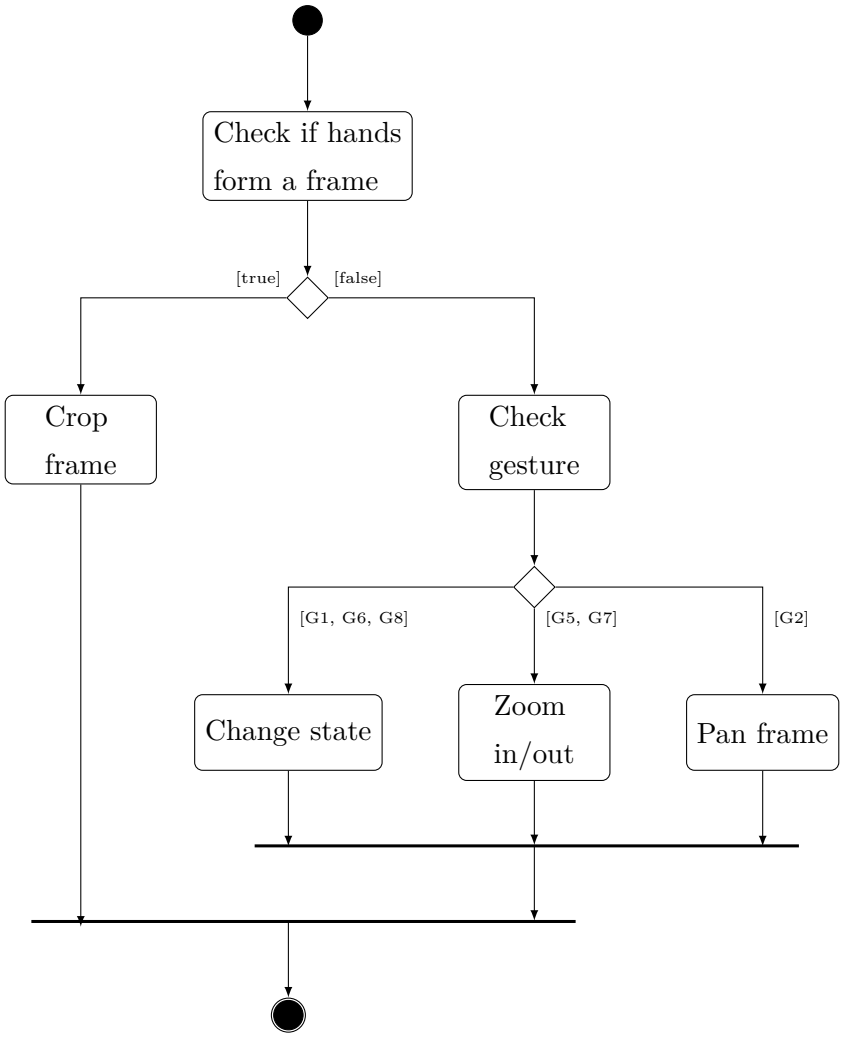


Figure 26: Static gesture state

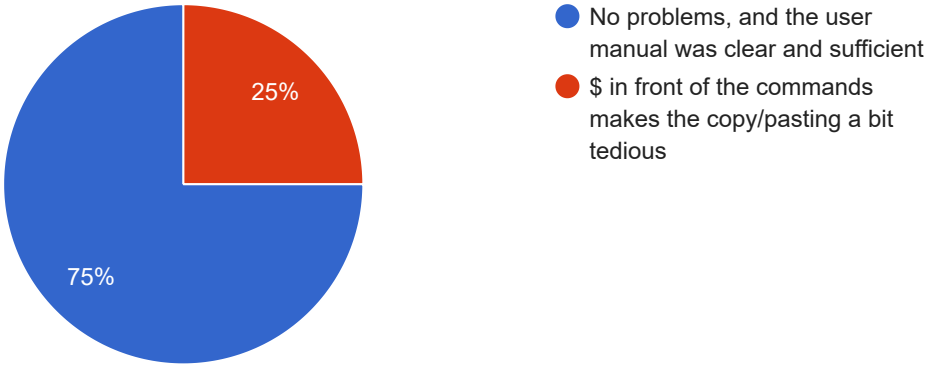
8 Appendices

8.1 User Testing Results

Installation

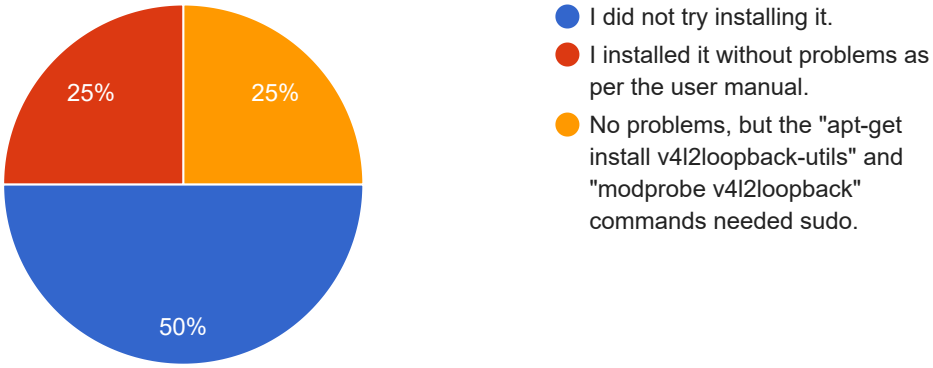
What was your experience of installing the product? (Note: Not counting the optional services).

4 svar



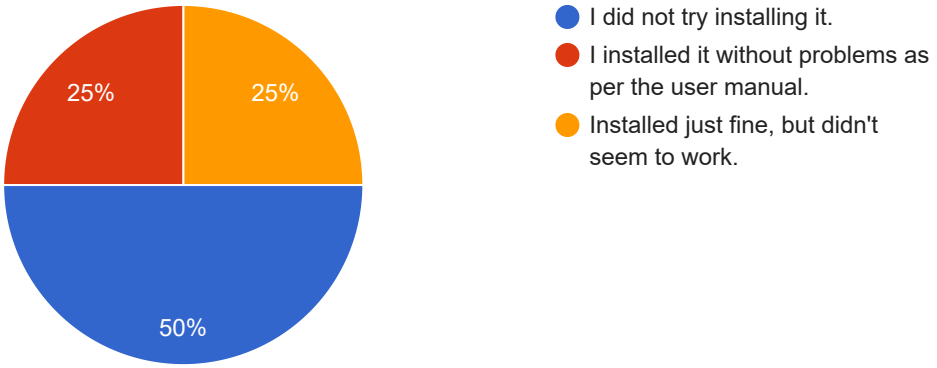
What was your experience installing the (optional) virtual camera service?

4 svar



What was your experience installing the (optional) volume control service?

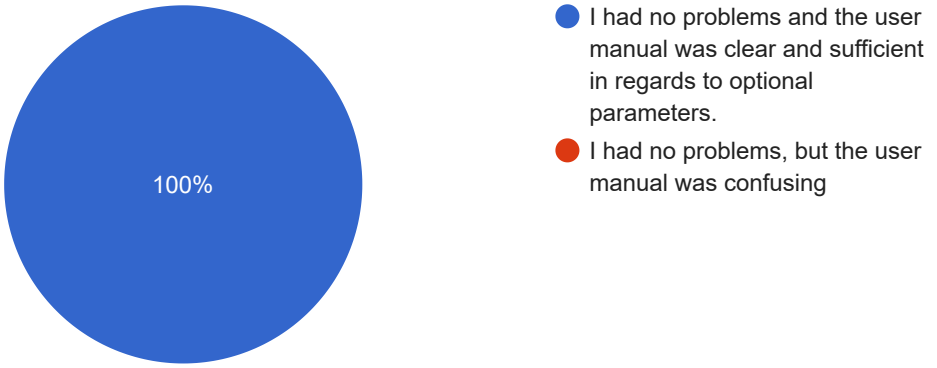
4 svar



Running the demo

What was your experience with running the demo?

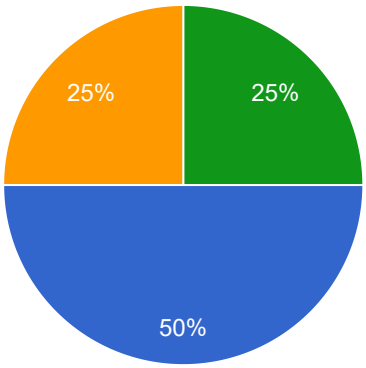
2 svar



Interaction through gestures

What was your experience of Zooming in/out through gestures?

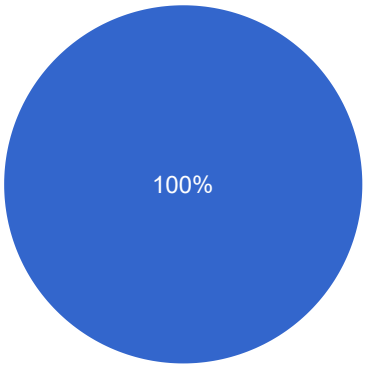
4 svar



- It worked well and the user manual was sufficient.
- I didn't try it
- It worked well with smooth zooming, though a slower framing would be better to accurately zoom in as much a…
- it didnt find my hands quickly, but theuser manual was sufficient

What was your experience of panning through gestures?

