Håkon Larsen Eckholdt

# Pose Estimate for Quadcopter Using Stereo Camera

Master's thesis in MTTK
Supervisor: Tor Engebret Onshus
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Håkon Larsen Eckholdt

# Pose Estimate for Quadcopter Using Stereo Camera

Master's thesis in MTTK
Supervisor: Tor Engebret Onshus
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

## Preface

The basis for the work done in this project is the master thesis written by Bendik Bjørndal Iversen[17], and the accompanying source-code for a visual odometry (VO) algorithm that to some degree manages to track the position of the camera. The source code was made for a Jetson TX1, which is a powerful computer the size of a small computer chip. I had access to this and a stereo-camera called Duo M, which account for the camera which position the algorithm is to track, during the entire project. However, the algorithm and the Duo M camera did not work when I received it. This was due to changes in the libraries and drivers between the time Iversen delivered his master and the time I received the code. The theory described in this master is either acquired from sources online, cited where used, or from different computer vision related subjects I have completed before or parallel to working on this project (TDT4265 and TTK21, respectively). Most of the theory in this master was compiled during the project I completed this fall (TTK4550). Apart from this the project was not used.[9]

The list of hardware I have had access to and use for is listed below, and it was received at the very beginning of the project.

- Jetson NVIDIA TX1 with accompanying developer board.
- Power supply, SD-card and peripheral devices such as keyboard and screen.
- Duo M stereo camera.
- Nordic BLE dongle.
  - This was rendered obsolete once the server changed its communication protocol.
- Desktop needed for setup.
- Laptop used for portable programming. (Ubuntu 16.04)

During this project I have had help from my supervisor in determining the right path of my work, on what areas to focus my time and who to contact when special expertise was needed. On account of that I received important help from the department engineer at the Department of Engineering Cybernetics when the Duo M camera drivers were no longer working on the TX1.

## Abstract

This master thesis is a continuation of an ongoing Lego-robot project, comprised of several robots that cooperate on solving a maze. Each robot has different sensors and information, giving them different abilities to orient themselves in the maze. Combing the information from all the robots in a Java server application should make all the robots able to solve the maze. The drone part of the Lego project focus on constructing a drone that maps the labyrinth from above, and reports this back to the server. At this stage in the process, the main task is to implement a system to replace GPS (global positioning system) localization, as the maze exists in an indoor, GPS isolated environment. This master is a continuation of the work performed by Bendik Iversen[17], which consists of a visual odometry algorithm that to some extent manages to track the location of a stereo camera, and provide basic server communication. This master thesis aims to increase the performance of this algorithm, making it able to locate the drone's position within an acceptable distance of error.

The main workload in this master has been to change the system to accept a dataset called KITTI as the image stream, rather than the Duo M stereo camera. This was done in order to get consistent results that were not contaminated by inaccurate lighting, movement or other factors. On this altered system I performed several tests of different functions used by a promising visual odometry algorithm, the SOFT (stereo odometry based on careful feature selection and tracking) algorithm. Based on the results from these test it looks promising that the results on the KITTI dataset are good enough to give input to a drone, especially considering that the VO algorithm only aims to replace the GPS portion of the drones hardware, and not the IMU(Inertial measurement unit) portion (which takes care of the high frequency pose adjustment).

The system should still be tested in a real test scenario, on a multirotor using the Duo M stereo camera. Until this is done, there is no way to know for sure that the SOFT algorithm in combination with the Duo M camera provides high enough accuracy. What we do know, is that the updated VO works much better than the original VO, based on the results found and discussed in this thesis.

## Sammendrag

Denne masteroppgaven er en fortsettelse på et pågående Lego-robot prosjekt, som består av flere roboter som samarbeider om å løse en labyrint. Hver robot har ulike sensorer og informasjon, og ved å samle informasjonen fra alle robotene i en Java-server applikasjon ønsker man å gi alle robotene muligheten til å løse labyrinten ved samarbeid. Droneaspektet ved Lego-prosjektet setter søkelys på det å lage en drone som kartlegger labyrinten ovenfra, og rapporterer dette tilbake til serveren. Foreløpig er hovedoppgaven å implementere et system som kan erstatte GPS lokasjon, siden labyrinten er innendørs. Denne masteroppgaven er en fortsettelse på arbeidet utført av Bendik Iversen [17], som består av en VO algoritme som til en viss grad klarer å lokalisere hvor stereokameraet befinner seg, og oppretteholder enkel serverkommunikasjon. Denne masteren har som mål å øke presisjonen og frekvensen til algoritmen, slik at den blir i stand til å lokalisere dronen innen en akseptabel feilverdi.

Hovedarbeidet i denne masteren har vært å restrukturere systemet så det aksepterer bilder fra KITTI-datasettet, istedenfor bildestrømmen som kommer fra Duo M stereokameraet. Dette ble gjort for å oppnå konsekvente resultater, uten påvirkning fra dårlige lysforhold, ukonsekvent bevegelse eller andre faktorer som forringer resultatene. På det nye systemet ble det utført en rekke tester av ulike kombinasjoner av funksjoner som alle er en del av en veldig lovende visuell odometrialgoritme, SOFT. Basert på resultatene fra disse testene ser det lovende ut at resultatene på KITTI-datasettet er gode nok til å gi input til en drone, spesielt med tanke på at algoritmen kun skal erstatte GPS-posisjonen til dronens egen maskinvare, og ikke IMU-delen (som tar hånd om den høyfrekvente posisjonskorreksjonen).

Det gjenstår fortsatt å teste systemet i et ekte scenario, på en multirotor og med Duo M stereokameraet som bildekilde. Til dette er gjort, er det ingen måte å vite med sikkerhet at SOFT algoritmen kombinert med Duo M gir høyt nok resultat. Det vi derimot vet, er at den nye algoritmen er mye bedre enn den forrige basert på resultatene og diskusjonen i denne masteren.

## Conclusion

Using the KITTI dataset for testing proved useful when testing the different algorithms during this master. It made for a stable and permanent test environment, and the results are applicable to the actual system where the DUO M is used instead. The only downside to this dataset is that it does not provide enough movement in the Z axis to determine the algorithms performance in this direction, and that it only tests which algorithm is best, not how well it will be once on the drone.

Testing the different algorithms proved why the different functions of the SOFT algorithm are important and gave insight on which of them should be implemented into the finished system. The bucketing function resulted in a much higher framerate in all tests where it was active. The FPS (frames per second) was almost doubled on all accounts. Not only did the framerate increase, but due to evenly distributed features, the algorithm accuracy increased. Bucketing is the most important function and should be implemented.

The “removal of invalid points” function caused both a slightly higher FPS and accuracy. This function should also be added, as any benefit in accuracy that does not slow down FPS is welcome. Because the function will make more of a difference in scenes where different features are more similar, as this makes them prone to being tracked to the wrong place during circular matching, the function adds robustness. Similar features might prove to be the case when using the DUO M in the labyrinth environment, as the maze environment is a repeating pattern.

Point retention alone does not provide a better pose estimate and reduce the FPS severely. However, if combined with bucketing and invalid point removal, it provides a much higher accuracy than when not used. This is a good feature if combined with bucketing, as it gives bucketing function the ability to prioritize features based on their age. The increased accuracy is worth a slight drop in FPS, considering that the FPS is high when bucketing is active.

The best results are when using all the features from the SOFT algorithm, and the very best result comes from using the non-GPU (graphics processing unit) accelerated FAST (Features from accelerated segment test) feature extractor. This does sacrifice a few FPS, but it makes up for it in accuracy. With a total error of 5 159 it is the clear winner, and the FPS of 17.51 should still be enough for this project.

When using the Duo M camera combined with the tracking algorithm based on SOFT, and the triangulation and pose estimation used for the KITTI dataset, the results are terrible. This comes down to calibration. The camera used to make the KITTI dataset was much larger than the small Duo M camera. The results found in this thesis prove that the original algorithm performs much worse than the SOFT algorithm when calibrated correctly. Using the original triangulation and pose estimation from the original algorithm is an alternative solution to manually calibrating the camera, as this automatically calibrates from the Duo API (application program interface).

# Abbreviations

| | | |
|---|---|---|
| API | = | Application Programming Interface |
| BLE | = | Bluetooth Low Energy |
| BRIEF | = | Binary Robust Independent Elementary Features |
| CPU | = | Central Processing Unit |
| CV | = | Computer Vision |
| CISC | = | Complex Instruction Set Computer |
| DoF | = | Degrees of Freedom |
| FAST | = | Features from Accelerated Segment Test |
| FPS | = | Frames Per Second |
| GPS | = | Global Positioning System |
| GPU | = | Graphics Processing Unit |
| HOG | = | Histogram of Oriented Gradients |
| IMU | = | Inertial Measurement Unit |
| I/O | = | Input / Output |
| ORB | = | Oriented FAST Rotated BRIEF |
| RANSAC | = | Random Sample Consensus |
| rBRIEF | = | rotation-aware BRIEF |
| RISC | = | Reduced Instruction Set Computer |
| SAD | = | Sum of Absolute Differences |
| SDK | = | Software Development Kit |
| SIFT | = | Scale Invariant Feature Transform |
| SOFT | = | Stereo Odometry based on careful Feature selection and Tracking |
| SVO | = | Fast Semi-Direct monocular Visual Odometry |
| USB | = | Universal Serial Bus |
| VO | = | visual odometry |

# Contents

# 1 Introduction

## 1.1 Background and previous work

### 1.1.1 Lego-robot project

The visual odometry algorithm described in this thesis is part of a collaboration between several students, where the final goal is to get several robots to cooperate in mapping and escaping a maze. The algorithm created in this master will be used on a drone which purpose is to fly above the maze, mapping the walls of the maze. The drone will need to know its relative position and orientation in order to stay safely in the air and provide useful insight into where the different maze-walls and robots are located. The information of both the position of the drone and the mapped wall-segments should be sent to a server running a Java-application using BLE (Bluetooth Low Energy). The server will then make a map of the maze by piecing together and refining information gathered by the drone, as well as the other robots.

## 1.2 Problem Description

Currently the system can find the relative position of the drone with sub-optimal accuracy. The system fails during fast movement, and even when moved slowly the position estimation is drifting. This needs to be solved by a combination of faster framerate and accommodation of larger movement between frames. This master aims to experiment with different setups and algorithms, and then determine whether they make an improvement on the system. The best combination of algorithms will then be implemented, in order to shift the focus of the project from VO to mapping of the maze in the future. I will also give insight into why the algorithms Increase or decrease the overall performance of the system. Communication with the server and other Lego robots, as well as mapping of the wall segments is not the primary goal of this master thesis and will not be addressed. The primary goal is to get the system running and determining what changes can be made to make it capable of flight, and ideally implement those changes. The goal of the master thesis can be summarized into the following milestones:

- Make the system operational on the TX1.
- Perform benchmark of current system.
- Implement SOFT algorithm and compare to original VO algorithm.
- Implement best combination of functions.

The milestones will be done sequentially and are in prioritized order by how much of an impact they are expected to make on the performance, or by dependencies on earlier tasks.

# 2 Theory

Odometry is the technique of using data from a sensor to determine the position of an object as a result of the change in position over time[1]. Traditionally this is performed using some sort of wheel or rotational encoders dragging or moving across a surface. For instance, a wheel might be mounted on the back of a dogsled in order to know the distance the sled has traveled in a direction. This is, however, prone to errors as the wheel might slip and otherwise react unpredictably to changes in climate or other external stimuli. In this thesis the reason why traditional odometry is not ideal is obvious: the drone will never be in contact with any surface during operation. This is where visual odometry comes into play.

## 2.1 Visual odometry

Visual odometry, much like traditional odometry, is a means to establish a location by measuring the change in position over time. Unlike regular odometry, visual odometry does not rely on tracing the movement of moving parts across a surface, but rather the movement that had to occur to account for the difference between two images taken at two different instances of time. This process is also being used on ground-based vehicles, since you remove the uncertainty that arise from drift caused by slipping in regular odometry. As computers get faster and methods in artificial intelligence excel, the effectiveness of these methods increase. An essential part of visual odometry is the ability to recognize objects or features in multiple different images, so that you can calculate how the object has changed in size, position or orientation from one image to the next. This is done using algorithms from the field of computer vision.
A drone, such as a quadcopter, has six degrees of freedom (6 DoF). Up/down, right/left, forwards/backwards, yaw, pitch and roll, as you can see in Figure 1. The visual odometry algorithm will have to know how the camera moves in all these directions.

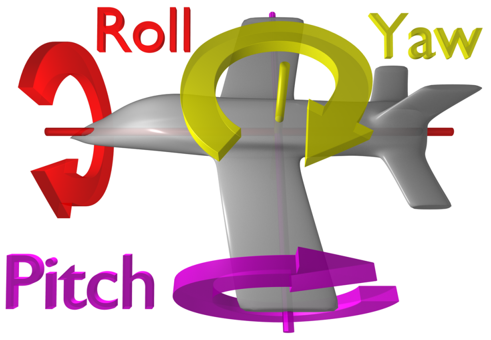


Figure 1: Roll, yaw and pitch of an air-frame [33]

## 2.2 Direct vs Indirect

When referring to visual odometry there are two main methods to achieve the goal of estimating the relative pose of the camera (the translation and rotation between the two images), namely the direct method, and the indirect method. They have a significant difference in the approach of estimating movement, and in applications like this project there have been a vast majority of direct algorithms, and that is indeed the case for the previous work done on this very project.

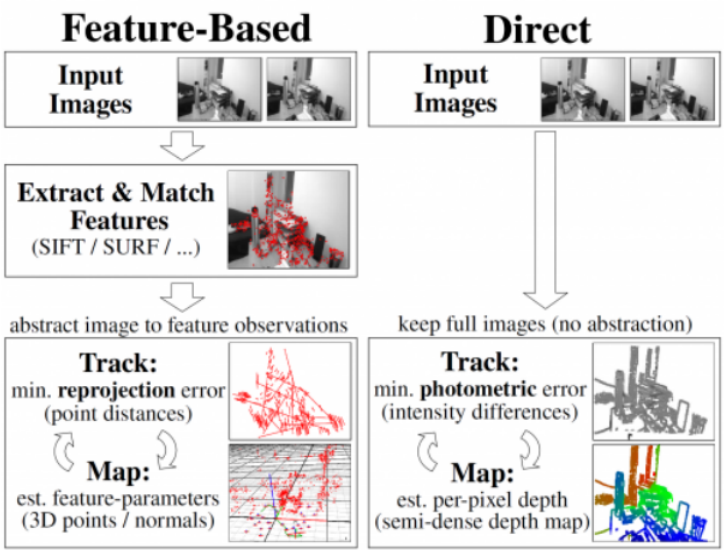


Figure 2: Comparison between Feature-Based and Direct method of estimating position [14]

The main differences are illustrated in Figure 2. Where you in the indirect method (appropriately named Feature-Based in the figure) you extract features before estimating the movement of said features. The direct method, on the other hand, compares the individual pixels without attention to specific features. The following gives a more comprehensive explanation of the different methods and their strengths and weaknesses.

### 2.2.1 Indirect method

The indirect method is based on the concept of features. A feature is the combination of a key-point and a descriptor of said key-point. A key-point is usually a corner or another segment of an image which is easily distinguished from the rest of the image and would most likely be found again in an image with a slightly different pose. A feature descriptor is some way of describing the point, or rather the points immediate surroundings, making it possible to distinguish different points from one-another. This makes it possible to match points from one image to another.

The indirect method is all about minimizing the geometric error that occur between the triangulated matching points, and the assumed pose of the camera. When this is optimized over the pose of the camera the correct pose is approximated. This can be written as:

$$T_{cw}^* = \underset{T_{cw}}{\operatorname{argmax}} \sum_j ||\pi(T_{cw}\tilde{x}_j^w) - u_j||^2$$

The first element in the sum, $\pi(T_{cw}\tilde{x}_j^w)$, is the prediction part of the equation. This predicts where the points would be on the 2D frame on the image plane given a pose $T_{wc}$. When minimizing the geometric error of this prediction with the actual value $u_j$ the ideal correct pose is reached. This is the pose of the world as seen by the camera, what we want is the opposite:

$$T_{wc}^* = \underset{T_{wc}}{\operatorname{argmax}} \sum_j ||\pi(g(T_{wc}, x_j^w)) - u_j||^2$$

As mentioned in the beginning of this section, we need features in order to perform this tracking of the camera movement. The different parts that goes into this will be described in the following sections.

**Feature detection** A feature or a key-point, can be classified as a small area of an image that can be found in different images, and the strength of these points depend on how unambiguous they are. If a key-point is found in two images there should be a relatively small distance of potential error. Imagine a key-point located at a white spot on a white wall in two images. There is almost no way of determining where on the wall the points are located, and there is a large amount of uncertainty. This is a bad point. If the point is on an edge the result is better, as it is along a line, but it could still be anywhere on that line. The best key-points are those that are in corners. They do not need to be corners in the traditional sense where two edges clearly meet, but rather a point where the image intensity has a clear gradient in all directions. The latter is the key-points we aim to find when using the indirect method. These different points can be seen in Figure 3.

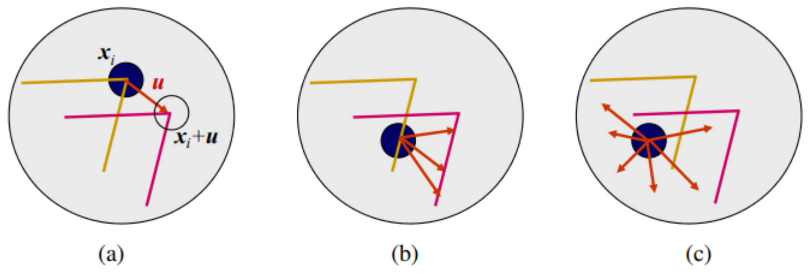


Figure 3: The three main categories of key-points. (a) Corner (b) Edge (c) Flat surface [27] [pp. 210]

There are many different methods for finding key-points, or features. There are multiple considerations when choosing the detector for a project. Speed and accuracy are the most apparent, but there are other qualities such as scale invariance, rotational invariance and affine invariance. Ideally, we would achieve a high enough framerate so that the difference between two images is small enough for these considerations to be a mere luxury and not necessary for a good result. This is not necessarily the case, and especially rotation is a likely distortion as the drone could be expected to roll in order to keep its position.
Among the many feature detectors, we find the Canny edge detector, Harris corner detection and FAST. The FAST algorithm will be presented in more detail below, as it is the detector used in this system.

**FAST - Features from accelerated segment test**
FAST is a corner detector (as opposed to edge or blob detector) which is computationally very efficient. This is the main reason why it is widely used for real time applications, including this project. It is a natural choice as the framerate might become an issue with the current algorithm when used on a real drone. It is also the feature detector used by ORB(Oriented FAST rotated BRIEF) algorithm, mentioned in 2.2.1. The FAST algorithm[30] works by addressing a potential point which is to be classified as a point or not, $p$. Around this potential point there is created a Bresenham circle consisting of 16 pixels (see Figure 4 for reference). A threshold value $t$ which represents the change in luminosity between the luminosity at the interest pixel $p$ ($I_p$) and the luminosity at one of the 16 points in the circle needed for the difference to be considered significant is also defined. The last thing needed is the number of pixels in the circle who needs to be either sufficiently darker or lighter than the potential point $p$, this number is $n$. With these values defined, the process is as follows:

- Define the point $p$ you want to categorize.
- Check if there consist a set of $n$ points in the circle surrounding $p$ that all fulfill $I_n \leq I_p - t$, or all fulfill $I_n \geq I_p + t$, that is, that all points are either dark enough or light enough.
- If such a set exist $p$ is a corner.

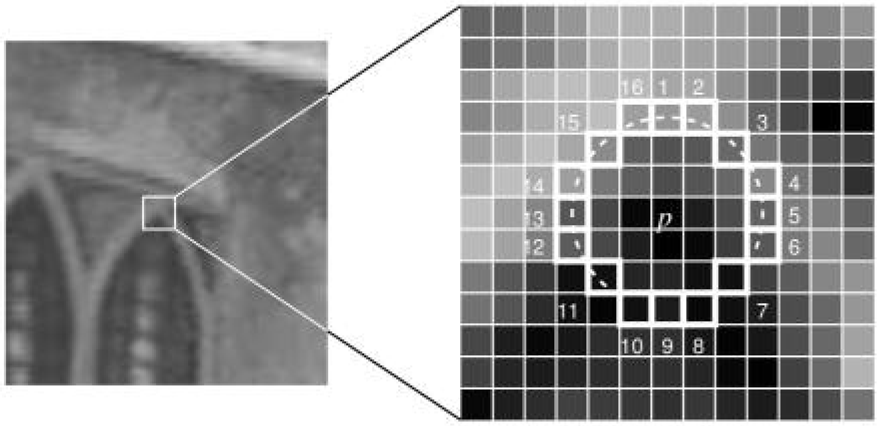


Figure 4: The circle that is evaluated in order to determine if a point is a corner in the FAST algorithm [30]

An attempt at making this even faster is to include a pre-test that first check if points 1, 5, 9 and 13 has the property that three or more are all brighter or all darker than the luminosity of $p$ plus the threshold. If that is not the case the point can immediately be discarded as not being a corner. This saves the algorithm up to twelve comparisons. However, this poses a few problems:

- It can not be done if $n < 12$, as you can't discard a corner based on just four comparisons when the set-size is that small.

- The efficiency of the test is dependent on which pixels are compared first.
    - These two first problems can be solved using machine learning.
- Multiple features can be detected adjacent to each other.
    - This can be solved using Non-maximal suppression which essentially is to discard the feature with the least average luminosity difference between the point $p$ and the points in the circle.

**Feature Matching** There are two main ways of determining the correspondence of features from an image to another, feature matching and feature tracking (2.2.1). The main difference between these is that feature matching finds features in two images, and then match them in order to estimate movement by minimizing the geometric error. This method is the cleanest form of the indirect method, and an example is shown in Figure 5. Feature tracking on the other hand, only finds features in the first image and then tracks them to the next using a local search technique.
The goal of feature matching is to match features found in image $k-1$ to features found in the current image $k$. This relies on some sort of method to distinguish the different key-points, other than their relative position, as that by all accounts have changed. The previous key-point algorithm FAST does not provide that, since it simply determines what a corner is. In order to match point we also need a **feature descriptor**. This is something that, ideally, uniquely identifies a point based on its soundings. If the feature detector does not provide this, we need to add one before matching. There exist multiple, such as SIFT (Scale Invariant Feature Transform), HOG (Histogram of Oriented Gradients) and BRIEF (Binary Robust Independent Elementary Features). The descriptor used in this projects original algorithm is SIFT.

