Odometry based Map Building

Final Report for CS39440 Major Project

Author: Stefan Klaus (stk4@aber.ac.uk)
Supervisor: Dr. Myra Wilson (mxw@aber.ac.uk)

21st April 2012
Version: 1.0 (Draft)

This report was submitted as partial fulfilment of a BSc degree in AI & Robotics, (GH76)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, UK

Declaration of originality

In signing below, I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for plagiarism and other unfair practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the sections on unfair practice in the Students' Examinations Handbook and the relevant sections of the current Student Handbook of the Department of Computer Science.
- I understand and agree to abide by the University's regulations governing these issues.

Signature .....

Date .....

Consent to share this work

In signing below, I hereby agree to this dissertation being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Signature .....

Date .....

Acknowledgements

I'd like to thank my project supervisor, Myra Wilson, for her support and suggestions during this project.

Abstract

Include an abstract for your project. This should be no more than 300 words.

CONTENTS

LIST OF FIGURES

Listings

Chapter 1

Background & Objectives

This section should discuss your preparation for the project, including background reading, your analysis of the problem and the process or method you have followed to help structure your work. It is likely that you will reuse part of your outline project specification, but at this point in the project you should have more to talk about.

Note:

- All of the sections and text in this example are for illustration purposes. The main Chapters are a good starting point, but the content and actual sections that you include are likely to be different.
- Look at the document on the Structure of the Final Report for additional guidance.

1.0.1 SLAM - Simultaneous Localization And Mapping

The SLAM problem is a current research topic which is based on different localisation algorithms and using a range of different sensor to effectively map an target area. A lot of different approaches have been done and many research papers have been written, the one this project is based on is a paper about a SLAM solution designed for autonomous vehicles [5].
While the research area of this paper is based on a much larger scale, it does still give me an insight upon the SLAM problem.
E.g. the problem with localisation in an dynamic environment, the paper tackles this problem by using global reference points and a millimetre wave radar, however for my project I do use an static environment and simple laser range finders. So this paper is only used a reference to the localisation problem, especially the idea of using ”global” reference points for the created map.
As the test environment and the sensors available for the E-puck sensors are limited this project will assume that the starting location of the robots is known.

Another paper which was read about this problem used an approach much more similar to this project, by using different mobile robots which have no GPS access and simply use 2D laser range finders. However the approach described in this paper was based around the mapping of one ”lead” robot and the traversing the same map again with a second robot using the map generated

by the first for localisation purposes.
The second robot would then scan the target area again and refine the already generated map though using the (now stationary) first robot as an reference point. Since this project is using single a robot for mapping purposes and multi robot usage would only be added if enough time is available, this paper was not inherently useful, however gave some useful insight for information sharing between robots or sensor stations as well as localisation of robots using a global reference point.
Since this project aims at real time localisation and mapping communication with a ”uplink” point would be essential to achieve this in a real world setting. By using a similar implementation to the one described in the paper it would be possible to rescan a mapped area if it is traversed again, and by that refine the mapping. This is requires however very good localisation techniques.

1.0.2 Deployment

The deployment strategy is an important part of this project as it defines how effective the robot will cover the target area which will define how long it will take to scan and map the whole area. One research paper which was read proposed an solution of a communication network where the comm nodes keep track of the robots positions and guide them in directions which have not been explored in the last time period [2]. The paper uses a solution which is based on small comm nodes deployed by the robot, to make the solution fitting for this project the developer would have to use multiple E-Pucks and define some of these as communication nodes which remain on a fast position and guide the ”scout” E-puck based on area which have been least visited by the other robots.
However since multi-robot usage is planned as an addition if enough time is available this deployment strategy is not fitting for the major part of work.

1.1 Analysis

The background reading clearly showed the main aspects which were needed.
These were an effective deployment pattern, localisation of the robot and sensing of obstacles to map.

1.1.1 Deployment pattern

An effective deployment pattern is needed as it is important to traverse the whole of the environment in order to map all obstacles inside it.
Characteristics for a effective deployment pattern are to traverse the whole environment and doing so in as short a time frame as possible.

1.1.2 Localisation

Arguably the most important aspect of the project is to have a good localisation solution, since without the mapping will be ineffective. There are a number of different approaches which can

be done for this, papers which were researched showed approaches such as using global reference points, GPS and compass data as well as approaches which do not require data from sensors like GPS or compass modules.
One of the aspects wanted for this project is to be able to locate the robot without the use of GPS or compass data and rather use pure odometry calculations to reach a solution of the localisation problem.

1.1.3 Mapping

To be able to create a map of the environment the right sensing method needs to be found, i.e. different sensors perform different in similar environments, also the range of sensors varies a lot. The choice of sensors is limited as the work is based on the E-Puck robot platform. However the quality of the mapping is strongly dependent on the deployment pattern and the localisation solution.

1.2 Process

The life cycle model used for this project is ”Feature driven development”, as it seams more appropriate for this project as other models.

Chapter 2

Design

This section will describe the design as of the date of the project outline description.
This is the design which will be followed and tried to implement, however this is more of a guideline rather than exact plan since at this moment it is uncertain what the API and simulator are able to do and what is possible to implement inside the given time.

2.1 Environment Design

The final environment for this project will consist of one large room with obstacles placed in it.
The program will be tested on a number of different sized rooms with different amount and sized obstacles in it.

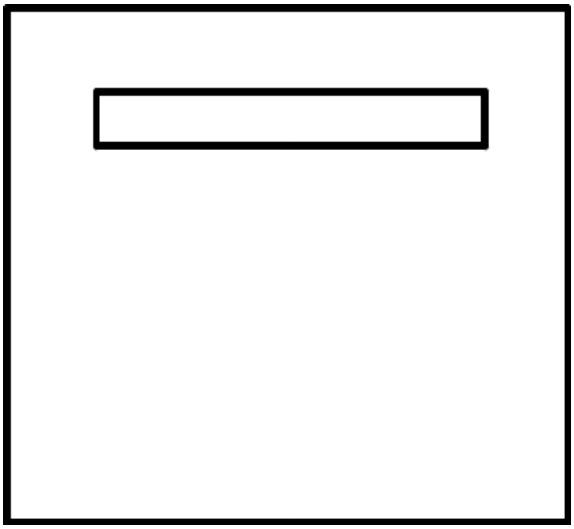

Figure 2.1: Environment Design

2.2 Mapping and Swarm Size

For mapping I will use a occupancy grid, and the occupancy will be acquired by the E-Puck's laser sensors. I will yet have to decide on a resolution, for the grid.
I will use a swarm of 5 E-Puck robots, of which 1 will remain stationary and only function as the "Uplink point" to which all robot send the acquired data. The stationary robot will also be used as reference point for the localisation method.

2.3 Deployment

It is at this moment still undecided which deployment strategy will be implemented.
There are 2 deployment strategies which are based on the background research which has been done.

2.3.1 Considered Deployment algorithms

One which is based on a random walk though the environment which will turn to a random heading when an obstacle has been reached.
This approach could be combined with a wall following algorithm which would trigger a random walk when the same position is reached again. This would allow for the complete traverse of a obstacle/wall. This would require a good localisation solution as without it the robot will be stuck inside an eternal loop.

The other deployment strategy would implement a more controlled movement pattern. This pattern would move the robots inside a rectangular pattern which would implement a function to move around obstacles on the way before moving back into the original pattern. While this reason is more controlled I am not sure which one will turn out to be more effective, that it why I will implement both inside a testing phase and will then decide which of them I will use.

2.3.2 Adopted design

It was decided to adopt and implement the more controlled movement pattern which can be seen in figure 2.2.
The reason for this being that the developer believes a random walking approach, because of its random behaviour, would be sub-optimal in a more complex environment whereas the controlled movement pattern would perform better. This is assumed as for future implementation more complex environments and possibly multi-robot usage is planned, however more about this can be found in chapter 5 on page 53.

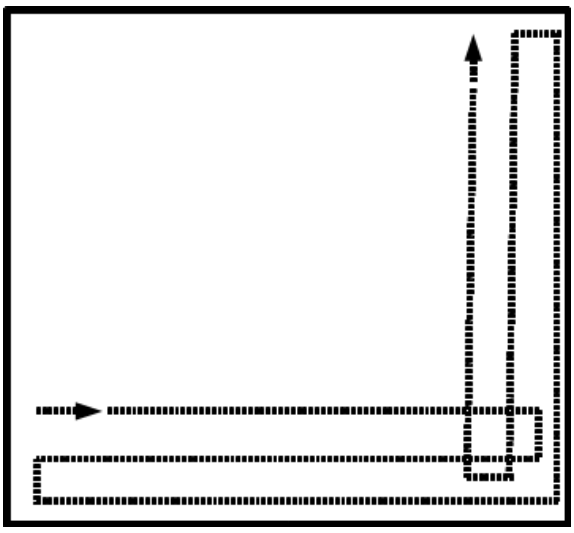

Figure 2.2: Controlled movement Pattern

Chapter 3

Implementation

3.1 Early Development

After the first weeks where background reading were done and designing a plan for the final program, work using the Webots™ simulator was started.
More information about the different stages can be found in the following sub-sections.

3.1.1 Webots Tutorials

In the beginning different tutorials and demo following programs included in the Webots installation were followed. This helped to see how the simulator works, what it can do and what possibilities the Webots API gives an developer.
The tutorials used were provided by the Webots wikibooks site [1].
Starting off with simple tutorials and movement and sensor reading the developer was soon able to implement simple programs which were based on simple forwards movement with obstacle avoidance functions, based on the proximity sensor values. After some of the more ”advanced”(in comparison to what had been done to this point) tutorials which involved e.g.: reading the robot encoders look at example programs of more advanced types of movement as well as SLAM examples were taken.

3.1.2 First Program Iterations

After that the experience of the tutorials was taken and it was started on to actually implementation the first prototype of the program.
The first prototype was still heavily based on a Webots example program called ”Intermediate Lawn Mower” which involved a simple movement pattern based on a basic finite state machine(FSM).
The first attempt was based recreating the FSM to get the same basic movement capabilities as the the program it was based upon and than changing it to achieve the level of movement control which was needed.

[1] `http://en.wikibooks.org/wiki/Cyberbotics\%27_Robot_Curriculum`

In the following iteration it was tried to implement the functionality of moving a given distance based on resetting the stepper motors encoders and moving the robot until it reaches a given encoder value. It was quickly realised that this would not work with the way the FSM was currently implemented so it was decided to move the functionality of the FSM, which was currently based on a simple switch statement, to a set of separate functions. The idea was that this would give the freedom needed to be able to call a method, i.e. *move_forward()*, and keep it running until a predefined encoder value has been reached.
However this did not work based on the overall way the program was designed at this point. It was designed around an switch statement which based its decisions on the proximity sensors of the E-Puck and then setting the movement speed for the robot motors. As it was based on this calling separate functions to move and turn did not work, which resolved in problems around the current idea of using the stepper motor encoders.

As it was needed to be able to move a certain distance in order to effectively implement the rectangular movement pattern, various approaches which were all based around the FSM were tried. It was realised soon that this was not getting anywhere which this way of thinking so it decided to come back to this problem later and went to another problem: turning a given number of degrees.
As the simulator simulates friction between the robot wheels and the environment the turning function, as it was in it current state, was far to inaccurate to be usable.
It was based on a very basic odometry calculation, taking the encoder values and calculating the turning distance based on the wheel diameter and axle length of the robot. While this is not a bad approach it had no procedures of slowing down the movement as it got closer to the target state so overshooting the wanted position or rotating far to less if the motor speed would have been set to low.
It was tried to counter the problem by alternating the values it used for the odometry calculations, and while it came close to a solution it was far from accurate. Another problem with this solution was that it only worked for 90 degree rotations based on the modified values, meaning all rotations to another point were impossible without modifying the values to fit the target heading. Which was counter productive as it would be best to have 1 function able to orientate the robot to any wanted heading.

3.2 Mid stage Development

After the problems which were encountered during the first program iterations, it quickly realised that this way of thinking and understanding of the API was flawed. A lot of the problems which were encountered were the result of insufficient programming skills for what was tried to do combined with bad understanding of how odometry worked, and how this could be used it to control the robot.
Seeing these kinds of problems it was decided to take a step back as the thought process for the program design was clearly ”moving in circles” and encountering the same kind of problems over and over again with the current approach.

It was decided to take once more a look at the provided example programs and to study their odometry functions in order to gain some new insight into odometry calculations, and find a new approach based on that.

3.2.1 Advanced tutorials

The example programs provided together with the simulator are categorised after the estimate level of knowledge needed to complete them. Theses categories are: Beginner, Novice, Intermediate and Advanced. The advanced programs were already looked at before as they feature a couple of examples on odometry and slam, however they were not understood well enough at the time.
Looking closer at the odometry functions provided it was managed to understand a bit better how the advanced programs worked and how they calculated the movement of the robot.

A look was also taken at the code and notes which were taken during the Robotics module on the second year, while the API worked different for the Player/Stage environment the idea behind rotation and movement was still the same and had only to be applied using the Webots API.

3.3 Movement

After having studied the odometry functions of the provided programs it was decided to start a new approach to fix the movement of the robot.
This approach is based on a set of different functions, much smaller and more refined than my previous approach. This approach took a while to implement and test but the results were rather satisfactory. This section is going to describe the different aspects of the movement solution and describe how the most important functions work.
The code of the major functions will be added as appendices, the site numbers will be added to each subsection. It is worth noting that the code displayed in the appendix is the final version of the code, and often updated in comparison to what they are at this stage in the development process.

3.3.1 Moving Forward

The code for this function can be found in Appendix B, section 2.1 at page 63.
It is worth noting at that the code display in appendix B is the final version, the earlier versions of this code did not hold the odometry update function *odometry_track_step()*. The *move_forward* function takes 2 doubles as parameters, 1 being the speed with which the robot is ordered to move and the distance it should move. The last parameter is a link to the global odometry struct, this struct is used to update the odometry values.
The function first checks that neither of the parameters are 0 and then calculates the number of steps each motor has to drive(1 step being 1 step of the stepper motor) to reach its target position. This calculation is done by dividing the number of steps needed for a full wheel rotation, 1000 in this case, by the product of π times the wheel diameter times 2 and multiplication this result with the distance defined in the parameter of the function. The steps needed for a full rotation have been taken from the E-Puck documentation and have been confirmed during on of the odometry

tutorials.
It will then read the current encoder positions and calculate the stop position for each motor by the sum of the encoder values for each motor and the previously calculated encoder steps needed to reach the target area. It will then set the motor speed based on the speed defined in the parameters.

It then enters the control code which will stop the robot once it reaches it target, position.
This is controlled by comparing the current encoder positions with the calculated target encoder positions and updating them all the time. Once the robot reached a pre-defined minimum distance of 20 encoder steps it will slow down the movement to a minimum speed of 10 steps per second. This is down to prevent the robot from overshooting the target area should it move with to much speed. The optimal minimum distance and speed has been found experimentally, and both values give good results and also prevent the robot from undershooting. Once it has reached it target location it will stop the robot, and force the simulator to take a simulation step, effectively moving onward to the next command.

This function allows the robot to move forward and stop after the predefined distance within a minimal error space, which will always exist given the friction simulated inside the simulator. This method required a lot of testing in order to get right as first versions did not include the control statement which slowed down the robot after a minimum difference between the encoders and the target encoder value has been reached. So the robot used to overshoot the target.
After the control statement was implemented it still required some testing and calibration of the minimum difference and speed values in order to avoid over and undershooting. However the found values work well and the movement error has been reduced to minimum.

3.3.2 Turn a given angle

The code for this function can be found in Appendix B, section **??** at page **??**.
The turn_angle function takes 2 doubles as parameters, one being the angle the robot will turn to the other the speed with which the robot will turn.
First the factor by which the robot will turn is calculated by dividing 360, the value of a full rotation, with the defined angle. Once the factor has been calculated it will then the number of steps the motor have to do until the target position is reached. This is done by dividing the product of the steps needed for a full wheel rotation, 1000, and the size of the wheelbase by the product of the calculated turning factor and 2 times the wheel radius. It will then use a function to return the current motor encoder positions. This function simply uses the Webots™ API and returns the values.

If the rotation angle defined as the function parameter is positive the robot will turn to the right.
When the robot is turning to the right it will calculated the stopping positions of the encoders by adding the calculated step count to the left motor encoder and subtracting it from the right encoder.

This will lead to the wheels turning against each other and will result in the robot turning on the spot rather than only moving 1 wheel to turn which would result in a displacement of the robot.
And then set the the speed of the motors using the given function parameter value. The right motor will receive a negated value so that it will turn backwards. It will then compare the left encoder positions and updated them all the time. Similar to how the forward movement function worked, it will detect when a given minimum difference between the current and target encoder values is reached and slow the robot down to a minimum speed.

If the rotation angle defined as the function parameter is negative the robot will turn to the left. The only difference between turning left rather than to the right is that the calculations, obviously, are reversed. Meaning to calculated the stop positions of the motor encoders it will subtract the calculated step count from the current left encoder value and add the step count the the right encoder value, same switch of negation has been done where the motor speeds are set. The calculations of how long to turn and when to slow down are identical to how they work when turning right, only difference being that the the operators to which check how long to turn are different.
Once the target position has been reached, by either turning left or right, the robot stops. It will then force to the simulator to take a simulator step, effectively moving on to the next command.

