Artificial Intelligence and Terrain Handling of a Hexapod Robot

Markus Malmros

Amanda Eriksson

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6010
ISSN 0280-5316

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

Printed in Sweden by Tryckeriet i E-huset
Lund 2016

Abstract

The focus of this master's thesis has been getting a six-legged robot to autonomously navigate around a room with obstacles. To make this possible a time-of-flight camera has been implemented in order to map the environment. Two force sensing resistors were also mounted on the front legs to sense if the legs are in contact with the ground. The information from the sensors together with a path planning algorithm and new leg trajectory calculations have resulted in new abilities for the robot.

During the project comparisons between different hardware, tools and algorithms have been made in order to choose the ones fitting the requirements best. Tests have also been made, both on the robot and in simulations to investigate how well the models work.

The results are path planning algorithms and a terrain handling feature that works very well in simulations and satisfyingly in the real world. One of the main bottlenecks during the project has been the lack of computational power of the hardware. With a stronger processor, the algorithms would work more efficiently and could be developed further to increase the precision of the environment map and the path planning algorithms.

Acknowledgements

First of all we would like to thank Combine Control Systems and our supervisor there, Simon Yngve, for giving us the possibility of writing this thesis for them and also for providing us with the equipment needed and helping us in the times we got stuck. A special thanks goes to Sara Gustafzelius and Andreas Tågerud for all the help with the camera implementation and computer hacking stuff.

We would also like to thank the Department of Automatic Control, at Lund University for providing us with tools and hardware during this master thesis. Our supervisor at the department, Anders Robertsson, has guided us through this project and helped us a lot with his ideas during our weekly meetings.

Finally we would like to thank Dan Thilderkvist and Sebastian Svensson for the previous work they did that made our work possible and for answering our questions about the Hexapod.

Acronyms

UAV	Unmanned Aerial Vehicle
IMU	Inertial Measurement Unit
RGB-D	Red Green Blue - Depth
TOF	Time-of-Flight
FSR	Force Sensing Resistor
RANSAC	Random Sample Consensus
RANSOP	Random Sample Optimization
RRT	Rapidly Exploring Random Trees
PRM	Probabilistic Road Map
BFS	Breadth First Search
DFS	Depth First Search
LIDAR	Light Detection and Ranging
UCS	Uniform Cost Search
SLAM	Simultaneously Localization And Mapping
CAD	Computer Aided Design

Contents

1

Introduction

1.1 Background

The growing trend of developing and using automated, unmanned vehicles could be very useful for making the work in dangerous and non human-friendly environments safer and more efficient. A robot could for example be used in the ruins after a big earthquake to find survivors and possible paths for emergency workers. This is one application that has inspired the work and the goals of this thesis. The project is based on a previous master thesis for Combine Control Systems in Lund where locomotion and movement patterns were developed for a six-legged robot, also known as hexapod. Since the hexapod is stable, has many legs and is relatively small it could work very well together with sensors and a path planning algorithm for applications in non human-friendly environments. An autonomous terrain handling hexapod is also a great possibility for Combine Control Systems to demonstrate their work and a way to present their competence and business idea at a technical fair.

1.2 Previous Work

The initial development of the hexapod used in this project was made by two master students, Dan Thilderkvist and Sebastian Svensson. They mounted a six-legged robot and programmed a micro-controller to work with a remote control. They also developed a feature for stabilizing the body of the robot as the angle of the ground changed. This was done by measuring the body angle with an IMU (Inertial Measurement Unit). The stabilization feature could however only be used when the robot was standing still. The software development process was performed according to principles in model-based design in Simulink and Matlab.

The robot used is of the model PhantomX AX Hexapod Mark II from Interbotix Labs, see Figure 1.1. It has six legs, uses three servos per leg and thereby provides 18 degrees of freedom. The hand-held control is wireless and communicates with the hexapod by using XBee modules [Thilderkvist and Svensson, 2015]. The computer

used on the robot is of the model BeagleBone Black and has a 1 GHz processor [*BeagleBone Black*].

Another master thesis used as an inspiration in this project was done by Sara Gustafzelius, also for Combine Control Systems in Lund. The objective of Sara's project was to use an RGB-Depth camera together with a path planning algorithm on an Unmanned Aerial Vehicle (UAV) to perform a dynamic path planning. The RGB-D camera was used to map the world so that the robot could avoid obstacles in the environment. The same camera, an Asus Xtion Pro Live and a version of the path planning algorithm, D*, was later used in this thesis.

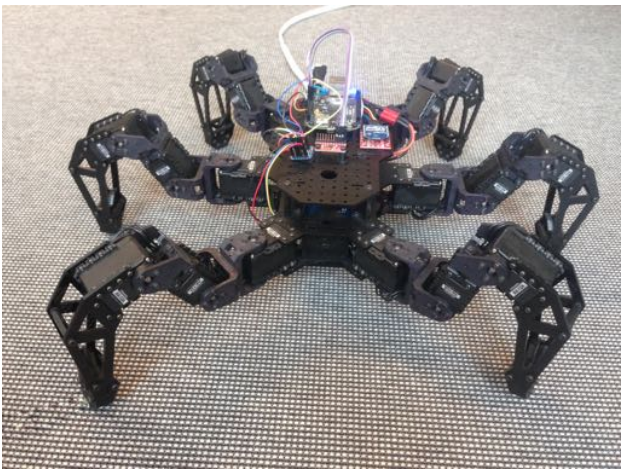


Figure 1.1 PhantomX AX Hexapod Mark II

1.3 Goals

One of the main goals of this master thesis was to develop the hexapod to analyze the environment and autonomously make dynamic, optimal decisions to avoid obstacles and reach a goal. This required using mapping sensors, finding a way to detect the current position, programming algorithms for mapping the environment and planning the path as well as control the movements and the path following of the robot. The other main goal was to develop the terrain handling of the robot. In normal walking mode the hexapod could only walk on flat ground without terrain but one of the desires from Combine Control Systems was that the robot also should sense the ground and thereby be able to avoid cliffs and walk on stairs. To make this possible a way of sensing the ground needed to be found as well as reprogramming the trajectories for the legs of the hexapod.

1.4 Resources and Division of Labor

To make the project work more efficient the labor was divided between Amanda and Markus during the thesis. Markus has more of a computer science background with a focus on artificial intelligence, statistics and planning. Therefore he focused on researching and implementing the systems for performing planning, mapping and path-following autonomously.

Amanda, on the other hand, has taken more courses in mechatronics, electronics and embedded systems and therefore focused more on the hardware. That involved for example setting up the camera and get it to work with Simulink, find, buy and install the force sensors and activate the accelerometer to test whether it could be used as a positioning tool.

1.5 Tools

The two main tools used in this thesis are Matlab [*Matlab*], with the Simulink environment [*Simulink*], and the robot simulator V-REP [*coppeliarobotics.com*]. More about V-REP and why it was chosen as a simulation tool can be read in Chapter 2 and 3.

Figure 1.2 Matlab Simulink

Figure 1.3 V-REP

1.6 Outline

The first chapter of the report contains background information to give the reader an introduction to the rest of the thesis. The purpose of the project and a couple of previous projects that made this thesis possible are also mentioned in the introduction as well as the different goals and constraints that have been followed throughout the project.

In Chapter 2 the theory behind the methods later used is explained. Some concepts and models are also explained to make it easier for the reader to understand the following chapters. The theory section also contains information about the tools used, for example V-REP, and explains how the sensors work, both the time-of-flight (depth) camera and the force sensing resistance.

The following chapter, Chapter 3, contains more information about why different programs, sensors and tools were used and how they were used. This includes some comparisons between the chosen program, sensor or tool and other, similar ones. The methods used to achieve the goals are also listed and described as well as some changes from the previous master thesis by Thilderkvist and Svensson [Thilderkvist and Svensson, 2015].

The trials that were made to test the theory and the different methods, as well as the results from the tests, are described thoroughly in Chapter 4. It contains both tests made in the simulation environment and in the real world on the actual robot. Trials to test the algorithms to find the most optimal ones are also listed in Chapter four as well as some tests on the force sensors.

Chapter 5 is the last chapter of the report and contains a discussion of the chosen methods and the results as well as some improvements that could be made and suggestions of further developments.

2

Theory

2.1 Modeling and Simulation Software

The main software used in this project to model, simulate and analyze the code and systems are Mathworks Simulink and V-REP.

Simulink and Matlab

Simulink and Matlab are two products from the corporation Mathworks. Together they can be used to create a model of a system and simulate it and/or run it on external hardware. External code can be included to the model which makes the code development process easy and efficient. The software is also good for analyzing the results of the simulations [*Matlab*] [*Simulink*]. All the code that is used on the hardware in this project is programmed directly in or is code-generated through Simulink.

Simulink Code Generation

In order to write the code from Simulink to the hardware, code generation is required. Code generation transforms the source code from the Simulink model into executable code that can be run on the hardware. However, this requires the Mathworks-tool Embedded Coder [*Understanding C Code Generation*]. The setup used is based on the setup from the previous master's thesis on the hexapod by Dan Thilderkvist and Sebastian Svensson [Thilderkvist and Svensson, 2015].

V-REP

V-REP is a robotics simulator with broad capabilities in simulation of mobile robots, factory automation and a tool for fast algorithm development. It is cross-platform compatible and works on Windows as well as Mac and Linux. V-REP provides three different physics-engines for dynamic calculations and collision detection between meshes. A wide range of sensors and terrain capabilities are also included [*Enabling the Remote API - client side*].

2.2 Force Sensing Resistor

The force sensing resistors, FSRs, used in this project are manufactured by the company Interlinc Electronics. They are 5 mm in diameter and have a thickness of 0.30 mm, hence, they are small enough to fit under the feet of the hexapod. When a force is applied to the circular part of the sensor the resistance is lowered and with increasing force the resistance is decreasing even more [*FSR Force Sensing Resistor Integration Guide and Evaluation Parts Catalog*]. The two ends of the sensors are connected according to the circuit in Figure 2.2. One of the sensor pins are connected to VDD and the other pin is connected to both ground together via a 10 k Ohm resistor and to one of the analog inputs of the BeagleBone Black. The changes of the analog input pin can thereafter be read and the information can be used to program the walking pattern of the hexapod.

Figure 2.1 The force sensing resistor [*Picture of FSR*].

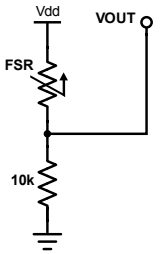


Figure 2.2 The electrical circuit of the sensor connection.

2.3 Time-of-Flight (depth) Camera

Time-of-flight cameras have reached an upswing during the last couple of years when they have become cheap consumer products, mainly from the introduction of the Microsoft Kinect camera. The cameras are a compact and convenient way of gathering 3D information about the world. They offer high sample rates and accurate depth readings.

A TOF camera works by emitting pulses of infrared light which illuminates the environment. The wave length is usually around 850 nm and not visible to the human eye. The light spreads across the environment and will be reflected on objects. The reflected light arrives back at a photosensitive imaging sensor designed for the infrared spectrum. The time of flight of the light is measured. Since the speed of light is known this information can be used to calculate the distance to a certain point in space. This is done simultaneously for all pixels in the image sensor ma-

trix. The equation used for calculating the depth is presented in Equation 2.1 where d is the depth distance in the current pixel [Li, 2014]. The speed of light is denoted c, Δt is the amount of time during which the illumination occurs, that is, pulse width. The reflected energy is measured by two out of phase windows, C_1 and C_2. Electrical charges accumulated during these two windows are integrated and denoted Q_1 and Q_2. Figure 2.3 illustrates the time aspects of measuring the depth distance.

$$d = \frac{1}{2} c \Delta t \frac{Q_2}{Q_1 + Q_2} \tag{2.1}$$

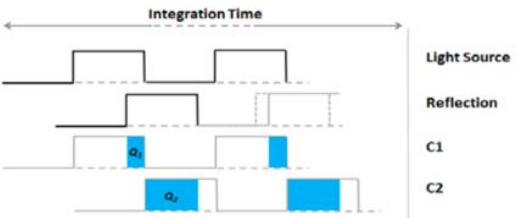


Figure 2.3 Time-of-flight time integration [Li, 2014]

The TOF camera will generate a matrix with depth values. An example is presented in Figure 2.4 where brighter areas are close by and darker areas are farther away. The completely black areas contain zeros, which happens when the sensor does not perceive enough reflected light. This can happen when an object is too close, too far away or if no light is reflected back from the surface. The last case can occur on smooth surfaces, such as glass or mirrors, which do not spread the incoming light. When the light is not dispersed in a large amount of directions on impact with an object, there might be no reflected light coming back to the sensor.

Figure 2.4 Example of a depth map

2.4 Depth Map To Coordinates

In order to get a global view of the environment, the depth matrix does not provide enough information. But if the matrix is transformed into coordinates in space, a point cloud emerges. In order to do this, four intrinsic camera parameters have to be known. The transformation equation is presented in Equation 2.2 where f_x and f_y are focal lengths and c_x and c_y are the pixel coordinates where the principal axis and image plane meets. A pixel indexation is denoted (i, j) where i is the pixel index on the x-axis and j is the pixel-index on the y-axis. I_d is the depth map and $z = I_d(i, j)$ is the measured distance for index (i, j) [Gustafzelius, 2015]. The concept is illustrated in Figure 2.5 where x_c,y_c and z_c is the camera coordinate system and the grid is the image plane.

$$(x, y, z) = (\frac{(i - c_x)z}{f_x}, \frac{(j - c_y)z}{f_y}, z) \tag{2.2}$$

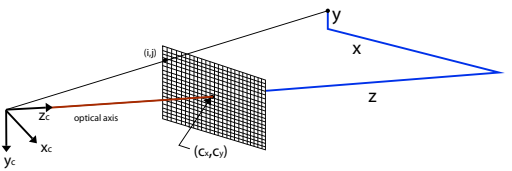


Figure 2.5 Depth camera [Gustafzelius, 2015]

Equation 2.2 has to be applied to all pixels in the depth map. Each pixel contains a depth which after transformation will be a point in space. An example of a full point cloud originating from one depth image is presented in Figure 2.6.

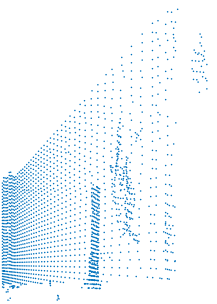


Figure 2.6 Example of a resulting point cloud

2.5 Random Sample Consensus (RANSAC)

RANSAC is a method for fitting model parameters to data containing outliers and inliers. The outliers are not to be included in the model and the inliers should be included, though they may be subject to noise. The following steps are repeated for a suitable amount of iterations [Fischler and Bolles, 1981].

1. Choose a subset of the points in space at random. This subset is seen as the hypothesis.
2. Perform an estimation of a hypothesis model based on the hypothesis set of points in space.
3. Evaluate all the data points against the hypothesis model according to some evaluation function. The data points close enough to the model are added to a set called the consensus set.
4. The model may be re-estimated using the data in the consensus set.

When a number of hypothesis subsets have been tested the models are compared to each other and the best ones are used as the final model. Figure 2.7 shows an example of RANSAC where a line is fitted to the data points. The blue points have been found to be inliers by the RANSAC algorithm while the red ones are outliers.

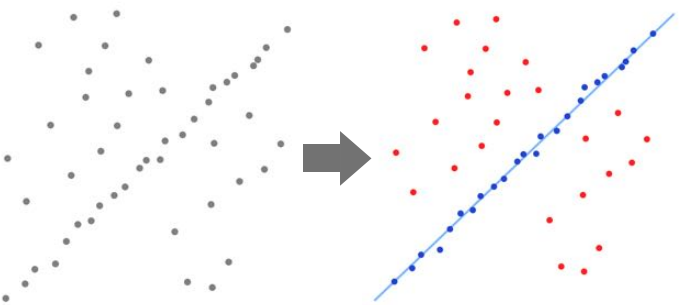


Figure 2.7 RANSAC example, [*Example of RANSAC*]

2.6 Random Sample Optimization (RANSOP)

RANSOP was developed during this master's thesis in an attempt to overcome the drawbacks of RANSAC. The main drawback to overcome was the fact that RANSAC does not take into account initial information about a probable range for the parameter estimates. Another goal was to keep computational time low.

The standard RANSAC assumes that there is no information about the model in addition to the data points and model structure. But using the RANSOP instead of the standard RANSAC comes at the cost of robustness. There are no guarantees that RANSOP will find the correct plane and there is always the risk of statistical variance in the chosen points to throw the results off.

A plane in space can be defined in its most sparse version needing only four parameters according to Equation 2.3. a, b and c are parameters which describe the orientation of the plane, that is, the normal vector, $\bar{n}$, to the plane. d is the perpendicular distance between the plane and the origin. These are the four parameters which are to be estimated by RANSAC or RANSOP. See Figure 2.8 for an illustration.

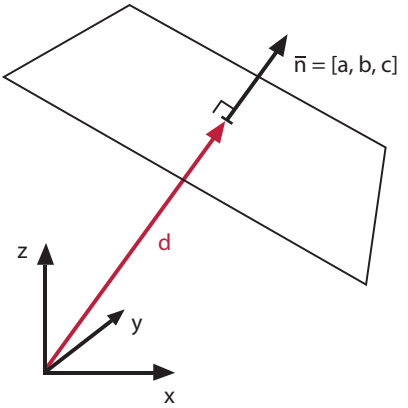


Figure 2.8 Basic description of a plane.

$$ax + by + cz + d = 0 \tag{2.3}$$

The algorithm works by sampling a subset of points from the point cloud and also guesses initial model parameters based on external information. The set of points are kept as validation points for evaluating the model fit. Then choose three points for a number of iterations, fit the model to the points. Compute a loss value based on the difference between the fitted plane and how well the plane fits the validation points. The sampled set of three points with the best loss value is deemed the best model. There are two parameters in the algorithm which trade-off between robustness and computational time. The choices are how many validation points to use and how many sample sets to use. The RANSOP algorithm is presented in pseudo code as Algorithm 1 below.

