NTNU
Norwegian University of
Science and Technology

Stereo Visual Odometry for Indoor Positioning

Bendik Bjørndal Iversen

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

The LEGO-robot project already consists of several autonomous ground robots with on-board Infrared (IR)-sensors able to take measurements of the walls in an indoor maze while communicating with a server running a Java application, which creates a 2D map of the maze based on IR-data received from the robots. This master's thesis is a continuation of the work done in [18], where the final goal is to have a drone assist the robots in the LEGO-robot project in the mapping of the maze. In [3], an application for detecting wall segments of a maze, by analyzing images captured of the maze from above using computer vision algorithms, was developed on a Raspberry Pi Model 3 B. In [18], the system was further developed such that the Raspberry Pi was able to communicate with the server in the same way as the other robots. The next step, which is one of the main challenges regarding the imagined drone system, is how to estimate the pose of the drone. Since the drone is supposed to operate indoors, Global Positioning System (GPS) cannot be used. In [18], visual odometry was suggested as a way to obtain the pose estimates. The development of a visual odometry algorithm has therefore been the main goal of this master's thesis.

The algorithm is mainly based on theory found in [43] and [44], with an additional inlier detection step which is an implementation of an algorithm proposed in [16]. The algorithm uses stereo vision and advanced computer vision algorithms to analyze images captured by a small stereo camera and estimates the pose of the camera relative to the initial pose, where the origin is at the center of the left lens. It quickly became obvious that the processing power of a Raspberry Pi was not sufficient for running a visual odometry algorithm and computing the pose at a satisfactory frequency. A significantly more powerful system, the Nvidia Jetson TX1, was therefore purchased and used throughout the rest of the project. The visual odometry algorithm has been implemented on the TX1 and it is able to compute pose estimates at a rate of 5-10 Hz when taking images with a resolution of 320x240. The communication module developed in [18] has also been implemented on the TX1, and it is able to connect to the server and in addition, send the pose estimates produced by the visual odometry algorithm continually to the server. The TX1 also appears as a robot in the map created by the Java application when sending the pose estimates to the server. The mapping application developed in [3] hos not been implemented on the TX1, however.

The pose estimates produced by the current version of the visual odometry algorithm is not accurate enough to be successfully used in a drone system. Several suggestions of improvements are presented in chapter 10. This thesis does, however, show that pose estimation using stereo visual odometry is possible on hardware small enough to be placed on a drone, and the envisioned system where a drone assists in the mapping of an indoor maze in the LEGO-robot project is one step closer to realization.

Sammendrag

LEGO-robotprosjektet består av flere autonome bakkeroboter utstyrt med IR-sensorer som er i stand til å måle avstanden til veggene i en innendørs labyrint samtidig som de kommuniserer med en server, der serveren kjører en Java-applikasjon som lager et 2D kart over labyrinten basert på IR-dataen mottatt fra robotene. Denne masteroppgaven fortsetter arbeidet utført i [18], der det endelige målet er å ha en drone som er i stand til å hjelpe de andre robotene i LEGO-robotprsjektet i kartleggingen av labyrinten. I [3] ble det utviklet en applikasjon på en Raspberry Pi 3 Model B som detekterer veggsegmenter i en labyrint ved å analysere bilder tatt av labyrinten ovenfra ved bruk av maskinsynalgoritmer. I [18] ble systemet utviklet videre slik at Raspberry Pi'en var i stand til å kommunisere med serveren på samme måte som de andre robotene. Det neste steget, som også er en av de største utfordrinene i forbindelse med det tenkte dronesystemet, er å estimere posisjonen og orienteringen til dronen. Siden dronen skal opperere innendørs kan ikke GPS brukes. I [18] ble visuell odometri foreslått som en god løsning. Utviklingen av en visuell odometri-algoritme har derfor vært hovedmålet i denne masteroppgaven.

Algoritmen er hovedsaklig basert på teori fra [43] og [44], med et ytterliggere "inlier detection" steg fra [16]. Algoritmen bruker stereosyn og avanserte maskinsynalgoritmer til å analysere bilder tatt av et lite stereokamera og beregner posisjonen og orienteringen til stereokameraet relativt til kameraets initielle posisjon og orientering, der origo ligger på den venstre linsen til stereokameraet. Det ble tidlig klart at Raspberry Pi'en ikke var kraftig nok til å kunne kjøre en visuell odometri-algoritme og beregne posisjon og orientering raskt nok. Nvidia Jetson TX1, en vesentlig kraftigere enhet, ble derfor kjøpt og brukt i resten av prosjektet. Den visuelle odometri-algoritmen er implementert på TX1'en og er i stand til å estimere posisjon og orientering ved en frekvens på mellom 5-10 Hz når bildene blir tatt med en oppløsning på 320x240. Kommunikasjonsmdulen utviklet i [18] har også blitt implementert på TX1'en slik at den er i stand til å koble seg opp til serveren. I tillegg sender den posisjons-og orienteringsestimatene kontinuerlig til serveren, slik at enheten vises i kartet som Java-applikasjonen lager. Kartleggingsapplikasjonen utviklet i [3] har derimot ikke blitt implementert på TX1'en.

Posisjons-og orienteringsestimatene beregnet av den gjeldene versjonen av den visuelle odometri-algoritmen er foreløøig ikke nøyaktig nok til å kunne brukes i reguleringssystemet til en drone. Flere forslag til forbedringer er presentert i kapittel 10. Denne masteroppgaven viser derimot at det er mulig å estimere posisjon og orientering med visuell

odometri på maskinvare som er liten nok til å kunne festes på en drone, og den fremtidige visjonen om en drone som bidrar i kartleggingen av en innendørs labyrint i LEGO-robot prosjektet er et steg nærmere i å bli realisert.

Preface

The basis for the work done in this master's thesis originates from the work I did in my project thesis the previous semester and my interest in computer vision, which I've aquired through two computer vision courses I took in my fourth and fifth year on NTNU (TDT4265 and TTK25, respectively). At the beginning of this semester I had access to a Raspberry Pi that was able to connect to the server that the other robots in the LEGO-robot project communicate with, using the same protocols and hardware (a Nordic Semiconductor Bluetooth Low Energy (BLE) dongle) as the other robots. The Raspberry Pi was not, however, able to send its position (or any other payload of some sort) to the server. I had also done a brief case study of visual odometry and concluded that the implementation of a visual odometry algorithm on portable hardware such as the Raspberry Pi (or, most likely, something with a bit more processing power) was feasible.

During this semester I've had access to (or been allowed to buy) hardware that I've felt was necessary to be able to perform the work I've wanted to do, which includes the following:

- A desktop computer with sufficient processing power for my work (supplied by NTNU)
- Nvidia Jetson TX1 developer kit (bought)
- 64GB microSDHC memory card (bought)
- DUO M Stereo Camera (bought)

During the work and development of this project, my supervisor has helped me with issues concerning the overall project. When I've had questions regarding which directions I should take or which goals I should pursue regarding the project in general, my supervisor has been helpful in guiding me in the right direction. Information regarding the technical details of my master's thesis, such as theory and possible approaches regarding visual odometry, has been acquired from sources found on the internet.

Table of Contents

List of Figures

Abbreviations

6DoF	=	Six Degrees of Freedom
API	=	Application Programming Interface
ARQ	=	Automatic Repeat reQuest
BLE	=	Bluetooth Low Energy
BM	=	Block Matching
BRIEF	=	Binary Robust Independent Elementary Features
CPU	=	Central Processing Unit
CRC	=	Cyclic Redundancy Check
FAST	=	Features from Accelerated Segment Test
FPS	=	Frames Per Second
GPIO	=	General-Purpose Input/Output
GPS	=	Global Positioning System
GPU	=	Graphics Processing Unit
HDMI	=	High-Definition Multimedia Interface
IMU	=	Inertial Measurement Unit
IR	=	Infrared
L4T	=	Linux For Tegra
NCC	=	Normalized Cross Correlation
NTNU	=	Norwegian University of Science and Technology
ORB	=	Oriented FAST and rotated BRIEF
OS	=	Operating System
PnP	=	Perspective-n-Point
RANSAC	=	Random Sample Consensus
SAD	=	Sum of Absolute Differences
SDK	=	Software Development Kit
SGBM	=	Semi-Global Block Matching
SIFT	=	Scale-Invariant Feature Transform
SSD	=	Sum of Squared Differences
TCP	=	Transmission Control Protocol
UART	=	Universal Asynchronous Receiver-Transmitter
USB	=	Universal Serial Bus

Chapter 1

Introduction

1.1 Background and previous work

1.1.1 LEGO-robot project

This project is a part of the LEGO-robot project which has existed since 2004. The overall goal of the project is to have several robots work together in the mapping of an indoor maze using sensors attached to the robots. It currently consists of multiple autonomous ground robots as well as a server running a Java application. The robots send measurements taken by the onboard IR-sensors to the Java application using BLE technology, and the Java application creates a 2D map of the maze that the robots are mapping using the received measurements.

1.1.2 Introduction of the drone system and a Raspberry Pi

In 2016, a new project within the LEGO-robot project was initiated that explored the possibility of having a drone assist the other robots in the maze mapping. In [3], an application was developed on a Raspberry Pi 3 Model B which made it possible to take images of a maze from above using a camera connected to the Raspberry Pi and detect wall segments using computer vision algorithms. The idea was to attach the Raspberry Pi to a drone in the future and have the drone communicate with the server in the same way as the already existing robots in the LEGO-robot project. The envisioned drone system would send the detected wall segments to the Java application, which could use the information when creating the map of the maze. In [18], the Raspberry Pi was further developed to be able to connect to the server in the same way as the other robots. The other robots use a Nordic

Semiconductor BLE dongle and a communication protocol developed in [23] to connect to the server running the mapping application. The protocol was originally developed and implemented specifically on hardware running FreeRTOS, a real-time operating system developed for embedded devices [28], and the protocol was ported to regular Ubuntu and implemented on a Raspberry Pi in [18].

1.1.3 Visual odometry for pose estimation

If a drone were to assist in the mapping of the maze, the pose of the drone would need to be estimated using other technology than GPS, since the drone is supposed to operate indoors. Visual odometry is a method in the field of computer vision and robotics for estimating the pose of a vehicle by analyzing images taken consecutively by a camera attached to the vehicle. In [18], in addition to the implementation of the communication protocol on a Raspberry Pi, the feasibility for using visual odometry for pose estimation in this project was explored. Visual odometry was found to have worked well in many applications and it was concluded that it was a very good alternative for pose estimation in the scope of this project as well. However, the Raspberry Pi was found to most likely lack the computational performance needed to be able to run a potential visual odometry algorithm at a satisfactory frequency, since visual odometry is a computationally expensive algorithm, and the acquisition of more powerful hardware was suggested.

1.2 Problem description

This master's thesis aims to continue the development of the project where the main goal is to have an autonomous drone assist the other robots in the LEGO-robot project in the mapping of an indoor maze. As described in section 1.1, an application that is able to detect wall segments of a maze using an onboard camera on a Raspberry Pi 3 Model B was developed in [3], and in [18], a Raspberry Pi was successfully integrated into the LEGO-robot project such that it was able to communicate with the server running the mapping application. The next step is to develop a pose estimation algorithm, which is necessary for the drone to be able to operate indoors.

In [18], visual odometry was found to be the most promising method for estimating pose indoors using sensors onboard a vehicle, and developing such an algorithm is the main goal of this project. The algorithm will use images taken from a DUO M stereo camera, described in section 3.3. Since it is very likely that the Raspberry Pi is not powerful enough to be able to run a visual odometry algorithm at a satisfactory frequency, an alternative platform needs to be acquired. In that case, the communication system as well as the wall segment detection application needs to be implemented on the new platform as well. In addition to this, the system should be able to continually send the calculated pose estimates to the Java application while connected to the server, such that the system appears as a participating robot in the map that the Java application creates.

The objectives can be summarized in the following list:

- Most likely: Acquire hardware powerful enough to run a visual odometry algorithm at a satisfactory frequency and implement the already developed applications on the new platform
- Develop a visual odometry algorithm able to estimate the pose of a moving vehicle operating indoors
- Test the accuracy and performance of the algorithm and suggest improvements
- Include the pose estimation into the application that is already able to communicate with the server, and make the system send the pose estimates to the server such that the system appears in the map created by the Java application

Chapter 2

Theory

2.1 Stereo visual odometry

2.1.1 Introduction

The following two paragraphs is copied directly from [18]:

Odometry, in the field of navigation, is the process of estimating a moving objects change of position over time using encoders on moving actuators, such as rotary encoders on wheels, to measure the wheel rotations. Rotary encoders is mostly what the existing robots in the LEGO-robot project use today to estimate their position, but this is not a possible approach for a drone.

A drone can move in six degrees of freedom (6DoF), and the complete pose at time t is described by the vector $[x_t, y_t, z_t, \alpha_t, \beta_t, \gamma_t]$, where $[x_t, y_t, z_t]$ represents the position (in Cartesian coordinates) and $[\alpha_t, \beta_t, \gamma_t]$ represents the orientation (here given in the Euler angles) of the drone. Determining position and orientation using a camera and analyzing the associated camera images is called visual odometry. The algorithm uses sequential images taken from a camera that can be mounted on any kind of vehicle, such as a drone, to estimate the motion from one time instance to the next. The camera can either be a single lens-camera or a stereo-camera, in which the latter provides two images of the same scene at each time instance. Using a single lens-camera for pose estimation is called monocular visual odometry, where pose estimation using a stereo camera is called stereo visual odometry. For monocular visual odometry, scale ambiguity is an inherent problem, which is eliminated using a stereo camera setup [1]. Stereo visual odometry is therefore a preferable approach for the pose estimation.

A standard stereo visual odometry algorithm can be summarized in the following steps [43]:

1. Do only once:
   1.1. Capture two frames I_{k-2}, I_{k-1}
   1.2. Extract and match features between them
   1.3. Triangulate features from I_{k-2}, I_{k-1}
2. Do at each iteration:
   2.1. Capture new frame I_k
   2.2. Extract features and match with previous frame I_{k-1}
   2.3. Compute camera pose (PnP) from 3-D-to-2D matches
   2.4. Triangulate all new feature matches between I_k and I_{k-1}
   2.5. Iterate from 2.1).

This is the outline of the visual odometry algorithm implemented in this project. Some other steps are also included, such as an inlier detection step to discard features detected in I_k and I_{k-1} that are considered as inconsistent. Also, the algorithm presented above computes the pose of the camera at I_k relative to I_{k-1}, and a final step is needed to compute the global pose (e. g. the current pose relative to the initial pose at $k = 1$). The basic theory behind these steps are presented in the following sections.

An important assumption in the algorithm proposed above is that the scene is rigid between two consecutive frames. The inlier detection step, described in section 2.1.7 below, also assumes a rigid scene. The envisioned drone system, as described in section 1.1, is meant to aid in the mapping of a maze together with other robots. The robots will move around in the maze, making the assumption of a rigid scene possibly false. This assumption and how it may affect the pose estimation will be discussed in section 8.2.3.

2.1.2 Feature detection

Features from Accelerated Segment Test (FAST) feature detection One possible way of detecting features in an image is the FAST feature detector [41], which is particularly suitable for when real-time performance is needed (whereas other feature detectors such as Scale-Invariant Feature Transform (SIFT) [24] is too computationally expensive to be able to operate in real-time). A brief summary of the FAST feature detector is presented below. For more detail, see the original paper.

First, choose a suitable threshold value t. Then, for every pixel p in the image, consider a circle around p consisting of 16 pixels, see figure 2.1.

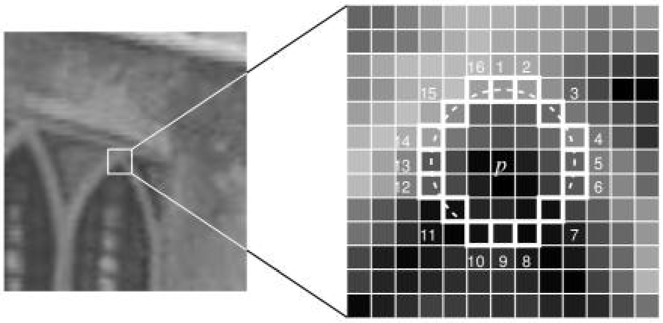


Figure 2.1: A circle of 16 pixels around a pixel p. Image taken from the original paper

Now, if the intensity of every pixel in a set of N contiguous pixels in the circle are all brighter than the intensity of the center pixel p plus the threshold t, or the intensity of all the N contiguous pixels in the circle are all darker than I_p minus the threshold t, then the feature candidate is considered to be a corner. N is usually chosen as 12. A high-speed test to exclude non-corners are performed. Four pixels are examined, where at least three of them need to pass the test for the feature candidate to be a corner:

1. Examine pixel 1, 9, 5 and 13 in the circle.
2. First, see if both I_1 and I_9 (the intensity of pixel 1 and 9) are between $[I_p - t, I_p + t]$. If this is the case, then p is not a corner.
3. Do the same for I_5 and I_{13}.
4. If none of the four pixels are within $[I_p - t, I_p + t]$, the pixel p is marked as a corner.
5. If exactly three of these pixels are within the range, the test is somewhat inconclusive, and the rest of the pixels in the circle are examined.

This is a fast way of detecting corner, however with several weaknesses:

1. The test does not perform as well when $N < 12$
2. The efficiency of the detector depends on the order that the pixels are checked and where in the image the corners are located

3. Multiple features are detected adjacent to one another

The first two weaknesses are solved using a machine learning approach, whereas the third weakness is solved using non-maximum suppression. See the paper for further details.

2.1.3 Feature tracking

There are two main approaches for finding corresponding features in two images, feature tracking and feature matching [44]. Feature tracking detects features in an image (i. e. using FAST) and tracks them to the next image using some local search technique, such as correlation. Feature matching, however, detects features in both images, constructs descriptors for each feature and match them based on some similarity metric between the descriptors. Feature tracking is suitable for small motion, while feature matching is more suitable for larger motion or when a considerable viewpoint change is expected. In the application developed in this project, the images will be taken consecutively at a relatively high frame rate, and since the motion is assumed to be small, feature tracking is the approach that will be focused on. However, feature matching will also be presented briefly, as it is a relevant alternative to feature tracking (as discussed in section 10.1.2).

Optical flow analysis Feature tracking can be done using optical flow analysis, which is the problem of estimating the motion of pixels from one frame at time t to the next frame at time $t + 1$. Optical flow analysis makes the following assumptions [2]:

1. The intensities of the pixels stay constant between consecutive frames
2. The motion between consecutive frames are small

For a pixel $p(x, y)$, moving a distance $(\Delta x, \Delta y)$ during a time interval Δt, we get the following using the first assumption:

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t) \tag{2.1}$$

If we assume that the motion is small, we can expand the right hand side by a first-order Taylor series expansion:

$$I(x + \Delta x, y + \Delta y, t + \Delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t \tag{2.2}$$

Inserting 2.2 into 2.1, we get:

$$I(x, y, t) = I(x, y, t) + \frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t \tag{2.3}$$

Dividing 2.3 by Δt and subtracting $I(x, y, t)$ from both sides yields the following:

$$\frac{\partial I}{\partial x}\frac{\Delta x}{\Delta t} + \frac{\partial I}{\partial y}\frac{\Delta y}{\Delta t} + \frac{\partial I}{\partial t} = 0 \tag{2.4}$$

which finally can be written as:

$$I_x V_x + I_y V_y = -I_t \tag{2.5}$$

where

$$\begin{aligned} I_x &= \frac{\partial I}{\partial x}, I_y = \frac{\partial I}{\partial y} \\ V_x &= \frac{\Delta x}{\Delta t}, V_y = \frac{\Delta y}{\Delta t} \end{aligned}$$

Equation 2.5 is called the Optical Flow equation. The equation consists of two unknowns, V_x and V_y, and we need other assumptions and constraints to be able to solve it.

