Dynamic path planning of initially unknown environments using an RGB-D camera

Sara Gustafzelius

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--5980--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2015 by Sara Gustafzelius. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2015

Abstract

In this thesis an RGB-D camera was used with the goal to perform dynamic path planning in an initially unknown environment. Depth data from an RGB-D camera together with a discretizising algorithm is continuously used for maintaining an obstacle map of the environment which within the path planning algorithm D* Lite [S. Koening, 2005] is performed on the flight.

Experiments were conducted on two different systems, on Combine's hexacopter and on a Gantry Tau robot at the Robot Lab of the Department of Automatic Control, LTH. On Combine's hexacopter different tracking algorithms such as ICP, Translation Approximation and SDF where evaluated for 3D positioning while the robots internal positioning where used on the Gantry Tau robot.

For discretization purposes we compare the use of Box Approximation and Signed Distance Function (SDF) for creating the obstacle map.

Acknowledgments

First of all I want to thank my company supervisor Simon Yngve who has guided and supported me all through my thesis. His visual and sometimes overwhelming ideas in combination with all kind words have been vital for this thesis.

I also want to thank my university supervisor Prof. Anders Robertsson from the Department of Automatic control for sharing a little piece of this enormous expertise in automation and robotics with me as well as my second university supervisor Erik Bylow from the Department of Mathematics for all the help he gave me with the image analysis and his patience with me.

I also want to thank Maj Stenmark from the Department of Computer Science for her help with the Labcomm robot communication setup as well as her help with RobotStudio.

Finally I want to thank my family for always pushing me in the right direction.

Symbols

A	Weighting coefficient matrix
α	Weighting coefficient
C	Camera matrix
d	Denotes a distance
H	Homogenous transformation
I_d	Depth image
I_{RGB}	RGB image
M	Transformation matrix that centers the cameras around the origin
P	Denotes a centroid
R	Rotation matrix
t	Translation vector
X	A 3D point
X_G	Global 3D point
X_L	Local 3D point

Acronyms

CPU	Central Processing Unit
ICP	Iterative Closest Point
IMU	Inertial Measurement Unit
RGB	Red Green Blue, standard way of representing colors in images.
RGB-D	Red Green Blue and Depth values
SDF	Signed Distance Function
TLC	Target Language Compiler
TUM	Technische Universität München
UAV	Unmanned Aerial Vehicle

Contents

1. Introduction **15**
1.1 Previous work 15
1.2 Problem formulation 16
1.3 Thesis outline 16

2. Preliminaries and Background **18**
2.1 Pinhole Camera 18
2.2 Signed distance function 22
2.3 Iterative closest point 23
2.4 Shortest path problem 24
2.5 Priority heap 25
2.6 D* Lite 27

3. System description **29**
3.1 System overview 29
3.2 Image retrieval 31
3.3 The obstacle map 32

4. Updating the obstacle map **34**
4.1 Transforming depth data into global coordinates 34
4.2 Why do we need data discretization? 34
4.3 Discretization methods 36

5. Path planning **40**
5.1 Implementation using the obstacle map 40
5.2 Method 42

6. Gantry-Tau robot **44**
6.1 Previous work 44
6.2 Overview 44
6.3 Camera-Robot transformation 45
6.4 Experimental setup 51

7. Combine hexacopter **53**
7.1 Previous work 53

7.2 Hardware assembly 54
7.3 Tracking 55
7.4 Experimental setup 56

8. Software implementation **57**
8.1 Simulink - C/C++ interface 57
8.2 Simulink model 59

9. Result **62**
9.1 Image retrieval 63
9.2 Updating the Obstacle map 64
9.3 Combine hexacopter 66
9.4 Gantry-Tau robot 71
9.5 Path planning 72

10. Conclusions **73**
10.1 Image retrieval 73
10.2 Updating the Obstacle map 73
10.3 Combine hexacopter 74
10.4 Gantry-Tau robot 74
10.5 Path planning 74

11. Discussion **75**
11.1 Problems 75
11.2 Further work 75

Bibliography **76**

A. Appendix A **79**
A.1 Camera calibration 79
A.2 Hand-Eye Calibration 81

B. Appendix B **84**
B.1 Labcomm client 84
B.2 Simulink wrappers 86

1

Introduction

There are currently many fields of applications where traversing non human friendly environments with unmanned vehicles can be highly useful. Further, as the use of, for example, Unmanned Aerial Vehicles (UAVs) increases it is not hard to see the benefit of not only unmanned but also automated unmanned vehicles.

In this thesis an RGB-D camera was used with the goal to perform dynamic path planning. Depth data from an RGB-D camera is used to create and continuously update an obstacle map of the environment. The obstacle map contains information of which coordinates can be traveled and not traveled. When a new obstacle is detected by the RGB-D camera the location of the obstacle is calculated and marked as untraversable in the obstacle map. The path planning algorithm is initially given a start and goal position and a first path is calculated using the obstacle map. When a new obstacle occurs on the obstacle map the path planning algorithm dynamically re-plans the local path part affected by the obstacle.

1.1 Previous work

In [P. E. Hart, 1968] the authors presented an extension of the well known Edsger Dijkstra's algorithm called A*. It uses heuristics to achieve better time performance. 1994 A. Stentz introduces a dynamic version of A* called D* [Stentz, 1994]. The algorithm is considered dynamic since the graphs edge costs can change during the traversal. Thereafter a version called Focused D* [Stentz, 1995] was presented. Focused D* reduced the run-time by two to three times by reducing state expansions. This is accomplished by using heuristics that consider (focus) the direction of the robot. If no costs are changed during the traverse the solution is identical to A*. Later Lifelong Planning A* (LPA*) [S. Koening, 2002b] was presented. LPA* is an incremental algorithm that uses heuristics from A*, focus the search and reuses unchanged parts from previous searches. LPA* uses DynamicSWSF-FP (dynamic SWSF fixed point) [G. Ramalingam, 1992] to recompute all goal distances affected by changes during the traverse and is able to re-plan faster than both

A* and DynamicSWSF-FP. The first iteration is identical to A*. Short after publishing LPA* the same authors presented the D* Lite algorithm [S. Koening, 2005] that build on LPA*. D* Lite plans the same path as the more complex algorithm Focused Dynamic A* (D*) but is algorithmically different and easier to overview. It is stated that that D* Lite is at least as efficient as D* [S. Koening, 2002a].

1.2 Problem formulation

This thesis aims to:

- Investigate how path-planning and automatic re-planning can be performed using an RGB-D camera in an initially unknown environment.
- Investigate how depth data can be used to efficiently discretisize an initially unknown environment and compare different approaches.

1.3 Thesis outline

Chapter 2 Preliminaries and Background - Here we present basic notations, the pinhole camera model, depth images, rigid body transformation of local coordinates into global coordinates and basic theory about signed distance functions, iterative closest point and path planning algorithms.

Chapter 3 System description - Here we present the system overview and describe the hardware setup of the UAV such as the Pandaboard and the chosen RGB-D camera as well as an argumentation of this choice.

Chapter 4 Updating the obstacle map - In this chapter we show the basic transformations used and explain the need of a discrete grid for data representation and two possible approaches of achieving this.

Chapter 5 Path planning - In this chapter we describe a method of path planning in three dimensions.

Chapter 6 Gantry Tau robot - This chapter contains information about the Gantry Tau robot experiments and covers previous work and brief information about the Gantry Tay robot as well as the Labcomm robot communication protocol and practical issues such as timing problems and how those where solved in the experimental setup.

Chapter 7 Combine hexacopter - Here we talk about the Combine hexacopter experiment and cover previous work done on the hexacopter, hardware assembly and two different tracking algorithms for estimating the camera trajectory.

Chapter 8 Software implementation - In this chapter we motivate the basic software implementation and explain the main software problems and found solutions.

Chapter 9 Result - Here we show the results of the methods used in this thesis. We compare time and performance of the different discretization methods for updating the obstacle map. For the Combine hexacopter we compare the tracking algorithms on ground truth data sets and show the results of the camera-robot transformation on the Gantry-Tau robot. Finally we evaluate the path planning and overall model on live data.

Chapter 10 Conclusions - Here we present the conclusions that can be drawn by this thesis.

Chapter 11 Discussion - In this chapter we evaluate the results in Chapter 9 and discuss improvements of our methods and system setup.

2

Preliminaries and Background

This chapter aims to give the reader basic knowledge about the different algorithms and data structures needed to comprehend the content of this thesis.

It covers an introduction of the pinhole camera used to receive information from the depth images, a short explanation of the signed distance function that can be used for discretizaton and tracking and the well-known iterative closest point algorithm (ICP) used for tracking. It also describes the programming data structure called priority heap that is used to improve the efficiency of the path planning algorithm, an introduction to the shortest path problem which describes basic graph theory used in the path planning, as well as the path planning algorithm D* Lite.

2.1 Pinhole Camera

The pinhole camera model can be used to project a 3D point into the image plane, this is illustrated in Figure 2.1. To project a 3D point into the image plane we need to know the focal length f of the camera, which is the distance between the camera center and the image plane, as well as the pixel coordinates c_x and c_y where the principal axis and the image plane meets.

In Figure 2.2 which shows a regular RGB camera using the pinhole camera model, only the relative distance between the x, y and z coordinates can be measured. By using triangulation we can calculate where on the image plane a certain 3D point is projected.

Looking at Figure 2.3 we see that there are two similar triangles (O_c, x, f) and (O_c, x', z') where (O_c, x, f) is a triangle between the camera origin, the x coordinate on the image plane and the focal length f and (O_c, x', z') is a triangle between the camera origin, the x coordinate of the 3D point and the distance between the camera origin and the 3D point along the optical axis. Using this the following equation can be stated.

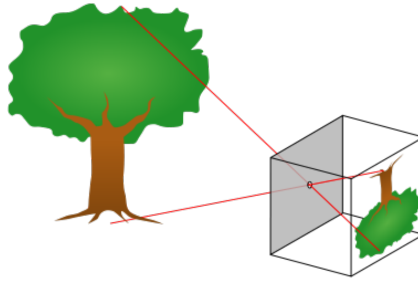

Figure 2.1 Pinhole camera model

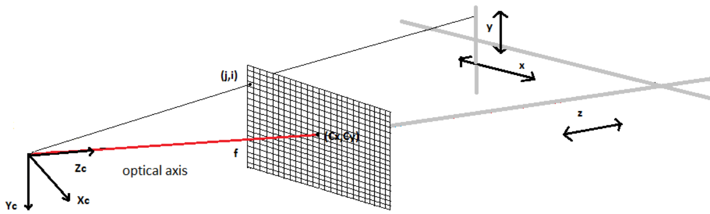

Figure 2.2 Pinhole camera from RGB camera. The gray lines marks the relative distance between the coordinates.

$$\frac{x}{f} = \frac{x'}{z'} \iff x = f\frac{x'}{z'} \tag{2.1}$$

The y coordinates are derived analogously.

We realize that a 3D coordinate $X = (x,y,z)$ is projected onto a pixel on the image plane $I_d(i,j)$ as,

$$I_d(i,j) = (\frac{f_x x}{z} + c_x, \frac{f_y y}{z} + c_y),$$

where f_x, f_y, c_x and c_y are intrinsic camera parameters.

When using a RGB-D camera the distance d along the optical axis from the camera to the 3D point can be measured. When a 3D point is projected on the image

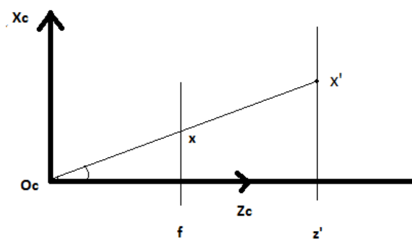

Figure 2.3 Triangulation between the camera origin, the x coordinate on the image plane along the z axis and the triangle between the camera origin and the x coordinate of 3D point along the z axis.

plane the distance d is saved in every pixel. Since d is known in every pixel we can use triangulation in the same way as above to calculate the x and y coordinates of the 3D point. Looking at Figure 2.3 again we see that,

$$\frac{x}{f} = \frac{x'}{z'} \iff x' = z'\frac{x}{f}. \tag{2.2}$$

Just as before the y coordinates are derived analogously.

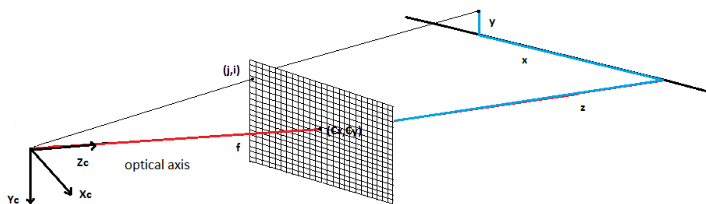

Figure 2.4 Pinhole camera from depth sensor

Figure 2.4 shows the relationship between the pixel location on the image plane and the 3D coordinates when an RGB-D camera is used. The blue lines marked with x, y and z in Figure 2.4 illustrated the distances between the 3D coordinate derived from the center pixel and the 3D coordinate derived from the pixel we are interested in. Since the depth z is known the 3D coordinates can be fixed and therefore the blue lines symbolizes the absolute distance.

We realize that a pixel (i,j) in a depth image I_d with depth $z = I_d(i,j)$ is transformed to its local 3D coordinate $X = (x,y,z)$ as,

$$X = (x,y,z) = (\frac{(i-c_x)z}{f_x}, \frac{(j-c_y)z}{f_y}, z)^T,$$

where $(i,j) \in I_d$, $z = I_d(i,j)$ and f_x, f_y, c_x and c_y are intrinsic camera parameters.

In order to combine local coordinates from different cameras we need to use a common frame by using global coordinates. To move the local coordinates to global coordinates we need to know the global configuration of the camera in terms of rotation and translation.

The camera matrix C will move the local 3D coordinates (x_L, y_L, z_L) into global 3D coordinates (x_G, y_G, z_G) using a rigid body transformation.

$$C = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$$

where R is a rotation matrix in $\mathbb{SO}^3$, rotations about the origin in a three-dimensional Euclidean space, and t is the translation in $\mathbb{R}^3$.

We then get,

$$X_G = C \cdot X_L,$$

where X_L are the local coordinates and X_G are the global coordinates.

It is well known that multiplication between two matrices is only defined if the numbers of columns in the first matrix equals the number of rows in the second matrix. To be able to do the calculation $C \cdot X_L$ we then have to choose X_L as $X_L = (x_L, y_L, z_L, 1)$. By using $C = \begin{pmatrix} R & t \\ 0 & 1 \end{pmatrix}$ we are able to get $X_G = (x_G, y_G, z_G, 1)$

2.2 Signed distance function

One way of representing the global model is by using a voxel grid of vertices equally spread out in the room. Then a Signed Distance Function (SDF) can be used to represent the location of surfaces in the room.

A signed distance function is a function which gives the signed distance d between a point $X \in \mathbb{R}^3$ and the closest point X_s on a surface S.

$$SDF(X) = d,$$

where $d < 0$ if X is outside the surface and $d > 0$ if X is inside the surface, as described in [Bylow, 2012, p. 17].

The SDF of a vertex will be negative if the vertex is in front of the surface, zero if the vertex is on the surface, and positive if the surface is closer to the camera then the vertex. For two dimensions this can be illustrated as in Figure 2.5.

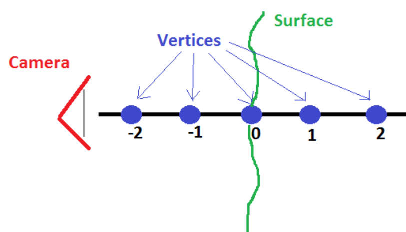

Figure 2.5 SDF example in two dimensions

2.3 Iterative closest point

The task of iterative closest point (ICP) is to find an optimal rigid body 3D transformation M that moves a model set s to a data set r so that the total error between corresponding points is minimal. ICP was introduced in [P.J. Besl, 1992]. Each point in the data set should be matched to its closest model point. Then we want to find a camera matrix C so that the sum of the square of the distances between the transformed model set points s and the data set points r is minimized.

$$C = arg\ min_C E,$$

where the error function is defined as,

$$E_{point-to-point} = \sum_i (Cs_i - r_i)^2,$$

if we use point-to-point error metrics [Treiber, 2013, p. 141-142], and defined as

$$E_{point-to-plane} = \sum_i ((Cs_i - r_i)^T n_i)^2,$$

if we use point-to-plane error metrics where n_i is the unit normal vector at r_i [Low, 2004, p. 1]

The point-to-point error metrics, as suggested by the name, is simply the distance between each source point s_i and its matched destination point r_i. The point-to-plane error metrics is the distance between each source point s_i and the tangent plane at the matched destination point r_i as shown in Figure 2.6.