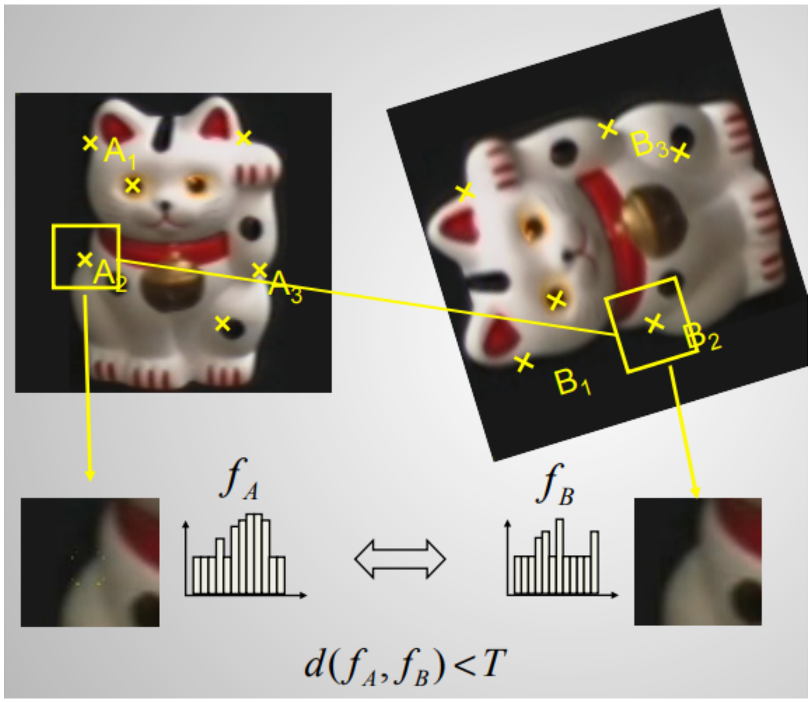


Figure 5: Visualization of feature matching where point A2 and B2 are matched because the difference between the descriptors $f_A$ and $f_B$ are smaller than the threshold value T[13]

**SIFT - Scale Invariant Feature Transform** The SIFT[20] algorithm creates a normalized 128 dimensional feature vector which can be compared to others, usually using a nearest neighbour approach. The algorithm works like this:

- Extract a 16x16 patch around a key-point. Divide this into 16 4x4 windows.
- Inside each 4x4 window, calculate the gradients and place them in a histogram of 8 bins.
- Apply a Gaussian weighing function to make the points farthest away from the key-point make less of an impact.
- Collect the histograms of the 16 4x4 windows into a feature vector of 128 (4x4x8) numbers.
- Normalize the vector. The vector looks similar to $f_A$ in Figure 5, only with 128 values.

**BRIEF - Binary Robust Independent Elementary Features**
The BRIEF method [21] is very fast compared to other descriptors, as it directly calculates a binary string from the image patches around the key-points. Not only are the calculations faster than computing a comprehensive descriptor but using the Hamming distance (Bit-wise difference between strings) is extremely fast to compute on modern CPU's (Central Processing Unit). The bit string, or the descriptor, is comprised of multiple tests within

the patch. The tests are simply a comparison of the luminosity values of two pixels in the patch. This can be expressed mathematically as

$$\tau(\mathbf{p}; x, y) := \begin{cases} 1, & \text{if } \mathbf{p}(x) < \mathbf{p}(y) \\ 0, & \text{otherwise} \end{cases}$$

with $\mathbf{p}(.)$ representing the pixel intensity at a point and $\tau$ the test result. The numbers of test to compute on a patch is optional, but the original paper[21] has good results with both 128, 256 and 512 tests. (32 and 64 bytes). The binary string is comprised of $n_d$ tests like this

$$f_{n_d}(\mathbf{p}) := \sum_{1 \leq i \leq n_d} 2^{i-1} \tau(\mathbf{p}; x_i, y_i)$$

where $n_d$ usually is either 128, 256 or 512. The final matching of descriptors is done using

$$L = \sum_{0 \leq i \leq n_d} XOR(f^1_{n_{d_i}}, f^2_{n_{d_i}})$$

which is the Hamming distance between a patch in image 1 and image 2. If L is smaller than some threshold it is considered a match

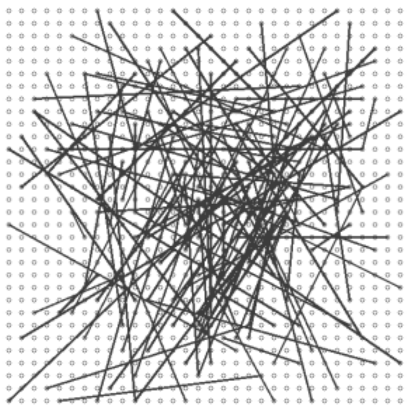


Figure 6: Visualization of tests on in a BRIEF descriptor. Lines stretch between pixels that are to be compared. This particular test distribution is achieved using an isotropic Gaussian distribution.[21]

This was the basics of the BRIEF algorithm, but there are two other aspects to consider, which I will not go further into but should be mentioned.

- The patch should be smoothed out. If this is not done, single pixel-values are used, which are very susceptible to noise.
- The test locations $x_i$ and $y_i$ have many different possibilities, and must obviously remain unchanged between every descriptor. The best results were achieved by using an isotropic Gaussian distribution as seen in Figure **??**
- The BRIEF feature descriptor does not take orientation into consideration like for instance SIFT.

**ORB - Oriented FAST Rotated BRIEF** ORB[10] is, as the name suggests the combination of finding key-points using the FAST2.2.1 algorithm, and then perform feature matching using the BRIEF descriptor. There are some adjustments made to the detection part (FAST) of the algorithm in order to make it more robust. First, they introduce a Harris corner measure in order to maximize the number of key-points that represent corners and not edges. This increases the chances of key-points in one image correctly matching points in the next image. Furthermore, the FAST algorithm is not scale invariant. This is taken care of by adding a scale pyramid and producing features on all the different levels. In addition to these modifications, there was also a need to implement some sort of rotational awareness. This is done by taking advantage of the intensity centroid. The intensity of centroid is determined by taking the moments of the image patch

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y), p, q \in [0,1]$$

, and from that determine the centroid as

$$C = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}})$$

By looking at the moments formula you see that $m_{00}$ simply gives the sum of all non-zero (by intensity) pixels in the patch, or if you like: the area of the patch. The $m_{10}$ and $m_{01}$ is the "mass" of x and y respectively. That implies that $C$ is actually $C = (\overline{x}, \overline{y})$ where $\overline{x}$ and $\overline{y}$ is the average of x and y. The patch orientation is simply determined as $\Theta = \arctan 2(m_{01}, m_{10})$.

Following the key-point detection is the feature matching. In order to achieve this a descriptor is needed. This is where BRIEF comes into play. As mentioned in 2.2.1, the BRIEF descriptor will fail if rotation is introduced. ORB solves this by introducing what they call rBRIEF, Rotation-Aware BRIEF. This is the result of using steered BRIEF, and then running a greedy search on all the possible binary tests. The ones with means close to 0.5, high variance and that also are uncorrelated are chosen by the greedy algorithm. The "steered BRIEF" can be described as the rotated matrix $S_\Theta$, where $S_\Theta = R_\Theta S$, and $S$ is the matrix of all the binary tests in a patch. The rBRIEF allows rotation and thus makes the ORB rotation invariant.

Table 1: Computation of ORB steps

Table showing the computational time on a 640x480 image on an Intel i7 2.8 GHz processor. Results are from the original ORB paper[10].

| **ORB:** | Pyramid | oFAST | rBRIEF |
|---|---|---|---|
| Time (ms) | 4.43 | 8.68 | 2.12 |

As seen in Table 2.2.1 the computational speed on a desktop processor is very fast, and by looking at Table 2.2.1 ORB proves to be vastly faster to compute than comparable methods. It also outperforms SIFT and SURF in the outdoor tests performed by the team behind ORB when it comes to accuracy.

Table 2: ORB vs SURF vs SIFT

Table showing the computational time on an Intel i7 2.8 GHz processor for the ORB, SURF and SIFT algorithm. It was run on 2686 images at 5 scales, averaging about 1000 features on all the tests. Results are from the original ORB paper[10].

| Descriptor | ORB | SURF | SIFT |
|---|---|---|---|
| Time per frame (ms) | 15.3 | 217.3 | 5228.7 |

**Feature Tracking** Feature tracking is another way of determining the motion between two frames. This method relies on the movement to be small enough so that the points does not move out of the scope of the chosen local search method. It could be stated that slow motion is an option in this project, if the multirotor is stabilized through other means. This is the currently implemented solution in the algorithm and it is also part of the circular matching function in the SOFT algorithm.
One way to achieve feature tracking is using **optical flow**[31]. Optical flow is the method of estimating movement of the surroundings, or perceived movement of the surroundings caused by camera-movement, by tracking the movement of features from one frame in time, to the next frame in time. As mentioned in the previous paragraph, there are some assumptions to this method that are potential tripwires if used in this project's application.

- The pixel intensity is assumed constant between successive frames.
- Neighboring pixels have similar motion.
- The motion between successive frames are small.

Optical works on the principle of the optical flow equation. Given the assumptions above you have:

$$I(x, y, t) = I(x + dx, y + dy, t + dt)$$

which using Taylor series expansion can be shown to be

$$f_x u + f_y v + f_t = 0$$

given

$$f_x = \frac{\partial f}{\partial x}; f_y = \frac{\partial f}{\partial y}$$

$$u = \frac{dx}{dt}; v = \frac{dy}{dt}$$

. This can not be solved without another method, like for instance **Lukas-Kandle**. Lucas-Kandle uses the assumption that neighboring pixels move similarly, by looking at a 3x3 pixel square and finding $(f_x, f_y, f_t)$ for these 9 pixels. Using the Least squares fit method which ultimately yields:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix}$$

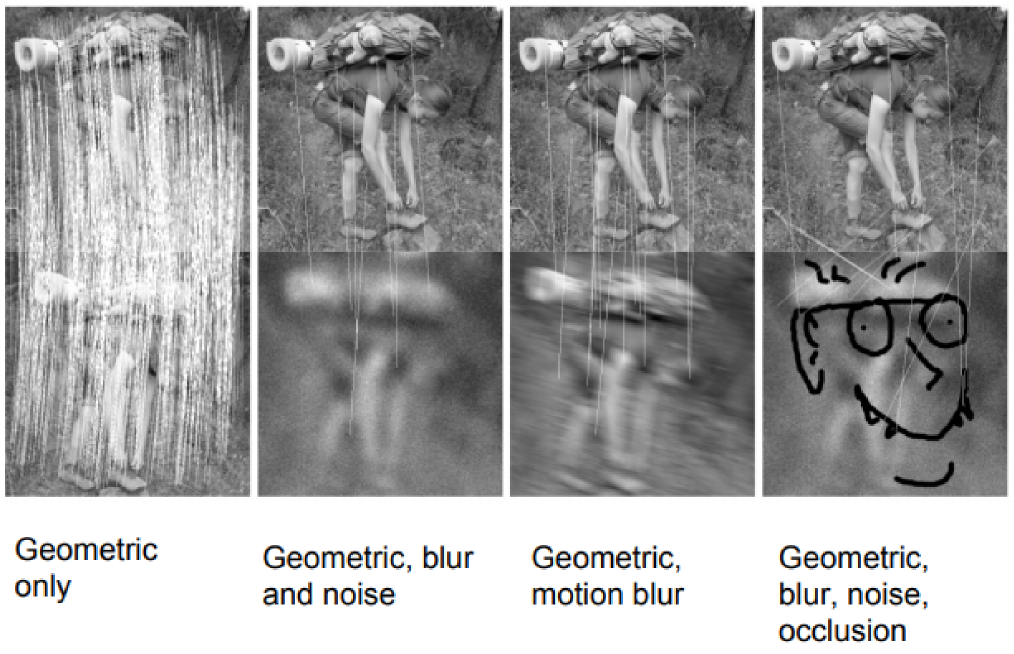
Solving this equation of two unknowns above gives the answer to the problem.

### 2.2.2 Direct method

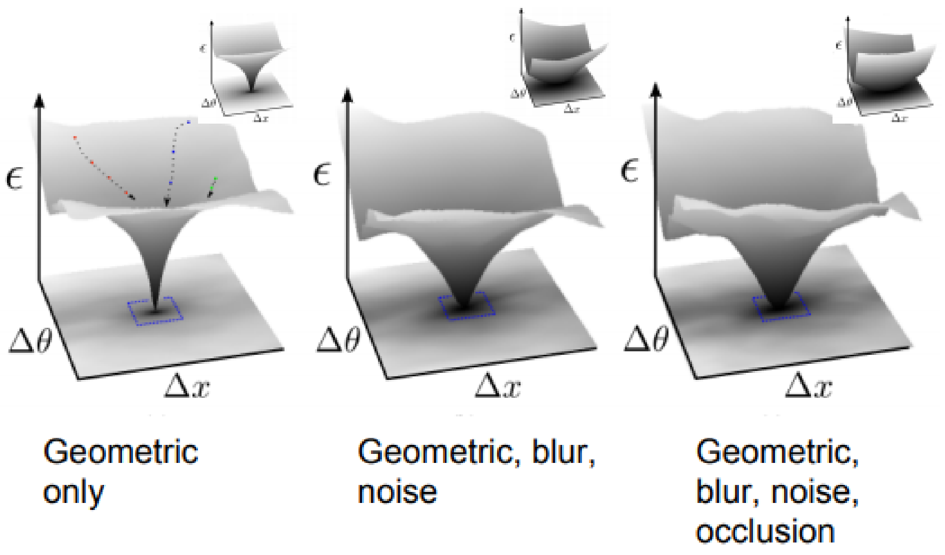
In the previous chapter I presented some of the theory behind the indirect method of estimating movement using consecutive frames of a camera. In contrast to the indirect method, which is based around the concept of features, the direct method work on individual pixels of the entire image. As seen in Figure 2, the direct method skips the step of extracting features and descriptors. The step of estimating the pose over all the pixels is, however, more computationally expensive than doing so over a potentially sparse set of features. This method is not seen as often in settings similar to those of this project due more computational power results in either bigger computers with a higher power consumption. It might in fact prove impossible to create a real-time VO based on this if the computer is not sufficiently powerful. However, there are some advantages to this method. Advantages described in the next paragraph that might be very useful on a moving airborne platform such as the indoor drone in this project.

If there is induced blur, noise, motion blur or other forms of image degradation a feature detector will have a hard time finding features at all or finding features that are good enough to match in a meaningful way. This problem is absent, or at least much less prominent when using the direct method which is based on individual pixel intensity. This is shown in Figure 7. Especially motion blur can be expected as the drone will often be in some sort of movement, even when attempting to stay completely still.

In the direct method you also have the distinction between dense and semi-dense method, which is the distinction between using every single pixel in the image, or only using pixel patches of high gradients. Nevertheless, the main element of the direct method is that the pose is estimated based on pixel intensity directly, as opposed to features. How this is done is presented briefly below.



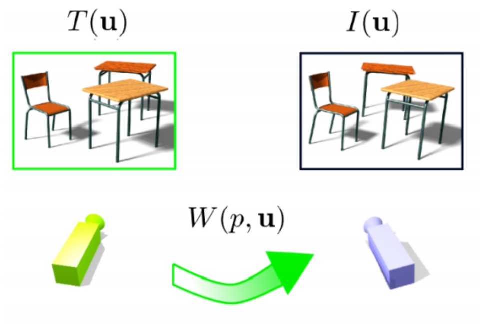
(a) White lines represent features matched when different image degradation's are introduced



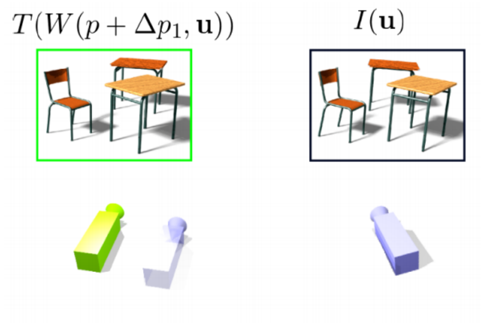
(b) The direct method still manages to find clear global minimum

Figure 7: Illustrations to show that the direct method is much more robust to image degradation such as motion blur, blur and noise. [24]

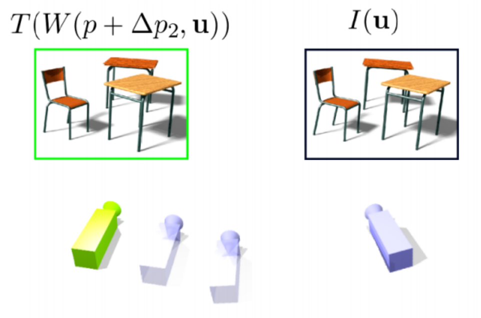
**Direct tracking** Direct tracking relies on minimizing the photometric error $(I_{k-1}(w(d, \mathbf{u}, \mathbf{T})) - I_k(\mathbf{u}))^2$, using the warp function $w(d, \mathbf{U}, \mathbf{T}) = \pi(\mathbf{T}\pi^{-1}(d, \mathbf{u}))$. In simple terms this can be visualized as seen in Figure 8 (a).



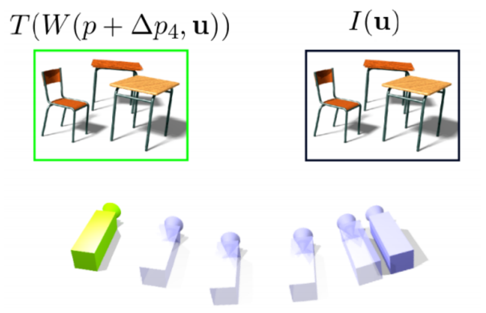
(a) Initial pose estimate (usually last known pose)



(b) After one iteration of the image cost minimization



(c) After two iterations of the image cost minimization



(d) Final pose estimate. The $W(p, \mathbf{u})$ that represents the transform from picture 1 to here represents the pose change

Figure 8: Illustration of minimization of image cost using Warp [14]

The process of determining the new pose with the direct method begins with a predicted image, a proposed image that some prediction algorithm believes is the state of the image($T(\mathbf{u})$). This is often set as the last image. The image to the right of each figure ($I(\mathbf{u})$) is an illustration of how the actual environment observed at that time looks like. You also see an arrow showing the warp ($w(p, \mathbf{u})$) that was needed to go from the image $T(\mathbf{u})$ to $I(\mathbf{u})$. The $p$ in the warp function represents the camera parameters, in this lies the pose of the camera. This is what we would like to determine. As the warp function is increasingly perfected by iterating through different camera parameters $p$, the predicted image $T(W(p + \Delta p_i, \mathbf{u})$ gets more and more similar to the actual image $I(\mathbf{u})$. The way the iterative change is determined is by the function

$$\Delta p_i = \underset{\Delta p}{\operatorname{argmin}} \sum (I(\mathbf{u}) - T(W(p + \Delta p, \mathbf{u})))^2$$

$$p \longleftarrow p + \Delta p_i$$

which is shown in Figure 8a with one iteration, Figure 8b with two iterations and Figure 8c with four iterations. It is important to notice that we iterate over the camera parameters $p$, which is why this eventually gives us the camera pose if the model is good enough. In Figure 8d you see that the images are almost identical, and the isometric error would be under a given threshold. The movement to the next frame could be calculated the same way.

## 2.3 Stereo camera

The advantage of using a stereo camera as opposed to a single lens-camera is that it provides the ability to assess the depth of the objects in the image. In Figure 9 you see a simple sketch of how the geometry of a stereo-camera work. In this example both cameras point in the same direction, and we assume that the only translation is in the X-direction of the sketch. "b" is the distance between the two lenses, and this is a measurement that must be known precisely. The depth (in this figure the Z coordinate) can be established using the following formula:

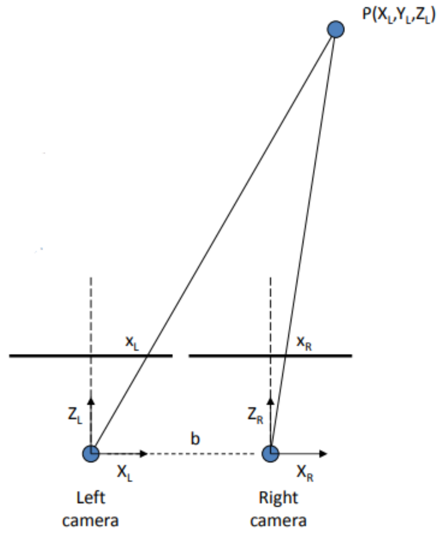


Figure 9: The geometrics of a stereo-camera

$$x_L = f\frac{X_L}{Z_L}, f\frac{X_R}{Z_R}$$
$$Z_L = Z_R = Z$$
$$X_L = X_R + b$$
$$\implies x_L = f\frac{X_R + b}{Z}$$
$$where, "d" = Disparity = x_L - x_R$$
$$d = x_L - x_R = f\frac{(X_R + b) - X_R}{Z} = f\frac{b}{Z}$$
$$Z = f\frac{b}{d}$$

This is quite a simplification, as we only assume translation in one direction. However, the same principles apply to systems of multiple dimensions. As seen in Figure 10 you can find the absolute position by triangulating the position (intersecting the rays). All you need is the intrinsic parameters of the camera (either supplied by the stereo-camera manufacturer or established using careful measurements and calibration). Points in the image from the left camera must be found and matched to points found in the right image.

