|  |
| --- |
|  |
| **Autonomous Maze Mapping and Running Rover** |
| Operation Systems and Embedded Linux, Parallel Programming |
|  |
| **Team 3 / IOSLX4, IPARP4** |
| **5/1/13** |

**Supervisors:**

Ole Rydahl, Henning Haugaard

**Student name Student number Signature**

Johannes Jørgensen 120428

Benjamin Prakash 100100

Ivo Drlje 101108

Daniel Machon 092039

Table of contents

1. Introduction 5

1.1 Project start 5

1.2 Problem formulation 5

1.3 Milestone plan 5

2. Problem analysis 6

2.1 Requirements 6

2.2 Analysis 6

2.2.1 Navigation 6

2.2.2 Mapping 7

2.2.3 Path finding 8

2.2.4 Choice of programming language 8

3. Proposed solution strategy 9

3.1 Git 10

3.1.1 Branches 11

3.1.2 Issues 12

3.2 Resources 12

3.2.1 Hardware 12

3.2.2 Software 13

4. Problem solution 14

4.1 Navigation 16

4.1.1 Design process 16

4.1.1.1 PID controller algorithm 17

4.1.1.2 Development process - considerations and issues 18

4.1.1.3 One-cell PID only 19

4.1.1.4 Iterative approach 20

4.1.1.5 Back to PID 22

4.1.1.6 Final PID adjustments 22

4.1.1.7 Conclusion 23

4.1.2 Implementation 23

4.1.3 Test 25

4.2 Mapping 26

4.2.1 Design 26

4.2.1.1 Position 26

4.2.1.2 Coordinate system 27

4.2.1.3 Mapping algorithm 27

4.2.1.4 Robot / PC - strategy 28

4.2.2 Implementation 29

4.2.2.1 Robot / PC communication 31

4.2.3 Test 32

4.3 Path finding 33

4.3.1 Design 33

4.3.1.1 Defining the graph 33

4.3.1.2 Robot / pc strategy 35

4.3.2 Implementation 36

4.3.3 Test 37

5. Conclusion 38

5.1 Process assessment 38

6. Bibliography 38

7. Glossary 39

8. Appendix 41

Appendix 1. Milestone plan 41

Appendix 2. Theory 41

**1.1.1.** **Motor** 41

8.1.1.1 Stepper Motor 41

8.1.1.2 DC Motor 41

8.1.2 Auto correction input 42

8.1.2.1 Odometry 42

 Mouse 42

 Wheel Encoders 42

8.1.2.2 Sensors 43

8.1.3 Auto correction 44

8.1.3.1 PID 44

8.1.3.2 Iterative 45

8.1.4 Path finding 46

8.1.4.1 Dijkstra 47

8.1.4.2 A-star 47

8.1.4.3 Flood fill 48

8.1.5 Communication 48

8.1.5.1 Zeroconf 48

8.1.6 Mapping 49

8.1.6.1 Depth first search 49

8.1.6.2 Graph 49

Appendix 3. Test cases 50

Appendix 4. Hardware diagram 55

Appendix 5. Process diagram 56

Appendix 6. Mapping logic flowchart 57

Appendix 7. Mapping flowchart I dunno 58

Appendix 8. Workflow diagram 59

Appendix 9. 59

Appendix 10. 59

Appendix 11. Sequence diagram 60

Appendix 12. 61

Appendix 13. 61

Appendix 14. 61

Appendix 15. 61

Appendix 16. 61

Appendix 17. 61

Appendix 18. 61

Appendix 19. Motor controller documentation and tests (MISSING!) 61

Appendix 20. Navigationsystem tests 62

Appendix 21. PID turn90 error 65

Appendix 22. PID turn90 error revision 2 67

Appendix 23. Mice odometry documentation 71

Appendix 24. IR-sensors documentation and tests 74

Appendix 25. RPi-TMC222 baud rate Documentation 81

**Preface**

The report consists of two parts, a main part and appendix part. The context of the main part will throughout the report use refers to the appendixes part for illustrations. Included is a CD containing Python code files and a word/PDF document of the report.

# Introduction

## Project start

This paper documents the development process of the 4th semester project “**Autonomous maze mapping and running rover**”, which is an interdisciplinary project, involving the courses ‘Parallel programming’ and ‘Operating systems and embedded Linux’.

The project comprises the construction of an autonomous rover, using available hardware parts provided by supervisors, along with the development of a software solution to control the robot and map the maze. The software solution also includes the implementation of a client computer that will request the mapped maze, and show it graphically.

Given a multitude of different technologies, software strategies and general development approaches. We as a group will discuss and compare these to delimitate our project solution, and present what we believe is the correct and best solution to our project requirements from an engineering point of view.