Estimating optical flow using the Lucas-Kanade method A commonly used method for estimating optical flow is the Lucas-Kanade method [25]. It assumes that for a pixel, all the neighbouring pixels are moving with approximately the same speed and direction as the pixel in consideration, and solves the optical flow for all the neighbouring pixels using a least-squares method. In other words, equation 2.5 holds not only for the pixel in consideration, but also for the neighbouring pixels. By considering a patch of n pixels centered around the pixel $p(x, y, t)$ in consideration, we get the following set of equations:

$$\begin{aligned} I_x(q_1)V_x + I_y(q_1)V_y &= -I_t(q_1) \\ I_x(q_2)V_x + I_y(q_2)V_y &= -I_t(q_2) \\ &\vdots \\ I_x(q_n)V_x + I_y(q_n)V_y &= -I_t(q_n) \end{aligned}$$

where $q_1, q_2, \ldots, q_n$ are the pixels in the patch and $I_x(q_i)$, $I_y(q_i)$ and $I_t(q_i)$ are the partial derivatives of the image I with respect to x, y, and t, respectively, evaluated at pixel q_i. For $n = 9$, we have 9 equations and just two unknowns. The set of equations can be written as

$$Av = b \tag{2.6}$$

with

$$A = \begin{bmatrix} I_x(q_1) & I_y(q_1) \\ I_x(q_2) & I_y(q_2) \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ I_x(q_n) & I_y(q_n) \end{bmatrix} \quad v = \begin{bmatrix} V_x \\ V_y \end{bmatrix} \quad b = \begin{bmatrix} -I_t(q_1) \\ -I_t(q_2) \\ \cdot \\ \cdot \\ \cdot \\ -I_t(q_n) \end{bmatrix} \tag{2.7}$$

The set of equations is solved by a least-squares principle, where the final solution to the problem is found by solving the following equation (see the original paper for more details):

$$\begin{bmatrix} V_x \\ \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i) I_y(q_i)^2 \\ \\ \sum_i I_y(q_i) I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i) I_t(q_i) \\ \\ -\sum_i I_y(q_i) I_t(q_i) \end{bmatrix} \tag{2.8}$$

2.1.4 Feature matching

Feature matching basically consists of detecting features in the images, creating a descriptor for each feature, and then find the corresponding points by comparing the descriptors. The simplest way for doing this is to compare all the feature descriptors in the first image with the feature descriptors in the next image and find corresponding points using some sort of similarity measure of the descriptors. A basic FAST feature detector, described in section 2.1.2, only detects feature points and does not create descriptors. SIFT, however, does both. It detects blobs, unlike FAST which detects corners, and the similarity measure between SIFT descriptors is the euclidean distance. An overview of SIFT (and of many other detectors as well) can be found in [46].

Oriented FAST and rotated BRIEF (ORB) [42] is another algorithm that both detects and describes features and, like the FAST detector, detects corners. ORB basically detects features using FAST and creates a Binary Robust Independent Elementary Features (BRIEF) descriptor also invariant to rotation (unlike the original BRIEF descriptor). ORB was developed to provide a faster and less computationally expensive alternative to SIFT (as the title of the original paper suggests). Generally, corners are less distinctive than blobs, but faster to compute [44]. Whether to detect corners or blobs depends on the application.

When performance is essential, such as in this project, corner detection may be the best approach. ORB, which is capable of being used for real-time performance according to [42], is therefore a good candidate for a suitable feature detector for feature matching.

2.1.5 Triangulation

Triangulation, in the field of computer vision, is the method of determining the 3D world location of a point in a set of stereo images, given the disparity and the geometry of the stereo setting. A brief summary of the subject is presented below. Further details can be found in [17].

A camera model often used for 3D reconstruction is the pinhole camera model, which is a simple mathematical model of a camera without a lens and a single, small aperture which describes the relationship between a point $p(x, y)^T$ in the image plane and the corresponding 3D location of the same point, $P(X, Y, Z)^T$. The geometry of the pinhole camera model can be seen in figure 2.2.

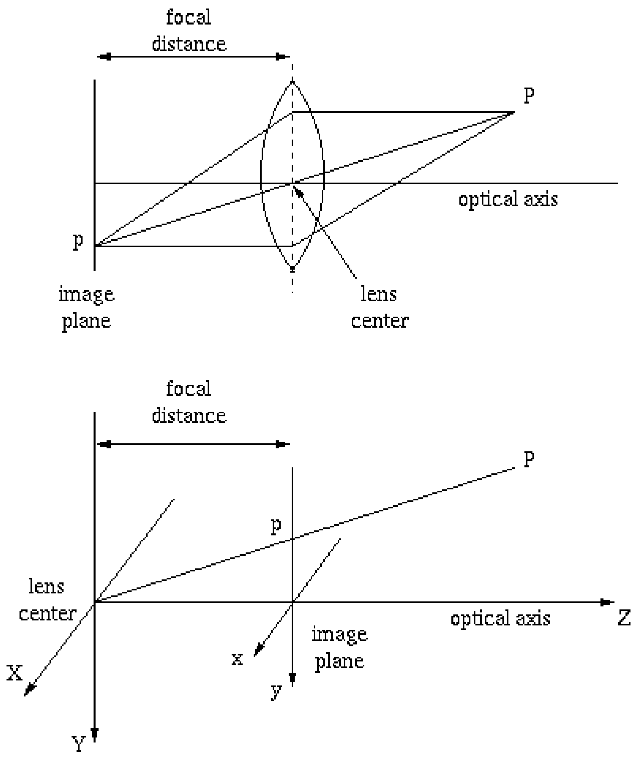


Figure 2.2: Geometry of the pinhole camera model

The relationship between a 3D point and the corresponding point in the image plane is given by:

$$x = \frac{fX}{Z} \tag{2.9}$$

$$y = \frac{fY}{Z} \tag{2.10}$$

where f is the focal length, i. e. the distance from the optical center of the camera and the image plane along the optical axis.

For a stereo setup of two cameras, where it is assumed that the optical axes of the cameras are parallel and that the images are undistorted and rectified according to their calibration parameters (detailed information about stereo camera calibration and image rectification can be found in [47] and [48]), the geometry can be seen in the following figure:

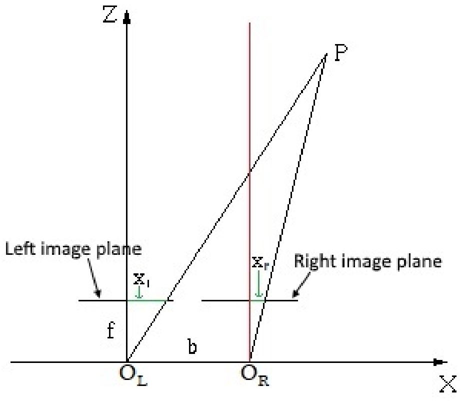


Figure 2.3: Geometry of a stereo camera setup where the optical axes are parallel

where O_L and O_R is the optical center of the left and right camera, respectively, and the red line is the optical axis of the right lens. x_l and x_r is the distance from the optical axis to the same 3D point in the left and right image plane, respectively. The difference between these two distances is called disparity:

$$d = x_l - x_r \tag{2.11}$$

A basic algorithm for calculating a disparity map, e. g. the disparity for all the points in a stereo-pair image, is described in the next section. The 3D coordinate $P(X, Y, Z)^T$ of a point $p(x_l, y_l)$ in the left camera can then be calculated using the following equations:

$$Z = \frac{bf}{d} \tag{2.12}$$

$$X = \frac{x_l}{f}Z = \frac{x_l b}{d} \tag{2.13}$$

$$Y = \frac{y_l}{f}Z = \frac{y_l b}{d} \tag{2.14}$$

where b is called the baseline.

2.1.6 Block matching for disparity map calculation

A standard method for computing the disparity map of a stereo-pair image is the simple block matching algorithm [27]. It basically takes a region of pixels in the left image and searches for the closest matching region in the right image. Assuming that the images are rectified, a feature in the left image will appear in the right image along a horizontal line going through the point in the left image (the epipolar line), such as in figure 2.4.

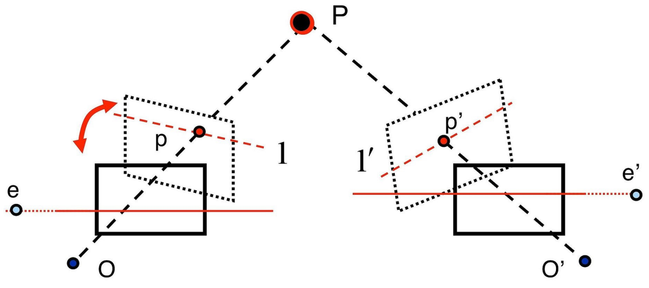
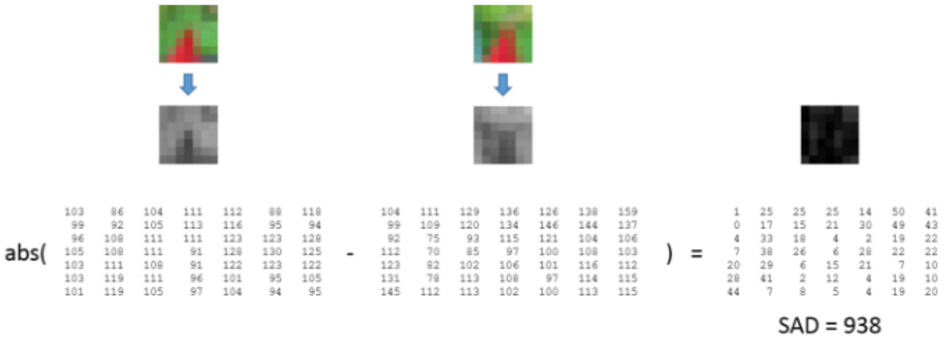


Figure 2.4: Example of a stereo-image pair, where the images are rectified such that the epipolar lines are horizontal [48]

The comparison between regions in the left and right image is simply done using the SAD. The SAD is computed by subtracting the grayscale value of each pixel in the left region with the grayscale value of each corresponding pixel in the right image and summing up all the differences to get one single number. Figure 2.5 illustrates the concept:



$$\text{abs}\left(\begin{matrix} 103 & 86 & 104 & 111 & 112 & 88 & 118 \\ 99 & 92 & 105 & 113 & 116 & 95 & 94 \\ 96 & 108 & 111 & 111 & 123 & 123 & 128 \\ 105 & 108 & 111 & 91 & 128 & 130 & 125 \\ 103 & 111 & 108 & 91 & 122 & 123 & 122 \\ 103 & 119 & 111 & 96 & 101 & 95 & 105 \\ 101 & 119 & 105 & 97 & 104 & 94 & 95 \end{matrix} - \begin{matrix} 104 & 111 & 129 & 136 & 126 & 138 & 159 \\ 99 & 109 & 120 & 134 & 146 & 144 & 137 \\ 92 & 75 & 93 & 115 & 121 & 104 & 106 \\ 112 & 70 & 85 & 97 & 100 & 108 & 103 \\ 123 & 82 & 102 & 106 & 101 & 116 & 112 \\ 131 & 78 & 113 & 108 & 97 & 114 & 115 \\ 145 & 112 & 113 & 102 & 100 & 113 & 115 \end{matrix}\right) = \begin{matrix} 1 & 25 & 25 & 25 & 14 & 50 & 41 \\ 0 & 17 & 15 & 21 & 30 & 49 & 43 \\ 4 & 33 & 18 & 4 & 2 & 19 & 22 \\ 7 & 38 & 26 & 6 & 28 & 22 & 22 \\ 20 & 29 & 6 & 15 & 21 & 7 & 10 \\ 28 & 41 & 2 & 12 & 4 & 19 & 10 \\ 44 & 7 & 8 & 5 & 4 & 19 & 20 \end{matrix}$$

SAD = 938

Figure 2.5: The computation of the SAD of two, possibly matching, regions. Image taken from [27]

The region with the lowest SAD are considered as a match, and the disparity is computed according to equation 2.11.

The SGBM algorithm, proposed in [15], is another block matching algorithm for computing disparity maps. SGBM is the method mostly used in this project since it seems to give better results than simple block matching, as will be discussed in section 7.4. Both algorithms are provided by the SDK that comes with the stereo camera, which is further described in section 6.1.5.

Disparity maps are usually represented as a grayscale image, where the closer the objects are to the camera, the higher the grayscale values are (brighter/more white). Lower grayscale value (darker image) means objects that are further away from the camera. Complete black spots in the disparity map represent pixels with invalid disparity values. Examples of disparity maps will be presented in chapter 7.

2.1.7 Outlier rejection/inlier detection: Max clique approximation

It is safe to assume that the set of point correspondences found during feature detection and tracking contains a lot of outliers, which will result in loss of accuracy if not dealt with before using them in the further calculations of the visual odometry algorithm. A method for dealing with outliers is proposed in [16], which is actually an inlier detection method. It assumes that the scene is rigid, i. e. the scene does not change from time instance t to time $t+1$. Then, for two 3D feature points detected in the frame at time t and tracked to the next frame at time $t+1$, the relative distance between them should stay the same:

$$|P_i - P_k| = |C_i - C_k| \qquad \text{with } i, k = 1...n. \tag{2.15}$$

Where P_i and P_k are two points in the first frame, and C_i and C_k are the same points detected in the next frame. If this distance is not the same, an error have occured either during the feature detection or tracking, or something went wrong in the triangulation. We want the largest subset of points where every point is consistent with each other.

A problem is that equation 2.15 are almost always wrong for noisy values. [16] deals with this problem by introducing a measure of error which takes into account that in stereo vision, errors propagate non-linearly from the image plane into the 3D-position of a point. Two corresponding points are set to be consistent if the distance between them are less than this error. Se the paper for more details. The matrix m is then created according to the following::

$$m_{ik} = \begin{cases} 1, & \text{if the corresponding points } P_i, P_k, C_i \text{ and } C_k \text{ are consistent} \\ 0, & \text{otherwise} \end{cases}$$

When the matrix m is finally created, we want to find the largest subset of points where all the points are consistent with each other. This is a NP-complete problem, for which no fast solution to the problem exists. However, a good approximation is to find a large subset, if not the largest, of consistent points. Details of how this is done can be found in the paper.

2.1.8 Motion estimation

As described in [43], the relative motion between two consecutive frames is given by the transformation matrix T_k, and can be computed from the 3D-to-2D correspondences X_{k-1} and p_k, where X_{k-1} represent the 3D point at frame $k-1$ corresponding to the 2D point p_k at frame k. With a set of n 3D-to-2D correspondences p and $\hat{p}$ and the intrinsic camera parameters K of the calibrated stereo camera, the transformation is included in the perspective projection model for a camera:

$$sp = KT\hat{p} \tag{2.16}$$

which can be written as

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & \gamma & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.17}$$

where p and $\hat{p}$ are in homogeneous coordinates, the intrinsic parameters K consists of the focal lengths of the camera, f_x and f_y, skew-parameter γ (often equal to zero) and the principle point (u_0, v_0), and the transformation matrix T as $T = [R|t]$, where R is a rotation matrix and t is a translation vector.

T_k can be found by minimizing the image reprojection error according to the following equation:

$$\arg\min_{T_k} \sum_i \left\| p_k^i - \hat{p}_{k-1}^i \right\|^2 , \tag{2.18}$$

where $\hat{p}_{k-1}^i$ corresponds to the reprojection of X_{k-1}^i into the image plane at frame k according to the transformation T_k. This is known as the Perspective-n-Point (PnP) problem. A minimum of three 3D-to-2D correspondences is needed to solve this problem, which is then called *perspective from three points* (P3P), but in this case, four possible solutions are returned which can be disambiguated using at least one extra point. Therefore, at least four 3D-to-2D point correspondences are needed to get a unique solution the PnP problem. See [43] for more details.

2.1.9 Concatenation of transformations

The solution of the PnP problem is a rotation matrix and a translation vector that represents the position and orientation of the frame k relative to the previous frame $k-1$. Using the notation and rigid body kinematics theory found in [11], this can be written as the following rotation matrix and translation vector:

$$R_{k-1}^{k}$$
$$t_{k,k-1}^{k}$$

where R_{k-1}^{k} is the rotation matrix from k to $k-1$ and $t_{k-1,k}^{k-1}$ represents the vector from the origin of frame $k-1$ to the origin of frame k, written with respect to the frame $k-1$.

For a moving frame, we want to concatenate the transformations at each iteration such that we get the total transformation from the initial frame to the latest frame, i. e. R_k^0 and $t_{0,k}^0$.

At each iteration we have the previous transformation given by R_{k-1}^0 and $t_{0,k-1}^0$. This can be written as the homogeneous transformation matrix

$$T_{k-1}^{0} = \begin{bmatrix} R_{k-1}^{0} & t_{0,k-1}^{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.19}$$

(**note**: for the first iteration, $R_{k-1}^{0} = I$, where I is the identity matrix, and $t_{0,k-1}^{0} = \mathbf{0}$)

If we reverse the transformation, it can be written as

$$T_{0}^{k-1} = \begin{bmatrix} R_{0}^{k-1} & t_{k-1,0}^{k-1} \\ \mathbf{0}^T & 1 \end{bmatrix} = \begin{bmatrix} (R_{k-1}^{0})^T & -(R_{k-1}^{0})^T t_{0,k-1}^{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.20}$$

The homogeneous transformation from frame k to frame $k-1$ can be written as

$$T_{k-1}^{k} = \begin{bmatrix} R_{k-1}^{k} & t_{k,k-1}^{k} \\ \mathbf{0}^T & 1 \end{bmatrix} \tag{2.21}$$

where R_{k-1}^{k} and $t_{k,k-1}^{k}$ are the solution to the PnP problem at the current iteration. The homogeneous transformation from frame k to frame 0 can now be written as

$$\begin{aligned} T_0^k &= T_{k-1}^{k} T_0^{k-1} \\ &= \begin{bmatrix} R_{k-1}^{k} & t_{k,k-1}^{k} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} (R_{k-1}^{0})^T & -(R_{k-1}^{0})^T t_{0,k-1}^{0} \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} R_{k-1}^{k}(R_{k-1}^{0})^T & -R_{k-1}^{k}(R_{k-1}^{0})^T t_{0,k-1}^{0} + t_{k,k-1}^{k} \\ \mathbf{0}^T & 1 \end{bmatrix} \\ &= \begin{bmatrix} R_0^k & t_{k,0}^{k} \\ \mathbf{0} & 1 \end{bmatrix} \end{aligned}$$

Now we have the complete transformation from the latest frame to the initial frame, represented by R_0^k and $t_{k,0}^{k}$. The transformation with respect to the initial frame can be computed as

$$R_k^0 = (R_0^k)^T \tag{2.22}$$

$$t_{0,k}^0 = -(R_0^k)^T t_{k,0}^k \tag{2.23}$$

2.2 Communication with the server

The protocol used for communication with the server in this project was developed in [23] and later ported to regular Ubuntu in [18], as mentioned in section 1.1. The protocol consists of multiple layers. A network layer specifies how data transmitted by the BLE dongles shall be formatted, and a layer above the network is also included to ensure reliable communication, handling situations such as loss of connection between the dongles, packed loss etc. Furthermore, the protocol is using a handshake routine based on Transmission Control Protocol (TCP) during the initial connection between a robot and the server. A more detailed summary about the protocol can be found in [18]. For a complete description, see the original thesis [23].