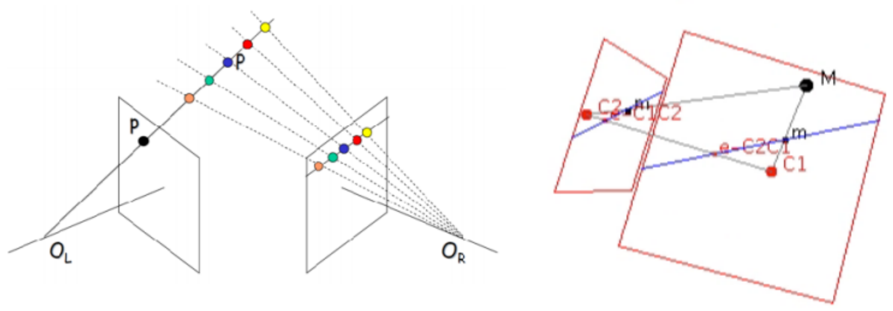


Figure 10: The geometrics of a stereo-camera

## 2.4 SOFT - Stereo odometry based on careful feature selection and tracking

SOFT, or stereo odometry based on careful feature selection and tracking, is a robust visual odometry algorithm that performs well on readily available data-sets such as the KITTI database[12]. The original paper for this algorithm [16] states that the software (with IMU aid) runs at 20 Hz on an ODROID U3, which is significantly less powerful than the Jetson TX1. This indicates that the algorithm is promising for our purpose, as it is robust, fast and accurate. In this section the inner workings of the SOFT algorithm will be presented.

### 2.4.1 Feature matching in SOFT

Sum of absolute differences (SAD), is one of the fastest and simplest methods of establishing correspondence between two images, in the case of this algorithm it is the comparison of corners. The SAD algorithm works, as the name suggests by summing the absolute differences between each pixel in the test image, and the search image. For instance, if it has a corner and search image with pixel values as illustrated in (2), the results of each pixel comparison $|corner_{x,y} - searchImage_{x,y}|$, would be equal to 1. The sum of all nine computations would be calculated and the SAD score would be 9. If the SAD score is low enough it can be considered a match. This method is very fast since it is based on very simple mathematics, but it is also susceptible to outliers. However, in the SOFT algorithm it is a first step towards finding good feature matches, as it reduces the number of features on which to perform more computationally expensive algorithms.

$$Corner = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} searchimage = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix} \tag{2}$$

After this, circular matching is performed. Circular matching is the process of matching features from the left and the right image, for both the current and the previous time-step. It is performed in a circular pattern, as it both ends and starts at the left image in the previous time-step $t_0$. As seen in Figure 11, a feature in the previous left image is matched to the right image at the same time instance, which then is matched to the current right image, to the current left image and again to the initial previous left image. The matching can be performed in different ways, for instance using the Lucas-Kandle method. The matching is performed a total of 4 times, and if any one of these fails, the circle is not closed and the feature is discarded. If they all succeed, however, the circle is considered closed and the point is to the next step in the SOFT algorithm, which is the slower but more reliable NCC (normalized cross correlation). Finally RANSAC is used to remove further outliers.

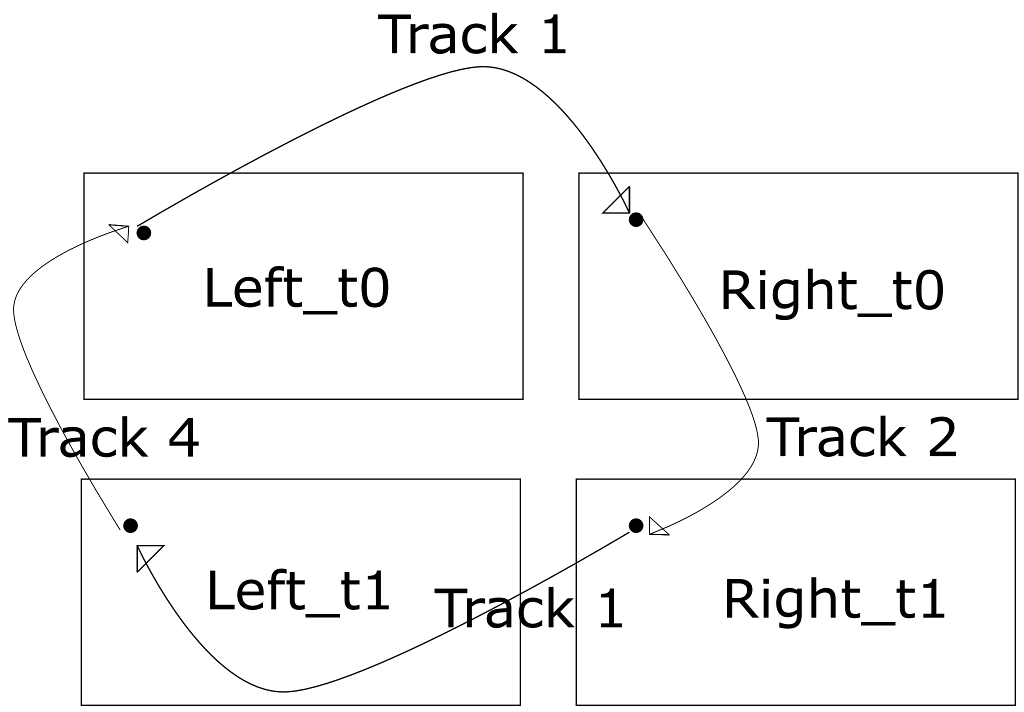


Figure 11: Figure illustrating the sequence that makes up the Circular Matching algorithm. The small circle represents a feature tracked to each frame.

### 2.4.2 Feature selection in SOFT

One important factor that makes SOFT fast enough for real time application is the selection of features. The algorithm does not use all the features when calculating the pose, but instead relies on carefully selected features. This is in fact not only faster, but also provides better results than using all the features. The process of selecting features in SOFT is called "Bucketing". Bucketing is the process of dividing the image into 50x50 pixel squares, and only keeping a small amount of the features. This ensures that the features are spread evenly across the image, and that points from both near and far are retained, giving the best conditions for good pose estimation. The process of determining what features are kept as described in the original paper [16] are:

- Features are separated into four distinct classes (corner max, corner min, blob max and blob min).
- Features inside each class is sorted by strength, i.e. by pixel value in image created by filtering the input image with blob and corner masks.
- The strongest feature from each class is pushed into the final list.
- step 3) is repeated until all features are pushed into the final list.
- The first n features from the final list are selected for motion estimation.

There is a slight modification to this list that was added since it was proven that features that has been tracked for a long time, are more reliable than points that have just recently appeared. This is caused by older points having less chance of being outliers or false positives. The actual algorithm for sorting the points is modified to the following:

$$select(f_1, f_2) = \begin{cases} stronger(f_1, f_2), & \text{if } age(f_1) = age(f_2) \\ older(f_1, f_2), & \text{if } age(f_1) \neq age(f_2) \end{cases}$$

### 2.4.3 Pose calculation in SOFT

The feature tracking in the SOFT algorithm uses the initial descriptor of the features throughout the features lifetime. This results in the feature dying earlier, but it also reduces drift significantly. The tracking of the features is performed twice. First the rotation and translation is performed using only one feature from each 50x50px bucket, and then it is performed once more using more features, but using the initially calculated rotation and translation as constraint. The pose of the camera is then calculated with regards to rotation and translation separately.
The rotation is calculated directly by the five-point method. This method does not rely on a stereo camera, and only the left camera is used. To solve the 5-point equations RANSAC is used on several random 5-point subsets, and the subset resulting most inliers represent the solution.
The translation vector is calculated by utilizing the capabilities of the stereo camera. A 3D point cloud is created for the current and previous frame based on the feature's depth and position. The error is then minimized using the photometric error, as described in 2.2.2. The SOFT algorithm can be further improved by implementing IMU-aid.

## 2.5 Local bundle adjustment

Bundle adjustment[32] is recommended as part of VO algorithms. It is the concept of estimating the imaging geometry by minimizing the reprojection error with respect to some amount of camera poses and points observed by those cameras. There are different kinds of bundle adjustment; Motion-only bundle adjustment, structure-only bundle adjustment and full bundle adjustment. You also have the option of performing bundle adjustment on the entire set of images after initialization, or to perform local bundle adjustment, that is to perform it on the $m$ last frames. Bundle adjustment can be performed using Gauss-Newton optimization which, for motion only (assuming the camera moves) simplified can be expressed like this:)
We wish to minimize the error over state variable $\Theta = \mathbf{T}_{wc}$

$$\Theta^* = \underset{\Theta}{\operatorname{argmin}} \sum_j ||\pi(g(\mathbf{T}_{wc}, \mathbf{x}_j^w)) - \mathbf{x}_{n,j}||^2$$

We then want to, using Taylor expansion, linearize the measurement prediction function

$$h(\mathbf{T}_{wc} exp(\xi_\Delta^\wedge; \mathbf{x}^w) \approx h(\mathbf{T}_{wc}; \mathbf{x}^w) + \mathbf{F}\xi_\Delta$$

, having $\xi_\Delta$ being a small perturbation in the camera frame. It can then be shown that we have a linear least-square problem of the form

$$\underset{\xi_\Delta}{\text{argmin}} \sum_j ||\mathbf{A}_j \xi_\Delta - \mathbf{b}_j||^2$$

by linearizing around $\Theta^{it}$. This is solved using the normal equations $(\mathbf{A}^T\mathbf{A})\xi_\Delta^* = \mathbf{A}^T\mathbf{b}$ which is using the Gauss-Newton optimization. Then finally update the non-linear estimate $\Theta^{it+1}$

$$\mathbf{T}_{wc} \longleftarrow \mathbf{T}_{wc} exp(\xi_\Delta^{*\wedge})$$

If it has not converged, linearize around the new estimate and repeat. For the complete calculations refer to the slides in [14].
The process for full and structure bundle adjustment is similar and can be found in [14], as well as the complete calculations on motion-only bundle adjustment.

# 3 Software Setup

The software that is created in this thesis cannot run on a computer without first installing the right operating system and third-party programs and libraries. Some of these have proved to have inadequate installation instructions. In this section the installation process of the necessary software is described.

## 3.1 Setting up the system on a laptop

In section 4.3 it is described why the choice was made to install the software on a laptop instead of the Jetson TX1. This requires that the laptop has ubuntu 16.04 installed. I initially attempted to following a simple guide [25] on how to install all the programs and packages needed to run the software. The tutorial did not work on my system, and extensive time-consuming troubleshooting was needed. It turned out to be impossible to install CUDA the way it was described in the tutorial, so it was installed using .deb files instead. The packages installed on the clean Linux (Ubuntu 16.04) were:

- Nvidia-384 (Graphics driver).
- Cuda 7.
- Cuda Toolkit.

After successfully installing this I attempted to install the OpenCV library, making sure to include the OpenCV_Contrib extra library. This failed at 21% due to the error "Compute_60". A workaround for this problem was to install CUDA 8.0 rather than CUDA 7. I installed the CUDA 8.0 using the .deb file found on Nvidia's homepage[22]. It is important to add the "nvcc" to the Linux ".bashr" file in order to make CUDA part of the PATH. This still failed at the end of the compilation. After some searching in the CMAKE file found in the OpenCV folder, the problem was discovered. The error turned out to be that the compiler ran a piece of deprecated code because it failed to identify the computer's "Cuda Compute Capability".

The computer used in this project has cuda compute capability 5.0. As shown in the code below the Cmake file should have noticed that it has a "Maxwell" GPU, and set CCC to 5.0 and 5.2. This failed, however, and the code ran to the next part where it assumed "no _cuda_arch_bin detected". Since the Cuda version is 8.0 it set CCC to (2.0 3.0 3.5 3.7 5.0 5.2 6.0 6.1). It turned out that CCC 2.0 is deprecated and causes trouble, so to compile the system I had to add this line to the Cmake file "OpenCVDetectCUDA.cmake": set(__cuda_arch_bin "5.0") on line 128.

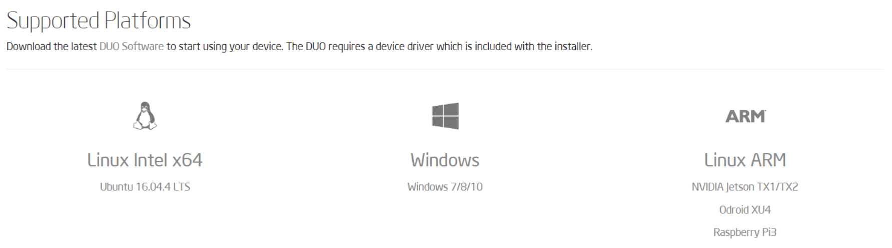
```
set(__cuda_arch_ptx "")
if(CUDA_GENERATION STREQUAL "Fermi")
  set(__cuda_arch_bin "2.0")
elseif(CUDA_GENERATION STREQUAL "Kepler")
  set(__cuda_arch_bin "3.0 3.5 3.7")
elseif(CUDA_GENERATION STREQUAL "Maxwell")
  set(__cuda_arch_bin "5.0 5.2")
elseif(CUDA_GENERATION STREQUAL "Pascal")
  set(__cuda_arch_bin "6.0 6.1")
elseif(CUDA_GENERATION STREQUAL "Volta")
  set(__cuda_arch_bin "7.0")
elseif(CUDA_GENERATION STREQUAL "Turing")
  set(__cuda_arch_bin "7.5")
```

```
.
.
.
      if(CUDA_VERSION VERSION_LESS "9.0")
        set(__cuda_arch_bin "2.0 3.0 3.5 3.7 5.0 5.2 6.0 6.1")
      elseif(CUDA_VERSION VERSION_LESS "10.0")
        set(__cuda_arch_bin "3.0 3.5 3.7 5.0 5.2 6.0 6.1 7.0")
      else()
        set(__cuda_arch_bin "3.0 3.5 3.7 5.0 5.2 6.0 6.1 7.0 7.5")
      endif()
    endif()
  endif()
  set(\_\_cuda\_arch\_bin "5.0") #Added by Haakon Eckholdt
```

It is important to note that all the files in the "Build" folder has to be removed before running CMAKE, as CMAKE skipped the files they believed were already done, and they still contained the error found in CCC 2.0. After these changes OpenCV installed correctly with CUDA support.

## 3.2 Setting up the DUO M Stereo Camera

Setting up the Duo M Stereo Camera can be done following the guide on the official DUO3D web page [7], however there are some important aspects of this process I want to highlight. First of all, this installation was performed on the laptop sporting an Intel processor and not an ARM style processor as is mentioned in 4.3. This dictates that I had to install the x64 version of the driver, as opposed to the ARM version. That might be the reason as to why it installed without problems, and that is also why I have chosen to include the workaround I had to employ on the Jetson TXI in the appendix A.1

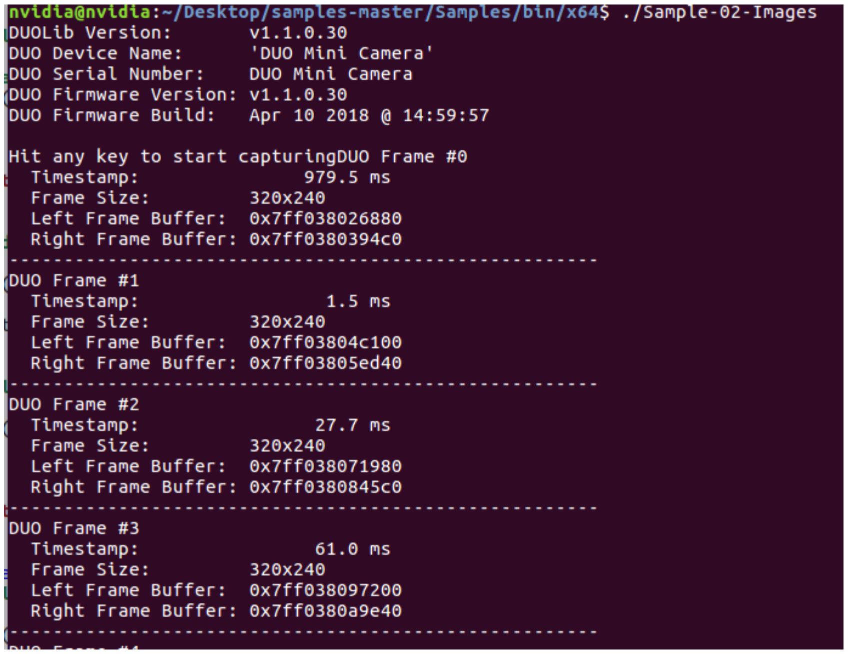

Supported Platforms

Download the latest DUO Software to start using your device. The DUO requires a device driver which is included with the installer.

Figure 12: Screenshot from the DUO installation guide[7] showing the supported platforms for the DUO M stereo camera. In this text I show how to install it using the Linux Intel x64 version, but the ARM guide can be found in the appendix.

The guide as of April 2019 works on a laptop, however I find it helpful to clarify step two of the installation process [8]. They state that you should test if the "duo-1024.ko" works. There is no definite way to test this without doing the full installation process with this module inserted. If the "duo dashboard" application does not open after installation is complete, simply "kill" the DUO dashboard in the "Task Manager" and perform the installation process again from step two, making sure to use "duo-512.ko" instead. Before inserting the new module in step two, you need to perform the "sudo rmmod -f duo" step.

The main issue when installing the driver was that the DUO dashboard worked as expected, but the code could not access the camera through the Duo API[5]. Trouble-shooting problems arising from the Duo-camera/API itself is time consuming as the documentation is sparse. One step of the solution was to copy the folder named ”SDK” from the DUO installation folder (i.e CL-DUO3D-1.1.0.30) to the ”drone-system” folder, replacing the one already receding in this folder. This is probably a necessary task to perform every time you install a new version of the DUO SDK, or if you swap from ”512.ko” to ”1024.ko”. However I cannot be sure, as moving the folder does not seem to be mentioned in any documentation provided by the Duo team as of now.



```
nvidia@nvidia:~/Desktop/samples-master/Samples/bin/x64$ ./Sample-02-Images
DUOLib Version:        v1.1.0.30
DUO Device Name:       'DUO Mini Camera'
DUO Serial Number:     DUO Mini Camera
DUO Firmware Version: v1.1.0.30
DUO Firmware Build:    Apr 10 2018 @ 14:59:57

Hit any key to start capturingDUO Frame #0
  Timestamp:                979.5 ms
  Frame Size:          320x240
  Left Frame Buffer:   0x7ff038026880
  Right Frame Buffer: 0x7ff0380394c0
-----------------------------------------------------
DUO Frame #1
  Timestamp:                  1.5 ms
  Frame Size:          320x240
  Left Frame Buffer:   0x7ff03804c100
  Right Frame Buffer: 0x7ff03805ed40
-----------------------------------------------------
DUO Frame #2
  Timestamp:                 27.7 ms
  Frame Size:          320x240
  Left Frame Buffer:   0x7ff038071980
  Right Frame Buffer: 0x7ff0380845c0
-----------------------------------------------------
DUO Frame #3
  Timestamp:                 61.0 ms
  Frame Size:          320x240
  Left Frame Buffer:   0x7ff038097200
  Right Frame Buffer: 0x7ff0380a9e40
-----------------------------------------------------
```

Figure 13: Screenshot of the terminal after running the DUO Sample - Image. This screenshot is after debugging and fixing the problems connecting to the camera using the API. Previously the only output was ”Could not connect to DUO camera”.

I found it very helpful to use the DUO official ”sample” code to test if the Duo camera API was successfully installed or not. Since the complete program for this project is large and based on code spanning several years, it is a bad idea to assume everything still works as intended. Using the sample code in [6] assures that any problem is related to the driver installation or the camera, rather than having to debug hardware/driver problems parallel to project-code problems. ”Sample 2 - Image data” is the one I used as the DUO M camera does not have an inherent IMU, as sample 1 requires. In Figure 13 I show the output of a successful test, showing that the API successfully communicates with the Duo M Stereo Camera.

## 3.3 Install eigen3

Eigen3 is a library which is helpful in this project as it is a high-level library for operating on for instance matrices, which is an essential part of VO. Installing eigen3 is very easy, and since it is a simple header library, the files to download are the same without regard to the operating system. In order to use the library, the files have to be downloaded from the official eigen3 site [28]. In the ”Install” file in the folder they state two ways of installing the library:

```
Method 1. Installing without using CMake
****************************************
You can use right away the headers in the Eigen/ subdirectory.
In order to install, just copy this Eigen/ subdirectory to your
favorite location. If you also want the unsupported features,
copy the unsupported/ subdirectory too.

Method 2. Installing using CMake
********************************
Let's call this directory 'source_dir' (where this INSTALL
file is). Before starting, create another directory which we
will call 'build_dir'.

Do:

  cd build_dir
  cmake source_dir
  make install

The "make install" step may require administrator privileges.
```

I took advantage of the Cmake method, Method 2. The installation was as simple as described above. After this the only additional step is to include the following lines in the CmakeLists.txt file in the TX1 folder of the drone-mapping system.

```
find_package (Eigen3 REQUIRED NO_MODULE)
INCLUDE_DIRECTORIES ( "${EIGEN3_INCLUDE_DIR}" )
message(STATUS "EIGENPATH: " ${EIGEN3_INCLUDE_DIR})
```

## 4 Hardware

The Nvidia Jetson TX1 (or Tegra x1, where Tegra is Nvidia's mobile chip line), is a powerful small full-featured computer on a chip. It is specifically designed to be a good choice when performing computer vision tasks on mobile platforms where size and weight is a concern. The Jetson TX1 should be powerful enough to run a visual odometry algorithm without trouble. The system runs Linux, and comes with a development-board with several I/O connections and features specified in Figure 14. The chip uses the Maxwell architecture with 256 CUDA cores, and 64-bit CPU's. Nvidia has released a new version, Nvidia TX2[3], but the changes do not seem important enough for an upgrade to be considered. However, the increased on-board memory might be important if the algorithm's memory footprint exceeds the 15GB on the TX1 (including necessary support software mentioned in section 3).

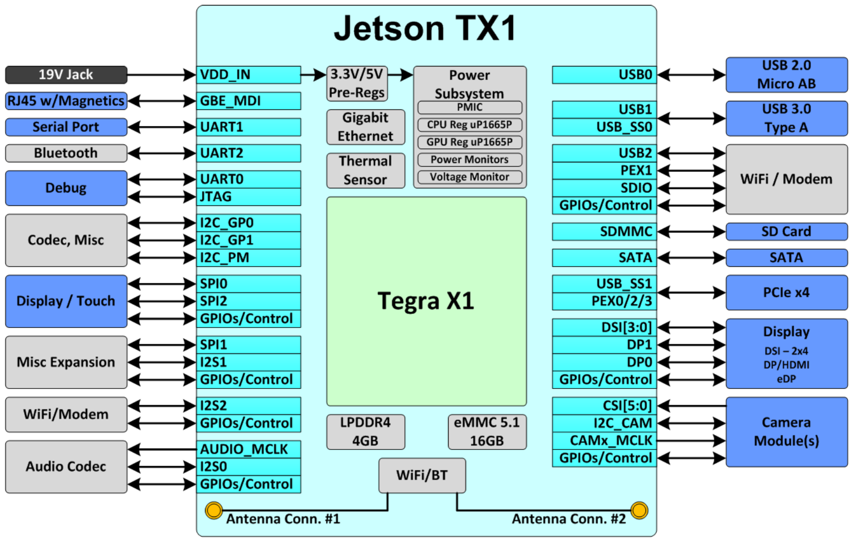


Figure 14: Complete overview of the I/O on the jetson TX1 Dev-board.