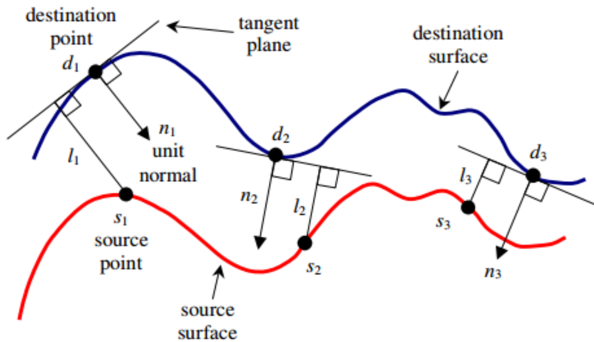

Figure 2.6 Point-to-plane error in two dimensions as described in [Low, 2004, p. 1]

2.4 Shortest path problem

The path planning in this thesis can be reduced to a shortest path problem. A shortest path problem is the problem of finding the shortest path between two vertices in a graph so that the total cost for the path is minimized. In graph theory a map of roads and intersections are called a graph. The roads are called edges and the intersections between the roads are called vertices or nodes. Every edge has a cost for traveling on it and this can be thought of as road fee or the length of a road.

An example of this is to find the shortest route from one city to an other. The cities are then the nodes and the roads between two cities are the edges. The weight of an edge is then the distance of the road.

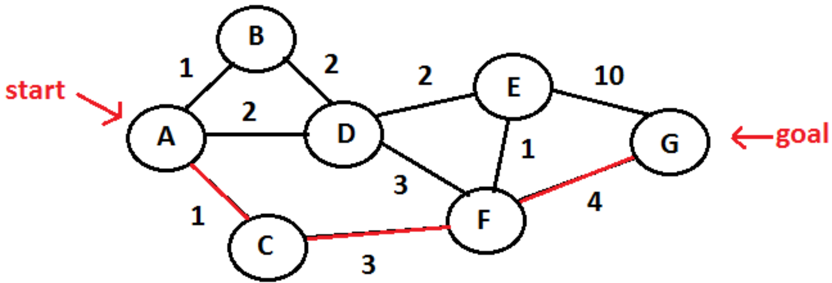

Figure 2.7 Example of a shortest path problem

In Figure 2.7 we see an example on a shortest path problem. The goal is to find the shortest path between vertex A and vertex G. The numbers next to the edges are the cost for traveling on the edge. We realize that the shortest path between A and G that minimized the total path cost is the path marked with red. $A \rightarrow C \rightarrow F \rightarrow G$ gives a total cost of 8.

2.5 Priority heap

A binary heap is a programming data structure that can be used to structure data in an efficient way. The data will be organized so that the node with highest priority always will be at the top of the heap. This can be used in shortest path problems where the highest priority symbolizes the edge with lowest travel cost.

The priority heap uses a binary tree but with two additional properties:

Structure property Each level except the bottom level must be completely filled. The bottom level is filled from left to right. This means that no holes are tolerated in the binary heap and makes the height of the heap with N elements $log_2(N)$.

Heap-order property The heap can be implemented in either a max-heap or a min heap. In a max-heap the element with the highest key has the highest priority and in a min-heap the element with the lowest key has the highest priority. In this thesis a min-heap is used. The key of every node is lower than or equal to the key of its parent node except for the root node which has no parents. Therefore the root node always has the lowest key in the heap. Insert and remove operations must preserve the heap-order property.

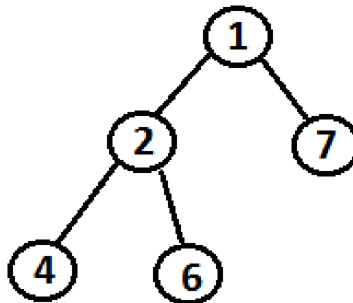

Figure 2.8 Example of structure and heap-order properties of a min-heap.

The main reason for using a priority heap is the fast insert and remove operations. An example of a min-heap can be seen in Figure 2.8.

Since the height of the heap is $log_2(N)$ the cost for insert and remove is $\mathcal{O}(log_2(N))$. A short explanation to the operation is given below.

Insert

1. Insert the new node at the next available slot in the heap.
2. Heap size is increased by one.
3. Percolate up the node until the heap-order property is satisfied.

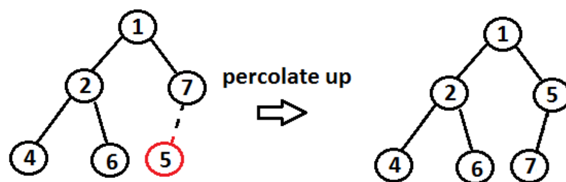

Figure 2.9 Example of insert operation in a min-heap

Remove

1. Delete the node and move the last node into the hole.
2. Heap size is decreased by one.
3. Percolate down the node until the heap-order property is satisfied.

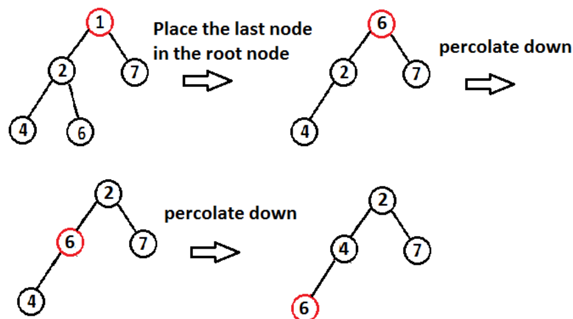

Figure 2.10 Example of remove operation in a min-heap

2.6 D* Lite

D* Lite is a path-planning algorithm [S. Koening, 2005] that builds on Lifelong planning A* (LPA*). D* Lite plans the same path as the more complex algorithm Focused Dynamic A* (D*) by Stentz but is algorithmically different and easier to overview.

The main idea of D* Lite is to do an initial search identical to the A* algoritm and to start move along the calculated path. When an obstacle that affects the planned route is detected the subsequent search uses information from the previous searches to locally re-plan the path. D* Lite uses a priority heap to sort the vertices with decreasing travel cost.

Terminology

Every vertex has a:

g The true distance from a specific node to the goal node.

h The Manhattan distance between the current start node and the current node.

rhs The estimated distance to the goal node.

Key The key is used to order the priority queue, where the lowest key has the highest priority, i.e.,

$$key(s) = [min(g(s), rhs(s)) + h(s, start) + k_m, min(g(s), rhs(s))],$$

where s is the node of interest.

The key consists of two values: the first which is most significant, is the shortest possible way left to the goal node, the smallest of the nodes g value and the rhs value, plus the h value, plus k_m. k_m is the number of steps we moved since the first start node. The second key value is the smallest of the nodes g value and rhs value.

Every edge has a:

C The cost for moving on the edge from one node to another.

A vertex is called locally consistent if g(s) = g(rhs), this means that the vertex has been expanded. If g(s) != rhs(s) the vertex is called locally inconsistent. A locally inconsistent vertex is called locally overconsistent if g(s) > rhs(s) and a locally underconsistent if g(s) < rhs(s).

If all vertices are locally consistent we can find the shortest path from any vertex to the goal vertex. The algorithm uses a priority queue that contains locally inconsistent vertexes and are used to expand the vertices in increasing order of their keys.

In order to avoid reordering the heap every time the robot takes a step along the chosen path a key modifier k_m is used.[S. Koening, 2005]

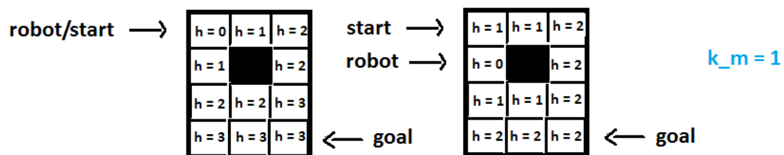

Figure 2.11 The figure to the left shows the initial h values and the figure the right shows the h values after the first iteration

3

System description

3.1 System overview

The system architecture is described in Figure 3.1 and below we give a description for each of the blocks.

Image retrieval This block fetches the images from the RGB-D camera. This is done by using the open source software OpenNI2 [*OpenNI 2 SDK*] interfaced to Simulink see Chapter 8.

Updating the obstacle map In order to use the information stored in the depth images we need to transform pixel/depth information into coordinates. First the local coordinates are calculated using the pinhole camera approach. Given the camera position the global coordinates are calculated. To save computational space and power the global coordinates are discretisized into a grid structure forming an obstacle map.

Combine hexacopter Since the Combine hexacopter does not have any internal positioning a tracking algorithm is used. The tracking algorithm calculates the camera position of a new image given the last image and last camera position.

Gantry Tau robot The Gantry Tau robot implements internal position and therefore no tracking algorithm is needed.

Path planning Given a defined start and goal position the algorithm should calculate a suitable path (if one exist) from the start node to the goal node in an initially unknown environment. The algorithm should be dynamic and automatically update the chosen path if obstacles are positioned on the current path.

Moving phase This is where the vehicle should move to the calculated position given in global coordinates.

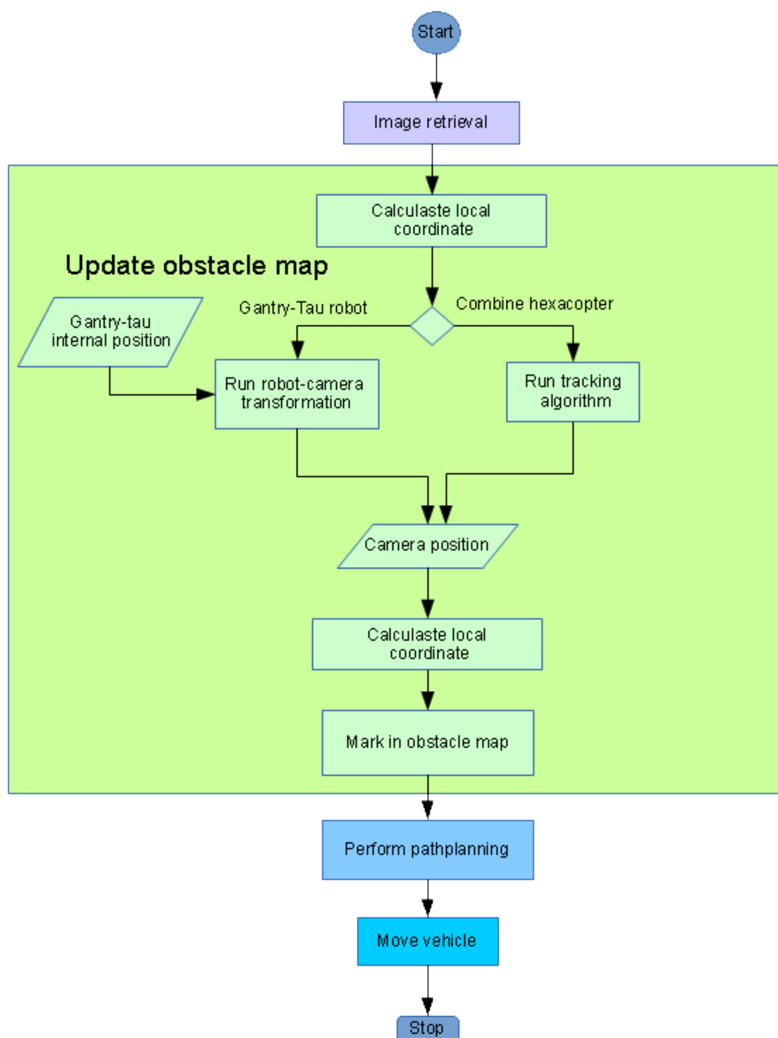

Figure 3.1 Flowchart over the system blocks described in Chapter 3.1.

Figure 3.1 shows a flowchart of the system and gives a better understanding of where he Gantry-Tau robot and Combine hexacopter uses different sub processes.

3.2 Image retrieval

In this thesis an RGB-D camera is to be used to collect depth data. We reviewed two different cameras to see which best fitted our purpose, the Microsoft Kinect for Windows and the Asus Xtion Pro Live.

The Kinect for Windows contains a RGB camera that stores three channel data in a 1280×960 resolution, an infrared (IR) emitter and detector that captures a depth image, multi-array microphone and an accelerometer [*Kinect for Windows Sensor Components and Specifications* 2011]. The product dimensions are $38.1 \times 38.1 \times 12.4$ cm and it weights about 1.09kg. A 640×480 RGB-D image can be obtained at 30 Hz. The impossibility to lower the resolution if wanted is seen as a disadvantage. Another disadvantage is that power is drawn from an external power supply which is highly unpractical when mounted on a moving vehicle.

The Xtion Pro Live features an RGB camera, an infrared (IR) emitter and detector that captures a depth image and a microphone. Xtion Pro Live has the product dimensions $18 \times 3.5 \times 5$ cm which makes it noticeably smaller than the Kinect [*Xtion PRO LIVE*]. It weights about 170 grams and uses a USB2.0/ 3.0 interface for power consumption. A 640×480 or 320×240 RGB-D image can be obtained at 30 Hz.

The small format and light weight in combination with the fact that an external power supply is not needed and the possibility to receive 320×240 pixel resolution made us decide to use the Asus Xtion Pro Live in our hardware setup.

The camera delivers two matrices, one with depth data and one with RGB data. In the depth image we get a depth measurement in every pixel. In Figure 3.2 we see depth data from the Freiburg1 Teddy sequence [*Freiburg1 Teddy sequence* 2011] interpreted as an greyscale. In figure 3.3 we see the corresponding RGB image.

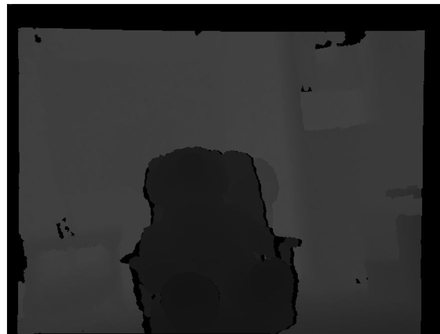

Figure 3.2 Depth data from the Freiburg1 Teddy sequence [*Freiburg1 Teddy sequence* 2011]

Figure 3.3 RGB data from the Freiburg1 Teddy sequence [*Freiburg1 Teddy sequence* 2011]

3.3 The obstacle map

The main idea of the obstacle map is to avoid building a graph for the path planning and instead use a structure that both the discretization algorithms and the path planning algorithm can use. The obstacle map will be represented as a three-dimensional matrix. In Figure 3.4 we see an environment in two dimensions seen from above. The environment contains two obstacles.

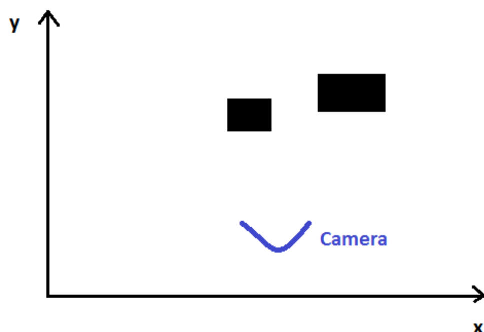

Figure 3.4 An environment in two dimensions seen from above.

We apply a grid in order to discretize the environment. The result can be seen in Figure 3.5 where every square in the grid is an index in the obstacle map and will be referred to as nodes in the path planning.