Chapter 3

Hardware specification

3.1 Introduction

This chapter is a description of the hardware used throughout this project. Visual odometry is something that requires high computational performance, and the hardware running the algorithm needs a lot of processing power in order to run the algorithm at a sufficient frequency. The following hardware was chosen with that in mind.

3.2 Nvidia Jetson TX1

The Nvidia Jetson TX1 Developer Kit is a full-featured development platform for visual computing developed by Nvidia [36]. It is a powerful computer able to do demanding tasks in the fields of artificial intelligence and computer vision, and is small and light enough for mobile applications, with power requirements such that it can be powered by a battery. It comes pre-compiled with Linux and supports a lot of Application Programming Interface (API)'s as well as Nvidia's complete developer tool chain [33]. The developer kit is built around the Jetson TX1 Module. According to Nvidia, "the world's first supercomputer on a module, Jetson TX1 is capable of delivering the performance and power efficiency needed for the latest visual computing applications" [34]. This makes it very suitable for the scope of this project, and opens up several possibilities for future projects as well.

Main specifications:

- **Graphics Processing Unit (GPU):** NVIDIA Maxwell™ GPU with 256 NVIDIA® CUDA® Cores

- **Central Processing Unit (CPU):** Quad-core ARM® Cortex®-A57 MPCore
- **Memory:** 4 GB LPDDR4
- **Internal storage:** 16 GB eMMC 5.1 Flash Storage
- **Power (max):** 5V @ 7A (35 W) [35]

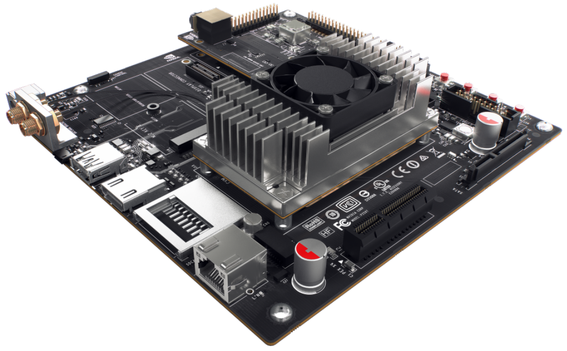


Figure 3.1: Nvidia Jetson TX1 Developer Kit

Other technical specifications: [35]

- 1 x USB 3.0 Type A
- 1 x USB 2.0 Micro AB (supports recovery and host mode)
- High-Definition Multimedia Interface (HDMI)
- GPIOs, I2C, I2S, SPI*
- Full-Size SD
- PCI-E x4
- Display Expansion Header*
- Camera Expansion Header*
  - *I/O expansion headers: see [35] for header specification.

The full block diagram of the features of the TX1 developer kit can be seen in figure 3.2.

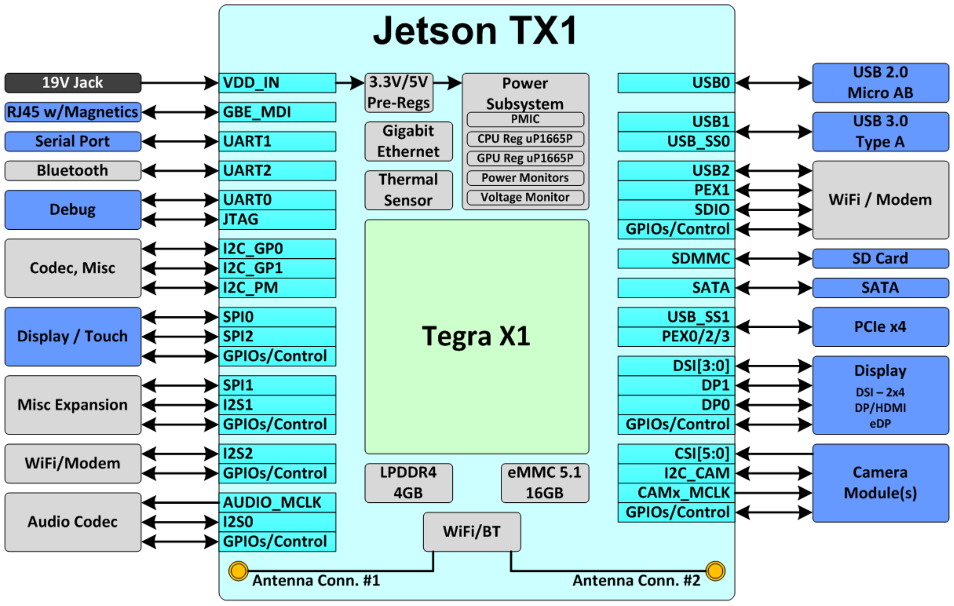


Figure 3.2: Jetson Carrier Board Block Diagram

3.3 Duo M Stereo Camera

The Duo M is a lightweight stereo camera produced by Code Laboratories ideal for mobile applications where small size and low weight is essential. The DUO M is fully compatible with the Nvidia Jetson TX1 developer kit.

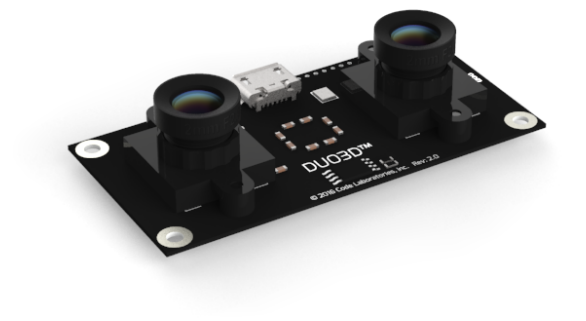


Figure 3.3: Duo M stereo camera

Main specifications [8]:

- Calibrated stereo sensor
- Monochrome/Global shutter
- Configurable framerates: 0.1-3000+ Frames Per Second (FPS)
- Configurable resolutions: Ranging from 752x480 @ 45FPS to 320x120 @ 320 FPS
- Two 170° wide angle m8 lenses
- Focal length: 2.0mm - 2.1mm
- Baseline: 30mm
- Pixel size: 6.0 x 6.0 μm
- Power consumption: 2.5 Watt @ +5V DC from USB
- Dimensions: 52x52x12 mm
- Interface: USB 2.0
- Recommended operating range: 0.23-2.5m

3.4 nRF51422 BLE dongle

One of the achievements in [18] was a Raspberry Pi 3 Model B able to communicate with the server in the same way and using the same BLE dongle as the other robots in the LEGO-robot project. The dongle was a nRF51422 BLE dongle developed by Nordic Semiconductor, and the same dongle is used in this project to have the Nvidia Jetson TX1 communicate with the server in the same way as the Raspberry Pi in [18].

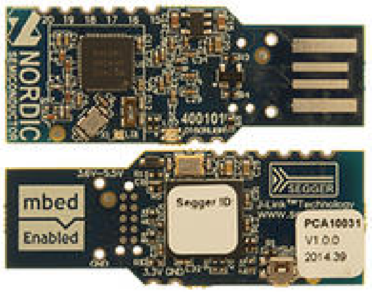


Figure 3.4: Nordic Semiconductor nRF51422 BLE dongle

3.5 SD card

The internal memory of the TX1 is only 16GB, which is not sufficient for this project. A 64GB SanDisk Class 10 UHS-I microSDXC card was therefore used throughout the project. How to make the TX1 boot and run from the SD card is described in section 5.4.

Figure 3.5: 64GB SanDisk microSDXC Class 10 UHS-1 80 MB/s

Chapter 4

Hardware setup

4.1 System architecture

Only a few of the features displayed in the full block diagram of the TX1 developer kit in figure 3.2 are used in this project. A simplified version of the overall system architecture, based on the full block diagram, can be seen in figure 4.1.

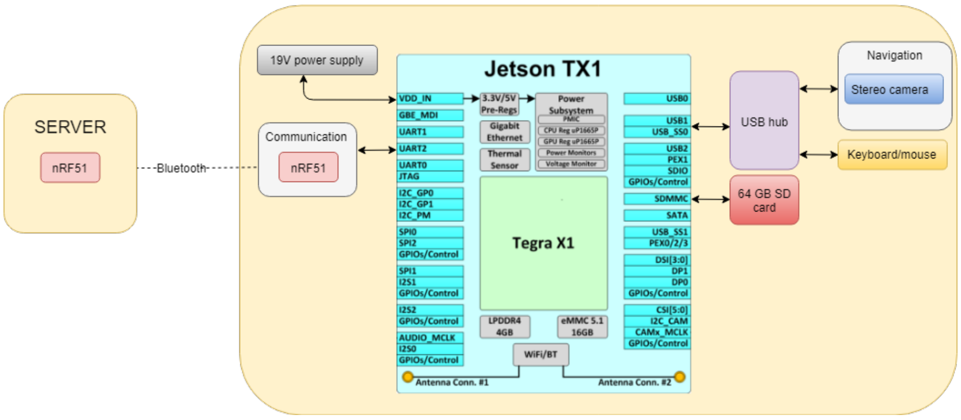


Figure 4.1: Complete (simplified) system architecture for the system

4.2 System setup

A more detailed description of how each component is connected to the Nvidia Jetson TX1 is presented below.

4.2.1 IO devices

Keyboard/mouse (and monitor) The TX1 is more or less plug-and-play, since it is delivered with a common distribution of Linux already pre-compiled. In other words, it works right out of the box, and it is possible to just connect it to a monitor (using a HDMI cable), connect a keyboard and mouse to the USB input, plug in the power adapter and press the power on button to start using it. The TX1 has only one USB input, however, and a USB-hub was proven necessary.

External storage The TX1 has only got 16 GB of internal storage, which is not sufficient for this project. However, it is possible to have the TX1 run from an external SD card. A 64 GB microSDXC card is therefore used, which is plugged into the SD card reader on the TX1 (using a microSD-SD adapter). The procedure for how to copy the initial content on the internal storage over to the SD card and have the TX1 boot from the SD card is described in section 5.4.

4.2.2 Duo M stereo camera

The Duo M stereo camera is connected to the TX1 using a regular microUSB-USB cable, as can be seen in figure 4.2. Since the development during this project has been done on the device, a keyboard and mouse connected to the TX1 was necessary at all times, along with the stereo camera. As mentioned in the previous section, a USB-hub was used throughout this project, and luckily, the Duo M worked right away without any problems when connected to the USB hub.

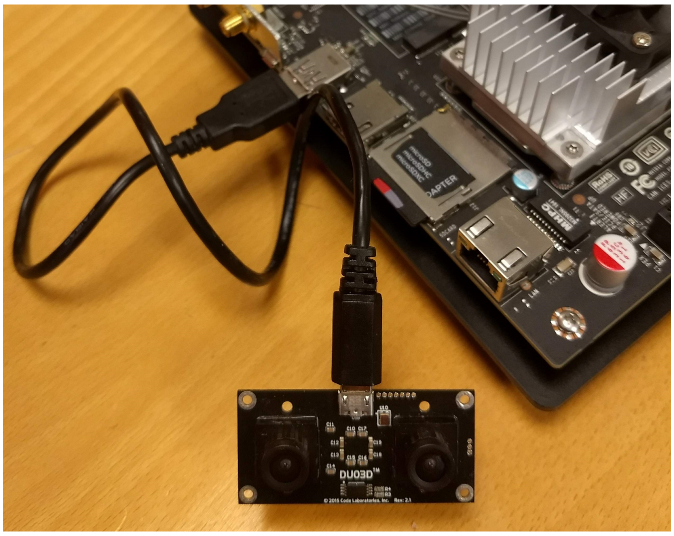


Figure 4.2: Duo M stereo camera connected to the TX1 with a USB cable

The coordinate frame of the stereo camera, set by the Dense3D module in the DUO SDK (which is described later in section 6.1.5), is located with its origin at the left lens and axes as shown in figure 4.3. The camera is attached to a 3D printed case, later described in section 4.3.

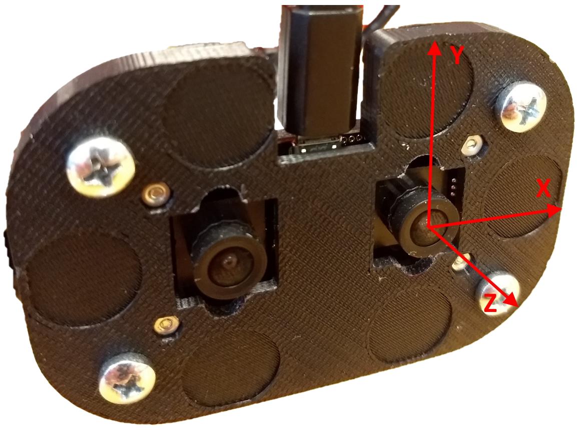


Figure 4.3: Duo M stereo camera and the coordinate frame defined by the duo SDK

4.2.3 nRF51422 BLE dongle/UART

The nRF51422 BLE dongle used for communication with the server is connected to one of the UART connections on the TX1. The TX1 has several UARTs, but the one suitable for the BLE dongle is the "UART1", which is the 6 pin GPIO header marked as "J17" on the TX1, as seen in figure 4.4.

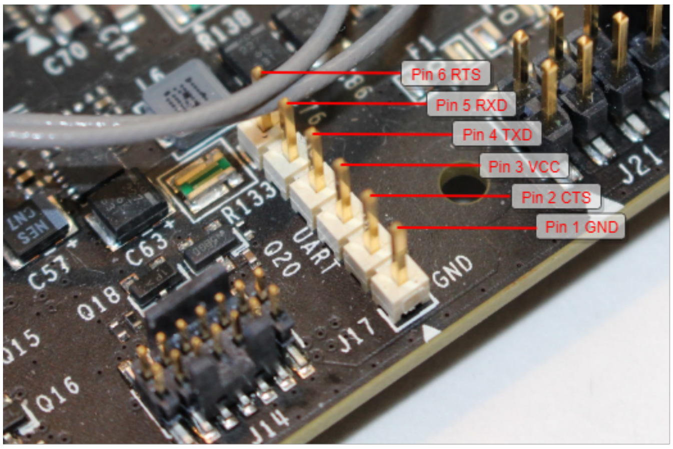


Figure 4.4: The UART1 pins on the Jetson TX1

The physical wiring between the BLE dongle and the UART can be seen in figure 4.5. Only the four pins "Pin 1 GND", "Pin 3 VCC", "Pin 4 TXD" and "Pin 5 RXD" are needed (ground, power, transmitting pin and receiving pin, respectively).

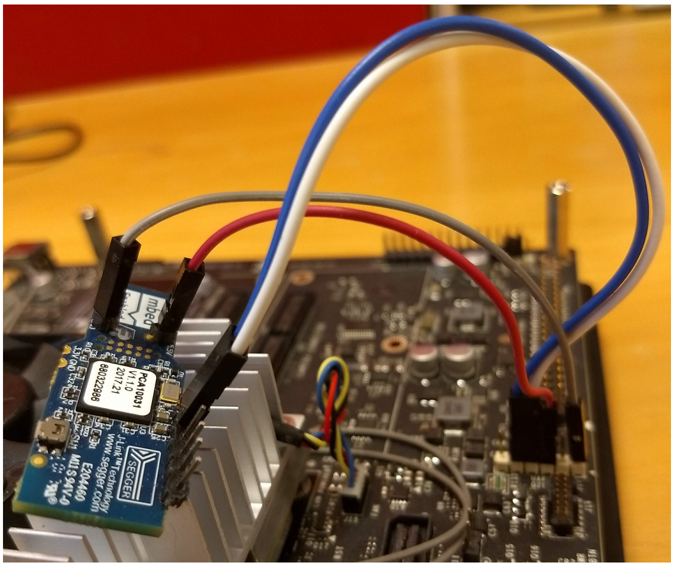


Figure 4.5: The nRF51422 BLE dongle connected to UART1 on the TX1

4.3 Complete hardware setup

It is advantageous to have every hardware component connected and attached to the TX1 when performing tests of the system, and necessary components were made by an employee at a workshop at Norwegian University of Science and Technology (NTNU) in order to achieve this. The components, including a 3D printed case custom made for the DUO M camera seen in figure 4.3, made it possible to attach the system on a camera stand, together with the possibility to have the stereo camera pointing both outwards and vertically downwards. Figure 4.6 shows the complete system with every component attached and with the stereo camera pointing outwards. Figure 4.7 shows the complete system while attached to a camera stand. Having the system attached to a camera stand makes it easier to move the system in a way that can be replicated, which may be useful when testing the system in the future.

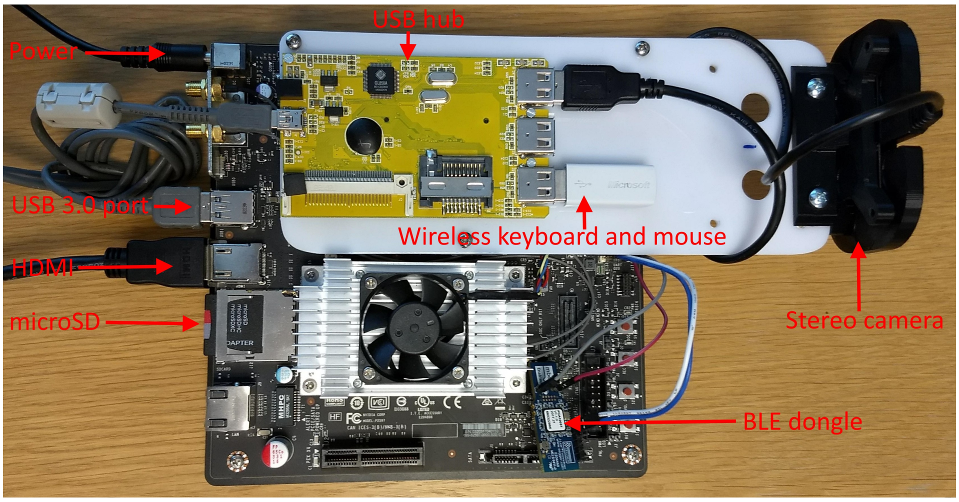


Figure 4.6: The complete system setup, stereo camera pointing outwards

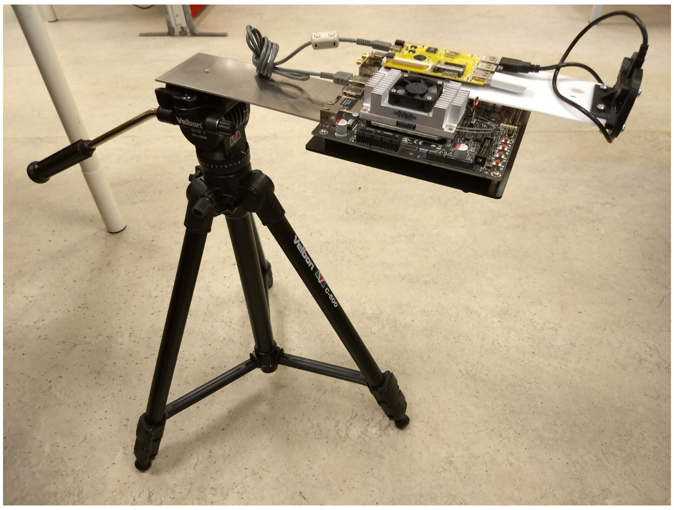


Figure 4.7: The complete system setup, stereo camera pointing outwards

The complete system while the camera is pointing downwards can be seen in figure 4.8.