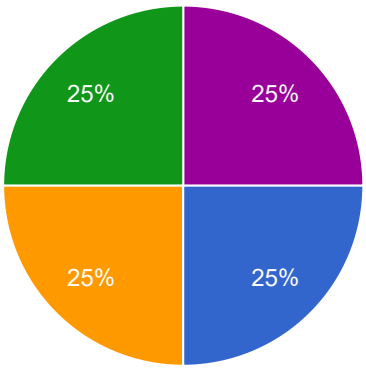
4 svar



- It worked well and the user manual was sufficient.
- I didn't try it

What was your experience of cropping to a frame through gestures?

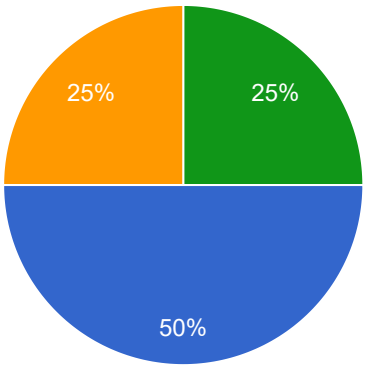
4 svar



- It worked well and the user manual was sufficient.
- I didn't try it
- Worked well, but the frame wobbles a bit.
- it was uncomfortable on my hands, the user manual couldve been better at specifying han…
- Started with a very small crop and when trying to do a bigge…

What was your experience of setting values with sliders through gestures?

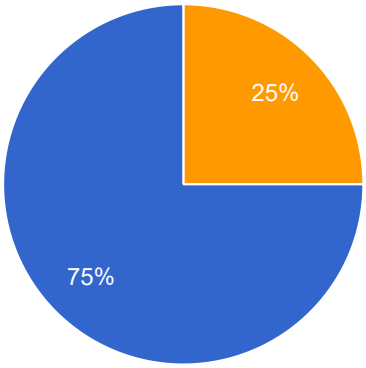
4 svar



- It worked well and the user manual was sufficient
- I didn't try it
- Both worked, but the horizontal (exposure) slider would sometimes flicker and jump between 50-100 in value instantly, seemingly due to sli…
- it was bad, and the slider image felt buggy

What operating system did you test the demo on?

4 svar



- Linux
- Windows
- MacOS
- Solaris

Any other comments?

3 svar

Very cool stuff! It's not always intuitive which gestures does what feature, but that's more related to design and UX rather than functionality.

the cli help was insufficent, and should maybe direct to the readme

Really cool demo and I was able to run it without any issues.
With annotations it would have been really cool with a visual representation of the zoomed out picture with some indication of were the hands are. This to be able to track the hands when the video is zoomed in.

8.2 User Manual

Preface

The following is a user manual for the gesture-recognition repository. This product is a proof-of-concept demonstration of using machine learning powered gesture recognition, to interactively engage with camera and other systems in real time.

The product and manual is intended for users with basic programming and software development experience. It is therefore expected that the user has a working Python version >= 3.6, experience with using Python environments and packages, and knows how to interact with a program through the command line. Furthermore a web camera is required.

Installation

The demo requires at least python 3.6, but we recommend python 3.8 (as this is the recommended version for MediaPipe Hands).

Using the Conda package manager, set up a working 3.8 environment with

```
$ conda create -n gestures python=3.8
$ conda activate gestures
```

Installing requirements

```
$ pip install -r requirements.txt
```

This should be sufficient on most systems. Note that some packages (OpenCV) may require additional steps to be properly configured. For the specific case of OpenCV, this page may be of use.

Remember to set pythonpath, i.e.

```
$ export PYTHONPATH=`path to this repo`
```

Optional: Using program output as virtual webcamera

Note: This extra functionality is not required. The program supports using the program output as a virtual webcamera. This can then for instance be used in

video-chat applications such as Google Meets. This functionality has currently only been tested on Linux systems, therefore the following is specifically oriented towards Linux systems.

Support for a camera loopback device has to be installed. We recommend using the v4l2loopback-utils module:

```
$ sudo apt-get install v4l2loopback-utils
```

The loopback kernel module has to be inserted

```
$ sudo modprobe v4l2loopback
```

If a v4l2loopback device was properly initialized, you can see its status and which interface it is running on with

```
$ v4l2-ctl --list-devices | grep Dummy -A1
```

That interface can then be used with the -l / –loopback option to enable a virtual webcam. Keep in mind that once the program is started with loopback enabled, the virtual webcam gets set to the resolution used at that runtime. If you want to later change the loopback resolution you will need to remove and insert the kernel module again. Removing the module can be done with

```
$ sudo modprobe -r v4l2loopback
```

Optional: Controlling microphone or output volume

Note: This extra functionality is not required. Volume control is implemented for linux machines with alsaaudio. To use this you will need alsaaudio as your audio driver and manually install pyalsaaudio.

```
$ pip install pyalsaaudio
```

The program can then be run with the flag -s / –sound to enable this feature.

Running the demonstration

The demo should in most cases run without any arguments, but it can be started with default values as such

```
$ python src/main.py --interface 0 --num_hands 1 --res 1280 720
```

The program has the following optional arguments

- -i | –interface : Input webcamera interface index (usually 0)
- -l | –loopback : The Interface id of virtual camera for output
- -n | –num_hands : Maximum number of hands to track
- -r | –res : Input and outout stream resolution. Width then height seperated by a space
- -d | –detection_confidence : Mediapipe detection confidence threshold in percent
- -t | –tracking_confidence : Mediapipe tracking confidence threshold in percent

and the following optional flags

- -a | –annotate : Draws annotations of hands and gestures
- -s | –sound : Enables sound control. Requires alsaaudio

The -h or –help flag can also be passed to show help in the terminal.

To exit gracefully, either press the ‘q’ on any window or interrupt the program with Ctrl + c.

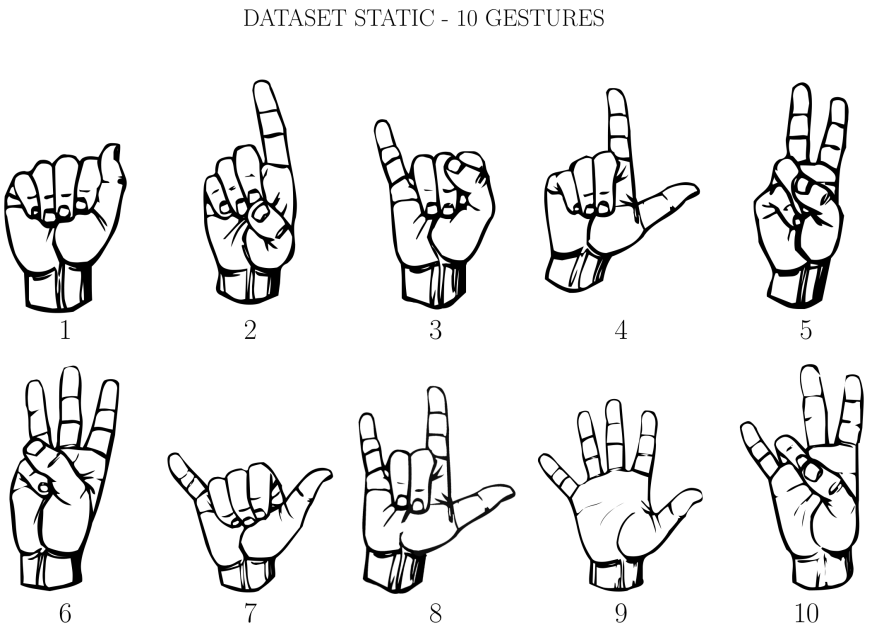
Tips for better user experience

The demos has a bit of a learning curve to it, it is therefore recommended that you use the program with annotation mode on to begin with. This will quickly give you a general feel of when and how the camera will be able to detect your hands.

If you face any problems with the program not finding your hands, try to face the palm of you hand directly towards the camera. Using the program in a well lit area can also improve the detection accuracy quite a bit.

Interacting through gestures

The following gestures are defined



Gesture detection works best when your palms are facing the camera, as shown in the illustration. Furthermore we recommend running the program with the default value `--num_hands=1`, which gives the absolute best user experience (as of now, the only use for `--num_hands=2` is for the cropping feature).

By default, the session starts in idle mode (indicated by the red circle in the top left). To start interacting, you need to enter gesture mode. This is done by holding gesture 3 (raised pinky) until you see the circle (mode indicator) change color to green. To get back to idle mode at any time, you can hold gesture 6 (raised index-middle-ring).

When in control mode, there are a few ways to interact with the camera.

Reset

Before using the other features, it is handy to know how to reset the session settings. This can be done with gesture 10 (thumb and ring finger connected). You can use

this gesture from any state but Idle mode. Resetting will return you to gesture mode, reset all adjustable session settings.

Zooming

You can zoom in at a point through gesture 5 (✌️). To zoom out again, use gesture 7 (👇).

Panning

Panning the camera can be done with gesture 2 (☝️). Note that the camera must be in a zoomed in state for this to have any effect!