If a square contains an obstacle the corresponding index in the matrix will be set to one. This means that this node will be untraversable when applying the path planning algorithm. If a square is free from obstacles the corresponding index will

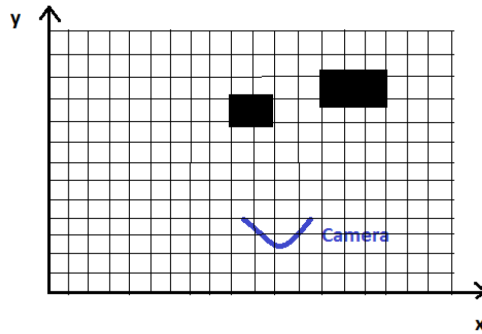

Figure 3.5 The environment seen in Figure 3.4 when a grid is applied.

be set to zero. For the path planning algorithm this means that this node will be traversable. The result of this an obstacle map filled with ones and zeros as in Figure 3.6.

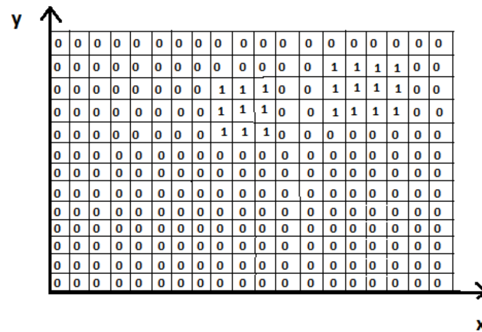

Figure 3.6 The obstacle map corresponding to the environment in Figure 3.4.

4

Updating the obstacle map

4.1 Transforming depth data into global coordinates

The depth image data received by the RGB-D camera is transformed into local coordinates using the pinhole camera model.

$$X = (x,y,z)^T = (\frac{(i-c_x)z}{f_x}, \frac{(j-c_y)z}{f_y}, z)^T,$$

where $(i, j) \in I_d$, $z = I_d(i, j)$ and f_x, f_y, c_x and c_y are intrinsic camera parameters (see Section 2.1).

To be able to use data from different camera positions we need to transform the local coordinates into global coordinates. We use,

$$X_G = C \cdot X_L,$$

where X_L are the local coordinates, X_G are the global coordinates and C is the camera matrix i.e., the position and rotation of the camera.

4.2 Why do we need data discretization?

In order to generate a map of the world frame that later can be used for path planning the global 3D coordinates should preferably be discretizised and entered into some kind of grid.

If we for instance use the 640×480 format for the depth images one image will contain 307200 uint16_t values that in the worst case will generate 307200 different 3D coordinates represented by floats or doubles. Since we are interested in viewing the room in meters integer values are not suitable as data representation.

The standard sampling rate for the Kinect or Asus Pro live is 30fps. This means that running for just one second will in worst case create $30 \times 307200 = 921600$ 3D coordinates which is 3686.4kB if floats are used and 7372.8kB if doubles are used. If we decide to create 3D coordinates for 5 minutes with 30fps without using

some kind of discretizising grid we will end up with $921600 \times 60 \times 5 = 276480000$ 3D points, 1105.92MB if we use floats and 2211.84MB if we use doubles. We understand that the data size will grow uncontrolled if we do not use some kind of grid that it is needed for computational reasons.

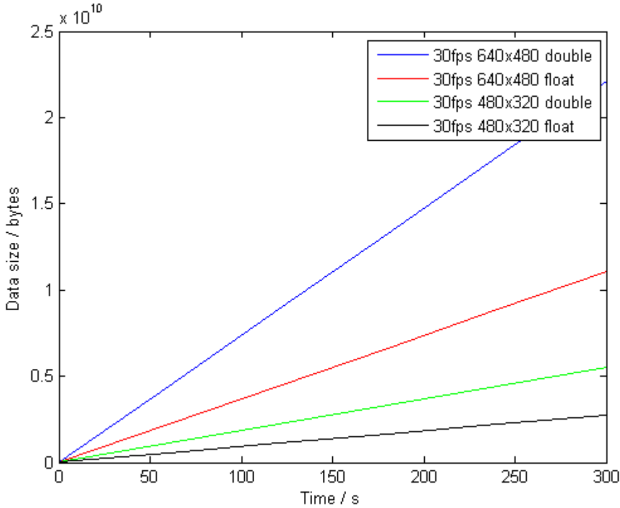

Figure 4.1 Data growth over five minutes when not using a discretizising grid for two different resolutions and two different data representations

In this thesis we have tried two different approaches to discretizise the room. The first one is using an SDF in a voxel grid and the second one is to use a method we call box approximation. In both cases we will use the same size of the room and the same amount of nodes. The grid bounds of the room and grid resolution should of course be chosen to fit the environment in which we want to use the application.

To symbolize the purpose of using a grid we will calculate the amount of 3D points as above. If we assume that we want a resolution of 5 cm, a resolution that should be good enough to perform path planning with our UAV and that our indoor environment is $10 \times 10 \times 4m^3$ large we would need a grid of size $200 \times 200 \times 80$ voxels. This gives us a total amount of 3200000 voxels and possible 3D coordinates. If we do not care about the colors in the room and just want to know if a node in the grid is occupied (1) or free (0) we just need to use a Boolean (1 byte) which gives a total of 3,2MB. If we want to use the color values received by the RGB camera, that is, if we want a colored map, we can use uint16_t (2 bytes) which gives a total of 6,4MB. Notice that the amount of 3D points in the discretizised room is independent of sample rate and sample time.

4.3 Discretization methods

Signed distance function

We will use a voxel grid to symbolize the signed distances we get from the SDF. We want to try to project each voxel in the voxel grid on to the image plane using,

$$I_d(i,j) = (\frac{f_x x}{z} + c_x, \frac{f_y y}{z} + c_y),$$

where f_x, f_y, c_x and c_y are intrinsic camera parameters (see Section 2.1).

First we have to transform the global voxel V_G into a local voxel V_L. This is done by multiplying the voxel with the inverse of the camera matrix that would transform the local 3D point to a global 3D point. If z in V_L is more than zero this means that the voxel is in front of the camera and we can try to project the vertex onto the image plane. If the vertex can be projected onto the image plane we calculate the signed distance along the local z axis between the local 3D point X_L and the local voxel V_L. Since the z value of the local 3D point $z = D(i,j)$, where $D(i,j)$ is the depth value in the pixel the local voxel were projected on, we get d_{SDF} = z_{V_L} - $D(i,j)$. If the signed distance is more than or equal to 0 it means that there is a surface on or between the global voxel and the camera, and therefore we set the value of the global voxel to 1. If the signed distance is less than 0 it mean that there are no surfaces on or between the camera and the global voxel, and therefore we set the global vertex to 0. Listing 4.1 shows the Matlab implementation of the algorithm.

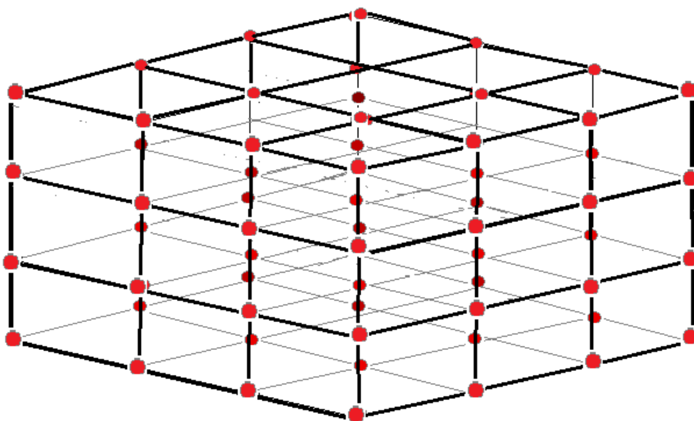

Figure 4.2 Example of how a voxel grid can look like where each red dot is a voxel.

```
for globalVoxel in voxelGrid
  //Use camera matrix to go from global to local coordinates.
  localVoxel = C*inv(globalVoxel);
  vx = localVoxel.x;
  vy = localVoxel.y;
  vz = localVoxel.z;
  if(vz >=0)
    /*project voxel grid on image plane */
    imageX = round(fx*vx/vz + cx);
    imageY = round(fy*vy/vz + cy);
    /*Check if index is inside image range */
    if(0< imageX && imageX <= xImageResolution && 0 < imageY &&
    imageY <= yImageResolution){

      /* Calculate singed distance by using distance between local
    voxel coordinates and local surface coordinates */
      signedDistance = vz - D(ly,lx);
      /* If sd is less than zero voxel is in front of the surface
    else surface is in front of voxel so voxel should be marked as
    occupied. */
      if(signedDistance >= 0){
        globalVoxel.occupied = 1;
      }else{
        globalVoxel.occupied = 0;
      }
    }
  }
}
```

Listing 4.1 Pseudocode for the SDF algorithm.

Box approximation

In our other approach we will use a box grid to symbolize if a space in the room is free or not. We want to try to project each pixel in the image into the box grid using,

$$X = (x, y, z) = (\frac{(i - c_x)z}{f_x}, \frac{(j - c_y)z}{f_y}, z)^T,$$

where $(i, j) \in I_d, z = I_d$ and f_x, f_y, c_x and c_y are intrinsic camera parameters (see Section 2.1).

First we have to be check if $D(i, j)$ *i.e.,* the local z value is greater than zero, this means that the local 3D coordinate X_L is located in front of the image plane. Then we transform the local 3D coordinate X_L into a global 3D coordinate X_G. This is done according to,

$$X_G = C \cdot X_L,$$

where X_L are the local coordinates and X_G are the global coordinates.

Then we have to calculate in which box the 3D coordinate is located. Since we know the resolution and the size of the space in all dimensions we also know the exact boundaries of each box so that the box grid can be easily indexed to find the correct box using the global 3D coordinates. Finally, we just need to mark the correct box as occupied. Listing 4.2 shows the Matlab implementation of the algorithm.

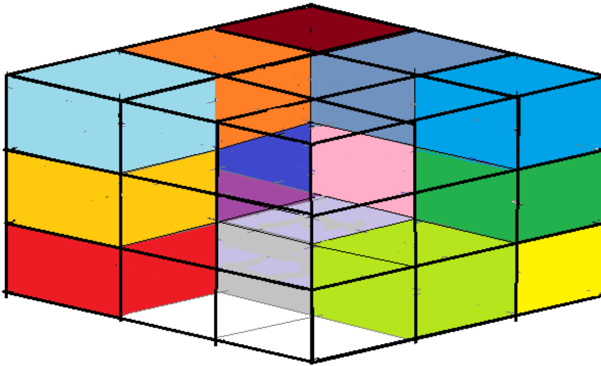

Figure 4.3 Example of a box grid where every filled box is occupied and every empty box is free.

```
for (pixelsX = 0; pixelsX < xImageResolution; pixelsX++){
  for (pixelsY = 0; pixelsY < yImageResolution; pixelsY++){
    if (D(pixelsXY,pixelsX) > 0){
      localPoint = getLocalCoordinates(pixelsX, pixelsX, D(pixelsY,
    pixelsX);
      /* Place coordinates in the world frame */
      globalPoint = C*localPoint;
      box = boxgrid(getBoxCoordinates(globalPoint)):
      box.occupied = 1;
    }
  }
}
```

Listing 4.2 Pseudocode for the box approximation algorithm.

5

Path planning

The reason for choosing D* Lite is that the algorithm is easy to overview and has proven to be faster than both Focused D* and A* on both terrain with random obstacles and on fractal terrain [S. Koening, 2005, Tables 1 and 2].

5.1 Implementation using the obstacle map

In this thesis we implemented the algorithm in 3 dimensions using an int*** matrix with ones and zeros received by the discretization block. In the implementation we used a $51 \times 51 \times 51$ matrix, in a room of $2m \times 2m \times 2m$ this will give a node resolution of $0.039m \times 0.039m \times 0.039m$. The matrix size and resolution should of course be fitted to available computational power and area of interest.

Every index in the matrix represents a global coordinate and is considered a node. If a node is blocked and untraversable it has the value 1 and if it is free and traversable it has the value 0. If a node is blocked there is no edges to or from that node. If a node is free if got edges to all surrounding free nodes, at most eight edges.

Since every node represents a global coordinate the weight of the edges are only effected by how many dimensions it covers, this is showed in Figure 5.1. The cost of an edge along one dimension is 1. The cost of traveling in two dimensions is the cost of traveling along the hypotenuse i.e $\sqrt{2} \approx 1.41$. Finally the cost of traveling in three dimensions is the euclidean distance between the nodes $\sqrt{3} \approx 1.73$. For simplicity reasons we choose to scale the cost and use the values 10 for one dimension, 14 for two dimensions and 17 for three dimensions. The implementation is made in C and uses a node object and a priority heap. The implementation of the node object can be seen in Linsting 5.1 and the implementation of the priority heap can be seen in Listing 5.2.

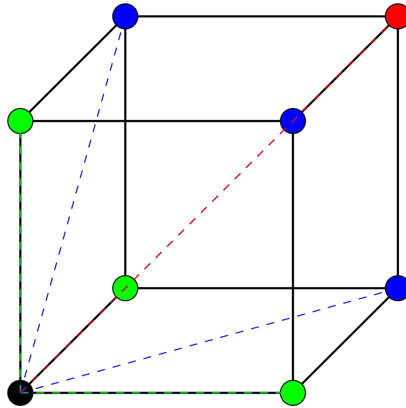

Figure 5.1 This figure shows the edge costs for traveling from the black node to its neighbors. The cost of traveling from black to green is 1, black to blue are $\sqrt{2}$ and from black to red are $\sqrt{3}$

```
typedef struct NODE {
    int row;
    int column;
    int depth;
    int heapIndex;
    int key[2];
    int rhs;
    int g;
} Node;
```

Listing 5.1 Node implementation

```
int heapSize();
int lessThan(int* key1, int* key2);
void insert(Node* node);
void removeNode(Node* node);
void percolateUp(Node* node, int hole);
void percolateDown(Node* node, int hole);
void percolateUpOrDown(Node* node, int hole);
Node* pop();
Node* top();
int* topKey();
```

Listing 5.2 Priority heap methods

5.2 Method

The method calls and code structure is described with psuedo code in Figure 5.2. Below we give a thorough description of the D* Lite algorithm.

Initialize The algoritm starts with an empty priority queue. The g value of all vertices are set to infinity and the *rhs* values of all vertices except the goal node are set to infinity. The rhs value of the goal node is set to zero and the goal node is inserted to the heap.

Compute Shortest Path The node with the highest priority is obtained from the heap. If the node is overconsistent we want to expand the node and the g value is set to the *rhs* value. The node is removed and the vertices of all of its predecessors are updated and added to the heap if they are inconsistent. If they are consistent and already in the heap they are removed. If the node is locally underconsistent the g value is set to infinity. The vertices of all of its predecessors are updated and added to the heap if they are inconsistent. If they are consistent and already in the heap they are removed. If a predecessor was affected by the change of the g value its *rhs* value is set to the smallest of its successors g value plus the cost for moving from s to the successor.

The iteration continuous until the *rhs* value of the current start node is smaller or equal to its g value or until the key of the start node is greater or equal to the key of the heap's top node. If the *rhs* value of the start node is infinity it means that there is no known path. Otherwise we move the robot to the successor of the start node which has the smallest g value plus the cost for moving from start to the successor. Finally the new start node is set to the robot's current position and the key modifier k_m is increased by the Manhattan distance between the new and old start node.

Before continuing we need to check if the cost of any edges has been changed. If there are no changed edge costs we move the robot to the successor of the start node which has the smallest g value + plus the cost for moving from start to the successor. Otherwise we need to update the *rhs* value of all edges that have been changed and may affect our current path.

There are two ways an edge can affect our planned path. The first is if an edge which earlier had high cost has changed to a lower cost $c_{old}(u,v) > c_{new}(u,v)$. In that case we set the *rhs* value of u to be the least of the current *rhs* value and the cost for moving from u to v plus the g value of v.