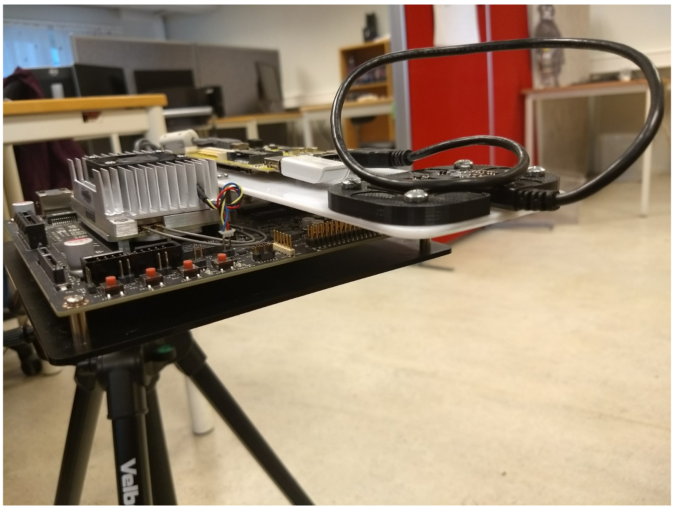


Figure 4.8: The complete system setup, stereo camera pointing downwards

4.4 Test setup

4.4.1 Feature points

Figure 4.9 shows the setup during a simple test of the feature detection, filtering and tracking steps of the algorithm, as well as the computation of the disparity map. It was made sure that the conditions for feature detection was good, i. e. a scene with good lighting conditions and many distinct features. The results are presented and discussed in chapter 7.

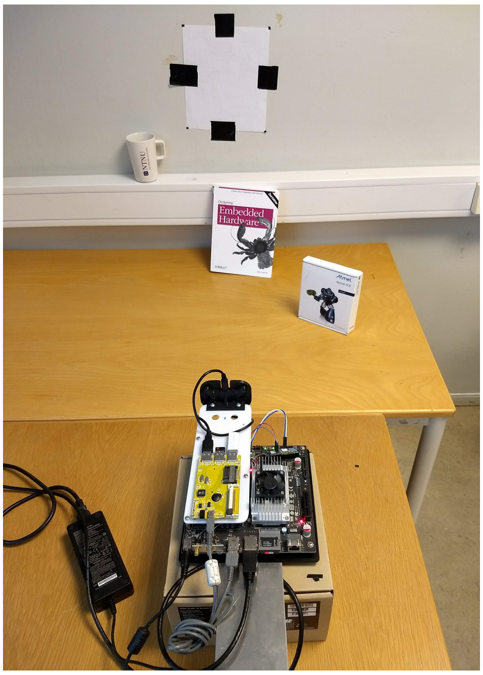


Figure 4.9: The TX1 during the testing of the feature point tests

4.4.2 Motion estimation with the OptiTrack Motion Capture system

The motion tests, described and discussed in chapter 8, was performed in the room B33 "Bevegelseslab" at NTNU. The "Bevegelseslab" has a OptiTrack motion capture system, which consists of a system of 16 cameras able to track the movement of a rigid body with sub-millimeter accuracy [29]. To track an object, retroreflective markers needs to be attached to the object. A minimum of three markers is required to track an object in 3D. However, since three cameras need to see at least three markers at all times in able to define the rigid body in three coordinates, using 4-12 markers is recommended. The TX1 with 7 markers attached is shown in figure 4.10. The pose estimates from the OptiTrack system was used as the "ground truth" during the testing of the visual odometry algorithm.

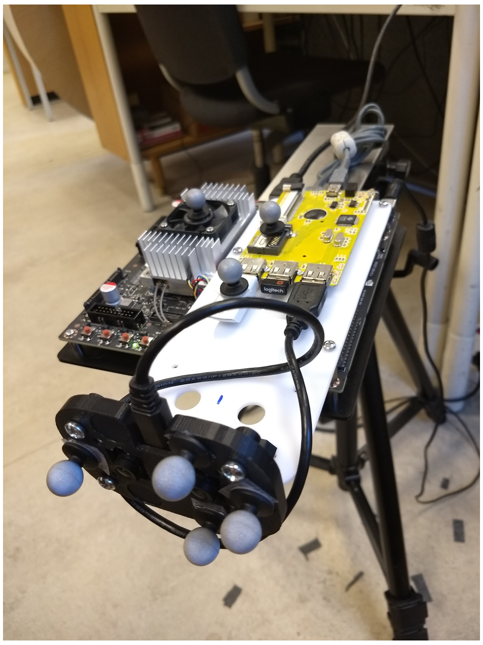


Figure 4.10: Retroreflective markers attached to the TX1

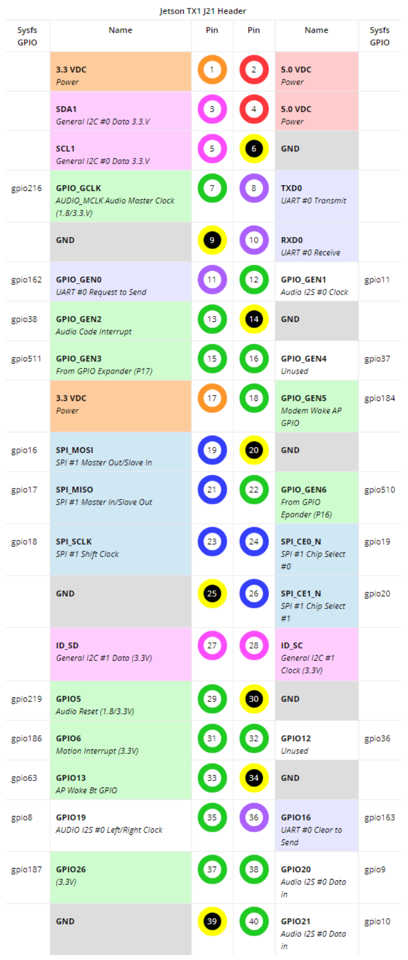
Chapter 5

Software setup on the Nvidia Jetson TX1

5.1 Debug using serial console

The serial console on the TX1 is valuable especially since it makes it possible to control the TX1 when a keyboard, mouse and monitor is not available, but it also gives the possibility to see whats going on during boot and even to choose menu entries for boot images during boot. The serial console can be found on the J21 GPIO header on the TX1. A serial console cable similar to the one in figure 5.1 is needed. The J21 GPIO header can be seen in figure 5.2. With a cable identical to the one in figure 5.1, the white cable is connected to pin 10 and the green cable is connected to pin 8 on the GPIO. With the USB connected to a free USB port on a computer, the TX1 can then be controller from the host PC using software like Termite, a simple RS232 terminal compatible with both Linux and Windows [9]. See [13] for further details on how to do this.

Figure 5.1: USB to UART serial console cable


Jetson TX1 J21 Header

Sysfs GPIO	Name	Pin	Pin	Name	Sysfs GPIO
	3.3 VDC *Power*	1	2	**5.0 VDC** *Power*	
	SDA1 *General I2C #0 Data 3.3 V*	3	4	**5.0 VDC** *Power*	
	SCL1 *General I2C #0 Data 3.3 V*	5	6	**GND**	
gpio216	**GPIO_GCLK** *AUDIO_MCLK Audio Master Clock (1.8/3.3 V)*	7	8	**TXD0** *UART #0 Transmit*	
	GND	9	10	**RXD0** *UART #0 Receive*	
gpio162	**GPIO_GEN0** *UART #0 Request to Send*	11	12	**GPIO_GEN1** *Audio I2S #0 Clock*	gpio11
gpio38	**GPIO_GEN2** *Audio Code Interrupt*	13	14	**GND**	
gpio511	**GPIO_GEN3** *From GPIO Expander (P17)*	15	16	**GPIO_GEN4** *Unused*	gpio37
	3.3 VDC *Power*	17	18	**GPIO_GEN5** *Modem Wake AP GPIO*	gpio184
gpio16	**SPI_MOSI** *SPI #1 Master Out/Slave In*	19	20	**GND**	
gpio17	**SPI_MISO** *SPI #1 Master In/Slave Out*	21	22	**GPIO_GEN6** *From GPIO Epander (P16)*	gpio510
gpio18	**SPI_SCLK** *SPI #1 Shift Clock*	23	24	**SPI_CE0_N** *SPI #1 Chip Select #0*	gpio19
	GND	25	26	**SPI_CE1_N** *SPI #1 Chip Select #1*	gpio20
	ID_SD *General I2C #1 Data (3.3V)*	27	28	**ID_SC** *General I2C #1 Clock (3.3V)*	
gpio219	**GPIO5** *Audio Reset (1.8/3.3V)*	29	30	**GND**	
gpio186	**GPIO6** *Motion Interrupt (3.3V)*	31	32	**GPIO12** *Unused*	gpio36
gpio63	**GPIO13** *AP Wake Bt GPIO*	33	34	**GND**	
gpio8	**GPIO19** *AUDIO I2S #0 Left/Right Clock*	35	36	**GPIO16** *UART #0 Clear to Send*	gpio163
gpio187	**GPIO26** *(3.3V)*	37	38	**GPIO20** *Audio I2S #0 Data in*	gpio9
	GND	39	40	**GPIO21** *Audio I2S #0 Data in*	gpio10

Figure 5.2: J21 GPIO header. Image from [20]

5.2 OS and NDIVIA JetPack SDK

The Jetson TX1 comes pre-compiled with a distribution of Linux, but during this project an older version was needed for the Duo M stereo camera to work with the TX1. Installation of different Linux versions on the TX1 is done via a host computer running an installation software called "JetPack". Different versions of JetPack install different distributions of Linux. For instance, the latest version of JetPack (as of june 2018), JetPack 3.2, comes with the latest version of Linux For Tegra (L4T), L4T 28.2. This will install Ubuntu 16.04 on the TX1. The host computer needs to be running Ubuntu as well, and for JetPack 3.2, running Ubuntu 16.04 on the host computer is recommended, as can be seen in the JetPack 3.2 release notes [32].

The DUO M stereo camera currently requires that Linux is installed on the TX1 using

JetPack 3.0 with L4T 24.2.1. This may change in the future, as the DUO team adds support for more recent versions of JetPack. JetPack 3.0 can be run from a host computer running either Ubuntu 14.04 or 16.04. At some point during the installation process the TX1 needs to be connected to the host computer using a microUSB-USB cable, connected to the microUSB port on the TX1 and a free USB port on the host computer.

Installing Linux on the TX1 using JetPack 3.0 with L4T 24.2.1 A full installation guide is given in [31], which can be followed more or less to the letter. Some remarks, however:

- At some point during the process, which components to be installed on the TX1 needs to be chosen. Not all of the components are not needed, such as the VisionWorks and TensorRT.
- At step 10 in the installation guide, the network layout for our specific environment needs to be selected. Connect the TX1 with an Ethernet cable to the same network switch as the host PC is connected to, and choose the first option, "Device accesses Internet via router/switch". Then, in step 11, select "eth0" as the network interface.

The following sections describe installation of software done on the device.

5.3 Duo M stereo camera

Getting the Duo M stereo camera to run on the Nvidia Jetson TX1 is done by doing the following steps [6]:

1. Download the DUO SDK from the duo website
2. Extract the content to the preferred location, e. g. to the desktop
3. Connect the Duo M to the TX1 USB port
4. Open a terminal in the folder ~\Desktop\SDKrootfolder\DUODriver (if the SDK was extracted to the desktop) and do the following commands:

```
$ chmod u+x duodriver.run
$ ./duodriver.run
```

5. Restart the TX1
6. Load either the duo-1024.ko or the duo-512.ko version of the driver:

- duo-1024.ko: 1024 bytes transfer, for better performance (recommended)
- duo-512.ko: 512 bytes transfer, for better compatibility

```
$ sudo insmod duo-1024.ko
```

or

```
$ sudo insmod duo-512.ko
```

The Duo M stereo camera is now ready for use.

5.4 Run the TX1 from an external SD card

At this point, the internal memory on the TX1 is starting to get full, and more storage is needed to be able to install other necessary software such as OpenCV. It is, however, easy to make the TX1 run from an external SD card. A full guide can be found in [19], while the main steps are presented in A.1.

5.5 OpenCV

OpenCV is an open source library containing multiple programming functions for computer vision and image processing applications [37]. It supports Windows, Mac and Linux and is written in C/C++ with additional support for Python as well. Most of the steps in the visual odometry algorithm, described in chapter 2, are implemented as OpenCV functions. OpenCV functions are accessed in the code using the `cv::` namespace.

GPU acceleration A GPU accelerated version of OpenCV exists which is written using CUDA, "a parallel computing platform and programming model developed by NVIDIA for general computing on GPUs" [30]. The OpenCV CUDA/GPU module has been shown to greatly increase performance of computer vision and image processing applications (up to 30x for primitive image processing [38]), and is specially designed for platforms such as the Jetson TX1. Functions in the CUDA module are accessed using the `cv::cuda::` namespace.

The CUDA module is enabled during the configuration of OpenCV by including the flag `WITH_CUDA=ON` when installing OpenCV using CMake. See A.2 for more information about the installation of OpenCV. It is important to note that the TX1 should be running from a SD card larger than 16GB before trying to install OpenCV, as the internal storage will not be sufficient.

5.6 Configuring the UART

The nRF51422 BLE dongle is connected to one of the the UARTs on the TX1, located at the J17 GPIO header on the TX1, as described in section 4.2.3. The only configuration needed on the TX1 in order to use the UART in an application is done by calling the following command in a terminal:

```
$ sudo chmod a+rw "/dev/ttyTHS2"
```

which adds the permission for reading from and writing to the UART, located at "/dev/ttyTHS2", to all users.

5.7 nRF51422 BLE dongle

The nRF51422 BLE dongle needs the latest client application developed by [23] installed to be able to connect to a server dongle. How this is done is described in the Software chapter (chapter 5) in [18]. The dongle software that worked on the Raspberry Pi in [18] works for the TX1 as well.

5.8 Compiling and running the complete system

When every component is connected to the TX1 and all necessary software is installed, the code for the application developed during this project can finally be downloaded, compiled and run. CMake, which is needed to compile the program, should already be installed during the OpenCV installation. How to compile and run the complete system application is given in A.3. The code is included as an attachment in B.2.

Chapter 6

Implementation

6.1 Stereo visual odometry algorithm

6.1.1 Introduction

The stereo visual odometry algorithm implemented on the Nvidia Jetson TX1 developer kit during this project is based on an approach given in [43], as mentioned in section 2.1.1. Section 2.1.3 describes two methods for finding feature points and their correspondences, feature tracking and feature matching, where the first approach is implemented in this project (future implementation of the second approach is discussed in section 10.1.2). Some additional steps are also included in the implementation, such as various filters removing noisy values, as well as an inlier detection step, which is an implementation of a method proposed in [16]. Details about the theory behind the different steps in the algorithm can be found in chapter 2, and how each step is implemented in practice is described in the following sections. The algorithm is implemented using C++, OpenCV and the DUO SDK. There are in general a lot of parameters associated with each step in the algorithm, which can be tuned in order to obtain potentially better performance. Some of the chosen parameter values will be presented, but not much time has been used to tune the parameters in order to try to achieve better performance. Choosing an optimal set of parameters is something that should be looked at in the future, which is discussed in section 10.1.7.

Many parts of the visual odometry algorithm are already implemented in functions included in the OpenCV library, which have been used in this project. A lot of OpenCV classes and variables are also used throughout the application. Not all of them will be explained in detail. For further details, please refer to the OpenCV documentation [39].

6.1.2 Capturing images

Initializing the DUO M stereo camera The DUO SDK provides an API which includes functions that gives access to all the necessary features of the DUO M stereo camera, such as initializing the camera, capturing images, and it even includes an API for depth reconstruction, called DUO Dense3D. The DUO M camera has usually been initialized with the following parameters:

- **Resolution:** 320x240
- **FPS:** 30
- **Gain:** 0
- **Exposure:** 85

The Dense3D API is later used for computing a disparity map of the captured frames, and it needs to be initialized as well. A lot of parameters are used to configure the API, and tweaking the parameters will result in different performance of the visual odometry algorithm, both regarding the accuracy of the pose estimates and the run-time of each iteration. Details about the Dense3D parameters can be found in [7].

Camera calibration The DUO M is factory calibrated, and the calibration parameters can be obtained using the function `GetDUOStereoParameters()` from the DUO SDK. See the DUO SDK [5] documentation for more details regarding the stereo parameters returned by the function.

Capturing and storing images At every iteration, the application captures a frame I_k, where each frame consists of two images, one for each of the lenses of the camera (except for the very first iteration, where two frames are captured, I_{k-2} and I_{k-1}). The images are stored in `cv::Mat()` variables, which is an OpenCV variable suitable for storing images.

In the following steps, only the images from the left lens of the camera are used (except when computing the disparity map in section 6.1.5, where stereo-pair images are needed).