This functions allows me to define the turn the robot by so many degrees as I need and it will turn there within an minimal error space. This error space exist because the simulator simulates friction between the robot wheels and the environment so 100% accurate movement will never happen.
There also existed the problem of over/undershooting with the turning however the values found during tests of the *move_forward* function turned out to also work well for the turning function. However one problem remains, since there never is going to be a perfect rotation the error value will add up over time, resulting in less and less accurate turns, overshooting the target rotation is going to be a real problem. I have at this point not yet a solution for this problem, however the function works well and is a great improvement to how it turning was implemented in previous iterations of the program.

3.4 Localisation using Odometry

After the studying the optometry functions provided and implementing the movement algorithms, it was time to implement the localisation using odometry calculations.
The odometry functions which were implemented are used for localisation the robot inside the environment and finding it heading. The functions only require the starting point and localisation of the robot, and are then able to calculate the movement and rotation of the robot with every movement done, within a certain degree of accuracy. The uncertainty in accuracy is based on the friction which get simulated inside the simulator.
These functions are largely similar to the ones provided with the Webots™ interface, however some minor changes has been done.
Similar to the way in which the movement algorithms have been described the odometry functions are going to be described.

3.4.1 Initialising the Odometry algorithms

The code for this function can be found in Appendix B, section 2.4 at page 66.
The code for the odometry struct can be found in Appendix B, section 2.3 at page 66.
To initialize the odometry algorithms, 2 functions are used.
The first function, *odometry_track_start* takes a *odometrtyTrackStruct*, which is defined in the class which calls the function, as parameter. It will then acquire the encoder positions of the robot and call the next function, *odometry_track_start_pos* which set's the starting values, including the encoder positions inside the odometry struct.

Also the distance travel when a wheel turns and the wheel conversion are calculated, used for this are parameters acquired during calibrations done in the Webots ™ example programs(the values are the same as it is the same virtual robot model) and the E-Puck documentation.

3.4.2 Updating the Odometry values

The code for this function can be found in Appendix B, section 2.5 at page 67.
To update the odometry values of the struct the function *odometry_track_step* is called. When this function is called inside another function a number of things happen.
The current encoder positions for the stepper motors are fetched, and used as a parameter inside the *odometry_track_step_pos* function call.
Inside this function the new X, Y coordinates and the rotation of the robot are calculated. This is achieved by first calculating the difference between the current encoder positions and the encoder previous encoder positions saved inside the struct. This difference is then multiplied by the wheel conversion, which gets calculated inside the initialization step.
The result of this is used to calculated the wheel movement of the left and right wheel, results which are used to calculate the rotation of the robot. The next step is to calculate the new X and Y coordinates of the robot, based on the sum of done left and right wheel movement and math calculations using the robots rotation. At the end the calculated X and Y coordinates as well as the rotation value are used to update to struct. And the current encoder positions are saved inside the buffer for later calculations.

3.4.3 Minimum speed and distance calibration

After these functions were created they were calibrated using following processes .

Figure 3.1 shows a simple movement pattern which was used to calibrate the minimum turn distances and tested that the *move_forward()* and *turn_angle()* methods worked as intended.
How it works is the E-Puck starts on 1 corner of a square(here the floor of the simulator was used as it has chessboard representation) and move forwards to the other end of the square, turn 90 degrees and so on until it reaches it start point. It then turns around and follows the same pattern back to its original starting position. This tests allows the notice of odometry error in the rotations and forward movement, after a while it would turn/move to far or not far enough and thereby not reach it accurate start point. Noticing these errors allowed for calibration of the speed and distance where the algorithms will slow done the movement to minimize this error as much as possible, see

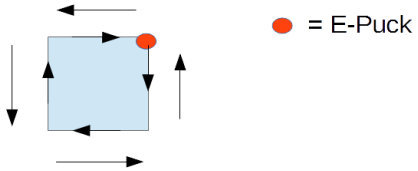

Figure 3.1: Movement algorithm test pattern

section 3.3.1 and 3.3.2 respectively for more information on this.

The test is based on the ”University of Michigan Benchmark” or UMBmark[2].

The problem with this calibration at the time was rather than using this approach to calibrate the effective wheelbase and the wheel radius at the time of this calibration the angles to turn were changed.
This led to rather than updating the wheelbase from the original 0.052(which has been done in a later stage of the project. **Note: the code which can be found in Appendix B uses the updated wheelbase**.
Using the non-updated wheelbase it was required to change specify an angle of 100° to move 90° effectively.

3.5 First results

After the calibration process of the previous section a new program iteration was started, using the new movement and localisation algorithm in order to test the mapping and continue implementing it.

This first implementation was not using the deployment pattern but rather simply following the walls, this was done so that the mapping could be tested. Rather than having an advanced wall following algorithm at this stage a simple collision avoidance algorithm was used which turned the robot by 90° when ever an obstacle is reached.

3.5.1 Early mapping methods and results

Listing 3.1: Proximity sensor reading

[2] `http://www.cs.columbia.edu/~allen/F13/NOTES/borenstein.pdf`

```
//distance sensors array definitions
int ps_value[NUM_DIST_SENS]={0,0,0,0,0,0,0,0};
int obstacle[NUM_DIST_SENS]={0,0,0,0,0,0,0,0};
int ps_offset[NUM_DIST_SENS] = {35,35,35,35,35,35,35,35};

// obstacle will contain a boolean information about a collision
for(i=0;i<NUM_DIST_SENS;i++){
      ps_value[i] = (int)wb_distance_sensor_get_value(ps[i]);
      obstacle[i] = ps_value[i] - ps_offset[i] > THRESHOLD_DIST;
}

//define boolean for sensor states for cleaner implementation
bool ob_front =
      obstacle[PS_RIGHT_10] ||
      obstacle[PS_LEFT_10];

bool ob_right =
      obstacle[PS_RIGHT_90];

bool ob_left =
      obstacle[PS_LEFT_90];
```

The code above shows the code used at this stage of the project to detect obstacles, and set boolean values if an obstacle is detected in the direction of the obstacle. As the test environment for this part of the project is a simple square area the turning of 90° is sufficient and no other control was needed at this stage to test the mapping.

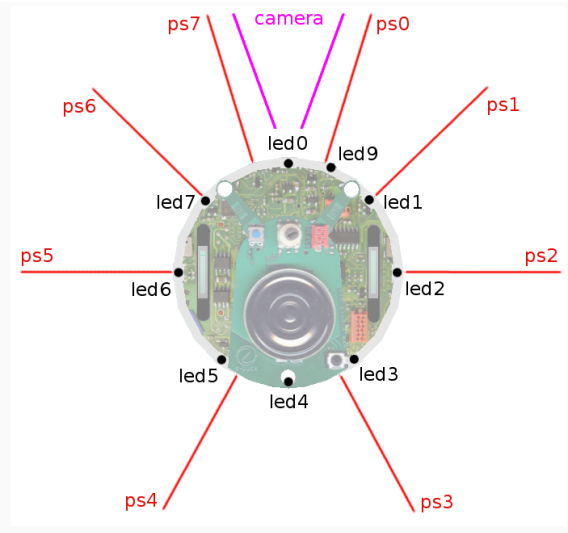

Figure 3.2: E-Puck sensor placement[3]

Figure 3.2 show the sensor placement on the E-Puck robot. Its is worth noticing that for sake of simplicity and better work flow the sensors have been renamed in the code. The new naming scheme does show which side of the robot the sensor is placed and its placement in degrees, rele-

[3]Figure 3.2 is taken from: `http://www.cyberbotics.com/cdrom/common/doc/webots/guide/section8.1.html`

vant to the forward facing center of the robot. As an example the sensor ps0 in figure 3.2 is named PS_RIGHT_10 in the code since it is placed on the right side of the robot with an orientation of 10° relevant to the center. Compared to this sensor ps5 is PS_LEFT_90.

The mapping algorithm used a similar approach to the obstacle detection algorithm.
One of the problems which were noticed at this stage was the problem of the noise simulated inside the simulator, this leads to the mapping algorithm needing a high threshold in order to map only actual obstacles, and not random noise spikes. This however leads to the problem of the robot needing to almost be in direct contact with the obstacles in order to map them. For the mapping at this stage this was not a problem however this strongly influenced movement pattern decisions at a later stage in the implementation process, more about that in a later section.

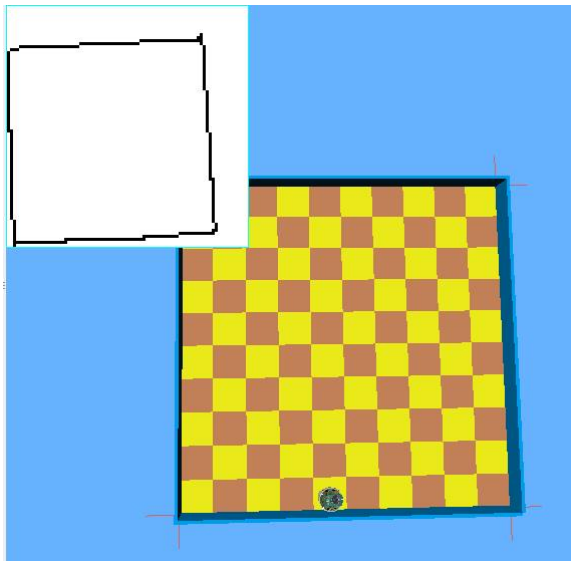

Figure 3.3: Early mapping result

Figure 3.3 shows clearly the strongest problems with this solution so far as well as demonstrates the need to have a good solution for the main aspects of this project, localisation and deployment.
As it can clearly be seen in the figure, the localisation and movement approach were far from perfect at this stage in the project. While the program generates a closed map the mapping is strongly skewed to the left side. This error was strongly based on odometry errors during turning caused by sup-optimal calibrated robot values as well as friction between the wheels and the floor.
At the time to movement algorithm did not update the odometry struct, and the struct were only updated once every loop iteration.
This was the stage of the project at the time of the mid-project demo.

3.5.2 Direction control based on heading

The approach done to fix the odometry error at this stage was done by specifying directions in which the robot is moving as north, east, south and west.

Figure 3.4 does show the directions used and the corresponding angles. The directions were

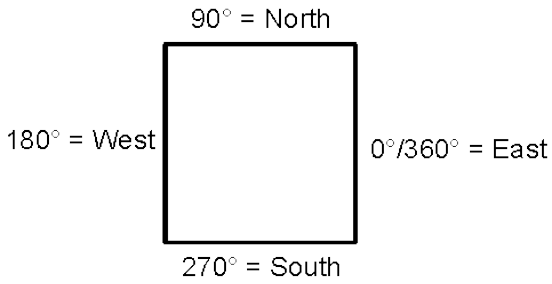

Figure 3.4: Directions and their corresponding Angles

represented in the code by using boolean values.
In order to get the direction controlled movement working a number of functions needed to be created.

Listing 3.2: Converting Radians to degrees

```
#ifndef M_PI
      #define M_PI 3.14159265358979323846264338327 95L
#endif

#define RTOD(r) ((r) * 180 / M_PI)

/**
returns the angle in which the robot is moving
*/
int return_angle(double rad){
      double rotation;

      if(RTOD(rad) < 0){
            rotation = RTOD(rad) + 360;
      }else{
            rotation = RTOD(rad);
      }
      printf("%f\n", rotation);
      return rotation;
}
```

This function converts radians, which are used by the simulator, to degrees.This was done to make it easier to work with. The next function uses this to check in which direction the robot is moving and set a boolean value representing a direction to *true*.

Listing 3.3: Early check of the movement direction

```
#define ANGLE_TOLERANCE 20
#define EAST 0
#define NORTH 90
#define WEST 180
#define SOUTH 270

//direction definitions
```

```
bool north,west,south,east;

/**
set booleans for the direction the robot is moving in
*/
void check_direction(double d){
      int i = return_angle(d);
      east = false;
      north = false;
      west = false;
      south = false;

      if(EAST < i + ANGLE_TOLERANCE || EAST > i - ANGLE_TOLERANCE){
            east = true;
      }else if(NORTH < i + ANGLE_TOLERANCE || NORTH > i -
         ANGLE_TOLERANCE){
            north = true;
      }else if(WEST < i + ANGLE_TOLERANCE || WEST > i -
         ANGLE_TOLERANCE){
            west = true;
      }else if(SOUTH < i + ANGLE_TOLERANCE || SOUTH > i -
         ANGLE_TOLERANCE){
            south = true;
      }
}
```

At this moment the deployment pattern described in chapter 2 was also implemented. This was done by checking which direction the robot is facing when an obstacle is found in front of the robot. At this point a couple of procedures were designed for what to do for each direction. Effectively the robot is performing u-turns when ever an obstacle is found directly in front of it.

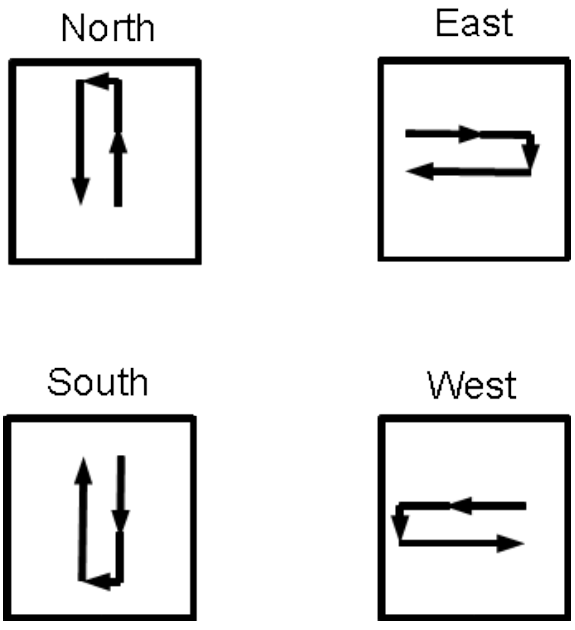

Figure 3.5: Directions and their corresponding U-Turn pattern

The u-turn patterns shown in figure 3.5 effectively led the robot to move from right to left and from the top downward, there wasn't any particular reason for this, other than an uniformed pattern was needed.
The next piece of code is a snipped from the deployment loop at that time in the project. It uses the previous shown *check_direction()* as well as the *controll_angle()* method which will be shown at a later point.
The control angle function is used to compare the current heading with the heading specified by the general direction the robot is moving in(north/east/south/west). As the code snipped shows at this point the the odometry struct is still not updated during the deployment loop, and the heading is only checked and corrected in the FORWARD state of the loop. The *turn_right()* and *turn_left()* functions turn the robot 90°.

Listing 3.4: Early Control loop snipped

```
double dMovSpeed = 500.0f;
double dDistance = 0.01f;
double dTurnSpeed = 100.0f;
double dTurnDistance = 0.05f;

switch(state){
      case FORWARD:
            move_forward(dMovSpeed, dDistance);
            controll_angle(&ot);
            if(ob_front){
                  state = STOP;
            }
            break;
      case STOP:
            stop_robot();

            if(ob_front && ob_left){
                  state = TURNRIGHT;
            }
            else if(ob_front && ob_right){
                  state = TURNLEFT;
            }
      else if(ob_front){
            check_direction(ot->result.theta);
            state = TURNRIGHT;
      }
      break;

      case TURNRIGHT:
            turn_right(dTurnSpeed);
            state = FORWARD;
            break;
      case TURNLEFT:
            turn_left(dTurnSpeed);
            state = FORWARD;
            break;
      case UTURN:
            if(north){
                  turn_left(dTurnSpeed);
```

```
                move_forward(dTurnSpeed, dTurnDistance);
                turn_left(dTurnSpeed);
                state = FORWARD;
            }else if(south){
                turn_right(dTurnSpeed);
                move_forward(dTurnSpeed, dTurnDistance);
                turn_right(dTurnSpeed);
                state = FORWARD;
            }else if(west){
                turn_left(dTurnSpeed);
                move_forward(dTurnSpeed, dTurnDistance);
                turn_left(dTurnSpeed);
                state = FORWARD;
            }else if(east){
                turn_right(dTurnSpeed);
                move_forward(dTurnSpeed, dTurnDistance);
                turn_right(dTurnSpeed);
                state = FORWARD;
            }
            east = false;
            north = false;
            south = false;
            west = false;
            break;
      default:
            state = FORWARD;
}
```

It is worth mentioning that this approach went through a number of modifications before this state was reached. The previous modifications were done to check to optimal placement of the *controll_angle()* function. It was found to be unproductive to have the check after every turn, as it did not matter on the small distances the robot was moving in the u-turn process.

Listing 3.5: Early heading correction algorithm

```
 /**
Controll the movement angle
*/
void controll_angle(struct odometryTrackStruct *ot){
      int rotation;
      int dSpeed = 100.0f;
      int corAngle = 5.0f;

      if(RTOD(ot->result.theta) < 0){
            rotation = RTOD(ot->result.theta) + 360;
      }else{
            rotation = RTOD(ot->result.theta);
      }

      //check movement to the right(east)
      if(EAST < rotation + ANGLE_TOLERANCE){
            turn_angle(-corAngle, dSpeed);
```

```
        }else if(EAST > rotation - ANGLE_TOLERANCE){
              turn_angle(corAngle, dSpeed);
        }

        //check movement to the north
        if(NORTH < rotation + ANGLE_TOLERANCE){
              turn_angle(-corAngle, dSpeed);
        }else if(NORTH > rotation - ANGLE_TOLERANCE){
              turn_angle(corAngle, dSpeed);
        }

        //movement to the west
        if(WEST < rotation + ANGLE_TOLERANCE){
              turn_angle(-corAngle, dSpeed);
        }else if(WEST > rotation - ANGLE_TOLERANCE){
              turn_angle(corAngle, dSpeed);
        }

        //movement to the south
        if(SOUTH < rotation + ANGLE_TOLERANCE){
              turn_angle(-corAngle, dSpeed);
        }else if(SOUTH > rotation - ANGLE_TOLERANCE){
              turn_angle(corAngle, dSpeed);
        }
}
```

The code above is an early version of the algorithm which checks and corrects the heading the robot is moving in. This algorithm was used to correct the odometry error caused by the friction in the environment and the, at this stage, sub-optimal calibrated robot attributes.
It worked by comparing the heading the robot is heading in with the wanted heading and correcting any errors. The algorithm at this stage has some short comings such as it is not possible to turn the robot more accurate than with a 5° threshold. If the threshold was reduced to less than 5°, the robot would constantly overturn.