The second way the planned path can be affected is if the vertex with a changed cost is on our planned path $rhs(u) = c_{old}(u,v) + g(v)$. Then we set the *rhs* value of u to the *rhs* value of the successor of u which has the smallest g value + plus the cost for moving from u to the successor.

Finally, we update the vertices as described above and compute the shortest path once again. This is iterated until the robot has reached the goal node.

```
procedure CalcKey(s)
{01'''} return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))];
procedure Initialize()
{02'''} U = ∅;
{03'''} k_m = 0;
{04'''} for all s ∈ S rhs(s) = g(s) = ∞;
{05'''} rhs(s_goal) = 0;
{06'''} U.Insert(s_goal, [h(s_start, s_goal); 0]);
procedure UpdateVertex(u)
{07'''} if (g(u) ≠ rhs(u) AND u ∈ U) U.Update(u, CalcKey(u));
{08'''} else if (g(u) ≠ rhs(u) AND u ∉ U) U.Insert(u, CalcKey(u));
{09'''} else if (g(u) = rhs(u) AND u ∈ U) U.Remove(u);
procedure ComputeShortestPath()
{10'''} while (U.TopKey() <̇ CalcKey(s_start) OR rhs(s_start) > g(s_start))
{11'''}    u = U.Top();
{12'''}    k_old = U.TopKey();
{13'''}    k_new = CalcKey(u));
{14'''}    if(k_old <̇ k_new)
{15'''}       U.Update(u, k_new);
{16'''}    else if (g(u) > rhs(u))
{17'''}       g(u) = rhs(u);
{18'''}       U.Remove(u);
{19'''}       for all s ∈ Pred(u)
{20'''}          rhs(s) = min(rhs(s), c(s, u) + g(u));
{21'''}          UpdateVertex(s);
{22'''}    else
{23'''}       g_old = g(u);
{24'''}       g(u) = ∞;
{25'''}       for all s ∈ Pred(u) ∪ {u}
{26'''}          if (rhs(s) = c(s, u) + g_old)
{27'''}             if (s ≠ s_goal) rhs(s) = min_{s' ∈ Succ(s)}(c(s, s') + g(s'));
{28'''}          UpdateVertex(s);
procedure Main()
{29'''} s_last = s_start;
{30'''} Initialize();
{31'''} ComputeShortestPath();
{32'''} while (s_start ≠ s_goal)
{33'''}    /* if (rhs(s_start) = ∞) then there is no known path */
{34'''}    s_start = arg min_{s' ∈ Succ(s_start)}(c(s_start, s') + g(s'));
{35'''}    Move to s_start;
{36'''}    Scan graph for changed edge costs;
{37'''}    if any edge costs changed
{38'''}       k_m = k_m + h(s_last, s_start);
{39'''}       s_last = s_start;
{40'''}       for all directed edges (u, v) with changed edge costs
{41'''}          c_old = c(u, v);
{42'''}          Update the edge cost c(u, v);
{43'''}          if (c_old > c(u, v))
{44'''}             rhs(u) = min(rhs(u), c(u, v) + g(v));
{45'''}          else if (rhs(u) = c_old + g(v))
{46'''}             if (u ≠ s_goal) rhs(u) = min_{s' ∈ Succ(u)}(c(u, s') + g(s'));
{47'''}          UpdateVertex(u);
{48'''}       ComputeShortestPath();
```

Figure 5.2 D* lite pseudo code presented in *Fast re-planning for navigation in unknown terrain* [S. Koening, 2005]

6

Gantry-Tau robot

6.1 Previous work

The three degree-of-freedom (DOF) Gantry-Tau parallel kinematic robot has been kinematically and dynamically modeled [Dressler, 2012].

6.2 Overview

The Asus Xtion Pro Live RGB-D camera was mounted on the Gantry-Tau robot and connected to a stationary computer running the Simulink model. A new labcomm module, making the communication between our stationary computer and the Gantry-Tau robot possible, was implemented and wrapped to work with our Simulink model. When the model enters the *moving phase* it sends the global coordinates (x, y, z) relative to the initial position along with the four quaternion elements $q1, q2, q3, q4$ to the robot using the labcomm module. The quaternion elements will for simplicity never be changed letting the robot keep the same rotation matrix along the experiment.

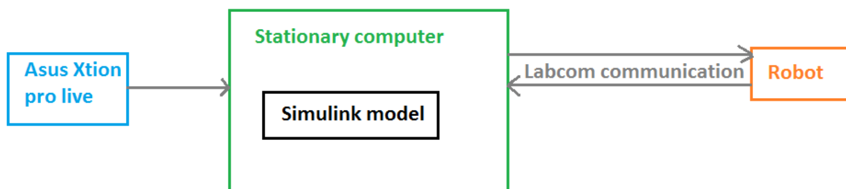

Figure 6.1 Simulink to robot communication flow

The Gantry Tau structure was invented by Torgny Broghårdh and developed in the SMErobot [*SMErobot* 2009] and MONROE [*MONROE* 2012] projects. In 2000 a new family of parallel kinematic 3 DOF robots developed at ABB was presented by Broghårdh where the six carbon fiber links where clustered in a 3-2-1 configuration [Dressler, 2012]. The robot has three prismatic joints implemented as carts on linear guide ways connected to the end-effector plate [Dressler, 2012].

Figure 6.2 L2 Gantry Tao Robot with mounted Xtion Asus Pro Live RGB-D camera.

6.3 Camera-Robot transformation

Since the coordinate system of the robot (world frame) and the coordinate system of the camera (camera frame) are known it should be possible for us to find a transformation from the camera to the robot that is good enough for our purpose.

The camera is tilted downwards as can be seen in Figure A.1. To find the tilting angle α we simply located the image center frame in the world frame by manually looking at the image and then the distance from the camera to the center point was measured along the world frames x axis and z axis (see Figure 6.3).

Then α was calculated as

$$\alpha = \arctan(\frac{a}{b})$$

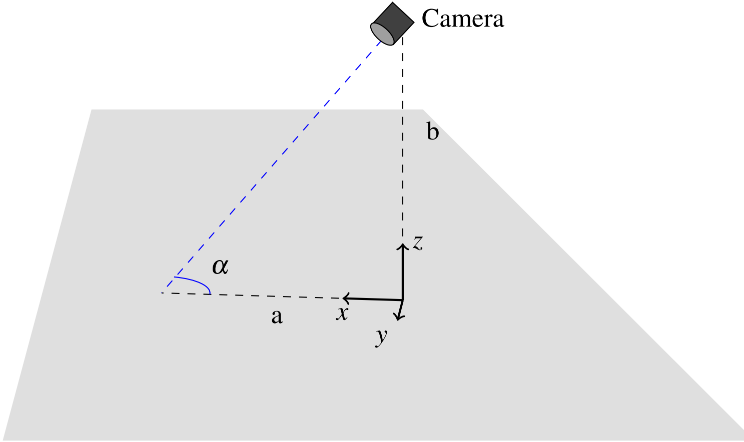

Figure 6.3 Simplified figure of the cameras downward tilt α

It might be hard to measure α with good precision. We solved this problem by using the measured α as an initial guess and plot the global coordinates of some images. The camera will always see a part of the floor and since we know that the floor should be about parallel to the robot's x, y plane we could manually tune α until the floor's global coordinates were parallel to the global x, y plane. The result can be seen in Figure 6.4.

In this image a box was placed on the floor. The red point cloud shows the downwards tilted camera which has not been compensated for. Here we measured our initial guess to be 58° which seems to be quite close to the truth. Finally we decided to go with $\alpha = 62^\circ$.

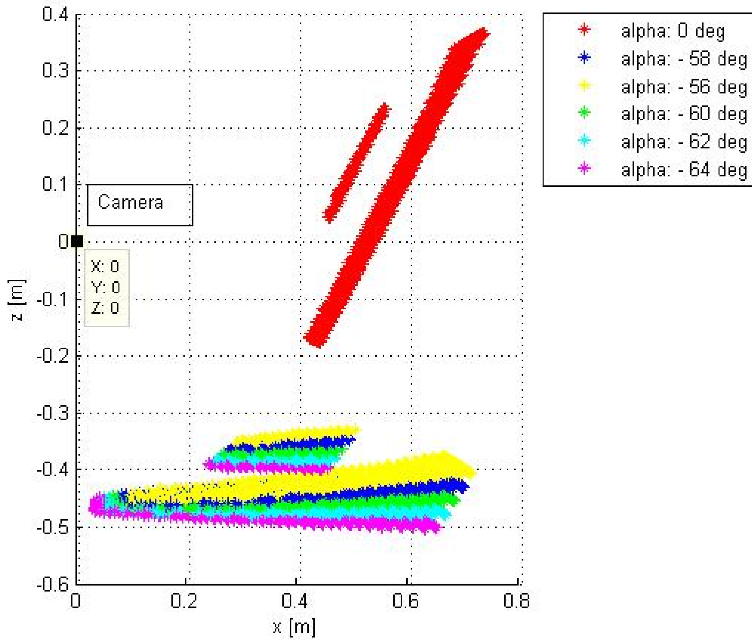

Figure 6.4 Global points for different alpha

Rotation

The robot will only be ran in Cartesian mode so the tool oriantation will not change in the global frame. To simplify our transformation matrix we lock the robot frame to the global frame, this way we do not have to consider any change of rotation in our transformation matrix. That means that once we calculated α the rotation matrix can easily be calculated as can be seen in Figure 6.5. We want to calculate the rotation matrix from the camera to the robot.

Frame 1 -> frame 2 First we want to un-tilt the camera by rotating $-\alpha$ around the x axis.

$$R_x(-\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-\alpha) & -\sin(-\alpha) \\ 0 & \sin(-\alpha) & \cos(-\alpha) \end{bmatrix}$$

Frame 2 -> frame 3 Then we want to rotate another $-\pi$ around the x axis to align the camera z axis with the world frame z axis.

$$R_x(-\pi/2) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}$$

Frame 3 -> frame 4 Finally we want to rotate $-\pi$ around the z axis to align the camera frame's x and y axes with the world frame's x and y axes.

$$R_z(-\pi/2) = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

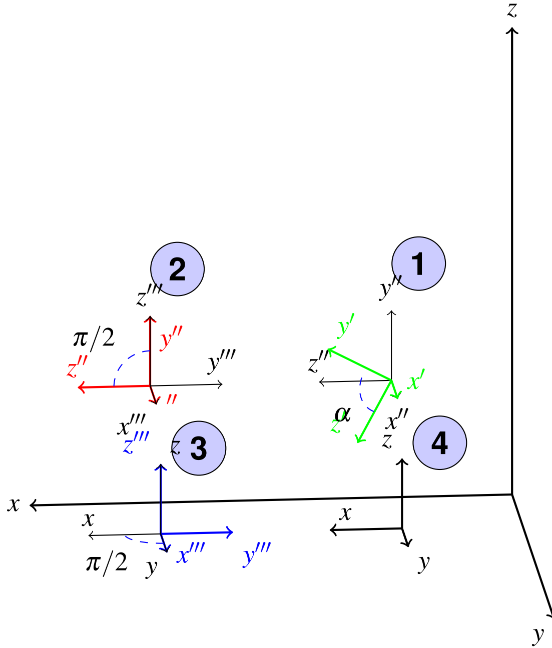

Figure 6.5 A modulation of the rotations between the camera and robot frame

The total rotation from the camera frame to the world frame is

$$R_{c2w} = R_z(-\pi/2) \cdot R_x(-\pi/2) \cdot R_x(-\alpha) =$$

$$\begin{bmatrix} 0 & \sin(\alpha) & \cos(\alpha) \\ -1 & 0 & 0 \\ 0 & -\cos(\alpha) & \sin(\alpha) \end{bmatrix} = \begin{bmatrix} 0 & -0.8829 & 0.4695 \\ -1 & 0 & 0 \\ 0 & -0.4695 & -0.8829 \end{bmatrix}$$

Translation

When the rotation is known we just need to find the translation between the camera and the robot along the camera axes (see Figure 6.6). We can measure the distance ΔZ_r and ΔX_r between the robot and the camera. This is used to calculate the euclidean distance e between the robot and the camera. Since we know the tilting angle of the camera α and can calculate β we can calculate the angle γ between the robot and the camera. Finally ΔY_c and ΔZ_c can be calculated using trigonometrical functions and the distance e. Since there is no offset along the camera's x axis we realize that $\Delta X_c = 0$. The trigonometry can be seen in Figure 6.6.

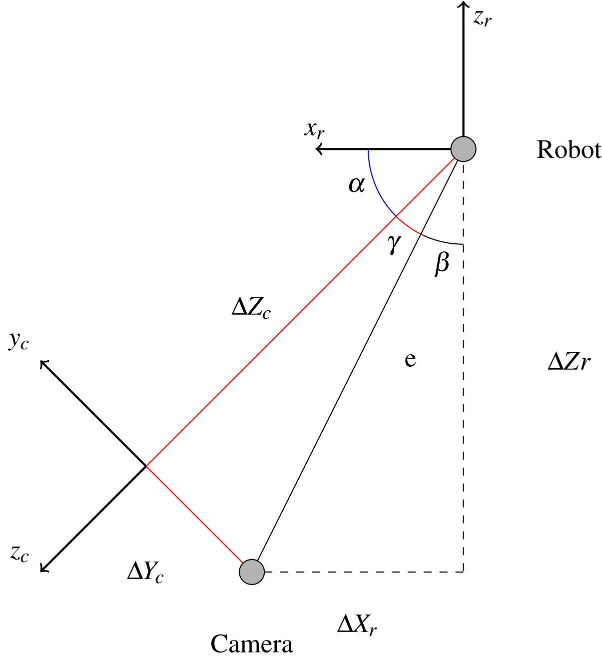

Figure 6.6 Simplified figure showing translation between camera and robot.

$$e = \sqrt{(\Delta Z_r^2 + \Delta X_r^2)}$$

$$\beta = \arctan(\frac{\Delta X_r}{\Delta Z_r})$$

$$\gamma = \pi/2 - \alpha - \beta$$

$$\Delta Y_c = sin(\gamma)e$$

$$\Delta Z_c = cos(\gamma)e$$

Measurements gave:

$$\Delta Z_r = 0.2m$$
$$\Delta X_r = 0.1m$$
$$e = 0.2236m$$
$$\beta = 26.56^\circ$$
$$\gamma = 1.44^\circ$$
$$\Delta Y_c = 0.0056m$$
$$\Delta Z_c = 0.2235m$$

We get $t_{r2c} = \begin{bmatrix}\Delta X_c & -\Delta Y_c & -\Delta Z_c\end{bmatrix}' = \begin{bmatrix}0 & -0.0056 & -0.2235\end{bmatrix}'$m and $t_{c2r} = \begin{bmatrix}\Delta X_r & -\Delta Y_r & -\Delta Z_r\end{bmatrix}' = \begin{bmatrix}-0.1 & 0 & -0.2\end{bmatrix}'$m.

Transformation

The final homogenous transformation H_{c2r} from the camera to the robot is then

$$H_{c2r} = \begin{bmatrix} R_{c2r} & t_{c2r} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} 0 & \sin(\alpha) & \cos(\alpha) & \Delta x_r \\ -1 & 0 & 0 & \Delta y_r \\ 0 & -\cos(\alpha) & \sin(\alpha) & \Delta z_r \\ 0 & 0 & 0 & 1 \end{bmatrix} =$$
$$\begin{bmatrix} 0 & -0.8829 & 0.4695 & 0.1000 \\ -1 & 0 & 0 & 0 \\ 0 & -0.4695 & -0.8829 & -0.2000 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If H_{c2r} is the homogenous transformation that transforms the camera position C to the robot position Q the inverse of H_{c2r} is the homogenous transformation that transforms the robot position Q to the camera position C.

$$H_{r2c} = H_{c2r}^{-1} = \begin{bmatrix} 0 & -1 & 0 & \Delta x_c \\ \sin(\alpha) & 0 & -\cos(\alpha) & \Delta y_c \\ \cos(\alpha) & 0 & \sin(\alpha) & \Delta z_c \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can use this to get the first camera position C_1 if we know the initial robot position Q_1,

$$C_1 = H_{C2R} \cdot Q_1.$$

Since we assume that the robot orientation is fixed we get,

$$Q_1 = \begin{bmatrix} 1 & 0 & 0 & tx_c^r \\ 0 & 1 & 0 & ty_c^r \\ 0 & 0 & 1 & tz_c^r \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where tx_c^r, ty_c^r and ty_c^r are the robot position in the camera frame. They can be calculated by transforming the robot frame's robot position to the camera frame using,

$$\begin{bmatrix} & R_{r2c} & & \begin{matrix} tx_c^r \\ ty_c^r \\ tz_c^r \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} & R_{r2c} & & \begin{matrix} 0 \\ 0 \\ 0 \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & tx_r^r \\ 0 & 1 & 0 & ty_r^r \\ 0 & 0 & 1 & tz_r^r \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

where tx_r^r, ty_r^r and ty_r^r represent the robot frame's robot position.