Algorithm 1: Random Sample Optimization (RANSOP)

Data: Point Cloud
Result: The four parameters which describe the fitted plane
input : $nbrValidationPoints$ - Number of points used for validation
$nbrSampleSets$ - Number of sets to sample
$planeInitNormal$ - Plane guess normal vector
$planeInitDistance$ - Plane guess distance
$pointCloud$ with $nbrPoints$ points each with x,y and z coordinates
output: $planeParamers$ - The four parameters describing the optimal plane
$validationSet$ = sample a uniform random subset of points from $pointCloud$;
$costVect$ = saves the cost for all sampled sets;
$distanceThreshold$ = Threshold for when a validation point is close enough to the model to be considered an inlier;
$penalty$ = Cost to add when validation point is not part of the inlier set;
for $iSet = 1 : nbrSampleSets$ **do**
  $samplePoints$ = sample three random points from $pointCloud$;
  $[normVect, d]$ = Find parameters for a plane using least square estimate on $samplePoints$;
  $angleDiff = abs(acos(planeInitNormal * normVect))$;
  $heightDiff = abs(planeInitDistance - d)$;
  $sumCost$ = 0;
  for $iValPoint = 1 : nbrValidationPoints$ **do**
    $distance = abs(validationSet(iValPoint) * normVect - d)/norm(normVect)$;
    if $distance < distanceThreshold$ **then**
      $sumCost = sumCost + distance$;
    else
      $sumCost = sumCost + penalty$;
    end
    $costVect(iSet) = sumCost + angleDiff * costWeight$;
  end
end
return: the plane parameters with lowest cost

The following paragraph is a reasoning regarding how many sample points are needed in order for the different success probabilities. The number of points is denoted $nbrPoints$, number of sample sets chosen are $nbrSampleSets$. Part of the point set is an inlier set for the model called $inlierSet$ with $nbrInliers$ points. When sampling a point from the point cloud uniformly at random, there will be a p chance of choosing a point within the inlier set according to Equation 2.4. When choosing three points independently there will be a p^3 probability that they are all in the inlier set.

$$P(point \in inlierSet) = \frac{nbrInliers}{nbrPoints} = p \tag{2.4}$$

The binomial distribution can be used to find the probabilities of successfully sampling three inlier points for different values of *nbrSampleSets*. The value of the binomial random variable is the number of "successes" out of a random sample of n trials. *nbrSampleSets* equals the number of trials and π is the probability of success for one trial, which equals p^3. The binomial probability formula is presented in Equation 2.5 where y is the number of successful trials. One success is enough in order for RANSOP to produce a working model, so finding the probabilities for $P(y \geq 1)$ is of importance for analyzing how many sets to be sampled.

$$P(\text{y successes in n trials}) = \frac{n!}{y!(n-y)!}\pi^y(1-\pi)^{n-y} \tag{2.5}$$

2.7 Planning Problem Formulation

Planning is concerned with moving an object, in this case a robot, from a start position and orientation to a goal. This has to be performed under certain constraints, either imposed by the properties of the robot or the environment. Below a couple of important concepts are presented [Lavalle, 2006]. In order to illustrate these, a simple example is constructed and presented in Figure 2.9. The magenta blocks are obstacles, the black ball is the robot, the green ball is the goal.

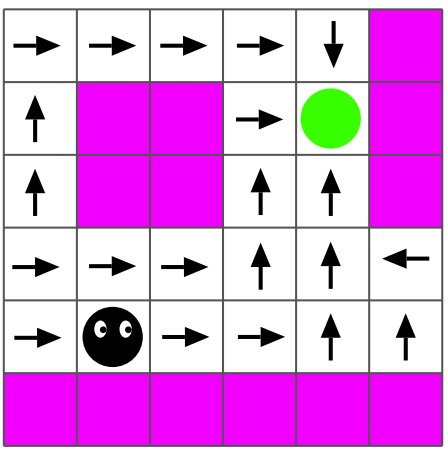


Figure 2.9 Problem formulation example

States The states are all possible situations in a planning problem. There could be a finite number of states in a discretized environment or an infinite number of states in a continuous environment. In the simple example above all white cells are possible states for the robot.

Configuration Space (C-space) The configuration space of a robot is a set of all possible states. C-free is the part of configuration space not occupied by an obstacle. In the simple example the c-space is the whole grid while c-free is the set of white cells, which are traversable.

Time Time always has to be taken into account. Either as an objective variable for optimizing the planning, or as a way of keeping a sequence in order.

Actions An action can be performed by the robot in order to change the state. The set of actions in the simple example is to move right, left, up or down.

Plan A plan is a strategy for determining which actions to take in every states in order to reach a goal state. The current plan in the simple example is illustrated by the arrows. There is an action associated with each possible state, that is, the preferred direction to walk in every cell is the direction of the arrows. The set of all of these actions constitutes a plan.

2.8 Non-holonomic System

A non-holonomic system is where the current state depends on the path which was taken to reach the state. It is described by a set of parameters subject to differential constraints, such that if a system evolves along a path in the c-space and then returns to the original set of values the system itself might not have returned to the original state [Lavalle, 2006].

2.9 Motion Planning

The motion planning problem consists of building an algorithm to be given a high-level specification of a task and converting it to low-level instructions. It has to be possible for an autonomous system to actuate the instructions. Motion planning is usually only concerned with the issue of kinematics and velocities, but does not regard dynamics and differential constraints. It is usually performed in a continuous world and the objective is to find a path from start to goal state in the free c-space without colliding with obstacles [Lavalle, 2006].

2.10 Overview of Planning in Continuous Space

The state space in the real world is uncountable infinite and a robot motion planning problem must therefore be simplified in some way. Configuration space is the

state space in motion planning. The number of dimensions of c-space corresponds to the number of degrees of freedom of the robot. Using the configuration space, motion planning will be viewed as a kind of search in a high-dimensional configuration space that contains implicitly represented obstacles. There are two ways of discretizing the c-space, combinatorial and sampling based [Lavalle, 2006]. A majority of planning methods contains two steps: Building the road map and performing search in the road map graph.

2.11 Road Map

A road map is a graph of connections between possible states, essentially the topology of a planning problem. A road map is usually the result of a discretization of the continuous c-space. In the graph each vertex is a state, while the edges denote possible transitions between the states, that is, a collision free path. For many of the planning methods presented, one of the first steps is to build a road map. An example is presented in Figure 2.10. The free space is filled with possible discretized states displayed as the blue graph network. The magenta obstacles can not be traversed and are therefore not connected in the graph. Note that this is a simple two degree-of-freedom example where the state space and road map can be visualized in a simple way.

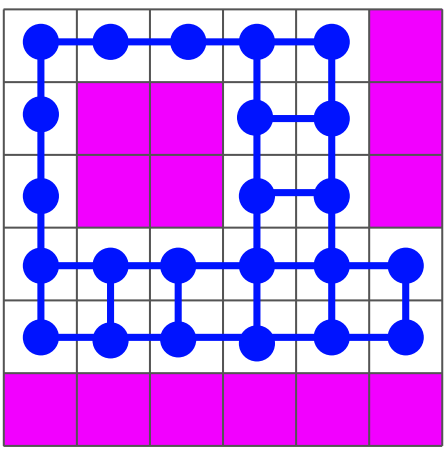


Figure 2.10 Road map graph example

Combinatorial Planning

Combinatorial planning characterizes free c-space explicitly by capturing the connectivity of free c-space in a graph. These methods have a couple of convenient features. They are complete, which means that if a solution exists it will be found, even though it might be time consuming. The solution is found by using search and a failure will be reported if no path exists. All of these methods produce a road map.

Uniform Grid Discretization This method of building a road map is one of the most straight forward. The whole space is discretized into a uniform grid. Cells are marked as either blocked or free. The free configuration space is discretized as free cells and these cells are put as vertices in the road map graph. They are connected with edges in order to create the map. The vertices can be connected in different ways. One option is to connect a vertex only to the four adjoining neighbors [Russel and Norvig, 2010]. Another method is to connect to the eight adjoining neighbors and thereby allowing diagonal movement in the path. Figure 2.11 provides an example where the first image is the continuous world with the pink obstacles. Image two contains the uniformly discretized map with the road map graph in blue.

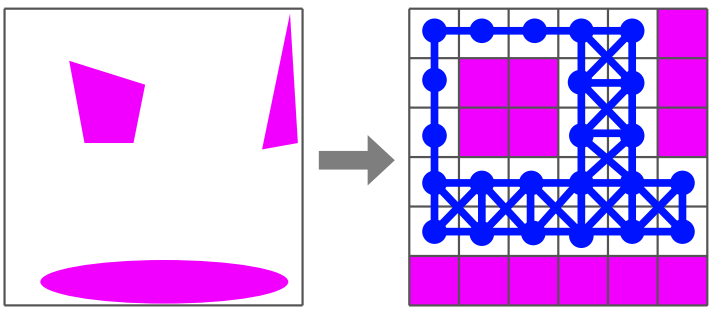


Figure 2.11 Uniform grid discretization example

Visibility Graphs If the world is assumed to contain polygonal obstacles there will be both convex and concave corners on the obstacles. Every corner is seen as a vertex. An attempt to connect all vertices to each other is made. An edge is added between two vertices when there is no obstacle obstructing a straight line between them [Russel and Norvig, 2010]. An example is presented in Figure 2.12, where red and green nodes represent start and goal.

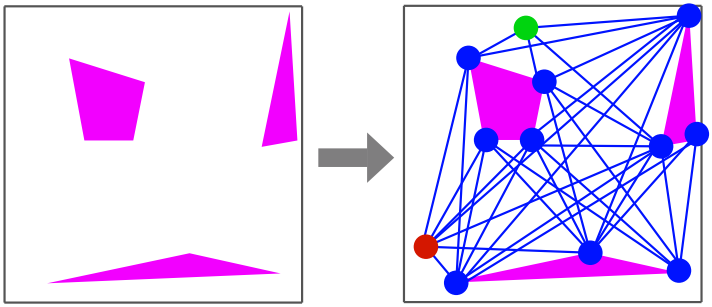


Figure 2.12 Visibility Graph example

Voronoi Diagram A Voronoi diagram is another method of building a road map. Vertices will be placed in locations with the same maximum distance from all the nearest obstacles. An example of a Voronoi diagram is presented in Figure 2.13. A Voronoi diagram will ensure that a high level of clearance is achieved since the distance to obstacles will always be at a maximum. On the other hand, paths will not be optimal and for open spaces there will be a strong attraction to the center of the open space [Russel and Norvig, 2010].

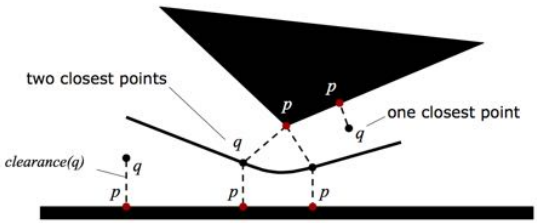


Figure 2.13 Voronoi diagram example, [Russel and Norvig, 2010]

Sample-Based Planning

Sample-based planning is based on probabilistic methods for sampling the state space in order to build a road map. Collision detection is used to probe and incrementally search the c-space for a solution. These methods are usually more efficient than the combinatorial planning methods, but they are not complete. This means that there is no guarantee of finding a path even though one exists. The sample-based methods are convenient and in many cases necessary for complex planning problems with a high number of degrees of freedom. The reason being that these methods can explore tight state spaces by running a high sampling frequency in the most difficult areas.

Probabilistic Road Map A probabilistic road map works by sampling the whole c-space. A random sample in c-space is declared a vertex in the road map if it is in the free c-space. The whole space can be sampled uniformly or sampling can be focused close to obstacles and narrow passages. Each new vertex is connected with its neighbors if there are no obstacles between the vertices. Two vertices can be considered neighbors if the euclidean distance between them is lower than a certain threshold. Another method is to use a clustering algorithm such as k-nearest neighbors to decide which vertices are neighbors. A probabilistic road map is a great way of building a road map efficiently. It is however neither complete nor optimal [Russel and Norvig, 2010]. An example of a probabilistic road map is presented in Figure 2.14 where the first image contains the sampled nodes and the second one illustrates the nodes connected to their neighbors in a graph.

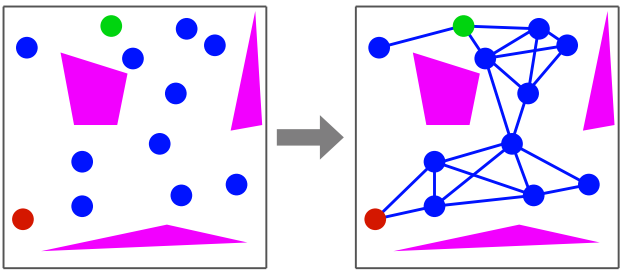


Figure 2.14 Probabilistic Road Map Example

Rapidly Exploring Random Trees The RRT is initialized by choosing an initial configuration which will act as the tree root node. From this root the tree will be expanded into the c-space. Sampling for another node is performed in the region around the graph. An edge is created between the sampled node and the nearest node in the graph if there is no obstacle between them. That is done by checking intermediate points at regular intervals between the two nodes. If no collisions are found, the edge is added. At an interval try to put the goal node as a new node to connect to the graph. If it is connected successfully, a path is found. An advantage of RRTs is that they combine an exploratory and exploitative approach which can be efficient in many cases [Lavalle, 2006]. An example of what a complete RRT might look like for different amounts of iterations is presented in Figure 2.15.

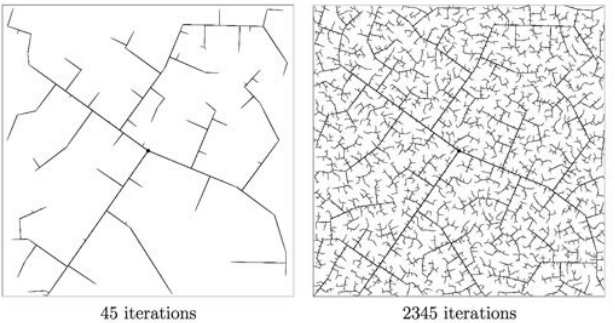


Figure 2.15 Example of RRT, [Lavalle, 2006]

2.12 Search Methods in a Discrete State Space Graph

The previously mentioned techniques are used in the building of a road map. This provides a connection between vertices, but it does not provide an actual path. For that purpose a search method is needed. Many instances of search in a graph is performed by expanding the graph into a tree structure instead. If nodes are allowed to be revisited the tree can become infinite, even if the graph is finite. The general structure of a graph search with a search tree is presented in Algorithm 2 and Figure 2.16. In this example nodes are not revisited. Nodes can be denoted in different ways depending on whether they have been expanded into the search tree or not. The nodes can be in three different states.

Unvisited nodes are those who have not been expanded into the search tree.

Alive nodes have been visited, but some of their children have not been visited.

Dead nodes have been visited along with all of their children.

The expansion of the tree nodes is done by storing a new set of alive states in a priority queue. These are the children of the expanded nodes, that is, fringe nodes. The difference between different search methods is the function used to sort the priority queue [Russel and Norvig, 2010].

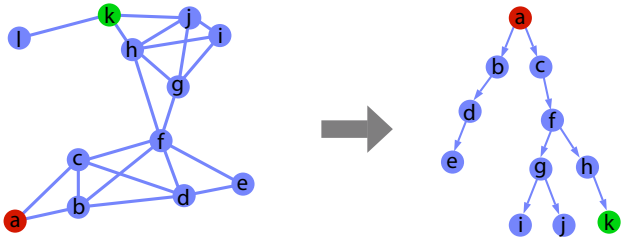


Figure 2.16 Building a search tree from a graph

Algorithm 2: General expansion of graph to tree

```
Initialize by putting the initial node as the root of the tree;
while Goal node has not been found do
    Select a node from the search tree;
    if node corresponds to the goal node then
        break loop;
    else
        expand the node by putting its successor nodes as children in the
        search tree;
    end
end
Walk up the tree and store all intermediary nodes to get the path;
```

Informed or uninformed Search Uninformed search are search methods which do not use any form of domain knowledge to perform the search. Only the actual graph is used in the search. Examples of uninformed search are breadth first and depth first search. Informed search on the other hand uses an evaluation function to decide which node is the most promising to explore. This heuristic can be knowledge such as the direction of the goal which might be used to bias a search in that direction. Commonly used informed search methods are greedy best first search or A* search [Russel and Norvig, 2010].

Performance Measures of Search

A couple of measures are needed to aid in the comparison of search algorithms.

Completeness is fulfilled if an algorithm always will find a solution if one exists.

Optimality means that an optimal path will be found if one exists. An optimal path is seen as the path which minimizes some cost function.

Complexity can be either seen as the memory or time needed in order to perform a successful search. The complexity is denoted in big $\mathcal{O}$ notation which essentially shows the scaling factor of time or space with growing problem size.

Branching factor, b is the number of children each node has.

Goal depth, d is the depth of the shallowest goal node, that is, the number of steps in the search tree from the start node to the closest goal node.

Max depth, m is the maximum depth of any node in the tree.

Breadth First Search

Breadth first search (BFS) will explore the graph by only expanding deeper into the graph when all of the nodes at the same level in the search tree has been expanded. In BFS the fringe nodes are kept in a FIFO, first in first out, queue [Russel and Norvig, 2010].

Completeness Yes, if the branching factor, b, is finite

Optimality If the cost function is the number of steps in the graph, then yes.

Complexity $\mathcal{O}(b^d)$ is the complexity for both time and space.

Depth First Search

DFS is the opposite of BFS and performs a search along one branch of the tree until it can not be expanded any more. The fringe node expanded is the most recently inserted into the tree. Fringe nodes are kept in a LIFO, last in first out, queue [Russel and Norvig, 2010].

Completeness Yes, if m is finite and there are no loops in the state graph or if states are not revisited.

Optimality No

Complexity $\mathcal{O}(b^m)$ is the complexity for time, while $\mathcal{O}(bm)$ is the space complexity.

Uniform Cost Search

BFS and DFS only use the topology of the graph in order to find a path. If a cost can be estimated for traversal of each edge, this information can be used in order to find a more optimal path faster. Uniform cost search uses a priority queue to store the fringe nodes where the priority is decided by total distance from root to node. The distances of the edges are added and the shortest total distance will have the highest priority and be expanded next [Russel and Norvig, 2010].

Completeness Yes, if b is finite and all edges have a cost larger than 0.

Optimality Yes

Complexity $\mathcal{O}(b^{1+floor(\frac{C}{\varepsilon})})$ is both the time and space complexity, where C is the cost of the optimal solution and ε the smallest possible edge cost.

A* Search

A* is similar to uniform cost search but in addition to using the cost for traversed distance along the edges a heuristic is used to estimate the distance to goal.

n: The current node

g(n): Cost from root to node *n*.

h(n): Estimated cost from node *n* to goal.

f(n) = g(n) + h(n): Total cost.

The node with the lowest f(n) will have the highest priority in the priority queue and will therefore be expanded first.

Completeness Yes, if all edges have a cost larger than zero.

Optimality Yes, if the heuristic is admissible.

Complexity Depends on which heuristic is used.

D* Search

A drawback of the previously mentioned search methods is that if there is a change in the planning map, position of the robot or goal, the whole search has to be re-performed. D* is short for dynamic A* and is a version of A* but with a minimum of recalculations for a changing environment. When the environment changes, D* only re-plans the path locally. D* Lite is another implementation of the D* algorithm and is at least as efficient. That makes D* Lite well suited for real-time application [Koenig and Likhachev, 2002]. The algorithm is presented in Algorithm 3.