At this time it was assumed that this inaccuracy was caused by the friction simulated inside the simulator, and that this movement was as accurate as it was going to be.
The odometry error at this stage did improve thanks to the *control_angle()* functions, the start of the mapping was much less skewed than the previous approach, which result can be seen in function 3.3 at page 15.

However the constant odometry error did accumulated over time, so that the location displacement became to big to able to map the environment effectively.
In order to try to fix the localisation error, a couple of approaches were done in order to reduce said error. Before those approaches were done some refactoring and code improvement was done.

3.6 Refactoring and improvement

In this section the refactoring and code improvements which were done at this stage of the project is described.
Some of the refactorings I have not addressed in this section are the movement of functions to other source files. While it clearly is good practise to create functions at the right place in the first place, a lot of prototype functions were created in the main source files for sake of simplicity during the first functions test. These functions were later moved to other source files were they are more appropriate, and to keep the overall source file size done.

3.6.1 Deployment algorithm refactoring

Before other approaches were tried it was decided that the deployment algorithm needed to refactored to make the code cleaner and easier.
The following piece of code is a snipped taken from one of the algorithm:

Listing 3.6: Deployment algorithm refactoring

```
double dSpeed = 500.0f;
double dDistance = 0.01f;

char no[] = "north";
char ea[] = "east";
char we[] = "west";
char so[] = "south";

case UTURN:
      if(north){
            printf("%s\n", no);
            turn_left(dTurnSpeed);
            for(it = 0;it < 5;it++){
                  move_forward(dSpeed, dDistance);
            }
            turn_left(dTurnSpeed);
            north = false;
            state = FORWARD;
      }else if(south)[...]
```

As the above snipped shows the unneeded variables of *dTurnSpeed* and *dTurnDistance* were removed as it was decided that the algorithm, for simplicities sake, use a single value type which can easily be changed. In order to get the same amount of movement during the u-turn phase a simple *for loop* is used. The same modifications as are shown in the code snipped have been done to the rest of the u-turn *case*.
In order to improve the code further, printout statements were added to the u-turn routine as well as to the *return_angle()* function. This needed to be done as the webots simulator does not have a debugger, so printouts were used to track to results of the algorithms.

3.6.2 Deployment algorithm improvement

The previous u-turn method did not support mapping obstacles while the u-turn was performed. As this is not optimal behavior the u-turn pattern was altered again, to support the mapping methods.

Listing 3.7: U-turn improved with obstacle detection and mapping

```
#define NUM_DIST_SENS 8 //number of distance sensors
#define OCCUPANCE_DIST 150 //obstacle mapping threshold
double dSpeed = 500.0f;
double dDistance = 0.01f;

static int wtom(float x){
      return (int)(MAP_SIZE / 2 + x / CELL_SIZE);
}

robot_x = wtom(ot->result.x);
robot_y = wtom(ot->result.y);

char no[] = "north";
char ea[] = "east";
char we[] = "west";
char so[] = "south";

case UTURN:
      if(north){
            printf("%s\n", no);
            turn_left(dSpeed);
            for(it = 0;it < 5;it++){
                  move_forward(dSpeed, dDistance);

                  //mark cells as occupied
                  wb_display_image_paste(display,background,0,0);
                  wb_display_set_color(display,0x000000);
                  for(i = 0;i < NUM_DIST_SENS;i++){
                        if(wb_distance_sensor_get_value(ps[i]) >
                            OCCUPANCE_DIST){
                               occupied_cell(robot_x, robot_y,
                                   ot->result.theta + angle_offset[i]);
                        }
                  }
                  wb_display_image_delete(display,background);
                  background = wb_display_image_copy(display,
                        0,0,display_width,display_height);
            }
            turn_left(dSpeed);
            north = false;
            state = FORWARD;
      }else if(south){[...]
```

The above code shows how the obstacle detection and mapping were added to the u-turn *case*. This map's the obstacles to the sides of the E-Puck while the robot moves along the side of it and proceeds to the next turn. While this mapping is not good, as it is very "spotty".

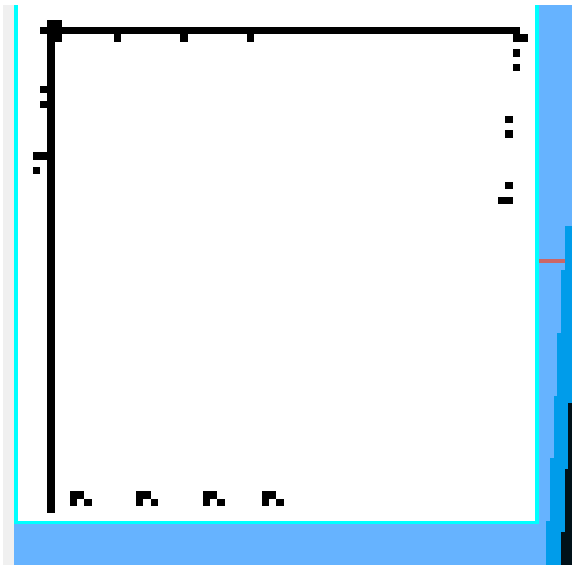

Figure 3.6: Dotted mapping during U-Turns

Figure 3.6 shows the dotted mapping which is caused during u-turns. While it is in no form a good representation of the environment it provides an outline of obstacles. The problem with this is however that because of the u-turn pattern it is needed for the robot to traverse a long each obstacle / wall in order to map them correctly.

3.7 Mapping

In this section the mapping algorithm are shown and described.
The mapping algorithms are one of the key algorithms of the project, however based on code from the Webots™ demo programs It is based on a occupancy map, which is a simple 2D map. The function used to mark a *cell* of the occupancy map.
The code for this function can be found in Appendix A, section 1.1 on page 61. The parameters for the function represent the X and Y coordinates as well as the rotation of the robot. These values are received from the odometry struct. The functions uses the Webots™ API to draw the cell in form of a 1x1 pixel rectangle on the display.

Listing 3.8: obstacle detection and mapping

```
#define NUM_DIST_SENS 8 //number of distance sensors
#define OCCUPANCE_DIST 150 //obstacle mapping threshold

static int wtom(float x){
      return (int)(MAP_SIZE / 2 + x / CELL_SIZE);
}


robot_x = wtom(ot->result.x);
robot_y = wtom(ot->result.y);

wb_display_image_paste(display,background,0,0);
```

```
wb_display_set_color(display,0x000000);
for(i = 0;i < NUM_DIST_SENS;i++){
      if(wb_distance_sensor_get_value(ps[i]) > OCCUPANCE_DIST){
            occupied_cell(robot_x, robot_y, ot->result.theta +
                angle_offset[i]);
      }
}
wb_display_image_delete(display,background);
background =
    wb_display_image_copy(display,0,0,display_width,display_height);
```

The code above shows for loop which checks the value reported by each 1 of the 8 distance sensors and compare this value to the defined threshold. If the value is larger than the threshold the cell will be marked.

3.8 Refactoring of the heading correction

In this section the refactoring of the heading correction functions is shown. The earlier version of this code can be seen in listing 3.5 on page 19. The earlier version of this code snipped worked, however it was way more complex than it had to be.

Listing 3.9: Refactored heading control code

```
/**
Function to compare the current heading to the wanted heading
and fix the heading should it surpass a threshold
*/
void check_rotation(double cur_rot, double want_rot, double dSpeed){
      double dthreshold = 1; //2,0
      double diff;
      char text[] = "Correcting";

      if(cur_rot + 20 >= 360){
            cur_rot -= 360;
      }

      if(cur_rot > want_rot + dthreshold){
            diff = cur_rot - want_rot;
            printf("%s\n", text);
            turn_angle(diff, dSpeed);
      }else if(cur_rot < want_rot - dthreshold){
            diff = want_rot - cur_rot;
            printf("%s\n", text);
            turn_angle(-diff, dSpeed);
      }
}
```

The newer version of this function only requires parameters with defines the wanted and the previous heading rather then the need of having global boolean values to specify in which direction the robot is moving. This refactored approach works by comparing the wanted and current

heading, an threshold is also included as the robot will never turn 100% accurate.
The function also converts degrees which are above 360 °, to a more sensible value. The code holds printout statements which allow to understand in which direction the robot thinks it is moving, which can be compared to the actual direction for testing purposes.

After this function was created the threshold was calibrated in a couple of runs. The threshold which was set in the first iteration, and was gradually decreased to a value of *3* at the time. The threshold value shown in the code snipped above is *1*, this value was reached by recalibrating the turning and movement functions, more about that can be found in section **FIXME(enter section number)**.

3.9 New approach tried and implemented

In this section the different approaches which were tried at this development stage are described. While not all approaches were successful not every aspect of them was discarded.

3.9.1 Dynamic U-Turn approach

At this point in the development process a new approach for turning the robot in the U-turn routine was tested. The idea was to change from using a static turning command *turn_left(dSpeed)*, which can be seen i code snipped 3.7 on page 22, to a more dynamic approach.
In this new dynamic approach the angle to turn was calculated from the current rotation and the wanted angle.

Listing 3.10: Code snipped of the new approach

```
double cur_rot;

else if(south){
      printf("%s\n", so);
      turn_right(dTurnSpeed);
      for(it = 0;it < 5;it++){
            move_forward(dTurnSpeed, dDistance);
            //mark cells as occupied
            wb_display_image_paste(display,background,0,0);
            wb_display_set_color(display,0x000000);
            for(i = 0;i < NUM_DIST_SENS;i++){
                  if(wb_distance_sensor_get_value(ps[i]) >
                      OCCUPANCE_DIST){
                         occupied_cell(robot_x, robot_y,
                             ot->result.theta + angle_offset[i]);
                  }
            }
            wb_display_image_delete(display,background);
            background =
                wb_display_image_copy(display,0,0,display_width,display_height);
      }
      odometry_track_step(ot);
```

```
        cur_rot = return_angle(ot->result.theta);
        turn_angle(cur_rot - 90, dTurnSpeed);
        //turn_left(dSpeed);
        south = false;
        state = FORWARD;
}else if(west)[...]
```

The code in listing 3.10, is a snipped from the U-turn routine.
However this approach did not work out as the robot always turned to far. Rather than getting stuck on this new approach, it was discarded and work was resumed with the old, static version of the code where the robot simply always turns 90°.

3.9.2 Stop on corner detection

At this stage it was noticed that some of the existing odometry error existed because the robot did not stop when an obstacle, like a corner was encountered in the U-turn routine. What actually happened is that the robot wheels kept turning, increasing the encounter values, which let to wrong odometry calculations.
To fix this problem corner detection was added to the algorithm. It is worth noting at this point that rather than only corner detection, general obstacle detection should have been added. However, as it described in chapter 4, the program was only ever tested in the same environment during development which let to many of the problems encountered in the tests. The reason that only corner detection was added at this point was because it was known that the only obstacle which will be encountered in the U-turn routine are going to be corners, so no other type of obstacles was even considered.

In the first iteration of this approach is corner detection was added **bellow** the section of code which mark's cells as occupied. Test runs quickly showed that it works better **before** said section, so the corner detection was moved.
The snipped shown in listing 3.11 is the final version with the corner detection above the cell marking section.

Listing 3.11: Corner detection added to the U-turn routine

```
if(north){
        printf("%s\n", no);
        turn_left(dSpeed);
        for(it = 0;it < 5;it++){
                if((ob_front && ob_right) || (ob_front && ob_left)){
                        state = STOP;
                        break;
                }
                move_forward(dSpeed, dDistance);

                //mark cells as occupied
                wb_display_image_paste(display,background,0,0);
                wb_display_set_color(display,0x000000);
```

```
            for(i = 0;i < NUM_DIST_SENS;i++){
                  if(wb_distance_sensor_get_value(ps[i]) >
                     OCCUPANCE_DIST){
                        occupied_cell(robot_x, robot_y,
                            ot->result.theta + angle_offset[i]);
                  }
            }
            wb_display_image_delete(display,background);
            background =
               wb_display_image_copy(display,0,0,display_width,display_height);
      }
      turn_left(dSpeed);
      north = false;
      state = FORWARD;
}else if(east){[...]
```

3.9.3 Added odometry update function to the movement algorithm

Until this stage the odometry values were only updated every control loop iteration.
In order to increase the number of odometry updates, which needed to be done in order to make the localisation more accurate, the odometry update function needed to be called more often then just once every iteration. This was done in an approach to reduce the overall odometry error by updating the *odometry struct* more often, before odometry errors has a change to accumulate further.

In order to implement this the odometry updated function was added to the *move_forward()* function. The code which can be found in Appendix B section 2.1 on page 63 is the version of the code with the odometry update function added to it.
While this updated function did not show any significant decrease of the localisation error, it was kept as updating the odometry functions as often as possible is helps decreasing the overall localisation error.

3.10 Recalibration of the movement algorithms

The approaches described in the previous section allowed to reduce the overall odometry error by small degrees, however not entirely remove it.
Research was done of how to reduce the odometry error and a paper written by Johann Borenstein and Liqiang Feng was found [3].

In this paper *Borenstein et al.* described why and how odometry error are generated and how to minimise them by having a good calibrated robot. They showed ways of calibration the parts of the robot which cause odometry error when turning and driving forward, the *wheelbase* and the *wheel diameter*. They also described what kind of errors can be generated by not having an good calibrated robot.
The information displayed in the paper will not be repeated in this document, however the main

aspects of the problems and of the calibration will be discussed.

There are 2 types of errors which are discussed in the paper, *systematic errors* and *non-systematic errors*. *Systematic errors* are caused(or prevented) by values which can be fixed in software. Such error sources are:

- unequal wheel diameters
- misalignment of wheels
- limited encoder resolution
- limited encoder sampling rate

Just to mention some.
Non-systematic errors are caused by environmental and physical problems with the problems. Such error sources are:

- slippery floors
- over-acceleration
- fast turning(skidding)
- nonpoint wheel contact with the floor

Just to mention some.
Non-systematic errors can be in this case be avoided because it known that the floor in the simulator is not slippery and there is also always wheel contact to the floor. To reduce the possibility of over-accelerating and skidding the moving and turning speed has been reduced in the program.
However *systematic errors* are more grave then *non-systematic*, this is because they accumulate constantly.

3.10.1 Wheelbase and Wheel diameters

The paper describes that the 2 most notorious error sources are *unequal wheel diameters* and *uncertainty about the effective wheelbase*.
Unequal wheel diameters are created in reality because of inaccuracy of the produced rubber wheels in robots. It is known that the wheels of the virtual E-Puck representation have unequal wheel diameters, this was discovered during the tutorials done in the very beginning of the development process.
For the duration of the entire development process up to this point, the values provided in these tutorials have been used. However in the beginning of the development process when the turning algorithm was implemented, a value of 100° was needed to turn 90°.
This was because of uncertainty about the wheelbase, this however was not realised until this point.

In order to describe what kind of errors uncertainty's about the wheel diameters and the wheelbase cause, the paper is going to be summarized.

Unequal wheel diameters:
While unequal wheel diameters do not cause an orientation error while turning, they will cause an curved movement path rather than a straight path.
The error can be denoted as :
$E_d = D_R/D_L$
Where D_R are D_L are the *actual* wheel diameters. The *nominal* ratio between the wheel diameters is of course 1.00 [3].
The values of the E-Puck wheels in the simulator are:
$D_R = 0.0404 = 40.4mm$
$D_L = 0.0416 = 41.6mm$
These values have been taken from the Webots™ tutorials and been confirmed in the later calibration process.

Uncertainty about the Wheelbase:
The wheelbase is defined as the distance between the drive wheels of the robot. This wheelbase must be known in order to compute the correct amount of encoder steps each motor must take in order to turn a certain amount of degrees. Uncertainty about this wheelbase can cause inaccuracy while turning, but does not have any effect on straight movement.
This error can be denoted as:
$E_b = b_{actual}/b_{nominal}$
Where b is the wheelbase of the vehicle [3].
The value of the wheelbase provided by the Webots™ tutorials is:
$0.052 = 52mm$
However the calibration process which is shown later shows that actual wheelbase is:
$0.058 = 58mm$
Which is an significant amount of uncertainty.