The Nvidia Jetson TX1 specifications are listed below, fetched from the Nvidia homepage [2]:

- **GPU** — NVIDIA Maxwell, 256 CUDA cores
- **CPU** — Quad ARM A57/2 MB L2
- **Memory** — 4GB 64 bit LPDDR4 25.6 GB/s
- **Data storage** — 16 GB eMMC, SDIO, SATA
- **USB** — USB 3.0 + USB 2.0

This is just a selection of the specifications that I consider important. For the entire specification portfolio visit the Nvidia homepage[2].

### 4.0.1 DUO3D Duo m Stereo Camera

The camera used in this project is the Duo M stereo camera developed by the DUO3D corporation. The combination of an off the shelf SDK (API), small form-factor, precise calibration and specification, light weight, USB interface and high speed makes it well-suited for visual odometry on a small aerial vehicle. As seen in the specifications listed in 4.0.1, the recommended depth is between 0.23m and 2.5m, perfect for indoor use. The camera is delivered as two lenses mounted on a PCB, stripped of any form of cover or protection. In order to use the camera in this project is was mounted in a 3D printed "case", which makes it heavier and bulkier than it will be once mounted on a drone.

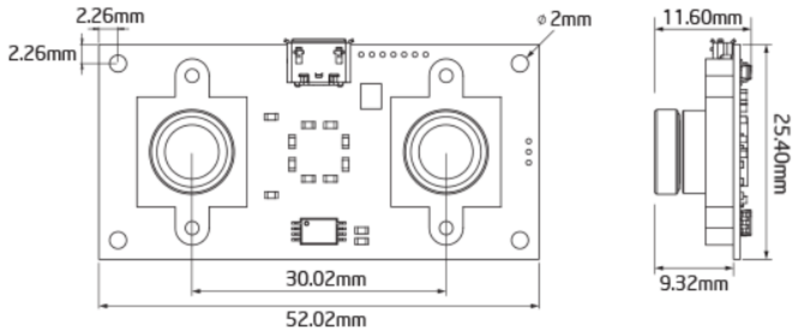


Figure 15: DuoM blueprint with size specifications [18]

- **Dimensions** — 52.02mm x 25.40mm x 11.60mm
- **Weight** — 6.5g
- **Stereo Resolutions** —
  - 45 FPS @ 752x480
  - 49 FPS @ 640x480
  - 98 FPS @ 640x240
  - 192 FPS @ 640x120
  - 86 FPS @ 320x480
  - 168 FPS @ 320x240
  - 320 FPS @ 320x120
- **Pixel size** — 6.0x6.0$\mu$m
- **Shutter speed** — 0.3 $\mu$sec - 10 sec
- **Baseline** — 30.02mm
- **Recommended depth range** — 0.23-2.5m
- **Field of View** — 165deg Wide Angle Lens with low distortion ¡ 8.5%
- **Focal Length** — 2.0mm - 2.1mm
- **Power consumption** — 2.5 Watt @ +5V DC from USB

When using the camera, it is important to note the coordinate system the camera is located in. The coordinate system as specified by the DUO3D code laboratories can be seen in Figure 16. The Z-axis represents distance between camera and any object, the X-axis represents the lateral movement (if the camera is facing forwards) and the Y-axis represents vertical movement.

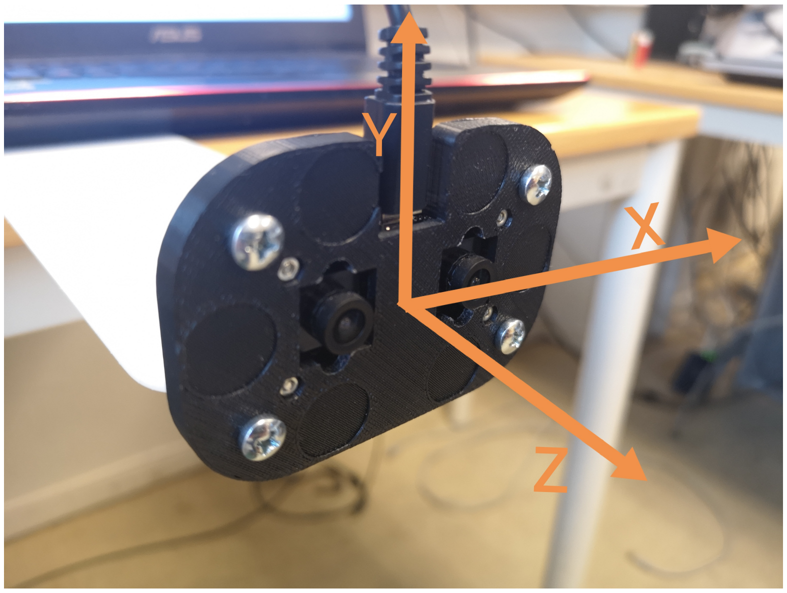


Figure 16: Image of the Duo M stereo camera with drawn on axes. The Z-axis is the viewing direction.

### 4.1 System Architecture

The system architecture is illustrated in Figure 17. There are only a small fraction of the I/O ports that are used for this project, and this indicates that a smaller developer board will suffice for the current set-up. That is a good thing considering that the developer board in this figure, which is the one I currently have access to, is quite large and heavy. It is not suited for aerial use.

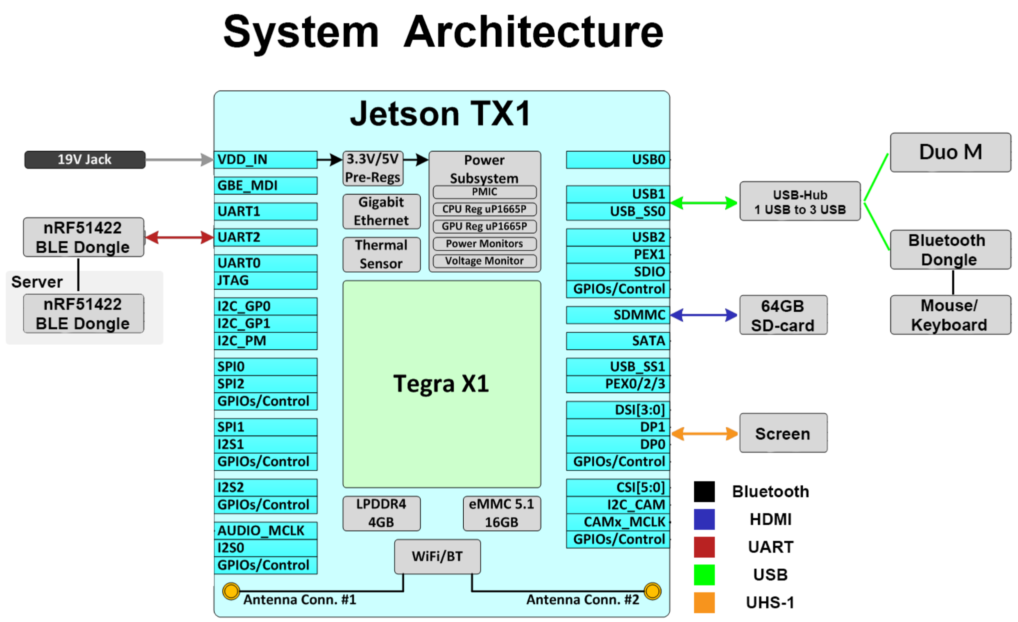


Figure 17: Simplified hardware setup using developer board schematic

## 4.2 Complete setup

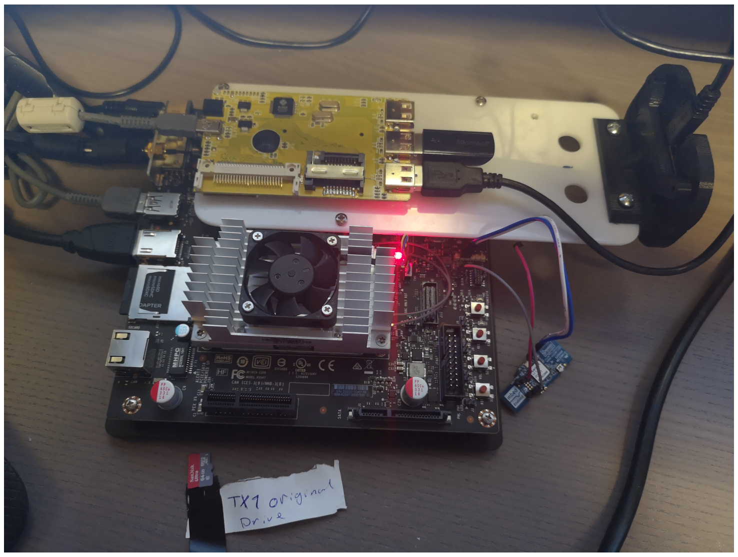


Figure 18: The complete hardware setup

The complete system consists of the Jetson TX1 on the development-board. Mounted to the PCB of the dev-board is a plastic adapter that contains both the USB-hub needed to connect more than one USB-attachment and the Duo stereo-camera. Both the Duo camera and the dongle for wireless mouse and keyboard is attached to the dongle. As seen in Figure 18 this allows for a pretty compact solution that is easy to move in the event of actual use, however, the need for a constant power-supply is troublesome, and the open PCB (Printed Circuit Board) makes transportation less than ideal. Furthermore, when used for testing, a screen must be connected in order to run the system and view the output. Attached to the system is also a Bluetooth dongle. This used to be connected directly to the header-pins of the development-board because all the USB-ports were occupied. However, this is no longer necessary since the combined keyboard/mouse dongle frees up a port. These connections can be seen in Figure 17, where the different kinds of connections are color coded. This USB dongle will have to be changed in the future as the communication protocol has changed.

## 4.3 Using a laptop for convenience

As seen in Figure 17 there is a lot of components needed in order to develop and run the system on the Jetson TX1 including, but not limited to, a screen and a constant power supply. Using a laptop makes testing and developing in different locations much easier as it is a self-contained unit with both a screen, battery and keyboard/mouse at disposal. There are, however, a difference in using the Jetson and a laptop. The two main differences are:

- The Jetson TX1 uses an ARM CPU, while the laptop uses a regular Intel i7 with the x86 instruction set.
- The Jetson TX1 has 256 Cuda cores[23]. The laptop, sporting an NVIDIA 850m has 640 Cuda cores[11].

The main difference in using ARM and Intel CPU, without going into the difference in efficiency and instruction sets(RISC vs CISC[26]), is the fact that programs might be installed differently. This is for instance true for CUDA Toolkit which is essential in this project. The difference once everything is running should be minimally affected by the CPU difference, since most of the heavy computing should be run using CUDA functions, running on the GPU.

The difference in the number of CUDA cores might represent a big difference in running-time, however. The laptop has 2.5 times more CUDA cores. Considering that the CUDA cores are what does the computation in the computer vision algorithms, this should in theory allow the laptop to run 2.5 times as many operations in parallel, increasing the frame rate significantly. The main objective of this master is not necessarily to define the exact number of frames one gets from the changes to the algorithm, so it does not introduce a problem that the laptop's and Jetson's performance differs.

## 4.4 Test Setup

I performed several tests on the system, and for convenience I made a simple test-setup which was used as an alternative to the OptiTrack Motion system in room B33 at NTNU. The different test setups are described below. There are some aspects that proved to be critical regardless of testing situation.

- Motive with easily recognized features and good contrast.

- No ”foreign” objects in the camera frame, such as chair-wheels moving differently than the floor.

On B33 the floor has a ”stone pattern” which has many edges and corners. However, it is almost monochrome which makes features harder to locate. I found two good solutions to this. One option is to test when the sun lights the room. The increased lighting resulted in plenty of features. When the room was dark the approach of laying out A4 papers helped the issue. In Figure 19 you see the features when the only source of light is the electrical lights in the ceiling. 9 features is a bad result, and they are almost exclusively located along the white line which is sub optimal.

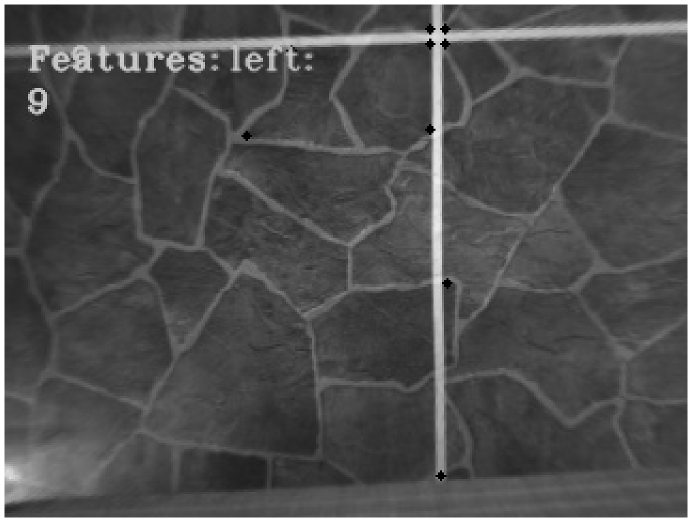


Figure 19: Image showing the number of features left after disparity filter is used. The number of features at this instance was the same regardless of filter. This test was at B33 with low lighting, resulting in few features, all located along the white line.

### 4.4.1 Straight line simple test

The simple straight-line test was performed by placing a chair so that the flat edge at the back of the chair aligned with the straight edge of the table. The camera and computer were placed on the chair and the chair was moved in a straight line and at a steady pace towards a box with a pattern that is well suited for finding features. The travel distance of the chair was fixed since it was limited by the known length of the table.

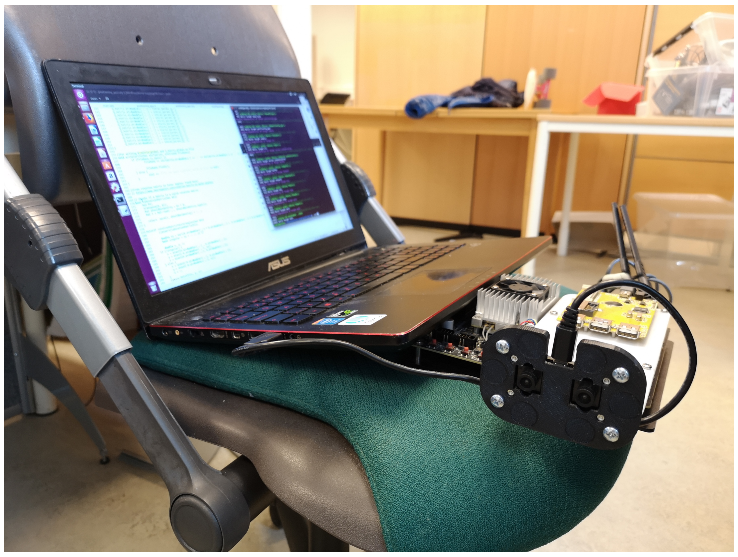


Figure 20: Image showing the laptop and the camera placed on the chair. In this image the camera was attached to the PVC board on the TX1, however this was not always the case, and does not affect the results.

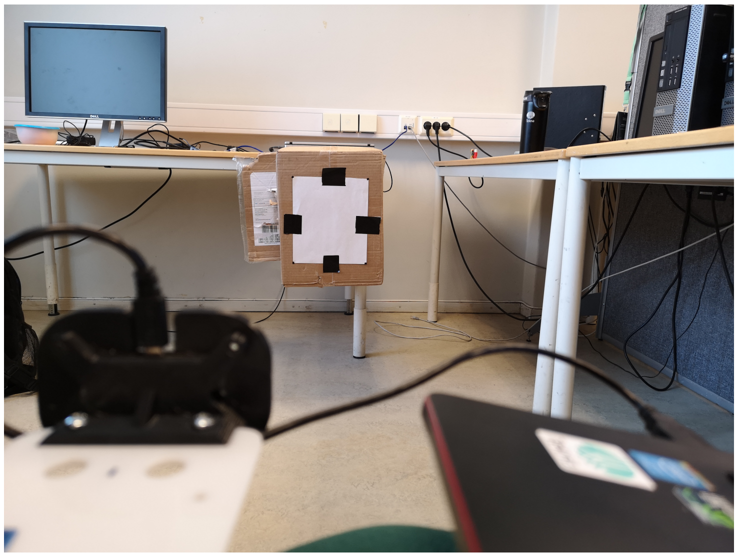


Figure 21: This is the view from the camera (seen in the foreground). The box has an A4 paper attached to it with black duct tape to ensure high contrast for the feature detectors.

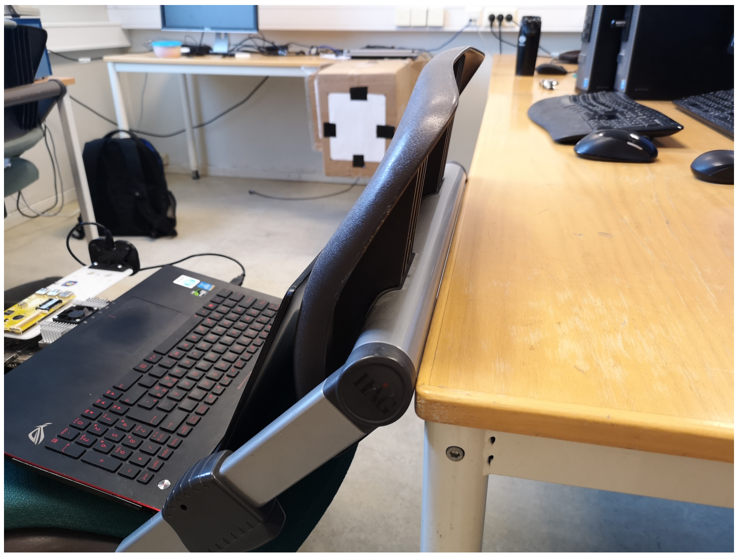


Figure 22: This image show how the flat edge of the chair glides smoothly along the flat edge of the table, assuring a single direction of movement.

One disadvantage of this test-setup is that there is no absolute measure of time, only what the software predicts. This means that any speed-variations might be missed, leaving only the end distance traveled to consideration. Another problem is that the camera might not be perfectly perpendicular to the table, giving it a false sense of moving in multiple directions.

### 4.4.2 Downwards facing camera

Another test method used was keeping the camera facing downwards. This has several benefits, amongst them that the distance between the camera and the key-points in the camera Z-direction will always stay the same (assuming a constant distance between the ground and camera). This makes troubleshooting more reliable, as you eliminate distortions and disparity-map and camera distance restrictions. This setup was performed by placing contrasting objects on the ground and moving the camera in different directions above these. Either by using a chair, a rolling table or by moving it "free-hand". The results of these movements were captured by the OptiTrack Motion system in room B33, providing an absolute measurement of position.

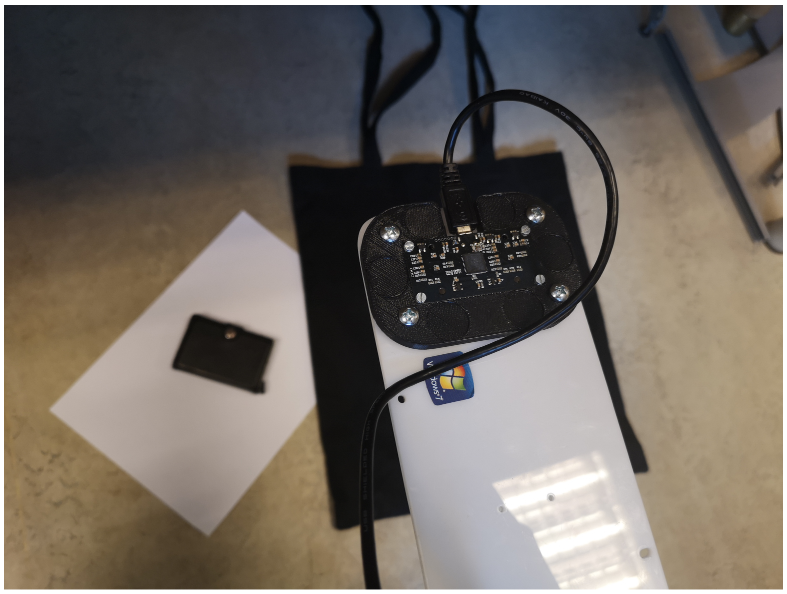


Figure 23: Image showing the camera facing downwards towards the floor, where contrasting objects are placed to ensure good keypoint detection.

### 4.4.3 Using the KITTI database

A common practice when testing CV algorithms is to use standard datasets. This has the huge advantage of resulting in the same test environment for every alteration made to the code, which means that i.e. bad lighting will not result in a good algorithm performing worse than a bad algorithm during testing. While it is useful to see the algorithm run on the actual video-stream provided by the Duo M stereo camera, using the KITTI database [12] provides the most reliable test-setup. The dataset used for this algorithm was of a car driving through a neighborhood. A test image can be seen in Figure 24. The calibration parameters associated with these images are known, which makes the test performed on this data set more valuable, as the tests performed on the Duo M camera suffers from different calibration parameters in the original and new algorithm. The difference in calibration is addressed in 5.4.

Figure 24: One of the images from the KITTI data set. The set consists of thousands of image pairs (left and right camera), and they form still images from a video-stream of a car driving through a neighbourhood. [12]

In order to get useful results from the tests performed on the KITTI dataset, the Euclidean distance between the cameras actual position and the calculated position was calculated at each frame, and then summarized to a total error. The actual position of the camera is available on the KITTI webpage, giving the ”ground truth” of the camera pose for each frame. In Figure 25, you see how the error was calculated. The circles in the illustration represents the ground truth, and the squares represent the calculated trajectory. The Euclidean distance, illustrated as the green line in the image is simply calculated as follows:

$$x_n = |x_{gt} - x_{calc}|$$

$$y_n = |y_{gt} - y_{calc}|$$

$$error = \sqrt{x_n^2 + y_n^2}$$

Where the denomination gt and calc represents the ground truth and the calculated pose respectively. This error provides a good indication as to how accurate the algorithm is, and combined with the illustrations of the calculated pose and the ground truth overlaying each other, it creates grounds for a conclusion as to how the different algorithms compare.