Cropping

Cropping the frame can be done by creating a rectangle with two hands, both doing gesture 4 (L with index and thumb). Note that this requires setting `--num_hands=2` when running the program. To get back from a cropped state, zoom out with gesture 7 (👇).

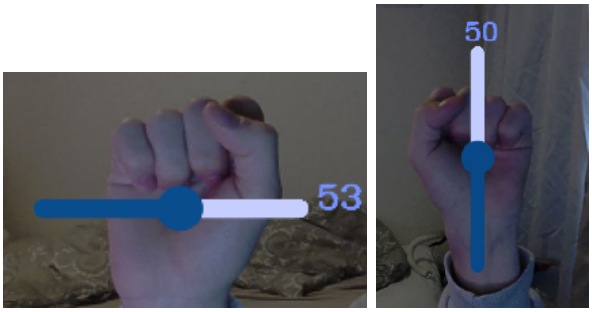
Setting values with sliders

Note: We strongly recommend only using this feature with `--num_hands=1`.

By holding gesture 1 (✊) an interactive slider appears. By moving your hand you can move the slider and thus change the intended value.

When this gesture is initialized on the left side of the screen (typically by using your left hand), the slider is orientated horizontally and controls the brightness of the video stream.

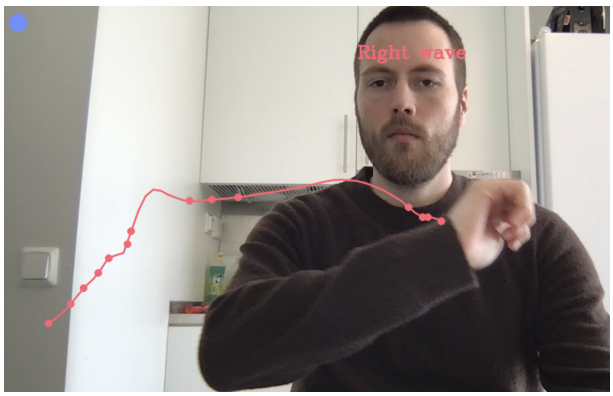
When it is initialized on the right side, (typically by using your right hand), it is orientated vertically and controls microphone volume. Note that this only has an effect if you have configured pyaalsaaudio and are running the program with the `--sound` option.



To exit this state, release gesture 1 (✊).

Experimental: Dynamic gestures A very basic experimental approach to recognition of dynamic gestures has been added as of 15 April 2021. To test this you can enter "dynamic gesture mode" by holding gesture 8 (🤟 [note the extended thumb]) until you see the circle in the top left corner change to a blue color.

This mode can recognize a wave directed either left or right to the screen. To try this, try doing a semi-slow wave to the camera in either direction. A text with either "Right wave" or "Left wave" should appear. Currently this gesture does not control any functionality.



To exit this state, either hold gesture 3 (raised pinky -> back to gesture mode), or gesture 6 (raised index-middle-ring -> back to idle mode).

8.3 Bezier Proof

Analytical Bézier interpolation for smoother user controlled motion

Jonathan Kjølstad

May 25, 2021

Motivation

Instead of zooming linearly with time, we could interpolate the original zoom-values onto some polynomial, to gain control over how the speed of zooming changes over time. A Bézier curve is perfect for this, allowing us to define a parametric polynomial which we can tune by moving some control points in the plane. By tuning this curve correctly, the acceleration and deacceleration of the zooming motion will give a more natural feel to the more mundane and unatural feel of linear motion.

A Bézier curve is a parametric curve given with a set of control points P_0 through P_n, where n is the curve's order. By defintion the first and last control points are always on the curve; however, the intermediate control points (if any) generally do not.

Problem

With the parametrized Bézier curve $B\colon [0,1] \to \mathbb{R}^2$, where $B(t) = (x(t), y(t))$. Given constant first and last control points $P_0 = (0,0)$ and $P_3 = (1,1)$, and the intermediate points $P_1 \in [0,1]^2$ and $P_2 \in [0,1]^2$. We want to construct a function $f\colon [0,1] \to [0,1]$ such that $f(x) = y$ for all $x \in [0,1]$.

The regular approach to solving this is done numerically, most notably css' implementation of the cubic Bézier that uses Newton-Raphson iteration to find an estimate for the function f. However, with the given contstraints we can prove that it is possible to derive the function f analytically, and also show how to construct such a function.

Proof

Given the Bernstein basis polynomial of degree n

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, \ldots, n$$

The Bézier curve $B\colon [0,1] \to \mathbb{R}^2$ can be explicitly parametrized with degree n and control points $P_i = (x_i, y_i)$ as

$$B(t) = (x(t), y(t)) = \sum_{i=0}^{n} b_{i,n}(t) P_i$$

It's derivate with respect to t is then

$$B'(t) = n \sum_{i=0}^{n-1} b_{i,n-1}(t)(P_{i+1} - P_i)$$

Furthermore, we can split the parameterization into its constituant parts

$$x'(t) = n \sum_{i=0}^{n-1} b_{i,n-1}(t)(x_{i+1} - x_i), \quad y'(t) = n \sum_{i=0}^{n-1} b_{i,n-1}(t)(y_{i+1} - y_i)$$

With four control points in two dimensions we get a cubic Bézier with degree $n = 3$. The first and last point being constant, $P_0 = (0,0)$, $P_3 = (1,1)$ and the intermediate points being $P_1 \in [0,1]^2$, $P_2 \in [0,1]^2$

Plugging in n and the constant x-values, $x_0 = 0$ and $p_3 = 1$ into $x'(t)$ gives

$$x'(t) = 3(1-t)^2 x_1 + 6(1-t)t(x_2 - x_1) + 3t^2(1 - x_2)$$

This can be refactored with respect to x_1 and x_2 as

$$x'(t) = 3x_1(3t^2 - 4t + 1) + 3x_2(-3t^2 + 2t) + 3t^2$$

We can simplfy the expressions by redefining the quadratic polynomials as functions, we then get

$$x'(t) = 3(x_1 p(t) + x_2 q(t) + r(t))$$

with

$$p(t) = 3t^2 - 4t + 1, \quad q(t) = -3t^2 + 2t, \quad r(t) = t^2$$

We can see that $r(t)$ is postive for all t. We can also conclude that the sum of the three polynomials $p(t) + q(t) + r(t) = t^2 - 2t + 1 = (t-1)^2 \geq 0$ for all t since it has a minima at 0 when $t = 1$.

If either $x_1 p(t)$ or $x_2 q(t)$ is negative we know they are the most negative when $x_1 = 1$ or $x_2 = 1$ respectively. We therefore get $x_1 p(t) \geq p(t)$ and $x_2 q(t) \geq q(t)$

$$\begin{aligned} x_1 p(t) + x_2 q(t) + r(t) &\geq p(t) + x_2 q(t) + r(t) \\ &\geq p(t) + q(t) + r(t) \\ &\geq 0 \end{aligned}$$

We can therefore conclude that $x'(t)$ is strictly positive on the open interval $t \in (0, 1)$, meaning $x(t)$ is strictly growing with control points $x_0 = 0$ and $x_3 = 1$. Similarly this holds for $y(t)$ with control points $y_0 = 0$ and $y_3 = 1$ on the same interval. More importantly this proves that both $x(t)$ and $y(t)$ are unambiguous for all $t \in [0, 1]$, meaning it is possible to construct a function such that $y = f(x)$.

Construction

$x(t)$ is a cubic polynomial and we can factor it with respect to t

$$\begin{aligned} x(t) &= 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 \\ &= (3x_1 - 3x_2 + 1)t^3 + (3x_2 - 6x_1)t^2 + (3x_1)t \end{aligned}$$

We consider the above-mentioned cubic polynomial equation for $x(t)$ in the form $at^3+bt^2+ct = x$, apply the substitution $t \equiv u-\lambda$, and divide the entire equation by a to get the cubic terms coefficient as 1.

$$a = (3x_1 - 3x_2 + 1), \quad b = (3x_2 - 6x_1), \quad c = 3x_1$$

$$\begin{aligned} x &= at^3 + bt^2 + ct \\ 0 &= a(u-\lambda)^3 + b(u-\lambda)^2 + c(u-\lambda) - x \\ &= a(u^3 - 3u^2\lambda + 3u\lambda^2 - \lambda^3) + b(u^2 - 2u\lambda + \lambda^2) + c(u-\lambda) - x \\ &= \left(u^3 - 3u^2\lambda + 3u\lambda^2 - \lambda^3\right) + \frac{b}{a}\left(u^2 - 2u\lambda + \lambda^2\right) + \frac{c}{a}(u-\lambda) - \frac{x}{a} \\ &= u^3 + \left(-3\lambda + \frac{b}{a}\right)u^2 + \left(3\lambda^2 - 2\lambda\frac{b}{a} + \frac{c}{a}\right)u + \left(-\lambda^3 + \frac{b}{a}\lambda^2 - \frac{c}{a}\lambda - \frac{x}{a}\right) \end{aligned}$$

The quadratic term is eliminated by letting $\lambda = \frac{b}{3a}$. The substition $t \equiv u - \frac{b}{3a}$ will then give the general, depressed cubic equation for $x(t)$

$$u^3 + \left(\frac{3ac - b^2}{3a^2}\right)u + \frac{9abc + 27a^2x - 2b^3}{27a^3} = 0$$

The above equation can be solved using Cardano's formula in normal cases, or with the trigonometric cubic equation when the quadratic term is zero, yielding all three solutions. The above proof shows that with our constraints one and only one of these will be real. By substituting t back in, we can use this to obtain a function $t = t(x)$, we can use this in conjunction with $y(t)$ to express the function $f\colon [0,1] \to [0,1]$ which interpolates a given input x-value to a unique y-value. $f(x) = y(t(x))$

8.4 Full Product Documentation

The following appendix is an overview of all public entities of the gesture recognition codebase. It is comprised of docstrings and signatures, and has been automatically generated with the pdoc[20] library. In addition to this pdf document, an online version is available at the reader's discretion at `https://kjxlstad.github.io/bachelor-homepage/documentation/src/index.html`. While both versions contain the same information, we personally recommend the online version.

