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 [4].
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.

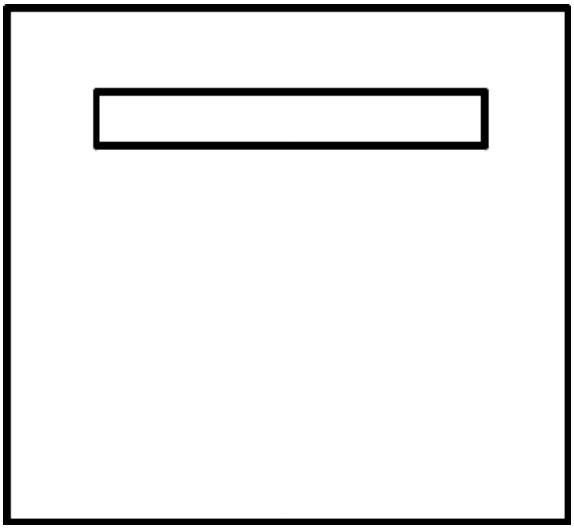

Figure 2.1: Environment Design

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 26.

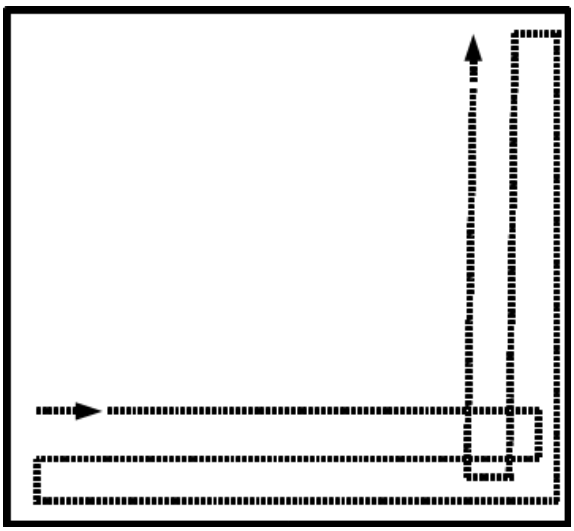

Figure 2.2: Controlled movement Pattern

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.

3.3.1 Moving Forward

The code for this function can be found in Appendix B, section 2.1 at page 34.
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.
It will then update the global odometry buffer by calling the *compute_odometry_data* function and 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 and the global odometry buffer is updated. 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 37.
The code for the odometry struct can be found in Appendix B, section 2.3 at page 37.
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 38.
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 .

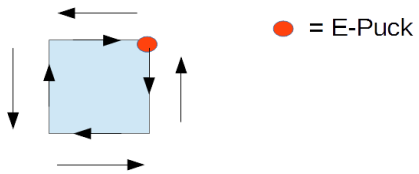

Figure 3.1: Movement algorithm test pattern

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 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].