The paper by *Borenstein et al.* describes the main danger about uncertainty of either the *wheel diameters* or the *wheelbase* as that modifying one of them can cause the other value to seem calibrated as well. This can be caused by using an *unidirectional* square path as a benchmark test, which is what have been done in the beginning of this project, see 3.4.3 on page 12 for more information.
The danger with this approach is that it is easy to analyse the odometry error being caused by either the wheel diameters, a slightly curved movement path, or the wheelbase which would caused over or under turning.
If only an unidirectional calibration approach is done either of these errors can be ”fixed” by modifying either the wheel diameters or the wheelbase and the result would show a ”correct” result. For example if the actual movement path is as displayed by the dotted line in figure 3.7, the movement error could be ”fixed” by modifying the wheelbase which would led to robot turn, say 93°, which would cause the path to appear correct. By doing that the path will look correct and thereby the robot calibrated, but in reality the robot overturns in order to compensate for the curved movement

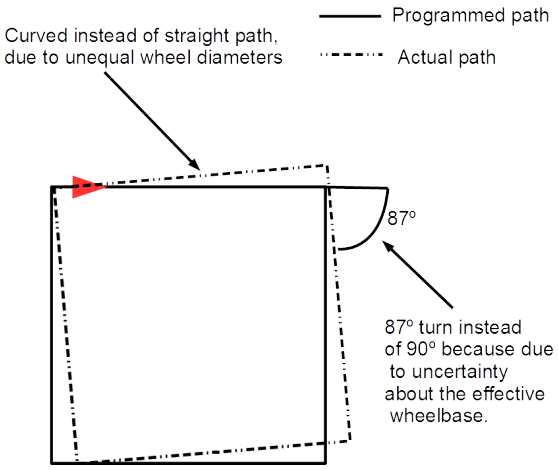

Figure 3.7: The effect of systematic odometry errors

path.
However this will lead to increased odometry errors as the robot will appear to turn accurately but the odometry error is in reality caused by the uncertainty about the wheel diameters which leads to a curved movement path. Of course this can also happen the other way around.

3.10.2 Bidirectional Square Path calibration: ”UMBmark”

As the preceding section shown an *unidirectional* square path is unsuitable for testing odometry performance as it can easily conceal 2 mutually compensating odometry errors.
To overcome this problem, the paper introduces a *bidirectional* square path experiment, called the *University of Michigan Benchmark* or for short *UMBmark*. In this experiment the robot moves both clockwise and counter clockwise in a square path. This way it easily shows the concealed odometry error which is caused when the robot has been calibrated by only an unidirectional approach.

Figure 3.8 shows the result of running a robot which has been ”calibrated” with the *unidirectional* approach in the opposite direction. It shows clearly the short comings of such ”calibration” as it now overturns in addition to the curved movement path.

The paper describes methods which allow of accurate mathematical solutions to the given problems. But since the Webots™ simulator does not posses inbuilt tools to measure robot movement accurately no measurements can be taken to which would allow to follow this procedure.
What has been done to calibrate the E-Puck robot is to implement the UMBmark functions inside the project and run them, altering the *wheel diameter* and *wheelbase* values over time and compare the results.

Listing 3.12: The UMBmark experiment procedure

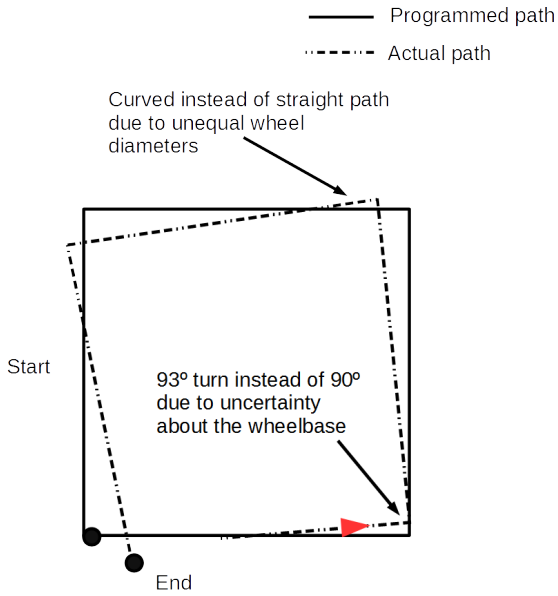

Figure 3.8: The odometry error when the unidirectional path is performed in the opposite direction

```
double dSpeed = 300.0f; //movement speed
double dDistance = 0.5f; //movement distance of 1 square. These
    distances have been changed over time

/**
University of Michigan Benchmark
*/
void UMBmark(double dSpeed, double dDistance){
      //move the robot clockwise and counterclockwise
      measure_clockWise(dSpeed, dDistance);

      measure_CounterClockWise(dSpeed, dDistance);

      stop_robot();
}
/**
Function to measure the movement accuracy by driving
a clockwise square.
This is part of the UMBmark algorithm
*/
void measure_clockWise(double dSpeed, double dDistance){
      int i, j;

      for(i = 0;i < NUMTOURNAMENTS; i++){
            for(j = 0;j < 4;j++){
                  move_forward(dSpeed, dDistance);
                  turn_right(dSpeed);
            }
            wb_robot_step(TIME_STEP);
      }
}
```

```
/**
Function to measure the movement accuracy by driving
a counter-clockwise square.
This is part of the UMBmark algorithm
*/
void measure_CounterClockWise(double dSpeed, double dDistance){
      int i, j;

      //turn the robot right for moving the same square counter clock
         wise
      turn_right(dSpeed);

      for(i = 0;i < NUMTOURNAMENTS;i++){
            for(j = 0;j < 4; j++){
                  move_forward(dSpeed, dDistance);
                  turn_left(dSpeed);
            }
            wb_robot_step(TIME_STEP);
      }
}
```

Listing 3.12 shows the code used for the UMBmark algorithm which is implemented inside the project. The start values are to move with a speed of *300*, which is an acceptable speed as well as moving a distance of *0.5*, which equals FIXME(check how many square this equals).
As a reminder the start values for the wheel diameters are:
$D_R = 0.0404 = 40.4mm$
$D_L = 0.0416 = 41.6mm$
Where D_R represents the right wheel diameter and D_L the left wheel.
The value of the wheelbase is:
$0.052 = 52mm$

Firstly the amount of degrees which the robot turns when the *turn_left()* and *turn_right()* methods has been set from 100° to 90°. The simulation was then been run 10 time times per test iteration to assure that the robot movements are not caused by random effects.
Between each iteration either of the wheel diameter and wheelbase values has either been decreased or increased in the simulation then been run for another 10 times. After each test iteration of 10 runs the value was either set back to its original value of the odometry error increased or if the error decreased the value was further changed.
At a later point when the odometry error seamed resolved multiple values were changed, in order to assure that they are correct.
The resulting result was that the wheel diameters are correct however the wheelbase value was updated from:
$0.052 = 52mm$
to: $0.058 = 58mm$
As it is simple to check the current calibration for a curved movement error the robot was moved the whole length of the simulation range to confirm that the movement path is indeed straight and not curved, which it was. This was done to recheck that the result from the UMBmark calibration are correct.

The paper suggest using larger movement patterns, where available, in order to check to correctness of the calibration. As this was easily done in the simulator the square was increased from 1x1 floor square to 2x2 and then 4x4. The results were largely the same, however by a square size from 4x4 the robot did not move full 4 squares. This shows that the squares on the floor, which until now have been expected to be FIXME(Check these values) 0.5 in code or equal to FIXME(add size), are not exactly that value but something close to it.
Because of the rest amount of time available was not big by the time of this calibration process the calibration results from 1x1 and 2x2 square paths have been deemed sufficient, rather than running multiple experiments trying to figure out the exact size of the squares.

3.11 Reference point approach

After the E-Puck values have been calibrated with the method described in the preceding section, the movement improvement has been checked in the development environment.

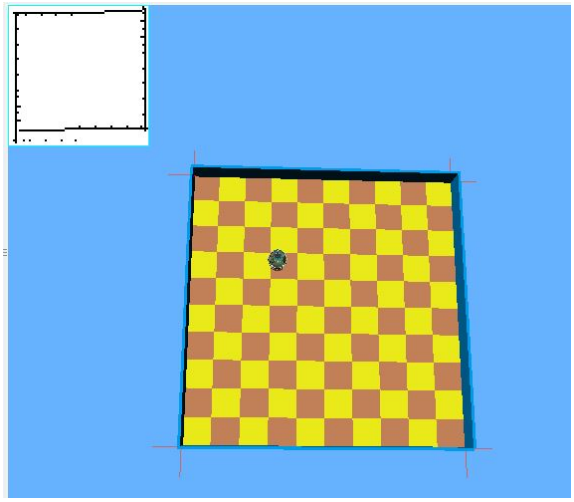

Figure 3.9: Improved mapping with the calibrated robot, but the localisation error still persists

Figure 3.9 shows the mapping result generated with the calibrated movement.
It shows that program is now able to generated a map of the environment with straight walls, compared to the much more skewed wall result which were generated before. FIXME(Rollback github repo and add screenshot about it)

However it is not entirely perfect at this stage as an localisation error still persists, as the robot while mapping the lower wall maps it inside the wrong place.
This is the result of noise generated inside the environment as well the nature of odometry calculations. Odometry is always, in best case, an estimate of the localisation, this combined with noise and friction inside the simulator environment caused the robot to never turn 100% accurate.
The algorithms implemented until this point, which compare the heading of the robot and turn it to fit the wanted heading as well as the recalibrated movement algorithms made huge improvements

to the robots movements.
But still a minimal odometry error persists, which accumulates over time and cause the odometry error which can be seen in figure 3.9.

At this point the use of reference points was considered in order the fix the localisation problem.
The approach taken for the reference point was to use the corners of the room as reference points. This would be achieved by creating new functions and a *struct* which use the already existing corner localisation implemented inside the U-Turn routine and save the X and Y coordinates the robot has calculated. That way the robot can reset it's X and Y coordinates to the coordinates previous saved inside the struct once the same reference point is reached again.
In the following subsections the functions created for this approach are being described.

3.11.1 Reference point struct

The code for the reference point struct can be found in appendix B section 2.6 on page 69.
To accommodate the idea of using the corner as reference points as structure was created which hold 4 members: *lower_left*, *lower_right*, *upper_left* and *upper_right*, where each member represents a corner of the environment.
Every member of the struct holds 2 float variables, which represent the X and Y coordinates of the corner. It is worth noting that this struct has been created in correlation with the development environment which is a simple room with 4 corners.

3.11.2 Checking for reference points

The code for this function can be found in appendix B section 2.7 on page 69.
It is called every time the main control loop finds a corner.
This function represents the major part of the algorithm. It compares the distance sensor data to specify which corner has been encountered and after that controls if the corner has already been saved. Should that not be the case the function *set_reference_point()* is called. This function will save the reference point to the struct, and is described further down.
If the found corner is already set as a reference point and the robots positions is within a threshold of 20, the function will reset the robots current X and Y coordinates to the values stored inside the struct. The threshold is used to make sure that the it is the same corner, rather than an unset corner inside an environment with more than 4 corners.
This function hold print statements which state at which corner the robot thinks it is resetting and to what coordinates.

3.11.3 Saving a new reference point

The code for this function can be found in appendix B section 2.8 on page 73.
This function is used to save the coordinates of a corner to the reference struct and thereby saving it as a reference point.
When the function *check_reference_points()* encounters a new corner which has not been set this

function is called. It requires a pointer to the odometry struct to be able to read the current X and Y coordinates of the robot, as well as an integer value which represents which corner has been found. This is done in the function *check_reference_points()* and then simply passed as a parameter to this function.
This function then simply read the current coordinates and saves them to the corresponding values of the reference struct.
This function also holds print statements to show when the algorithm saves a new corner and with what coordinates.

3.12 Mapping results using the new reference point approach

After implementing the reference point approach discussed in the previous section the mapping results improved massively.
Figure 3.10 shows the mapping results after the reference point algorithm has been implemented. The improve can easily be seen by comparing figure 3.9 on page 33 with the result of figure 3.10.

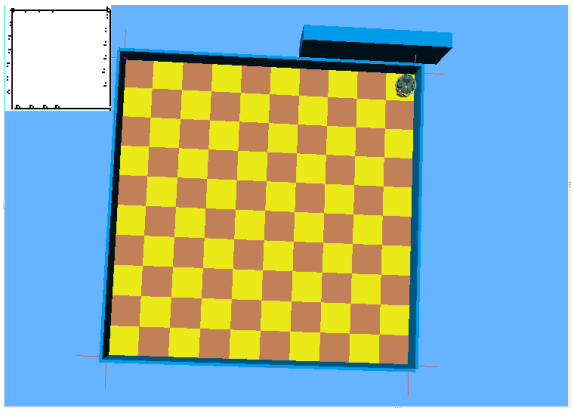

Figure 3.10: Results after implementing the reference point algorithm

Chapter 4

Testing

This chapter holds the overall testing results and test information about the final program. During implementation the program was only ever run in a single environment, **room 1**. In this chapter the results of running the program in different environments are documented. The environments differ in shape and size as well as well as in obstacle population.

4.1 Overall Approach to Testing

As the program requires an simulator to be run in, it is not possible to test the program in any other way than to run it in an simulator and document the results. This factor makes also automatic unit testing impossible. During implementation the program was only ever tested in a single environment called **room 1**. All changes which were done to the code and problems which were documented in early chapters have been the result of the robots performance in this environment.

The reason the program was never tested in different environment during implementation was since the program was not complete enough, it was assumed that before the program can be tested in multiple environments a solution for a simple environment must exist.
It is worth noting at this point that the project is **not** considered finished, however time restrictions do not allow for continued development.

4.2 Test environment: Room 1

Room 1 is the ”original” environment in which the program was implemented.
It is a simple, empty, room which size is 10x10 squares, figure 4.1 shows this environment. In order to measure the environment size the chess board pattern which makes up the floor of the simulator environment is used to measure it. **FIXME(enter size of a square in code)**

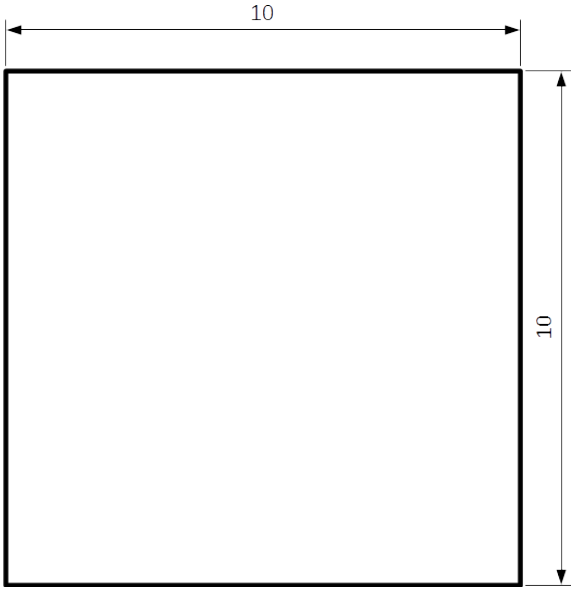

Figure 4.1: Room 1 empty test environment

4.2.1 Obstacle free test

The results gathered in this test environment, as it is the simplest, as well as the environment in which the program was developed are by far the best in from of final result.

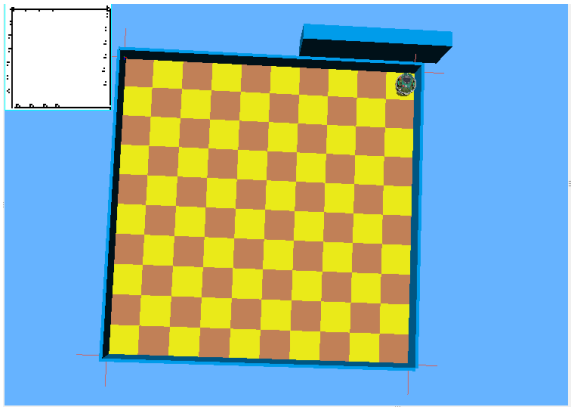

Figure 4.2: Room 1 empty environment result

Figure 4.2 shows the result of the environment after the map has been ”closet”. The resulting map clearly depictures the spotty nature of the U-turn mapping which is described in Chapter 3 subsection 3.6.2 on page 22 .

The generated map shows that the program is able to generate straight and accurate mappings of an environment as long as the robot is able to reset its odometry values at some point during the run. More info of what can happen if the odometry is not reset can be found **FIXME(enter**

section of page of describe accumulated odometry error).
The reason of why the program performs this well in the test environment compared to other environment is that is able to reset its odometry values by passing **corner 1**, before it moves to the uncharted **corner 2**. More information about the corner approach can be found **FIXME(link to section and page of corner approach describtion)**.
That the accumulated odometry error was growing constantly during program runs can be seen at the placement of the spots of the u-turn routine, these spots are marked red in figure 4.3.

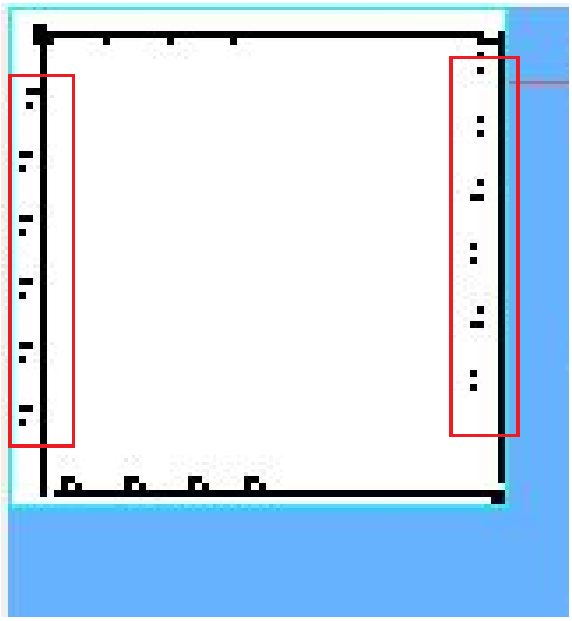

Figure 4.3: Room 1 map result with marked odometry dislocation

If these spots are compared to the one at bottom and top site of the generated map, it can be clearly seen that the odometry location error was crowing, and became very noticeable after the map had been generated about 50%.