[20] `https://pdoc3.github.io/pdoc/`

Contents

1 Module `src`

1.1 Sub-modules

- src.main
- src.processing
- src.recognition
- src.streaming
- src.util

2 Module `src.main`

Main module, the programs entry point

- Checks input arguments and parses these
- Prints settings and status to terminal
- Configures special functionality according to arguments
- Starts a StreamController with specified settings

3 Module `src.processing`

Functionality for image manipulation such as drawing, cropping, zooming, setting exposure etc.

3.1 Sub-modules

- src.processing.cam_control

- src.processing.drawing

4 Module `src.processing.cam_control`

Module for manipulating images as if it was a camera

4.1 Classes

4.1.1 Class `CamController`

```
class CamController(
    res: Tuple[int, int] = (1920, 1080)
)
```

Keeps a state for the current camera settings (zoom, center, exposure) uses this to process frames.

Zoom, pan and set_gamma changes fields in the object but return nothing. The process function is then used to process the image with the current settings. It is set up this way to support multiple hands controlling the camera at once.

Multiple calls to zoom and pan can be done on each input frame. But the image buffer will only be manipulated once.

Args ——=
`res` : Resolution of frames, tuple of (x, y)

4.1.1.1 Methods

Method `process`

```
def process(
    self,
    frame: numpy.ndarray
)  -> numpy.ndarray
```

Processes the frame with the current object settings.

First crops the frame to the current zoom and center values. Then applies gamma correction to the cropped image. If these values have not been altered since object instantiation, the corresponding functions will have no effect.

Args —=
frame : Image buffer

Returns —= Cropped, translated and gamma corrected image buffer.

Method interpolate_zoom

```
def interpolate_zoom(
    self,
    zoom: float
)  -> float
```

Interpolates zoom values with bezier

Args —=
Zoom : zoom value, [-1, 1], -1 and 1 being zooming out or in at max speed respectively

Returns —= Difference in zoom in respect to current zoom

Method zoom

```
def zoom(
    self,
    zoom: float,
    smoothing: bool = True
)  -> NoneType
```

Zooms in or out on center of frame

Adds zoom input to the zoom field and changes the size field accordingly. Size is essentially width and height of frame after crop

Args —=

zoom signed float representing zoom percent and direction
smoothing whether to interpolate zoom values with bezier or not

Method **pan**

```
def pan(
    self,
    pos: Tuple[float, float]
)  -> NoneType
```

Translates cropped frame

Scales position in landmark space to image dimensions and moves the center field to that point. Effectively translating cropped image on the input image.

Args —=
pos : position in landmark space

Method **hand_framed_crop**

```
def hand_framed_crop(
    self,
    multi_landmarks: Tuple[Iterable[mediapipe.
       framework.formats.landmark_pb2.
       NormalizedLandmark], Iterable[mediapipe.
       framework.formats.landmark_pb2.
       NormalizedLandmark]]
)  -> NoneType
```

Zooms in on region enclosed by two hands

Creates a corner point in the intersection between index and thumb for the first two hand_landmarks in multi_landmarks. Sets self.center and self.size to fit the to fit the resulting rectangle.

Args ——=
multi_landmarks : tuple containing two hands landmarks

Method **set_gamma**

```
def set_gamma(
    self,
    normalized_gamma: float
)  -> NoneType
```

Sets self._gamma from a normalized value.

Takes in a normalized value. If the value is below 0.5, gamma is set to 1 / c, else it is set to c. c has range [1.0 self.gamma_max] and is given by the linear function

$$c(x) = 1 + (x * (\gamma_{max} - 1))$$

where x is the absolute value of normalized_gamma shifted by -0.5, and γ_{max} = self.gamma_max

Args ——=
normalized_gamma : A normalized representation ([0.0, 1.0]) of the intended gamma value.

5 Module **src.processing.drawing**

Drawing functions.

These functions use opencv to draw on the provided frames. For performance, they draw in place, mutating the provided image.

5.1 Functions

5.1.1 Function **draw_fps_counter**

```
def draw_fps_counter(
    img: numpy.ndarray,
    fps: int
)  -> NoneType
```

Draws fps counter in the bottom left

Args ——=
fps : rounded down fps estimate

5.1.2 Function **label_bounding_box**

```
def label_bounding_box(
    img: numpy.ndarray,
    text: str,
    orig: Tuple[int, int]
)  -> NoneType
```

Draws label with text on bounding box in place.

Calculates size of text, draws a background of textsize and draws the input text over it. Both the drawing of background rectangle and text is done in place.

Args —=

img image buffer
text name of gesture
orig upper left of hand bounding box

5.1.3 Function `draw_mode_indicator`

```
def draw_mode_indicator(
    img: numpy.ndarray,
    color: str
)  -> NoneType
```

Draws circular indicator in place.

Based on input bool it sets color as either green or red and draws a circle in the top left corner in place.

Args —=

img Image buffer as numpy array.
color Color of mode indicator, One of {“BLACK”, “BLUE”, “WHITE”, “RED”, “GREEN”, “GRAY”}.

5.1.4 Function `bounding_box`

```
def bounding_box(
    img: numpy.ndarray,
    landmarks: Iterable[mediapipe.framework.formats
       .landmark_pb2.NormalizedLandmark],
```

```
    gesture: str,
    padding: int = 15
)  -> NoneType
```

Draws bounding box in place.

Takes in keypoints and predicted gesture, creates bounding box around hand with extra padding and draws the estimated gesture. Draws in place

Args —=

img Image buffer as numpy array.
landmarks List of hand landmarks for a single hand.
gesture Name of gesture.
padding Extra width and height to add outside the outermost points in landmarks

5.1.5 Function `draw_landmarks`

```
def draw_landmarks(
    img: numpy.ndarray,
    hand_landmarks: Iterable[mediapipe.framework.
       formats.landmark_pb2.NormalizedLandmark]
)  -> NoneType
```

Draws landmarks on image in place.

Draws joints as circles and the connection of joints as lines on image in place.

Args —=

img image buffer
hand_landmarks List of hand landmarks for a single hand

5.1.6 Function `draw_slider`

```
def draw_slider(
    img: numpy.ndarray,
    bounds: Tuple[float, float],
    slider_pos: float,
    slider_val: float,
    value_text: str,
    orientation: str = 'VERTICAL',
    size=20
)  -> NoneType
```

Draws a slider on the image (in place).

Draws a slider with the value text on the image. Can be oriented either horizontally or vertically. The slider will be 'filled' to the slider_val.

Args ——=

`img` Image buffer as a np array.

`bounds` Normalized coordinates for the bounds, is treated as x or y coordinates depending on HORIZONTAL or VERTICAL orientation respectively.

`slider_val` The normalized coordinate for the current slider value. Note: Normalized on the image

`value_text` Text to display.

`orientation` Orientation for the slider to be drawn, One of "VERTICAL" | "HORIZONTAL"

`size` Width/height in pixels depending on orientation being "VERTICAL" or "HORIZONTAL" respectively.

5.1.7 Function `draw_func_through_points`

```
def draw_func_through_points(
    img: numpy.ndarray,
    f: Callable[[float], float],
```

```
    normalized_points: List[Tuple[float, float]]
) -> NoneType
```

Draws the function f and the provided points in place.

Draws the provided points on the image. Then draws the function f from min x to max x of the provided points. Assumes that f is defined for [x_min, x_max] of the normalized points.

Args —=

img Image buffer as a np array.
f Function to be drawn/interpolate between the points.
normalized_points List of normalized points [(x0, y0), (x1, y1), ...] Can be unsorted.

6 Module `src.recognition`

Functionality related to the detection and classification of hand gestures.

6.1 Sub-modules

- src.recognition.gesture_classifier
- src.recognition.hand
- src.recognition.mphands
- src.recognition.persistence

7 Module `src.recognition.gesture_classifier`

7.1 Classes

7.1.1 Class `GestureClassifier`

```
class GestureClassifier(
    hand_landmarks: mediapipe.framework.formats.
       landmark_pb2.NormalizedLandmarkList
)
```

Gesture Classifier

Gesture Classifier initializes with a list of 21 landmarks belonging to one hand. It predicts a gesture from these landmarks by labeling each finger as either open (True) or closed (False).