Algorithm 3: D* Lite, [Koenig and Likhachev, 2002]

procedure CalculateKey(s)
{01'} return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m ; \min(g(s), rhs(s))]$;

procedure Initialize()
{02'} $U = \emptyset$;
{03'} $k_m = 0$;
{04'} for all $s \in S\ rhs(s) = g(s) = \infty$;
{05'} $rhs(s_{goal}) = 0$;
{06'} U.Insert(s_{goal}, CalculateKey(s_{goal}));

procedure UpdateVertex(u)
{07'} if $(u \neq s_{goal})\ rhs(u) = \min_{s' \in Succ(u)}(c(u, s') + g(s'))$;
{08'} if $(u \in U)$ U.Remove(u);
{09'} if $(g(u) \neq rhs(u))$ U.Insert(u, CalculateKey(u));

procedure ComputeShortestPath()
{10'} while (U.TopKey() $\dot{<}$ CalculateKey(s_{start}) OR $rhs(s_{start}) \neq g(s_{start})$)
{11'} $k_{old} =$ U.TopKey();
{12'} $u =$ U.Pop();
{13'} if ($k_{old} \dot{<}$ CalculateKey(u))
{14'} U.Insert(u, CalculateKey(u));
{15'} else if $(g(u) > rhs(u))$
{16'} $g(u) = rhs(u)$;
{17'} for all $s \in Pred(u)$ UpdateVertex(s);
{18'} else
{19'} $g(u) = \infty$;
{20'} for all $s \in Pred(u) \cup \{u\}$ UpdateVertex(s);

procedure Main()
{21'} $s_{last} = s_{start}$;
{22'} Initialize();
{23'} ComputeShortestPath();
{24'} while $(s_{start} \neq s_{goal})$
{25'} /* if $(g(s_{start}) = \infty)$ then there is no known path */
{26'} $s_{start} = \arg\min_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$;
{27'} Move to s_{start};
{28'} Scan graph for changed edge costs;
{29'} if any edge costs changed
{30'} $k_m = k_m + h(s_{last}, s_{start})$;
{31'} $s_{last} = s_{start}$;
{32'} for all directed edges (u, v) with changed edge costs
{33'} Update the edge cost $c(u, v)$;
{34'} UpdateVertex(u);
{35'} ComputeShortestPath();

Potential Field Methods

Many other methods aim at capturing the connectivity of the free c-space in a graph. Potential field methods offer an alternative where the robot is modeled as a particle under the influence of an artificial potential field. The potential field U is built such

that forces from obstacles repulse the particle and goal points attract it. The problem is solved with gradient descent. A common problem is that the particle might get stuck in local minima. There are however methods for overcoming this difficulty. An example of the potential-field method can be seen in Figure 2.17. The figure illustrates how a complete field is assembled by one function for the goal, where the goal is at the bottom of the function surface. This will yield attractive forces towards the goal and repulsive forces from obstacles. In Equation 2.6 the equation for the field is presented. U_{att} is the attractive field function and U_{rep} is the repulsive function. They depend on the position in space q. Equation 2.7 shows that the direction in which the path will head is the negative gradient of the field function for a certain position. The force vector is denoted $\overrightarrow{F}(q)$ and $\overrightarrow{\nabla}U(q)$ is the gradient of the potential field at the position q. Finding the path can be performed by using the optimization method gradient descent [*Potential Field Methods*].

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{2.6}$$

$$\overrightarrow{F}(q) = -\overrightarrow{\nabla}U(q) \tag{2.7}$$

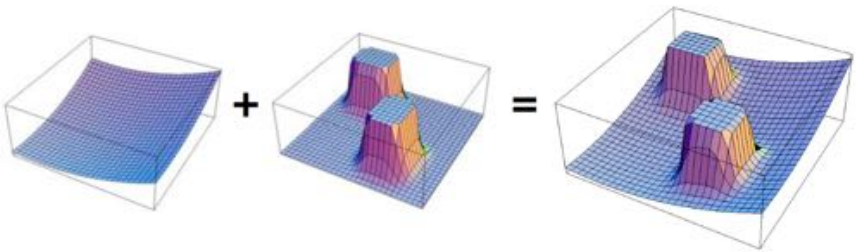


Figure 2.17 Illustration of potential-field method, [*Potential Field Methods*]

3

Method

This chapter includes comparisons between various theories, methods and hardware together with a small discussion on why certain programs, sensors and tools were used. Implementations of suitable methods and theories from Chapter 2 are also described.

First the software used are described and discussed, followed by the mapping sensors, an overview of the mode used for autonomous walk and a discussion of the sampling frequency. Thereafter positioning, mapping, planning and search problems of the hexapod are investigated and solved. Lastly, the changes of the remote control, the terrain handling problem and changes of the electrical assembly are described.

3.1 Simulink Model

The Simulink model developed in this project has its origin in the previous thesis by Dan Thilderkvist and Sebastian Svensson. It consists of three main blocks called modeSwitch, Control and PosArdu. Figure 3.1 shows the top layer of the Simulink model and the connection between the three main blocks. ModeSwitch decides when to switch mode and what should be done when a mode switches. This means that the camera S-function, the planning algorithms and the movement changes of the sensor mode are all included in this block. The output of the modeSwitch block is the new movement signal. More on the different parts of modeSwitch are explained below. The control block contains the locomotion controls which includes which leg to move, how it should be moved and how the rest of the body should follow. This is for example where the force sensors have been integrated. A lot of the changes since the last master's thesis have been done in the subblock called TrajectorySystem in MainControllerNew. The TrajectorySystem block creates the different positions for the leg trajectory and therefore this also contains the extended trajectory showed in Chapter 3. From the control block the different positions are sent to the third main block, called PosArdu and the mission of this block is to take the leg positions and forward them to the servos. The function and construction of

the control block and the PosArdu block are explained in more detail in the previous master's thesis by Dan Thilderkvist and Sebastian Svensson [Thilderkvist and Svensson, 2015].

modeSwitch

The block modeSwitch is, as mentioned previously, the block where mode changes are made. As can be seen in Figure 3.2, it consists of five sub-blocks called getCamera, checkStopSignal, checkMode, pathPlanning and finally changeMode. The inputs are the signal from the remote control, position and angle of the hexapod, goal position of the path-planning and a signal called noGroundStopSignal.

NoGroundStopSignal is used in mode 3, sensor mode, to check whether the active foot has ground contact at the end of the trajectory. If the sensors do not register any pressure the noGroundStopSignal becomes true and the movement is reversed.

getCamera The block getCamera basically gets the information from the camera and sends this information to the path planning block where this information is handled. The other output from the getCamera block is used in mode 2 where the robot is moving straight forward until the camera registers an obstacle in front of the camera. This signal simply becomes 1 if there is an obstacle and 0 if the space is clear.

checkStopSignal The only thing this block does is to change mode to mode 4 if the robot is in mode 2 and if the stop signal from the previous block is 1.

checkMode This block uses the information from the buttons to check which mode is active. It also checks the signal called noGroundStopSignal to see if the sensors have indicated ground contact. This is only done in mode 3. Output from the checkMode block is the active mode and it is thereafter used to activate the path-planning algorithm, if the mode is 5, or to tell the final block to change the origin of the movements.

pathPlanning The pathPlanning block takes the information from the camera to make a map of the world, it uses the position and angle of the hexapod to decide where in the room the robot is at the moment, it looks at the input goalPosition to decide where the robot should move and it checks which mode the robot is in. Since the path-planning calculations together with the mapping is demanding it is preferred to only perform these calculations while in mode 5, planning mode.

changeMode The last sub-block in the switchMode block is used to actually change the mode. After this block the rest of the model gets information about how it should move. For example, if mode 1 is active, the information about the movement should be taken from the remote control but if mode 2, the camera mode, is active the movement should just be straight forward or stop, depending on the camera stop signal.

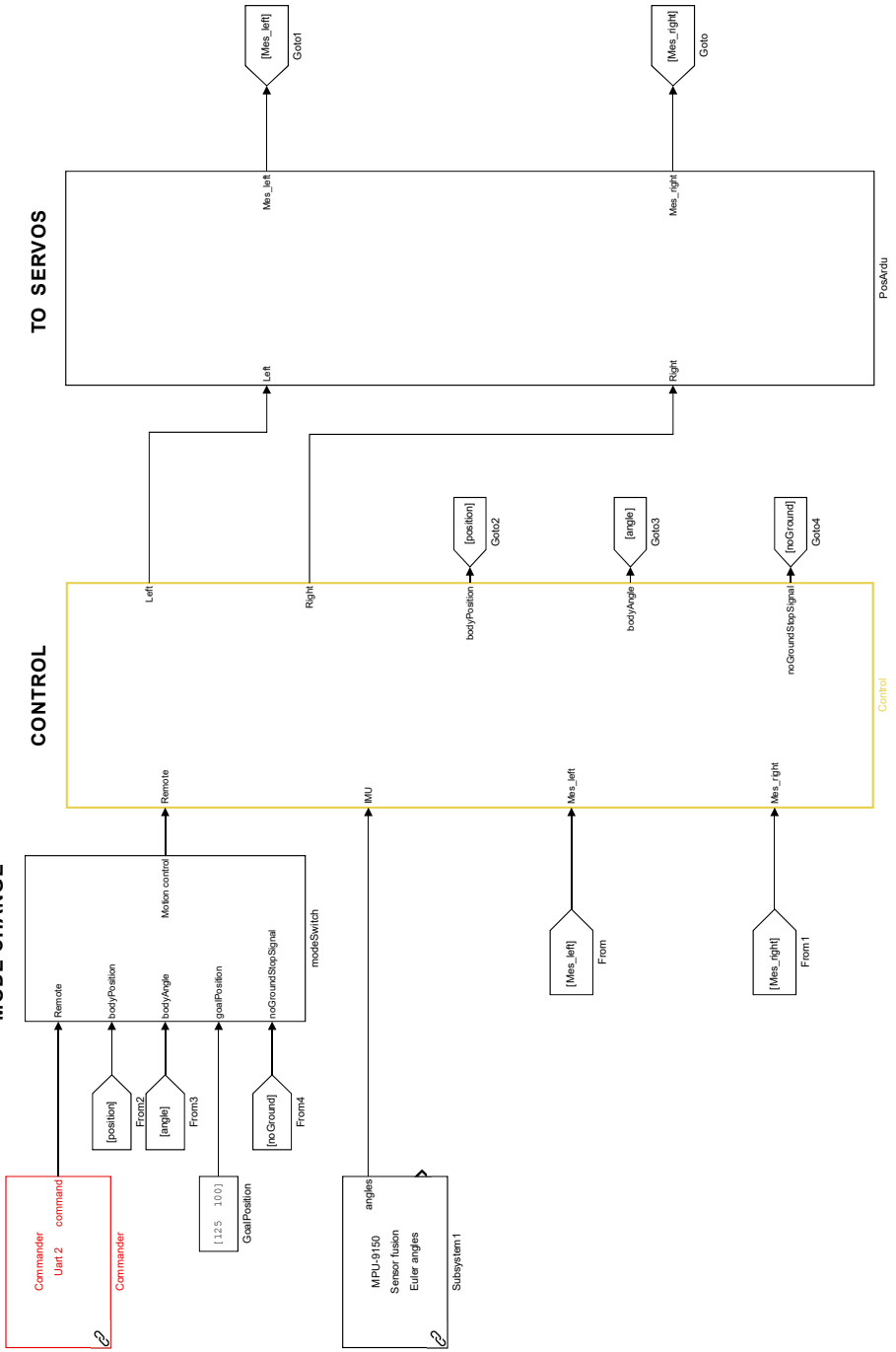


Figure 3.1 The top model of the system with the main blocks, modeSwitch, Control and PosArdu.

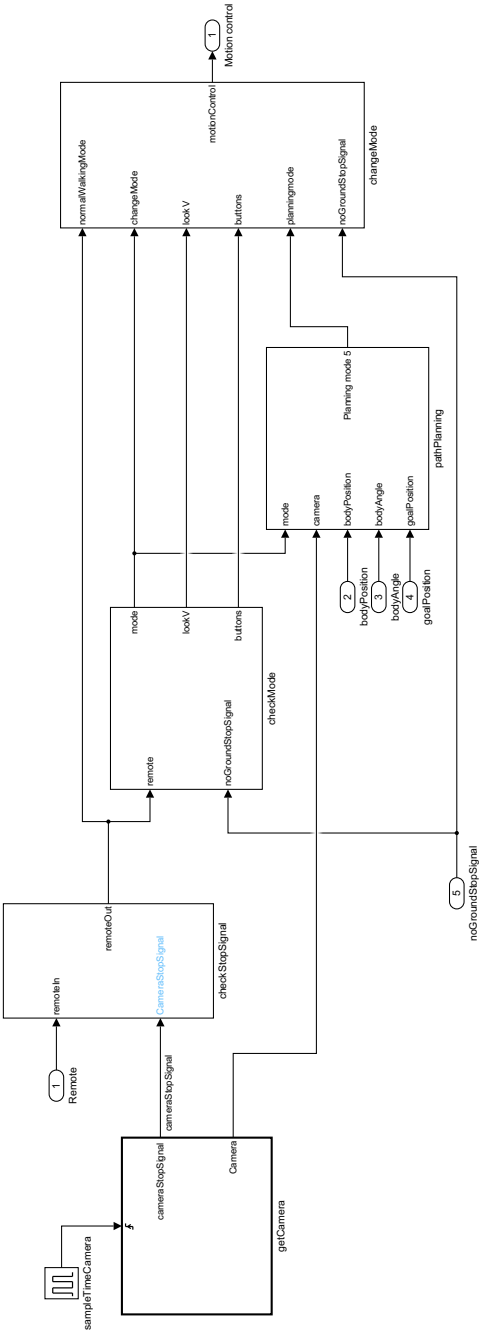


Figure 3.2 The construction of the block modeSwitch.

3.2 Friction Modeling in SimMechanics

A model of the robot was set up in SimMechanics according to Thilderkvist and Svensson [Thilderkvist and Svensson, 2015]. This model contained all mechanical parts of the robot complete with joints and the control system could be run on the model. However this could only verify the mechanical system isolated from an external world. Since a goal of this thesis was to navigate in an environment with obstacles and terrain another method or software had to be used. A first step was to improve the SimMechanic model with contact and friction forces between the feet of the hexapod and the ground. Since there was poor support for contact forces and friction in the native SimMechanics toolboxes an external package was used [*SimScape Multibody Contact Forces Library*]. The toolbox allowed for friction and contact force model blocks to be set up between the floor and the feet. This allowed for the Hexapod model to walk around on a flat surface.

3.3 Choice of Simulation Software

Since the solutions for modeling interactions between the robot and environment was cumbersome in SimMechanics, other alternatives for simulating robots were researched. Gazebo, MuJoCo, V-REP and Webots are all popular programs for simulating multi-body dynamics, force contacts and friction through their physic engines. Having support for CAD-parts and capabilities for simulating a virtual time-of-flight camera were also important features. They all have similar capabilities and either one of the programs seemed to work for our purposes. Since code generation to the hardware is performed from the Simulink environment simulink also had to be used when running simulations. None of the simulation programs had an API for Simulink connection, but they all had different types of Matlab API's. In the end we chose to go with V-REP because of its convenient Matlab API and free academic license.

3.4 Setting Up the Hexapod in V-REP

All of the individual parts of the Hexapod were provided in the form of CAD-models. These had to be converted into STL file format in order for them to be importable into V-REP. This meant that all parts of the hexapod had the correct weights and moment of inertia. The parts were also dynamically enabled in order for simulations to include collisions, falls and for gravity to act upon the parts. All parts were also enabled for sensor interaction and camera rendering so they could be captured by the depth-camera. The robot contains 18 leg parts and one solid body part. All of the parts were assembled with joints according to Thilderkvist and Svensson [Thilderkvist and Svensson, 2015].

Virtual Depth Camera

V-REP includes virtual sensors so the depth-camera can be simulated and send a depth matrix back to Matlab. Thereby the information between V-REP and Matlab will replicate the information which is to be sent between the BeagleBone and the camera. The resolution of the real camera used is 320x240 but unfortunately V-REP only supports 1:1 aspect ratios, therefore 240x240 was used in V-REP. The field of view and focal lengths of the real camera was used with the virtual one. The camera was mounted and the final V-REP assembly of the hexapod is presented in Figure 3.3.

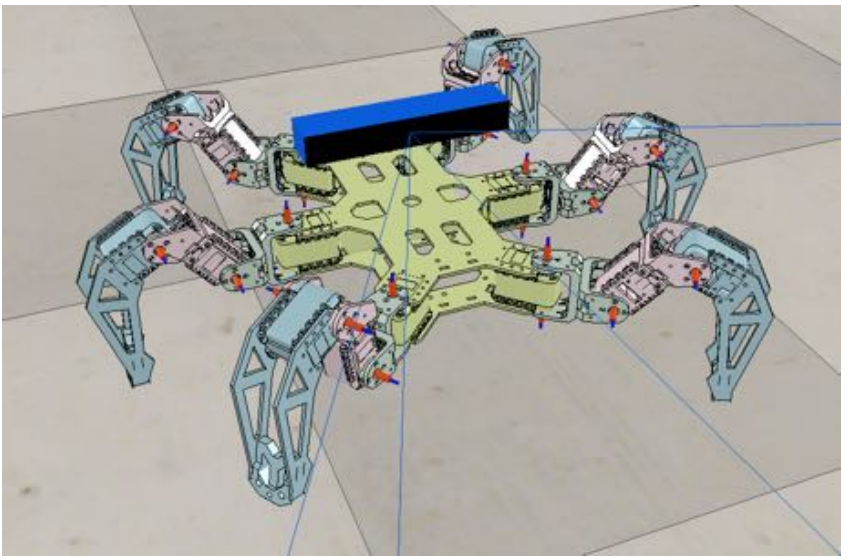


Figure 3.3 Assembled hexapod in V-REP

Modeling Dynamic Behavior in V-REP

Static shapes will not be influenced by gravity or dynamically enabled joints, while dynamic shapes will. Respondable shapes will be affected by other respondable shapes in collision. The computations of dynamic and respondable parts are heavy, primitive shapes were therefore created to represent the complete CAD-parts. All dynamic and collision computations are performed on the simple parts while the original parts are used in the rendering of the visualization. The tibia parts of the legs are first selected and a convex hull shape created. This shape is to be used in the collision computations. It is connected to the corresponding part and collision masks are set so there will be no collisions between connected parts of the leg [*Designing dynamic simulations*].

Connect to V-REP with Matlab and Simulink via Remote API

In order to connect Matlab to V-REP the correct paths has to be included in Matlab and the right port number has to be found [*Enabling the Remote API - client side*]. The API has a range of methods which can be applied to the V-REP object. With these functions angles for the servos can be sent to the V-REP simulation based on the Simulink controller. The virtual vision sensor will perceive the built environment and send the depth Matrix to Simulink. Another piece of information which is sent from V-REP to Simulink is the global position and orientation of the hexapod in the environment. The information flow between Simulink and V-REP was attempted to keep close to the flow between the BeagleBone and Simulink, see Figure 3.4 for an illustration of the flow of information.

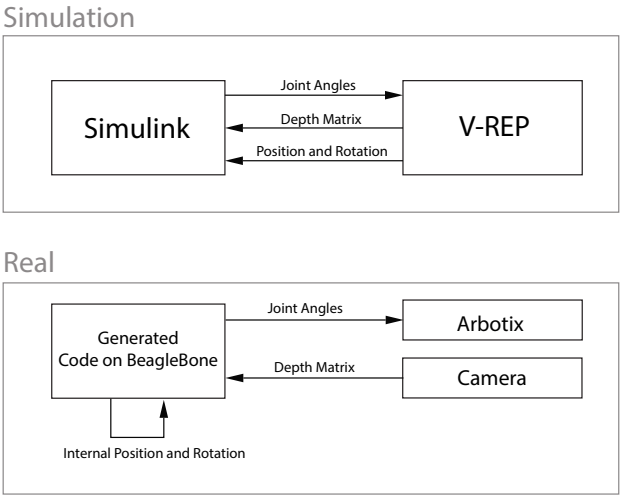


Figure 3.4 Information flow simulation compared to real robot

The coder in Matlab/Simulink does not support certain kinds of functions. The V-REP remote API could therefore not be run directly in the Matlab-function block but calls to external functions containing the V-REP connection code had to be made. A preference was to run V-REP in real-time, therefore we had to run Simulink in real-time as well in order for the commands sent to V-REP to be useful. A real-time synchronization block was added to the Simulink-model. The communication between Simulink and V-REP will by default take place asynchronously. Data will be sent irrespective of where in the simulation calculations V-REP is. Blocking function calls has to be used in order for the communication to take place synchronously. That is, Simulink will send data to V-REP and wait until the calculations have been performed. V-REP will then send data back to Simulink [*Remote API modus operandi*]. Running the simulations synchronously is therefore more

time consuming. It can be important to use synchronous execution when dynamics are of importance. In this case it was preferred to have faster execution.