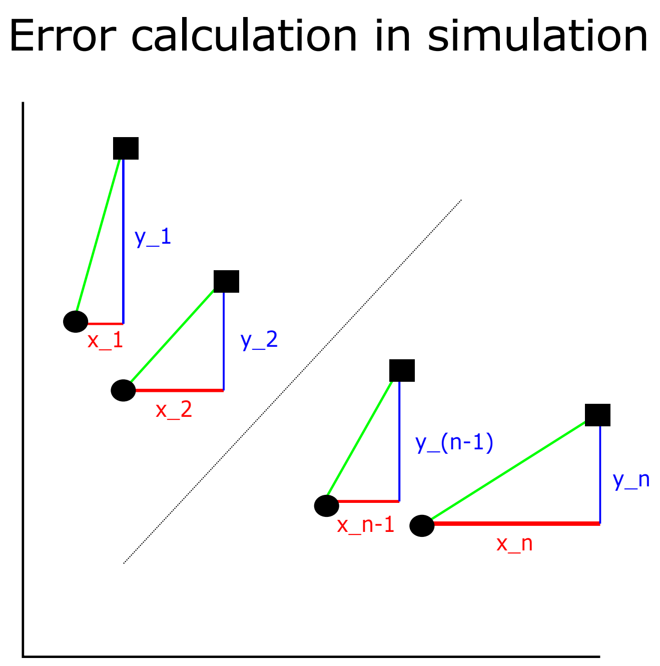
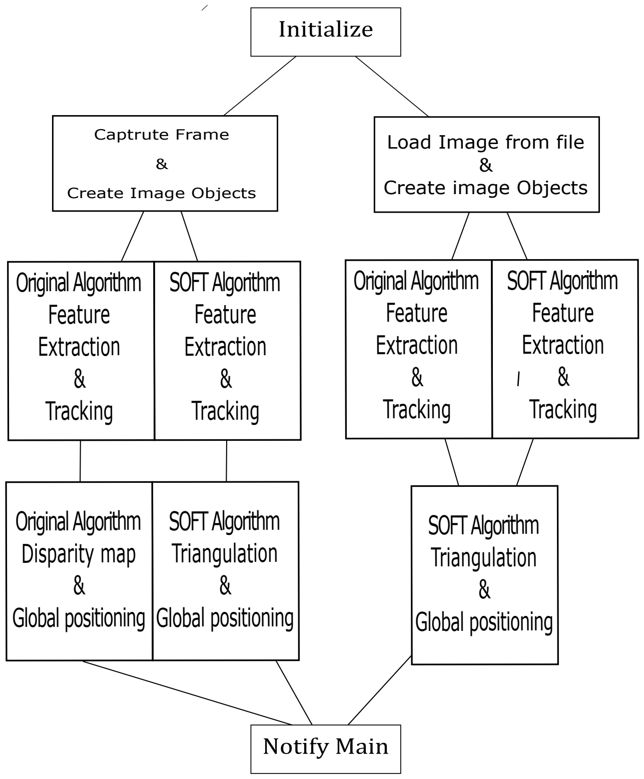


Figure 25: Illustration of how the error is calculated. The circles represent the true trajectory, the squares the predicted trajectory. The dotted illustrates that this is a general figure, and ”n” might be of any size, with n-2 points between point n and point 1.



Initialize
Captrute Frame & Create Image Objects
Load Image from file & Create image Objects
Original Algorithm Feature Extraction & Tracking
SOFT Algorithm Feature Extraction & Tracking
Original Algorithm Feature Extraction & Tracking
SOFT Algorithm Feature Extraction & Tracking
Original Algorithm Disparity map & Global positioning
SOFT Algorithm Triangulation & Global positioning
SOFT Algorithm Triangulation & Global positioning
Notify Main

Figure 26: Figure showing the flow chart of the visual odometry thread of the algorithm. The figure is strongly simplified and some of the algorithms can be seen in better detail in these figures: (30, 31, 27, 28)

## 5 Implementation

The complete system consists of multiple threads that communicate and operate asynchronously with the Main() thread. Since this master is focused on the improvement of the visual odometry thread of the system, I have mapped out the flow of the VO_Thread(). The VO thread can easily be isolated, and the other threads can be disregarded. In Figure 26 you see the program flow of the VO thread. After initializing the camera and different matrices and other parameters, this thread consists of the looping VO algorithm. The VO algorithm is currently the part of the system with the biggest potential for improvement (when the aim is precise position estimates). At the point where the flow chart branches into different boxes, as for instance when the flow is divided after "Initialize", the next path is decided by boolean variables that are set before running the program. This ensures that only one fixed path through the flow-chart will be utilized throughout a test. If a path is found to be superior during testing, the other could easily be removed to minimize the software storage footprint in the final software.

What is referred to as the "Original" program in this master thesis is the following flow:

List 1: This list refer to both the main Figure (26), and two of the expanded black boxes (30 and 27)

- Initialize
- Capture frame & create image object
- Original Algorithm Tracking
  - FastFeatureDetector
  - CalcOpticalFlowPyrKL
- Original Algorithm Position
  - Disparity Map
  - 3D image
  - Max Clique
  - Inlier Detection PnP
  - Calculate Global Pose
- Notify Main

It is important to note that the flow chart consists of black boxes, which inner workings are unspecified. However, the most important boxes has additional figures (30, 31, 27, 28) that illustrates the main principle of how they work.

The two main paths shown in the figure is the "Capture Frame" path and the "Load image from file" path. The capture frame path utilizes the DUO M stereo camera, and as such it is the code that will eventually be used on a drone, and the code we eventually intend to optimize. The "Load image from file" path uses the KITTI database [12] on the algorithm, and it is used to get consistent test results from either of the algorithms as described in 6.2. The reason for using the same triangulation method for both feature extraction paths when using the KITTI database is that the Duo Dense API does not support third party images. The pFrameData object received from the Duo camera includes disparity data needed to calculated the disparity map. This can be seen in Figure 31 and Figure 30.

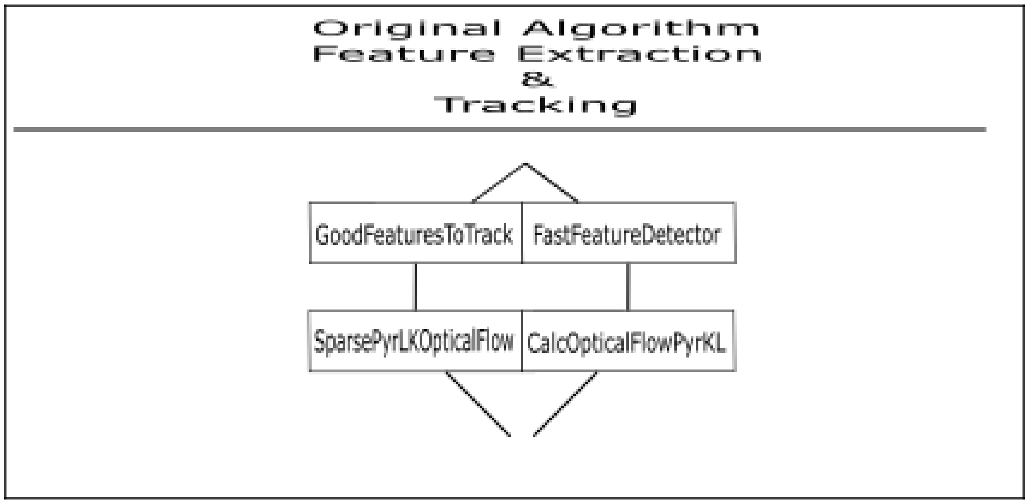


Figure 27: The original tracking algorithm allows for either the use of GoodFeaturesToTrack followed by SparsePyrLKOpticalFlow (Both GPU accelerated) or FastFeatureDetector followed by CalcOpticalFlowPyrLK (GPU accelerated and CPU run respectively)

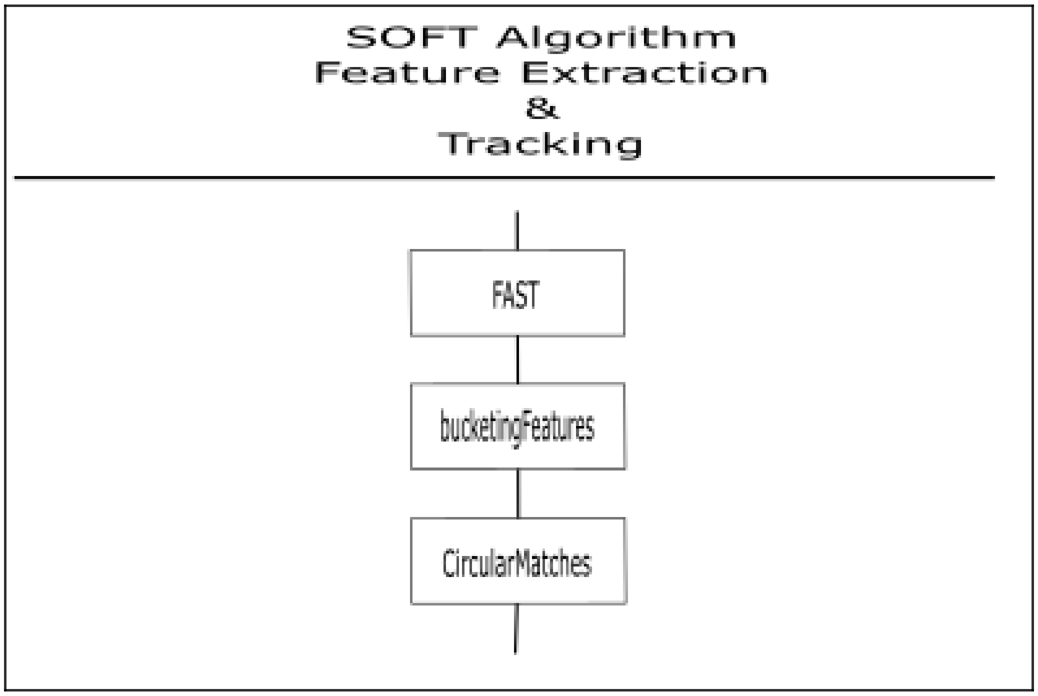


Figure 28: The feature extraction and tracking is performed using the function FAST, (Fast feature detector), followed by bucketing and circular matching.

## 5.1 Preliminary alterations due to changes in external software

The software I received at the beginning of this thesis was not operational even after installing the necessary software (OpenCV + OpenCV_Contrib, Duo driver ++). This was due to several issues, some of which was directly related to this project's code. The changes needed to run the program are outlined below:

- The included package allowing GPU support had to be changed, and in so the GPU prefix (gpu::) had to be changed to (cuda::) in the file "positioning_gpu.cpp".

```
from:
#include <Opencv2/gpu/gpu.hpp>
-----
to:
#include <Opencv2/Cudaarithm.hpp>
```

- Another interesting change that was necessary was the reversal of a function in the "camera.hpp" file. The Duo API no longer reports back if the "OpenDUO(DUOInsance *duo)" function fails, and thus returns false. I assume this was a feature in earlier versions. The documentation does not state that the function returns false on fail. However, the API documentation does specify that it returns "true" on success, and therefor reversing the function solved the problem.

```
from:
if(!OpenDUO(&_duo))
    return 0;

char tmp[260];
// Get and print some DUO parameter values
GetDUODeviceName(_duo,tmp);
printf("DUO Device Name:      '%s'\n", tmp);
GetDUOSerialNumber(_duo, tmp);
printf("DUO Serial Number:    %s\n", tmp);
GetDUOFirmwareVersion(_duo, tmp);
printf("DUO Firmware Version: v%s\n", tmp);
GetDUOFirmwareBuild(_duo, tmp);
printf("DUO Firmware Build:   %s\n", tmp);
// Set selected resolution
SetDUOResolutionInfo(_duo, ri);
return true;
---------------
to:
if(OpenDUO(&_duo))
  char tmp[260];
  // Get and print some DUO parameter values
  GetDUODeviceName(_duo,tmp);
  printf("DUO Device Name:      '%s'\n", tmp);
  GetDUOSerialNumber(_duo, tmp);
  printf("DUO Serial Number:    %s\n", tmp);
  GetDUOFirmwareVersion(_duo, tmp);
  printf("DUO Firmware Version: v%s\n", tmp);
  GetDUOFirmwareBuild(_duo, tmp);
  printf("DUO Firmware Build:   %s\n", tmp);

  // Set selected resolution
  SetDUOResolutionInfo(_duo, ri);
  return true;
else
  return 0;
```

## 5.2 OpenCV - GPU acceleration

The OpenCV library is very useful when creating VO algorithms. Most established methods in the computer vision field have existing functions in the OpenCV library. There are different versions of many functions, most important in this project is the separation of the normal OpenCV library, using the CV:: namespace, and the GPU accelerated library using the CUDA:: namespace. Functions of CV and CUDA namespace usually operate on different objects and might not be interchangeable without larger changes to the algorithm. Due to this fact the existing algorithm, while seemingly modular, is indeed difficult to change without a larger structural change.

In this thesis I chose to implement the SOFT algorithm. The function is implemented in the CV:: namespace for simplicity, and since all the needed functions were available in this namespace it makes sense to test if a GPU accelerated version is needed. The results are stored in a feature-structure which is used throughout the entire algorithm. This makes the algorithm easier to understand and faster to implement. However, this causes the GPU to be neglected, and most of Jetson TX1's power is unused.

## 5.3 Changes to algorithm

### 5.3.1 Loading images from DUO camera

In order to implement the SOFT algorithm, a different path was created by implementing a boolean variable:

```
ACTIVATE_ALTERNATIVE_VO
```

If this is set to "true" the SOFT algorithm is executed. The algorithm is executed as shown in Figure 28 and Figure 31, and the theory is described in 2.4. The algorithm is an adaptation of the algorithm proposed in the paper "SOFT-slam" [15]. It was originally created to work with the KITTI database [12], and it performs well on this dataset as can be seen in the results 6.2. The main complication has been to implement this algorithm in the VO thread of the existing software, using the Duo M stereo-camera as opposed to the KITTI database.

One of the obstacles was to make the different required libraries (for instance OpenCV, Eigen3, DUOSDK) function together, and at the same time load images from the Duo API (the live footage from the Duo M camera) as opposed to a folder with images on the computer. Below is a code extraction of how the frames are loaded into the function using the Duo M video feed, as opposed to the KITTI dataset.

```
//Create image objects from DUO M object

//Left image at instance t_0
Mat imageLeft_t0; //Changed to load DUO feed
imageLeft_t0 = Mat(Size(WIDTH, HEIGHT), CV_8UC1, pFrameData_t1->leftData);

//Right image at instance t_0
Mat imageRight_t0;
imageRight_t0 = Mat(Size(WIDTH, HEIGHT), CV_8UC1, pFrameData_t1->rightData);

// ---------------------------------------------
// Run visual odometry
// ---------------------------------------------
```

```
vector<FeaturePoint> oldFeaturePointsLeft;
vector<FeaturePoint> currentFeaturePointsLeft;

while((cvWaitKey(1) & 0xff) != 27) //  SOFT algorithm WHILE loop
{
  if (first) //first true only first pass of while loop
  {
    pFrameData_t2 = GetDUOFrame(); //image at t_1
  }else{
    pFrameData_t1 = pFrameData_t2;
    pFrameData_t2 = GetDUOFrame();
  }
```

Listing 1: Initiate DUO M as image source

### 5.3.2 Matching features

The biggest function in the SOFT algorithm is the "MatchingFeatures" function, which serves three purposes of the SOFT algorithm:

- Find features using the FAST feature extractor(2.2.1).
- Bucketing of features, as described in section 2.4.2.
- Circular matching to remove outliers, as described in section 2.4.1.

As mentioned above, the SOFT algorithm is implemented using non-GPU accelerated features. The CV:: functions used are:

**FAST**
The first cv:: function in the MatchingFeatures function is the FAST function below, and it is applied to detect the features in this step.

```
cv::FAST(image, keypoints, fast_threshold, nonmaxSuppression);
```

This is a standard FAST function as it is described in 2.2.1. The parameters are shown in Table 3, where both the original description from OpenCV [29] is listed, and the actual implementation in the project source code:

Table 3: Declaration of variables in the cv::FAST function defined by [29] and the project source code

| Parameter name | OpenCV Description | Implemented input in algorithm |
|---|---|---|
| image | greyscale image where keypoints are detected keypoints are detected | imageLeft_t0 from 1 |
| keypoints | keypoints detected on the image. | New element of type vector <cv::KeyPoint> |
| threshold | threshold on difference between intensity of the central pixel and pixels of a circle around this pixel. | 20 |
| nonmaxSuppression | if true, non-maximum suppression is applied to detected corners. | true |

**Circular Matching**

The next function that uses OpenCV code is the CircularMatching part of the MatchingFeatures algorithm. This algorithm relies heavily on Lucas Kandle Optical flow method, more specifically the OpenCV function:

```
void calcOpticalFlowPyrLK(InputArray prevImg, InputArray nextImg,
                          InputArray prevPts, InputOutputArray nextPts,
                          OutputArray status, OutputArray err,
                          Size winSize=Size(21,21), int maxLevel=3,
                          TermCriteria criteria, int flags=0,
                          double minEigThreshold=1e-4 )
```

This function is repeated 4 times, as presented in section 2.4.1. This means that the input is dependent on which of the four runs we look at, the four runs are presented below, and the OpenCV description of the inputs are described in Table 4

Table 4: Declaration of variables in the calcOpticalFlowPyrLK function defined by OpenCV

| Parameter name | OpenCV Description |
|---|---|
| prevImg/nextImg | The image for which features should be tracked from and to |
| prevPts/nextPts | The points on the input image, and the output after new position is calculated in nextIng |
| status | Output status vector of unsigned chars Each element of the vector is set to 1 if the flow for the corresponding features has been found, otherwise, it is set to 0 |
| err | output vector of errors; each element of the vector is set to an error for the corresponding feature |
| winSize | size of the search window at each pyramid level |
| maxLevel | 0-based maximal pyramid level number; if set to 0, pyramids are not used (single level) |
| TermCriteria | parameter, specifying the termination criteria of the iterative search algorithm |
| flags | Operation flags, either: OPTFLOW_USE_INITIAL_FLOW or OPTFLOW_LK_GET_MIN_EIGENVALS |
| minEigThreshold | the algorithm calculates the minimum eigen value of a 2x2 normal matrix of optical flow equations, divided by number of pixels in a window; if this value is less than minEigThreshold, then the corresponding feature's flow is not processed |

This is implemented in the project code as shown below, and the matching process is visualized in Figure 11

```
calcOpticalFlowPyrLK(LeftImage_t0, RightImage_t0, points_l_0, points_r_0,
    status0, err, winSize, 3, termcrit, 0, 0.001);
calcOpticalFlowPyrLK(RightImage_t0, RightImage_t1, points_r_0, points_r_1,
    status1, err, winSize, 3, termcrit, 0, 0.001);
```

```
calcOpticalFlowPyrLK(RightImage_t1, LeftImage_t1, points_r_1, points_l_1,
    status2, err, winSize, 3, termcrit, 0, 0.001);
calcOpticalFlowPyrLK(LeftImage_t1, LeftImage_t0, points_l_1,
    points_l_0_return, status3, err, winSize, 3, termcrit, 0, 0.001);

\\After this step all the features that had a failed tracking in either of
    the calcOpticalFlowPyrLK steps above is removed.
```

Circular matching cannot be deactivated when performing tests on this system, since we need the same number of features in both the right and the left camera. Circular matching ensures that this is the case.

**Bucketing** Bucketing of features is a term which refers to dividing the image into smaller squares, or buckets, and then removing some of the features from these areas. In this algorithm the image is divided into 50x50 px buckets, and all the features in a bucket is placed in an array. The features are then sorted, where old features are prioritized. In the last step of the bucketing function, the four best features in each bucket are returned to the feature-structure (currentVOFeatures) if there exists that many features in the bucket. The pseudo code can be seen under:

```
//Divide image into buckets
for (height in image)
    for (width in image)
        create Bucket(featuresMax = 4)

for (feature in currentVOFeatures)
    BucketFeature(Feature location, feature age)

currentVOFeatures.clear()

for (i = 0; i < numBuckets; i++)
    currentVOFeatures.append(features from bucket)


----
BucketFeature (location, age)
    if (age < 10) \\age 10 is threshold
        if (bucket.features.size() < featuresMax) \\4 in this case
            bucket.features.append(location, age)
        else
            bucket.features.youngest.remove()
            bucket.features.append(location, age)
```

**Removal of invalid points** The removal of invalid points function is one of the test variables, as shown in Table 5. This function only works after circular matching. It works by checking whether the tracking performed in the circular matching step actually found the right features. If multiple features look similar, the circular matching step might track a feature from RightImage_t1 to a completely different feature in LeftImage_t1 (see Figure 29). This would not cause the circular matching step to delete the feature, since it would report a successfull tracking. The removal of invalid points function works by checking that the position of the features we started with in LeftImage_t0, match the position of the features we have tracked to the same image at the end of the matching. The simplified code below shows how it works.

```
for (int i = 0; i < points.size(); i++)
        error = max(abs(points[i].x - points_return[i].x), abs(points[i].y -
    points_return[i].y));
```

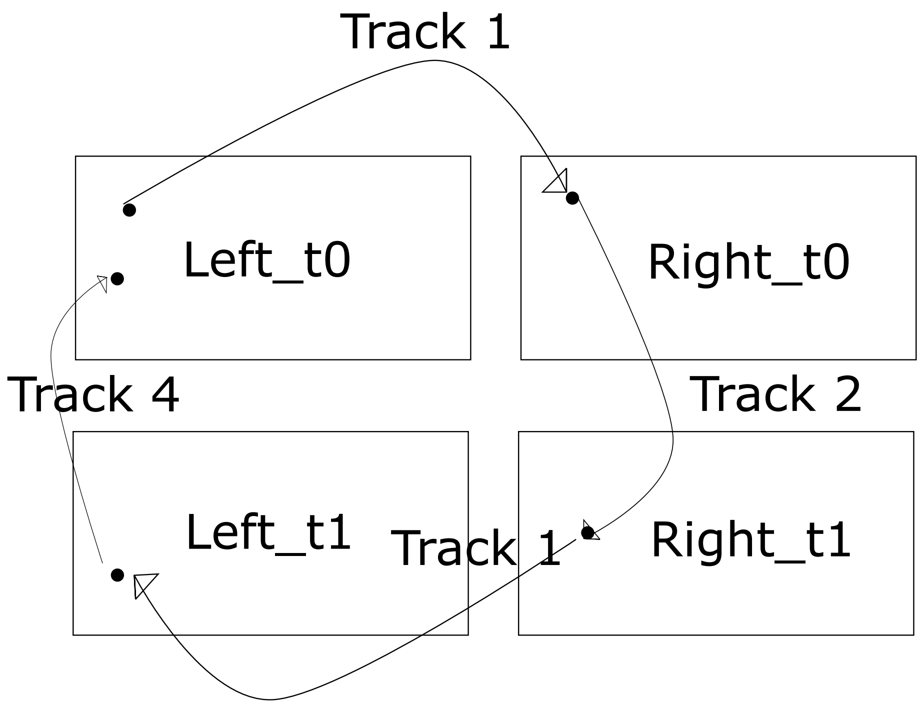


Figure 29: Figure illustrating the sequence that makes up the Circular Matching algorithm. This is a situation in which the feature is tracked to the wrong location. This feature would be removed by "remove invalid points, but not by the circular matching function.