6.1.3 Feature detection

The FAST feature detector is used for detecting features in this project. The theory behind FAST is described in section 2.1.2. OpenCV has implemented the FAST detector both in its regular library and in the GPU accelerated module (discussed in section 5.5). Both versions have been implemented in this project. The CUDA version has mostly been used, since it has shown to be faster than the CPU version (as will be discussed in chapter 8). The code for the function using the CUDA version of the FAST detector is given below, where the FAST threshold parameter is set to 20.

```
void featureDetection(cv::cuda::GpuMat gpu_img, vector<cv::Point2f>&
    fast_points){
  vector<cv::KeyPoint> d_keypoints; //vector to store detected features
  int fast_threshold = 20; //FAST threshold
  bool nonmaxSuppression = true;

  cv::Ptr<cv::cuda::FastFeatureDetector> d_fast = cv::cuda::
    FastFeatureDetector::create(fast_threshold, nonmaxSuppression,
    FastFeatureDetector::TYPE_9_16); //the FastFeatureDetector (pointer)
    object
  d_fast->detect(gpu_img, d_keypoints); //the actual detection

  cv::KeyPoint::convert(d_keypoints, fast_points, vector<int>()); //
    convert feature point vector to a vector of int's
}
```

The detected features are stored in a vector of `cv::Point2f` variables, called fast_points in the example above. `cv::Point2f` is an OpenCV variable suitable for storing 2D points as floating point numbers. The `cv::cuda::GpuMat` class is a `cv::Mat` variable uploaded to the GPU, which can be done using the following lines of code:

```
    cv::Mat img_cpu = cv::Mat(Size(WITDH, HEIGHT), CV_8C1);
    cv::cuda::GpuMat img_gpu; //GPU image variable
    img_gpu.upload(img_cpu); //upload the cv::Mat to the GPU
    // Process the image using the GPU
```

After processing the image using the GPU, it can be downloaded back to the CPU, e. g. for visualization:

```
    cv::Mat img_dst_cpu; //CPU image
    img_gpu.download(img_dst_cpu); //download image from GPU to CPU
    cv::imshow("Processed image", img_dst_cpu); //show the image in a
    figure
```

6.1.4 Feature tracking

After detecting features in the left image at frame I_{k-1}, the features need to be tracked to the next frame, I_k. This is done using a version of the Lucas-Kanade optical flow, which is implemented in the OpenCV function `cv::calcOpticalFlowPyrLK()`. The basic theory behind Lucas-Kanade optical flow is given in section 2.1.3. `cv::calcOpticalFlowPyrLK()` is an implementation of a sparse iterative version of the Lucas-Kanade optical flow in pyramids, an implementation proposed in [4]. A CUDA version of the function exists as well, but this has not been used in this project.

The following example shows how to track feature points found in an image `cv::Mat img_1`, stored in a vector `cv::Point2f points1`, to the next image, `cv::Mat img_2`, where the traced points in `img_2` are stored in a new vector `cv::Point2f points2`:

```
void featureTracking(cv::Mat img_1, cv::Mat img_2, vector<cv::Point2f>&
    points1, vector<cv::Point2f>& points2, vector<uchar>& status){
  vector<float> err;
  cv::Size winSize=cv::Size(21,21);

  cv::TermCriteria termcrit=cv::TermCriteria(cv::TermCriteria::COUNT+cv::
    TermCriteria::EPS, 30, 0.01);

  cv::calcOpticalFlowPyrLK(img_1, img_2, points1, points2, status, err,
    winSize, 3, termcrit, 0, 0.001);
}
```

Please refer to the OpenCV documentation for details regarding the parameters used in the feature tracking, such as the `cv::Size winSize` and `cv::TermCriteria termcrit`.

The tracking may fail for those points detected in the first image which has gone outside the frame in the second image if the camera moves between the two frames (which is a likely scenario). This is dealt with in the following lines of code, where the feature points that has gone outside the frame are removed. `status` is the same `vector<uchar>` variable used in `cv::calcOpticalFlowPyrLK()` above, which indicates whether the tracking for a specific feature point has been successful or not:

```
int indexCorrection = 0;
for (int i=0;i<status.size();i++){
  cv::Point2f pt = points2.at(i-indexCorrection);
  if( (status.at(i)==0) ||(pt.x<0) || (pt.y<0) ){
    if((pt.x<0)||(pt.y<0)){
      status.at(i) = 0;
    }
    points1.erase(points1.begin() + (i - indexCorrection) );
    points2.erase(points2.begin() + (i - indexCorrection) );
    indexCorrection++;
  }
}
```

6.1.5 Triangulation

Disparity map To be able to compute the 3D location of a feature point in the image plane, the disparity of the point is needed. Disparity is described in section 2.1.5. Included in the Dense3D module of the DUO SDK is the function `Dense3DGetDepth()` which computes a disparity map from a set of stereo-pair images. The function returns a `cv::Mat` variable of the same size as the input images containing the disparity value for every image point, calculated with respect to the left input image. In other words, the value at (i, j) in the returned `cv::Mat` variable corresponds to the disparity value for the point at (i, j) in the left image of the stereo-pair. The computation of the disparity map is based on a block matching algorithm described in section 2.1.6. The Dense3D module is able to compute the disparity map using four versions of the block matching algorithm:

- Simple Block Matching
- SGBM
- Simple Block Matching with post processing
- SGBM with post processing

Which algorithm to use is set during the initialization of the Dense3D module, and is a compromise between accuracy and speed. This will be further discussed in chapter 7.

Disparity filter The disparity map computed by the DUO SDK contains noise and may also return invalid disparity values, which is a problem for all the versions of the block matching algorithm used for computing the disparity map. One of the reasons for this is the working range of the stereo camera. As mentioned in section 3.3, the DUO team recommends using the DUO M within 0.23-2.5m when using the Dense3D module. The further away the objects in the scene are from the camera, the more inaccurate the disparity values become for those objects. This, together with the appearance of general image noise, makes it necessary to have a disparity filter for filtering out invalid or noisy values after the disparity map has been computed. The disparity values returned by the DUO SDK function are given in mm, and a filter is implemented currently removing all points with disparity outside the range $\pm[0.1, 10000]$ or with a disparity value equal to `inv` (invalid). Not much time has been used to check if the range values are optimal.

Projecting 2D to 3D Once the disparity map is computed, the 3D localization of the 2D feature points can be computed. The theory behind how to compute the 3D localization of a 2D point using the disparity value is described in section 2.1.5. In the current implementation of the visual odometry algorithm, this is done by first using an OpenCV function `cv::reprojectImageTo3D()`, which computes a 3D image containing the 3D coordinates of every pixel in the 2D image, given in mm. The 3D image is stored in a so-called "three-channel `cv::Mat`" variable, which is essentially a $M \times N \times 3$ matrix, where $M \times N$ is the size of the 2D image and where each pixel/element in the matrix has three values which corresponds to the 3D localization of that pixel/element. When the 3D image is computed, the 3D points corresponding to the 2D feature points are extracted from the 3D image and stored in a new vector of `cv::Point3f` variables. `cv::Point3f` is an OpenCV variable suitable for storing 3D points as floating point numbers.

Invalid 3D points filter The computation of the 3D image using `cv::reprojectImageTo3D()` may result in noisy or invalid 3D points, and a filter filtering out such points are implemented after the 3D feature points are extracted from the 3D image. The filter currently examines the Z value of each 3D point, and removes all feature points with a Z value outside the range $\pm[0, 10000]$ (given in mm), or with Z equal to `inv` (invalid) or `inf` (infinity). As in the disparity filter, not much time has been used to check if the valid range is optimal.

6.1.6 Inlier detection (max-clique approximation)

The inlier detection step, as mentioned in section 6.1.1, is an implementation of a method proposed in [16]. The theory behind the method is described in section 2.1.7. The actual implementation is developed by Lee Schloesser, and the code [45] has been provided by him personally. It worked out of the box in my application, detecting and keeping only the feature points that are regarded as consistent.

6.1.7 Motion estimation

The local motion estimate between the previous and the current frame, I_{k-1} and I_k, respectively, is calculated by solving the PnP problem, as discussed in section 2.1.8. The OpenCV function `cv::solvePnP()` solves the PnP problem and returns a translation and a rotation vector corresponding to the camera movement between the two frames. The rotation vector is a 3×1 vector, which can be converted to the corresponding 3×3 rotation matrix using the OpenCV function `cv::Rodrigues()`. `cv::solvePnP()` takes the 3D feature points at frame I_{k-1} and the corresponding 2D feature points at frame I_k as inputs, as well as a camera distortion matrix (which should be equal to zero, since the feature points are computed using rectified and undistorted stereo-pair images) and a flag for setting the method for solving the PnP problem. Multiple methods are available, but the one that has been used is a method based on Levenberg-Marquardt optimization. The OpenCV implementation basically solves the equation 2.18 using the Levenberg-Marquardt algorithm, which is used to solve non-linear least squares problems [22], [26], [21]. The method is set using the flag `SOLVEPNP_ITERATIVE`.

Motion estimate noise filter The last step of the algorithm before the global pose of the camera is calculated is a motion estimate noise filter. The filter simply checks how far the camera has moved between the two latest frames in the x, y and z direction according to the current result from `cv::solvePnP()`. In the current implementation, if the movement is calculated to be more than 30 mm in one or more of the directions (which would equal a velocity of 15 cm/s if the algorithm is running at 5 Hz), the movement is discarded and the current iteration is terminated, starting the algorithm again from the beginning. Such a filter is necessary because `cv::solvePnP()` suffers from catastrophic errors from time to time, resulting in useless pose estimates if not dealt with. However, since the current filter just discards the current estimate when such an error is discovered, the system believes that it stands still, which most likely is not the case. Suggestions for better solutions are discussed in section 10.1.5.

6.1.8 Concatenation of transformations to obtain global pose

The translation and rotation vector calculated by `cv::solvePnP()` represent the local transformation from the pose of the camera at frame $k-1$ to the camera pose at frame k, given with respect to the coordinate system at frame k. As described in section 2.1.9, the transformation can be written as the following rotation matrix and translation vector:

$$R^k_{k-1}$$
$$t^k_{k,k-1}$$

Up to this point we have the transformation from the initial pose (which is assumed to be at $(x_0, y_0, z_0, \alpha_0, \beta_0, \gamma_0) = (0, 0, 0, 0, 0, 0)$ in the global coordinate frame) to the pose at frame I_{k-1}, given by the following rotation matrix and translation vector:

$$R^0_{k-1}$$
$$t^0_{0,k-1}$$

We want the final transformation, given in global coordinates, which can be written as R^0_k and $t^0_{0,k}$. The mathematical calculations to obtain the final transformation is described in section 2.1.9. The calculations are implemented in the following C++ function, where `Mat rvec_pnp` and `Mat tvec_pnp` are the rotation and translation vectors calculated by `cv::solvePnP()`. `cv::Rodrigues()` is used to transform a rotation vector to a rotation matrix (both ways), and `cv::composeRT()` is used for concatenating the previous global transformation with the new, local transformation:

```cpp
void calculate_global_pose(cv::Mat rvec_pnp, cv::Mat tvec_pnp, cv::Mat&
    R_matrix_global, cv::Mat& t_matrix_global)
{
    //Related to the master's thesis:
    //[rvec1, tvec1] => T^{k-1}_0, previous pose
    //[rvec2, tvec2] => T^k_{k-1}, local transformation/pose update
    //[rvec3, tvec3] => T^k_0, new (/current) pose update
    cv::Mat rvec1, tvec1, rvec2, tvec2, rvec3, tvec3; //inputs and outputs
     for cv::composeRT()
    cv::Mat R_glob_inv;

    //Previous pose
    R_glob_inv = R_matrix_global.t();
    cv::Rodrigues(R_glob_inv, rvec1);
    tvec1 = -R_glob_inv*t_matrix_global;

    //New, local pose update
    rvec2 = rvec_pnp;
    tvec2 = tvec_pnp;

    //New, current/final pose update
    cv::composeRT(rvec1, tvec1, rvec2, tvec2, rvec3, tvec3);

    //update global pose
    cv::Rodrigues(rvec3, R_matrix_global);
    R_matrix_global = R_matrix_global.t(); //R^0_k
    t_matrix_global = -R_matrix_global*tvec3; //t^0_{0,k}
}
```

Now, `cv::Mat R_matrix_global` and `cv::Mat t_matrix_global` equals R_k^0 and $t_{0,k}^0$.

6.2 Communication and interaction with the server

The Ubuntu version of the communication protocol developed in [18], discussed in section 2.2, was almost perfectly compatible with the TX1. The only change that was needed was to change the address of the UART port, which on the TX1 is `"/dev/ttyTHS2"`, as opposed to `"/dev/ttyS0"` on the Raspberry Pi in [18]. The communication module has been further developed such that the TX1 is able to successfully maintain a more or less stable connection with the server.

The complete system has also been further developed such that the TX1 is able to continually send update messages to the server application, containing the pose estimates that the visual odometry algorithm computes. The update message also contains two coordinates, (x_{start}, y_{start}) and (x_{end}, y_{end}), which represent the coordinates of where a line segment detected by the robot starts and ends. Currently, these coordinates are just set to some random number. The application that was developed on a Raspberry Pi in [3], which is able to detect line segments of a maze from images taken of the maze from above, has not been implemented on the TX1 in this project. This needs to be done, and is discussed in section 10.2.1.

Chapter 7

Feature points

The performance of the steps of the visual odometry algorithm regarding feature detection, filtering of invalid feature points, detection of inliers and the feature tracking as well as the computation of the disparity is discussed in this chapter. Results regarding the actual motion estimation is presented and discussed in chapter 8.

This part of the visual odometry algorithm is discussed while looking at the results of the mentioned steps during one iteration of the algorithm. The results were generated in a setting with good lighting conditions and many distinct features, to see how the feature detection, filtering and tracking performs when conditions are close to optimal. The test was performed using a camera resolution of 480x480, the disparity map were computed using the SGBM algorithm (see section 6.1.5) and the feature detection was done using the CUDA version of the FAST detector (see section 6.1.3). Figure 4.9 in section 4.4.1 shows the complete setup during the test.

The feature tracking has not been included in the discussion of the feature detection and filtering in section 7.1, but it is important to note that the feature tracking actually happens right after the feature detection in the current implementation of the algorithm, as described in section 6.1.4. During the testing of the feature detection, filtering and inlier detection, the camera was not moving between the consecutive images at all, and all the features were successfully tracked to the next image, i. e. no feature points were removed during the tracking step. A separate test has therefore been performed where the TX1 was moved between the capture of the two frames to better illustrate the tracking of features between two consecutive frames, and the feature tracking is discussed in section 7.3.

Another thing to note about the results below is that the filters actually are applied to both of the two consecutively captured images. Since the camera was standing still between the capture of the two images in this example, and that every feature in the first image was successfully tracked to the next, as described above, the two images are in practice equal in this specific case. Therefore, only the first captured image are presented below. However, it is important to be aware of that in a setting where the camera moves between the captured frames, the filters may filter out invalid points found in either of the two consecutive images. If a feature is removed in one of the images, the corresponding point in the other image are removed as well, such that the number of features are always the same for both of the images.

7.1 Feature detection and filtering

FAST feature detection The FAST feature detector successfully detects a lot of feature points, as can be seen in figure 7.1. FAST detects corners and this is evident by studying the figure, especially when looking at the paper sheet hanging on the wall with four black square patches on it, where all the corners have been detected and marked as feature points. A lot of the detected points may not look like traditional "corners", but the FAST detector marks pixels as feature points that have the same intensity properties as a corner in a small neighbourhood around the potential feature point pixel, which is hard to see just by looking at the image. In other words, a pixel may have "corner properties" at pixel level even though it doesn't look like a traditional corner, which is the case for many of the detected features in this example. For detailed information about how the FAST detector marks a pixel as a feature point, see section 2.1.2. The FAST detector detected 390 points in this test (as marked on the image), which should be more than enough for the purpose of this application.

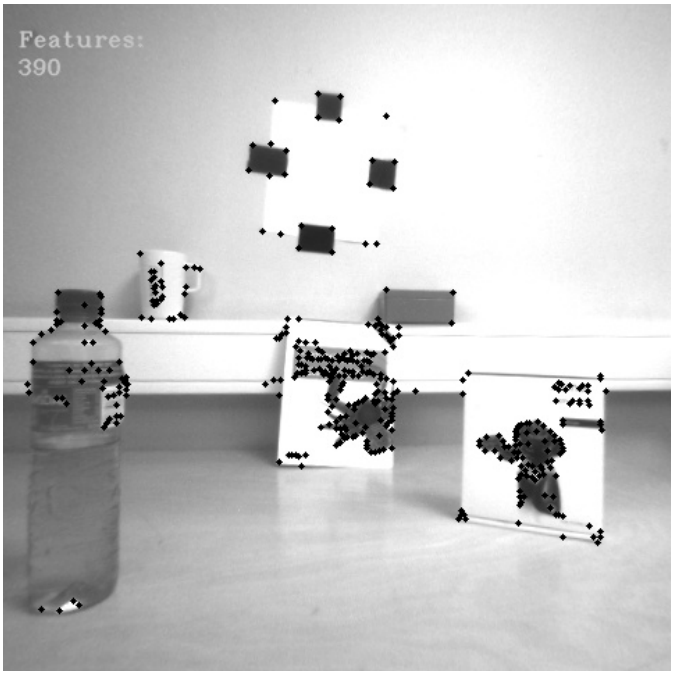


Figure 7.1: Detected feature points in the left image, before any filtering

In settings less optimal than the one in the test setup, the feature detection is likely to struggle more. The visual odometry algorithm is very dependant on finding good features to track, and it is important that the feature detection is as robust as possible. An improvement to the feature detection, which involves distributing the detected features more evenly across the whole image, is proposed in section 10.1.1.

Disparity filter The disparity filter removes feature points detected by the FAST detector that have invalid disparity values, as described in section 6.1.5. During this test, the disparity map, i. e. the image containing the disparity values of every corresponding pixel in the captured image, was computed using the SGBM algorithm (see section 2.1.6). Figure 7.2 shows the disparity map computed from the scene during the feature detection test together with the two images captured by the left and right lens of the stereo camera (where the left image is used in the feature detection). The computation of the disparity map is further discussed in section 7.4. However, taking a brief look at figure 7.2, where black spots represent invalid disparity values, all of the areas where feature points have been detected apparently have valid disparity value (i. e. no black spots), which was also the case during this test. No points were therefore removed by the disparity filter, as can be seen in figure 7.3, where all the 390 detected features remain after the disparity filter step.

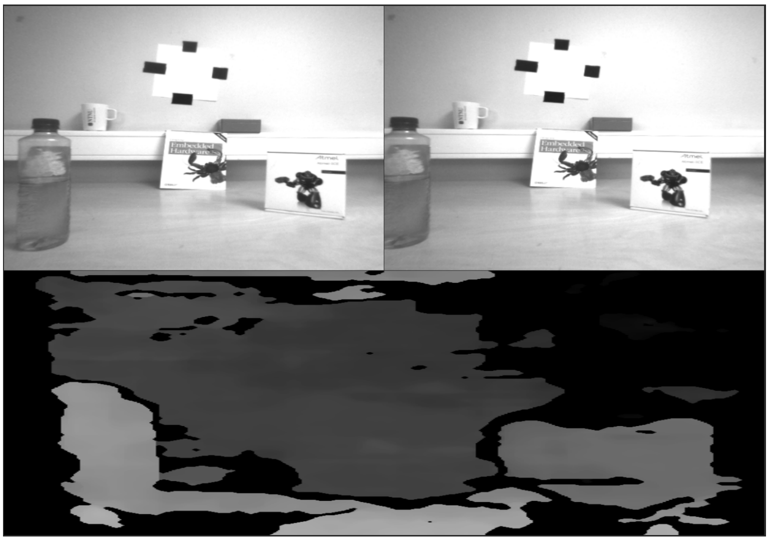


Figure 7.2: Disparity map computed of the scene during this test, using the SGBM algorithm to compute the disparity

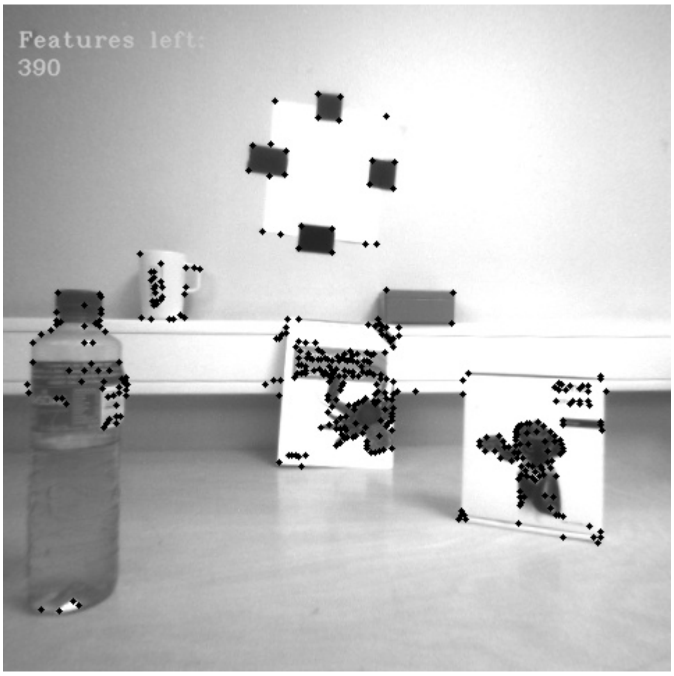


Figure 7.3: Every feature had valid disparity, and no feature points were therefore removed by the disparity filter.