Servo Joint Controllers

Movement of the hexapod is performed by controlling the joints of the servos. On the real robot an angle value is sent to the servos and an internal PID controller moves the joints [Thilderkvist and Svensson, 2015]. The same behavior is preferable in the virtual simulation environment in order for the main controller to send the same information in both the real case and the virtual. Luckily V-REP has built-in PID controllers in the joint objects which were to be used for this purpose. The maximum torques are 16.5 kgcm for the real servos, and the virtual torque limit was therefore set to the same. Since the real servo PID parameters were unknown the virtual PID parameters were found by trying to compare the behavior in simulation with the real behavior. The real servos never had any difficulties following the created trajectories. The PID controller in simulation was therefore tuned to also follow the trajectories.

3.5 Mapping sensors

In order for the robot to navigate autonomously in the world, some sort of mapping sensor is necessary. There is a myriad of sensors which can be used to perceive the environment but a Time-of-Flight camera was eventually used for convenience.

Time-of-Flight Camera

The TOF camera chosen in this project is an Asus Xtion Pro Live. It features among other things an infrared (IR) emitter and detector that captures the depth image and it features a RGB camera. The two different features are separated in two matrices when delivered. It is small and light weighted and therefor works fine on the hexapod [*Xtion PRO LIVE*].

Another 3D sensing camera considered was Microsoft's Kinect for Windows. The two cameras are much alike but some of the differences are explained in Table 3.1.

Table 3.1 Comparison between ASUS Xtion PRO LIVE and Microsoft Kinect

	ASUS Xtion PRO LIVE	**Microsoft Kinect**
Product Dimensions	18 x 3.5 x 5 cm	38.1 x 38.1 x 12.4 cm
Weight	0.170 kg	1.09 kg
Depth Space Range	Between 0.8m and 3.5m	Between 0.8m and 4m
Field of View	58°H, 45°V	57°H, 43°V
Frame rate	640 x 480: 30 FPS	640 x 480: 30 FPS
	320 x 240: 60 FPS	
Power Supply	USB2.0/3.0 interface	External
RGB Resulution	1280x1024	1280x960
Features	RGB camera	RGB camera
	Infrared (IR) emitter	Infrared (IR) emitter
	IR depth sensor	IR depth sensor
	Microphone	Multi-array microphone
		3-axis accelerometer
		Tilt motor with a tilt range of $\pm 27°$.

Additional comments for the Kinect [*Kinect for Windows Sensor Components and Specifications*]:

- Impossibility to lower the resolution.
- Possibility to use the accelerometer to determine the current orientation of the Kinect.
- Possibility to record audio as well as find the location of the sound source and the direction of the audio wave.

The reason why the ASUS camera was finally chosen was mainly because of the weight and size of the camera and the fact that the power was supplied via the USB cable. An image of the camera is presented in Figure 3.5.

Figure 3.5 ASUS Xtion Pro Live

3.6 Camera Implementation

Since Mathworks does not support use for Asus Xtion Pro Live OpenNi was needed in order to receive the depth images from the RGB-D camera [Gustafzelius, 2015]. OpenNI is a C++ open source SDK used for 3D sensing applications [*OpenNI 2 SDK Binaries & Docs*]. The communication between the camera and Matlab generated some problems, mainly because of the fact that Matlab does not support code generation to C from C++ code. The problems were solved with help from a previous thesis from Combine [Gustafzelius, 2015] and personal consultation from Sara Gustafzelius, the author of the thesis. A C-wrapper which interfaces the C++ methods was written and thereafter used to create an S-function C-file with Matlab's built-in legacy code commands. An *.so* library file was also needed, both on the target hardware and on the host computer, in order to solve the communication problem. In the user manual belonging to this project a small tutorial describes how to get the connection between the camera and Matlab to work [Eriksson and Malmros, 2016].

Integrate camera code to BBB code

When the camera was up and running the model needed to be integrated to the simulink model for the rest of the project. The camera code generated from

the S-function contained one .tlc-file, one .c-file, one .mex64-file and finally one .rtwmakecfg-file. The rest of the project also contained these files but in total the model could only have one rtwmakecfg-file. Therefore the two rtwmakecfg-files needed to be manually integrated in each other so that the camera block thereafter could be placed into the rest of the model and data from the camera could be taken.

3.7 Overview of Autonomous Mode

Autonomous mode runs the hexapod with information from the camera, plans a path from current position to a goal position and executes the path with the leg controller. The planning and mapping is dynamic and will therefore avoid new obstacles and re-plan the path when obstacles have been removed.

A number of different parts have to work together in order for the hexapod to move from one position to another. Figure 3.6 presents an overview of the different operational parts which have to work together in order for the hexapod to move around in the real world autonomously. The three colored areas denote different functional aspects of the autonomous mode.

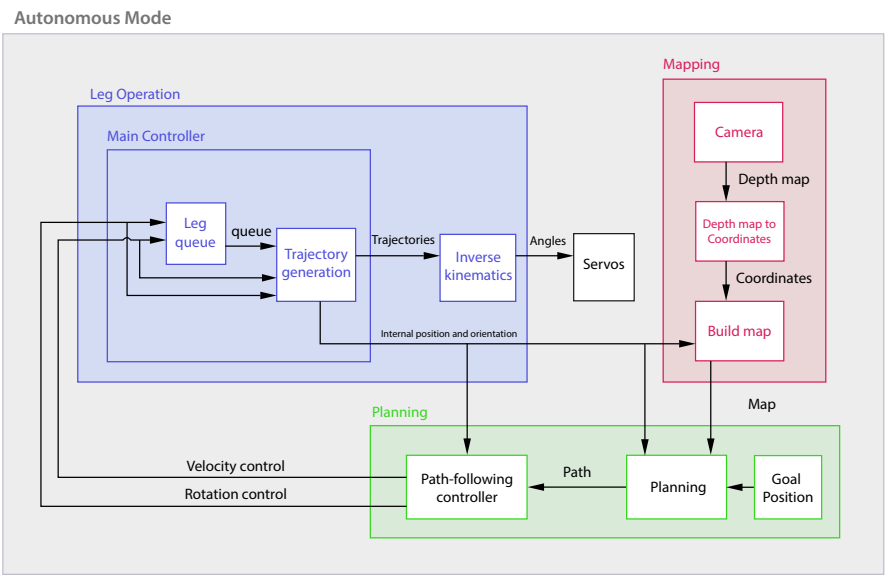


Figure 3.6 Functional overview of hexapod operation

Leg Operation The blue area contains the control and calculations for the legs. The inputs into the blue block are two signals. One signal for controlling the straight forward velocity of the hexapod and the other for controlling the rotation. An op-

tion could have been to include another control signal for controlling the sideways movement. But the sideways signal was left out since it provided small practical advantages but would have added complexity to the path-following controller. The leg queue block orders the legs with regards to which legs are the farthest behind the others when it comes to following the velocity signal. The leg farthest behind will be put first in the queue and the following legs in subsequent order. This queue is passed to the trajectory generation where discretized trajectories are created for all feet. These trajectories are based on the current position of all hexapod feet and their individual goal positions. The goal positions are calculated based on velocity and rotation control signals. The feet trajectories are fed to the inverse kinematics block. This block calculates the angles of all joints for the feet positions calculated in the trajectory generation. These angles are then fed to the servos via the Arbotix-card. The actual movement is then performed by the internal PID-controllers in the Dynamixel-servos [Thilderkvist and Svensson, 2015].

Mapping The red block contains all of the mapping calculations. It starts with a connection to the ASUS depth camera. This provides a depth map which essentially is a matrix with depth values in every pixel. The depth map is sent to the coordinate transform block. This block uses information about the intrinsic parameters of the camera in addition to the depth map to create a point cloud. The point cloud is essentially a set of points in space which consists of x, y and z values. The point cloud is cleaned by removing the floor which is assumed to be a plane. Reflections in windows can produce points which are unreasonably far away are removed. The set of points are then sent to the map building block. This block combines the information about obstacle coordinates in space and the robot position and rotation in a map. Obstacles are added and removed from the map depending on the points. A safety area is also added around all obstacles, in order for the hexapod to have a margin to avoid collisions.

Planning and Path-Following The green block performs all the planning operations where the goal position is set by the user. The planning block then uses the map with start and goal position to plan a path from the current position to the goal. Planning is performed by building a road map from the discretized grid map. All adjoining cells in the map are connected in the road map which is essentially a graph. An incremental heuristic search algorithm called D* lite is used to find a path in the road map. The path shows which cells to traverse in order to reach the goal as fast as possible. In order to follow the path, it is sent to the path-following controller. Each way-point is transformed into continuous coordinates. The controller compares the current position and rotation to the direction and distance of the next way-point in the path. Errors in direction and distance from way-point are used to calculate control signals. The controller used is essentially a proportional controller where the forward control signal is blocked if the rotational error is too large. Forward velocity and rotation control signals are sent into the leg block and the feedback loop is closed.

3.8 Real-Time Multi-threading

Simulink includes functionality for running multiple threads on one core. This is called multitasking operation in Simulink. All blocks running at the same sampling frequency will be running as the same task. By default the priorities of the threads are set so the fastest sampled thread will have the highest priority. The code generation takes care of setting up threads and includes a scheduler [*Simulink Coder User's guide*].

Different parts of the code are of varying importance and different sample frequencies are needed for robust operation. In Table 3.2 the sampling frequencies for different parts of the code are presented. The leg operations have to be performed at a high sampling frequency for smooth operation of the legs. The leg operations are set to highest priority since they are more important than the other blocks. More thorough testing of the sample frequency of the leg operations is performed in [Thilderkvist and Svensson, 2015]. Based on tests and [Thilderkvist and Svensson, 2015] the leg operation can be seen to possess a bit less than half of the computational power of the BeagleBone. The calculations of mapping and planning take approximately half a second to perform. The execution time of the path-following is negligible compared to the leg, mapping and planning calculations. The sample rate of mapping and planning was set to one sample per second. With good multi-tasking behavior the computational power should suffice to run the hexapod without interruptions. The one-second sampling time for mapping and planning should suffice for dynamic re-planning to work quickly enough.

Process	Sampling frequency [Hz]
Leg operation	40
Mapping and Planning	1
Path-following	10

Table 3.2 Sampling frequencies

3.9 Positioning of the Hexapod

Inertial Measurement Unit

To calculate the positioning of the hexapod, different methods where analyzed. Since the robot already had an IMU integrated it would be convenient if the accelerometer in the IMU could be used to estimate the position of the hexapod in the map. Unfortunately, the measurements were noisy and hard to read. A low-pass

filter was implemented as an attempt to make this better. Thereafter the filtered signals were fed through double integrators in order to estimate the position out of the acceleration but large drift in the position was observed. This is to be expected when double integrators are applied since tiny errors in mounting of the IMU and noise might yield large effects.

Internal velocity for positioning

As it would be time consuming to get the accelerometer to work properly. Therefore, it was decided to look for other methods. A simple heuristic way used for localization was to assume there is no uncertainty in the actuation of the controller. This way the trajectory for the feet will provide the position of the robot for each time step.

Point Cloud Processing

The planning is performed in 2 dimensions and all traversable surfaces are required to be flat. In order to build a map which represents all non-traversable areas as obstacles, the floor is to be found in the point cloud. With information about the orientation of the floor all points deviating more than a pre-determined threshold can be seen as obstacles. There are numerous ways of fitting a plane to a set of points in space of which a simple method is using least squares regression. This method will fit a plane to all points in space with a quadratic loss function. A result of this will be that points farther from the fitting plane will influence the orientation more than those close to it. The fitted plane will therefore be extremely biased and unpredictable unless the points are evenly distributed around the plane, for example, a Gaussian distribution. This case was highly unlikely, if not impossible, another method taking into account a vast amount of outliers therefore has to be used.

Random Consensus Optimization (RANSOP)

The RANSOP method was developed and tested in V-REP based on the assumption that the point clouds generated would be similar to those from the real camera. It was done by mounting the camera in a known position on the robot where the possible movements of the robot are approximately known. This yields approximate information of the position and orientation of the floor plane related to the camera. This extra information was used to improve the RANSOP algorithm for these purposes. The goal was to reduce the computational power needed since the algorithms are run in real-time. Another goal was to modify the algorithm so information about the initial guess of the plane could be used for better results. The algorithm was only to find the floor plane and no others. Unfortunately, the floor from the real camera was noisy and mostly incomplete. The use of RANSOP on the real robot therefore had to be abandoned.

3.10 Mapping and Planning Overview

The mapping and planning methods are tightly connected since the planning algorithm will have to plan in the map. In other words, they will have to be compatible and chosen in tandem. A first realization is that saving all points of the point cloud every sample will generate large amounts of data. However, it will not contribute with significant amount of extra information since many static objects will yield points in the same position. An interesting reasoning regarding the memory needs are discussed in the master's thesis by Sara Gustafzelius [Gustafzelius, 2015], where data grow quickly if no simplification or reduction is performed. Therefore, a method for modeling the world in a more simple way is needed.

2D Mapping

Assumptions and simplifications of the mapping problem had to be made in order for the computational power to be sufficient. The first assumption is flat floor operation. The hexapod only works on a flat surface in automatic mode and is not able to traverse differences in terrain height. All objects which protrude from the ground more than a set distance, are assumed to be non-traversable obstacles. The map is therefore built in 2D, a discretized occupancy grid map. The trade-off was between having a fine resolution map and computational efficiency when adding and especially removing obstacles. The map size was set to 400x400 cells with each cell having a side length of 0.05 meters. This covers an area of 20x20 meters which was deemed sufficient for these purposes. A matrix is used to represent the map, where occupied cells contain a one and free cells a zero. An example of an occupancy grid map is presented in Figure 3.7.

0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	1	1	1	1	0	0	0
0	0	0	1	1	0	0	0	0
0	0	1	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Figure 3.7 Occupancy grid map example

The map is built by projecting every point in the point cloud onto the floor plane. The cells which contain a point will be regarded as obstacles. In order for

a point to be added to the map additional conditions must apply. The point must be more than a certain distance from the floor and less than another distance, the reason being that the hexapod height is below certain obstacles. Before actually adding any obstacles to the map the location and rotation of the hexapod has to be taken into consideration. Those are acquired by the internal positioning for the real robot and can be measured directly in V-REP. Coordinates of the point cloud are first calculated in the coordinate system of the camera called the local coordinate system. The initial position of the robot is set to the center of the map and the orientation origin in a direction to the right. The point cloud is then translated and rotated based on the robots position and rotation in the global coordinate system. An illustration of the global mapping process is presented in Figure 3.8.

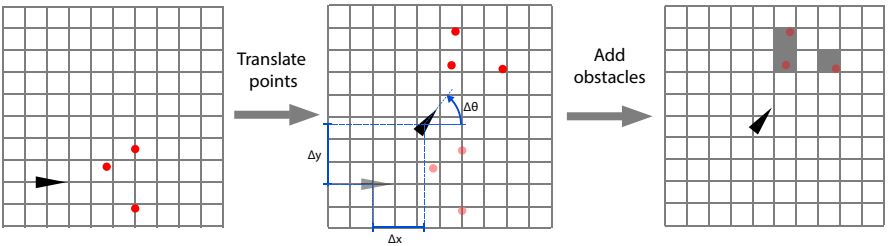


Figure 3.8 Global obstacle point transformation

In order to make the planning more efficient the size of the cells were doubled to 0.1 m side length. In the planning environment the robot is modeled as a point but in reality when standing in the standard pose it has an approximate length of 45 cm and a width of 50 cm. One way of solving this is to add a margin to all added obstacles in the planning map. The transition between the regular map and the courser planning map with margins is presented in Figure 3.9. The purple square is a point in the regular map, which is added as the dark gray area in the planning map. To the planning map a margin of three cells are added in all directions in order to cover worst case scenarios.

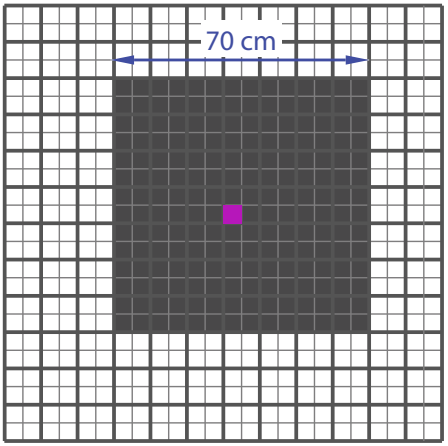


Figure 3.9 One cell obstacle added to planning map

Mapping Dynamic Environment

Adding obstacles to the map is fairly straight forward. But in order for the mapping to work in a dynamic world obstacles have to be removed from the map as well.

A simple heuristic method for dynamic mapping was however created which does not apply any probabilistic methods and always keeps the map which has been seen. All obstacles which have been seen are, as previously discussed, added to the map. The problem is how to know which obstacles to remove from the map. Adding obstacles is a simple task since there is a point in space representing the object. However, when removing an obstacle the information is only absence of a point. It is difficult to interpret when absences of obstacles warrant a removal of an obstacle from the map. The method used is illustrated in Figure 3.10. The magenta obstacles represent obstacles in the old map. The obstacles are removed by iterating from each new obstacle, green, to the robot and all obstacles in the path are removed from the old map. New obstacles are then added to the map.

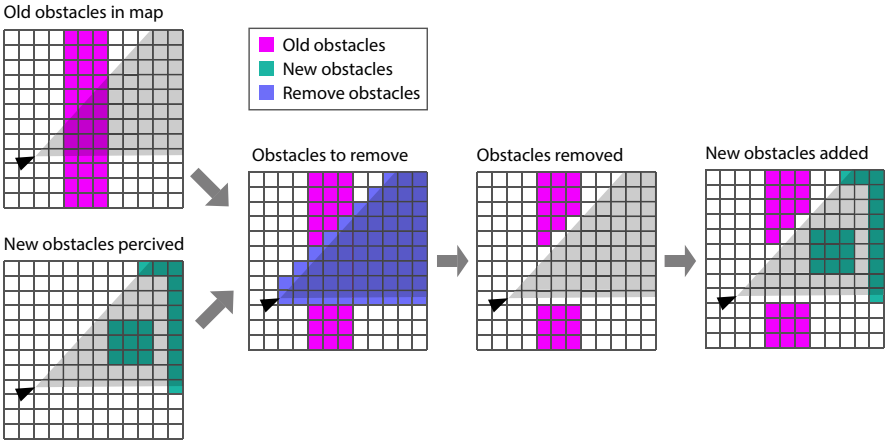


Figure 3.10 Process of removing obstacles

3.11 Planning

In the methods used to perform planning and actions to follow the plan, the planning and actuation have been completely decoupled. Meaning that the planning has not taken the limitations of actuating the plan into account. The reasoning is that since the robot can rotate and walk in any direction it will not be limited by differential constraints in the same way other vehicles might. Therefore the path will be followed in a correct manner. No dynamics are therefore included in the planning algorithms since this would make the planning problem harder and less suited for real-time operation [Lavalle, 2006].