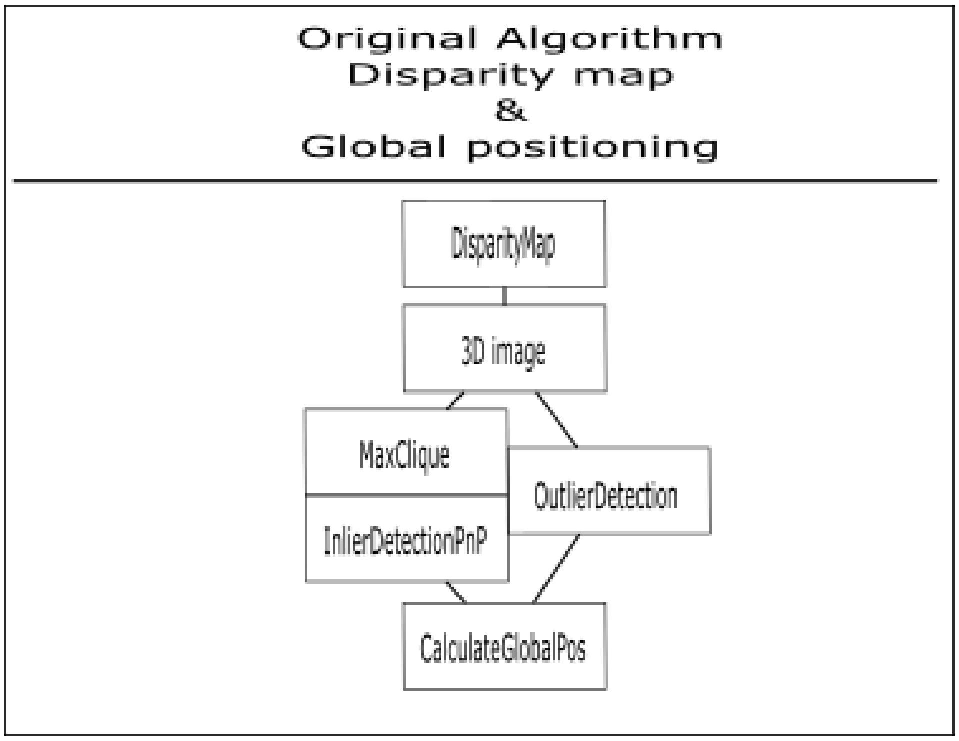


Figure 30: Figure showing the original positioning algorithm. The algorithm can either be run with inlier detection (MaxClique) or outlier removal(RANSAC). It uses the disparity map calculated by the DUO Dense API to estimate the depth.

```
        if(error > 0)
            points[i].delete
        else
            //nothing
```

### 5.3.3 Using KITTI dataset

Implementing support for the KITTI dataset in the original algorithm was tricky. This was, as mentioned earlier due to the positioning algorithm in the original system not being able to calculate the depth of third-party images. A problem when implementing the SOFT positioning algorithm with the original feature tracking is that the number of features in image t_0 and t_1 very rarely are identical. In the SOFT algorithm this is not an issue, since every feature is assessed using circular matching. This is why circular matching is mentioned at the top of the program flow above a dotted line in Figure 31. Circular matching had to be added to the original algorithm in order to get comparable results.

## 5.4 Calibration

The system algorithm can currently run with two different ways of calculating position, the original method which relies on a disparity map, shown in Figure 30, and the new

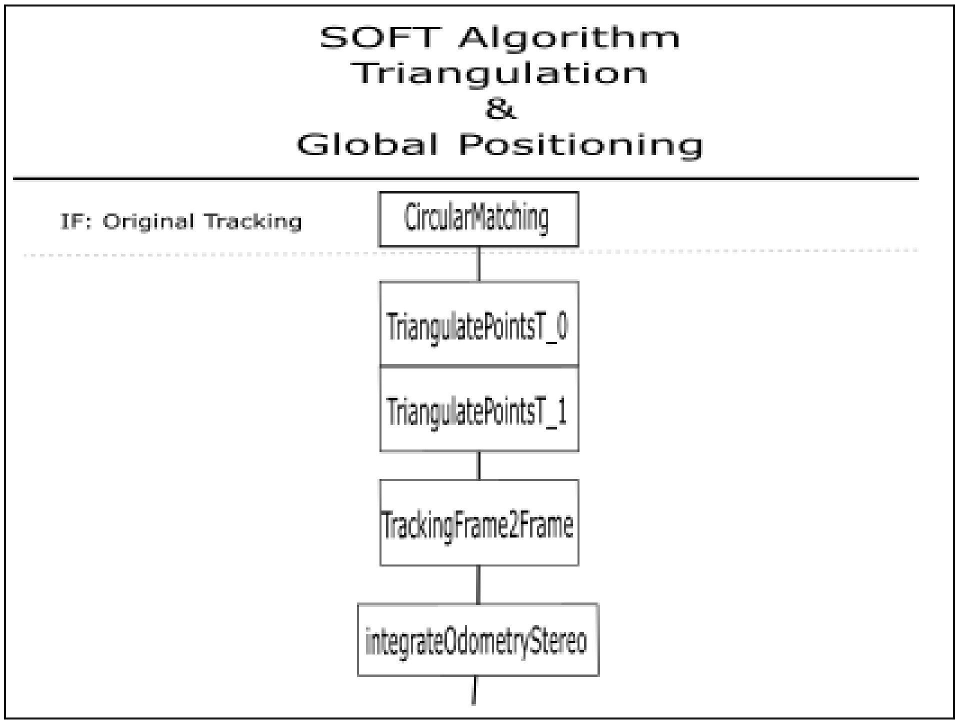


Figure 31: This function triangulates the features in the previous and current image pair using stereo camera geometry, and then calculates the position. If this function is used after Original matching and tracking the circular matching step is needed.

method which only relies on triangulation of the points, seen in Figure 31. One important difference between these is the way the calibration is used and found. In the old method the parameters are sourced from the Duo API by calling the "GetDUOStereoParameters()" function. This does not work on the dataset since it is not compatible with the Duo API. Provided with the dataset are the simple parameters of the intrinsic matrix. Attempting to use the parameters received from the "DUO calibration" application does not provide accurate results, and clearly a calibration procedure is needed in order to get accurate results using the camera. This is left as further work, and the KITTI simulations are used as a more reliable method of testing4.4.3.

# 6 Algorithm performance

## 6.1 Results

In order to compare the changes in the algorithm to the pre-existing software I had to perform some tests on the almost original system (some changes had to be made in order to run the system, as described in 5.1). There are two goals to performing these tests:

- Test that the system works approximately the same as before the change of computer and some code.
- Get a benchmark to compare the potential benefits of the changes in the code.

### 6.1.1 Straight line test old system

This test was performed in order see to what extent the drift was still present in the original system. The X axis shows the X axis of the camera (left/right) and the Y is in the Y direction of the camera (Up/down). Since the camera was moved straight forward (Cameras Z direction), a no drift result would be evident by only 0 or close to 0 values. Since the values reach 120mm and 400mm there is a lot of drift in the system.

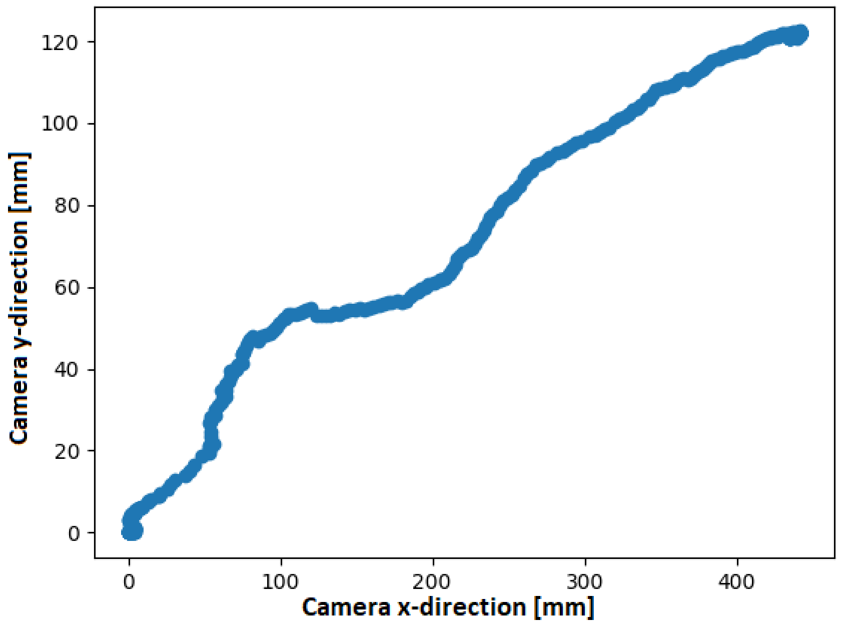


Figure 32: Plot of a test moving the system in a straight line across the room. The pose drifts around 120mm in the y direction and 400mm in the x direction. There is no plot indicating the actual movement, as there should be no movement in these axes.

I also performed a straight-line test where I first moved towards the box, and then back again in a straight line. The results are the same: there is a definite drift. However, the

uplifting news is that the position estimate is nearly the same at the end of the test as it is on the start.
The graphs do show severe drift in both the cameras X and Y axis, and this could be a result of a noisy rotation matrix, or bad tracking algorithm. It is worth noting that the Z translation also suffers from inaccuracy, as the real distance traveled is about 950mm and not close to 1200mm.

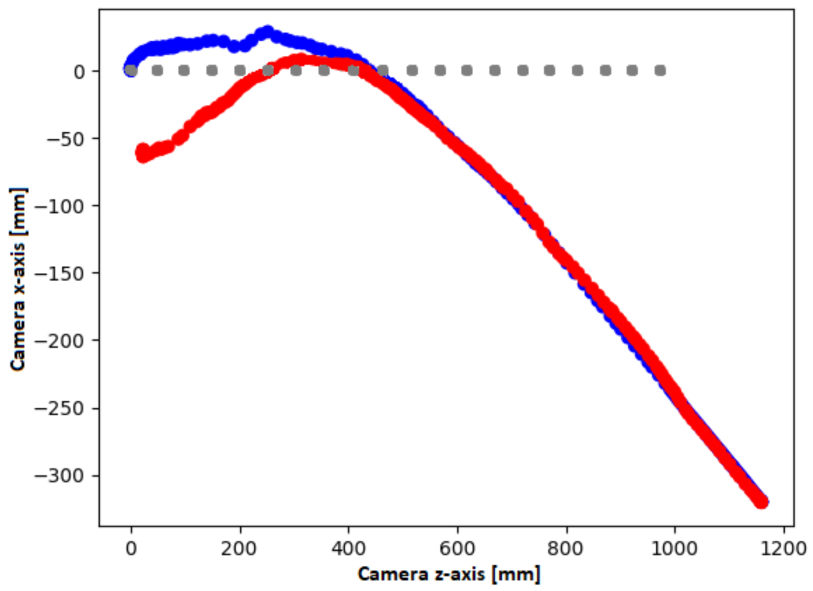


Figure 33: This image shows the cameras Z axis printed along the X axis of the graph, and the X axis along the y axis of the graph. Thus the graphs X axis represents the translation towards the box (forward/backward) and the Y axis represents a translation to the side. Grey dots indicate ideal path.

### 6.1.2 Square Test Old System

The VO algorithm can be configured in different ways depending for instance on what feature detector is used. A map of these paths is shown in Figure 26. The original algorithm, which is the setup that works the best [17], is the FastFeatureDetector followed by calkcOpticalFlowPyrLK. I performed a test of this system in B33 in order to have a benchmark of the original algorithm.
**FastFeatureDetector** $->$ **calkcOpticalFlowPyrLK** I performed this test using inlier detection, as this is how the code was presented in the beginning of the master. This is also the program flow listed in List 1. The result can be seen in Figure 34. These results seem pretty good. The square is somewhat ”round”, and the projected path is obviously not the same as the expected path (grey dots). The x and y-axis do, however, still show the correct ”max” distance of 1000mm, only somewhat twisted.

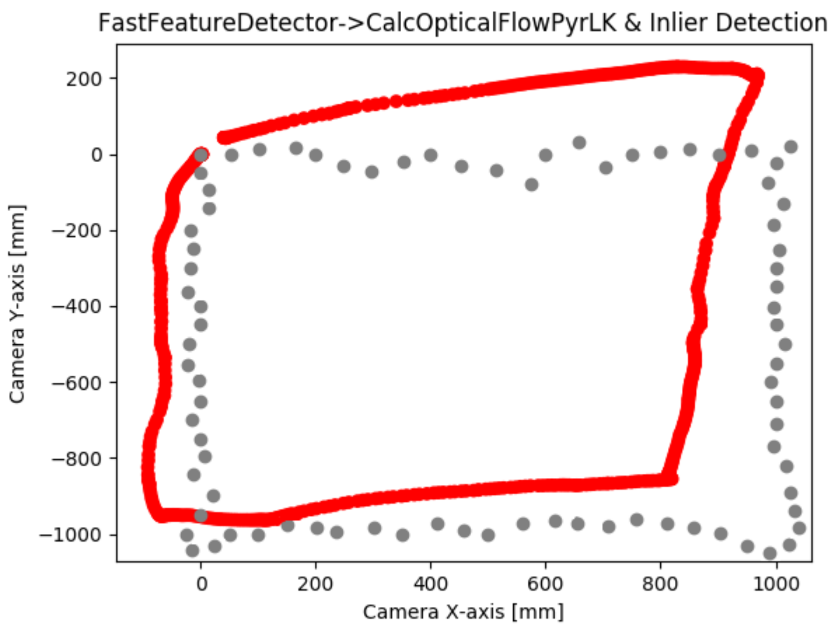


Figure 34: Plot of square-test on B33 using FastFeatureDetector followed by CalcOpticalFlowPyrLK and inlier detection. The grey dots represent the path that was attempted to move the camera in (some variation is expected due to the nature of the test setup).

## 6.2 Testing on the KITTI dataset

As described in section 4.4.3, different algorithm setups were tested on the KITTI dataset. The results of these tests will be presented in this chapter. A total of eight different setups were bench-marked, and each of the setups were tested six times, totaling 48 tests. The complete record of the test results can be found in the appendix A.2. For each of the six setups the average results are calculated, and presented in Table 6. For each of the tests a figure comprised of three sub-figures is included. The first sub-figure is an illustration of the features tracked from frame 1952 to 1953. These frames were selected because they were in a corner, resulting in a large ”angular” movement, creating large tracking lines making any difference between the setups more visible. The two additional sub-figures are of the actual tracking. The yellow line represents the ground truth (the actual position). This position is provided by the KITTI dataset. The first of these sub-figures, (b), represents the best track out of the six runs, and the last, (c), represents the worst.

It is worth noting that the illustrations do not contain any axes and scales. Since the results are overlapping themselves and moving in different directions it would not be easy to use the graphs for anything other than illustrations. The main interest is not how far it drifts away from the target in meters at any given point, but rather how far it drifts compared to the other algorithms, and how big the total error becomes. However, in order to give some insight as to the scale, the total travel distance of the ground truth (the yellow line) is 1447,87 meters. The largest difference in vertical position is 375 meters, and the largest difference in horizontal position is 462 meters (numbers apply only to the ground truth, as calculated paths strides out of this scope for some tests). The total number of frames are 1953. This dictates that if the predicted location at every single frame is 1 meter away from the ground truth (very good approximation considering the scale), the resulting total error would also be 1953.

The different test setups can be found by looking in Table 5, where the test numbers are constant. The meaning of the errors and how they are computed is presented in 4.4.3. The total error and the framerate will be the highest weighted results, as these best captures the algorithms absolute performance. However, I have included the number of features and x and y error, as they might give insight into why algorithms perform the way they do. Each test-run will be presented below.

Table 5: Setup used for different tests.

| Test | Feature Detector | Keep points | Bucketing | Remove Invalid Points |
|---|---|---|---|---|
| 1 | GPU | false | false | false |
| 2 | GPU | true | true | true |
| 3 | CPU | true | true | true |
| 4 | GPU | true | false | false |
| 5 | GPU | true | false | true |
| 6 | GPU | false | false | true |
| 7 | GPU | false | true | true |
| 8 | GPU | false | true | false |

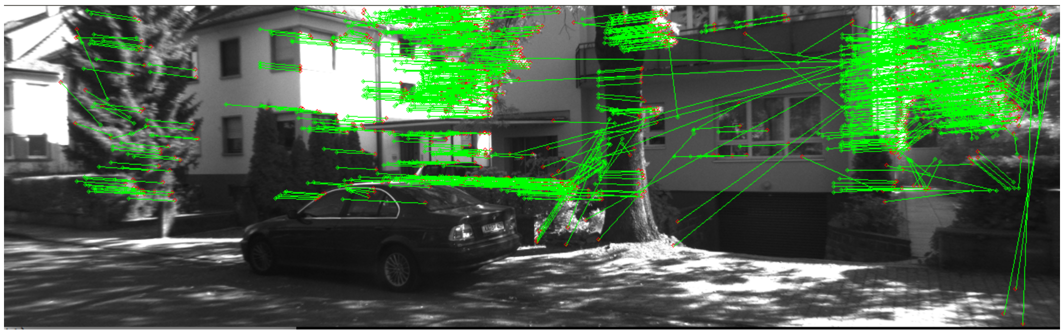
Table 6: Test results. These results are the average results of six independent tests. Full dataset can be found in appendix A.2. (test setup in Table 5).

| Test # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 10.1148 | 37 189 | 32 371 | 53 277 | 1226 |
| 2 | 21.3748 | 8 543 | 10 024 | 14 636 | 383 |
| 3 | 17.5117 | 3 886 | 3 043 | 5 159 | 665 |
| 4 | 6.9220 | 56 469 | 39 697 | 76 710 | 1806 |
| 5 | 7.5482 | 42 749 | 37 915 | 63 324 | 1744 |
| 6 | 11.2296 | 19 031 | 18 359 | 28 452 | 553 |
| 7 | 22.1889 | 22 232 | 20 005 | 32 135 | 121 |
| 8 | 17.8022 | 38 880 | 28 639 | 51 377 | 250 |

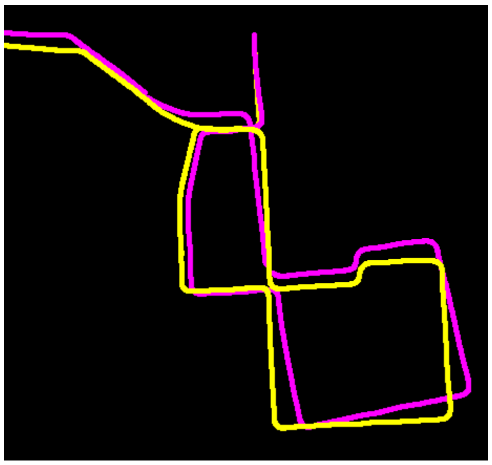
### 6.2.1 Test 1

Test 1 was performed using the GPU accelerated fast feature extractor found in the original algorithm. It discards the points after each iteration, resulting in no memory and no aging of features. Bucketing and removal of invalid points was not activated. This is the original algorithm with a different tracking algorithm, as discussed in section 5.
Looking at Table 6 we see that the average FPS is 10.1148, and the total error is 53 277. This is a rather large error, it is in fact the third largest error of all the tests. In Figure 35b, we see that it has potential, as the tracking is fairly good, and the algorithm keeps the shape of the tracking fairly good. However, looking at Figure 35c shows a bad track, where the algorithm has drifted severely, resulting in bad results. Looking at the appendix A.2 we

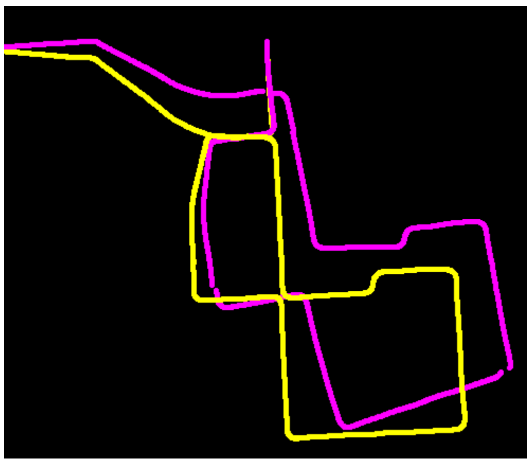
see that the good and bad tracking has an error of 30 105 and 101 366 respectively. This shows a big variation in performance. In Figure 35a we see that many of the features are tracked wrong. The correct tracking would show that the features has moved from the left to the right, but several of the features in the figure are incorrectly tracked diagonally and vertically. The features also seem to appear in clusters.



(a) Image showing the features tracked from frame 1952 to frame 1953 in the KITTI dataset. Screenshot taken from the best of the six tests.



(b) Best result using this setup.