6.4 Experimental setup

Timing

In order to get the Simulink simulation to run in real time we used the realtimer block developed by the Department of Automatic Control, LTH. The realtimer synchronizes the simulation time with the system clock making the simulation run with a controllable sampling time. For safety reasons an additional control was implemented on both the model and the robot side. The model uses an assertion triggered every sample that pauses the execution after the moving coordinates have been sent to the robot. The execution is then continued when the correct behavior of the robot is verified and the operator presses enter in the Matlab terminal. At the robot side the user needs to acknowledge the new coordinates before the robot starts to move there.

Labcomm

Labcomm is a binary protocol developed at LTH. Labcomm only requires one way communication and keeps the communication to a minimum. It consists of a protocol specification and a compiler that generates C or Java code for needed methods such as encode/decode [*Labcomm* 2014].

The Labcomm module sets up a TCP IP socket to the robot computer. The stationary computer running the Simulink model is server and the robot is the client. After the server is started the client can connect to the server. The Labcomm module also initializes and registers a reader and a writer that will handle the stream input/output on the server side.

The method used by the server side in the moving phase is

```
float setTarget(float x, float y, float z, float q1, float q2, float
    q3, float q4)
```

which encodes and sends the robot target and then calls the decoder that locks execution until an acknowledgement is sent from the client application. The robot target used in this experiment can be seen in Listing 6.1.

```
#ifndef PREDEFINED_LCRobot_robtarget
typedef struct {
  struct {
    float x;
    float y;
    float z;
  } trans;
  struct {
    float q1;
    float q2;
    float q3;
    float q4;
  } rot;
  int32_t status;
} LCRobot_robtarget;
#endif
```

Listing 6.1 The robot target used in the experiement contaning needed data

The client code that controls the robot is written in ABB's robot programming language RAPID [Berlin, 2012].

Experiment

In this experiment we test the system's overall performance as well as the path planning implementation. The Gantry-Tau robot was set into a well suited starting position with a goal position in front of it. Then an obstacle was placed between the Gantry-Tau robot and the goal position. The system starts with an empty obstacle map. After the first iteration the obstacle map starts filling with what the camera sees. After some iterations when the Gantry-Tau robot approaches the obstacle it appears on the camera and is marked in the obstacle map. The obstacle will then be located on the planned path and the algorithm is forced to re-plan and travel around the obstacle. The result of this experiment can be seen in Section 9.5.

7

Combine hexacopter

7.1 Previous work

This thesis is the third in a series of thesis done on Combine Control Systems in cooperation with the Department of Automatic Control, LTH, concerning the Combine hexacopter.

Model-Based Approach to Computer Vision and Automatic Control using Matlab Simulink for an Autonomous Indoor Multirotor System

In the first thesis by Niklas Ohlsson and Martin Ståhl [Niklas Ohlsson, 2013] the main focus was to assemble hardware and implement computer vision algorithms, position control and some autonomous behavior using a model based approach to get a working UAV fit for maneuvering in GPS-denied environments such as indoor environments. A Pandaboard [*OMAP TM 4 PandaBoard System Reference Manual* 2010], which is a portable software platform not very different from the popular Raspberry Pi [*Rasberry Pi*], was attached to the hexacopter. An Ethernet communication protocol was created to make communication between the Pandaboard and the hexacopter autopilot possible. Finally a Matlab Simulink model was developed controlling the hexacopter behavior. At the end of their thesis the UAV still had some stability problems left, due to the nature of the chosen computer vision algorithm (*template matching*), bad altitude hold and some lag probably caused by Simulink.

Autonomous navigation and control of a hexacopter in an indoor environment

The second thesis was performed by Johan Fogelberg in 2013 [Fogelberg, 2013]. The main objective of the thesis on the UAV was to improve the altitude control for the UAV with hope of thereby improving the performance of the computer vision algoritm, to improve horizontal control and thereby reduce the drif and, finally, use

nonlinear filtering for sensor fusion of IMU data and computer vision measurements to improve position and velocity estimates.

The system was modeled and various nonlinear filters were used in simulation according to the rules of model based approach. Finally, the Square root unscented Kalman filter was implemented fusing all useful data together and added together with the previous Matlab Simulink model. A simple PID controller was implemented with anti-windup, derivative filtering, and set point scaling. Even though this thesis made huge improvements on the stability of the UAV the estimation is only able to keep the UAV stable in a hovering state and is unable to follow a trajectory.

7.2 Hardware assembly

Pandaboard Hardware

The Pandaboard features a Dual-core ARM Cortex-A9 processor [*OMAP TM 4 PandaBoard System Reference Manual* 2010]. It uses the hexacopter's 12.6V LiPo battery and an external 5V switching regulator for power supply [Niklas Ohlsson, 2013]. The pandaboard is supported by native Simulink code generation and runs a MathWorks supplied Linux distribution on a SD memory card. It has two USB connections that can be used for a regular or RGB-D camera, an Ethernet port, and a Wi-Fi connection that makes external computer communication possible.

APM Autopilot

The ArduPilot Mega 2.5+ features an ATMEGA2560 micro processor as well as data storage, programming logics. An inertial measurement unit (IMU) with a 3-axis gyro, 3-axis accelerometer and magnetometer for attitude and yaw estimation, and a barometer and external sonar sensor are used for altitude measurements.

ArduCopter Software

The hexacopter features the open source autopilot software ArduCopter [*ArduCopter*]. The ArduCopter is responsible for reading sensors, receiving RC radio input as well as stabilizing the hexacopter during flight.

7.3 Tracking

Since the initial computer vision algorithm implemented on the UAV did not work sufficiently well when it came to position estimation and keeping the UAV stable a new tracking approach was needed. The angle estimates of the UAV done in previous thesis work comparably well so our idea was to use the known rotation together with a tracking algoritm.

Rotation fused ICP

In this approach we want to fuse the rotation matrix achieved by the traditional ICP algoritm with the estimated rotation matrix,

$$R = AR_{estimate} + (I - A)R_{ICP}$$

, where A is a weighting coefficient matrix $A = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & \alpha \end{pmatrix}$ and α is a weighting coefficient $0 \leq \alpha \leq 1$.

The weighting coefficient is used according to how much we trust the estimated rotation matrix in relation to the ICP rotation matrix. For example, if we are sure that the estimated rotation matrix is 100% correct we will use $\alpha = 1$.

Since the ICP algoritm has been implemented many times before we choose to use a C++ implementation called LIBICP by Dr. Andreas Geiger, research scientist at the Perceiving Systems department, Max Planck Institute for Intelligent Systems, Tübinge. [Geiger, 2008]. A related publication where the software was used is [Geiger et al., 2012]. The implementation allows use of both point-to-point and point-to-plane error metrics and uses k-d tree to speed up the matching process.

Translation approximation

The main idea of translation approximation is to use the known rotation matrix and only calculate the relative translation. The relative translation is approximated as the relative translation between two point clouds centroids. Notice that since we can only calculate the relative translation, the centroids are calculated from local point clouds.

A centroid P is calculated as,

$$P = \frac{1}{M} \sum_{i=0}^{n-1} m_i X_i,$$

where $X = (x, y, z)$, M is the total weight and m_i is the weight at X_i.

Since we decided to use the weight $w_i = 1$ the centroid is simply,

$$P = \frac{1}{n} \sum_{i=0}^{n-1} X_i,$$

i.e the mean value.

Then the translation is calculated as,

$$\Delta t = \Delta R^{-1} P_{k+1} - P_k,$$

where P_k is the local point cloud generated by the old frame and P_{k+1} is the point cloud generated by the new frame. We can now calculate the absolute translation using,

$$t_{k+1} = t_k + \Delta t,$$

the new camera matrix will be

$$C_{n+1} = \begin{bmatrix} R_{n+1} & t_{n+1} \\ 000 & 1 \end{bmatrix} C_n$$

The greatest advantage of using translation approximation is the linear complexity $\mathcal{O}(n)$ making the processing time extremely low.

7.4 Experimental setup

For measuring ground truth performance tests we have been using the automatic online evaluation tool for evaluation of tracking methods supplied by the TUM computer vision group [*Submission form for automatic evaluation of RGB-D SLAM results* 2009-2013]. As ground truth we used the *freiburg1_teddy* sequence [*Freiburg1 Teddy sequence* 2011]. In Table 7.1 some basic information about the test sequence is shown.

Table 7.1 Freiburg1 Teddy sequence [*Freiburg1 Teddy sequence* 2011]

Duration:	50.82s
Duration with ground-truth:	50.78s
Ground-truth trajectory length:	15.709m
Avg. translational velocity:	0.315m/s
Avg. angular velocity:	21.320°/s
Trajectory dim.:	2.42m x 2.24m x 1.43m

8

Software implementation

The software was mainly implemented in a Matlab/Simulink environment and some additional modules was done in C/C++. In the Combine hexacopter experiment we used the Simulink *generate to target platform* feature which generates C code to the chosen target hardware, in our case the Pandaboard. In the Gantry-Tau experiment we used the same code base which were slightly modified omitting the tracking algorithm and run locally on the stationary computer used.

8.1 Simulink - C/C++ interface

One of the main problems when it came to software implementation is that Mathworks does not support use for Asus Xtion Pro Live or the Microsoft Kinect. Therefore we needed to use an additional module for receiving the depth images from the RGB-D camera. Initially we tried using the open source *libfreenect* interface developed by openKinect but later decided to use OpenNI2 instead due to the poor status of the libfreenect interface. OpenNI2 is an C++ open source SDK used for 3D sensing applications [*OpenNI 2 SDK*].

Matlab/Simulink supports use of external C/C++ code using S-functions. Unfortunately if does not support code generation to C from C++ S-functions calls, therefore an additional C wrapper solution is needed for code generation to the target platform.

An illustration of the code wrapping can be seen in Figure 8.1. First an .so library file needs to be generated on the target platform and moved into a folder on the host computer reachable for the Simulink model. The same .so file also needs to be installed into the targets */usr/lib* folder and the used .h header files should be copied to the targets */usr/include* folder. This is so that the files and libraries can be reached from another map then the one where they where built. Then we need a C wrapper file on the host computer which interfaces the C++ methods we want to use from our .so library. After creating the C wrapper file we use Matlab's build in legacy code commands to create an S-function C file, a .tlc file (Target Language Compiler), a rtw makefile (makes it possible to use dependent source and

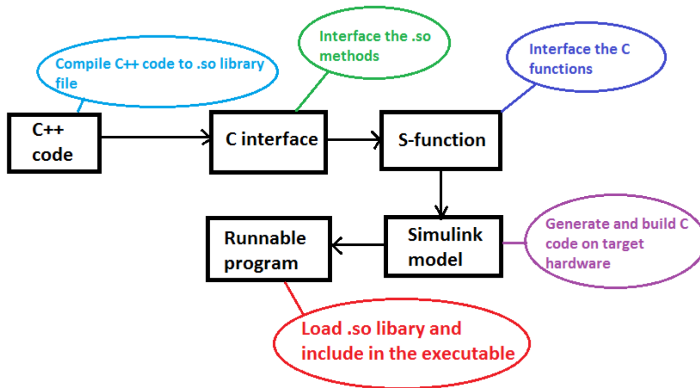

Figure 8.1 Illustration of the code wrapping process

header files in another folder than the S-function where they where generated) and a Simulink block in Matlab. Finally we compile the S-function C file into a mexfile using legacy code commands. Below is an example on how we used Matlab legacy code to generate a Simulink block for the OpenNI2 interface.

```
def = legacy_code('initialize');
def.SFunctionName = 'asus_sfcn'; % S-function name
% Polled function with specified output dimensions
def.OutputFcnSpec = 'asusCapture(uint16 y1[240][320])';
def.StartFcnSpec = 'asusInit()'; % Constructor
def.TerminateFcnSpec = 'asusTerminate()'; % Deconstructor
def.HeaderFiles = {'asus_capture.h'};
def.IncPaths = {'mypath\include'};
def.SampleTime = 'parameterized';
def.Options.isMacro = true;
def.Options.useTlcWithAccel = false;
legacy_code('sfcn_cmex_generate', def); % S-function
legacy_code('compile', def); % Compile
def.HeaderFiles = {'asus_capture.h' 'NI.h' 'wrapper.h'};
def.SourceFiles = {'wrapper.c'}; % Our C wrapper file
def.IncPaths = {'mypath\include'};
def.SrcPaths = {'mypath\src'};
def.TargetLibFiles = {'libNI.so'}; % Our .so file
legacy_code('sfcn_tlc_generate', def); %Tlc file
legacy_code('rtwmakecfg_generate', def);  %rtw makefile
legacy_code('slblock_generate', def) % Simulink block
\label{matlab_wrapper}
```

LIBICP, the ICP C++ implementation we choose to use, can be wrapped in the same way as OpenNI2. The files asus_capture.h, NI.h, wrapper.h and wrapper.c can be seen in Appendix A.

8.2 Simulink model

Due to to the lack of performance of the investigated tracking algorithms (see Chapter 9.3) the tracking was omitted in the final software implementation. Instead the internal positioning of the Gantry-Tau robot where used by using the position demanded by the path planning algorithm as translation in the camera matrix. Since we know that the Gantry-Tau robot will in fact move to the commanded positioning we do not need the positioning feedback from the tracking algorithm and our system can still be used even though tracking is omitted. Note that it was not possible to run the entire system on the Combine hexacopter due to the tracking algorithm's lack of performance and the limited processor power on the Pandaboard. The Combine hexacopter's internal positioning that was intended to be used in this thesis did not work sufficiently [Fogelberg, 2013]. The Simulink model for the main functions can be seen in Figure 8.2.

CaptureNI This block uses the openNI implementation and wrappers to receive images from the Asus Xtion Pro Live camera, see Figure 8.3.

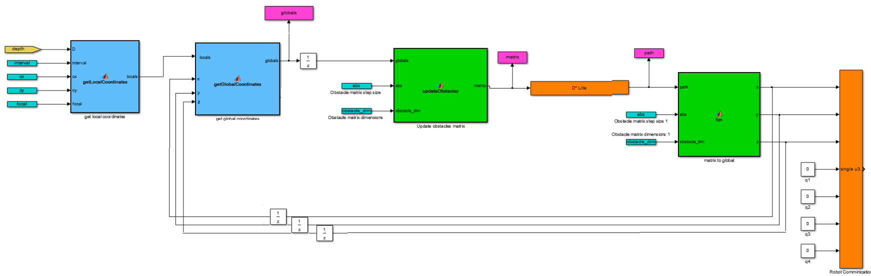

Figure 8.2 Simulink model showing the algorithm blocks.

Flip dimensions The initial image is mirrored so we flip it back to simplify further calculations.

Get local coordinates Transforms the depth data into local coordinates

Get global coordinates Transforms the local coordinates into global coordinates using the robot-camera transformation and the current camera position which is the previous coordinates given by the path planning algorithm. Since we currently don't have a need for changing the camera rotation we simply use the identity matrix.

Update obstacle matrix Performs box approximation to discretizise the global coordinates.