However the odometry location error was reset as the robot reached **corner 1**, which X and Y coordinates were saved as the robot traversed said corner for the first time.
This reset the odometry values of the odometry struct and made it possible to continuously map the environment with good results.
FIXME(enter figure and page of the figure which shows the location error).

However this test environment is not without shortcomings either. If the program is allowed to keep working and mapping the environment the odometry error is increasing again.

Figure 4.4 shows the accumulated odometry error after the environment has been traversed approximately 2 times.
As can be seen in this figure the odometry location error keeps accumulating even though it has been reset a few times, at **corner 1** and **corner 4**. But the mapped line which can be seen on the

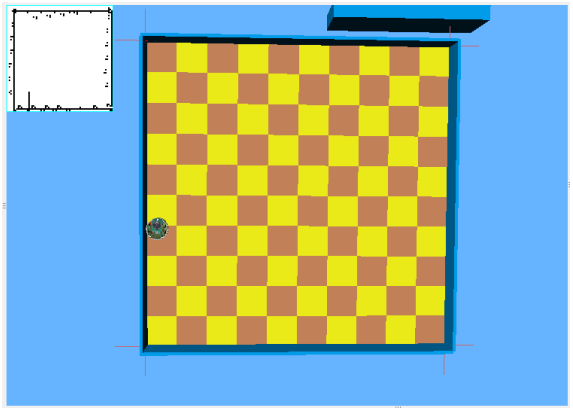

Figure 4.4: Room 1 map result with crowing odometry error

lower left corner of the generated map indicates that the localisation error became to big so that an area was map in the middle of the environment which clearly belongs to the left hand wall. This however is also an working example of the reset function which has been implemented for the corners of the environment. As the location information of the odometry struct have been reset as the corner was reached. The reset can be seen as the robot depicted in the figure already did another pass along the left hand wall up and down, after being reset in the lower left corner.

This localisation error however is a rather random event, as different results from other runs suggest that the odometry error is strongly based on the random nature of the noise in the simulator.

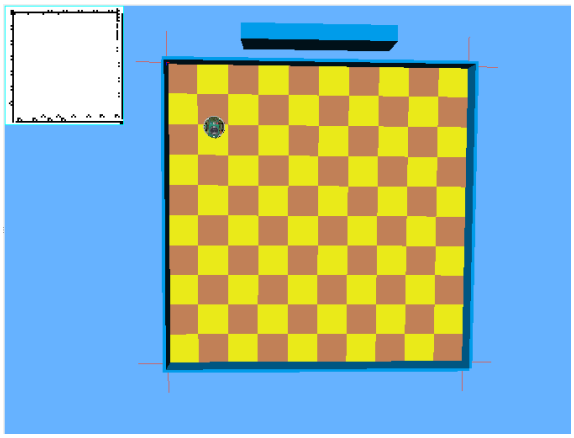

Figure 4.5: Room 1 map result with different odometry error

Figure 4.5 shows the result of another test run, here the robot only had a small miss mapping issue in the upper-right corner of the map, which happened right after the map was closed. However this error was only in a small area, and was completely reset when it reached the lower-right corner of the environment. The test run depicted in this figure has progressed further than the test run shown in figure 4.4, and it shows that the odometry localisation error did not happen again, in the same place.
In other test runs the odometry localisation error did not happen until the robot started traversing the environment the 3rd time, however 90% of times before it happened during the 2nd envi-

ronment traverse, this suggest that the problem lies within the random nature of the simulator, however that this random nature is within boundaries.

4.2.1.1 Conclusion for this test environment

While the mapping performs reasonably well within this test environment not all test runs are the same. The mappings differ minimal in places every simulation run, this happens because of the random nature of the simulator.
While the mapping of the whole environment can be performed with reasonably similar results, the results start to differ a lot more when the robot traverses the environment a 2nd or 3rd time. The figures 4.4 and 4.5 are the results of 2 different simulator runs which happened directly after another. Other times the 2nd simulator run returned no mapping error which was more noticeable than the common minor dis-localisation, but major errors as shown in these figures happened in later runs. The mapping error de-pictured in both documents that while the simulator random nature causes minimal map differenced every time, larger errors happen more seldom and more randomly.

4.2.2 Environment with obstacles

This subsection shows and describes the results of another test environment.
This environment is the same as described in the previous sub-section however this has an obstacles added to it.

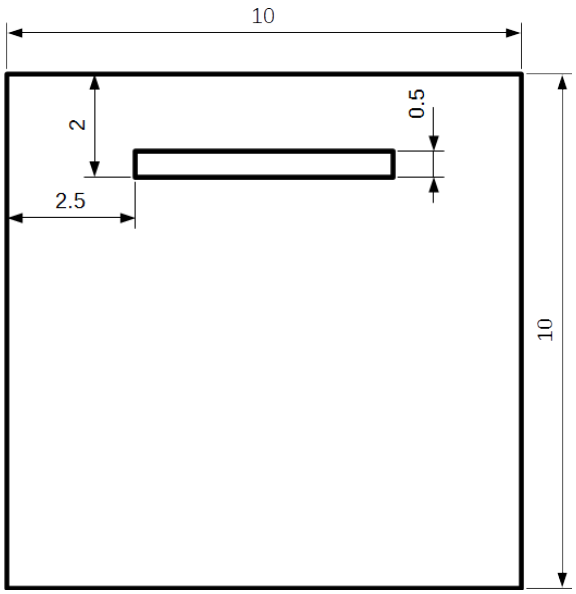

Figure 4.6: Measurements of room 1 with obstacle added to it

Figure 4.6 shows the measurements and location of the obstacles added to the environment, again all measurements are in *squares*.

Figure 4.7 shows the result for this environment.
As it can clearly be seen, this figure de-pictures the major short comings of the final program. On the lower part of the generated map, it can clearly be seen where the major short coming of this

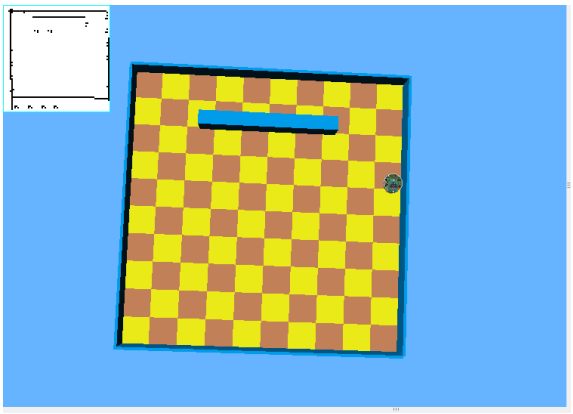

Figure 4.7: Room 1 with obstacle map result

solution, the odometry localisation error gets to big.
Unlike to the **room 1** example without any obstacle, which is described in the previous section, this result depicts what happens when the robot is not able to reach a saved reference point(already traversed and saved corners), but rather continues mapping without being reset.

What has happened in this case is that the obstacle prevented the robot to traverse the whole width of the map, which altered the movement enough for the robot to reach the unsaved lower-right corner of the map before it reached an already saved reference point which would led to the localisation information being reset.
This can cause a couple of problems. The biggest of them is, which can be clearly seen, that the localisation error gets to large for the map to be usable. The problem in this case is that the odometry information of the lower-right corner are saved, and every time the robot passes this corner the odometry information will be updated to the saved, wrong, information.

Besides this major error, it also shows the programs short comings in the mapping of obstacles which are placed in the middle of the room.

Figure 4.8: Obstacle mapping shortcomings

Figure 4.8 shows the problem when an obstacle in the middle of the room is being mapped with the current mapping approach.
The upper-side of the obstacle is mapped reasonable well, however the the other sides are lacking. There are some spots on the right side of the obstacle which have been marked by during the U-Turn routine, however the lower-side of the obstacle shows the real short comings. When the mapping of the lower and the upper side of the obstacle are compared it gets clear that the

main limitation with this approach is the usable mapping only happens when the robot traverse the obstacle which its side to it, so that it will be mapped. The spotty mapping points generated by the U-Turn routine are usable to outline the obstacle at best, however the specified movement pattern implemented in this project can prevent possible mapping of an obstacle should the obstacle be placed at a point which will not be passed by during the robot movements pattern.

Another problem is the odometry error in an environment like this. As has already be shown the localisation error gets to big and the altered movement pattern prevents the robot from resetting its localisation values, that is not without it getting reset to something wrong when it passes a long the reference point which has been saved with the wrong odometry data in the first place, in this case the lower-right corner.
This however does not only cause errors for the outline of the environment but also for the obstacles, as in some simulation runs visible localisation difference on the spotty outline of an obstacle can be seen.

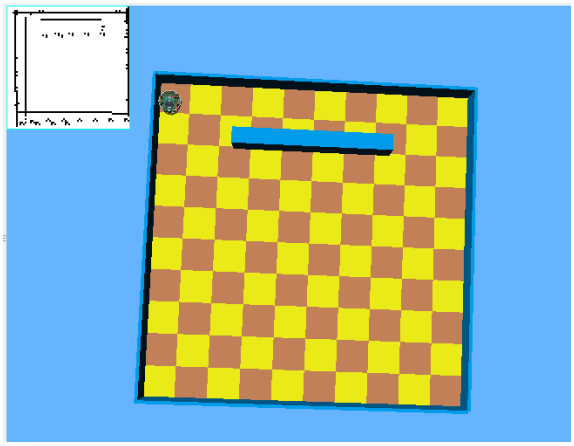

Figure 4.9: Accumulated odometry error

Figure 4.9 shows the map result at a later point during the environment traverse. The figure shows how much the localisation errors accumulate over time when the robot is not able to reset its localisation values.
It shows the already discussed localisation error for the lower side of the map, as well as the spotty outlining in the obstacle. However this map shows also that the localisation values have been reset in the corners which were passed before the localisation error became to big, as the spotty mappings on the lower side of the map show. While the entire lower wall of the environment is mapped at a wrong place, the localisation values have been reset in the upper right corner which causes the spotted, U-Turn based, mappings a long the lower right hand side.

In figure 4.10 these spots have been marked. The spots to the left hand side of the marked spots have been mapped after the robot had started, since it starts in the center of the map facing to the lower end of the environment.
As the marked spots are approximately at the Y axes as the other markings, can be seen that the

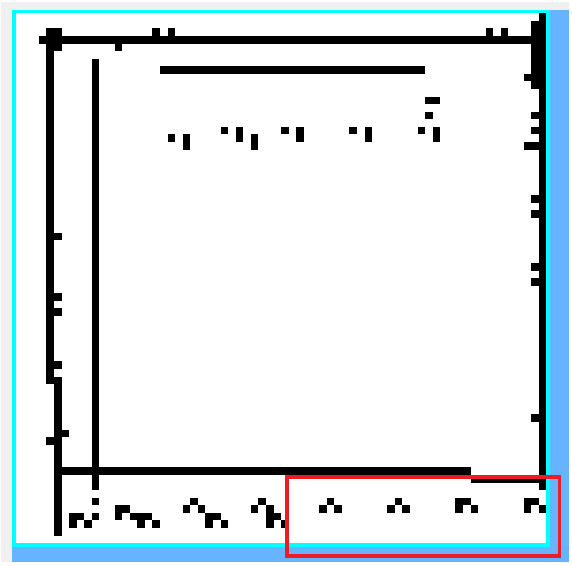

Figure 4.10: Marked spots

robot resets its localisation values to the correct values when it passes the upper-corner reference point.
However this figure also depicts what the accumulated odometry errors cause to an already scanned side. On the left hand side of the figure it can easily be seen where the robot remapped the left hand wall of the environment.

4.2.2.1 Conclusion for this test environment

This test environment shows the short comings of this program, the fast movement pattern can cause problems when mapping obstacles as the robot will only ever move in the same movement pattern. This however can, and in most cases will, lead to an large accumulation of localisation error.
It is apparent that the major problem of this program is that the odometry values need to be reset in given intervals in order to prevent an to large accumulation of errors. This is however, with the current solution, only possible when the robot manages to pass by an saved reference point before the localisation error either becomes to large, or another reference point is set with the wrong localisation information.

There are a couple of ways the problems which become apparent in this test environment could have been fixed. One possibility is to have a wall following algorithm which allows for the mapping of the outline of an obstacle before falling back into another movement pattern.
This would prevent problem with the mapping of obstacle which has been discussed in the previous section. Another solution would be to set reference points more often, or have a way of localizing the robot besides having set reference points, possibilities for this would be GPS or long range scanners which are able to keep track of global reference points, which have been set previous to the run.

There are a couple of possibilities which could have been implemented in order to make this mapping better, however the problems encountered during implementation have taken to much time to fix to be even able to map a single, empty, environment to be able to implement further algorithms after test runs had been done for environment with obstacles inside them.

4.3 Room 2

Room 2 is an narrower version of **room 1**, with a size of 6x10 squares.

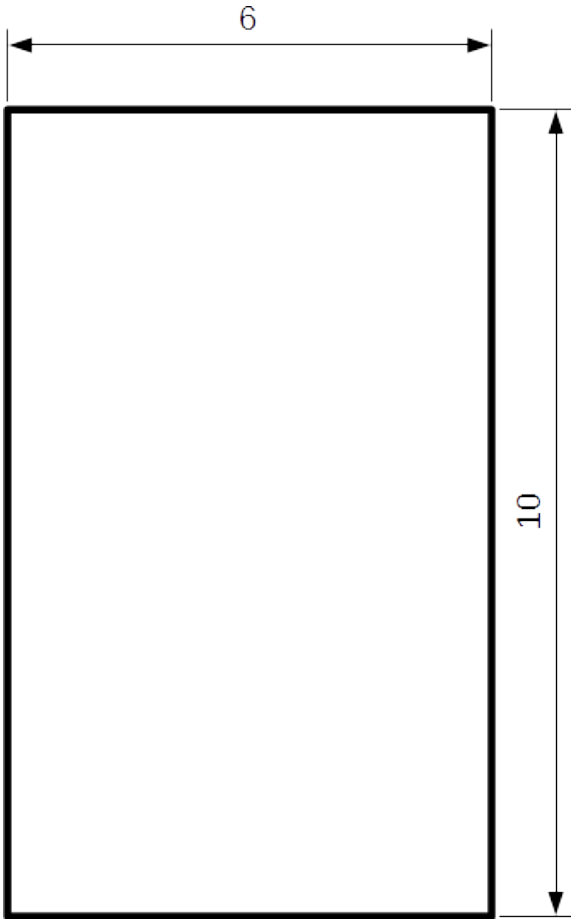

Figure 4.11: Room 2 measurements

4.3.1 Obstacle free test

In this section the results for the obstacle free version of **room 2** will be shown and discussed.

Figure 4.12 shows the result of the second test environment. While the map is not finished, the robot was stuck in an eternal loop at the lower wall of the environment.
This environment shows once more that the mapping works fine as long as the localisation error

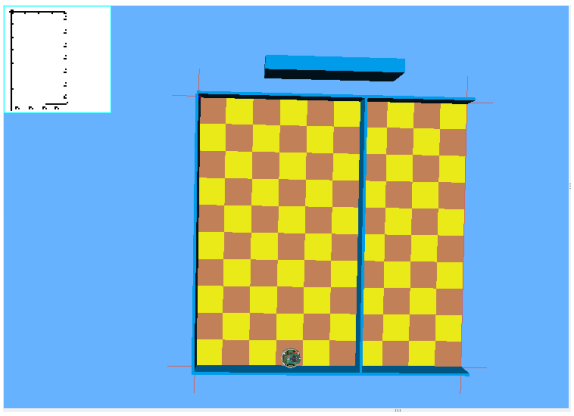

Figure 4.12: Room 2 results

has not accumulated it self to much.

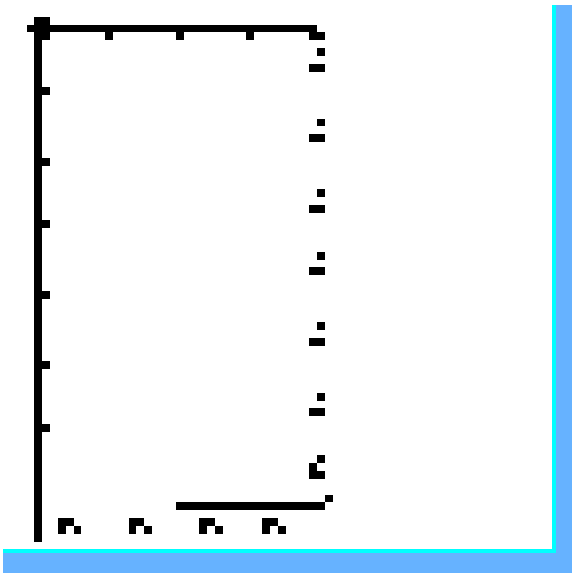

Figure 4.13: Room 2 shows that the localisation error depends on environment size

However figure 4.13 documents also that the increase of the localisation error strongly depends on the size of the environment and there by how far the robot traverses between each wall.
The localisation error strongly changes when the robot driver longer distances, even on straight lines. Even if the robot is perfectly calibrated which prevents curves movement caused by different wheel diameters **FIXME(insert link to where this is discussed in chapter 3)**, the robot will never turn an exact amount of degrees, this is caused by the friction between the wheels and the floor.
This however, as already discussed in chapter 3, leads to the robot believing it is somewhere while it actually is somewhere else. As figure 4.13 documents this error is reduced if the robot does not need to drive far until it has to turn again.
It does so by showing clearly that localisation error, while it still persists, is much smaller than in an empty environment where to robot has to drive further distances until turning. An example of

this can be seen in **FIXME(add link to localisation error as discussed in chapter 3).**

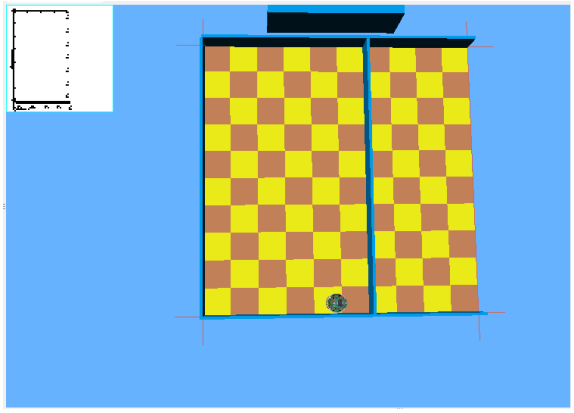

Figure 4.14: Room 2 results at a later point

However the localisation error is not the only problem. As can be seen in figure 4.13 also in this case the robot reaches an uncharted reference point before passing by an existing reference point to reset its localisation values. This leads to the same problem encountered in section 4.2.2. However the robot in this case is stuck within an eternal loop, where it resets to the right coordinates at the lower-left hand corner and drives to the right-hand side of the map where it gets reset to the wrong localisation values again.
The problem of this happening is a bug inside the code combined with odometry errors. The odometry error causes the robot to not directly drive close to the wall but rather stay a bit away from it. And the bug causes the robot to not change it overall direction whenever it reaches one of the corners but rather to stay in the U-Turn movement pattern.