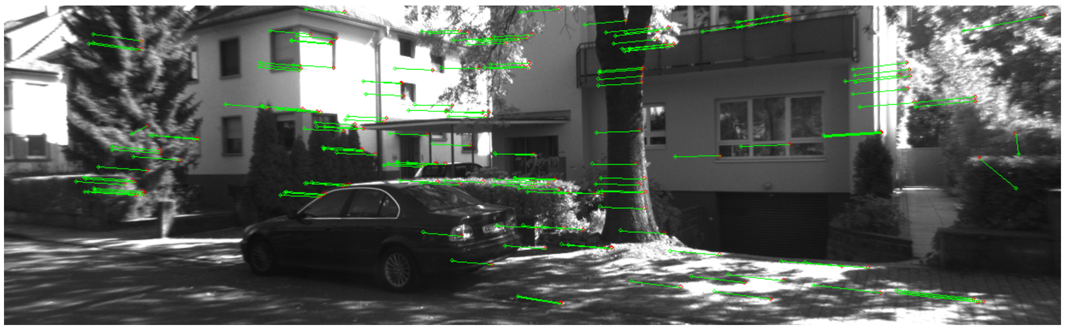
(c) Worst result using this setup

Figure 35: The results of test 1, see Table 5. The yellow line represents the ground truth. The purple line represents the algorithm's calculated trajectory. Total length of yellow line is 1447,87 meters. Maximum difference in the ground truth's vertical and horizontal position respectively is 375m/462m.

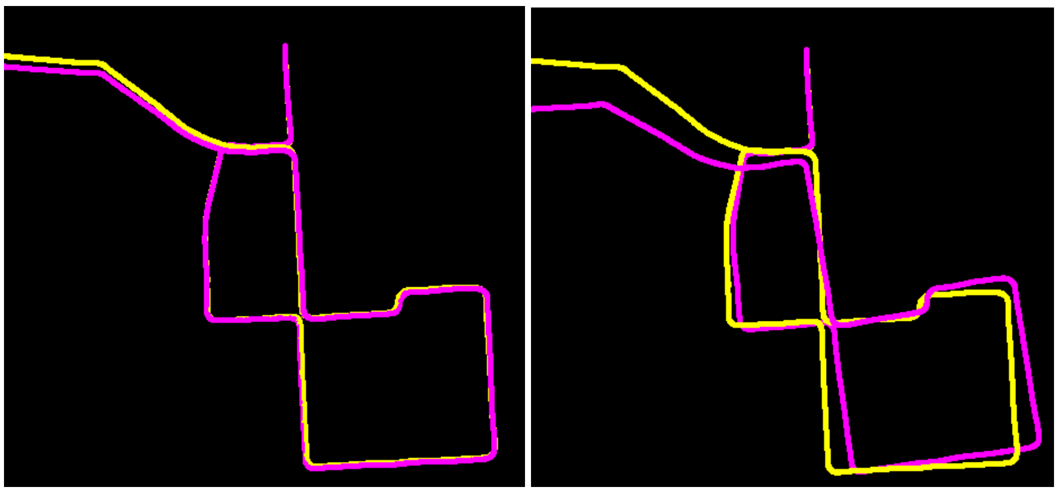
### 6.2.2 Test 2

This test was performed using point retention, GPU accelerated FAST feature detection, bucketing and removal of invalid points. This represents one version of the SOFT algorithm, as all of the key-features in the SOFT algorithm is active during this test. Due to the retention of points, new features are only extracted if fewer than 2000 points are retained. This limit is almost never reached and new features are extracted at close to every frame.

The bucketing should result in the features being spread out, and also result in older features being prioritized over newer ones. The latter is impossible to see in the figure, but the spreading of features is easily spotted by comparing the features from test one 35a and the features from this test 36a.
Taking a look at the results from Table 6 shows that this test had a frame-rate of 21.3748, and a total error of 14 636. This is the second best result, and almost twice as good as the third best result. There are only 383 features in the last frame. Features are removed in both the bucketing stage and the removal of invalid points stage, so this is not surprising. This is also easily noticed in Figure 36a. The same feature also shows that the "remove invalid points" step removed most of the features that was not tracked in the right direction. There are some incorrect matches in the right part of the image, however.
Taking a look at the tracking shows a promising result. Figure 36b shows an almost perfect tracking, where the ground truth and calculated path are overlapping most of the test. The worst result, 36c, does show a drift, but not nearly as much as in test 1. The smallest and largest total error from the tests were 6 628 and 30 221 respectively. This does indeed show that also this test has large variations, but it has a much better best and worst case result. The average error was 7.49 meters per frame, which to scale would be approximately 7.5 centimeters in the labyrinth environment (4x4 meters).



(a) Image showing the features tracked from frame 1952 to frame 1953 in the KITTI dataset. Screenshot taken from the best of the six tests.



(b) Best result using this setup.

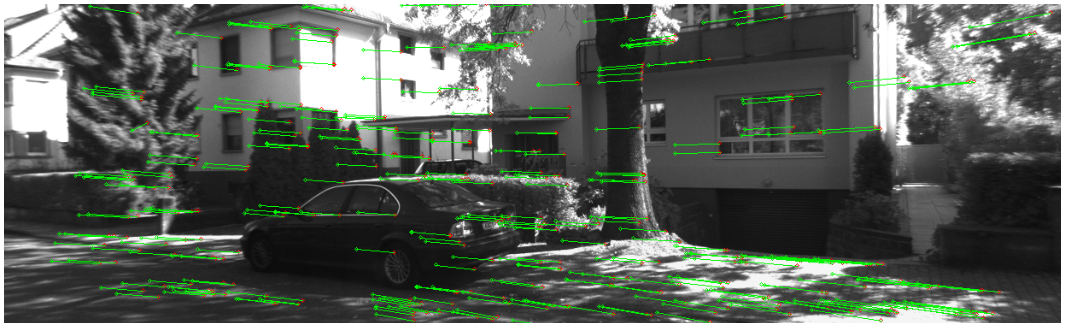
(c) Worst result using this setup

Figure 36: The results of test 2, see Table 5. The yellow line represents the ground truth. The purple line represents the algorithm's calculated trajectory. Total length of yellow line is 1447,87 meters. Maximum difference in the ground truth's vertical and horizontal position respectively is 375m/462m.

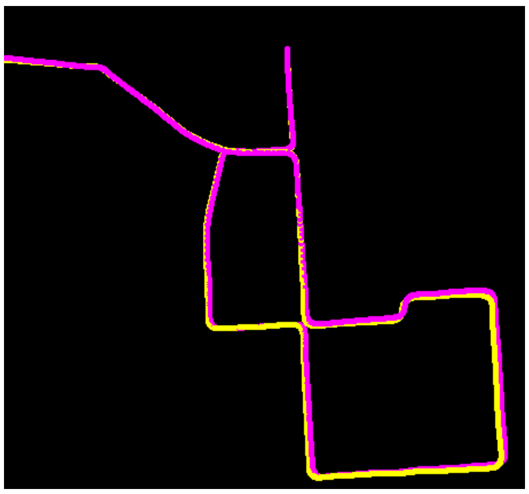
### 6.2.3 Test 3

Test 3, as shown in Table 5, uses CPU fast feature detector. It retains points and has both bucketing and removal of invalid points. This is almost the exact same algorithm as in test 2, with the distinction of using the CPU FAST feature detection as opposed to the GPU accelerated version. Since only the feature detection is different, we expect to see the same spreading of features as seen in test two. This is indeed the case, as seen in Figure 37a. We do see that there are significantly more features than in test two, 665 features as opposed to 368. Almost twice as many features. We also spot that all the features have good traces, as the lines are all moving in the correct direction. If you compare the features in test two and three (36a, 37a), it is worth noting that the features wrongly matched to the right in test 2, are not matched at all in test 3.

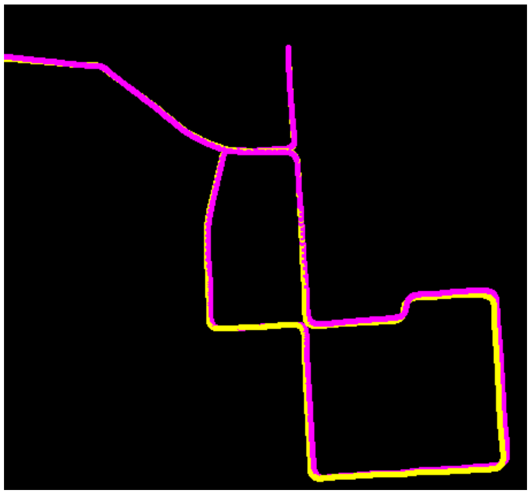
Inspecting Figure 37b and 37c shows that the best and worst calculated tracks are identical. This is indeed also the case for all the entries in the complete results table in the appendix A.2. This implies that this function is deterministic, providing no variation. The same features are found and tracked at every test. The testing shows a consistent total error of 5 159, which is the best result by a good margin. This would represent a 2.64 cm error in the maze (4x4 m maze). The frame-rate is the only entry of the table which is subject to change during the different tests, as a result of the CPU performing other tasks in parallel. The variation was minimal with an average of 17.5117, being the fourth highest frame-rate.



(a) Image showing the features tracked from frame 1952 to frame 1953 in the KITTI dataset. Screenshot taken from the best of the six tests.



(b) Best result using this setup.

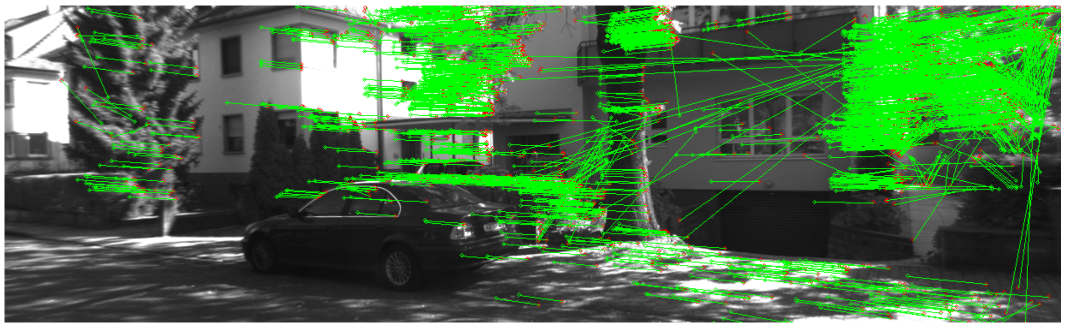


(c) Worst result using this setup

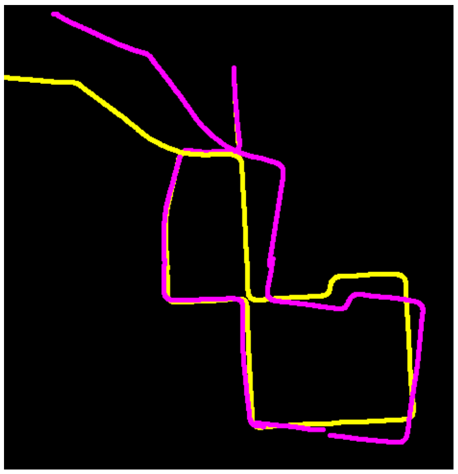
Figure 37: The results of test 3, see Table 5. The yellow line represents the ground truth. The purple line represents the algorithm's calculated trajectory. Total length of yellow line is 1447,87 meters. Maximum difference in the ground truth's vertical and horizontal position respectively is 375m/462m.

### 6.2.4 Test 4

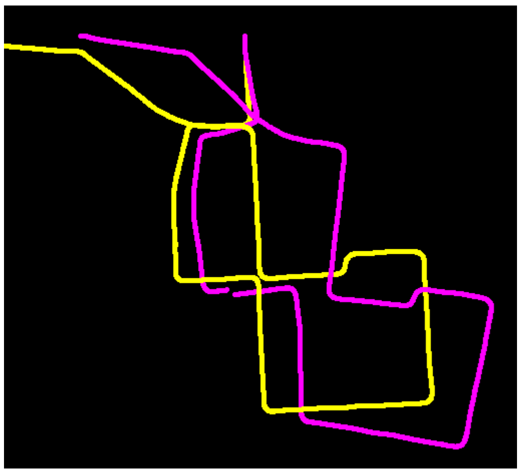
Test 4 was performed using the GPU accelerated fast feature detector and feature retention. Bucketing and removal of invalid points was not activated. This is, much like test one 6.2.1, similar to the original algorithm. What separates this from test one is the point retention. It is worth noting that age is not a factor in this test, since bucketing is not active. There will, however, be more features in each frame. Considering that "remove invalid points" is deactivated, this could result in bad points being retained, causing poor pose estimation.
Looking at Table 6 we see that the average FPS is 6.9229, and the total error is 76 710. This is a big error, in fact it is the worst performing algorithm on all counts. In Figure 38b, we see that even the best test result in poor tracking. Figure 38c shows the worst track, where the algorithm has drifted severely, resulting in very inaccurate results. The variation is also big, with the best and worst total error being 60 162 and 114 964 respectively. In Figure 38a we see that there are a lot of features (1661 in the best test), and as expected many of the features are traced wrong. This is in fact the test with the most features in frame 1953.



(a) Image showing the features tracked from frame 1952 to frame 1953 in the KITTI dataset. Screenshot taken from the best of the six tests.



(b) Best result using this setup.



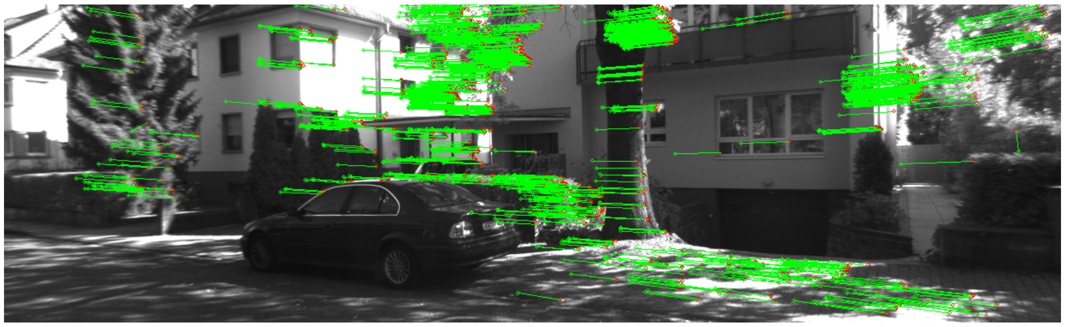
(c) Worst result using this setup

Figure 38: The results of test 4, see Table 5. The yellow line represents the ground truth. The purple line represents the algorithm's calculated trajectory. Total length of yellow line is 1447,87 meters. Maximum difference in the ground truth's vertical and horizontal position respectively is 375m/462m.

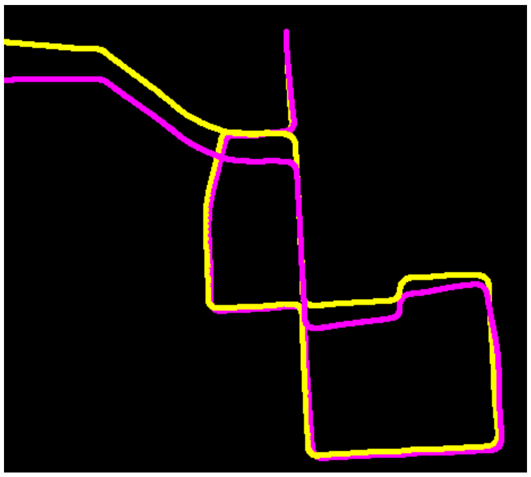
### 6.2.5 Test 5

In this test point retention and GPU accelerated FAST feature detection was used, and removal of invalid points was active (Table 5). This algorithm is identical to test 4, except for the removal of features that drift during the circular matching process. Table 6 shows that this does increase the frame-rate, as fewer points has to be accounted for in the pose estimation part of the VO algorithm. The framerate is an average of 7.5482, and the total error is 63 324. The last frame has an average of 1744 features, which is the second highest feature count by a good margin. However, Figure 39a does show that most of these features are traced in the right direction, as opposed to the features in test 4.
The total error mentioned above is still way to large. The best result shown in Figure 39b looks pretty accurate (total error of 21 650 11), but the variation is huge. The smallest error

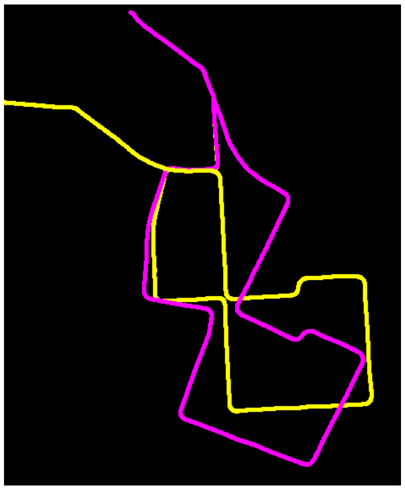
is 21 650 and the largest error is 123 873 . The worst result using this algorithm, shown in Figure 39c, is the worst of the 48 tests performed.



(a) Image showing the features tracked from frame 1952 to frame 1953 in the KITTI dataset. Screenshot taken from the best of the six tests.



(b) Best result using this setup.



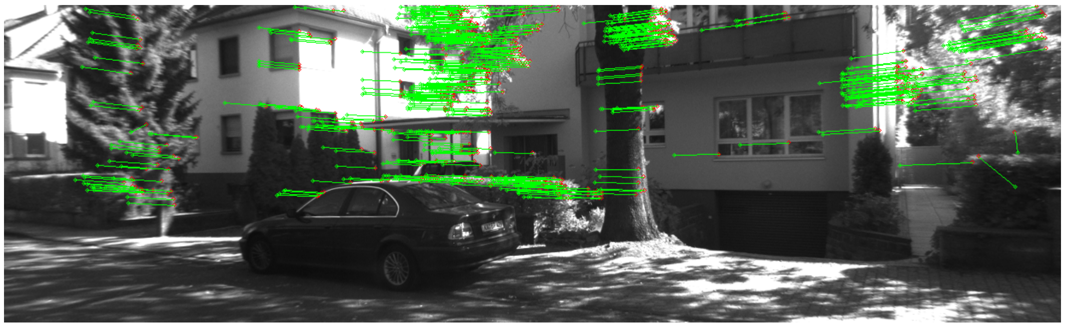
(c) Worst result using this setup

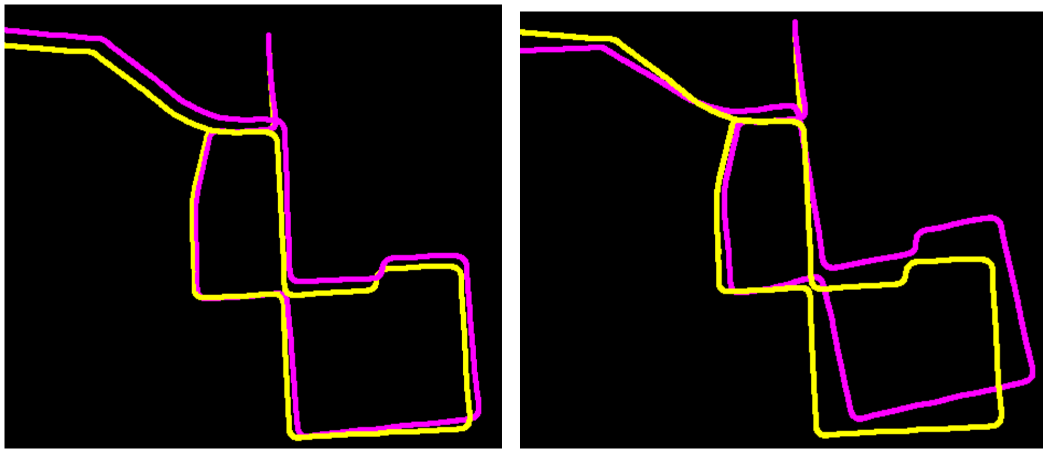
Figure 39: The results of test 5, see Table 5. The yellow line represents the ground truth. The purple line represents the algorithm's calculated trajectory. Total length of yellow line is 1447,87 meters. Maximum difference in the ground truth's vertical and horizontal position respectively is 375m/462m.

### 6.2.6 Test 6

Test 6 does not retain points and is instead finding new features using the GPU accelerated FAST feature detection at every frame. It does not bucket the features it finds, but it does remove points that drift during circular matching. As evident by Table 6, this results in a framerate of 11.2296, and an average total error of 28 452. As expected, the amount of features in the last frame is lower than the two previous test caused by lack of retention. The feature-tracing visualized by Figure 40a shows that the lack of bucketing results in many features tightly packed in localized parts of the image. This is not ideal for the triangulation

part of the pose estimation algorithm.
Figure 40b shows that this algorithm can produce a pretty accurate track of the camera, however, it is not perfect. The total error of the test shown in this figure is 18 061, a bit lower than the average performance of this algorithm. The worst test shown in Figure 40c is not terrible, but it is far from good enough with a recorded total error of 40 983.


(a) Image showing the features tracked from frame 1952 to frame 1953 in the KITTI dataset. Screenshot taken from the best of the six tests.


(b) Best result using this setup.

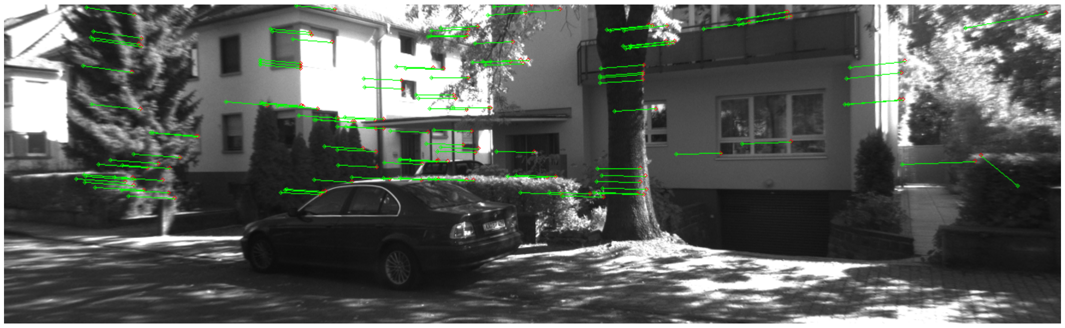
(c) Worst result using this setup

Figure 40: The results of test 6, see Table 5. The yellow line represents the ground truth. The purple line represents the algorithm's calculated trajectory. Total length of yellow line is 1447,87 meters. Maximum difference in the ground truth's vertical and horizontal position respectively is 375m/462m.