C-space The configuration space in the case of the hexapod is defined with three parameters, position along x-axis, y-axis and rotation, θ. The c-space is the set of all possible positions and rotations over the free space.

Uniform Grid Discretization Uniform grid discretization was used to build the road map in which planning was performed. The main reason being that it is a simple intuitive method. For comparisons and discussion regarding alternative planning and mapping methods, see the discussion part in the thesis.

3.12 Search

Uniform grid discretization provides a solid road map, but in order to actually find a path, search has to be performed in the graph. D* Lite was chosen as search algorithm since it is efficient and dynamic.

D* implementation

An implementation of D* Lite [Koenig and Likhachev, 2002] was found in C++ [*D* Lite C++ Implementation*]. Code in C++ can not be directly used in code generation and therefore a wrapper was written in C. This allowed us to finally run the planning algorithm on the BeagleBone hardware. In Figure 3.11 an example of D* Lite is presented. Obstacles were added step by step while the path was re-planned. The colors are denoted as follows.

Black Traversable free space

Green Traversable with changed cost

Purple Path and goal cell

Red Untraversable obstacle

Yellow Start cell

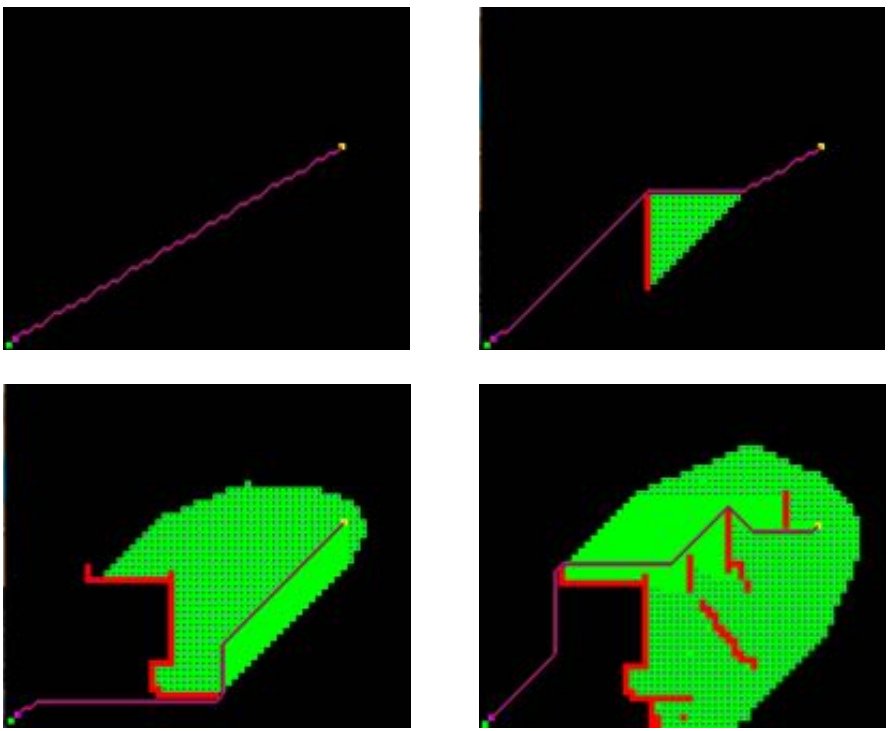


Figure 3.11 Illustration of the D* Lite implementation

Integrating the path planning code with the rest of the model

Once the path planning was done and worked satisfyingly it needed to be integrated with the rest of the model run on the hexapod. The procedure of this integration had a lot of resemblances with the integration of the camera code. Also here a wrapper was needed to be able to generate code to C from C++ and an *.so* library was needed to be created on the BeagleBone to get a symbolic link to the planning code. When the D* S-function was compiled in the model the new rtwmakecfg-file needed to be integrated to the old file in the same way the rwtmakecfg-file for the camera was integrated before. Lastly the simulink block for the D* S-function was created and placed in the model together with the rest of the planning code. In Figure 3.12 an example of a path created in the Simulink/Matlab environment with the C++ D* Lite code is presented.

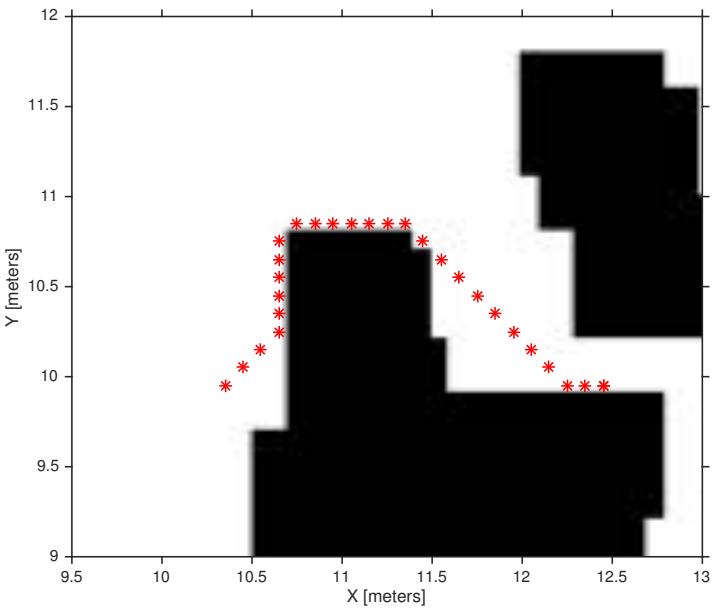


Figure 3.12 Illustration of a planned path

3.13 Path Following and Feedback

Planning using feedback has two separate approaches [Lavalle, 2006]. The explicit approach is when the possible future state drift uncertainty is taken into account in the planning. While the implicit approach says that no uncertainty is possible in the

states. But a feedback plan is present which provides the actions to take in order to revert back to the plan if it ends up in an unexpected state. Designing systems with the explicit approach is more complicated, complexity is larger and algorithms more difficult to implement. We are using the implicit approach where planning is performed by not taking into account the uncertainty in states, but instead corrects for uncertainties with a control law.

Actuation for Path Following

There are three input signals which can be used for actuating the robot. These correspond to walking forward, walking sideways and rotating. The path which is to be followed is a set of points, that is coordinates (x,y). An example of a path which is to be followed is presented in Figure 3.12. The issue of following the path can be seen as a control problem where the control signals have to be used in such a way that the robot follows the path efficiently. The error in the control problem can be seen as the distance between the robot and the next way-point in the path. In order to follow the path as smoothly as possible a couple of factors have to be taken into account. First of all a decision has to be made by the robot regarding which way-point to aim for. If a way-point to close is chosen the walking will be jerky and change direction frequently and quickly. However, if a way-point goal is set too far away the robot might cut corners and collide with obstacles. Another possibility is to use numerous way-points at the same time so adjustments can be made for direction of the next way-point even before the current one has been reached. This would add a large amount of complexity and the controller was kept simple by just aiming for one way-point at the time. Since no differential constraints are imposed on the hexapod it can walk and rotate in the direction necessary for following the path. A simple controller could therefore be used. It works by first checking if the current way-point is closer than a set threshold. If so, choose the next point in the path. If not, keep the current way-point as the goal. Then find the difference, ε_θ, in direction between the current direction and the direction of the way-point. Use the difference as the error in a proportional controller to minimize the direction error. If this direction error is below a magnitude threshold, walk forward. The error distance, ε_d is used to decide when the way-point is close enough to start aiming for the next way-point. In Figure 3.13 the errors used by the path-following controller are used.

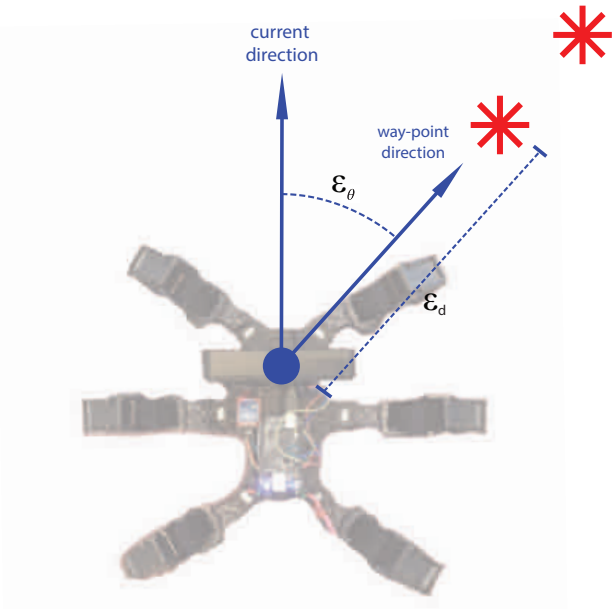


Figure 3.13 Illustration of the errors used by the path-following controller

3.14 Remote Control Changes

The remote control from the previous master's thesis [Thilderkvist and Svensson, 2015] uses 8 out of 12 buttons. Some of them worked fine and others were just for experiments and did not work perfectly in the beginning of this project. On the remote control there are 6 small buttons called R1, R2, R3, L4, L5 and L6, see Figure 3.14. These are used to change the mode of the Hexapod. L6 is used for mode 1 which means normal walking controlled by the user. L5 was not working from the previous master thesis and was therefore changed in this project to work as a first try of an automatic mode. This means that when L5 is being pressed, then the user can no longer use the remote to control the Hexapod. Instead the Hexapod walks straight forward and stops whenever the camera sees an object near the robot. This is called mode 2. When the camera sees an obstacle it sets a stop signal to 1. This signal switches mode to mode 4 which corresponds to button R3. In theory this means that the button R3 is being pushed and the robot stops. When the obstacle disappears the mode is going back to mode 2 and the robot starts to move.

L4, which corresponds to mode 3, is like mode 1 except the sensors are being used to check if the legs have touched ground or not. This means that the hexapod is controlled with the remote control as long as the sensors does not indicate that there is ground under the feet. In that case it starts to move in the opposite direction for a

couple of samples to avoid falling of an edge. Mode 3 is described more below.

The most interesting mode is mode 5, corresponding to button R2. When this button is being pushed the robot goes into path planning mode. It will now walk according to the mapping and planning calculations and avoid obstacles.

The last button R1 (mode 6) was used as a balancing mode in the previous master thesis and has not been changed since then. This mode uses the gyroscope from the IMU to balance the body of the hexapod.

In Table 3.3 a summary of the different buttons on the remote control can be viewed.

Figure 3.14 The remote control that is used to operate the hexapod.

Table 3.3 The different buttons with corresponding modes

Button	Mode	Description
L6	Mode 1	Normal walking mode
L5	Mode 2	Check for obstacle mode
L4	Mode 3	Sensor touch mode
R3	Mode 4	Obstacle stop mode
R2	Mode 5	Planning mode
R1	Mode 6	Balancing mode

3.15 Terrain Handling

Force Sensors

To get information of whether the feet of the Hexapod are in contact with the ground or not, several alternatives were considered. One of the first ideas was that the internal forces of the servos could be measured. The torque in the servos would depend on whether the foot had a force applied to it or not, that is, the servo must work harder when the foot is on the ground. Another idea was to use a small button which would be pushed in when the foot got contact and thereby trigger an I/O signal. This current could be measured and used as an indicator. The third idea that was considered and later chosen was to buy a couple of force sensing resistors.

Application of the Force Sensors

Once the force sensors were up and running the values could be measured in Matlab. It was found that the sensor was very sensitive and just a small pressure on the sensor showed a change in the values. Figure 3.15 shows the result that the company Interlinc Electronics got when they tested the sensors for different values of the added resistance, RM. Since the hexapod needed a sensitive sensor it was decided that a 10k Ohm resistor was going to be used, see Figure 2.2 from the theory section. Some simple tests were made that showed that the sensors worked according to the graph from Interlinc Electronics. A Matlab plot that shows the sensor in action can be viewed in Figure 3.16(a).

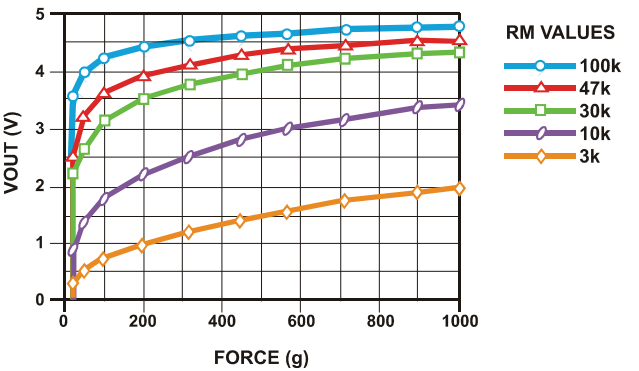
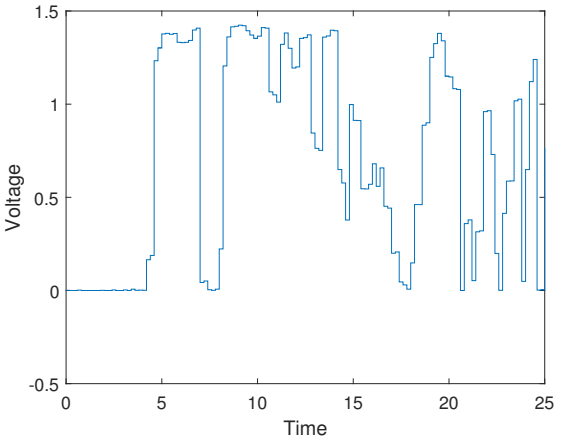
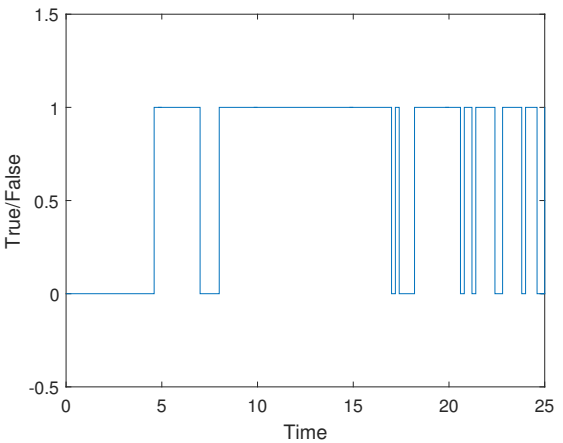


Figure 3.15 The voltage output from the sensor depending on the value of the added resistance, RM. [*FSR Force Sensing Resistor Integration Guide and Evaluation Parts Catalog*]



(a) The changes of the voltage.



(b) The boolean expression for ground contact

Figure 3.16 Force sensor example

The use of the sensor was decided to be relatively simple. If the voltage of the analog input was above 0.1 V the output of the function was set to true, otherwise it was set to false, see Figure 3.16(b). This boolean value was thereafter used in the code for the walking pattern to cancel the step if the foot was in contact with the ground, or to keep moving the leg until contact with the ground was found. In order to make these changes in the walking pattern the code needed to be supplemented with an extended moving trajectory for the leg. In the previous case the trajectory

was based on a half ellipse and hence, the trajectory was canceled when the leg had reached the end of the half ellipse. In order to make the leg keep moving a short vertical trajectory was added to the previous ellipse. The new trajectory is shown in Figure 3.17. In this case the leg will move down to 3 cm in negative z-direction if the sensor has not indicated ground touch. When the extended trajectory is done the leg will move back to its original position. Thereafter, to avoid falling off an edge, the hexapod will keep moving backwards for a couple of samples before it stops completely. This feature is only available in mode 3 to avoid disturbances of the sensors to conflict with the normal mode or the planning mode.

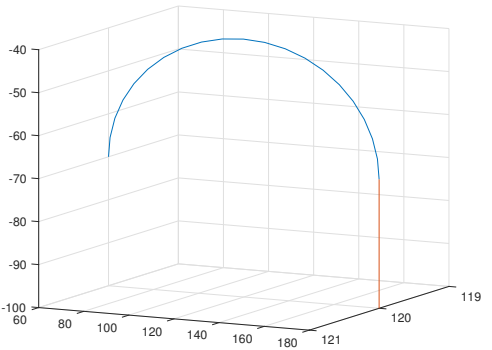


Figure 3.17 The old trajectory (blue) of the leg movement together with the extended trajectory (red). The start position is (120, 60, -70) and the goal of the ellipse is (120,180,-70).

Force Sensor State Flow

The state flow for the use of the force sensors are described in the diagrams in Figures 3.18 and 3.19.

When mode 3 is activated the hexapod will wait for instructions from the remote control and as soon as the hexapod starts to move it will check the sensor of the moving leg. If the sensor becomes active the leg will stop in the current position and next leg will start to move. If the sensor does not indicate that the leg has touched ground the state "Check for ground" will become active. The flow of this state is described in Figure 3.19. Here the leg can go through four states called moving state 0, moving state 1, moving state 2 and moving state 3. Moving state 0 corresponds to the normal trajectory that the leg is doing according to the old code from the master thesis by Sebastian Svensson and Dan Thilderkvist [Thilderkvist and Svensson, 2015] and moving state 1 corresponds to the new, extended trajectory, which is described in the previous section and Figure 3.17. This trajectory makes the leg move downwards for 3 cm to check if the ground can be reached or if there is an

edge. During moving state 0 and moving state 1 the leg movement can be canceled if the sensor check becomes true and in that case the next leg will start to move. If, however, the sensor check never becomes true, indicating no ground can be reached, the moving state 2 will become active. After this point the sensors will no longer be used. Moving state 2 and 3 are corresponding to 1 and 0 respectively. They use the same trajectory but inverted so that the leg moves backwards. The number of samples in the two back trajectories have also been changed for faster movement. When moving state 3 is done the leg will be in the exact same position as when it started to move in moving state 0. Thereafter the velocity of the hexapod is set to full speed backwards for a couple of samples before it completely stops. To activate the hexapod again and be able to control it with the remote control, the mode needs to be changed by pressing any of the six buttons showed in Table 3.3 and thereafter, if the force sensor mode is to be used, activate mode 3 again by pressing button L4.

At the moment the force sensors are only implemented on the two front legs and hence, this mode will only fork if the hexapod moves straight forward towards the edge.

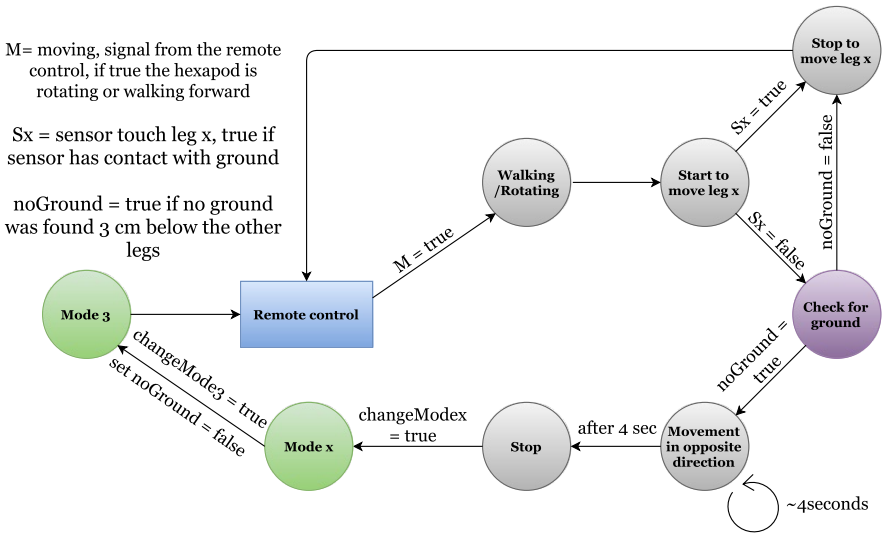


Figure 3.18 The state flow diagram for mode 3 where the force sensors are being integrated.