Finger 'openness' is estimated the following way: A given finger has three joints. The algorithm sums up the angles for these joints, and compares them to a threshold. If the accumulated angles are greater then the threshold, the finger is considered closed. Otherwise it is open.

Thumb 'openness' is a special case where both positive and negative values are considered. The sign of an angle is determined by estimating a normal vector for the palm, and using this as an orientation vector. The angle of the joint is then calculated through:

$$\sin\theta = \frac{(\boldsymbol{n} \times \boldsymbol{u}) \cdot \boldsymbol{v}}{|u||v|}.$$

$$\theta = \begin{cases} \arccos \frac{u \cdot v}{|u||v|}, & \text{if } \sin\theta \geq 0; \\ 2\pi - \arccos \frac{u \cdot v}{|u||v|}, & \text{if } \sin\theta < 0. \end{cases}$$

Where θ is the joint angle, $\boldsymbol{n}$ is the estimated palm normal, and $\boldsymbol{u}$ and $\boldsymbol{v}$

are the vectors extending from the joint. The thumb has its separate threshold for 'openness', defined in GestureClassifier_THUMB_ANG_TRESHOLD

7.1.1.1 Methods

Method `estimate`

```
def estimate(
    self
)  -> str
```

Estimate name of gesture

Checks finger states and tries to find a matching gesture in the list of defined gestures. If a matching gesture is found it returns that gestures name. If not it returns a fallback string representing the state of each finger.

Returns —= Label of predicted gesture

8 Module `src.recognition.hand`

8.1 Classes

8.1.1 Class `Hand`

```
class Hand(
    gesture_backlog: int = 20,
    threshold: float = 0.8
)
```

Hand - a wrapper for processing hand landmarks with persistence.

Since lots of different features need access to the GestureClassifier, hand ensures that a landmark list is only processed once.

After a gesture for the hand landmarks has been estimated, it is passed on to the persistence checker.

Args ——=

gesture_backlog How many gestures (queue size) form the basis for persistence checks.

threshold Minimum ratio ([0.0, 1.0]) for regarding the current gesture as intentional.

8.1.1.1 Instance variables

Variable has_intent Property that signifies if the object currently holds a gesture intended as intentful. From outside the class, this property is readable only

8.1.1.2 Methods

Method update_state

```
def update_state(
    self,
    landmarks: mediapipe.framework.formats.
       landmark_pb2.NormalizedLandmarkList
)  -> NoneType
```

Updates the state of the hand.

Takes the provided landmarks to estimate a gesture and passes this on to the persistence checker.

Args ——=

landmarks : A list of landmarks for one hand. Also accepts None, which is equivalent to no gesture being found.

9 Module `src.recognition.mphands`

9.1 Classes

9.1.1 Class `MPHands`

```
class MPHands(
    confidence_settings: Tuple[int, int] = (50, 50)
       ,
    max_hands: int = 1,
    static_image_mode: bool = False
)
```

Simple wrapper for mediapipe hands.

Args ——=

min_det Minimum confidence value ([0.0, 1.0]) for hand detection to be considered successful. See details in `https://solutions.mediapipe.dev/hands#min_detection_confidence.`

min_track Minimum confidence value ([0.0, 1.0]) for the hand landmarks to be considered tracked successfully. See details in `https://solutions.mediapipe.dev/hands#min_tracking_confidence.`

static_image_mode Whether to treat the input images as a batch of static and possibly unrelated images, or a video stream. See details in `https://solutions.mediapipe.dev/hands#static_image_mode.`

max_hands Maximum number of hands to detect. See details in `https://solutions.mediapipe.dev/hands#max_num_hands.`

9.1.1.1 Methods

Method **get_landmarks**

```
def get_landmarks(
    self,
    img: numpy.ndarray
)  -> List[mediapipe.framework.formats.landmark_pb2
   .NormalizedLandmarkList]
```

Processes the image using the mediapipe hand model pipeline.

Args ——=
img : An RGB image represented as a numpy ndarray.

Raises ——= RuntimeError : If the underlying mediapipe graph occurs any error.

ValueError If the input image is not three channel RGB.

Returns ——= A list with length equaling the max_hands arg passed to constructor. Each element in the list represents a hand through a list of 21 hand landmarks. Each landmark is composed of normalized x, y and z coordinates. See details in `https://google.github.io/mediapipe/solutions/hands#multi_hand_landmarks` (Note that handedness (i.e. left or right hand) is not currently available through this wrapper.)

10 Module **src.recognition.persistence**

10.1 Classes

10.1.1 Class **Persistence**

```
class Persistence(
    n: int = 20,
    threshold: float = 0.8
)
```

Persistence - checking for persistence in a queue of gestures.

The idea is to have a FIFO-queue of the last n frames gestures for each hand. Whenever the last element in that queue changes we check the if the ratio between frames containing and not containing that gesture reaches a certain threshold. If it does, we can safely say the person intended to do that gesture.

Args —=

n Size of first half of queue for gesture intention second half for zoom smoothing

threshold Minimum ratio ([0.0, 1.0]) for regarding the current gesture as intentional.

10.1.1.1 Methods

Method add

```
def add(
    self,
    gesture: str
)  -> NoneType
```

Adds the provided gesture to, and removes the first in the queue.

Args —=

gesture : Gesture string to be added to the queue. Accepts None, which is regarded as no gesture.

Method **check_intent**

```
def check_intent(
    self,
    gesture: str
)  -> bool
```

Checks if the provided gesture passes the threshold.

Args —=
gesture : Gesture to check against the queue.

Returns —= True if the provided gesture make up more than self._threshold of the internal queue.

Method **check_ratio**

```
def check_ratio(
    self,
    gesture: str
)  -> float
```

Check the ratio of the provided gesture in the queue.

If the queue is [“G1”, “G1”, “G2”, “G3”], calling check_ratio(“G1”) will return 2 / 4 = 0.5

Args —=
gesture : The intended gesture to check the ratio for.

Returns —= The ratio of the provided gesture in the current queue.

11 Module `src.streaming`

Functionality for controlling input and output video streams, and controlling the processing of individual frames.

11.1 Sub-modules

- src.streaming.frame_processor
- src.streaming.framerate_calculator
- src.streaming.io
- src.streaming.stream_controller

12 Module `src.streaming.frame_processor`

Module for processing individual frames

12.1 Classes

12.1.1 Class `FrameProcessor`

```
class FrameProcessor(
    num_hands: int,
    res: Tuple[int, int],
    annotate: bool,
    confidence_settings: Tuple[int, int],
    volume_mixer=None
)
```

FrameProcessor manages state and processes frame in the class context.

A FrameProcessor object starts in the Idle state. From this it can enter several different states, which effect what gestures will be detected, and how they in turn affect the camera session.

Camera settings (crop, zoom, exposure) is managed with a CamController class member. These settings are manipulated through interacting with gestures.

For more info on how to interact with this class through gestures, see the user manual (README.md).

Args —=

num_hands Max number of hands for inference.
res Resolution (width, height).
annotate Whether to draw annotations over images.
volume_mixer Alsaaudio volume mixer object.

12.1.1.1 Methods

Method process_frame

```
def process_frame(
    self,
    frame: numpy.ndarray
)  -> numpy.ndarray
```

Process a video frame in the current object context.

Processes a frame in accordance with the current state. This involves running inference for gestures and acting correspondingly. The class also manages current camera settings, i.e. zoom, cropping, volume settings, etc. See the user manual (README.md) for more info.

Args —=
frame : Image as a numpy array.

Returns ——= A new processed image as a numpy array. Can for instance be zoomed, cropped, contain drawn annotations etc.

12.1.2 Class **FrameProcessorState**

```
class FrameProcessorState
```

Abstract base class for FrameProcessor State.

The states that inherit this class specify how the program responds to gesture interaction. Different states allow for different gestures and responses. We recommend reading the user manual, (README.md) for a better grasp of how this works.

Note: For all inherited objects, the user can go to the idle state by performing the corresponding gesture.

12.1.2.1 Descendants

- src.streaming.frame_processor.DynamicGesture
- src.streaming.frame_processor.Gesture
- src.streaming.frame_processor.Idle
- src.streaming.frame_processor.RangeGesture

12.1.2.2 Methods

Method `process`

```
def process(
    self,
    fp: src.streaming.frame_processor.
       FrameProcessor,
    frame: numpy.ndarray
```

```
)  -> NoneType
```

Processes a video frame in the object state.

Args —=

fp The parent FrameProcessor object that controls the current session.
frame Image as a numpy array.

12.1.3 Class **Idle**

```
class Idle
```

Idle state

In this state, the program looks for gestures indicating that the user intends to enter either Gesture or Dynamic state.

12.1.3.1 Ancestors (in MRO)

- src.streaming.frame_processor.FrameProcessorState

12.1.4 Class **Gesture**

```
class Gesture
```

Gesture state

In this state, the program is open for interaction through gestures. The user can zoom, pan or crop the camera, reset the settings or enter any of the other states.