Step Runs D* Lite path planning algorithm where start and goal position are given as block arguments on index form and the obstacle matrix is an input parameter and next step indexes as out parameter.

Matrix to global Calculates the next step indexes into global coordinates.

Robot communication This block uses the Labcomm implementation and wrappers and sends the target coordinates to the robot.

Timing block This group makes the simulation run in real time and asserts a pause every h second, where h is the sample time. The assertion then needs to be acknowledged in the terminal to re-assume simulation, this is a safety precaution. The model can be seen in Figure 8.4.

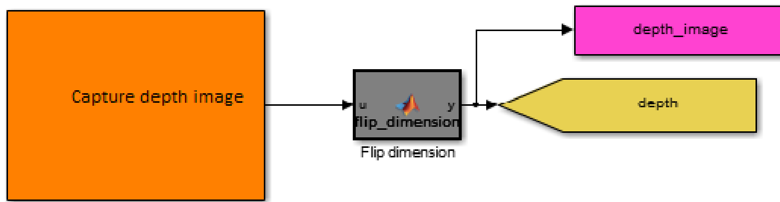

Figure 8.3 Simulink model showing the capture block.

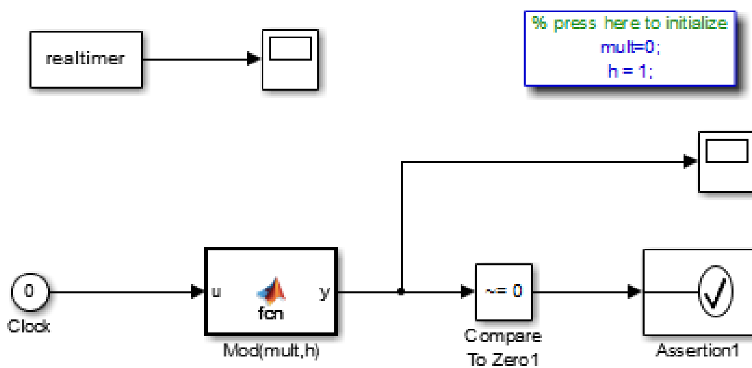

Figure 8.4 Simulink model showing the timing setup.

9

Result

This section presents the results of this thesis. First in Section 9.1 the time performance of the Asus Xtion Pro Live with the OpenNI2 C++ software implementation is presented. In Section 9.2 the measured CPU time over 1000 iterations are presented in a table and a plot for SDF and Box Approximation using every pixel and box approximation using every fifth pixel.

Then in Section 9.3 the results of *Tracking* are discussed. It includes results of the different tracking approaches including rotation fused ICP (Point-to-Point and Point-to-Plane) and translation approximation. Tables and plots of the measured CPU time for 1000 iterations show the time performance of the different algorithms. Performance on ground truth data sets are presented in an analogous way. Section 9.4 shows the result of the *Camera-Robot transformation* where the global coordinates of different camera positions are shown in a plot.

Finally, the results of the path planning algorithm implemented on the Gantry-Tau robot are shown in Section 9.5.

9.1 Image retrieval

For timing reasons we are interested in knowing the CPU execution time of the Asus Xtion Pro Live OpenNI2 C++ implementation. In Table 9.1 and Figure 9.1 we see the results of 1000 executions on the Pandaboard when a frame rate of 30fps and a resolution of 320×240 pixels were demanded.

OpenNI2	CPU time / s
Mean	0.0279
Median	0.0300
Max	0.0700
Min	<0.01
Std	0.0108

Table 9.1 CPU time for Asus Xtion Pro Live OpenNI2 C++ implementation with demanded framerate 30fps for 1000 executions.

The results in Table 9.1 tells us that the mean frame rate is 35.84 Hz, median frame rate is 33.33 Hz and that the minimum frame rate achieved in this test is 14.29 Hz.

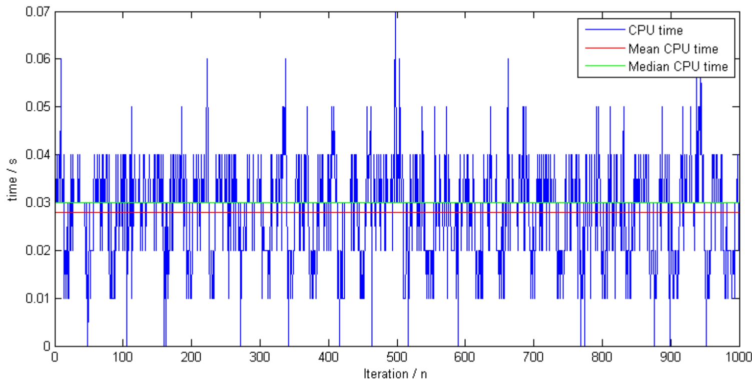

Figure 9.1 CPU time for Asus Xtion Pro Live OpenNI2 C++ implementation with demanded frame rate 30fps for 1000 executions.

9.2 Updating the Obstacle map

Time measurements

In Figure 9.2 we see the execution time of discretization using SDF, box approximation using every pixel and box approximation using every fifth row and every fifth column (every 25th pixel). Notice that this is not CPU time measured on the Pandaboard and can only be used for relative time performance between the different discretization algorithms. The time measurements are done on the *freiburg1_teddy* sequence [*Freiburg1 Teddy sequence* 2011]. The statistics are presented in Table 9.2.

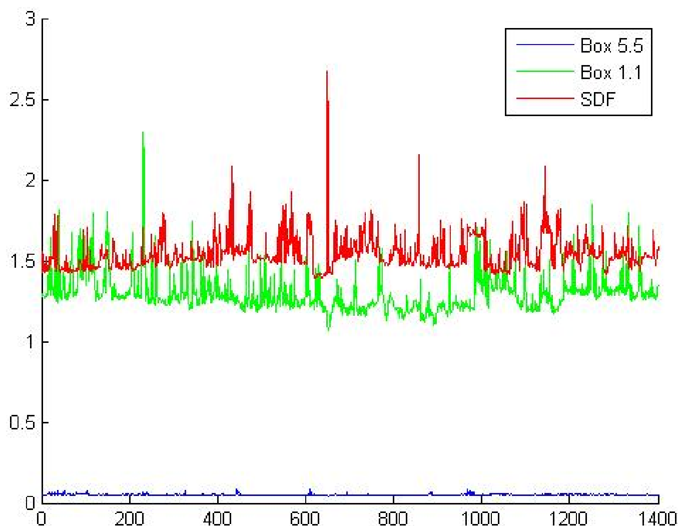

Figure 9.2 Executions time for discretization using SDF, box approximation using every pixel and box approximation using every fifth pixel over 1400 samples.

CPU time	SDF / s	Box 1.1 / s	Box 5.5 /s
Mean	1.51	1.307	5.29×10^{-2}
Median	1.50	1.305	5.28×10^{-2}
Max	2.28	2.002	8.40×10^{-2}
Min	1.39	1.127	4.53×10^{-2}
Std	8.92×10^{-2}	6.78×10^{-2}	0.30×10^{-2}

Table 9.2 Executions time for discretization using SDF, box approximation using every pixel and box approximation using every fifth pixel over 1000 samples.

Performance on ground truth data sets

In Figure 9.3 we see the discretisized version of the environment and a visual comparison between the box approximation and SDF method.

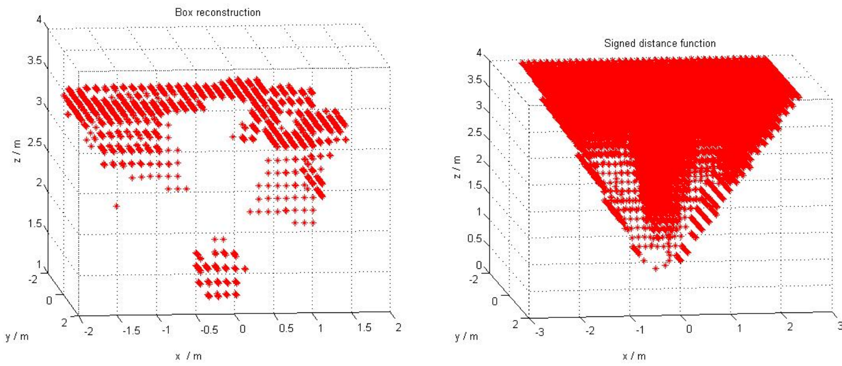

Figure 9.3 This figure shows the discretization of the *freiburg1_teddy* sequences first depth image. The box approximation approach can be seen to the left and the SDF approach can be seen to the right.

9.3 Combine hexacopter

Tracking

Rotation fused ICP

For time measurements we have measured the CPU times used by the implementation of interest while running on the Pandaboard. This was done by using the C method clock() as follows:

```
begin = clock();
/*Do ICP*/
end = clock();
time = (double)(end - begin) / CLOCKS_PER_SEC);
```

Time measurements For time measurements we have measured the CPU time for the original ICP implementation. Notice that this is the fused version. We see the time meassurements for Point-to-Point ICP in Figure 9.4 and for Point-to-Plane in Figure 9.5. The corresponding statistics are shown in Table 9.3.

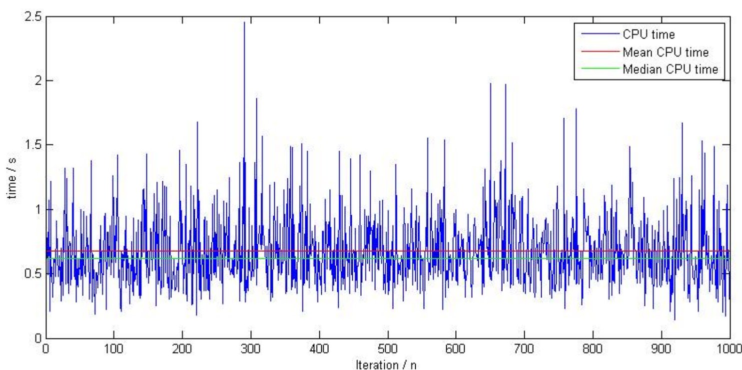

Figure 9.4 CPU time using fused ICP point to point C++ implementation with $\alpha = 1$ for 1000 executions.

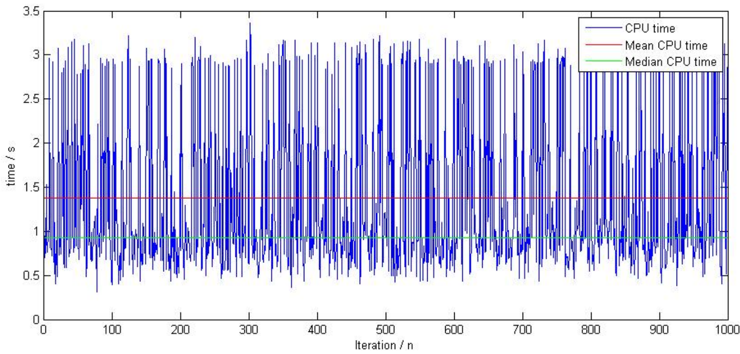

Figure 9.5 CPU time using fused ICP point to plane C++ implementation with $\alpha = 1$ for 1000 executions.

ICP	Point to Point / s	Point to Plane / s
Mean	0.674	1.38
Median	0.615	0.930
Max	2.45	3.36
Min	0.140	0.310
Std	0.288	0.934

Table 9.3 CPU time using fused ICP C++ implementation with $\alpha = 1$ for 1000 executions.

Performance on ground truth data sets For performance measurements we used the *freiburg1_teddy* sequence [*Freiburg1 Teddy sequence* 2011] as ground truth. We see the ground truth performance for Point-to-Point ICP in Figure 9.6 and for Point-to-Plane in Figure 9.7. The corresponding statistics are shown in Table 9.4.

ICP	Point to Point / m	Point to Plane / m
Rmse	0.751	0.422
Mean	0.659	0.380
Median	0.687	0.351
Std	0.360	0.183
Min	0.0430	0.0473
Max	1.38	0.755

Table 9.4 Absolute translation error for fused ICP C++ implementation with $\alpha = 1$ for 1400 samples.

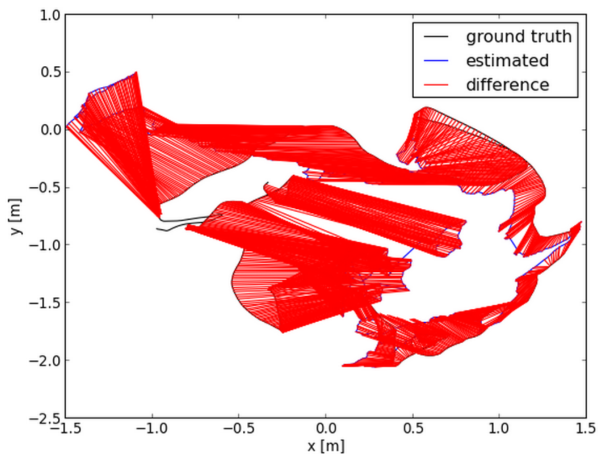

Figure 9.6 Absolute translation error using fused ICP point-to-point C++ implementation with $\alpha = 1$ for 1400 samples.

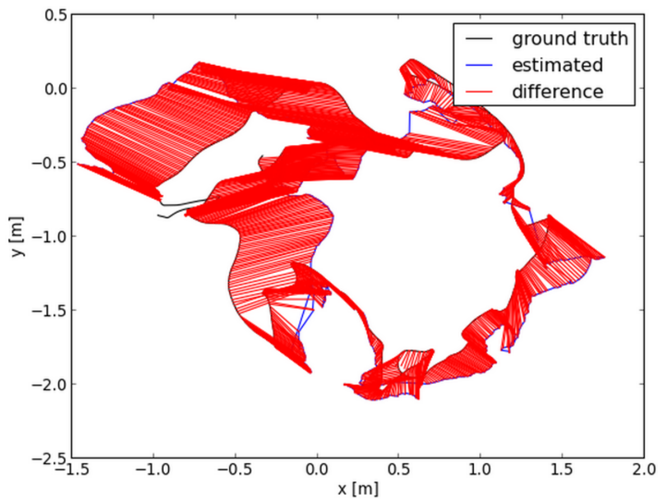

Figure 9.7 Absolute translation error using fused ICP point-to-plane C++ implementation with $\alpha = 1$ for 1400 samples.

Translation approximation

Time measurements For time measurements we have measured the times used by the implementation of interest while running on a Asus laptop. The laptop were running 64-bit Windows 7 Proffesional N featuring a Intel(R) Core(TM) i3-2310M CPU @2.10GHz processor and 8GB RAM.

This was done by using the tic toc Matlab method as follows:

```
tic
/*Do TA*/
time = toc;
```

The time measurements can be seen in Figure 9.8 and the statistics are presented in Table 9.5.

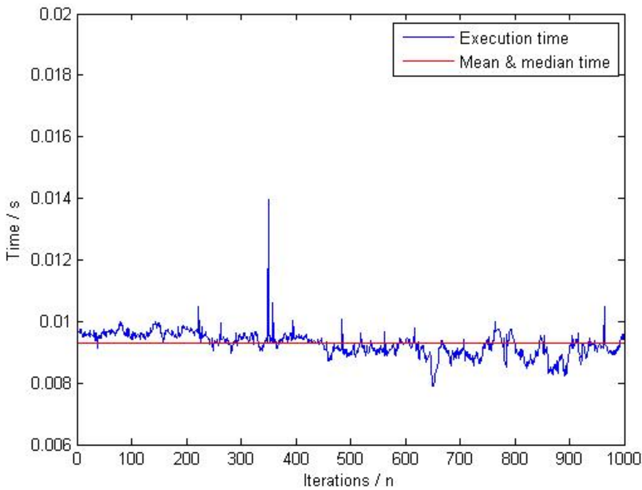

Figure 9.8 Execution time using translation approximation implementation for 1000 executions.

Translation approximation /s	
Mean	0.0093
Median	0.0093
Max	0.0172
Min	0.0079
Std	4.63×10^{-4}

Table 9.5 Execution time using translation approximation implementation for 1000 executions.