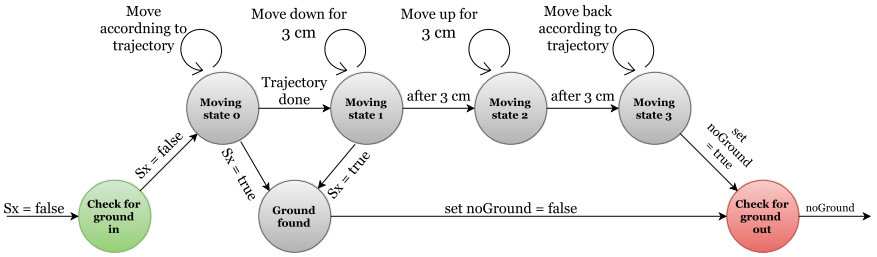


Figure 3.19 The state flow diagram for mode 3 when the state "check for ground" is activated.

Mounting of the Force Sensors

As mentioned before the sensors give a better precision if the force acting on them is perpendicular to the sensor. This requires that the mounting of the sensor catches the forces coming in with an angle and transform them to act better on the sensor. Another requirement is that the mounting does not create any internal forces on the sensors which might give inaccurate readings. Therefore, it is not possible to simply glue the sensors to the bottom of the feet. Hence, a holder for the sensor is needed.

Because of the two requirements mentioned above it was decided that the sensor holder would consist of two parts, one cylinder used to push the sensor when the ground is being touched and one housing part that holds the cylinder in place. A simple design of the two parts was constructed in the CAD program Creo, see Figure 3.20, and sent to a 3D printer. The final result can be viewed in Figure 3.21.

When everything was in its place some simple tests were made to see if the holder worked satisfyingly.

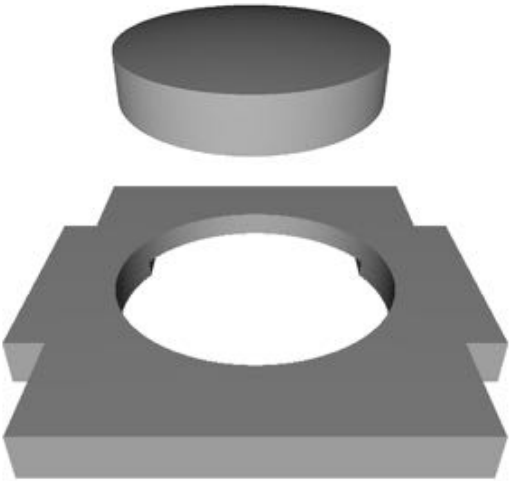


Figure 3.20 Sensor holder CAD design

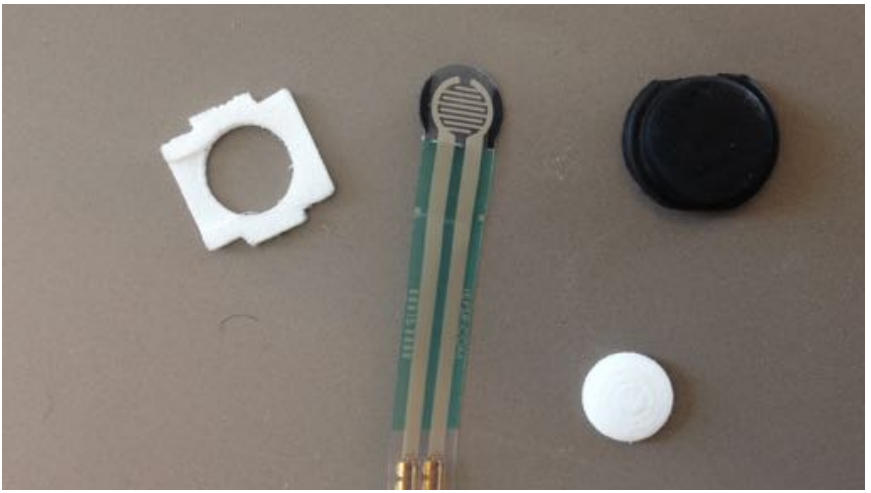


Figure 3.21 The 3D printed force sensor holder (white), the sensor and a pad to increase the friction (black).

3.16 Electrical Assembly

The electrical assembly of the hexapod is very much like the one in the previous master thesis from Dan Thilderkvist and Sebastian Svensson [Thilderkvist and Svensson, 2015] except that the camera and the force sensors have been added. The new circuit systems can be viewed in Figures 3.22 and 3.23 where the first one shows the power connections and the other figure shows the communication.

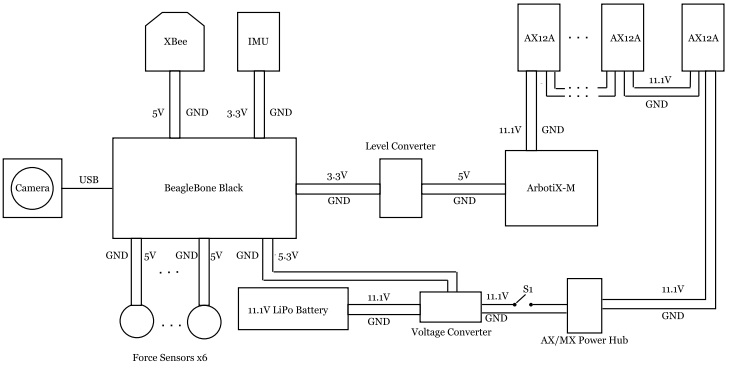


Figure 3.22 An overview of how the power is connected in the system.

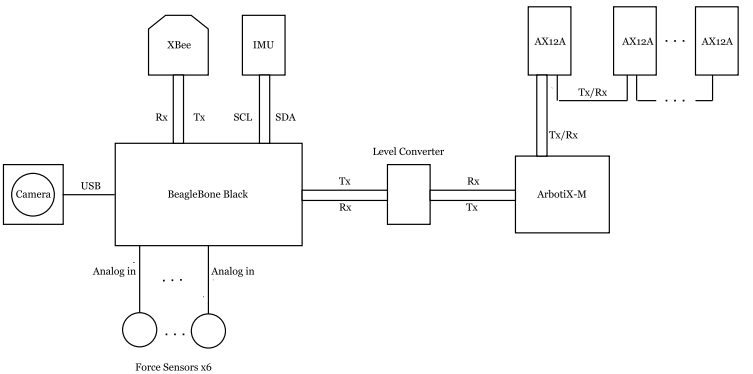


Figure 3.23 An overview of the communication circuit. Each wire is represented by a line.

The electrical assembly of the force sensors was constructed for all of the six sensors to make the connections easier. Figure 3.24 represents the model being used to construct the circuit board. The specific numbers of the pins on the BeagleBone are also printed on the model in order to make it clear for the user how to connect the cables. The 12 connections in the middle are used for the sensors and the six connections to the right represents the pins on the BeagleBone.

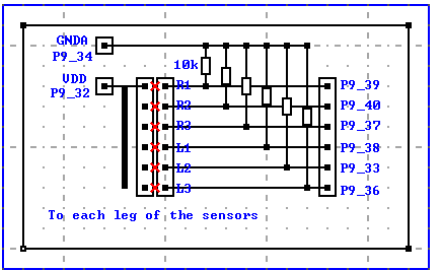


Figure 3.24 Force sensors circuit

4

Results

In this chapter all the tests that have been made are described. This includes tests for:

- RANSAC and RANSOP
- Mapping
- Complete planning test
- Force Sensors

For RANSAC and RANSOP tests were made both for the performance of the algorithms and comparisons of execution time. The tests for the Mapping were made in V-REP where the objective was to map an environment consisting of a couple of obstacle.

The next tests were three trials to test the planning, mapping and path-following. Three different scenarios were tested, one without any obstacle, one with a static obstacle and one test with a dynamic obstacle.

The last test performed was to see how well the force sensors and their implementation worked. That included mounting of the sensors as well as the programming of the leg movements.

Before the different tests are described more thoroughly a quick review of the hexapod hardware changes are presented.

In Figure 4.1 the current hexapod can be seen. What is new since the beginning of this project is mainly the implementation of the camera and the force sensors. Some changes regarding the mounting of different parts have also been made.

Figure 4.1 Hardware changes of the hexapod.

4.1 RANSAC and RANSOP

Testing

There are no guarantees or proofs that RANSOP will yield the correct results. However after running numerous tests it can be seen as working well. The real depth camera did not see the floor but an evaluation of the algorithm was still deemed relevant. Mainly since the algorithm might be used with other mapping sensors or in other robot applications with a better view of the floor in the future. For evaluation purposes point clouds from the V-REP simulation was used instead of real camera data. The depth camera simulation in V-REP does not have the same real world limitations as the real camera and therefore perfect point clouds were acquired. The resolution was 240x240 but decimated by a factor of 5 to 48x48 in order to speed up calculations. This decimation was also done on the real robot. Three different test cases were set up in V-REP on which both RANSOP and RANSAC were run. The algorithms were both run 100 times for each test case to yield a sufficient amount of test data. The execution time was measured for each run and each run was evaluated as being successful or not in order to determine the success rate. The tuned

parameters which are presented in Table 4.1 and Table 4.2 were kept through the whole testing process.

Parameters	Values	Description
nbrSampleSets	45	Number of random sets of points to try
nbrPoints	3	Number of coordinate points in each sample set
nbrValidationPoints	50	Number of points to use for evaluation of the sample sets

Table 4.1 RANSOP parameters

Parameters	Values	Description
nPoints	40	smallest number of points required
nIterations	10	number of iterations
threshold	0.05 meters	threshold used to id a point that fits well
nNearby	20	number of nearby points required

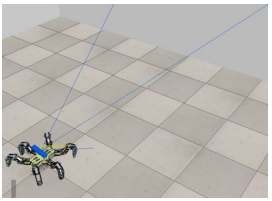
Table 4.2 RANSAC parameters

Algorithm Performance

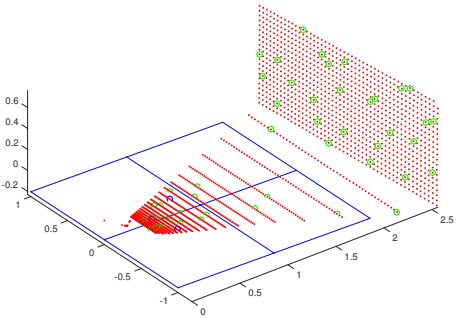
In the RANSOP figures the green circles encircle the randomly sampled validation points. The blue circles encircle the points chosen to estimate the model and the blue plane is the estimated plane. In the RANSAC figures all points encircled by a black circle are part of the set of inliers. Since RANSAC is run twice and the black circles are kept for both runs they could belong to either run.

Test 1 Test 1 was constructed as a simple environment only containing a wall and the floor as can be seen in Figure 4.7(a). During test 1 RANSAC had a success-rate of 100 % while RANSOP also had a success-rate of 100 %. In order to show that RANSOP is less robust than RANSAC a decrease in nbrSampleSets to 30 was tried. This decreased the success-rate for a test of 100 runs to 98 % for RANSOP.

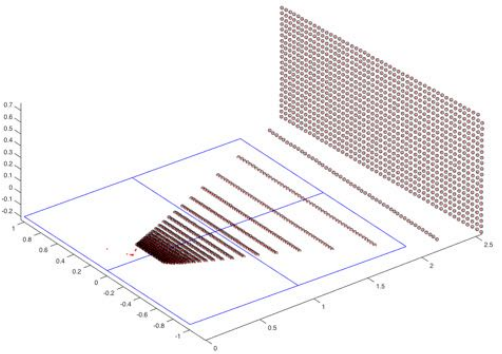
An increase in the number of sample sets will increase the probability of success, but there will always be a possibility of getting unwanted results with RANSOP.



(a) V-REP environment



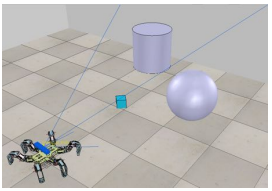
(b) Random Sample Optimization



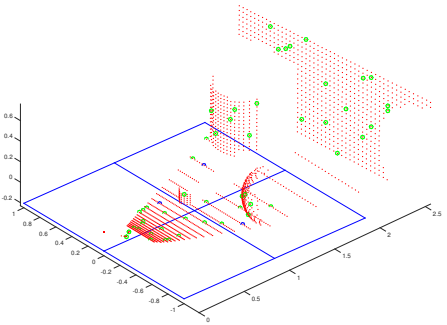
(c) Random Sample Consensus

Figure 4.2 Test 1

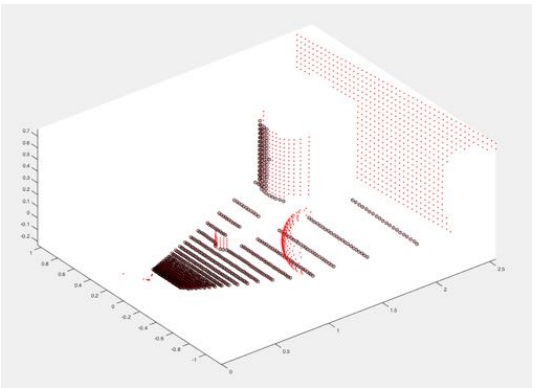
Test 2 Test 2 was constructed with a couple of objects obscuring the view but still not a complex scene with a fair amount of floor left visible. It can be seen in Figure 4.9(a). Of the 100 runs, RANSOP succeeded 96 times while the failed estimations were still really close to the real floor. RANSAC succeeded all runs.



(a) V-REP environment



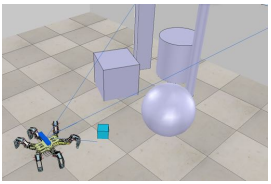
(b) Random Sample Optimization



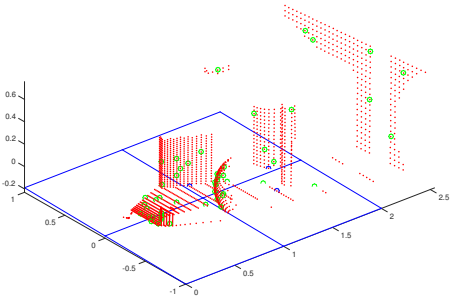
(c) Random Sample Consensus

Figure 4.3 Test 2

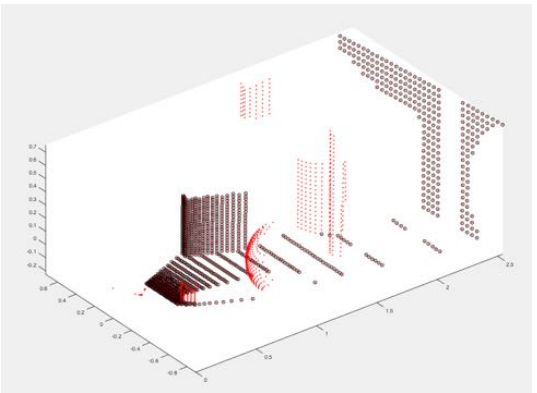
Test 3 In Test 3 even less floor was visible for the camera, with an even more complex environment and obstacles close to the robot. The environment can be seen in Figure 4.10(a). One result of RANSOP is presented in Figure 4.10(b). The RANSOP algorithm performed substantially worse in this test as compared to the previous ones, with a success-rate of only 72 %. This rate is obviously not acceptable so an attempt to increase number of sample sets to 1000 was tried. The success-rate was then increased to 99 %. RANSAC also struggled when only two planes were to be found since it often found two other planes than the floor plane. This was corrected by increasing the number of planes to four.



(a) V-REP environment



(b) Random Sample Optimization



(c) Random Sample Consensus

Figure 4.4 Test 3

The efficiency and robustness depends on which environment it is to be used in. But the conclusion is that in order for it to work in all environments the parameters used during test 3 has to be the ones used in production code.

Execution Time

The execution times were not measured on the target hardware and can therefore not be used as a measurement of the real-time possibilities on the robot, but more as a comparison between RANSAC and RANSOP. The algorithms were run 100 times on each of the test cases and the execution times were measured and are presented in Figure 4.5.

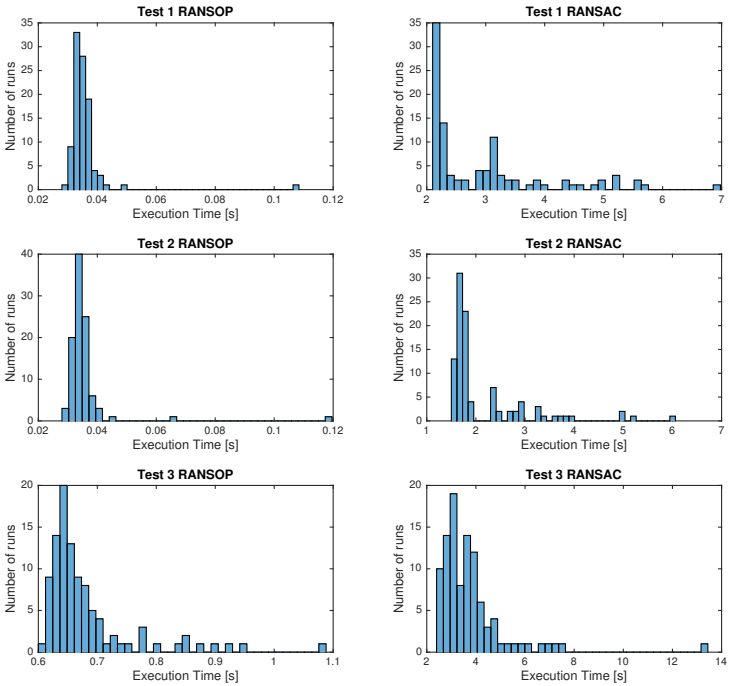
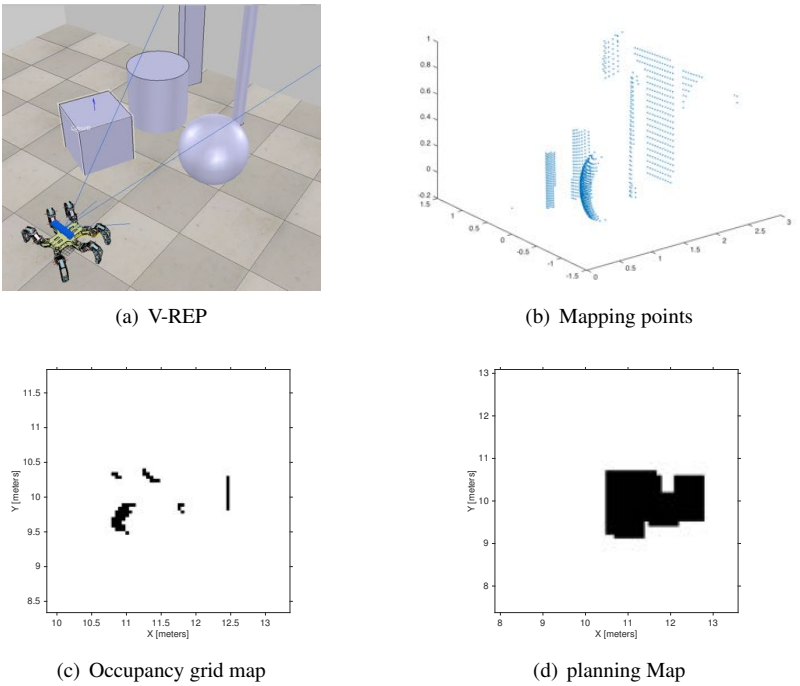


Figure 4.5 Execution times for RANSAC and RANSOP

4.2 Mapping

A complete example of mapping with point data from V-REP is presented in Figure 4.6. The scene was set up to contain a couple of obstacles. In (a) the V-REP visualization can be seen, (b) presents the point cloud acquired by transforming the virtual camera depth map to coordinates in space. The high-resolution occupancy grid map is presented in (c) and the planning map with lower resolution and margins in (d).