Invalid 3D points filter The remaining features after triangulating the feature points and filtering out the points with invalid 3D coordinates, as described in section 6.1.5, are shown in figure 7.4. 20 features were removed in this step, and comparing figure 7.4 to figure 7.3, many of the features removed were located at the left side of the bottle in the image. Looking at figure 7.2, the left side of the bottle are outside the frame captured by the right stereo camera lens, and should therefore have had invalid disparity values. Somehow these points were not removed by the disparity filter, and may be the result of a badly tuned disparity filter or an inaccurate disparity map. Either way, these points are correctly removed by the 3D filter.

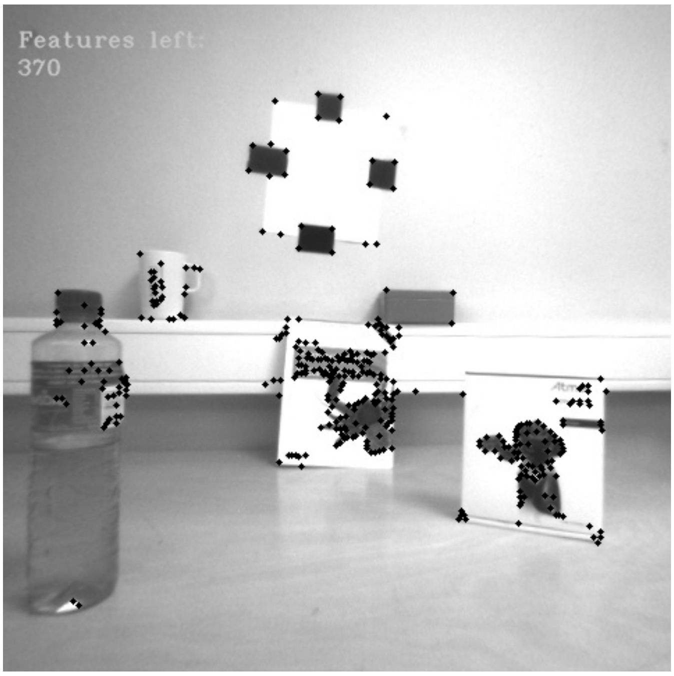


Figure 7.4: Remaining features after the invalid 3D points filter

7.2 Inlier detection

The remaining feature points after finding and keeping the points regarded as inliers in the max-clique approximation step of the visual odometry algorithm are shown in figure 7.5. Details about this inlier detection step is given in section 6.1.6. 260 of the 370 features were found to be consistent, and the remaining points after this step are the points used in the motion estimation.

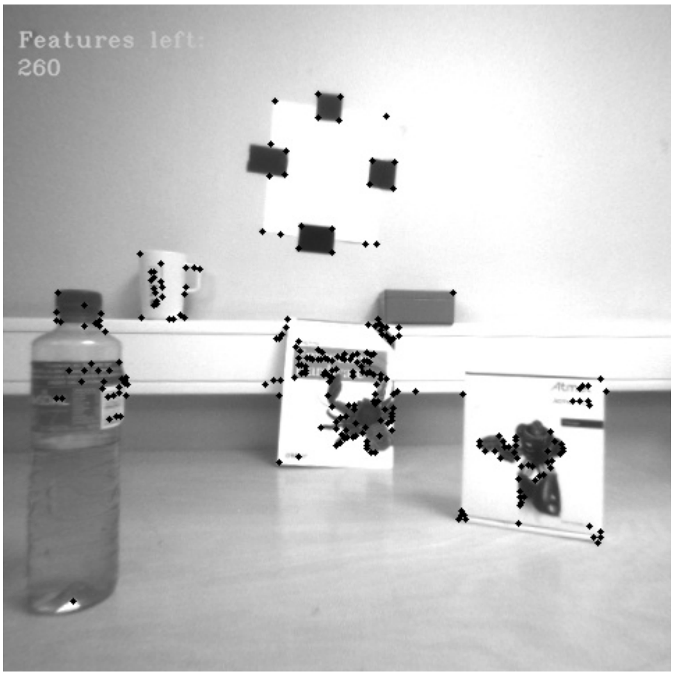


Figure 7.5: Remaining features after detecting inliers using max-clique approximation

This means that after the all the filtering steps during this iteration of the algorithm, 260 corresponding points of the original 390 detected points are regarded as good enough and are used in the last step of the algorithm that uses the point correspondences for solving the PnP problem and computing the relative pose between the two consecutively captured images. As discussed in section 2.1.8, a minimum of four 3D-to-2D correspondences are needed to solve the PnP problem which means that in this case, the number of remaining point correspondences are more than good enough to estimate the pose.

7.3 Feature tracking

A separate test has been performed to better illustrate the feature tracking step, as mentioned at the beginning of this section. Figure 7.6 shows two images captured consecutively by the stereo camera. Both images were captured by the left lens of the stereo camera, and the left image is the first image captured and the right image is captured right after the first. The camera was moved briefly to the right between the capture of the images, which can be seen especially by looking at the bottle, which has moved slightly to the left in the right image. The feature points plotted in the left image are a selection of the features detected by the FAST detector which were successfully tracked to the next image (plotting all the features would be very confusing). The feature points plotted in the right image are the corresponding points tracked in the second image. A line is drawn between every feature point that are assumed to be corresponding. Following the lines from each point in the left image to each point in the right image, apparently all the correspondences are "true". For instance, looking at the three feature points detected on the paper sheet hanging on the wall in the first image and following the lines to the next image, the points are tracked to the correct location in the second image, and this is the case for all the other features as well.

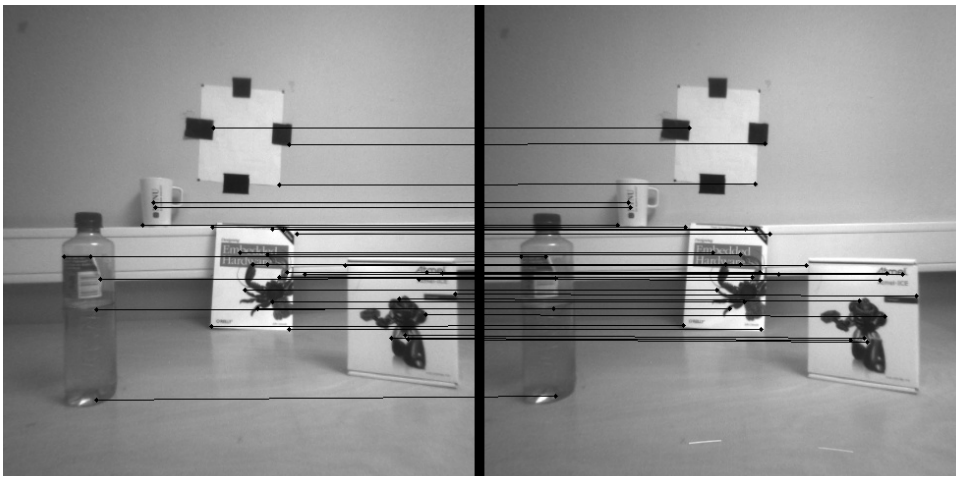


Figure 7.6: Successful tracking of features between two consecutive frames

In figure 7.6, the camera was only moved slightly to the right, and all the features were successfully tracked. Another example is shown in figure 7.7, where the camera was moved a lot more between the capture of the two frames, and where the feature tracker is seemingly unable to track a single point successfully. In a case such as this, all the points would most likely be considered as inconsistent in the inlier detection step and they would all be discarded, leaving no points left for the final motion estimation step. This is a typical result for feature tracking, which works best for small movement. In the scope of this project, the movement between consecutive frames is assumed to be small, however, this is a limitation and a potential large source of error, and an alternative approach is discussed in section 10.1.2.

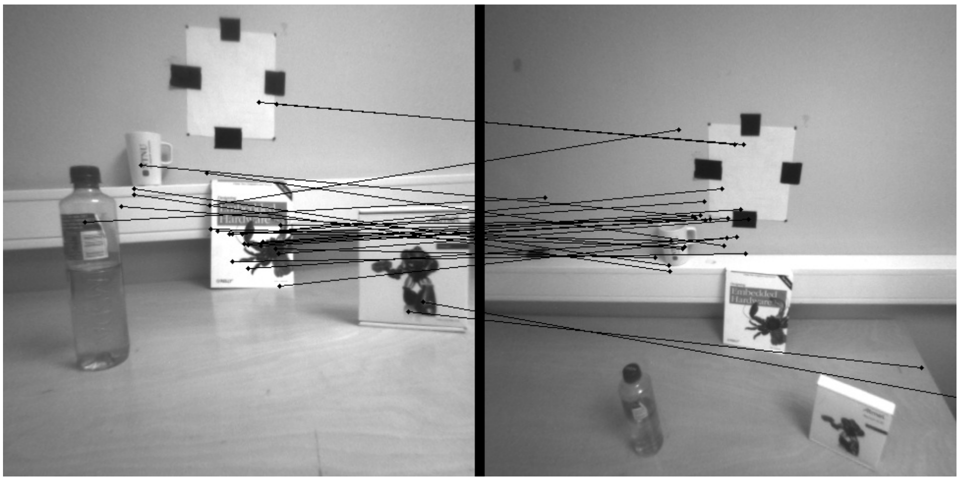


Figure 7.7: Unsuccessful tracking of features between two consecutive frames with large camera movement between the captured frames

7.4 Disparity map calculation

As mentioned in section 6.1.5, the disparity maps used in this project are computed using the Dense3D module of the DUO SDK, which includes implementations of multiple algorithms for computing disparity maps, including the SGBM algorithm used during the test presented in section 7.1.

Another algorithm for computing the disparity map, presented in section 2.1.6, is the simple block matching algorithm. The computation of disparity maps was seen to be faster when using simple block matching as compared to the SGBM algorithm. However, the remaining points after the disparity filter when using simple block matching were generally a lot less than when using SGBM, and in addition, the motion estimates produced when using simple block matching were less accurate than when using SGBM. This can be explained by looking at figure 7.8, which shows a disparity map computed of the same scene as the disparity map in figure 7.2. The disparity map is a lot more noisy and contains more black spots (i. e. invalid disparity) than the disparity map computed using SGBM, seen in figure 7.2, which is likely to result in many feature points with invalid disparity values. Inaccurate disparity values will affect the triangulation of the feature points, resulting in inaccurate 3D coordinates, which in turn affects the motion estimation. Therefore, the SGBM algorithm has been seen as a good compromise between performance and accuracy and is therefore the algorithm that has mostly been used in this project.

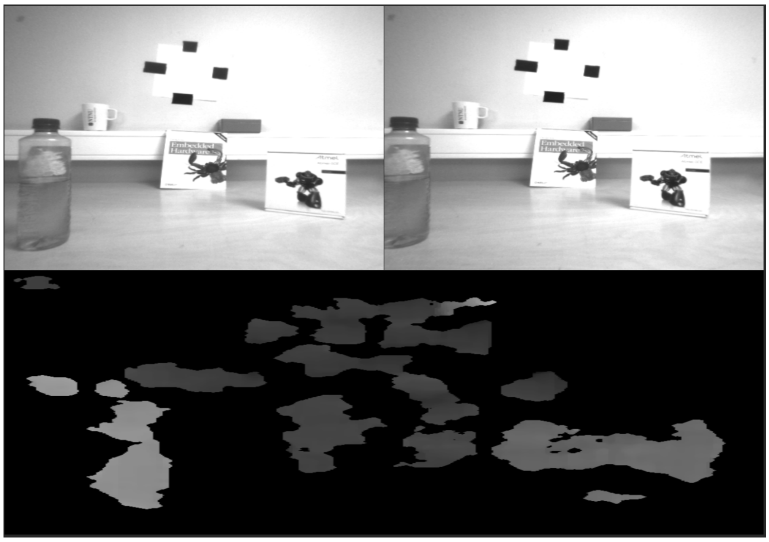


Figure 7.8: Disparity map computed from a stereo-image pair using simple block matching to compute the disparity

Chapter 8

Motion estimation

The results regarding the motion estimates calculated by the visual odometry algorithm are presented in this chapter. The rest of the algorithm, i. e. the feature detection, filtering, inlier detection, feature tracking and disparity map computation is discussed in chapter 7.

The motion estimation tests, generating the results presented below, was performed using the same application settings as when the results regarding the feature points were generated except for the camera resolution, which was set to 320x240 to obtain a higher frequency on the pose estimation.

While the results are quite inaccurate in general, the orientation estimates were especially noisy. Therefore, only the position estimates are presented and discussed in the sections below.

8.1 Results

Two simple tests has been performed to see how well the complete visual odometry system performs, which is presented in following sections. Both of the tests were done in the "Bevegelseslab" at NTNU, using the OptiTrack system implemented in the lab to track the actual pose of the TX1, as described in section 4.4.2. The TX1 was placed on a chair with wheels and in the first test, the chair was pushed slowly back and forth approximately in a straight line in the x-direction relative to the camera coordinate system. In the second test, the chair was moved in a square in the camera x-y plane. In both tests, it was made sure that the light conditions were good and that the scene contained distinct objects such that the feature detection would work well. The stereo camera was also pointing outwards in both test, just like in figure 4.7. For information about the coordinate system of the camera, see section 4.2.2. The coordinate system of the OptiTrack system was aligned with the coordinate frame of the camera, to make it possible to compare the pose estimated by the visual odometry algorithm with the OptiTrack pose directly. The results are presented below, and will be discussed further in section 8.2. It is important to notice that the axes on some of the plots have not the same range. The dimension on all the axes are mm.

8.1.1 Straight line test

Figure 8.1 shows the pose estimated by the TX1 together with the actual pose estimated by the OptiTrack system, plotted in the camera x-y plane, when moving the TX1 in a nearly straight line. Figure 8.6 shows the same plot, only zoomed in. Looking at the OptiTrack plot (ground truth) in figure 8.6, the TX1 starts in $(x, y) = (0, 0)$, moves approximately in a straight line 520mm to the left, then returns and ends up back at $(x, y) = (0, 0)$ (the few red marks just right of $(0, 0)$ is just noise, and the TX1 actually starts and ends up in $(0, 0)$). Looking at the blue plot in the same figure (pose estimated by the visual odometry algorithm), it starts in $(x, y) = (0, 0)$, moves approximately 380mm to the left before it returns and ends up approximately 80mm too far to the right. At the same time, the pose drifts some 40mm in the positive y-direction while the TX1 is moving to the left, but drifts the same distance back again when the TX1 moves to the right, and ends up correctly at approximately $y = 0$.

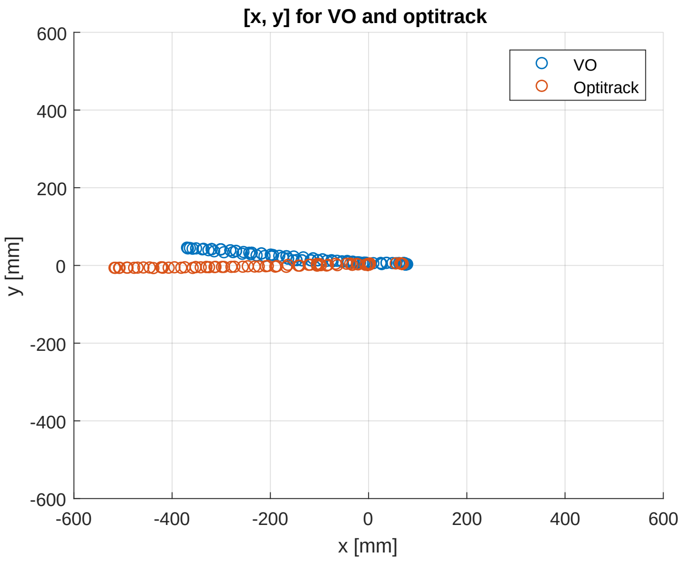


Figure 8.1: Moving the TX1 in a short line along the x axis, plotted in the x-y plane

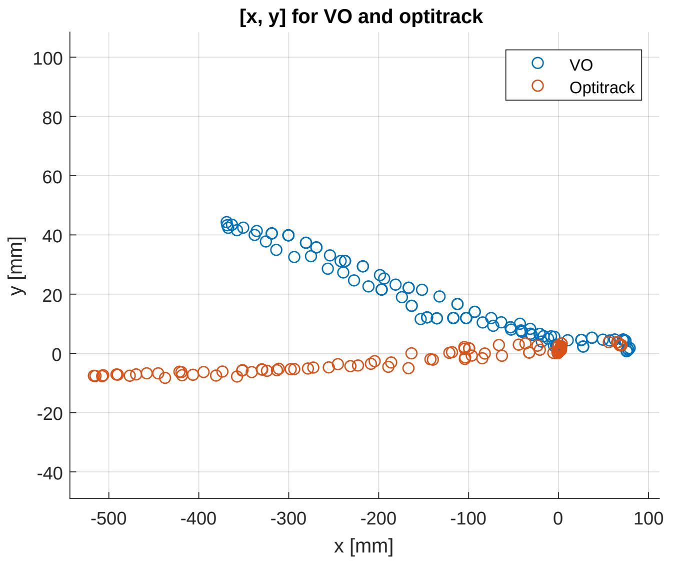


Figure 8.2: Figure 8.1 zoomed in. Notice the different range on the axes

Figure 8.3 shows the pose from the same test, now plotted in the camera x-z plane.

Figure 8.4 shows a zoomed in version of the same plot as in figure 8.3. Looking at the OptiTrack plot, the TX1 moves somewhat in the z-direction, but starts and ends up at approximately $(x, z) = (0, 0)$ (the red marks just right of $(0, 0)$ is just noise, as in the x-y plot). The visual odometry pose, however, drifts quite a lot in the z-direction, and ends up at approximately $(x, z) = (80, 30)$.