Performance on ground truth data sets For performance measurements we used the *freiburg1_teddy* sequence [*Freiburg1 Teddy sequence* 2011] as ground truth. The ground truth performance can be seen in Figure 9.9 and the statistics are presented in Table 9.6.

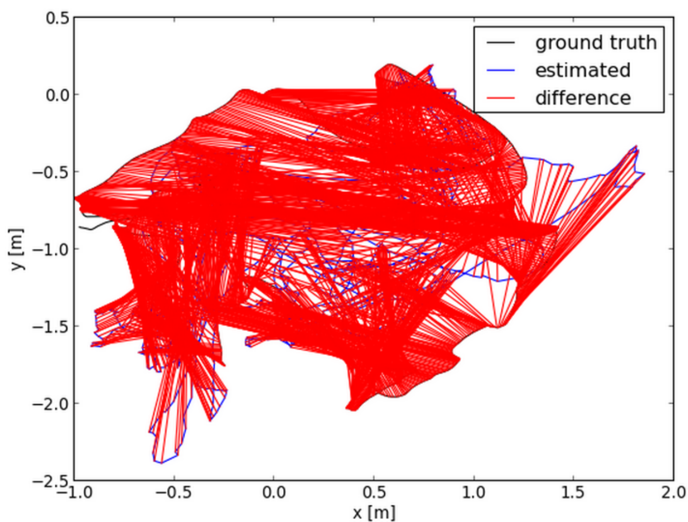

Figure 9.9 Absolute translation error using translation approximation for 1400 samples.

Translation approximation / m	
Rmse	1.04
Mean	0.965
Median	0.922
Std	0.399
Min	0.145
Max	2.12

Table 9.6 Absolute translation error using translation approximation for 1400 samples.

9.4 Gantry-Tau robot

Camera-Robot transformation

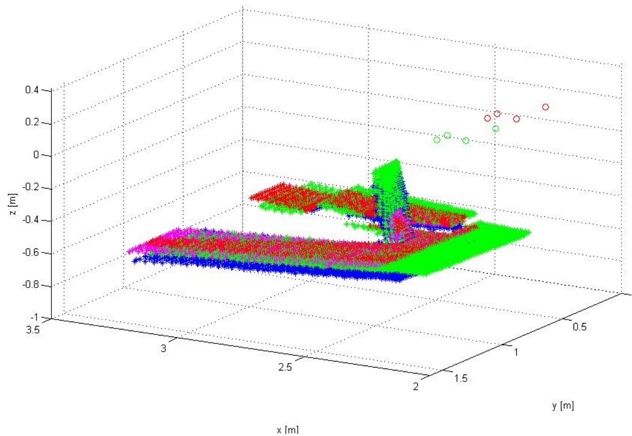

Figure 9.10 Global coordinates from multiple images and positions.

In Section 6.3 we calculated the transformation H_{c2r} from the camera to the robot needed to relate robot position to global point clouds. In Figure 9.10 we see the global coordinates of the floor and an obstacle from multiple images with different robot positions. The point clouds of each image have a unique color. The green dots are the camera positions and the red dots are the robot positions. Notice that the coordinate system used is the coordinate system of the robot and the plotted dots have been transformed using the calculated H_{c2r}. The consistency of the obstacle and the floor as well as the even distribution of points from different images implies that the calculated transformation H_{c2r} is a good approximation.

9.5 Path planning

Since the performance of ICP has a mean absolute error of 0.380m (point-to-plane) it could not be used in the experimental setup to verify the behavior of the path planning and discretisation. Therefore we used the robot's internal positioning to create the discrete environment needed for the path planning.

Performance

In this experiment a paper cylinder was placed about 30cm in front of and about 40cm under the robot (since the camera is pointed downwards). The start position of the robot is the upper blue dot marked in Figure 9.11. Then the robot is commanded to move to a position behind the cylinder, marked as the lower yellow dot in Figure 9.11. The shortest path from the start to the goal position would be straight through the cylinder but as the discrete map is created the path planning needs to re plan to avoid the obstacle resulting in a path traveling around the cylinder. The environment can be seen in Figure 9.11 where the discrete map of each camera is painted with the same color as the camera dot.

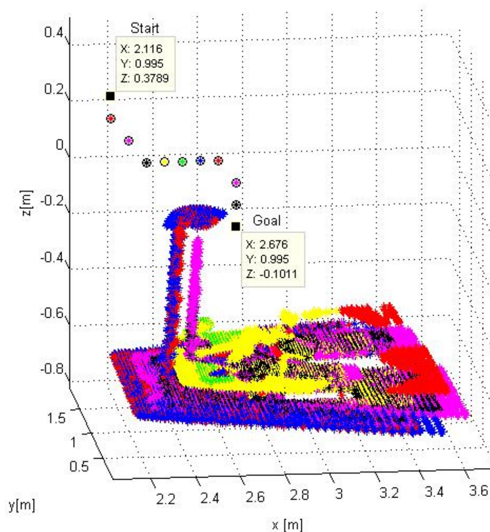

Figure 9.11 This figure shows camera position (circles) and obstacle map while running path planning. The left blue camera is the start position and the lower right yellow camera is the goal position.

We clearly see how more and more of the environment is discovered as the robot moves.

10

Conclusions

10.1 Image retrieval

After a comparison of the different avaliable RGB-D cameras the Asus Pro Live was chosen before the Kinect due to lower power consumption and the benefits for the image retrieval to use a lower resolution more suited for the task.

The Asus Pro Live performs good on timing test with a mean of 0.0279s and standard deviation of 0.0108s while claiming a sample time of 0.03s. The compact format of 320×240 is definitely an advantage since we do not want to handle more data than we can use with the chosen resolution of the box grid.

10.2 Updating the Obstacle map

Results show that box approximation using every pixel is faster than SDF and that Box Approximation using every fifth pixel is much faster than SDF. Performance wise SDF is better suited for image reconstruction with a higher need of precision but for graph generation box approximation is to prefer. SDF expects the area behind known objects to be untraversable until it is proven to be clear of obstacles and box approximation expects all areas to be traversable until an obstacle is spotted making it more compatible with the D* Lite algorithm which expects all paths to be free until proven otherwise. The main motivation for using box approximation in this application is that since SDF operates on the voxels a small object, for example a thin pole or hanging string, may lie in between two voxels and will not be represented in the discretization. Box Approximation on the other hand that operates on the pixels will instead fill the box even if just a small item is located there. This can of course lead the algorithm to believe that areas that could in fact be traveled are untraversable. Considering the nature of the application a rather diminished area avaliable for travel is better than a larger area with hidden and unrepresented obstacles.

10.3 Combine hexacopter

Hardware

From the results we can clearly see that the Pandaboard is not powerful enough to run all calculations online. Running rotation fused ICP alone on the Pandaboard had a mean time of 0.6735s using Point-to-Point ICP and 1.3774s using Point-to-Plane ICP. This is not nearly fast enough for the intended application.

Tracking

Looking at Table 9.3 we see that neither Point-to-Point ICP nor Point-to-Plane ICP are not nearly fast enough for running on the Pandaboard on-flight. Point-to-Point has a worst case of 2.45s while Point-to-Plane has a worst case of 3.36s, both unexceptionably slow. Looking at Table 9.4 we see that ICP Point-to-Plane is better than Point-to-Point performance wise but is still not good enough to be used in this application. The translation approximation algorithm is extremely fast but do not seem to work at all (see. Section 9.3).

10.4 Gantry-Tau robot

Camera-Robot transformation

As can be seen in Figure 9.10 the Camera-Robot transformation seems accurate. The plotted floor is well aligned with the robot's y axis. If the Camera-Robot transformation was not near the ideal transformation local coordinates from different camera positions would not be able to reconstruct the global coordinates of the floor and cylinder in an uniform way.

10.5 Path planning

The D* Lite performs well. When a new obstacle occurs it re-plans the path and manages to avoid the obstacle. Looking at Figure 9.11 we see that the shortest path from the start node to the end node is on the diagonal, which is the initially planned path. We can easily see that the path has been re-planned to avoid the cylinder.

11

Discussion

11.1 Problems

The problems which have shown up during the thesis work have mainly been related to Matlab and hardware restrictions. The Pandaboard was too slow to run the needed applications and the Asus Xtion pro live.

If a tracking algorithm is to be used with the current setup it needs to take the behavior of the depth data into consideration. Objects too close to the camera will be represented as zeros, this means that if we are traveling towards an obstacle eventually the distance to the obstacle will be reported as zeros even thought it in fact will be in the range between 0 and 0.8m [*Xtion PRO LIVE*]. If the tracking algorithm then simply tries to compare the transformation between two images where one contains zeros and the other contains the accurate distance to an obstacle it might be impossible to calculate the transformation.

11.2 Further work

If the setup is to be used on a UAV a more powerful processor is needed as well as an internal positioning system.

Bibliography

ArduCopter. Accessed 22 May 2015. APM COPTER. URL: http://www.arducopter.co.uk/.

Berlin, H., Program Manager (2012). *Text based robot programming made easy*. Accessed 22 May 2015. RobotStudio. URL: http://www.abb.com/blog/gad00540/1DDE6.aspx?tag=RAPID%20programming.

Bouguet, J.-Y. *Camera calibration toolbox for matlab*. Accessed 12 August 2014. URL: http://www.vision.caltech.edu/bouguetj/calib_doc/.

Bylow, E. (2012). *Camera Tracking using a Dence 3D Model*. Master's Thesis. Lund University, Faculty of Engineering, Center for Mathematical Sciences, Mathematics, Lund , Sweden.

Dressler, I. (2012). *Modeling and Control of Stiff Robots for Flexible Manufacturing*. PhD thesis ISRN LUTFD2/TFRT- -1093--SE. Department of Automatic Control, Lund University, Sweden.

Fogelberg, J. (2013). *Navigation and Autonomous Control of a Hexacopter in Indoor Environments*. Master's Thesis ISRN LUTFD2/TFRT- -5930--SE. Department of Automatic Control, Lund University, Sweden.

Freiburg1 Teddy sequence (2011). Last modified: 30 Sep 2011, 15:16. TUM. URL: http://vision.in.tum.de/data/datasets/rgbd-dataset/download#freiburg1_teddy.

G. Ramalingam, T. R. (1992). *An incremental algorithm for a generalization of the shortest-path problem*. University of Wisconsin, Madison.

Geiger, A. (2008). *ICP C++ implementation*. URL: http://www.cvlibs.net/software/libicp/.

Geiger, A., P. Lenz, and R. Urtasun (2012). "Are we ready for autonomous driving? The KITTI vision benchmark suite". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. Accessed 25 May 2015. URL: http://www.cvlibs.net/publications/Geiger2012CVPR.pdf.

Hand-Eye Calibration. Accessed 13 August 2014. Technische Universität München (TUM) Computer Vision Group, Garching, München, Germany: TUM. URL: http://campar.in.tum.de/Chair/HandEyeCalibration.

Kinect for Windows Sensor Components and Specifications (2011). Accessed 5 August 2014. Microsoft. URL: http://msdn.microsoft.com/en-us/library/jj131033.aspx.

Labcomm (2014). URL: http://wiki.cs.lth.se/moin/LabComm.

Low, K.-L. (2004). *Linear Least-Squares Optimization for Point-to-Plane ICP Surface Registration*. Tech. rep. Accessed 25 May 2015. Department of Computer Science, University of North Carolina at Chapel Hill. URL: http://www.comp.nus.edu.sg/~lowkl/publications/lowk_point-to-plane_icp_techrep.pdf.

MONROE (2012). Accessed 7 May 2015. Hyper-Modular Open Networked RObot systems with Excellent performance. URL: http://www.echord.info/wikis/website/monroe.

Niklas Ohlsson, M. S. (2013). *Model-Based Approach to Computer Vision and Automatic Control using Matlab Simulink for an Autonomous Indoor Multirotor System*. Master's Thesis. Department of Signals and Systems, Division of Automatic Control, Automation and Mechatronics, Chalmers University of Technology, Göteborg, Sweden.

OMAP TM 4 PandaBoard System Reference Manual (2010). 0,4. pandaboard.org. URL: http://makelinux.net/lib/ti/PandaBoard_SRM.

OpenNI 2 SDK. Accessed 14 April 2015. Open NI. URL: http://structure.io/openni.

P. E. Hart N. J. Nilsson, B. R. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics* **4**, pp. 100–107.

P.J. Besl, N. M. (1992). "A method for registration of 3D shapes". *IEEE Transactions on Pattern Analysis and Machine Intelligence - Special issue on interpretation of 3-D scenes-part II* **14**:2, pp. 239–256.

Rasberry Pi. Accessed 22 May 2015. Rasberry Pi Foundation. URL: https://www.raspberrypi.org/.

R.Y. Tsai, R. L. (1988). "Real Time Versatile Robotics Hand/Eye Calibration using 3D Machine Vision". *Robotics and Automation, Proceedings of the 1988 IEEE International Conference on* **1**, pp. 554 –561.

— (1989). "A new technique for fully autonomous and efficient 3D robotics hand/eye calibration". *Robotics and Automation, IEEE Transactions on* **5** (3), pp. 345–358.

S. Koening, M. L. (2002a). “D* Lite”. *Proceedings of the AAAI Conference of Artificial Intelligence (AAAI)*, pp. 476–483.

— (2002b). “Incremental A*”. In: *Advances in Neural Information Processing Systems (NIPS)*. Georgia Institute of Technology, College of Computing, Atlanta, pp. 1539–1546.

— (2005). “Fast replanning for navigation in unknown terrain”. *Transactions on Robotics* **21**:3, pp. 354–363.

SMErobot (2009). Accessed 7 May 2015. The European Robot Initiative for Strengthening the Competitiveness of SMEs in Manufacturing. URL: `http://www.smerobot.org/`.

Stentz, A. (1994). “Optimal and efficient path planning for partially-known environments”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA ’94)*. Vol. 4, pp. 3310 –3317.

— (1995). “The Focussed D* Algorithm for Real-Time Replanning”. In: *In Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 1652–1659.

Submission form for automatic evaluation of RGB-D SLAM results (2009-2013). Last edited 17.08.2012 15:51 by Juergen Sturm. TUM. URL: `http://vision.in.tum.de/data/datasets/rgbd-dataset/online_evaluation`.

Treiber, M. A. (2013). *Optimization for Computer Vision : An Introduction to Core Concepts and Methods*. Springer. ISBN: 9781447152828.

Wengert, C. *Fully automatic camera and hand to eye calibration*. Accessed 13 August 2014. URL: `http://www.vision.ee.ethz.ch/software/calibration_toolbox/calibration_toolbox.php`.

Xtion PRO LIVE. Accessed 5 August 2014. Asus. URL: `http://www.asus.com/se/Multimedia/Xtion_PRO_LIVE/specifications/`.

A

Appendix A

A.1 Camera calibration

In order for the camera to give correct images the camera was calibrated using the *Camera Calibration Toolbox for Matlab* following the guide *First calibration example - Corner extraction, calibration, additional tools* [*Camera Calibration Toolbox for Matlab*]. The calibration was done by calculating the intrinsic and extrinsic camera parameters. The intrinsic parameters describe the lens and camera attributes.

Focal length The camera focal length for a (perfect) Asus Xtion pro live is 525px.

Principal point The principal point for a (perfect) Asus Xtion pro live is $cx = 319.5$ $cy = 239.5$.

Skew coefficient In a (perfect) camera the screw coefficients are 0° since they define the angle between the x and y pixel axes.

Distortions The radial and tangential distortions in the camera lens.

The extrinsic parameters describe the 3D location of the calibration grid in the camera reference frame. In Table A.1 we see the calibrated intrinsic camera parameters. The calibration setup can be seen in Figure A.1.

Rotation The 3×3 rotation matrix.

Translation The 3×1 translation vector.