4.3.1.1 Conclusion of this test environment

This test environment did not really showcase any other problems with code then the ones already covered in section 4.2. It did however show case an until now unknown bug which would need to be patched out, if more time were available.
This test environment however documented, the already existing assumption, that the size of the localisation error depends on how long distances the robot needs to move until turning again. This localisation error in the first place is however caused by inaccurate turning caused by friction in the environment.

4.3.2 Environment with obstacles

In this subsection the test results an environment which has the same size as **room 2**, however holds an extra obstacle.

Figure 4.16 shows the result of this test environment.
As it can clearly be seen the map of this environment is not finished, the reason for that are bugs

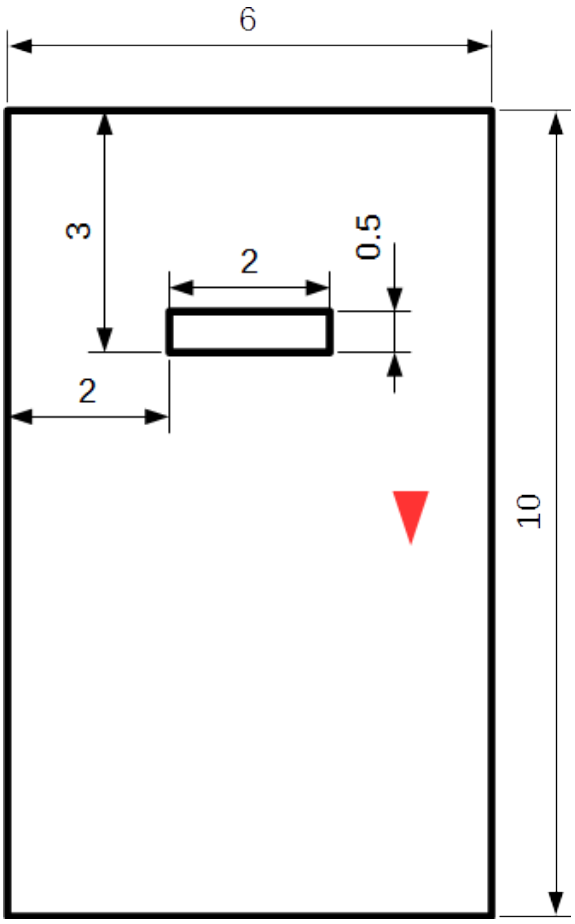

Figure 4.15: Room 2 with obstacles measurements

in the code and odometry errors. The robot starts at the marked starting position, red arrow, and proceeds to map the environment as usual. It can clearly be seen where the robot mapped the outline of the obstacle as it passed it. When the robot passed the upper-left reference point this point is saved as usual.
The robot then maps the left-hand side wall of the environment, and saves the lower-left hand corner as usual.

However here is where the bug in the program causes the robot also save the same corner as the lower-right hand corner and change the direction value of the odometry struct to west wards, while the robot is actually moving north warts.
Figure 4.17 shows the readout generated by the program. It shows how the robot,correctly, moved southwards then saved the corner as corner 1, which is also correct, but then saves the corner also as corner 2 and sets its own direction to west ward, while actually moving to the north.

This robot then proceeds to move upwards along the left-hand side wall, however the mapping the robot does is outside the map area as it believes to move west ward. The robot drives 1/4 of the the environment size upwards before the simulator crashes. This crash has happened every

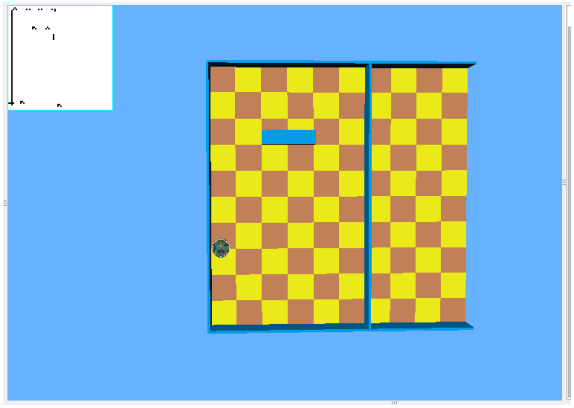

Figure 4.16: Room 2 with obstacles result

single time this environment was tested.

```
[Main] 269.999997
[Main] south
[Main] 269.999997
[Main] 269.999997
[Main] 269.999997
[Main] Saving at corner: 1 with cords: -0.483197 and -0.447394
[Main] south
[Main] stopping
[Main] 269.999997
[Main] 269.999997
[Main] 3
[Main] south
[Main] thinking
[Main] 269.999997
[Main] Saving at corner: 2 with cords: -0.483197 and -0.459759
[Main] stopping
[Main] 269.999997
[Main] Resetting at corner: 2 with cords: -0.483197 -0.459759
[Main] 179.936389
[Main] 179.936389
[Main] 179.936389
[Main] west
[Main] 179.936389
[Main] 179.936389
```

Figure 4.17: Buggy software causes causes the robot to save wrong values

4.3.2.1 Conclusion of this test environment

This test clearly showcases another major bug within the program. Unfortunately not more information can be gather from the test runs as the simulator crashes every time at the same place.

4.4 Room 3

This section holds the description and results of test of environment **room 3**.
This test environment differs from the previous test environments as it is a more ”complex” environment compared to the others. It is more complex as it is completely square, this is done to test the programs performance in a environment like this.
This section was also not tested with obstacles inside it. The reason being that the previously gathered data shows that obstacle inside the room disturb the mapping and prevent a complete

mapping. Figure 4.18 shows the environment and its measurements.

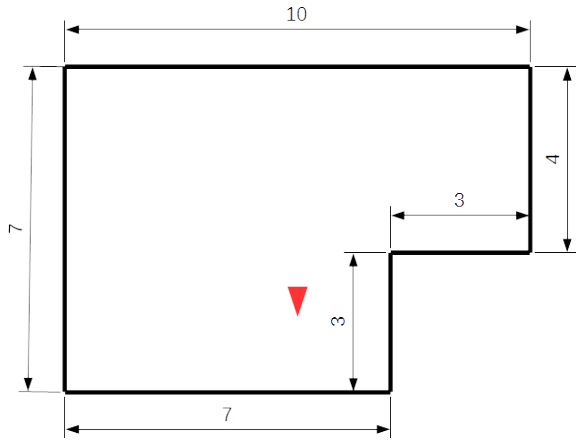

Figure 4.18: Measurements of room 3

Figure 4.19 shows the map result after the robot traversed a part of the map. It can be seen that the robot mapped the outline of the right-hand wall, however it can be clearly seen that the localisation error accumulated to much which prevents an accurate mapping of the lower wall.

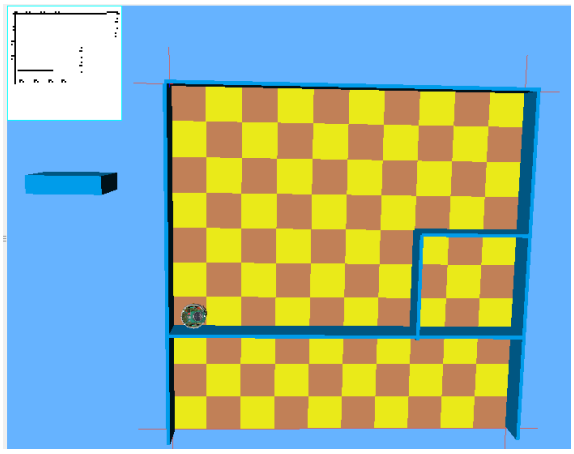

Figure 4.19: Result of the third room

It resets the localisation values at the lower left corner, however then proceeds to reset the values to the equivalent of the upper-right corner immediately. The robot then starts to move back and forth along the lower wall, then proceeded to move up the left wall.
It resets the localisation values again when the lower-left hand corner is reached for the second time, then moves upwards along the left wall, however the odometry struct now holds the wrong orientation values: the robot believes to move east(to the right). The robot then proceeded moving along the walls in the normal movement pattern, however with the localisation and orientation values all wrong the mapping was unusable, and the test was ended at this point.

4.4.1 Conclusion of this test environment

This test environment did not produce a lot of new insight into the final program. While it documented the shortcomings of the reference point algorithm this is not new data, as previous test environments showed the same data.
The partly created map with can be seen in figure 4.19 indicated that the mapping process of the wall formation at the lower right side of the environment would have been lacking at best. This is based on the moving pattern where the robot moves given distances in the turns, should an wall or obstacle not be ”luckily” lay to the side of this path chances are that the robot will not be able to map it, in best cast map the outline.

4.5 Conclusion

The test results inside this chapter give clear insight into the weak and strong sides of the program. Unfortunately there are more weak points. During the implementation phase of the project a couple of problems were encountered, problems with odometry, localisation and mapping. While these problems were countered and fixed the solution seamed good inside the implementation environment. This is the main reason why some of the programs shortcomings exist.

The algorithm which handles the reference points is based on prior knowledge which was gathered during test runs in the implementation phase. It was known prior to design of the algorithm in which direction the robot is moving and from what direction it will approach a corner of the environment. This approach works fine inside the **room 1** environment but **room 2** and **room 3** have shown that this approach failed if the robot should move in a slightly different path than to the one the algorithm was designed under.
If the robot reaches a corner but its direction is different from the one specified in the algorithm, it will set or reset its localisation values to different corners than it should. This problem however could be fixed by adding more control statements to the algorithm, rather than simple approach which is implemented at this moment.

Another weak point of the program is the movement pattern, if obstacles are placed inside the environment their presence can disturb the movement pattern enough to cause severe odometry and localisation errors. It has been observed in the testing phase that an E-Puck has collided with the wall, but rather than to stop it continued driving, however skewed itself along the side of the obstacle changing it path and increasing the localisation error. While this can happen because of the absence of bumper sensors of the E-Puck robot platform and the placement of the distance sensors around the E-Puck’s top side, it shows that this robot platform is not the best suited for map building.
Another example is the mapping of**room 3** which can be seen in figure 4.19 and the obstacle mapping of **room 1** from figure 4.8. These 2 figures show the problems with the current movement pattern on walls and obstacles. There is no direct way of fixing the movement pattern in this program. One approach could be to add a wall-following algorithm to whenever a new obstacle is

detected, or in case a completely different approach, some of them were discussed in Chapter 1 and 2.

The biggest problem however is the localisation approach.
The localisation error accumulated over time and needs to be reset at some points during the program run in order to prevent large mapping errors which can be seen in all test cases. One of the aims with this project was to be able to localise the robot without the need of GPS or Compass data or long range scanners which can keep track of global reference points at all times. However odometry calculations , which is what was used to locate the robot in this project, have limitations. In a non-perfect environment(an environment without friction or sensor noise) odometry calculations are always going to be an estimate at best, this was the biggest challenge of this project from the very beginning. Without functioning odometry algorithms it was not able to turn the robot an accurate amount of degrees or move it a given distance, even though these algorithms were successfully implemented they are not 100% accurate, because of friction and sensor noise.

Without the ability to calculate how much the robot has moved or turned it was impossible to locate the robot accurately. The localisation and movement algorithms in them self function perfectly, however since odometry calculations are really just estimated odometry and localisation errors accumulated over time. Functions were created to tackle these problems, algorithms which check the robots heading and ensure it is only ever deviating with a maximum threshold from their path, and to set reference points at points the robot is able to locate, the corners. However the algorithm which controls the heading is only ever accurate as long as the calculated orientation values are right, extreme odometry errors or collisions with walls/obstacles can lead to errors, as the robot posses no possibility to check the actual position and heading in the simulator[1].
Even considering a case where the heading control algorithms work completely fine(in most cases they do) the localisation error still increases over time. The algorithm designed to set reference points and reset the robots localisation values helped with this problem, however the short comings of this algorithm has already been discussed.

The overall conclusion drawn on the localisation problem is that is possible to calculate a robot positions, to an extend. However the error will always become to big, and reference points set by the robot it self are unreliable. While it is completely possible to use odometry calculations to tackle the localisation problem it is necessary to use predefined reference points or GPS and compass data. In order to use predefined reference points very strong and accurate sensors are required, which are not found for each robot platform, as well as the structure of the environment. GPS and compass data would make the localisation as accurate as possible. Of course GPS sensors have also a limited accuracy but the result will certainly be better than without GPS data. An possible approach could also be to combine GPS and odometry calculations to decrease the localisation error as much as possible.

[1]While it is possible to include API functions which return accurate localisation and rotation informations on the virtual robot such functions were never used as they would break the challenges and key points of this project.

To summarize the overall conclusion of this project it can be said that, while the program does not perform well or manage to generated accurate maps(unless it is **room 1**, it has certainly been a learning experience. It has shown the limitations of solely odometry based systems and have given some insight in how to minimize odometry errors, even though it is impossible to remove them completely.
The program in itself has strong problems, the movement pattern is far from optimal, but ideas and insight on this was gathered as well. The test have shown the limitations of the implemented algorithms as soon as it was tested on more complex environment, but also provided informations on these shortcomings and ideas on these could be fixed in the future.

Chapter 5

Future Plans

In this chapter ideas and inspirations for future improvements and implementations of this project. Most of these ideas were gathered during the research phase at the start of the project. While many of these ideas sounded really interesting and challenging, the time frame available for this project was not large enough in order to have a problem free implementation,as it would have been quite ambitious.

5.1 Swarm robotics

One of the largest inspirations gained during background reading was convert this project to a swarm robotics project.
The meaning of this would be the use of swarm of robots in order to map a larger, more complex environment. Using a swarm of robots rather than a single robot would require a couple of things, a more advanced and swarm suitable, deployment method, and a communications solution for map sharing between robots.

In the next few subsections possible solutions to either of these problems are disguessed, again these are based paper read during the background reading phase.

5.1.1 Swarm communication

While it is possible to transfer information easily between robots since a simulator is used one keypoint would be to try implement it as close to a realistic scenario as possible, meaning that the communication range for the robots is limited. In an realistic scenario every robot would have to send the acquired map back to the static start point/lead robot so that an overall map of the environment can be created.
Since the communication range for such small robots is limited and can be even further obstructed through obstacles like walls it is important to designated some robots as communication nodes. Such comm nodes would than remain stationary and link the ”scout” robots, which do the exploration, back to do the lead robot.
Obviously the most effective way to do this is by implementing different behaviour patterns for

scouts or comm robots, and implement a decision model which allows the robot to change between either pattern as the needs of the swarm change. E.g. in the start of the exploration no comm robots will be needed as the robots would most likely be inside the comm range of the lead robot, though this may change if the swarm is big and spread out enough.

To surpass the problems of obstacles obstructing the communication the comm robots would need to position them self on logical places i.e. in order to scan a room it would be important that a comm robot places it self inside, or close to, the doorway so that others can explore the room and still communicated back to the rest of the swarm.
Figure 5.1 demonstrates this behavior. The robot would need to stay inside the doorway as signals can not always travel through walls and the energy reserves of mobile robots are limited so they most likely can not send high power signals.
It has not been decided how this would be implemented since as of now it is not sure if the E-puck models implemented inside the simulator are able to send signals to other robots. While it is possible to transfer signals through the E-puck's laser sensors it is not know if this would actually be a better implementation than using radio transmitters and receivers.

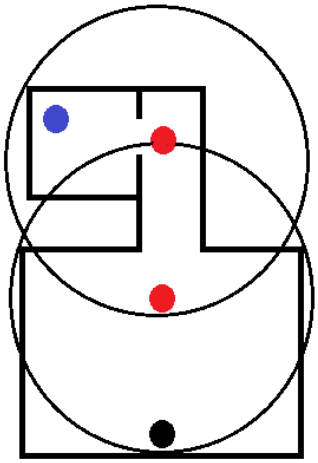

Figure 5.1: An example of the communication link needed for rooms

The theory of what could be implemented uses a defined maximum communication range for the robots and a grouping strategy which specifies that each scout robot need to stay in contact which at least 1 comm robot while the comm robots always need at least 1 other comm robot inside their communication range. If implemented correctly the comm robots would on this way create a communication link back to the lead robot/starting location which the scout robots can use to transfer all new information back.