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

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

```
//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.

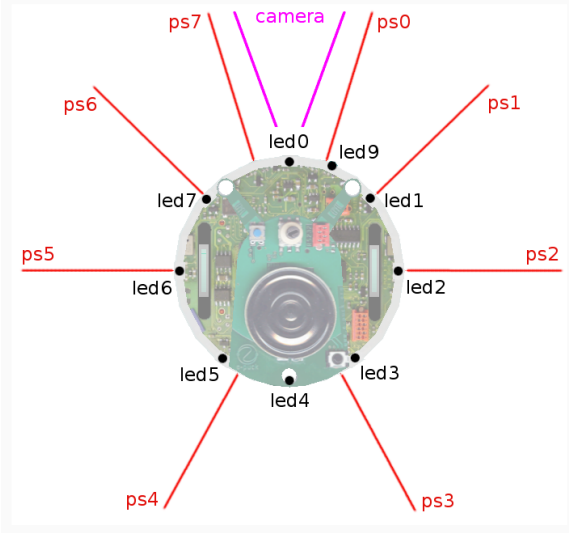

Figure 3.2: E-Puck sensor placement[3]

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

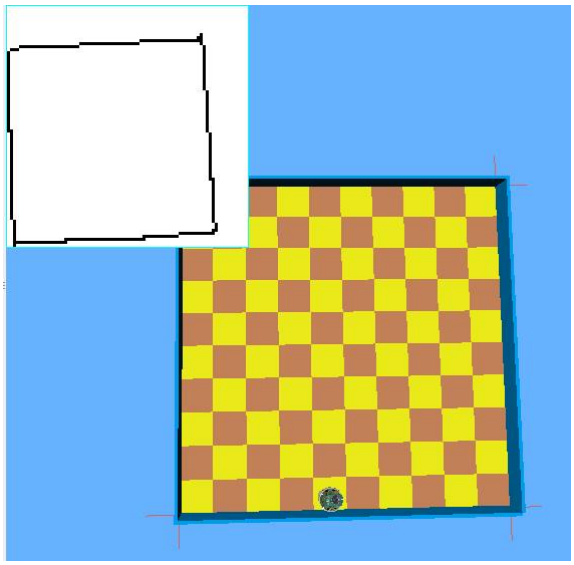

Figure 3.3: Early mapping result

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.

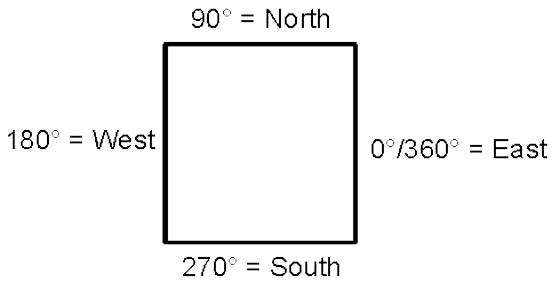

Figure 3.4: Directions and their corresponding Angles

Figure 3.4 does show the directions used and the corresponding angles. The directions were 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 3.14159265358979323846264338327951
#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.

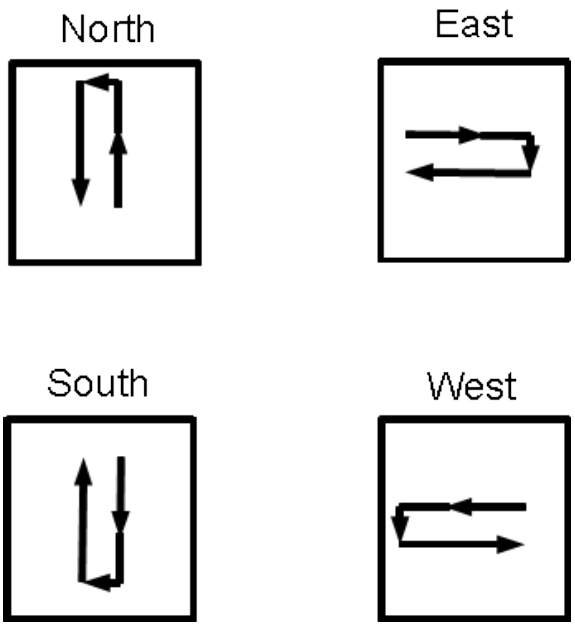

Figure 3.5: Directions and their corresponding U-Turn pattern

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 16.

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" FIXME(Find and add screenshot of this). However it provides a good outline of the environment and obstacles.

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 32. 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.

Chapter 4

Testing

Detailed descriptions of every test case are definitely not what is required here. What is important is to show that you adopted a sensible strategy that was, in principle, capable of testing the system adequately even if you did not have the time to test the system fully.

Have you tested your system on real users? For example, if your system is supposed to solve a problem for a business, then it would be appropriate to present your approach to involve the users in the testing process and to record the results that you obtained. Depending on the level of detail, it is likely that you would put any detailed results in an appendix.

The following sections indicate some areas you might include. Other sections may be more appropriate to your project.

4.1 Overall Approach to Testing

4.2 Automated Testing

4.2.1 Unit Tests

4.2.2 User Interface Testing

4.2.3 Stress Testing

4.2.4 Other types of testing

4.3 Integration Testing