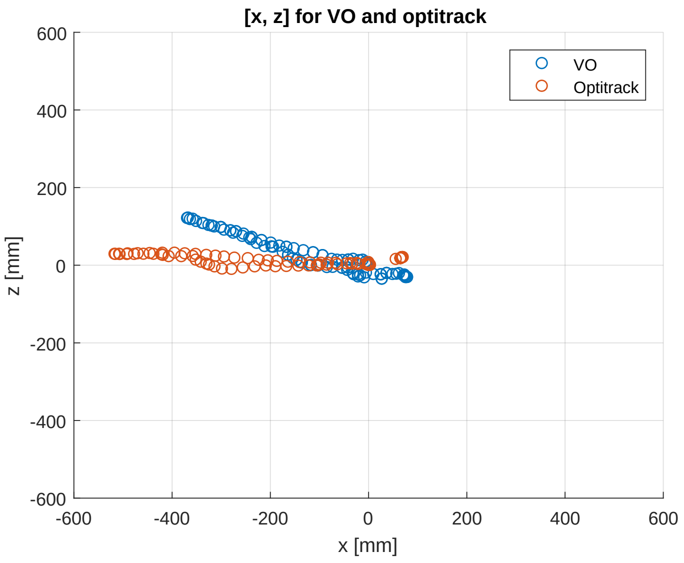


Figure 8.3: Moving the TX1 in a short line along the x axis, plotted in the x-z plane

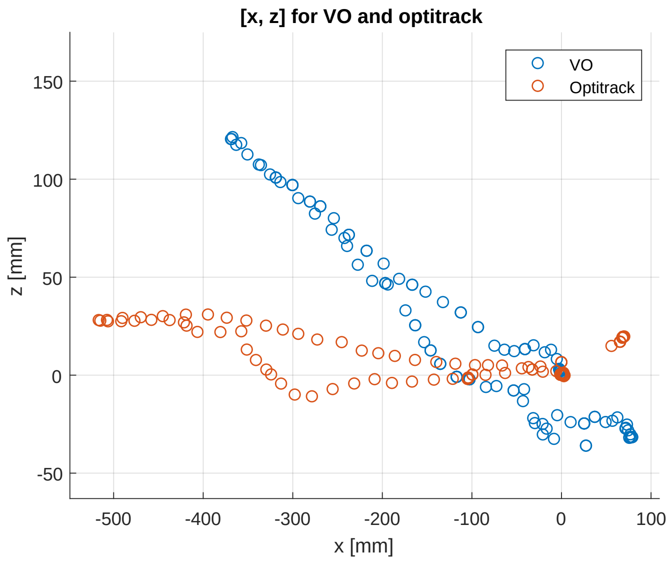


Figure 8.4: Figure 8.3 zoomed in. Notice the different range on the axes

8.1.2 Square test

Figure 8.5 shows the OptiTrack and the visual odometry pose plotted together when moving the TX1 in a square in the camera x-y plane. Figure 8.6 shows a zoomed in version of figure 8.5. Looking at the OptiTrack pose, the TX1 starts at $(x, y) = (0, 0)$, moves counterclockwise in a nearly straight square and ends up at approximately $(x, y) = (-100, 30)$. The pose estimated by the visual odometry algorithm resembles a square, but it is quite inaccurate, and it believes that the TX1 moves in a shorter distance in every direction than it actually does. For instance, when the TX1 at first moves 100mm in the negative y-direction, the visual odometry pose stops at approximately $y = -80$. And right after that, when actually moving approximately 590mm to the left, the visual odometry pose stops just after roughly 400mm. For the last movement straight to the right, however, the OptiTrack plot shows that TX1 moves nearly 500mm in the positive x-direction. Now, the visual odometry pose also estimates that the TX1 moves almost 500mm in the x-direction, from $x = -440$ to $x = 130$. Since it believes that it starts at $x = -440$, and not $x = -590$, however, it ends up roughly 130mm too far to the right. At the same time, it drifts quite a lot in the negative y-direction.

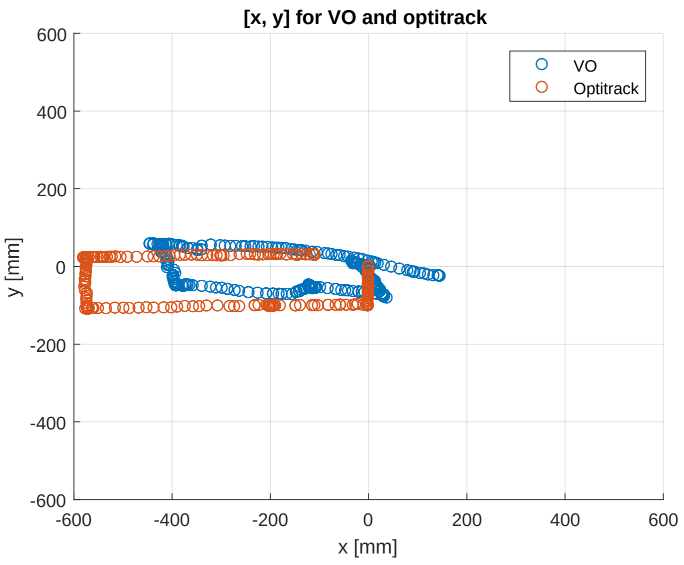


Figure 8.5: Moving the TX1 in a square in the x-z plane, plotted in the x-z plane

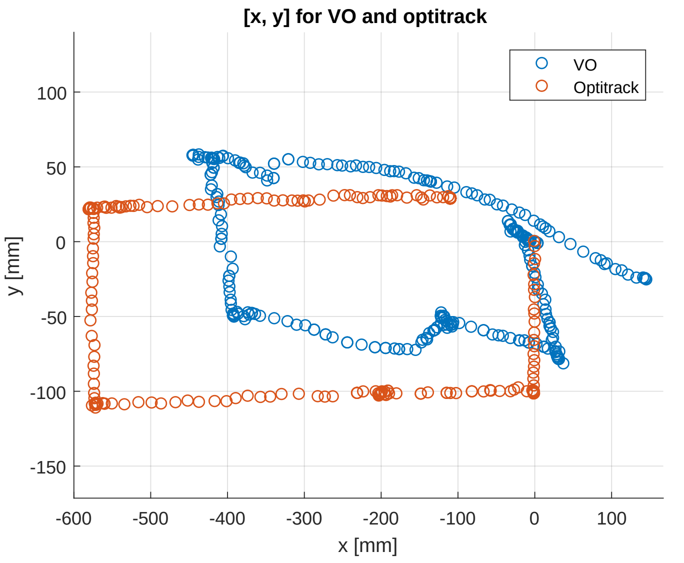


Figure 8.6: Figure 8.5 zoomed in. Notice the different range on the axes

8.2 Discussion

8.2.1 Motion estimation test

The pose estimation from the visual odometry algorithm is able to track the position of the TX1/stereo camera to some extent. As seen in figure 8.1, when moving the TX1 only in the x-direction, the x-and y-coordinate of the TX1 is not that bad. The y-coordinate drifts only a little and the x-coordinate seems to be able to follow the actual x-value, but seem to "stop" a little to soon. The z-coordinate drifts quite a lot, as can be seen in figure 8.3. However, the final position from the visual odometry pose is fairly accurate, both for x, y and z.

The pose estimates produced when moving the TX1 in a small square in the x-y plane can be seen in figure 8.5. The visual odometry pose plot resembles the correct square produced by the OptiTrack system, however somewhat deformed. As in the test when moving the TX1 only in the x-direction, the TX1 seems to "stop" a little too soon, at least for the first "three sides of the square" in figure 8.6 (where the TX1 was moved counterclockwise). A likely reason for this is the noise filter implemented in the algorithm just after solving the PnP-problem, as described in section 6.1.7. After solving the PnP-problem at each iteration of the algorithm, and thus acquiring the local transformation between two consecutive frames, the noise filter discards the pose estimate if the movement is found to be above some threshold (usually set to 30mm in either direction during this project, a value found empirically. See section 6.1.7 for more details). In other words, in those cases, it believes that the estimate contains too much noise. And, since it simply discards the pose estimate and does nothing else to compensate for the discarded estimate, the system will believe that it hasn't moved at all. This explains why the visual odometry algorithm "believes" that it has moved a shorter distance than what is true in both of the tests performed. This is an obvious weakness that should be fixed, and an improvement is suggested in section 10.1.5.

8.2.2 Pose estimate update frequency

The frequency that the visual odometry algorithm computes pose estimates depends on many factors, such as the resolution of the images taken by the stereo camera, the method used for the computation of the disparity map, if the feature detector use the CPU og GPU/CUDA version of the FAST detector and how many features the feature detector detects, to mention some. In tests performed similar to the one presented in this chapter, the frequency has been measured to be around 5-10 Hz. In [40], a multicopter was controlled using pose estimates from a visual odometry algorithm producing pose estimates at 20

Hz. How high the pose estimate update frequency needs to be in order to control a drone depends very much on the specific application, but in general, the higher the better. 5 Hz is likely too low, and a higher frequency should be strived for.

8.2.3 Rigid scene assumption

As mentioned in section 2.1.1, the visual odometry algorithm assumes that the scene is rigid between two consecutive frames. If the imagined drone system is supposed to aid in the mapping of a maze together with other robots, this assumption may occasionally be wrong, since the robots will move around in the maze and may be within the vision of the stereo camera. How much this will affect the pose estimation is hard to predict, however, the assumption is only that the scene is rigid between two consecutive frames. If the pose is estimated 5-10 times each second, the vehicles may not have moved a lot between two captured frames, which may reduce the problem. Regardless, it be a source of error in the estimated pose and is something that needs to be kept in mind in the further development of the system.

Chapter 9

Discussion

9.1 Visual odometry algorithm

As discussed in chapter 7, the current implementation of the visual odometry algorithm is able to detect a lot of features, successfully track features (provided that the motion is small enough), calculate a decent disparity map, filter out points regarded as invalid and only keep the point correspondences regarded as consistent. In most cases, at least in settings with good lighting conditions and many distinct features, enough points are remaining after these steps for the algorithm to be able to compute the change of pose between two consecutively captured frames.

However, as discussed in chapter 8, the global pose estimated by the algorithm contains a lot of noise, seemingly drift in every direction, and is in general too inaccurate for using it in any real application (such as in the control system of a drone). One possible reason for this may be the inlier detection step of the algorithm. The algorithm is able to detect a lot of features in the captured image, but if the 3D-to-2D correspondences used in the final motion estimation step are not correct, the pose estimates will be wrong. In other words, if the inlier detection detects false positives, the pose estimation will be strongly affected in a negative way. This is therefore a very important step in the algorithm, and the detection of false positive point correspondences may be the source of some of the inaccurate pose estimates observed in the current version of the algorithm. This is something that should be investigated, and an alternative to the inlier detection implemented in this project is also proposed in section 10.1.3.

A consequence of inaccurate motion estimates is that the estimates are discarded, making the system believe that it is not moving at all, as discussed in section 8.2.1. This is an obvious weakness in the system. Making the motion estimation more accurate will minimize this problem, and ideally, such a "noise filter" shouldn't be necessary at all. However, since the current algorithm, in addition to generally produce inaccurate estimates, also produces catastrophic errors from time to time (as discussed in section 6.1.7), such a filter is necessary. A suggestion to an alternative to the noise filter is proposed in section 10.1.5.

Also, if the TX1 moves too fast, the feature tracker is not able to successfully track points detected in the first image to the next, resulting in too few 3D-to-2D correspondences for the algorithm to be able to solve the PnP problem and compute a pose estimate. Feature matching, as an alternative to feature tracking and described in section 2.1.4, should be considered, as discussed in section 10.1.2.

9.2 Communication and interaction with the server

The TX1 is able to connect to the server using a BLE dongle in exactly the same way as the Raspberry Pi did in [18]. It is also able to send the position estimates that the visual odometry algorithm calculates to the server in (near) real-time, such that the TX1 appears in the map that the server application creates. These position update messages also contain coordinates of wall segments that the may TX1 detect, however, mapping has not been implemented in this project and the coordinate variables are currently just random numbers. The implementation of mapping on the TX1 is discussed in section 10.2.1.

A screenshot of the server application when the TX1 is connected to the server and sending position updates to it is shown in figure 9.1, where the TX1 is marked as "Drone". Another robot part of the LEGO-robot project was also connected to the server application at the same time as the TX1, marked as "ARD" in the figure below.

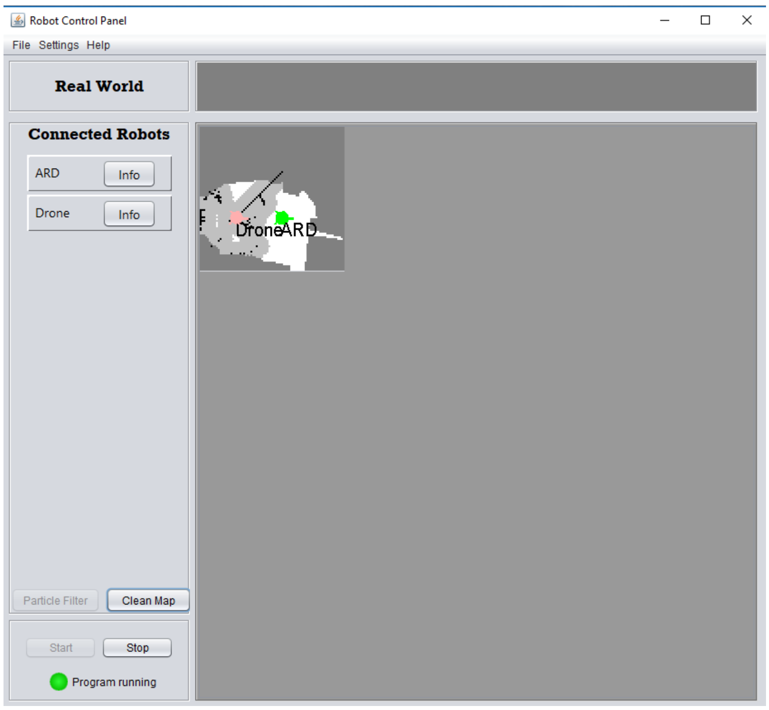


Figure 9.1: Screenshot of the server application with the TX1 ("Drone") and the Arduino based robot ("ARD") connected to the server

Chapter 10

Further work

10.1 Visual odometry algorithm

The most important work that needs to be done in the future regarding the continuation of this project is to improve the visual odometry algorithm such that it is able to produce stable and more accurate pose estimates. The following sections describe suggestions which could potentially improve the pose estimation.

10.1.1 Distribute detected features across the whole image

According to [44], the distribution of the detected features in the image majorly affects the results of a visual odometry algorithm. In general, more features provide better results than fewer features, but where the features are located is also very important, and the features should cover the whole image as evenly as possible. This can be done by dividing the image into a grid and applying the feature detector to each grid and at the same time make sure that a minimum of features are found in each subimage. Looking at an example from the FAST feature detection in figure 7.1, the features are not particularly distributed evenly in the image. This is something that has a lot of potential regarding improvement of the visual odometry, and an implementation of this should be prioritized in the future.

10.1.2 Feature matching instead of feature tracking

As discussed in section 2.1.4, two main approaches exist for finding corresponding features between two images, feature tracking and feature matching, where the first approach has been implemented in this project. As written in [44], feature tracking is a suitable approach when it is assumed that the scene is static and that the motion between adjacent frames is small. Feature matching, however, may work better for larger motion. Since the current implementation only works (to a degree) when moving the TX1 slowly, as discussed in section 9.1, feature matching could be a good alternative to feature tracking, and this should be investigated. The ORB feature detector is a good candidate for a suitable feature detector if feature matching is to be implemented, as discussed in section 2.1.4.

10.1.3 Outlier removal instead of inlier detection

As discussed in section 9.1, the inlier detection is a very important step of the visual odometry algorithm. Consistent point correspondences used in the final motion estimation step of the algorithm is necessary for accurate motion estimation. Therefore, it should be investigated if the inlier detection step is working as it should, since the current algorithm produces inaccurate pose estimates. Also, other methods for removing inconsistent points exists. A widely used approach for outlier removal is Random Sample Consensus (RANSAC) [14], which should be considered as an alternative to the inlier detection currently used. [44] presents details regarding the use of RANSAC for outlier removal in a visual odometry algorithm.

10.1.4 Local bundle adjustment

As emphasized in [43], a step that ensures more accurate pose estimates which always should be performed after every iteration in a visual odometry algorithm is called local bundle adjustment. As written in [43], "An iterative refinement over the last m poses can be performed (...) to obtain a more accurate estimate of the local trajectory. This iterative refinement works by minimizing the sum of the squared reprojection errors of the reconstructed 3D points over the last m images". This is an important step and is something that needs to be implemented.

10.1.5 Pose estimate noise filter alternative

As discussed in section 8.2, the pose estimates suffer from noise, and in the current implementation, if the estimates are considered to be too noisy, e. g. if the movement is more than some threshold in either direction, the current pose estimate is discarded, leaving the system to believe that it is standing still. A simple solution is to calculate an estimate of the current velocity of the system, for instance an average based over the n last pose estimates, and calculate a new pose according to the velocity estimate and the current update frequency of the algorithm. This could work if the system is moving slowly, and in the current version of the algorithm, slow movement is necessary for it to work at all. However, if many consecutive pose estimates are too noisy, the velocity estimates would be based on pose estimates computed from previous velocity estimates, possibly resulting in even worse accuracy and large drift. Regardless, this is a problem which is important to solve, and this simple method is a good place to start.

Another solution which could possibly work better is to integrate a Inertial Measurement Unit (IMU) into the system, and use the IMU measurements when calculating a local pose estimate if the visual odometry estimate is too noisy. However, if a IMU is to be integrated into the system, the IMU measurements can be utilized in a better way, which is discussed in the next section.

10.1.6 Faster pose estimation

The update frequency of the pose estimates are currently around 5-10 Hz, which is most likely too low if they are to be used in the control system of a drone. Only the feature detection step takes advantage of the GPU on the TX1. In the future, as many as possible of the steps in the visual odometry algorithm should be implemented using CUDA versions of the OpenCV functions currently used in the algorithm, which is likely to increase the real-time performance of the algorithm. It should also be investigated if parts of the current implementation of the algorithm can be done in a more efficient way, and in general, further development of the system should be done with high performance in mind.

Measurements from a IMU, as mentioned in section 10.1.5, can also be used to obtain a faster pose estimation. In [40], IMU measurements are fused with the pose estimates obtained by the visual odometry to achieve higher bandwidth on the pose estimation, and as written in the thesis, "The sensor fusion is able to detect pitch and roll drift and can compensate for the time delay caused by the calculation time of the visual odometry". Integrating a IMU into the system is therefore something that should be considered in the future.

10.1.7 Tuning of parameters

As mentioned in section 6.1.1, a lot of parameters are associated with each step of the visual odometry algorithm, which can be tuned in order to obtain potentially better performance. However, not much time has been spent trying to do this, and more time needs to be invested into this.