12.1.4.1 Ancestors (in MRO)

- src.streaming.frame_processor.FrameProcessorState

12.1.5 Class **RangeGesture**

```
class RangeGesture(
    point,
    filled_to,
    orientation
)
```

RangeGesture state

In this state, the program renders a interactive slider that can used to set session settings. The slider orientation is dependant on where it was created, left side of the screen -> horizontal, right side -> vertical. The user can exit this state by releasing the fist gesture, which keeps the value setting. Alternatively, they can use the reset gesture to reset the session. Both options enter Gesture state.

12.1.5.1 Ancestors (in MRO)

- src.streaming.frame_processor.FrameProcessorState

12.1.6 Class **DynamicGesture**

```
class DynamicGesture
```

EXPERIMENTAL: DynamicGesture state

In this state, the program looks for dynamic gestures. Currently, the only support is for (somewhat slow) hand waving. A recognized wave will render "Left/Right wave!" to the screen. Does not currently control any functionality.

The user can exit this state either by entering Gesture or Idle state, with their corresponding gestures.

12.1.6.1 Ancestors (in MRO)

- src.streaming.frame_processor.FrameProcessorState

13 Module `src.streaming.framerate_calculator`

Module for calculating framerate

13.1 Classes

13.1.1 Class `FramerateCalculator`

```
class FramerateCalculator(
    length: int
)
```

Keeps track of timestamps, and uses these to calculate average and live framerate

Args ——=
`length` : The size of the backlog of times, the number of frames to average over.

13.1.1.1 Methods

Method **timestamp**

```
def timestamp(
    self,
    time: float
)  -> NoneType
```

Timestamps current time

Calculates framerate based on the difference in time from this and the previous call to this method. It then adds that to the backlog.

This method can then be run once per frame, and should yield an accurate framerate.

Args ——=
time : Current time.

Method **get_fps**

```
def get_fps(
    self
)  -> int
```

Gets current framerate from last item in backlog

Method **get_average_fps**

```
def get_average_fps(
    self
)  -> int
```

Calculates average over all framerates in backlog

14 Module `src.streaming.io`

Reader and writer modules for turning video streams into singular frames and vice versa.

These sub modules follow these simple conventions:

- All readers and writers must implement begin() and end() methods.
- All readers implements a frame_generator() method, that should return an iterable of the frames.
- All writers implement write() method.

14.1 Sub-modules

- src.streaming.io.video_reader
- src.streaming.io.video_writer
- src.streaming.io.webcam_writer
- src.streaming.io.window_writer

15 Module `src.streaming.io.video_reader`

Module for reading video files or webcam streams

15.1 Classes

15.1.1 Class `VideoReader`

```
class VideoReader(
```

```
        interface: Union[int, str],
        res: Union[tuple, NoneType] = None
    )
```

An instance can be created for either a videofile with interface being path or for a webcam stream with interface being the video interface number.

Contains methods for beginning and ending stream, and a generator of frames

Args —=

interface Either path to videofile or webcam interface number
res Resolution provided a tuple on the form (width, height).

15.1.1.1 Methods

Method **begin**

```
def begin(
    self
)  -> NoneType
```

Starts videocapture and sets caputure resolution equal to the videofile or webcam resolution if no resolution is specified.

Method **end**

```
def end(
    self
)  -> NoneType
```

Safely ends videocapture

Method `frame_generator`

```
def frame_generator(
    self
)  -> Iterable[Union[numpy.ndarray, NoneType]]
```

Creates iterable frame generator

Creates a generator for frames that works for both finite (videofile) and infinite (webcam) input frames. A frame is read, if retrieved apply selfie-mirror, and convert it to rgb before yielding it. If it is not retrieved end the generator by yielding None

Returns ——= Iterable generator for instances of either an image buffer or None.

16 Module `src.streaming.io.video_writer`

Module for writing to video files

16.1 Classes

16.1.1 Class `VideoWriter`

```
class VideoWriter(
    path: str,
    res: Tuple[int, int],
    fps: int = 30
)
```

An instance is made with path to output file, resolution and framerate.

Contains methods for starting and stopping video output, and a method that takes in a image and writes it as a frame in videobuffer.

Args ——=

path Path to output file with file extension. Defaults to .mjpg
res Resolution of the saved video.
fps Desired framerate of output video, 30 fps yields realtime in most cases.

16.1.1.1 Methods

Method **begin**

```
def begin(
    self
)  -> NoneType
```

Sets codec and starts video output

Method **end**

```
def end(
    self
)  -> NoneType
```

Stops generation of output file and safely saves it.

Method **write**

```
def write(
    self,
    img: numpy.ndarray
)  -> NoneType
```

Writes image to videobuffer

Takes input image, converts it from RGB to BGR color and writes it to videobuffer.

Args —=
img : image buffer as numpy ndarray.

17 Module `src.streaming.io.webcam_writer`

Module for writing to a virtual webcam that can be used anywhere.

This module only runs on linux machines as it requires that the external v4l2loopback-utils package is installed and that the v4l2loopback driver is inserted into the kernel. This also means the begin and end methods are empty and just pass.

17.1 Classes

17.1.1 Class `WebcamWriter`

```
class WebcamWriter(
    interface: int,
    res: Tuple[int, int]
)
```

An instance is made with a webcam interface number and resolution.

Looks for v4l2loopback devcies on /dev/videoN, where N is the interface number

Contains methods for starting and stopping video output and a method that takes in a image and writes it as a frame in videobuffer.

17.1.1.1 Methods

Method `write`

```
def write(
    self,
    img: numpy.ndarray
)  -> NoneType
```

Writes image to videobuffer

Takes input image, mirrors it and writes it to loopback device

Args ——=
`img` : image buffer as numpy ndarray.

18 Module `src.streaming.io.window_writer`

Module for writing to a window

18.1 Classes

18.1.1 Class `WindowWriter`

```
class WindowWriter(
    name: str
)
```

Writes output to a opencv window, contains methods for starting and stopping the stream and a method for writing an image to the window.

Args ——=
`name` : The title of the window.

18.1.1.1 Static methods

Method key_pressed

```
def key_pressed(
    key: str
)  -> NoneType
```

Detects keypress on the open window

Args ——=
key : The key to check if have been pressed during this frame.

18.1.1.2 Methods

Method begin

```
def begin(
    self
)  -> NoneType
```

Opens resizeable window with given name

Method end

```
def end(
    self
)  -> NoneType
```

Destroys the opened window and stops writing output.

Method `write`

```
def write(
    self,
    img: numpy.ndarray
)  -> NoneType
```

Displays frame in window

Takes provided image and converts it from RGB to BGR before showing in window

Args ——=
`img` : The image buffer as an numpy ndarray.

19 Module `src.streaming.stream_controller`

Module that links an input stream to an output stream through a frame processor

19.1 Classes

19.1.1 Class `StreamController`

```
class StreamController(
    interface: Union[int, str],
    num_hands: int,
    res: Tuple[int, int],
    annotate: bool,
    confidence_settings: Tuple[int, int],
    sound_control: bool = False
)
```

A controller for stream objects. Input and output stream types need to be defined here. Implements begin, end and iterate_frames public methods

begin and end starts and stops whole program iterate_frames is the main program loop, reading a frame, processing it and drawing it to screen

Args —=

interface Either path to videofile or webcam interface number
num_hands Maximum number of hands to support
res Resolution of processing on the form (width, height)
annotate If True, adds annotations to stream
confidence_settings Confidence threshold for mediapipe model
sound_control If True, control sound with sliders

19.1.1.1 Methods

Method `iterate_frames`

```
def iterate_frames(
    self
)  -> NoneType
```

Iterates over and processes every frame

Starts by initiating specified input and output streams. Frames are then read from the input stream, processed with frame processor and written to the output stream.

Once we stop receiving input frames or it is manually stopped, it will safely quit both input and output streams.

20 Module `src.util`

Utilitary functionality

20.1 Sub-modules

- src.util.bezier_interpolator
- src.util.cubic_polynomial
- src.util.intervaled_value
- src.util.landmark_utils
- src.util.math_utils

21 Module `src.util.bezier_interpolator`

Module for interpolating values with two dimensional bezier curves

21.1 Classes

21.1.1 Class `BezierInterpolator`

```
class BezierInterpolator(
    p_1: Tuple[float, float],
    p_2: Tuple[float, float]
)
```

Interpolates values with two dimensional cubic bezier curve

The curve $B\colon [0,1] \to \mathbb{R}^2$, where $B(t) = (x(t), y(t))$ is defined by the static control points, $p_0 = [0,1]^2$ and $p_3 = [0,1]^2$, and the itermediate control points $p_1 \in [0,1]^2$ and $p_2 \in [0,1]^2$.

By using $y(t)$ and the inverse of $x(t)$ we can interpolate an input value $[0,1]$ to an output value also on $[0,1]$ with the interpolate() method.

Args —=

`p_1` The first intermediate control point, P_1.

p_2 The second intermediate control point, P_2.

21.1.1.1 Methods

Method **interpolate**

```
def interpolate(
    self,
    x: float
)  -> float
```

Interpolates input with given bezier curve

Takes input value x, and returns the corresponding y value on the bezier curve defined by the control points.