4.4 User Testing

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. 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.

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.

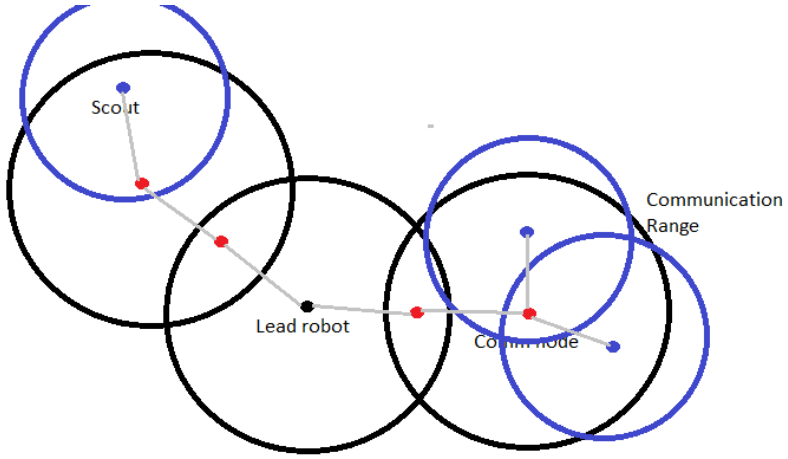

Figure 5.1: An example of the communication link

Figure 5.1 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 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 [7]. 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.

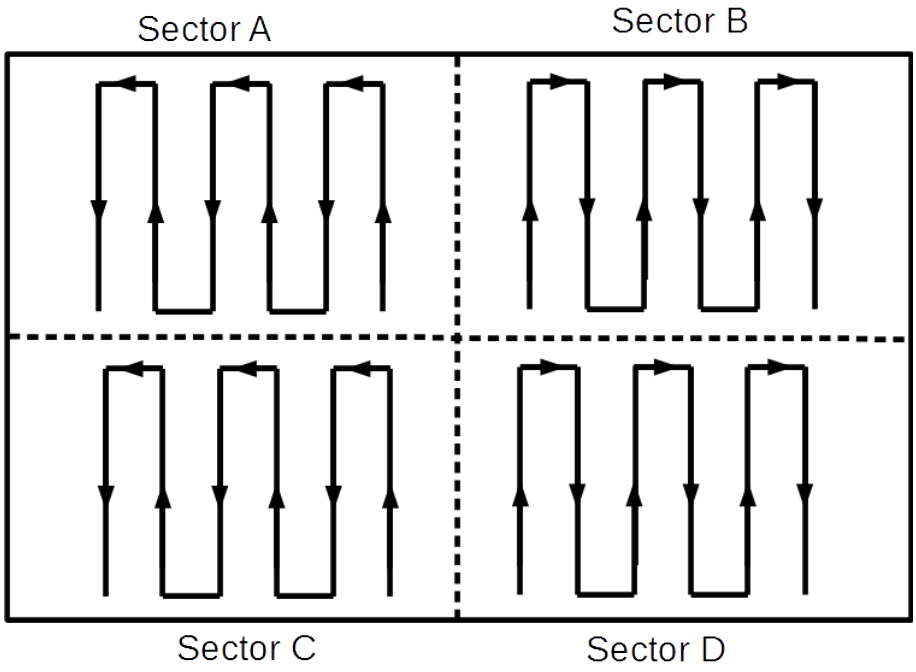

Figure 5.2: Sector based deployment pattern

Figure 5.2 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.

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 10.

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 3.14159265358979323846264338327795L
#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 11.

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;

                  point_dOdometryData = compute_odometry_data();

                  set_motor_speed(dSpeed, -dSpeed);

                  while((point_dEncPos[0] < dStopPosLeft) &&
                      (point_dEncPos[1] > dStopPosRight)){
                        //get odometry data
                        point_dOdometryData = compute_odometry_data();

                        //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;
```

```
            point_dOdometryData = compute_odometry_data();

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

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

                  point_dOdometryData = compute_odometry_data();
                  point_dEncPos = get_encoder_positions();
                  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();
      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 13.

Listing B.5: Updating odometry struct

```
#ifndef M_PI
```

```
      #define M_PI 3.14159265358979323846264338327950L
#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];
}
```

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] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] 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

[14] 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