Figure 5.2 shows one possible example of the communication link, where the lead robot or in some cases a stationary uplink point is in the center and the communication robots(in red) placed

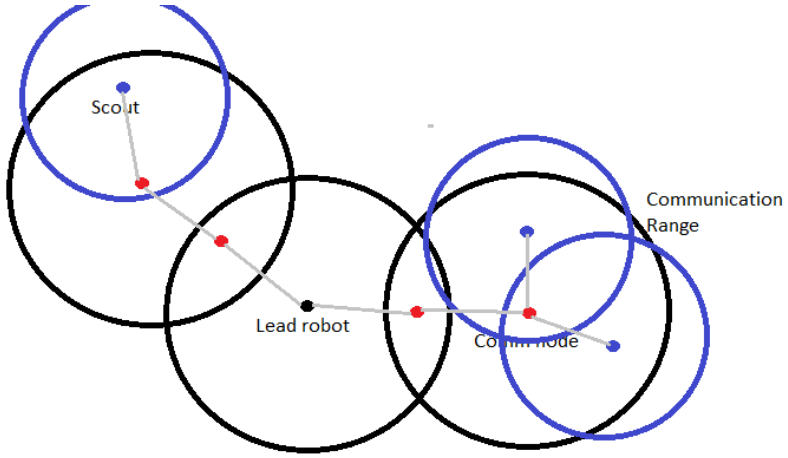

Figure 5.2: An example of the communication link

in such positions that their comm range overlaps the comm range of other robots and the lead robot.
This configuration allows the scouts(in blue) to move and explore anything inside the communication range of the different comm robots. When the robots at the right side of the figure would now try to move outside the comm range one of them would have to change their behaviour pattern to ”communication mode” at the outer range of the other comm nodes range while the last remaining scout continuous exploring in this direction.
This example shows that it is important to have a swarm of a suitable size for an environment to be able to cover at much area with the robots at hand and for cases in which this is not possible to be able to move the whole swarm in one unified direction to explore unmapped locations. This is however only doable when there is a lead robot since a stationary comm/uplink point is by definition, stationary.

5.1.2 Swarm deployment

The deployment strategy is an important part of a project like this as it defines how effective the swarm will cover the target area which will define how long it will take to scan and map the whole area. Another important aspect is how many robots can the swarm hold and effectively deploy using the current deployment strategy.
One research paper which was found proposed an solution of a communication network where the comm nodes keep track of the robots positions and guide them in directions which have not been explored in the last time period [2]. The paper uses a solution which is based on small comm nodes deployed by the robot, to make the solution fitting for this project it would be necessary to define some of my E-pucks as communication nodes which remain on a fast position and guide the ”scout” E-pucks based on area which have been least visited by the other robots.
This is certainly an possible solution however it could be considered a waste of robots in an small environment. Since an simulator is used there is no communication range problem, but as the

indentions are to make it as close to a possible real world application as possible this must still be considered. That is why an maximum communication range for the robots needs to be implemented however more about that can be found in section 5.1.1.

Another paper proposed an solution which is based on an Nearest-Neighbour algorithm, meaning every robot must have always a minimum of ”N” other robots inside its communication range [8]. In this solution to robots would emit signals to other robots which would manoeuvre the robots away from each other until only ”N” robots remain inside the robots communication range. This solution would cover a large area with the swarm fairly fast and also be adaptable for a swarm of any size, however a solution of moving the whole swarm in on direction would be needed in order to cover the whole target area, assuming the robot swarm is not big enough to cover it once completely deployed. This would therefore need both a lead robot which decides the movement of the swarm and communication robots which would always have at least 1 link to another comm robot in order to have a communication line back to the lead robot.
These are 2 deployment strategies ideas which were found during the background reading.

Another approach could be to use a similar approach as is taken during the project so far, using an rectangular movement pattern for each robot. This could be a reasonable approach when the approximate size of the environment is known.

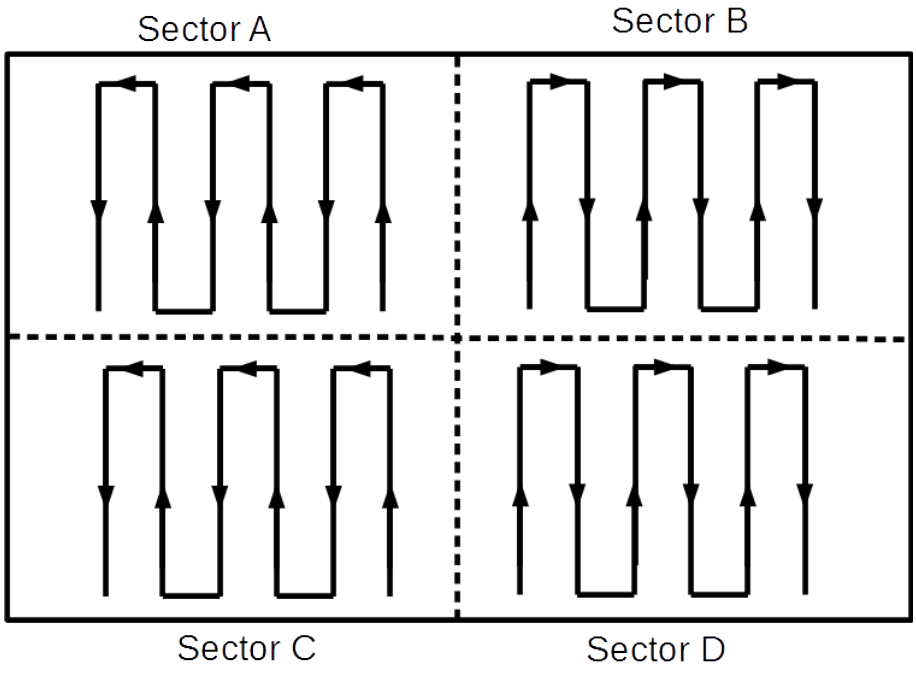

Figure 5.3: Sector based deployment pattern

Figure 5.3 shows how a deployment pattern similar to the one implemented inside the project now. The altered deployment pattern would be based on splitting up the environment into scan sectors. Were each robot, in this case 4, would get an sector allocated which it will then proceed

to scan. This however would require at least an reasonable estimate about the environment size, in order to specify into how many sectors the environment would be split up.

However this approach would require a swarm fitting to the environment size, where of the previous discussed deployment methods are able to traverse large environment even with smaller swarms, however for this circumstances an advanced movement pattern which is able to move the whole swarm in one direction would be needed.
However the sector moving pattern would work best, maybe even only, in square environment where as it most likely would fail in more advanced environments.

5.2 Autonomous Swarm behaviour

The previous sections describes different deployment strategies and addresses the communication problem, however one feature which could be implemented is intelligent robot behaviour in form of finding unexplored areas automatically.
This feature would depend heavily on that functioning deployment and communication algorithms are in place. The aspect of this algorithm would be to detect an unexplored area by checking the already created map. The idea for this is to ,as an example, scan an room which is connected to another room through a doorway. This would requite the map to be good enough and the doorway to be distinctive enough from the rest of the environment to be able to picked up.

Figure 5.4 shows a example of an advanced environment, for which such a autonomous behaviour would be needed in order to first, move the robots into the corridor and after that into the adjacent rooms. This is assuming the swarm starts in the large area at the bottom.

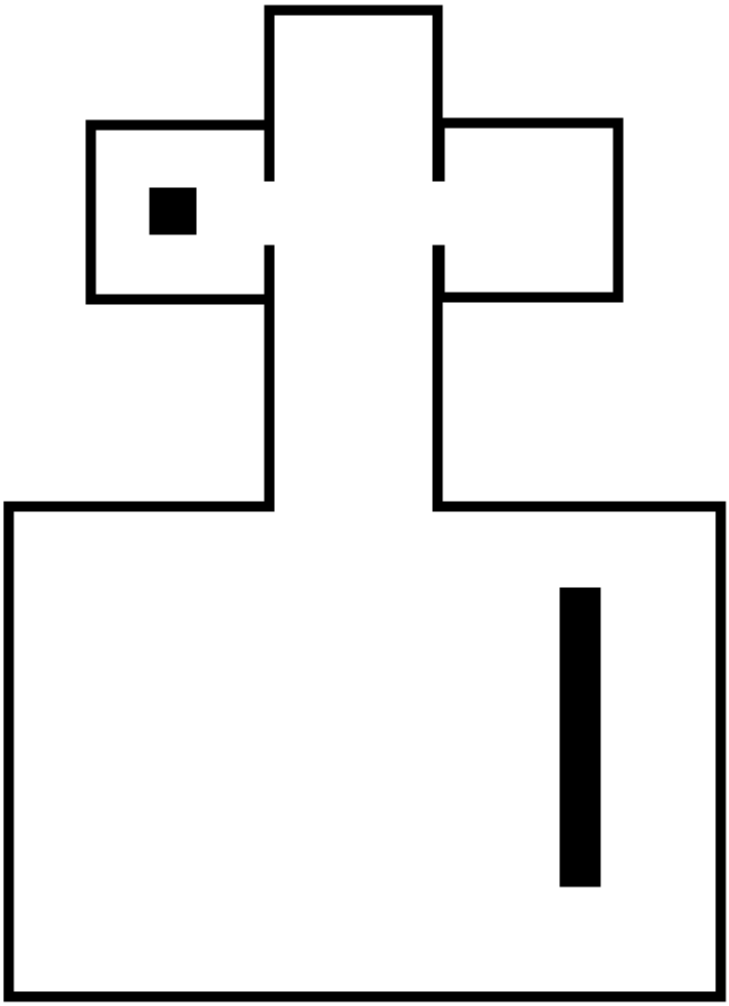

Figure 5.4: Example of an advanced environment

Chapter 6

Evaluation

Examiners expect to find in your dissertation a section addressing such questions as:

- Were the requirements correctly identified?
- Were the design decisions correct?
- Could a more suitable set of tools have been chosen?
- How well did the software meet the needs of those who were expecting to use it?
- How well were any other project aims achieved?
- If you were starting again, what would you do differently?

Such material is regarded as an important part of the dissertation; it should demonstrate that you are capable not only of carrying out a piece of work but also of thinking critically about how you did it and how you might have done it better. This is seen as an important part of an honours degree.

There will be good things and room for improvement with any project. As you write this section, identify and discuss the parts of the work that went well and also consider ways in which the work could be improved.

Review the discussion on the Evaluation section from the lectures. A recording is available on Blackboard.

Appendices

Appendix A

Third-Party Code and Libraries

1.1 Mark a cell as occupied

This function marks a cell of the occupancy map, an simple 2D array, as marked. It is part of a demo program provided with the webots installation. On a normal windows installation this file can be found here:
`C:\ProgramFiles\Webots\projects\samples\curriculum\controllers\advanced\_slam\_1.c`

Listing A.1: Mark an cell as occupied

```
#ifndef M_PI
      #define M_PI 3.14159265358979323846264338327795L
#endif

/**
 * Set the corresponding cell to 1 (occupied)
 * and display it
 */
void occupied_cell(int x, int y, float theta){

      // normalize angle
      while (theta > M_PI) {
            theta -= 2*M_PI;
      }
      while (theta < -M_PI) {
            theta += 2*M_PI;
      }

      // front cell
      if (-M_PI/6 <= theta && theta <= M_PI/6) {
            if (y+1 < MAP_SIZE) {
                  map[x][y+1] = 1;
                  wb_display_draw_rectangle(display,x,display_height-y,1,1);
            }
      }
      // right cell
      if (M_PI/3 <= theta && theta <= 2*M_PI/3) {
```

```
            if (x+1 < MAP_SIZE) {
                  map[x+1][y] = 1;
                  wb_display_draw_rectangle(display,x+1,display_height-1-y,1,1);
            }
      }
      // left cell
      if (-2*M_PI/3 <= theta && theta <= -M_PI/3) {
            if (x-1 > 0) {
                  map[x-1][y] = 1;
                  wb_display_draw_rectangle(display,x-1,display_height-1-y,1,1);
            }
      }
      // back cell
      if (5*M_PI/6 <= theta || theta <= -5*M_PI/6) {
            if (y-1 > 0) {
                  map[x][y-1] = 1;
                  wb_display_draw_rectangle(display,x,display_height-y-2,1,1);
            }
```

Appendix B

Code samples

2.1 Moving forward a given distance

This function moved the robot a given distance with a given speed.
The robot will slow down to a minimum speed on a minimum difference to the wanted position.
This method will update the global odometry information.
Description can be found in chapter 3 at section 3.3.1 at page 9.

Listing B.1: Moving forward

```
#define WHEELBASE 0.058
#define INCREMENT_STEP 1000 //how many steps the motor takes for a
    full wheel rotation
#define MIN_DIST 20.0f //minimum difference to the new heading
#define MIN_SPEED 10.0f //speed when slowing down

#ifndef M_PI
      #define M_PI 3.14159265358979323846264338327 95L
#endif

/**
Function to move the robot forward a given distance at a given speed
*/
void move_forward(double dSpeed, double dDist, struct
    odometryTrackStruct * ot){
      double dStepCount = 0.0f;
      double dStopPosLeft = 0.0f;
      double dStopPosRight = 0.0f;
      double *point_dEncPos;

      if((dDist > 0.0f) && (dSpeed > 0.0f)){
            //calculate the number of steps
            dStepCount = (INCREMENT_STEP/(M_PI * WHEEL_DIAMETER / 2))
                * dDist;

            /*read the current encoder positions of both wheels and
            calculate the encoder positions when the robot has to
            stop at the given distance ... */
```

```
            point_dEncPos = get_encoder_positions();

            dStopPosLeft = point_dEncPos[0] + dStepCount;
            dStopPosRight = point_dEncPos[1] + dStepCount;

            //compute odometry data
            point_dOdometryData = compute_odometry_data();
            odometry_track_step(ot);


            //set speed
            set_motor_speed(dSpeed, dSpeed);

            //step tolereance test
            while((point_dEncPos[0] < dStopPosLeft) &&
                (point_dEncPos[1] < dStopPosRight)){
                  //get odometry data
                  point_dOdometryData = compute_odometry_data();
                  odometry_track_step(ot);
                  //get wheel encoders
                  point_dEncPos = get_encoder_positions();

                  //slow down the closer the robot come to the
                     destination, reduces the error
                  if(fabs(dStopPosLeft - point_dEncPos[0]) <=
                     MIN_DIST){
                        set_motor_speed(MIN_SPEED, MIN_SPEED);
                  }
            }
      }
      stop_robot();

      //update odometry data
      point_dOdometryData = compute_odometry_data();
      odometry_track_step(ot);
      wb_robot_step(TIME_STEP);
}
```

2.2 Turning a given Angle

This function turns the robot a given amount of degrees with a given amount of speed.
The robot will slow down to a minimum speed on a minimum threshold to the wanted heading.
Description can be found in chapter 3 at section 3.3.2 at page 10.

Listing B.2: Turning an angle

```
#define LEFT_DIAMETER 0.0416
#define RIGHT_DIAMETER 0.0404
#define WHEEL_DIAMETER (LEFT_DIAMETER + RIGHT_DIAMETER)
#define WHEELBASE 0.058
#define INCREMENT_STEP 1000 //how many steps the motor takes for a
```

```
   full wheel rotation
#define MIN_DIST 20.0f //minimum difference to the new heading
#define MIN_SPEED 10.0f //speed when slowing down

/**
Function to turn the robot a given angle with a given speed
*/
void turn_angle(double dAngle, double dSpeed){
      double dFactor = 0.0f;
      double dStepCount = 0.0f;
      double dStopPosLeft = 0.0f;
      double dStopPosRight = 0.0f;
      double *point_dEncPos;

      if((dAngle != 0.0f) && (dSpeed > 0.0f)){
            //calculate turn factor
            dFactor = fabs(360.0f/dAngle);

            //calculate the number of step counts for the rotations
            dStepCount = (INCREMENT_STEP * WHEELBASE)/(dFactor *
                WHEEL_DIAMETER / 2);

            point_dEncPos = get_encoder_positions();

            //turn right
            if(dAngle > 0){
                  //calculate the target encoder positions
                  dStopPosLeft = point_dEncPos[0] + dStepCount;
                  dStopPosRight = point_dEncPos[1] - dStepCount;

                  set_motor_speed(dSpeed, -dSpeed);

                  while((point_dEncPos[0] < dStopPosLeft) &&
                      (point_dEncPos[1] > dStopPosRight)){

                        //get wheel encoders
                        point_dEncPos = get_encoder_positions();

                        //slow down the closer the robot come to the
                            destination, reduces the error
                        if(fabs(dStopPosLeft - point_dEncPos[0]) <=
                            MIN_DIST){
                               set_motor_speed(MIN_SPEED, -MIN_SPEED);
                        }
                  }

      } else { // turn left ...
            dStopPosLeft = point_dEncPos[0] - dStepCount;
            dStopPosRight = point_dEncPos[1] + dStepCount;

            // turn left the robot ...
            set_motor_speed(-dSpeed, dSpeed);

            while((point_dEncPos[0] > dStopPosLeft)
```

```
                &&(point_dEncPos[1] < dStopPosRight)){

                   point_dEncPos = get_encoder_positions();
                   if( fabs(dStopPosLeft - point_dEncPos[0]) <=
                       MIN_DIST ){
                      set_motor_speed(-MIN_SPEED, MIN_SPEED); }
                   }
             }
      }
      stop_robot();

      wb_robot_step(TIME_STEP);
}
```

2.3 Odometry Struct

This section shows the global odometry struct, which is used and updated throughout the program.