Args ——=
x : the value we want to interpolate $[0, 1]$.

Returns ——= The interpoalted value, also $[0, 1]$.

22 Module **src.util.cubic_polynomial**

Module for handling and finding roots of cubic polynomaials
Used for bezier interpolation.

22.1 Classes

22.1.1 Class **CubicPolynomial**

```
class CubicPolynomial(
```

```
    a: float,
    b: float,
    c: float,
    d: float
)
```

Args ——=

a Coefficient for the third degree term.
b Coefficient for the second degree term.
c Coefficient for the first degree term.
d Coefficient for the constant term.

22.1.1.1 Methods

Method **get_function**

```
def get_function(
    self
)  -> Callable[[float], float]
```

Makes a callable function from the coefficients

Gives the function $f(t) = at^3 + bt^2 + ct + d$

Returns ——= f : A callable function float $\to$ float.

Method **solve**

```
def solve(
    self,
    x: Union[float, NoneType] = 0
)  -> Tuple[complex, complex, complex]
```

Solves the equation

$$at^3 + bt^2 + ct + d = x$$

by depressing the coefficients into a general cubic and using the substitution $t = u - \frac{b}{3a}$.

Args ——=
x : The right hand side of the equation

Returns ——= A tuple of the three solutions, either real or complex

23 Module `src.util.intervaled_value`

23.1 Classes

23.1.1 Class `IntervaledValue`

```
class IntervaledValue(
    value: float,
    interval_size: float,
    normalized_value: float = 0.5
)
```

Intervaled Value

Holds an updateable value on a static interval. This class is for continuosly updating a value, on some static interval that is set on instantiation.

Example usage:
iv = IntervaledValue(value=3, interval_size=10, normalized_value=0.3)
iv.set_value(100)
iv.get_value() -> 10.0
iv.get_normalized_value() -> 1.0

The class will instantiate an interval of specified size, around the given value. The value is placed on the interval as per normalized_value.

Example usage: IntervaledValue(value=3, interval_size=10, normalized_value=0.3) will instantiate an object that holds the value 3.0 on the range [0, 10].

Args —=

value The initial value to be held.
interval_size Size of the interval.
normalized_value A normalized representation of value.

23.1.1.1 Static methods

Method fit_to_interval

```
def fit_to_interval(
    value: float,
    lower_bound: float,
    upper_bound: float
)  -> float
```

Maps a value from < inf, inf > -> [lower_bound, upper_bound]

If value is already on the interval [lower_bound, upper_bound], then this function returns the same value.

If the value is outside the interval, it returns the lower or upper bound, corresponding to (value < lower_bound) or (value > upper_bound) respectively.

Example: `fit_to_interval(-1, 0, 1) -> 0`

23.1.1.2 Methods

Method **set_value**

```
def set_value(
    self,
    value: float,
    invert=False
)  -> NoneType
```

Sets the held value inside the static interval.

Args —=

value A new value to update the held value with. If the value is outside the interval, it sets self.value to the lower or upper bound, corresponding to (value < lower_bound) or (value > upper_bound) respectively.
invert If True, inverts the given value on the interval.

Method **get_normalized_value**

```
def get_normalized_value(
    self,
    invert=False
)  -> float
```

A normalized representation of the held value.

Args —=
invert : If True, the inverts the return value, i.e 0.25 -> 0.75

Returns —= A normalized ([0.0, 1.0]) representation of the held value.

Method **get_value**

```
def get_value(
    self,
    invert=False
)  -> float
```

Gets the held value.

Args ——=
invert : If True, inverts the value on the interval.

Returns ——= The held value (is contained on the interval).

24 Module **src.util.landmark_utils**

Landmark utils

Collection of functions related to landmarks, specifically hand landmarks and collections of hand landmarks.

24.1 Functions

24.1.1 Function **thumb_index_intersection**

```
def thumb_index_intersection(
    hand_landmarks: Iterable[mediapipe.framework.
       formats.landmark_pb2.NormalizedLandmark]
)  -> Tuple[float, float]
```

Finds intersection between thumb and index finger.

Creates a 2d-line for both thumb and index finger, and returns the intersection of the two lines

Args ——=
hand_landmarks : landmarks for a single hand.

Returns ——= (x, y) coordinate for the point of intersection

24.1.2 Function **get_fingertip**

```
def get_fingertip(
    hand_landmarks: mediapipe.framework.formats.
       landmark_pb2.NormalizedLandmarkList
)  -> Tuple[float, float, float]
```

Fingertip coordinates of the provided hand landmarks.

Args ——=
hand_landmarks : A list of landmarks for one hand.

Returns ——= (x, y, z) fingertip coordinates of the provided hand.

24.1.3 Function **calc_palm_center**

```
def calc_palm_center(
    hand_landmarks: mediapipe.framework.formats.
       landmark_pb2.NormalizedLandmarkList
)  -> numpy.ndarray
```

Approximates center of palm by way of a triangle centroid.

Args ——=
hand_landmarks : A list of landmarks for one hand.

Returns ——= A 3d-vector of x-y-z coordinates representing the palm center.

25 Module `src.util.math_utils`

Math utils.

Collection of mathematical functions used in various parts of this project.

25.1 Functions

25.1.1 Function `angle_between_vectors`

```
def angle_between_vectors(
    u: numpy.ndarray,
    v: numpy.ndarray
)  -> float
```

Finds the angle between two vectors.

Args ——=

u An arbitrary dimensional vector.
v An arbitrary dimensional vector.

Returns ——= Angle between the vectors in radians, domain : [0, pi]

25.1.2 Function `signed_angle`

```
def signed_angle(
    u: numpy.ndarray,
    v: numpy.ndarray,
    n: numpy.ndarray
)  -> float
```

Signed angle theta between vectors u and v given normal vector n.

$$\sin\theta = \frac{(\boldsymbol{n} \times \boldsymbol{u}) \cdot \boldsymbol{v}}{|u||v|}.$$

$$\theta = \begin{cases} \arccos \frac{u \cdot v}{|u||v|}, & \text{if } \sin\theta \geq 0; \\ 2\pi - \arccos \frac{u \cdot v}{|u||v|}, & \text{if } \sin\theta < 0. \end{cases}$$

Args ——=

u An arbitrary dimensional vector.
v An arbitrary dimensional vector.
n Plane normal for vectors u & v.

Returns ——= Signed angle in radians between the two vectors u and v, given the orientational normal vector n. Domain [-pi, pi].

25.1.3 Function `to_unit_vector`

```
def to_unit_vector(
    v: numpy.ndarray
)  -> numpy.ndarray
```

Calculate unit vector corresponding to input vector.

Does not support zero vector.

Args ——=
v : An arbitrary dimensional vector.

Returns ——= The unit vector corresponding to the vector v.

Raises ——= AssertionError : If the given vector is a zero vector.

25.1.4 Function `transform_length`

```
def transform_length(
    v: numpy.ndarray,
    new_length: float
)  -> numpy.ndarray
```

Calculate a new vector parallell to the vector v, with length new_length.

Args ——=

v An arbitrary dimensional vector. Zero vector not supported.
new_length The desired length for the returned vector. Must be non-negative.

Returns ——= A vector of same shape and direction as v, with magnitude equaling new length.

Raises ——= AssertionError : If the given v is a zero vector.

25.1.5 Function `landmark_to_vector`

```
def landmark_to_vector(
    lm: mediapipe.framework.formats.landmark_pb2.
       NormalizedLandmark
)  -> numpy.ndarray
```

Convert landmark to a vector

Args ——=
lm : A normalized landmark.

Returns ——= A 3d-vector corresponding to the input landmark.

25.1.6 Function `landmarks_to_vectors`

```
def landmarks_to_vectors(
    lms: Iterable[mediapipe.framework.formats.
       landmark_pb2.NormalizedLandmark]
)  -> numpy.ndarray
```

Convert an iterable of landmarks to an array of corresponding vectors.

Args ——=
lms : An iterable of normalized landmarks.

Returns ——= An array of vectors corresponding to the landmarks in lms.

25.1.7 Function `line_intersection`

```
def line_intersection(
    a1: numpy.ndarray,
    a2: numpy.ndarray,
    b1: numpy.ndarray,
    b2: numpy.ndarray
)  -> numpy.ndarray
```

Intersection between two lines from four points.

Constructs two lines in the xy-plane from a1 -> a2, b1 -> b2, and finds the intersection between these two.

Args ——=
a1, a2: Two corresponding points as np.arrays of length 2, forms the first line.
b1, b2: Two corresponding points as np.arrays of length 2, forms the second line.

Returns ——= The intersection point between the constructed lines as a np.array of length 2. Parallel lines generates the return value np.array([np.nan, np.nan]) (see below).

Raises ——= UserWarning : For a set of points that gives parallel lines (no intersection to be found).

25.1.8 Function `normalized_to_pixel_coordinates`

```
def normalized_to_pixel_coordinates(
    normalized_x: float,
    normalized_y: float,
    width: int,
    height: int
)  -> Union[NoneType, Tuple[int, int]]
```

Converts a normalized value pair to pixel coordinates.