(a) V-REP (b) Mapping points

(c) Occupancy grid map (d) planning Map

Figure 4.6 Mapping Test

4.3 Complete Planning Tests

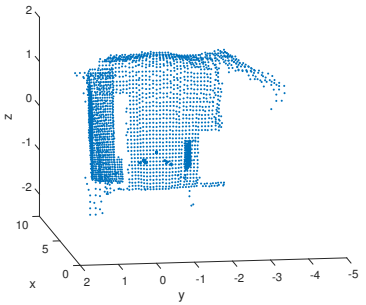
In order to test the complete hexapod three test cases were run. A first simple test was made with no obstacles blocking the path between start and goal. During the second test a static obstacle was placed between the hexapod and the goal. Finally, the third test was performed within a dynamic environment to force a complete re-planning of the path. This was done by starting with no obstacles and then adding an obstacle in front of the hexapod while it was walking.

Test 1, Static Environment With No Obstacles

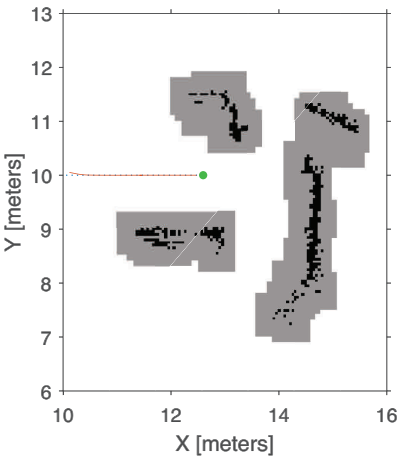
In Figure 4.7 the results for test 1 are presented. A depth map is presented in (a) where the room can be seen. Darker areas in the depth map are closer to the viewer while lighter areas are more far away. The depth map was captured right at the start of the test and then converted into points in (b). The final map is shown in (c), where the black areas are obstacles and the gray areas are planning obstacles with safety margin. The path planned is seen as the tiny blue dots, while the actual path walked is the red line. In (d) the hexapod can be seen in the real world with the green dot

being the goal. Figure 4.8 displays the path more precisely with the green dot being the goal.

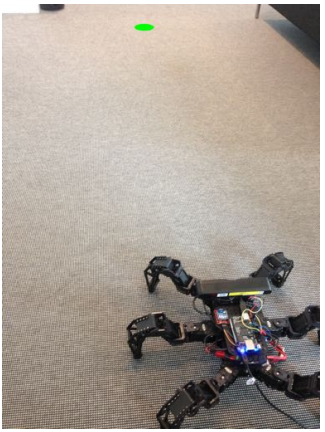
(a) Depth Map



(b) Mapping points



(c) Occupancy grid map and corresponding planning map



(d) Real world

Figure 4.7 Test 1

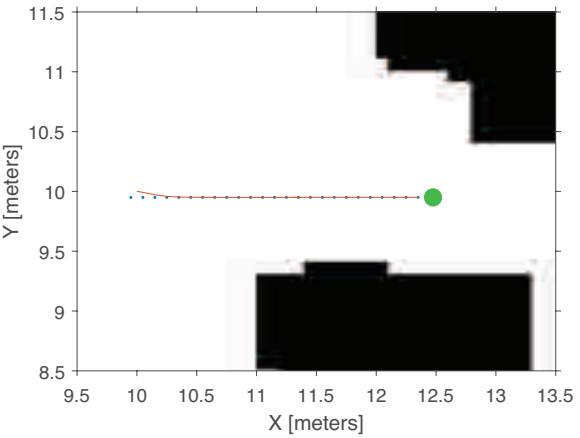
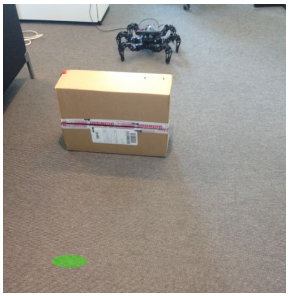


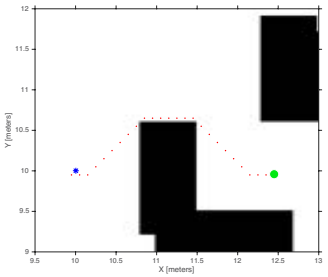
Figure 4.8 Test 1

Test 2, Static Environment With Obstacles

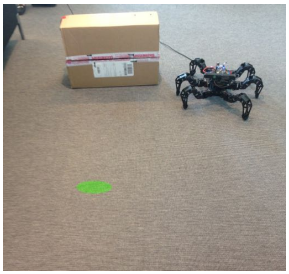
A box was placed in-between the goal and start position of the robot. The environment was kept static and the robot walked the planned path perfectly. In Figure 4.9 three sampled times of the planning are presented. The left column of images display real world images of the hexapod at time t. The right column contains the map at time t, the red dots are the planned path and the blue dots display the path traversed by the robot. The blue star denotes the hexapod's current position and the green circle is the goal.



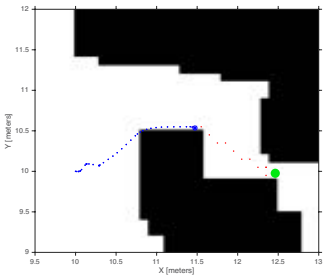
(a) Real world, time = 0 s



(b) Map and Plan, time = 0 s

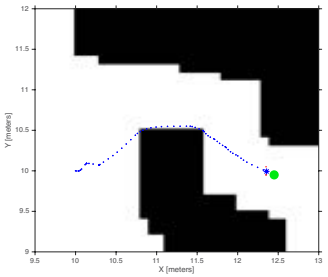


(c) Real world, time = 20 s



(d) Map and Plan, t = 20 s

(e) Real world, time = 35 s

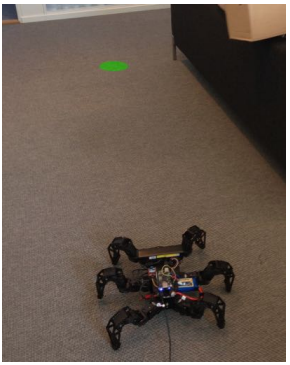


(f) Map and Plan, t = 35 s

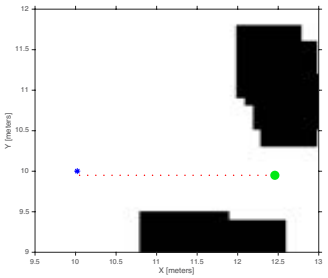
Figure 4.9 Test 2

Test 3, Dynamic Environment

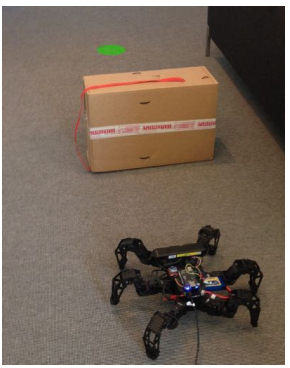
The start and goal positions are the same as in previous tests. However, an obstacle is added after five seconds during this test. Figure 4.10 shows three snap shots of the run.



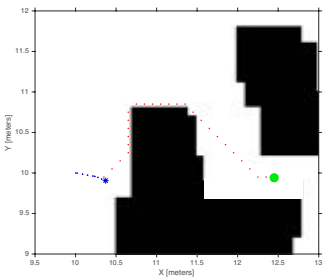
(a) Real world, time = 0 s



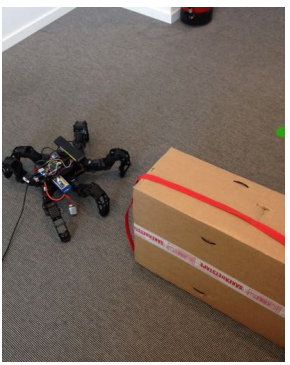
(b) Map and Plan, time = 0 s



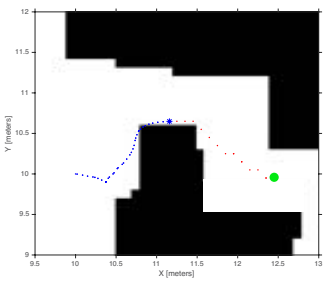
(c) Real world, time = 5 s



(d) Map and Plan, t = 5 s



(e) Real world, time = 20 s



(f) Map and Plan, t = 20 s

Figure 4.10 Test 3

4.4 Force Sensors

To test the force sensors mode 3 was run on different terrains. The first test was to see if the leg stopped earlier if the sensor got into contact with something above ground, for example, a stone. This test was successful and the leg stopped at the correct position. The next test done was to see if the leg could keep going below ground level and then stop if the sensor indicated ground about one cm below the ground level. Also this test was successful.

Finally, a test was made where the sensor never indicated ground. To do this test the robot was placed on a table and was allowed to walk straight forward. When one of the front leg got to the edge it responded exactly as wanted and hence, the hexapod did not fall of the table.

The only time the tests were not successful was when the leg slipped in such a way that the sensor never indicated ground touch.

At the moment the force sensors are only integrated on the two front legs, mainly because other things were prioritized before integrating them on every leg. This means that the tests made above only works when the robot is moving towards the edge with its front.

5

Discussion and Conclusions

5.1 Discussion

Simulink and Model-Based Design with Verification in V-REP

Simulink has some advantages and disadvantages when it comes to building a mapping and planning system. It has been convenient to generate code to the BeagleBone and small changes in the Simulink code can make for large differences in the generated code which make for a fast testing and iteration process. However, when systems with large matrices or data structures have to be built it is better to implement them in C/C++ or Matlab code and run it in the Simulink model.

SimMechanics

There were a couple issues with modeling friction in SimMechanics. The contact friction package in SimMechanics only supported friction between pre-set shapes, such as between a sphere and a plane. The robot is to interact with all sorts of shapes in the terrain. One solution might have been to approximate different shapes with large amounts of spheres or planes. This would mean that there had to be an explicit friction model between all of these shapes and the feet. This is a cumbersome solution when it comes to creating the SimMechanics model, which would need friction blocks between all planes and the spheres on the feet. Another possible solution would be to change the position and orientation of the friction model plane depending on the topography of the terrain. So the same object would be used for all contacts between feet and ground, but this object would move to the position of a foot when it was to be in contact with the ground. A similar solution was proposed by Burkus and Odry in a paper where they suggest modeling the ground using polygons generated by Matlab functions [Burkus and Odry, 2014].

Mapping Sensors

A Time-of-flight camera was used on the robot, but many different mapping sensors were discussed during the thesis. One option was to use ultrasonic sensors which utilize sound to find obstacles. These can only see distances along one axis and were discarded immediately. The reason being that a more high resolution map was preferred in order to perform robust planning. Other options were depth cameras, also called time-of-flight cameras and LIDAR (Light Detection And Ranging). LIDARs are commonly used in world mapping applications for autonomous vehicles. They work by essentially shooting a laser pulse and measuring the reflection time. By sweeping the environment a full map can be built point by point. Time-of-flight cameras work in a similar fashion but in comparison the whole 3D mapping is done with one pulse of infrared light, which covers the whole area of interest. The reflected light is then captured by a photo-sensor like on a regular camera [Varghese and Boone, 2015]. Yet another option is to use a stereo camera, which essentially is two cameras with a known distance between them. Stereo cameras find matching points in the two images captured and measure the distance between them. This is fairly heavy computationally and accuracy is a problem. Therefore, a time-of-flight camera is preferred over a stereo camera [Li, 2014]. A full scanning LIDAR would have been preferred over a time-of-flight camera since there are versions which can map in 3D 360 $^{\circ}$ around the robot. LIDARS with full mapping capabilities are however expensive while there exist a fair amount of good consumer grade time-of-flight cameras. The time-of-flight cameras are widely used and lots of open source code for implementation and use can be found. These were the two reasons a time-of-flight camera was chosen over a LIDAR for mapping purposes.

Depth Camera

The depth camera is a convenient tool for mapping the environment during the right circumstances. Numerous limitations have however been discovered during the course of this thesis. These limitations are mostly intrinsic based on physical fundamentals of electromagnetic waves. The camera works by sending a pulse of light in the infrared spectrum which is reflected by surfaces in the environment. The reflected light beams are measured by a photo sensor similar to what is used in a regular digital camera. The time between when the light pulse is sent and when the different pixels sense the reflected light is measured and the distance is calculated. If a pixel in the sensor does not acquire enough infrared light, then there will be no measurement in that pixel. Therefore there has to be a beam reflected back to the camera from the surface in order for a depth value to be registered. If the surface is not right in front of the camera there has to be a certain roughness in order for light beams to also be spread back to the camera sensor. This is not a problem for most surfaces but an example is glass or other reflective surfaces. The general rule goes that the shinier and more reflective an object is, the worse the depth camera will handle it. Another factor which was discovered to play a role in the ability of

the depth camera was the angle between the camera and surface. For the hexapod the outgoing beams were at a small incoming angle to the floor. The result was that a majority of the beams were reflected away from the floor at the same, or similar out-angle. But not enough light was reflected back to the camera and therefore the floor would not be included in the point cloud. This has not been an issue when the floor is equally reflective so no points are added to the point-cloud. But when the floor reflects a small amount of the light this might lead to inconsistent behavior in the algorithms. Another flaw is the effect of surrounding light. Indoors this is not a problem, but outdoors there are large amounts of ambient infrared light. This disturbs the camera to the extent that it does not even work outdoors.

RANSAC and RANSOP

RANSOP was never used in the final version of the robot for two reasons. The first reason being that the data collected by the depth camera had a hard time registering the floor. For most types of floor surfaces there were no readings since the floor reflected the infrared light away from the camera. The second reason being that even though execution time is lower than original RANSAC it would add some time for calculations and still not add much extra functionality. RANSOP will be less robust for environments where only a small part of the number of points are points in the plane which is to be found. In this case, the robot is assumed to walk in an environment with a flat floor and a fairly open space. Based on these assumptions a substantial part of the points should be points from the floor. A difficulty which might skew the results of the tests are the many parameters of both RANSOP and RANSAC. The tuning of these might affect the outcome both when it comes to execution time and success-rate. The tuning was done by trial and error until both performance and robustness seemed reasonable.

Force Sensors

In Chapter 3 two alternatives to the force sensing resistors were mentioned, one was to place small buttons under the feet and the other to measure the internal forces of the servos. The reason why the resistors were finally chosen was because they are easy to use, precise and gives more information about the pressure than just if it has contact or not. The buttons would only give true or false depending on ground touch or not and even if this information is enough in this master thesis it could be a nice feature in the future to now how much pressure there is on the feet, for example to be able to balance the body. Another disadvantage of the push button is that the force must be applied perpendicular to the button. If the foot slides a little the button might not be triggered. This can also be a problem with the force sensing resistors, since they work better if the force is perpendicularly to the sensor but it has a better chance of catching forces coming from the side. The internal measurements of the forces could work, but would require a lot of testing and calibrations before they would give a satisfying result and even then it was not certain that the measurements

would be accurate enough. Since the force sensing resistors worked so nicely and were easy to use it was decided to use them. In the Result section it can be seen that they are precise and work as expected.

The difficult part about the force sensing resistors was to mount the them in a proper way. In the end a solution was found and with help from a 3D printer a prototype could be build that worked as required. A 3D printed prototype is cheap and very easy to construct but there may be some flaws with 3D printing a product. The different materials that can be chosen are limited and the prototypes are often not as strong as traditionally manufactured parts. The dimension errors in the prototype can also cause problems, for example, some of the force sensor holders needed to be filed down in order to fit the hexapod [*Disadvantages of 3D Printers*].

Localization

In order to localize the position of the hexapod in the environment only the internal positioning was used due to a lack of time. This solution is useful in predictable environments where the feet are not likely to slip. Another issue with this method is that all small uncertainties will incrementally add up and produce a drift in the localization. However, this method would prove to work well for smaller distances. Since the map is updated continuously there would only be problems if some area of the map is obscured for a longer period of time. More measurement sources were however available and could have been used when estimating the position. The IMU provided measurements from the accelerometer which could be integrated twice to get changes in position. The IMU does produce a large drift, which could have been solved by fitting the camera information to the previous map information to get the global position. These extra sources of information could have been joined in a Kalman filter for position estimation, or a method called Simultaneous localization and mapping could have been used [Huang et al., 2005].

Alternative Mapping Methods

Below a number of alternative mapping methods are discussed. They were not used mainly as a result of lacking computational power and time constraints when writing the thesis.

Primitive Shape Estimation

One way of reducing the amount of data is by estimating primitive shapes from the point cloud. The parameters of the primitive shapes are kept as a representation of the environment. When a new point cloud is captured, these points are compared against the primitive shapes to find which are inliers of the shapes and which are outliers which might be part of new shapes. The algorithm which in many cases is the base for these methods is RANSAC [Fischler and Bolles, 1981]. One way of using object recognition with shape matching [Somani et al., 2013] is to fit a number of primitive shapes to the point cloud. Surfel estimation [Henry et al., 2014] is

another method where surfaces are approximated to the point cloud which reduces the amount of data to save. Methods within these areas of shape matching and estimation are, however, computationally heavy and not suitable for real-time operation on an embedded processor.

3D Voxel Discretization

Another method for mapping a 3D environment is by discretizing the whole space. This essentially fills the space with boxes, also called voxels. A voxel is filled if a point from the point cloud exists in the box. This way memory usage is known but more memory is used than actually necessary. Memory usage can be decreased by representing all known voxels, both occupied and free, in a tree structure [Maier et al., 2012]. The boxes can also be associated with occupancy probabilities for dynamic environments. Another advantage is that unobserved boxes are implicitly modeled since they are not included in the box-tree. An illustration of the box method is presented in Figure 5.1.

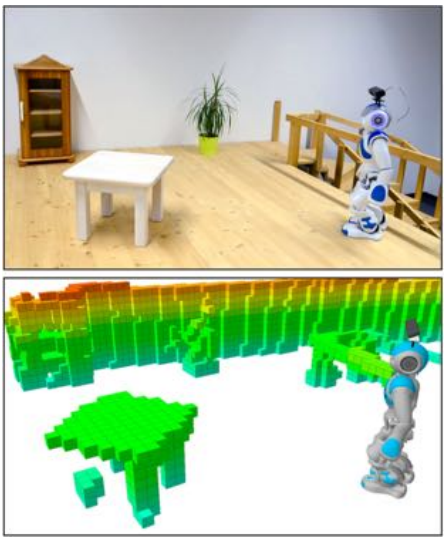


Figure 5.1 Voxel discretization, [Maier et al., 2012]

Mapping the complete 3D environment would have been preferred from an information point of view. This would also have allowed for planning in multi-level environments and terrain traversal. However, the restrictions placed by the embedded processor once again forced other approaches to be considered.