Listing B.3: Odometry struct

```
struct odometryTrackStruct {
      struct {
            float wheel_distance;
            float wheel_conversion;
      } configuration;
      struct {
            int pos_left_prev;
            int pos_right_prev;
      } state;
      struct {
            float x;
            float y;
            float theta;
      } result;
};
```

2.4 Initializing odometry struct

These functions are used to initialize the odometry struct and set its attributes. Further description of this can be found in chapter 3 at section 3.4.1 at page 12.

Listing B.4: Initializing odometry struct

```
#define LEFT_DIAMETER 0.0416
#define RIGHT_DIAMETER 0.0404
#define WHEEL_DIAMETER (LEFT_DIAMETER + RIGHT_DIAMETER)
#define WHEELBASE 0.058
#define INCREMENTS 1000.0 //how many steps the motor takes for a full
   wheel rotation
```

```
#define SCALING_FACTOR 1

#ifndef M_PI
      #define M_PI 3.14159265358979323846264338327 95L
#endif

/**
Initializes the odometry algortihms
*/
int odometry_track_start(struct odometryTrackStruct * ot){
      double* point_dEncPos;
      point_dEncPos = get_encoder_positions();
      return(odometry_track_start_pos(ot, point_dEncPos));
}

/**
Start the odometry tracking
Updates the info in the odometryTrackStruct for the first time
*/
int odometry_track_start_pos(struct odometryTrackStruct * ot, double*
    dEncPos){
      ot->result.x = 0;
      ot->result.y = 0;
      ot->result.theta = 0;

      ot->state.pos_left_prev = dEncPos[0];
      ot->state.pos_right_prev = dEncPos[1];

      ot->configuration.wheel_distance = axis_wheel_ratio *
          SCALING_FACTOR * (WHEEL_DIAMETER / 2);
      ot->configuration.wheel_conversion= (WHEEL_DIAMETER / 2) *
          SCALING_FACTOR * M_PI / INCREMENTS;

      return 1;
}
```

2.5 Odometry step

These functions are used to update the the odometry struct. Further explanations of these functions can be found in chapter 3 at section 3.4.2 at page 12.

Listing B.5: Updating odometry struct

```
#ifndef M_PI
      #define M_PI 3.14159265358979323846264338327 95L
#endif

/**
Updates an odometry data,
fetches the encoder positions and initialize the
odoemetry_track_step_pos function
```

```
*/
void odometry_track_step(struct odometryTrackStruct * ot){
      double* point_dEncPos;

      point_dEncPos = get_encoder_positions();
      odometry_track_step_pos(ot, point_dEncPos);
}

/**
Updates all odometry data for the struct.
Calculates the X and Y positions and orientation.
*/
void odometry_track_step_pos(struct odometryTrackStruct * ot, double*
    dEncPos){
      int delta_pos_left, delta_pos_right;
      float delta_left, delta_right, delta_theta, theta2;
      float delta_x, delta_y;

      //calculate the difference in position between the previous and
         current encoder positions.
      delta_pos_left = dEncPos[0] - ot->state.pos_left_prev;
      delta_pos_right = dEncPos[1] - ot->state.pos_right_prev;

      //calculate the rotation based on the displacement of the
         stepper motors
      delta_left = delta_pos_left * ot->configuration.wheel_conversion;
      delta_right = delta_pos_right *
         ot->configuration.wheel_conversion;
      delta_theta = (delta_right - delta_left) /
         ot->configuration.wheel_distance;

      // calculate the x and y displacement
      theta2 = ot->result.theta + delta_theta * 0.5;
      delta_x = (delta_left + delta_right) * 0.5 *cosf(theta2);
      delta_y = (delta_left + delta_right) * 0.5 * sinf(theta2);

      //update the x, y and theta of the struct
      ot->result.x += delta_x;
      ot->result.y += delta_y;
      ot->result.theta += delta_theta;

      if(ot->result.theta >=361){
            odometry_track_step(ot);
      }

      if(ot->result.theta > M_PI){
            ot->result.theta -= 2*M_PI;
      }
      if(ot->result.theta < -M_PI){
            ot->result.theta += 2*M_PI;
      }

      //save current encoder positions to the global buffer
      ot->state.pos_left_prev = dEncPos[0];
```

```
        ot->state.pos_right_prev = dEncPos[1];
}
```

2.6 Reference point struct

This section holds the code for the reference struct.
A further description of this struct can be found in chapter 3 section 3.11.1 on page 34

Listing B.6: Reference point struct

```
struct referencePos {
        struct {
                float x;
                float y;
        } lower_left;
        struct {
                float x;
                float y;
        } lower_right;
        struct {
                float x;
                float y;
        } upper_left;
        struct {
                float x;
                float y;
        } upper_right;
};
```

2.7 Reference point check

This is the function which checks if a reference point has been reached and if so either saves it or resets the robot variables.
A more detailed explanation of this code can be found in chapter 3 section 3.11.2 on page 34.

Listing B.7: Function which checks if a reference point has been reached

```
/**
This function checks the current robot position with the previous
    recorded reference position.
If a relation is found, the current encoder position are updated with
    the saved encoder postions.
The parameters are a pointer to the struct odometrtTrackStruct and
    refrencePos
and an int which defines which corner it is.
The corner is defined in the run() function where this function will
    be called.
1 = lower_left
2 = lower_right
```

```
3 = upper_left
4 = upper_right
*/
void check_reference_points(struct odometryTrackStruct * ot, struct
    referencePos * ref){
      double dThreshold = 20.0f;
      int i;
      double *point_dEncPos = get_encoder_positions();
      double dCurPosX = ot->result.x;
      int *point_SensorData;
      int corner = 0;
      int obstacle[NUM_DIST_SENS]={0,0,0,0,0,0,0,0};
      int ps_offset[NUM_DIST_SENS] = {35,35,35,35,35,35,35,35};
      int direction = check_direction(ot->result.theta);
      bool ob_front, ob_left, ob_right;
      char message[] = "Resetting";

/*    get distance sensor data*/
      point_SensorData = get_sensor_data(NUM_DIST_SENS);
      for(i = 0;i < NUM_DIST_SENS;i++){
            obstacle[i] = point_SensorData[i] - ps_offset[i] >
               THRESHOLD_DIST;
      }

      ob_front =
      obstacle[0] ||
      obstacle[7];

      ob_right =
      obstacle[2];

      ob_left =
      obstacle[5];

      /*find out what corner it is */
      //north
      if(direction == 1){
            if(ob_front && ob_left){
                  corner = 3;
            }else if(ob_front && ob_right){
                  corner = 4;
            }
      }

      //east
      if(direction == 2){
            if(ob_front && ob_left){
                  corner = 4;
            }else if(ob_front && ob_right){
                  corner = 2;
            }
      }

      //south
```

```
if(direction == 3){
      if(ob_front && ob_left){
            corner = 2;
      }else if(ob_front && ob_right){
            corner = 1;
      }
}

//west
if(direction == 4){
      if(ob_front && ob_left){
            corner = 1;
      }else if(ob_front && ob_right){
            corner = 3;
      }
}

/*Check if the current corner is set to 0, if so force update*/
if(corner == 1){
      if((ref->lower_left.x == 0.00) && (ref->lower_left.y ==
         0.00)){
            set_reference_point(ot, ref, corner);
            return;
      }
}else if(corner == 2){
      if((ref->lower_right.x == 0.00) && (ref->lower_right.y ==
         0.00)){
            set_reference_point(ot, ref, corner);
            return;
      }
}else if(corner == 3){
      if((ref->upper_left.x == 0.00) && (ref->upper_left.y ==
         0.00)){
            set_reference_point(ot, ref, corner);
            return;
      }
}else if(corner == 4){
      if((ref->upper_right.x == 0.00) && (ref->upper_right.y ==
         0.00)){
            set_reference_point(ot, ref, corner);
            return;
      }
}

/*If the corner is set updated the odometery information of the
   robot */
if(direction == 1){ //north
      if(((ref->upper_left.x <= dCurPosX + dThreshold) ||
         (ref->upper_left.x >= dCurPosX - dThreshold)) &&
         corner == 3){
            ot->result.x = ref->upper_left.x;
            ot->result.y = ref->upper_left.y;
            ot->state.pos_left_prev = point_dEncPos[0];
            ot->state.pos_right_prev = point_dEncPos[1];
```

```
                printf("%s at corner: %d with cords: %f %f\n",
                    message, corner, ref->upper_left.x,
                    ref->upper_left.y);
            }else if(((ref->upper_right.x <= dCurPosX + dThreshold )||
                (ref->upper_right.x >= dCurPosX - dThreshold)) &&
                corner == 4){
                ot->result.x = ref->upper_right.x;
                ot->result.y = ref->upper_right.y;
                ot->state.pos_left_prev = point_dEncPos[0];
                ot->state.pos_right_prev = point_dEncPos[1];
                printf("%s at corner: %d with cords: %f %f\n",
                    message, corner, ref->upper_right.x,
                    ref->upper_right.y);
            }
        }else if(direction == 2){//east
            if(((ref->upper_right.x <= dCurPosX + dThreshold) ||
                (ref->upper_right.x >= dCurPosX - dThreshold)) &&
                corner == 4){
                ot->result.x = ref->upper_right.x;
                ot->result.y = ref->upper_right.y;
                ot->state.pos_left_prev = point_dEncPos[0];
                ot->state.pos_right_prev = point_dEncPos[1];
                printf("%s at corner: %d with cords: %f %f\n",
                    message, corner, ref->upper_right.x,
                    ref->upper_right.y);
            }else if(((ref->lower_right.x <= dCurPosX + dThreshold) ||
                (ref->lower_right.x >= dCurPosX - dThreshold)) &&
                corner == 2){
                ot->result.x = ref->lower_right.x;
                ot->result.y = ref->lower_right.y;
                ot->state.pos_left_prev = point_dEncPos[0];
                ot->state.pos_right_prev = point_dEncPos[1];
                printf("%s at corner: %d with cords: %f %f\n",
                    message, corner, ref->lower_right.x,
                    ref->lower_right.y);
            }
        }else if(direction == 3){ //south
            if(((ref->lower_left.x <= dCurPosX + dThreshold) ||
                (ref->lower_left.x >= dCurPosX - dThreshold)) &&
                corner == 1){
                ot->result.x = ref->lower_left.x;
                ot->result.y = ref->lower_left.y;
                ot->state.pos_left_prev = point_dEncPos[0];
                ot->state.pos_right_prev = point_dEncPos[1];
                printf("%s at corner: %d with cords: %f %f\n",
                    message, corner, ref->lower_left.x,
                    ref->lower_left.y);
            }else if(((ref->lower_right.x <= dCurPosX + dThreshold) ||
                (ref->lower_right.x >= dCurPosX - dThreshold)) &&
                corner == 2){
                ot->result.x = ref->lower_right.x;
                ot->result.y = ref->lower_right.y;
                ot->state.pos_left_prev = point_dEncPos[0];
                ot->state.pos_right_prev = point_dEncPos[1];
```

```
                printf("%s at corner: %d with cords: %f %f\n",
                    message, corner, ref->lower_right.x,
                    ref->lower_right.y);
            }
        }else if(direction == 4){ //west
            if(((ref->upper_left.x <= dCurPosX + dThreshold) ||
                (ref->upper_left.x >= dCurPosX - dThreshold)) &&
                corner == 3){
                    ot->result.x = ref->upper_left.x;
                    ot->result.y = ref->upper_left.y;
                    ot->state.pos_left_prev = point_dEncPos[0];
                    ot->state.pos_right_prev = point_dEncPos[1];
                    printf("%s at corner: %d with cords: %f %f\n",
                        message, corner, ref->upper_left.x,
                        ref->upper_left.y);
            }else if(((ref->lower_right.x <= dCurPosX + dThreshold) ||
                (ref->lower_right.x >= dCurPosX - dThreshold)) &&
                corner == 1){
                    ot->result.x = ref->lower_left.x;
                    ot->result.y = ref->lower_left.y;
                    ot->state.pos_left_prev = point_dEncPos[0];
                    ot->state.pos_right_prev = point_dEncPos[1];
                    printf("%s at corner: %d with cords: %f %f\n",
                        message, corner, ref->lower_left.x,
                        ref->lower_left.y);
                    printf("Ot is: %f and %f", ot->result.x,
                        ot->result.y);
            }
        }
}
```

2.8 Saving a reference point

This section holds the code to save a new reference point to the reference struct. An explanation of this code can be found in chapter 3 section 3.11.3 on page 34.

Listing B.8: Function to save a new reference point to the reference struct

```
/**
Sets the reference point in the referencePos struct.
The parameters are a pointer to the referencePos struct and an int.
The int specifies which corner will be set.
1 = lower_left
2 = lower_right
3 = upper_left
4 = upper_right
*/
void set_reference_point(struct odometryTrackStruct * ot, struct
    referencePos * ref, int corner){
      char text[] = "Saving";
      printf("%s at corner: %d with cords: %f and %f\n", text, corner,
```

```
            ot->result.x, ot->result.y);
      if(corner == 1){
            ref->lower_left.x += ot->result.x;
            ref->lower_left.y += ot->result.y;
      }else if(corner == 2){
            ref->lower_right.x = ot->result.x;
            ref->lower_right.y = ot->result.y;
      }else if(corner == 3){
            ref->upper_left.x = ot->result.x;
            ref->upper_left.y = ot->result.y;
      }else if(corner == 4){
            ref->upper_right.x = ot->result.x;
            ref->upper_right.y = ot->result.y;
      }
}
```

Annotated Bibliography

[1] M. Abolhasan, T. Wysocki, and E. Dutkiewicz, "A review of routing protocols for mobile ad hoc networks," *Ad Hoc Networks*, vol. 2, no. 1, pp. 1–22, Jan. 2004. [Online]. Available: http://dx.doi.org/10.1016/s1570-8705(03)00043-x

[2] M. Batalin and G. Sukhatme, "Coverage, Exploration, and Deployment by a Mobile Robot and Communication Network," in *Information Processing in Sensor Networks*, ser. Lecture Notes in Computer Science, F. Zhao and L. Guibas, Eds. Springer Berlin Heidelberg, 2003, vol. 2634, pp. 376–391. [Online]. Available: http://dx.doi.org/10.1007/3-540-36978-3_25

[3] J. Borenstein and L. Feng, "Measurement and correction of systematic odometry errors in mobile robots," *Robotics and Automation, IEEE Transactions on*, vol. 12, no. 6, pp. 869–880, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1109/70.544770

[4] W. Burgard, M. Moors, D. Fox, R. Simmons, and S. Thrun, "Collaborative multi-robot exploration," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, vol. 1. IEEE, 2000, pp. 476–481 vol.1. [Online]. Available: http://dx.doi.org/10.1109/robot.2000.844100

[5] M. W. M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba, "A solution to the simultaneous localization and map building (SLAM) problem," *Robotics and Automation, IEEE Transactions on*, vol. 17, no. 3, pp. 229–241, June 2001. [Online]. Available: http://dx.doi.org/10.1109/70.938381

[6] D. Fox, W. Burgard, and S. Thrun, "Markov Localization for Mobile Robots in Dynamic Environments," in *Journal of Artificial Intelligence Research*, vol. 11, 1999, pp. 391–427. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.372

[7] Y. Mei, Y.-H. Lu, Y. C. Hu, and C. S. G. Lee, "Deployment of mobile robots with energy and timing constraints," *Robotics, IEEE Transactions on*, vol. 22, no. 3, pp. 507–522, June 2006. [Online]. Available: http://dx.doi.org/10.1109/tro.2006.875494

[8] S. Poduri and G. Sukhatme, "Constrained coverage for mobile sensor networks," in *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, vol. 1. IEEE, Apr. 2004, pp. 165–171 Vol.1. [Online]. Available: http://dx.doi.org/10.1109/robot.2004.1307146

[9] C. W. Reynolds, "Flocks, Herds and Schools: A Distributed Behavioral Model," in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '87, vol. 21, no. 4. New York, NY, USA: ACM, July 1987, pp. 25–34. [Online]. Available: http://dx.doi.org/10.1145/37401.37406

[10] K. Singh and K. Fujimura, "Map making by cooperating mobile robots," in *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*. IEEE, May 1993, pp. 254–259 vol.2. [Online]. Available: http://dx.doi.org/10.1109/robot.1993.292155

[11] S. Thrun, W. Burgard, and D. Fox, "A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, vol. 1. IEEE, 2000, pp. 321–328 vol.1. [Online]. Available: http://dx.doi.org/10.1109/robot.2000.844077

[12] S. Thrun, "Learning Occupancy Grids with Forward Models," in *In Proceedings of the Conference on Intelligent Robots and Systems (IROSâ2001*, 2001, pp. 1676–1681. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.6014

[13] S. Thrun, W. Burgard, D. Fox, H. Hexmoor, and M. Mataric, "A Probabilistic Approach to Concurrent Mapping and Localization for Mobile Robots," in *Machine Learning*, 1998, pp. 29–53. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.1128

[14] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust Monte Carlo Localization for Mobile Robots," 2001. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.8488

[15] H.-X. Yang and M. Tang, "Adaptive routing strategy on networks of mobile nodes," *Physica A: Statistical Mechanics and its Applications*, vol. 402, pp. 1–7, May 2014. [Online]. Available: http://dx.doi.org/10.1016/j.physa.2014.01.063