Args —=

`normalized_x` x-coordinate, must be in domain [0, 1]
`normalized_y` y-coordinate, must be in domain [0, 1]
`width` Width of image.
`height` Height of image.

Returns —= An (x, y) tuple of pixel coordinates. Note that the max for the return values is one less than width / height.

25.1.9 Function `get_extreme_points`

```
def get_extreme_points(
    points: Iterable[numpy.ndarray],
    width: int,
    height: int
)  -> Tuple[Tuple[int, int], Tuple[int, int]]
```

Get corners of provided points.

Given an iterable of normalized points together with image width and height, returns pixel coordinates of the outer points.

Args —=

points Normalized points of with x, y coordinates at 0th and 1st index respectively.
width Width of the image that the return value should be fitted to.
height Height of the image that the return values hould be fitted to.

Returns ——= A tuple of two tuples : (top_left, bot_right) = ((x_min, y_min), (x_max, y_max)).

These corners gives a bounding box surrounding the provided points. Note that origin is located at top left.

25.1.10 Function `centroid`

```
def centroid(
    triangle: Tuple[numpy.ndarray, numpy.ndarray,
       numpy.ndarray]
)  -> numpy.ndarray
```

Calculate centroid of a triangle.

The centroid of a triangle is the point of intersection of its medians.

Args ——=
triangle : Three-tuple of the triangle corners as numpy arrays. The arrays must have the same length >= 2

Returns ——= Centroid of the triangle as a np.ndarray.

25.1.11 Function `cube_root`

```
def cube_root(
    n: Union[float, complex]
)  -> Union[float, complex]
```

Computes cubic root of a real/complex number while maintaining its sign

When raising negative numbers to fractional powers, which in essence is a multivalued function, python favors the priciple solution (that is the solution with the largest real part), which results in complex results for negative real inputs.

This is not what we want in this case though, we want the cube root of a real to stay real. Ex: cube_root(-1) = -1. This is easily achieved with this function.

25.1.12 Function `solve_general_cubic`

```
def solve_general_cubic(
    a: float,
    b: float,
    sub: float
)  -> Tuple[complex, complex, complex]
```

Solves general cubic polynomial

Given a and b, the coefficients of a depressed cubic polynomial on the form $t^3 + at + b$, and the substitution used to obtain the depressed form, it will find the three roots of the polynomial.

By using either Cardano's formula or the trigonometric cubic equation, we garatuee three solutions. With cubics stemming from bezier curves we can further guarantee a real solution on $[0, 1]^2$.

Args ——=

a coefficient of the linear term
b constant term
sub the lambda value used for substitution when calculating the depressed form, $t = u - lambda$

Returns ——= A tuple of three roots of the equation, one is guaranteed to be real, the others are either complex or real.

Acronyms

CLI Command Line Interface. 7, 59

CPU Central Processing Unit. 65, 67, 68

e2e End-to-End. 7, 59

GPU Graphical Processing Unit. 24, 47, 65–69

HD (1920 x 1080 pixels). 66, 67

ML Machine Learning. 11, 24, 26, 28

SD (1280 x 720 pixels). 66, 67

Glossary

binary crossentropy Also known as log loss. A loss function that can be used for evaluating the probability output of a classifier, as opposed to just it's discrete output. 21

bounding box The smallest rectangle that can be drawn around a set of points, such that all the points are inside it, or exactly on one of its sides. The sides of the rectangle are parallel to either the x or the y axis. 21, 26

confusion matrix A visualization of the performance of a classification algorithm / model. In our examples, each row represents an actual class, and each column the predicted class. More on this and classification metrics is available in (Powers, 2008) . 32, 36–38, 42, 51, 61

cuda An application programming interface for parallel computing. . 65

inference The process of using a trained ML model to make predictions on some given input data. 7, 65–68

interpolate From the Cambridge Dictionary (2021): "to add a number or item into the middle of a series, calculated based on the numbers or items before and after it". In the context of this project, this usually entails finding some function that goes through a set of points. This function can then be used to estimate new points between the given points. 45

IoU The intersection over union (IoU) is a popular evaluation metric for object detection. In the case of predicting bounding boxes, let's say you have a ground truth box A and a predicted box B. The IoU is given by the area of overlap between A & B divided by the area of union between A & B.

$$IoU = \frac{|A \cap B|}{|A \cup B|}$$

. 20

keras An open-source library for artifical neural networks, that provides a layer of abstraction on top of tensorflow . 24

letterboxing The process of adding black side bars to a picture to make it fit a wider or taller frame . 35

loss function When training ML models, you need someway to optimize performance. One aspect of this is done by using loss functions, which take the output from the model compared to the ground truth target, and maps this to a number representing the error of the prediction. 20, 21

model We have used the term model as an abbreviation for the more specific 'machine learning model'. From an AWS web article: "The process of training an ML

model involves providing an ML algorithm (that is, the learning algorithm) with training data to learn from. The term ML model refers to the model artifact that is created by the training process." (*Training ML Models*, n.d., para. 5) . 11, 19–26, 31, 35, 49–53, 58, 59, 65, 69

MVP A minimum viable product (MVP) is a first version of a new product that satisfies the minimum of requirements. Typically used in agile processes, where after inception, the MVP is continuously improved in iterative increments. 17, 25, 34, 58

overfitting The production of an analysis which corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably. (*Overfitting*, 2021). 23

ROC-curve A receiver operating characteristic curve is a performance metric that illustrates the classification capability of a binary classifier. (Gneiting & Vogel, 2018). 22

tensorflow A wide-spread open source platform for machine learning. . 65

References

Bambach, S., Lee, S., Crandall, D. J., & Yu, C. (2015, December). Lending a hand: Detecting hands and recognizing activities in complex egocentric interactions. In *The ieee international conference on computer vision (iccv).*

Company information. (2020). Huddly AS. Retrieved 22-05-22, from `https://www.huddly.com/content/uploads/2020/01/Company-Information.pdf`

Dictionary, C. (2021). *Interpolate.* Retrieved 2021-05-21, from `https://dictionary.cambridge.org/dictionary/english/interpolate`

Fan Zhang, G. S., Valentin Bazarevsky. (2019). *Model card, mediapipe hands.* Google. doi: https://mediapipe.page.link/handmc

Fan Zhang, V. B. (2019, August 19). *On-device, real-time hand tracking with mediapipe.* Google Research. Retrieved from https://ai.googleblog.com/2019/08/on-device-real-time-hand-tracking-with.html

Gneiting, T., & Vogel, P. (2018). *Receiver operating characteristic (roc) curves.*

Hazewinkel, M. (1997). *Encyclopaedia of mathematics: Supplement* (No. v. 1). Springer Netherlands. Retrieved from https://books.google.no/books?id=3ndQH4mTzWQC

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, *abs/1512.03385*. Retrieved from http://arxiv.org/abs/1512.03385

Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., ... Murphy, K. (2016). Speed/accuracy trade-offs for modern convolutional object detectors. *CoRR*, *abs/1611.10012*. Retrieved from http://arxiv.org/abs/1611.10012

Lin, T., Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., ... Zitnick, C. L. (2014). Microsoft COCO: common objects in context. *CoRR*, *abs/1405.0312*. Retrieved from http://arxiv.org/abs/1405.0312

Marin, G., Dominio, F., & Zanuttigh, P. (2015a). Hand gesture recognition with jointly calibrated leap motion and depth sensor. *Multimedia Tools and Applications*, *75*, 14991-15015.

Marin, G., Dominio, F., & Zanuttigh, P. (2015b, 01). Hand gesture recognition with leap motion and kinect devices. *2014 IEEE International Conference on Image Processing, ICIP 2014*, 1565-1569. doi: 10.1109/ICIP.2014.7025313

Mozilla smil. (2011). Mozilla. Retrieved 23-05-22, from https://github.com/ehsan/mozilla-history/blob/master/content/smil/nsSMILKeySpline.cpp

Overfitting. (2021). Oxford English Dictionary. Retrieved 2021-05-22, from https://www.lexico.com/definition/overfitting

Powers, D. (2008, 01). Evaluation: From precision, recall and f-factor to roc, informedness, markedness & correlation. *Mach. Learn. Technol.*, *2*.

Training ml models. (n.d.). Amazon Web Services. Retrieved from

https://docs.aws.amazon.com/machine-learning/latest/dg/training-ml-models.html

Zhang, F., Bazarevsky, V., Vakunov, A., Tkachenka, A., & Sung, G. (n.d.). *Mediapipe hands.* Retrieved from https://google.github.io/mediapipe/solutions/hands.html

Zhang, F., Bazarevsky, V., Vakunov, A., Tkachenka, A., Sung, G., Chang, C., & Grundmann, M. (2020). Mediapipe hands: On-device real-time hand tracking. *CoRR*, *abs/2006.10214*. Retrieved from https://arxiv.org/abs/2006.10214

Zhong, Y., Wang, J., Peng, J., & Zhang, L. (2018). Anchor box optimization for object detection. *CoRR*, *abs/1812.00469*. Retrieved from http://arxiv.org/abs/1812.00469

Zucker, I. J. (2016). The cubic equation - a new look at the irreducible case. Retrieved from https://doi.org/10.1017%2FS0025557200183135