Discretization Issues

The world is discretized in 5x5 cm cells for the mapping which is then transformed to a grid of 10x10 cm cells for planning. The computations for planning would have been too cumbersome had the grid resolution been finer. The coarse planning grid does however yield problems for the margins required around the obstacles in order to assure collision free paths. An illustration of this can be seen in Figure 3.9 in the Method. It is clear that the margins around the robot are larger than really necessary. But if margins on either of the sides were removed a full 10 cm of margin would disappear. The modeled hexapod is placed within a certain cell of the map as long as it is positioned within it, even if it is positioned at the edge of the cell. The result of decreasing the margins and a case where the hexapod is positioned at the edge of a cell, would yield a collision. These large margins result in the hexapod not being able to traverse narrow passages even when there is enough space. Possible solutions for this problem could be to replace the processor with a more powerful one and increase the resolution of the planning map. A probably better solution would be to use another method of building the road map, such as Probabilistic Road Map or Rapidly Exploring Random Tree. This would allow for the fine resolution map to be used in planning enabling a more precise use of margins.

Comparison of Planning methods

A couple of alternative planning methods were explained in the theory section. These methods are discussed and compared below.

Potential Field Methods Potential field methods are based on the simple idea of having repulsive and attractive forces act on the robot. The resultant force vector decides in which direction the robot moves. There are some inherent difficulties in dealing with potential field methods. One problem is that the robot might end up in local minima for certain obstacle configurations. This also apply for narrow passages. Another common problem while planning close to multiple obstacles is an oscillating behavior in the path. This is due to the force of one obstacle sending the path closer to another obstacle which creates oscillating overcompensation [Koren and Borenstein, 1991]. An example of a narrow corridor instability is shown in Figure 5.2. There are methods for overcoming these problems but the difficulties still deterred further investigations of potential field methods.

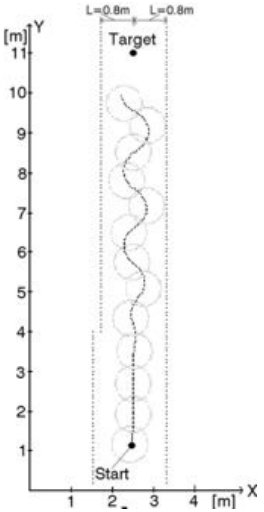


Figure 5.2 Example of unstable potential field path, [Koren and Borenstein, 1991]

Visibility Graphs and Voronoi Diagrams These are two potential methods for building a road map to perform search in. They will yield opposite types of road maps. Visibility graphs creates a graph which rounds corners as closely as possible while Voronoi diagrams keeps the largest clearance possible. Neither case is optimal from a planning point of view where margins in an uncertain world is important but also keeping the path as short as possible. They are not appropriate for changes in the environment since the road map will have to be reconnected when obstacle positions change. For these reasons visibility graphs and Voronoi diagrams were discarded as methods for road mapping.

Probibilistic Road Map and Rapidly Exploring Random Trees These are two methods where sampling is used to build the road map. They are both efficient methods for planning in narrow spaces and with a high dimensional configuration space. In this case, however, with only three degrees of freedom in the c-space it did not seem to provide any additional advantages over simpler methods.

Comparison of Search Methods

Different search methods are discussed below. In order for search to be performed the position of the robot has to be translated into a node in the road map graph. This is easily done by checking which cell the hexapod is positioned within the grid and set this cell as the start node.

Depth First Search and Breadth First Search These are solid methods for finding a good path. Depth-first search will always find the optimal path when it comes

to number of steps in the graph. However, none of these methods utilize the information which is present about the world, such as the position of the graph nodes and position of the goal.

Uniform Cost Search The problem of not utilizing the position of the graph nodes is solved in uniform cost search. Here each node is associated with the corresponding cell and the edges are labeled with their actual length. This edge value is used in uniform cost search in order to decide which node to expand. Had the node connections only been vertical and horizontal, this method would have yielded the same results as breadth first search, the reason being that the number of steps would have been proportional to the actual path length. This is however not the reason when the diagonal edges are included since their cost is different from the horizontal and vertical costs. Uniform cost search does however not utilize the information about where the goal is and therefore UCS was also discarded.

A* Search In order to use information about where the goal is located A* search was considered. A* adds an estimate of how far away the goal is to the decision of which node to expand. This makes the search biased towards the goal, which decreases the number of nodes expanded and thereby the search time.

D* Search While A* is good at efficiently finding a path in a static environment, the whole path still has to be recalculated when something in the environment changes. Dynamic A*, D*, is the solution to these problems. D* only recalculates local areas which have been changed by moving obstacles. This makes it more efficient for real-time operation in dynamic environments.

Planning

Using D* as a search method has been working really well and it is truly a great search method for dynamic and fast search. However the implementation used came with a few draw-backs. It is limited by the assembly of the road map. The road map only allows traversal between cells horizontally, vertically or diagonally. For most cases the optimal path will be at another angle. A possible solution could be to use another method for assembling the road map such as a PRM. Since it samples the space in order to find possible positions for nodes it will be free of the grid structure. There also exist a myriad of alternative search methods closely related to D* Lite. Theta* [Daniel et al., 2010], also called "Any Angle Theta" is a method worth further investigation. It is based on A* and employs a line-of-sight algorithm in order to cut the path as short as possible which also allows paths at any angle.

Path-Following

The simple path planning method used is working very well for static paths, which are followed smoothly without cutting corners too narrowly. However, in some cases when the path is re-planned the current way-point is shifted quickly from one direction to a completely different direction. The rotation control signal therefore might

go from the maximum positive value to the maximum negative value in one sample. This might result in jerky movements with the risk of leg collision. It can be somewhat solved by lowering the allowed maximum rotational signal. In order to really solve the problem a low-pass filter might be used to filter out high frequencies and thereby blocking the quick changes.

Cornering Issues

One issue found during testing was the issue of walking around obstacles where only the side of the obstacle facing the hexapod has been seen. When the hexapod rounds the corner it has never added the new side of the obstacle to the map. Since the TOF-camera has a blind distance the first 0.5 m it will never add and see the other side of the obstacle which increases the risk of collision, see Figure 5.3 for an example. Here can be seen how the green part of the obstacle is seen with the camera, but the red parts are never added to the map. Since the hexapod wants to reach the goal right behind the obstacle the path is planned cutting the corner since that part of the obstacle never is added to the map.

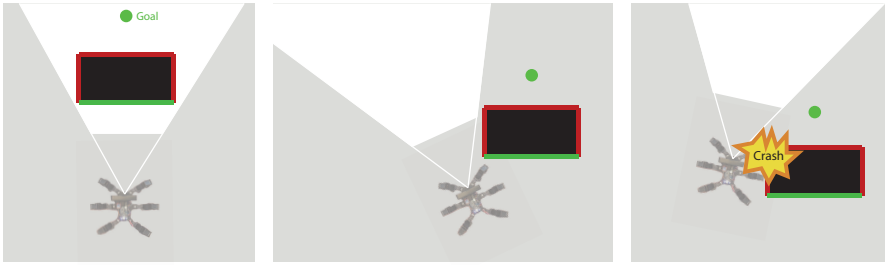


Figure 5.3 Illustration of a failed corner rounding

One solution to this problem might be adding more hardware to the hexapod. Other sensors could pick up the slack within the dead zone of the camera. A couple of ultrasound sensors could have been added just in order to avoid collision with close obstacles, especially in a dynamic world.

Another possible solution could be to use a decision-making model for how the hexapod is supposed to traverse the path during different circumstances. It could then rotate and walk sideways around corners with a margin in order to map the other side of obstacles before trying to traverse those areas.

5.2 Conclusions

Overall this project has been successful, despite a lack of computational power and some time consuming implementations, the hexapod became autonomous. V-REP

has been a great way of testing algorithms before putting them on the hardware. It allowed for a much faster iteration testing process than testing directly on the hexapod would have been. Being able to plot all signals in real-time and having control of the environment also helped in the testing. Uncertainties in the real world have been a large issue during the thesis. Even though the planning and mapping algorithms worked fine after testing in V-REP there were almost always issues when moving the testing onto the real hardware in the real world. With more testing and tweaking most of the algorithms ended up working fine.

The TOF camera has many positive aspects, despite issues in locating the floor and not being able to use it outdoors. It is a convenient and cheap solution for mapping a complete 3D environment. Capturing the depth map was not at all computationally heavy even though processing the point cloud was. Mapping the environment in an accurate dynamic way has been a challenge. The errors in localization of the position and rotation did not effect the map during the short tests which were run. During longer tests it might have been problematic.

The planning part using D* Lite was tricky to implement using S-functions. However, it did work efficiently and with great results when it was implemented.

Finally, the terrain handling mode works really well. The force sensing resistors used are sensitive, small and fairly cheap. The mounting generated some problems but in the end a solution was found that works satisfyingly. Since a 3D printer could be used the parts needed to mount the sensors were easy to construct.

The hexapod ended up being able to plan a path, avoid obstacles and handle simple terrain. The potential improvements in performance, number of features and robustness in planning are immense.

5.3 Future Improvements

Many of the future improvements have been discussed as possibilities through-out the thesis. One of the largest obstacles in development has been the computational power of the BeagleBone Black. It has a good processor which is sufficient for generating trajectories, performing inverse kinematics and some mapping and planning. However for some of the methods, especially in a 3D environment, computations tend to become cumbersome.

Full 3D Planning

In order to make the hexapod able to traverse terrain it would probably be necessary to perform planning in a fully three dimensional model. Mapping a 3D environment was briefly mentioned earlier in the thesis, but was then discarded since it was too computationally heavy.

Mapping dynamic environments with probabilistic methods

The mapping only contains two possible states, either a cell is occupied or it is free. A more advanced system could be built where cells could be free, occupied or uncertain. The current version regards cells which have not been seen as free cells. Even though it actually is uncertain whether they are free or not. This information could be used to make more robust maps and planning. One example of how to use the extra information is presented in [Hähnel et al., 2013].

Decision-Making

The hexapod could be developed with a specific goal in mind. For example to map the environment or to search for a specific object. This would require some form of decision-theoretic model in order for the system to itself make decisions regarding where to walk.

Simultaneous Localization and Mapping

A better system for mapping and localizing the hexapod in the world would be necessary if the goal is to create accurate maps.

Force Sensors

In the future, one goal could be to apply the sensors to every foot and integrate them with the balancing mode and thereby get the hexapod to re-balance as the legs are in different positions.

Bibliography

ASUS. *Xtion pro live*. http://www.asus.com/se/3D-Sensor/Xtion_PRO_LIVE/. Accessed: 2016-05-09.

BeagleBoard, F. *Beaglebone black*. https://beagleboard.org/black. Accessed: 2016-06-03.

Burkus, E. and P. Odry (2014). *Implementation of 3D Ground Contact Model for Uneven Ground Using SimMechanics*. Tech. rep. 10.1109/SISY.2014.6923572. Subotica Tech, Subotica, Serbia.

Cornell, S. *Disadvantages of 3d printers*. http://yourbusiness.azcentral.com/disadvantages-3d-printers-2212.html. Accessed: 2016-06-07.

Daniel, K., A. Nash, S. Koening, and A. Felner (2010). *Theta*: Any-Angle Path Planning on Grids*. Tech. rep. Journal Of Artificial Intelligence Research, Volume 39, pages 533-579, 2010, DOI: 10.1613/jair.2994. Computer Science Department University of Southern California, Los Angeles,USA. URL: http://www6.in.tum.de/Main/Publications/Somani2014a.pdf.

Eriksson, A. and M. Malmros (2016). *Developer Manual, Artificial Intelligence and Terrain Handling of a Hexapod Robot*. Users guide. Department of Automatic Control, Lund University, Lund, Sweden.

Example of RANSAC. https://en.wikipedia.org/wiki/RANSAC. Figure reference, Accessed: 2016-06-09.

Fischler, M. A. and R. C. Bolles (1981). *Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography*. Tech. rep. ACM New York, NY, USA, Vol. 24, Issue 6, pp. 381-395, DOI: 10.1145/358669.358692. Columbia University.

Gustafzelius, S. (2015). *Dynamic path planning of initially unknown environments using an RGB-D camera*. Master's Thesis TFRT--5980--SE. Department of Automatic Control, Lund University, Lund, Sweden.

Henry, P., M. Krainin, E. Herbst, X. Ren, and D. Fox (2014). *RGB-D Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments*. Research paper. Springer-Verlag GmbH Berlin Heidelberg, Vol. 79, DOI 10.1007/978-3-642-28572-1_33, Print ISBN: 978-3-642-28571-4, Online ISBN: 978-3-642-28572-1, Series ISSN: 1610-7438.

Huang, G., A. Rad, and Y. Wong (2005). *Online SLAM in Dynamic Environments*. Tech. rep. ICAR '05. Proceedings., 12th International Conference on Advanced Robotics, DOI: 10.1109/ICAR.2005.1507422, Print ISBN: 0-7803-9178-0. Department of Electrical Engineering, The Hong Kong Polytechnic University, Hong Kong.

Hähnel, D., R. Triebel, W. Burgard, and S. Thrun (2013). *Map Building with Mobile Robots in Dynamic Environments*. Research Paper. Kluwer Academic Publishers, Vol. 19, Issue 1, DOI: 10.1007/s10514-005-0606-4, Print ISSN: 0929-5593, Online ISSN: 1573-7527. University of Freiburg, Department of Computer Science, Freiburg, Germany and Carnegie Mellon University, School of Computer Science, PA, USA.

Interlink electronics. *FSR Force Sensing Resistor Integration Guide and Evaluation Parts Catalog*. Users guide. Camarillo, CA, USA.

Koenig, S. and M. Likhachev (2002). *D* Lite*. Paper. American Association for Artificial Intelligence. Georgia Institute of Technology, Atlanta, USA, Carnegie Mellon University, Pittsburgh, USA.

Koren, Y and J Borenstein (1991). *Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation*. Paper. Proceedings of the IEEE Conference on Robotics and Automation, Sacramento, California, April 7-12, 1991, pp. 1398-1404. The University of Michigan.

Lavalle, S. M. (2006). *Planning Algorithms*. `http://planning.cs.uiuc.edu`. Cambridge University Press, Camebridge, UK.

Leonard, S. *Potential field methods*. `http://www.cs.jhu.edu/~sleonard/week03.pdf`. Accessed: 2016-03-17.

Li, L. (2014). *Time-of-Flight Camera – An Introduction*. Company Tech Report. Literature number: SLOA190B, http://www.ti.com/lit/wp/sloa190b/sloa190b.pdf. Texas Instrument.

Maier, D., A. Hornung, and M. Bennewitz (2012). *Real-Time Navigation in 3D Environments Based on Depth Camera Data*. Research Paper. 2012 12th IEEE-RAS International Conference on Humanoid Robots, DOI: 10.1109/HUMANOIDS.2012.6651595, Print ISBN: 978-3-642-28571-4, ISSN: 2164-0572. University of Freiburg, Germany.

Mathworks. *Matlab*. `http://se.mathworks.com/products/matlab/`. Accessed: 2016-06-02.

Mathworks. *Simscape multibody contact forces library*. http://www.mathworks.com/matlabcentral/fileexchange/47417-simscape-multibody-contact-forces-library. Accessed: 2016-02-02.

Mathworks. *Simulink*. http://se.mathworks.com/products/simulink/. Accessed: 2016-06-02.

Mathworks. *Simulink Coder User's guide*. Users guide. URL: http://se.mathworks.com/help/pdf_doc/rtw/rtw_ug.pdf.

Mathworks. *Understanding c code generation*. http://se.mathworks.com/help/dsp/ug/understanding-code-generation.html. Accessed: 2016-05-31.

Microsoft. *Kinect for windows sensor components and specifications*. https://msdn.microsoft.com/en-us/library/jj131033.aspx. Accessed: 2016-05-09.

Neufeld, J. *D* lite c++ implementation*. https://github.com/ArekSredzki/dstar-lite. Accessed: 2016-03-11.

Picture of FSR. http://www.sedoniatech.co.nz/force-sensing-resistor-touch-sensor.htm. Figure reference, Accessed: 2016-05-25.

Robotics, C. *Coppeliarobotics.com*. http://www.coppeliarobotics.com. Accessed: 2016-02-02.

Robotics, C. *Designing dynamic simulations*. http://www.coppeliarobotics.com/helpFiles/en/designingDynamicSimulations.htm. Accessed: 2016-02-02.

Robotics, C. *Enabling the Remote API - client side*. http://www.coppeliarobotics.com/helpFiles/en/remoteApiClientSide.htm. Accessed: 2016-02-02.

Robotics, C. *Remote API modus operandi*. http://www.coppeliarobotics.com/helpFiles/en/remoteApiModusOperandi.htm. Accessed: 2016-02-02.

Russel, S. and P. Norvig (2010). *Artificial Intelligence, A Modern Approach*. http://aima.cs.berkeley.edu. Pearson Education, New Jersey, USA.

Somani, N., C. Cai, A. Perzylo, M. Rickert, and A. Knoll (2013). *Object Recognition using constraints from Primitive Shape Matching*. Paper. DOI: 10.1007/978-3-319-14249-4_75, Print ISBN: 978-3-319-14248-7, Online ISBN: 978-3-319-14249-4, ISSN: 0302-9743. Cyber-Physical Systems, fortiss - An-Institut der Technischen Universität München. Technische Universität München, Fakultät für Informatik.

Structure. *Openni 2 sdk binaries & docs*. http://structure.io/openni. Accessed: 2016-02-02.

Thilderkvist, D. and S. Svensson (2015). *Motion Control of Hexapod Robot Using Model-Based Design*. Master's Thesis TFRT--5971--SE. Department of Automatic Control, Lund University, Lund, Sweden.

Varghese, J. and R. Boone (2015). *Overview of Autonomous Vehicle Sensors and Systems*. Tech. rep. Proceedings of the 2015 International Conference on Operations Excellence and Service Engineering Orlando, Florida, USA, September 10-11, 2015, pp. 178-191. Department of Electrical Engineering, University of Michigan, United States.

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

Document name
MASTER´S THESIS

Date of issue
September 2016

Document Number
ISRN LUTFD2/TFRT--6010--SE

Author(s)
Markus Malmros
Amanda Eriksson

Supervisor
Simon Yngve, Combine
Anders Robertsson, Dept. of Automatic Control, Lund University, Sweden
Rolf Johansson, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle
Artificial Intelligence and Terrain Handling of a Hexapod Robot

Abstract

The focus of this master's thesis has been getting a six-legged robot to autonomously navigate around a room with obstacles. To make this possible a time-of-flight camera has been implemented in order to map the environment. Two force sensing resistors were also mounted on the front legs to sense if the legs are in contact with the ground. The information from the sensors together with a path planning algorithm and new leg trajectory calculations have resulted in new abilities for the robot.

During the project comparisons between different hardware, tools and algorithms have been made in order to choose the ones fitting the requirements best. Tests have also been made, both on the robot and in simulations to investigate how well the models work.

The results are path planning algorithms and a terrain handling feature that works very well in simulations and satisfyingly in the real world. One of the main bottlenecks during the project has been the lack of computational power of the hardware. With a stronger processor, the algorithms would work more efficiently and could be developed further to increase the precision of the environment map and the path planning algorithms.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-92

Recipient's notes

Security classification

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