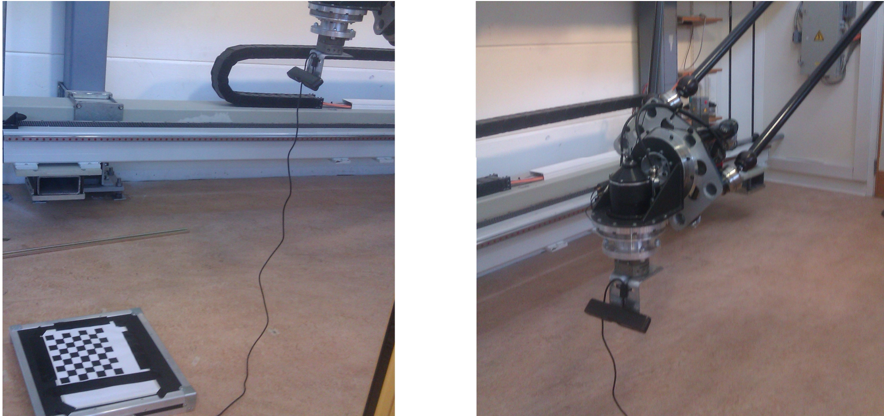

Figure A.1 Asus Xtion pro live camera mounted on Gantry-Tau robot.

Focal Length (mm) (fx, fy)	525.50 531.17
Focal Length uncertainty (mm)	8.70 8.99
Principal point (mm) (cx, cy)	327.97 245.57
Principal point uncertainty (mm)	8.58 8.61
Skew (rad)	1.90×10^{-3}
Skew uncertainty (rad)	2.09×10^{-3}
Distortion:	11.52×10^{-3} -210.41×10^{-3} -5.76×10^{-3} -1.54×10^{-3} 0
Distortion uncertainty	34.95×10^{-3} 132.85×10^{-3} 4.48×10^{-3} 3.87×10^{-3} 0
Pixel error	0.26 0.24

Table A.1 Calibration results (with uncertainties). Note: The numerical errors are approximately three times the standard deviations (for reference) [*Camera Calibration Toolbox for Matlab*].

In Figure A.2 we see a comparison of the estimates intrinsic parameters and the standard intrinsic parameters. In this image a rectangular box where placed on the floor. The box was 15.5 cm wide and 38 cm long which can be verified by looking at the data tips in Figure A.2. We also see that a small difference in the intrinsic parameters will not have a large impact on the local coordinates.

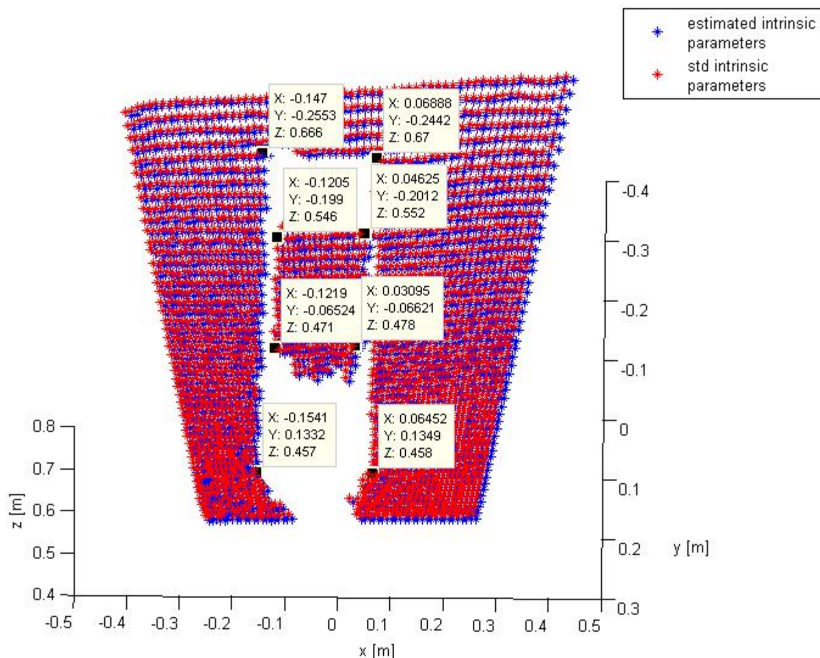

Figure A.2 Local coordinates generated with estimated intrinsic parameters from Table A.1 as well as standard intrinsic parameters.

A.2 Hand-Eye Calibration

Since our camera is mounted on the robot we also need to calculate the transformation between the robot and the camera. This is done by doing a hand-eye calibration using a *Camera Calibration Toolbox for Matlab* add-on implemented by Christian Wengert [*Fully automatic camera and hand to eye calibration*]. The calibration got its name from the robotics community since the camera (eye) is mounted on the robot gripper (hand) while doing the calibration [*Hand-Eye Calibration*]. The implementation is based on the Tsai-Lenz papers [R.Y. Tsai, 1988] [R.Y. Tsai, 1989].

The main idea is to log the robot position while taking numerous picture of a stationary pattern. The robot's coordinate system is considered the world coordinate system. From the extrinsic image parameters a camera-to-grid transformation can be calculated. Given the homogeneous transformation from the robot's coordinate system to the world's coordinate system H_{r2w} and the homogeneous transformation from the camera's coordinate system to the grid's coordinate system H_{c2g} the homogeneous transformation from the camera's coordinate system to the robot's

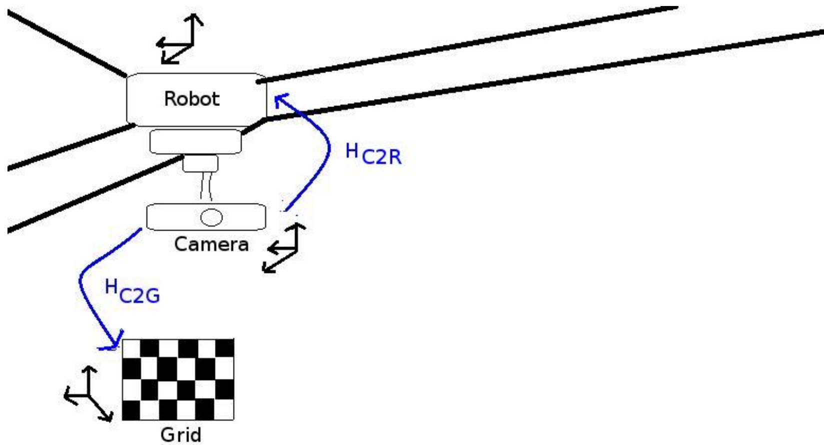

Figure A.3 Homogenous transformations between coordinate systems.

coordinate system H_{c2r} can be calculated using the Tsai-Lenz approach.

Running the application did not give a suiting result. Some of the matrices used in the Tsai-Lenz got ill conditioned and the applications was unable to give good results. Maybe this is due to the steep angle that the camera is mounted on the robot with.

When we used just a few carefully selected images the application managed to find a transformation matrix,

$$H_{c2r} = \begin{pmatrix} -0.0530 & 0.7818 & 0.6212 & -0.1938 \\ 0.9163 & -0.2092 & 0.3415 & -0.8496 \\ 0.3970 & 0.5873 & -0.7053 & 0.3789 \\ 0 & 0 & 0 & 1.0000 \end{pmatrix},$$

$$\text{with err} = \begin{pmatrix} 0.1293 \\ 234.4 \end{pmatrix} \text{px and back projection_error} = 55.84px.$$

The result can be seen in Figure A.4. We can see that there is an impossible relationship between the camera in [-0.1938 -0.8496 0.3789] and the floor since the right part of the point cloud should be pointed towards the camera. We also see that the floor is not aligned with the x-y plane. We conclude that this transformation is not good enough.

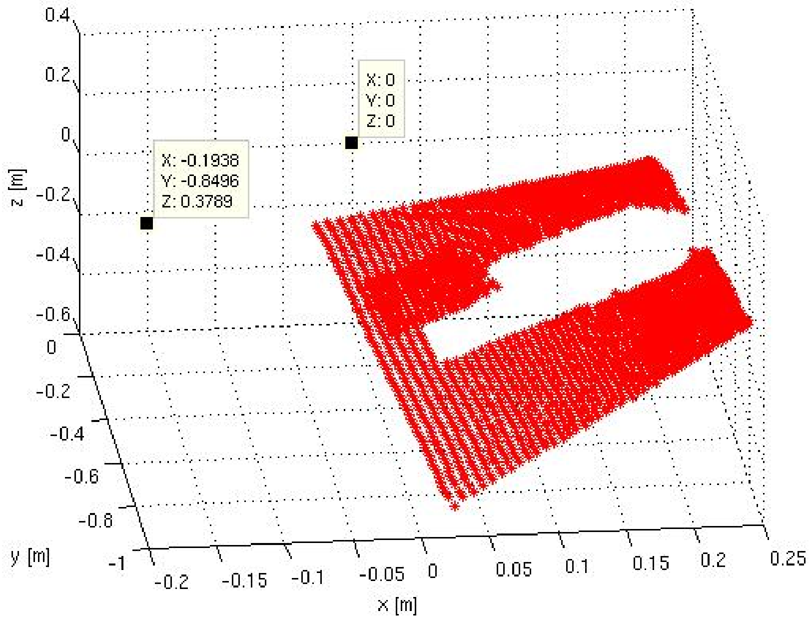

Figure A.4 Global coordinates generated with the Tsai-Lenz transformation matrix and robot placed in the origin.

B

Appendix B

B.1 Labcomm client

In Listing B.1 we see the client's mainmodule for the Labcomm communication.

```
MODULE mainModule

TASK PERS robtarget nextTarget :=
    [[375.269,-83.8462,629.143],[0.700909,0,0.71325,0],
[0,0,0,0],[0,0,0,0,0,0]];
    TASK PERS robtarget initPosition :=
    [[373.989,-84.1662,628.743],[0.700909,0,0.71325,0],[0,0,0,0],
    [9E+09,9E+09,9E+09,9E+09,9E+09,9E+09]];
    TASK PERS jointtarget initPositionJ :=[[0,0,0,0,1,0],
    [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];

        VAR LCRobot_robtarget tmp;

    PROC handle_command(LCRobot_robtarget val)
        VAR robtarget slask;
        slask:=CalcRobT(initPositionJ , tool0);
        nextTarget.trans.x := slask.trans.x + val.trans.x;
        nextTarget.trans.y := slask.trans.y + val.trans.y;
        nextTarget.trans.z := slask.trans.z + val.trans.z;
        nextTarget.rot.q1 := slask.rot.q1;
        nextTarget.rot.q2 := slask.rot.q2;
        nextTarget.rot.q3 := slask.rot.q3;
        nextTarget.rot.q4 := slask.rot.q4;
        TPWrite "Received values ---------------- ";
        TPWrite "x "\Num:=val.trans.x;
        TPWrite "y "\Num:=val.trans.y;
        TPWrite "z "\Num:=val.trans.z;
        TPWrite "q1 "\Num:=val.rot.q1;
        TPWrite "q2 "\Num:=val.rot.q2;
        TPWrite "q3 "\Num:=val.rot.q3;
        TPWrite "q4 "\Num:=val.rot.q4;
        TPWrite "New position ---------------- ";
        TPWrite "x "\Num:=nextTarget.trans.x;
        TPWrite "y "\Num:=nextTarget.trans.y;
```

```
        TPWrite "z  "\Num:=nextTarget.trans.z;
        TPWrite "q1  "\Num:=nextTarget.rot.q1;
        TPWrite "q2  "\Num:=nextTarget.rot.q2;
        TPWrite "q3  "\Num:=nextTarget.rot.q3;
        TPWrite "q4  "\Num:=nextTarget.rot.q4;

        move nextTarget;

    ENDPROC


    PROC main()
        VAR LabComm_Stream st;
        SocketCreate st.soc;
        SocketConnect st.soc, "130.235.83.245", 6555;
        TPWrite "Connected to server!";
        Receive_targets st;

    ENDPROC

    PROC Receive_targets(VAR LabComm_Stream st)
        VAR Decoder d;
        VAR LabComm_Decoder_Sample s{1};
        VAR LabComm_Encoder_Sample en{1};
        VAR Encoder enc;

        Init_Encoder enc, st;
        Init_Decoder d, st;

        %"LCRobot:Enc_Reg_robtarget"% enc, st, en{1};
        %"LCRobot:Dec_Reg_robtarget"% s{1}, "handle_command";
        WHILE TRUE DO
            Decode_One d, st, s;
            %"LCRobot:Encode_robtarget"% enc, st, en{1}, tmp;
        ENDWHILE
    ERROR
        SocketClose st.soc;
        RETURN;
    ENDPROC



PROC move(robtarget target)
 SingArea\Wrist;
  MoveL target, v100, z5, tool0\WObj:=wobj0;
ENDPROC

ENDMODULE
```

Listing B.1 Client mainModule run on the ABB IRC5 system during experiments with the Gantry-Tau robot.

B.2 Simulink wrappers

In Listing B.2 we see the C wrapper and in Listing B.3 the respective header file used to interface the C++ code in an S-function. In Listing B.5 we see the headerfile used by Simulink's s-function and in Listing B.4 we see the header file for the C++ code.

```
#include "wrapper.h"
#include "NI.h"

void asusInit(){
   initNI();
}
void asusCapture(uint16_T* depth){
    captureNI(depth);
}
void asusTerminate(){
    terminateNI();
}
```

Listing B.2 wrapper.c used to make it possible to run Openni C++ code with the Simulink code generation (see Listing 8.1).

```
#ifndef _wrapper_H_
#define _wrapper_H_
#include "rtwtypes.h"

#if defined(_RUNONTARGETHARDWARE_BUILD_)

extern void asusInit();
extern void asusCapture(uint16_T* depth);
extern void asusTerminate();

#else
/* Used in rapid accelerator mode */
#define asusInit()
#define asusCapture(depth)
#define asusTerminate()
#endif

#endif /* _wrapper_H_ */
```

Listing B.3 wrapper.h

```
#ifndef NI_H
#define NI_H
#ifdef __cplusplus
extern "C"{
    typedef unsigned short uint16_T;
    void captureNI(uint16_T *depth);
    void initNI();
    void terminateNI();
}
#endif /* __cplusplus */
#endif /* _NI_H_ */
```

Listing B.4 NI.h

```
/* Copyright 2012 The MathWorks, Inc. */
#ifndef ASUS_CAPTURE_H
#define ASUS_CAPTURE_H

#if defined(MATLAB_MEX_FILE)
/* This will be compiled by MATLAB to create the Simulink block:
            */


/* Model Start function*/
#define asusInit()

/* Model Step function*/
#define asusCapture(depth)

/* Model Terminate function*/
#define asusTerminate()

#else

#include "wrapper.h"
/* This will be called by the target compiler:     */

 /*Following prototype mapping is done in the code generation*/

#endif /*MATLAB_MEX_FILE*/
#endif /*ASUS_CAPTURE_H*/
```

Listing B.5 Asus_capture.h

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER´S THESIS

Date of issue
June 2015

Document Number
ISRN LUTFD2/TFRT--5980--SE

Author(s)
Sara Gustafzelius

Supervisor
Simon Yngve, Combine Control Systems AB
Erik Bylow, Mathematics, Faculty of Engineering, Lund University, Sweden
Anders Robertsson, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle

Dynamic path planning of initially unknown environments using an RGB-D camera

Abstract

In this thesis an RGB-D camera was used with the goal to perform dynamic path planning in an initially unknown environment. Depth data from an RGB-D camera together with a discretizising algorithm is continuously used for maintaining an obstacle map of the environment which within the path planning algorithm D* Lite [S. Koening, 2005] is performed on the flight.

Experiments were conducted on two different systems, on Combine's hexacopter and on a Gantry Tau robot at the Robot Lab of the Department of Automatic Control, LTH. On Combine's hexacopter different tracking algorithms such as ICP, Translation Approximation and SDF where evaluated for 3D positioning while the robots internal positioning where used on the Gantry Tau robot.

For discretization purposes we compare the use of Box Approximation and Signed Distance Function (SDF) for creating the obstacle map.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-87

Recipient's notes

Security classification

http://www.control.lth.se/publications/