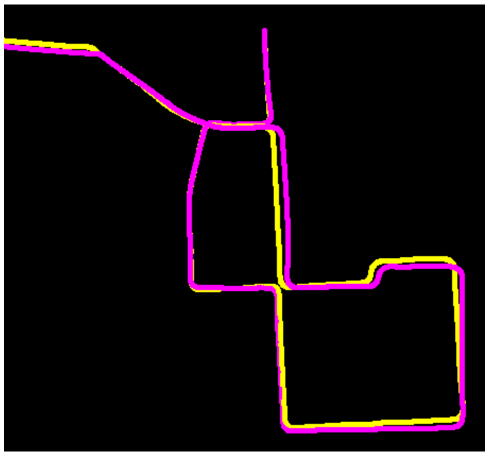
### 6.2.7 Test 7

This test finds new points at every frame using the GPU accelerated FAST feature detector with no retention. It has bucketing and removal of invalid points activated. The bucketing function does provide some functionality in this setup, but it is not utilized to its full potential. Since there is no point retention, all the points have the same age. We can therefore expect to see the features spread out, but the points will be chosen by strength

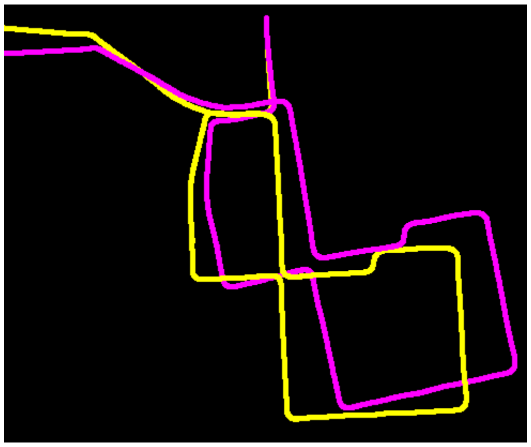
alone, and not age. However, this is not possible to recognize in these images as the features are not found deterministically when using the GPU function. By inspecting the image of the features (Figure 41a), we see that most of the features move in the desired direction, and that we avoid the clustering of features we saw in test 6, however, there are some tracings moving in the wrong direction. This is also the method with the lowest number of features (121 on average).
The best result using this function was very accurate, almost as accurate as the CPU-SOFT implementation in test 3 (6.2.3). This can be seen in Figure 41b, and the total error was no more than 13 171. It is also wort noting that the second-best test on this algorithm earned an almost identical total error. The average total error, however, is 32 135. This shows that the algorithm is not reliable and has a high degree of variation. The worst test, which is shown in Figure 41c, had a total error of 57 594. This total error is more than 14 000 larger than the worst result in test 6. The FPS on the other hand is on average 22.1889. In other words, this is the fastest of the algorithms tested in this master.



(a) Image showing the features tracked from frame 1952 to frame 1953 in the KITTI dataset. Screenshot taken from the best of the six tests.



(b) Best result using this setup.



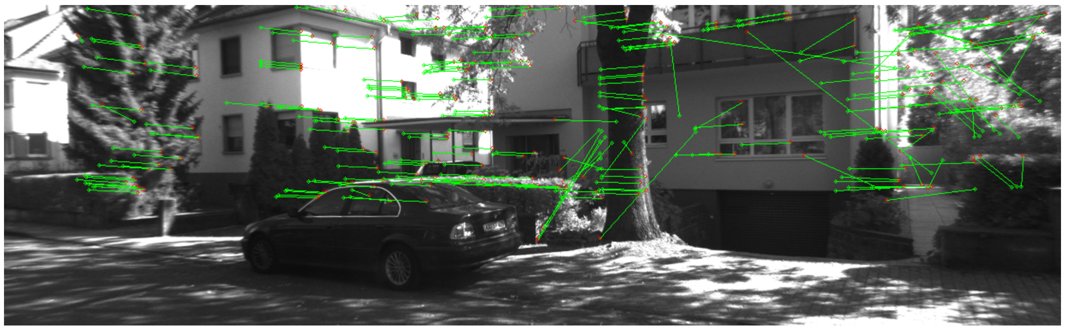
(c) Worst result using this setup

Figure 41: The results of test 7, see Table 5. The yellow line represents the ground truth. The purple line represents the algorithm's calculated trajectory. Total length of yellow line is 1447,87 meters. Maximum difference in the ground truth's vertical and horizontal position respectively is 375m/462m.

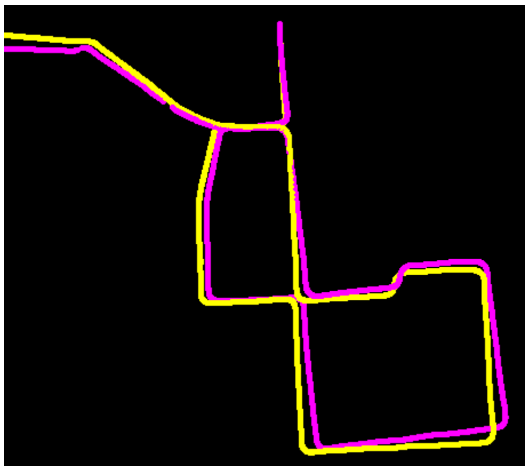
### 6.2.8 Test 8

In the final test, I used the GPU accelerated feature detector with bucketing, but without point retention and invalid point removal. In this case, as in test 7 (6.2.7), the bucketing is not utilized to it's full potential due to aging being neglected. The features are still spread across the image, which helps avoid clustering. What is different in this test as compared to the previous test, is the lack of invalid point removal. This means that many features that are traced incorrectly are used for triangulation. In fact, we see this by looking at Figure 42a. The features are not clustered, but several of the features are traced incorrectly. Comparing with test 7 where the invalid points are removed, the difference is clear. This test has an average of 250 points as opposed to the 121 features of the previous test, showing just how many features are removed by the invalid features step.
The average FPS of the system is 17.8022. This is ranking among the fastest algorithms

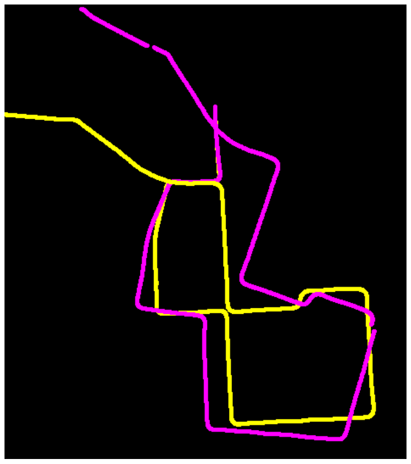
but it comes at a cost. The total error is on average 51 377. While the best of the result out of the six tests, shown in Figure 42b, has a decent tracking estimate (16 167), the high average makes it clear that this not a good performing algorithm. The worst-case tracking estimate shown in Figure 42c clearly show that is algorithm potentially can yield entirely wrong motion estimates. This estimate has a total error of 88 916.



(a) Image showing the features tracked from frame 1952 to frame 1953 in the KITTI dataset. Screenshot taken from the best of the six tests.



(b) Best result using this setup.



(c) Worst result using this setup

Figure 42: The results of test 8, see Table 5. The yellow line represents the ground truth. The purple line represents the algorithm's calculated trajectory. Total length of yellow line is 1447,87 meters. Maximum difference in the ground truth's vertical and horizontal position respectively is 375m/462m.

# 7 Discussion

## 7.1 Pose estimation on KITTI dataset

The results from the tests in section 6.2 tells us how the different parts of the SOFT algorithm affects the pose estimation of the system. Looking at Table 6 we see the different errors and FPS of the different combinations. As mentioned earlier in this thesis, a high frequency (FPS) is needed in order to get a VO algorithm suited for areal flight, as the movements might be large. Comparing the results from Table 6 and 5 we see that test 1,4,5 and 6 stand out, with the lowest FPS of any of the setups. The only thing they all have in common (except GPU accelerated feature detector) is that the bucketing function was disabled. This makes sense, as the bucketing function reduces the number of features to a maximum of $25 * 11 * 4 = 1100$. This is because the images in the dataset can fit 25 buckets horizontally, and 11 vertically, and each bucket contains 4 features. 1100 features are still unlikely, since it would indicate that the features found were spaced evenly across the image. All of the tests where bucketing is disabled have more than 1100 features, except for test 6. Test 6 does have removal of invalid points and comparing it to test 1 which is identical except for this fact, shows that there are many features removed by this function. All the features where bucketing is disabled suffers from a low FPS, but they also suffer from high total error, which dictates that bucketing is a feature that should be implemented in the finished system.

Comparing test 6 and test 7 is interesting, as they both uses GPU, does not have point retention and both have removal of invalid points. The difference lies in test 7 having bucketing, as opposed to test 6. The interesting result is that test 6 performs better than test 7, despite not having bucketing. This is most likely due to bucketing removing too many features when there is no retention. If there are too many features in one spot, or too few features all together, bucketing will remove the few points that are discovered. This can also be seen in the "features in last frame" entry. Therefor it is important that the final system can find many points, and good contrast in the maze is important.

Taking a look at the functions that take advantage of the bucketing feature, we are left with test 2, 3, 7 and 8. Test 2 and 3 both take advantage of all the functions, but test 2 uses the GPU accelerated feature detector, while test 3 uses a CPU feature detector. Test 7 and 8 differs from test 2 by both having point retention deactivated, and test 8 also having the removal of invalid points deactivated. This illustrates how important the remove invalid features step is. The average total error is about 60% higher in test 8 than test 7 (32 135 vs 51 377).

As expected, the two tests where all the support functions are active performed the best, with test 2 and test 3 scoring a total error of 14 363 and 5 159 respectively. The framerate of system 2 is 21.3748 while it is 17.5117 in test 3, a difference of 3.8631. This is a substantial difference in FPS, but it is considerably better than the FPS of the original algorithm (10.2812 at best). Test 3 using the CPU feature detector performed subliminally compared to all the other algorithms, and it always performed at the same level of accuracy, while the other algorithms had large variations. For this reason, algorithm 3 is implemented in the VO system.

The average error for each frame in the complete CPU SOFT algorithm is 2.65 meters.

When scaled down to the size of the room where the maze will be placed (4x4m), we see that this equals an error of 2.6 cm. This is well within what I would consider sufficient for this kind of algorithm. The only way to remove the drift completely would be to implement bundle adjustment. I do not think this should be a priority considering the precision of this algorithm, unless the drone is expected to fly for a long period of time, or unless further testing proves that the drift is significantly worse during actual use.

## 7.2 Accuracy of the VO

The VO algorithm in this project could be an infinity project, there is always something that can be done to improve accuracy, run-time or memory footprint, and the research on the field creates new and better algorithms all the time. While this master solely focused on the VO part of the algorithm, there are many other parts that needs to be worked on before the drone will be of any use. Therefore, at some time it must be accept that the VO is good enough. Before concluding in either direction, more testing in a more realistic setting should be performed, perhaps hanging the system under a real drone and flying it above the labyrinth. Before this is done, there is no way to know for sure. The tests performed on the KITTI dataset shows that the SOFT algorithm performs well, but none of the algorithms are perfect, and all have some degree of drift. Another aspect is that the drone needs to account for all three dimensions of space, while the tests on the kitty dataset only provides useful information in the X and Y direction (as the car drives on a flat surface). The movement in the Z direction is calculated and could easily be used, but the dataset does not provide enough stimuli in this direction to conclude on the accuracy in this direction.

# 8 Future work

## 8.1 Calibrate DUO M Camera

The algorithm currently works well on the KITTI dataset. However, it does not perform well and encounters a lot of drift when used on the Duo M stereo camera. This is most likely caused by bad values in the intrinsic matrix. In order to use the algorithm for precision use (such as controlling a drone), these parameters need to be found at a much higher level of precision. This can be done using calibration software but has not been done in this master thesis.

## 8.2 Implement bundle adjustment

Even though I do not believe bundle adjustment will be necessary once the camera has been calibrated, I do believe this should be mentioned as a good algorithm for making long sustained flight possible. Bundle adjustment eradicates drift by remembering old features and their location, and updating its calculated position based on this knowledge. Considering the drone will be contained in a small area, complete bundle adjustment, where landmarks from the entire room is used might be feasible. This would make the algorithm completely without drift. This is worth looking into if the current system still has too much drift, even after calibration.

## 8.3 Finish drone system

The end goal of this project, as described in 1.1.1, is for the drone to be able to know its position, fly to locations determined by the server and map the labyrinth. The VO algorithm described in this master thesis should perform well enough to maintain flight and report back a good enough position estimate. Therefor, further work should to a larger degree be focused on the other parts of the complete system

### 8.3.1 Communication with the server

Communication with the server was not addressed in this master thesis. This should be prioritized in the future. Other students on the Lego project has changed the protocol for sending and receiving data from the server, which means a fresh start might be in order. The old system of communication will either have to be modified to accommodate the new protocol, or an entirely new communication system must be made.

### 8.3.2 Mapping the maze

The old system for mapping the maze was developed to work on the Raspberry Pi camera on the raspberry PI. This means that it is incompatible with the current system setup. I am not aware of how well the old wall detection and mapping system worked. Considering the old nature of the code, as well as it being developed for an outdated system, a new code will have to be implemented.

### 8.3.3 Drone

If the VO algorithm performs at the level expected for flight, a drone will have to be purchased. This will require some research, both regarding the physical drone, and the

drone control-board. The control-board will have to be able to take commands from the TX1, as opposed to a radio receiver, or a radio receiver will have to be connected to the TX1. The drone will also have to balance being large enough to carry the TX1, the cameras and power supply, but also small enough to be safe and practical for indoors use.
In this process the Jetson TX1's carrier board should also be substituted with a smaller one, and options for providing power to the TX1 should be explored. The development board currently in use accepts voltages from 5 to 22 V, meaning that most Lithium batteries would provide enough voltage.

# References

[1] Chris Clark. Arw – lecture 01 odometry kinematics. 2019.

[2] Nvidia Corporation. Nvidiatx1 spec. 2019.

[3] Nvidia Corporation. Nvidiatx2. 2019.

[4] Deepak. How to prepare drivers for duo-install. 2019. Forum post posted approx. 12. Des 2016.

[5] DUO3D. Duo api. 2019.

[6] DUO3D. Duo c++ samples. 2019.

[7] DUO3D. Duo sdk installation guide. 2019.

[8] DUO3D. Duo sdk installation guide for linux. 2019.

[9] Håkon Larsen Eckholdt. Pose estimate for quadcopter usingstereo camera. 2019.

[10] Kurt Konolige Ethan Rublee, Vincent Rabaud and Gary Bradski. Orb: an efficient alternative to sift or surf. 2019.

[11] GeForce. Geforce gtx 850m — specifications. 2019.

[12] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[13] Trym Vegard Haavardsholm. Feature detection, description and matching, pp in tdt4265 2018 ntnu, 2019.

[14] Trym Vegard Haavardsholm. Short term tracking, pp in ttk21 2018 ntnu, 2019.

[15] Ivan Markovic Josip cesic Igor Cvišić, Ivan Petrović. Stereo odometry based on careful feature selection and tracking. 2018.

[16] Ivan Petrović Igor Cvišić. Stereo odometry based on careful feature selection and tracking. 2015.

[17] Bendik Bjørndal Iversen. Stereo visual odometry for indoor positioning. 2019.

[18] DUO3D Code Laboratories. Duom spec. 2019.

[19] DUO3D Code Laboratories. Installation instructions duo sdk. 2019.

[20] David G. Lowe. Distinctive image features from scale-invariant keypoints. 2019.

[21] Christoph Strecha Michael Calonder, Vincent Lepetit and Pascal Fua. Brief: Binary robust independent elementary features*. 2019.

[22] NVIDIA. Cuda toolkit 8.0. 2019.

[23] NVIDIA. Jetson tx1 module. 2019.

[24] University of Washington Richard Newcombe. Introduction to dense visual camera tracking, 2019.

[25] Adrian Rosenbrock. Setting up ubuntu 16.04 + cuda + gpu for deep learning with python. 2019.

[26] Gary Sims. Arm vs x86. 2019.

[27] Richard Szeliski. *Computer Vision: Algorithms and Applications.* 2010.

[28] Eigen team. Eigen, getting started. 2019.

[29] OpenCV team. Feature detection and description. 2019.

[30] OpenCV Development Team. Fast algorithm for corner detection. 2019.

[31] OpenCV Development Team. Optical flow opencv. 2019.

[32] the free encyclopedia Wikipedia. Bundle adjustment. 2019.

[33] the free encyclopedia Wikipedia. Roll, yaw and pitch of an air-frame, 2019.

# A Appendix

## A.1 Installing DUO Driver on Jetson TX1

The standard procedure for installing the drivers on the TX1 does not currently work, and I had to do a work-around. This is how I installed the drivers: First, download the drivers for the ARM SDK from the duo homepage[19]. Run a terminal window on the TX1 (alt + ctrl + t). Type the following into the terminal in order to prepare driver-compilation[4]:

```
$ cd /usr/src/linux-headers-'uname -r'
$ sudo make modules_prepare
```

Then open a second terminal and run the code (assuming that the filename for the SDK package is CL-DUO3D-ARM-1.1.0.30):

```
$ cd CL-DUO3D-ARM-1.1.0.30/DUODriver
$ while :;do chmod +x duodriver/run.sh;done
```

Then, in another terminal, compile the driver modules.

```
$ cd CL-DUO3D-ARM-1.1.0.30/DUODriver
$ ./duodriver.run
```

then load the dou-512-ko module. The duo-1024-ko module gives better performance, but it was not compatible with the current setup.

## A.2 Results from tests

Table 7: Results Test 1 (test setup in table 5).

| Run # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 9.8710 | 32 920 | 28 572 | 45 330 | 1226 |
| 2 | 10.2812 | 18 772 | 19 531 | 30 105 | 1226 |
| 3 | 10.1967 | 40 342 | 45 414 | 65 915 | 1226 |
| 4 | 9.9000 | 82 369 | 50 539 | 101 366 | 1226 |
| 5 | 10.2199 | 26 799 | 20 347 | 37 527 | 1226 |
| 6 | 10.2199 | 21 937 | 29 825 | 39 424 | 1226 |

Table 8: Results Test 2 (test setup in table 5).

| Run # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 21.2119 | 3 461 | 5 378 | 7 218 | 385 |
| 2 | 21.1228 | 9 412 | 8 219 | 13 511 | 381 |
| 3 | 21.9267 | 8 125 | 12 104 | 15 173 | 382 |
| 4 | 20.766 | 19 537 | 18 986 | 30 221 | 383 |
| 5 | 21.9326 | 7 997 | 10 118 | 15 069 | 382 |
| 6 | 21.2889 | 2 730 | 5 340 | 6 628 | 383 |

Table 9: Results Test 3 (test setup in table 5).

| Run # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 17.5856 | 3 886 | 3 043 | 5 159 | 665 |
| 2 | 17.4286 | 3 886 | 3 043 | 5 159 | 665 |
| 3 | 17.4345 | 3 886 | 3 043 | 5 159 | 665 |
| 4 | 17.5907 | 3 886 | 3 043 | 5 159 | 665 |
| 5 | 17.4854 | 3 886 | 3 043 | 5 159 | 665 |
| 6 | 17.5465 | 3 886 | 3 043 | 5 159 | 665 |

Table 10: Results Test 4 (test setup in table 5).

| Run # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 6.9714 | 54 353 | 45 472 | 78 983 | 1658 |
| 2 | 6.9219 | 43 032 | 25 199 | 53 470 | 2559 |
| 3 | 6.8975 | 36 601 | 39 534 | 60 162 | 1661 |
| 4 | 6.8975 | 105 351 | 34 937 | 114 964 | 1655 |
| 5 | 6.9219 | 56 767 | 55 346 | 89 892 | 1648 |
| 6 | 6.9219 | 42 710 | 37 694 | 62 789 | 1656 |

Table 11: Results Test 5 (test setup in table 5).

| Run # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 7.7154 | 17 162 | 23 916 | 32 071 | 1790 |
| 2 | 7.6250 | 46 996 | 33 953 | 60 636 | 1616 |
| 3 | 7.5953 | 42 575 | 32 002 | 55 445 | 1779 |
| 4 | 7.5077 | 91 895 | 63 298 | 123 873 | 1836 |
| 5 | 7.5077 | 4 586 | 21 650 | 23 088 | 1611 |
| 6 | 7.3383 | 53 285 | 52 673 | 84 836 | 1835 |

Table 12: Results Test 6 (test setup in table 5).

| Run # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 11.1543 | 17 419 | 8 333 | 20 655 | 554 |
| 2 | 11.2832 | 19 886 | 25 168 | 34 337 | 549 |
| 3 | 11.1543 | 11 449 | 13 053 | 18 061 | 554 |
| 4 | 11.2184 | 23 875 | 29 981 | 40 983 | 558 |
| 5 | 11.3488 | 18 921 | 11 153 | 23 124 | 553 |
| 6 | 11.2184 | 22 641 | 22 466 | 33 553 | 547 |

Table 13: Results Test 7 (test setup in table 5).

| Run # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 22.4368 | 9 431 | 8 492 | 13 782 | 113 |
| 2 | 22.4680 | 9 181 | 7 666 | 13 172 | 110 |
| 3 | 21.9326 | 46 721 | 29 184 | 57 594 | 129 |
| 4 | 22.1818 | 24 941 | 34 294 | 44 332 | 125 |
| 5 | 22.1818 | 26 146 | 20 581 | 35 889 | 122 |
| 6 | 21.9326 | 16 976 | 19 818 | 28 042 | 126 |

Table 14: Results Test 8 (test setup in table 5).

| Run # | FPS | Total X error | Total Y error | Total error | features in last frame |
|---|---|---|---|---|---|
| 1 | 17.9083 | 68 438 | 50 906 | 88 916 | 248 |
| 2 | 18.0741 | 43 117 | 42 577 | 64 887 | 251 |
| 3 | 17.9083 | 42 108 | 18 826 | 49 705 | 252 |
| 4 | 17.9083 | 31 864 | 11 817 | 35 033 | 254 |
| 5 | 17.5856 | 12 108 | 9 254 | 16 167 | 252 |
| 6 | 17.4286 | 35 649 | 38 458 | 53 554 | 245 |

# B Delivered alongside this report

## B.1 Source code

The code used and implemented in this project.

## B.2 Photos

The photos included in the report.

## B.3 Previous reports and theses

Previous reports and code that this project is a continuation of.

## B.4 Datasheets

Datasheets of some of the hardware used in this project.87