10.2 Further work regarding the envisioned drone system

10.2.1 Implementing the maze mapping application

The maze mapping application that was originally developed in [3] and briefly worked on further in [18] needs to be implemented on the TX1, such that the TX1 is able to map wall segments of a maze from above. A challenge regarding this is that in [3] and [18], a Raspberry Pi Camera V2 was used for the mapping, which is not compatible with the TX1. The stereo camera used in this project could possibly be used for the mapping as well, but in that case, the camera needs to be pointed downwards, which may not be optimal for the pose estimation. If this is not the case, another camera needs to be obtained. These are challenges that needs to be addressed and solved in the future.

10.2.2 Drone system in general

Drone When the visual odometry algorithm is able to produce pose estimates regarded as accurate enough, it is time to think about the envisioned drone system. A suitable drone needs to be acquired, and as discussed in [18], several considerations needs to be taken into account when buying the drone, such as size, weight, lift capacity etc. The drone needs to be programmable, it should be delivered with a well documented SDK and it should be possible to attach external hardware on it (such as the TX1). Also, a suitable autopilot needs to be acquired, where an open-source autopilot may be the best choice, as suggested in [18].

Carrier board for the Jetson TX1 The default TX1 developer kit, used during this project, is most likely too large and heavy for attaching it to a drone. However, the full developer kit is not needed, as not all of the ports that the developer kit provides interfaces for is used. Therefore, a smaller carrier board containing interfaces to only the ports and connections needed for this project should be acquired. A carrier board which looks promising, providing access to all the connections needed in this project, such as USB, HDMI, microSD and UART, is the "Orbitty Carrier for NVIDIA Jetson TX2 & Jetson TX1" [10]. An image of the carrier board, while mounted on the TX1 module, can be seen in the following figure:

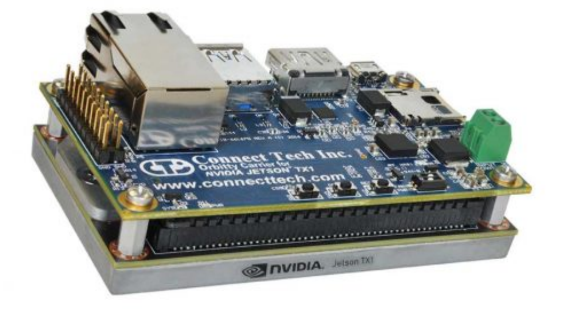


Figure 10.1: The Orbitty Carrier board mounted on a Jetson TX1 module

10.2.3 Further interaction with the server running the mapping application

Send detected wall segments to the server After implementing the maze mapping application, as discussed in section 10.2.1, the detected wall segments needs to be sent to the server application, and is something that needs to be implemented in the future. In [12], the server application was made able to receive measurements in the form of line segments represented as two coordinates, (x_{start}, y_{start}) and (x_{end}, y_{end}). This can be used to represent the wall segments detected by the maze mapping application as well.

Navigational orders The current system is more or less able to connect and stay connected to the server while continually sending the pose estimates computed by the visual odometry algorithm to the server application. The complete drone system, when it is able to both fly using pose estimates from the visual odometry and detect wall segments of a maze using the application developed in [3], is also supposed to be able to get navigational orders from the mapping application, consisting of coordinates where the drone is supposed to navigate to, and use these coordinates as position references in the drones control system and autopilot.

Bibliography

[1] Agrawal, M., Konolige, K., 2006. Real-time localization in outdoor environments using stereo vision and inexpensive GPS.

[2] Beauchemin, S. S., Barron, J. L., 1995. The computation of optical flow. ACM Computing Surveys (CSUR).

[3] Bjørnsen, K., 2017. Mapping a maze with a camera using a raspberry pi. Master's thesis.

[4] Bouguet, J.-Y., 2000. Pyramidal implementation of the lucas kanade feature tracker. Intel Corporation, Microprocessor Research Labs.

[5] Code Laboratories, 2018. DUO API DOCS. Retrieved 10.05.18.
URL `https://duo3d.com/docs/articles/api`

[6] Code Laboratories, 2018. DUO ARM Installation. Retrieved 15.05.18.
URL `https://duo3d.com/docs/install-arm`

[7] Code Laboratories, 2018. DUO Dense3D API. Retrieved 10.05.18.
URL `https://duo3d.com/docs/articles/dense3d-api`

[8] Code Laboratories, 2018. DUO M - Configurable Ultra-compact Stereo Camera. Retrieved 08.05.18.
URL `https://duo3d.com/product/duo-mini-lv1#tab=specs`

[9] CompuPhase, 2018. Termite: a simple RS232 terminal. Retrieved 09.05.18.
URL `https://www.compuphase.com/software_termite.htm`

[10] Connect Tech Inc., 2018. Orbitty carrier for nvidia jetson tx2 & jetson tx1. Retrieved 27.05.18.

URL http://connecttech.com/product/orbitty-carrier-for-nvidia-jetson-tx2-tx1/

[11] Egeland, J., Gravdahl, T., 2003. Modeling and Simulation for Automatic Control. Marine Cybernetics AS.

[12] Eikeland, G., 2018. Implementation of mapping and navigation on an autonomous robot. Master's thesis.

[13] eLinux.org, 2018. Jetson/TX1 Serial Console. Retrieved 15.05.18.
URL https://elinux.org/Jetson/TX1_Serial_Console

[14] Fischler, M. A., Bolles, R. C., 1981. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography.

[15] Hirschmuller, H., 2007. Stereo processing by semiglobal matching and mutual information. IEEE Transactions on Pattern Analysis and Machine Intelligence.

[16] Hirschmuller, H., Innocent, P. R., Garibaldi, J. M., 2002. Fast, unconstrained camera motion estimation from stereo without tracking and robust statistics. Control, Automation, Robotics and Vision.

[17] Iocchi, L., 1998. Stereo Vision: Triangulation. Retrieved 02.05.18.
URL http://www.dis.uniroma1.it/~iocchi/stereo/triang.html

[18] Iversen, B. B., 2017. Integrating a raspberry pi into the lego-robot project. Master's thesis, Norwegian University of Science and Technology.

[19] JetsonHacks, 2017. Run Jetson TX1 from SD Card. Retrieved 09.05.18.
URL http://www.jetsonhacks.com/2017/01/26/run-jetson-tx1-sd-card/

[20] JetsonHacks, 2018. NVIDIA Jetson TX1 J21 Header Pinout. Retrieved 15.05.18.
URL http://www.jetsonhacks.com/nvidia-jetson-tx1-j21-header-pinout/

[21] Kanzow, C., Yamashita, N., Fukushima, M., 2004. Levenberg-Marquardt methods with strong local convergence properties for solving nonlinear equations with convex constraints. Journal of Computational and Applied Mathematics.

[22] Levenberg, K., 1944. A Method for the Solution of Certain Non-Linear Problems in Least Squares. Quarterly of Applied Mathematics.

[23] Lien, K., 2017. Embedded utvikling på en fjernstyrt kartleggingsrobot. Master's thesis.

[24] Lowe, D. G., 1999. Object recognition from local scale-invariant features. Proceedings of the International Conference on Computer Vision.

[25] Lucas, B. D., Kanade, T., 1981. An iterative image registration technique with an application to stereo vision. Proceedings of Imaging Understanding Workshop.

[26] Marquardt, D., 1963. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. SIAM Journal on Applied Mathematics.

[27] McCormick, C., 2018. Stereo vision tutorial - part i. Retrieved 27.05.18.
URL `http://mccormickml.com/2014/01/10/stereo-vision-tutorial-part-i/`

[28] MIT Licence, 2018. FreeRTOS. Retrieved 25.05.18.
URL `https://www.freertos.org/`

[29] NaturalPoint, Inc., 2018. OptiTrack for Robotics. Retrieved 02.06.2018.
URL `https://optitrack.com/motion-capture-robotics/`

[30] NVIDIA Corporation, 2018. CUDA Zone. Retrieved 09.05.18.
URL `https://developer.nvidia.com/cuda-zone`

[31] NVIDIA Corporation, 2018. Download and Install JetPack L4T. Retrieved 15.05.18.
URL `https://docs.nvidia.com/jetpack-l4t/index.html#developertools/mobile/jetpack/l4t/3.0/jetpack_l4t_install.htm`

[32] NVIDIA Corporation, 2018. JetPack Release Notes. Retrieved 15.05.18.
URL `https://developer.nvidia.com/embedded/jetpack-notes`

[33] NVIDIA Corporation, 2018. Jetson Developer Tools. Retrieved 08.05.18.
URL `https://developer.nvidia.com/embedded/develop/tools`

[34] NVIDIA Corporation, 2018. Jetson TX1 Module. Retrieved 08.05.18.
URL `https://developer.nvidia.com/embedded/buy/jetson-tx1`

[35] NVIDIA Corporation, 2018. NVIDIA Jetson TX1/TX2 Developer Kit Carrier Board. Retrieved 01.06.2018.
URL `http://developer.nvidia.com/embedded/dlc/jetson-tx1-tx2-developer-kit-carrier-board-spec-20170615`

[36] NVIDIA Corporation, 2018. Unleash Your Potential with the Jetson TX1 Developer Kit. Retrieved 08.05.18.
URL `https://developer.nvidia.com/embedded/buy/jetson-tx1-devkit`

[37] OpenCV team, 2018. OpenCV - About. Retrieved 09.05.18.
URL `https://opencv.org/about.html`

[38] OpenCV team, 2018. OpenCV - CUDA. Retrieved 09.05.18.
URL `https://opencv.org/platforms/cuda.html`

[39] OpenCV team, 2018. OpenCV modules. Retrieved 10.05.18.
URL `https://docs.opencv.org/3.3.0/`

[40] Post, T. T., 2015. Precise localization of a uav using visual odometry. Master's thesis, University of Twente, master's thesis.

[41] Rosten, E., Drummond, T., 2006. Machine learning for high-speed corner detection. In: Leonardis A., Bischof H., P. A. (Ed.), Computer Vision - ECCV 2006. Vol. 3951. Springer, Berlin, Heidelberg.

[42] Rublee, E., Rabaud, V., Konolige, K., Bradski, G., 2011. Orb: An efficient alternative to sift or surf. IEEE Computer Vision (ICCV).

[43] Scaramuzza, D., Fraundorfer, F., 2011. Visual odometry, part i: The first 30 years and fundamentals. IEEE Robotics & Automation Magazine.

[44] Scaramuzza, D., Fraundorfer, F., 2012. Visual odometry, part ii: Matching, robustness, optimization, and applications. IEEE Robotics & Automation Magazine.

[45] Schloesser, L., 2018. Odometryutils. Retrieved 01.06.18.
URL `https://gist.github.com/llschloesser/5ce412e652b5c126e18c4c40c9d31185`

[46] Siegwart, R., Nourbakhsh, I., Scaramuzza, D., 2011. Introduction to Autonomous Mobile Robots. MIT Press Ltd.

[47] Wikipedia contributors, 2018. Camera resectioning — Wikipedia, the free encyclopedia. Retrieved 27.05.18.
URL `https://en.wikipedia.org/wiki/Camera_resectioning`

[48] Wikipedia contributors, 2018. Image rectification — Wikipedia, the free encyclopedia. Retrieved 27.05.18.
URL `https://en.wikipedia.org/wiki/Image_rectification`

Appendix A

Installation instructions

A.1 Run TX1 from an external SD card

For a complete guide, see [19]. The main steps are presented below.

1. Format the SD card with a ext4 format, with minimum one partition of a size of at least 16GB

2. Insert the SD card into the SD card reader on the TX1, start the TX1 and navigate to the SD card folder

3. Copy the contents of the root directory of the internal flash memory over to the SD card (called 'SD Root' here):

   ```
   $ sudo cp -ax / '/media/ubuntu/SD Root'
   ```

4. Backup the extlinux.conf file, then edit it:

   ```
   $ cd /boot/extlinux
   $ sudo cp extlinux.conf extlinux.conf.original
   $ sudo gedit /boot/extlinux/extlinux.conf
   ```

5. Edit the file such that it looks like this:

   ```
   TIMEOUT 30
   DEFAULT sdcard
   ```

```
MENU TITLE p2371-2180 eMMC boot options

LABEL sdcard
MENU LABEL SD Card
LINUX /boot/Image
INITRD /boot/initrd
FDT /boot/tegra210-jetson-tx1-p2597-2180-a01-devkit.dtb
APPEND fbcon=map:0 console=tty0 console=ttyS0,115200n8 androidboot.
    modem=none
androidboot.serialno=P2180A00P00940c003fd androidboot.security=non-
    secure tegraid=21.1.2.0.0 ddr_die=2048M@2048M
ddr_die=2048M@4096M section=256M memtype=0 vpr_resize
    usb_port_owner_info=0 lane_owner_info=0 emc_max_dvfs=0
touch_id=0@63 video=tegrafb no_console_suspend=1 debug_uartport=
    lsport,0 earlyprintk=uart8250-32bit,0x70006000
maxcpus=4 usbcore.old_scheme_first=1 lp0_vec=${lp0_vec}
    nvdumper_reserved=${nvdumper_reserved} core_edp_mv=1125
core_edp_ma=4000 gpt android.kerneltype=normal androidboot.
    touch_vendor_id=0 androidboot.touch_panel_id=63
androidboot.touch_feature=0 androidboot.bootreason=pmc:software_reset
    ,pmic:0x0 net.ifnames=0 root=/dev/mmcblk1p1
rw rootwait

LABEL internalemmc
MENU LABEL Internal EMMC
LINUX /boot/Image
INITRD /boot/initrd
FDT /boot/tegra210-jetson-tx1-p2597-2180-a01-devkit.dtb
APPEND fbcon=map:0 console=tty0 console=ttyS0,115200n8 androidboot.
    modem=none
androidboot.serialno=P2180A00P00940c003fd androidboot.security=non-
    secure tegraid=21.1.2.0.0 ddr_die=2048M@2048M
ddr_die=2048M@4096M section=256M memtype=0 vpr_resize
    usb_port_owner_info=0 lane_owner_info=0 emc_max_dvfs=0
touch_id=0@63 video=tegrafb no_console_suspend=1 debug_uartport=
    lsport,0 earlyprintk=uart8250-32bit,0x70006000
maxcpus=4 usbcore.old_scheme_first=1 lp0_vec=${lp0_vec}
    nvdumper_reserved=${nvdumper_reserved} core_edp_mv=1125
core_edp_ma=4000 gpt android.kerneltype=normal androidboot.
    touch_vendor_id=0 androidboot.touch_panel_id=63
androidboot.touch_feature=0 androidboot.bootreason=pmc:software_reset
    ,pmic:0x0 net.ifnames=0 root=/dev/mmcblk0p1
rw rootwait
```

6. Save the file and restart the TX1. It will now boot from the SD card

Connecting a serial cable to the TX1, as described in chapter 5 section 5.1, can be useful during this process. Especially to see that the TX1 actually boots from the SD card.

A.2 Installing OpenCV

The following is an instruction of how to install OpenCV 3.3.0 on the TX1 with extra modules, including the GPU accelerated CUDA module.

First, update all Ubuntu packages:

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Then, install the necessary packages (the following command can be copied and pasted into the terminal to install all packages at once):

```
$ sudo apt-get install \
    libglew-dev \
    libtiff5-dev \
    zlib1g-dev \
    libjpeg-dev \
    libpng12-dev \
    libjasper-dev \
    libavcodec-dev \
    libavformat-dev \
    libavutil-dev \
    libpostproc-dev \
    libswscale-dev \
    libeigen3-dev \
    libtbb-dev \
    libgtk2.0-dev \
    pkg-config
```

Fetch the OpenCV repository from github and get the desired version (3.3.0 in this case):

```
$ git clone https://github.com/opencv/opencv.git
$ cd opencv
$ git checkout 3.3.0
$ cd ..
```

Fetch the extra contrib modules, also from github:

```
$ git clone https://github.com/opencv/opencv_contrib.git
$ cd opencv_contrib
$ git checkout 3.3.0
$ cd ..
```

Create a build folder inside the newly created opencv directory and navigate to it:

```
$ cd opencv
$ mkdir build
$ cd build
```

Now, configure the building of OpenCV using CMake. The configuration command includes a lot of different flags. Flags to take extra note of are the following:

- -DOPENCV_EXTRA_MODULES_PATH=/home/Ubuntu/Desktop/opencv_contrib/modules
- -DWITH_CUDA=ON

which enables OpenCV to be built with the contrib modules and with the CUDA/GPU module, respectively. It is here assumed that the contrib modules are located at "/home-/Ubuntu/Desktop/opencv_contrib/modules". Another thing to be aware of is that the flag "-DOPENCV_EXTRA_MODULES_PATH=" has to come first, or else the extra contrib modules won't be built.

```
$ cmake
-DOPENCV_EXTRA_MODULES_PATH=/home/ubuntu/Desktop/opencv_contrib/modules \
    -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_INSTALL_PREFIX=/usr/local \
    -DBUILD_PNG=OFF \
    -DBUILD_TIFF=OFF \
    -DBUILD_TBB=ON \
    -DBUILD_JPEG=OFF \
    -DBUILD_JASPER=OFF \
    -DBUILD_ZLIB=OFF \
    -DBUILD_EXAMPLES=ON \
    -DBUILD_opencv_java=OFF \
    -DBUILD_opencv_python2=ON \
    -DBUILD_opencv_python3=OFF \
    -DENABLE_PRECOMPILED_HEADERS=OFF \
    -DWITH_V4L=ON \ -DWITH_QT=ON \
    -DWITH_OPENGL=ON \
    -DWITH_OPENCL=OFF \
    -DWITH_OPENMP=OFF \
    -DWITH_FFMPEG=ON \
    -DWITH_GSTREAMER=OFF \
    -DWITH_GSTREAMER_0_10=ON \
    -DWITH_CUDA=ON \
    -DWITH_GTK=ON \
    -DWITH_VTK=OFF \
    -DWITH_TBB=ON \
    -DWITH_1394=OFF \
    -DWITH_OPENEXR=OFF \
    -DCUDA_TOOLKIT_ROOT_DIR=/usr/local/cuda-8.0 \
```

Then, compile (using four CPU cores), install and update ldconfig:

```
$ make -j4
$ sudo make install
$ sudo ldconfig
```

The compilation process may take up to a couple of hours to complete.

A.3 Compile and run the complete system application

The source code for this project is given in B.2.

After booting the Jetson TX1 with username/password (both): ubuntu, first make sure that the DUO M driver is loaded, as described in section 5.3. Then, given that the source code is extracted and copied to the Desktop, navigate to a folder named "TX1", then create a folder named "build" and navigate to it:

```
$ cd ~/Desktop/drone-mapping/TX1
$ sudo mkdir build
$ cd build
```

Then, compile the application using CMake (the program is compiled according to specific settings and "rules", which are set in the file "CMakeLists.txt" in the TX1-folder):

```
$ cmake ..
$ make
```

Now, an executable file named "main" is created in the "build" folder. Running the file with the following command will launch the application:

```
$ ./main
```

Appendix B

Description of attachments

B.1 Images

Images used in the report.

B.2 Source code

The source code developed and used in this project.

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.