## Problem formulation

We want to explore the different approaches to navigate an autonomous robot and find the most appropriate inputs and algorithms to interpret these inputs in order to map a maze and avoid collision with the walls by driving straight. Furthermore we want to generalize the mapped maze into a data-model and a graph for use in path finding as well as a way for the robot to receive this path and drive to the target destination.

## Milestone plan

The milestone plan is enclosed in the appendices: see Appendix 1

# Problem analysis

## Requirements

|  |  |
| --- | --- |
| **Functional Requirements** | |
| **R1** | The robot must be able to map any maze. |
| **R2** | When the robot has finished mapping it must be able to transfer the map to the PC. |
| **R3** | The PC displays the map in a GUI. |
| **R4** | The PC will be able to find the fastest path between two points in the map. |
| **R5** | When a path has been defined it will be transferred to the robot. |
| **R6** | The robot converts the path into instructions, traverses the path and stop at the target destination. |
| **R7** | During mapping and traversal the rover should record a log of sensor readings and motor instructions. |
| **R8** | The robot should be able to autocorrect its direction to avoid collision. |

|  |  |
| --- | --- |
| **Non Functional Requirements** | |
| **R1** | The robot must use 2 stepper motor. |
| **R2** | The robot must use a Raspberry Pi model B for controlling the unit. |
| **R3** | The robot must use 3 sensors for determining its surroundings. |
| **R4** | The robot must be programmed in the object-oriented language Python. |

## Analysis

### Navigation

* The robot needs to turn using differential drive.
* The robot will use sensors in order to detect surroundings.
* The robot has to use the sensory inputs to autocorrect its position in the middle of the path.
* The robot should be able to log important information about I/O as well as the navigation logic.
* The robot needs functionality to calculate its traversed distance.

Identified problems:

|  |  |  |
| --- | --- | --- |
| Subject | Problem | Proposed solution |
| Sensors | The robot needs a way to identify surroundings | IR-rangefinders, Ultrasonic sensors |
| Odometry | Distance traversed, current position and direction | Wheel-encoders, Mouse-sensors, Step-counters |
| Auto correction | The ability to stay center in the corridors | PID, iterative trigonometric corrections |

### Mapping

The robot will need to be able to map the maze autonomously.

It has to use an algorithm that takes care of loops and dead-ends but not open spaces when its mapping the maze.

The robot needs a data model for storing cells of the maze which are already explored, and a way to transfer this model to the PC.

The PC has to be able to display the maze in a graphical interface as well as take commands from the user.

When the robot has finished mapping the maze it has to stop and wait for further instructions.

The robot has to be able to traverse and map any maze excluding open spaces.

Identified problems:

|  |  |  |
| --- | --- | --- |
| Subject | Problem | Proposed solutions |
| Mapping algorithm | Explore and store the maze without redundancy. | Depth first search, flood fill |
| PC communication | Transfer the maze model to the PC | JSON, Pickles |
| PC GUI | Receive a maze map and display it graphically | Java/Swing, Python/QT |

### Path finding

The PC has to be able to find the fastest path between any given two points in the maze.

The PC needs to be able to take user commands and in the end send the path to the robot.

The robot will have the ability to translate the received path into instructions for driving through the maze and stop at the target destination.

The robot has to return to idle state when it's finished driving a path.

The robot has to be able to transfer its current position to the PC so the PC can use this position as source when finding a path.

The robot has to be able to stop at any given target in the maze including points in the middle of a corridor.

Identified problems:

|  |  |  |
| --- | --- | --- |
| Subject | Problem | Proposed solutions |
| Fastest path algorithm | Find the fastest path between two points in the maze | Dijkstra, a-star, flood fill |

### Choice of programming language

The software on the robot has to be object-oriented. The two main programming languages of choice have been C and Python. We want to use an interpreted language since development will be faster without cross compiling. We will focus on a language with a large standard library that runs same code on many platforms. The language needs to be able to use the I2C bus of the R-Pi.

As a Python developer, you can spend most of your time focusing on problem break-downs and data type design. These are important components of programming in any language; Python just lets you get there more quickly, as a result of easier syntax.

Python’s interactive interpreter allows you to test features while programming. This is a real advantage because you can see what a particular code snippet does in real time and modify it for desired results without the need for compilation.

|  |  |  |  |
| --- | --- | --- | --- |
| Python | | C | |
| Pros | Cons | Pros | Cons |
| Easy syntax | Slow | Hardware near | Difficult syntax |
| Extremely fast coding | Global interpreter lock | C examples during lectures | Slow coding |
| Many libraries |  | Fast runtime execution | Needs compiling |
| No compiling |  |  | Needs cross-compiling |
| Interpreted |  |  |  |
| Easy testing |  |  |  |

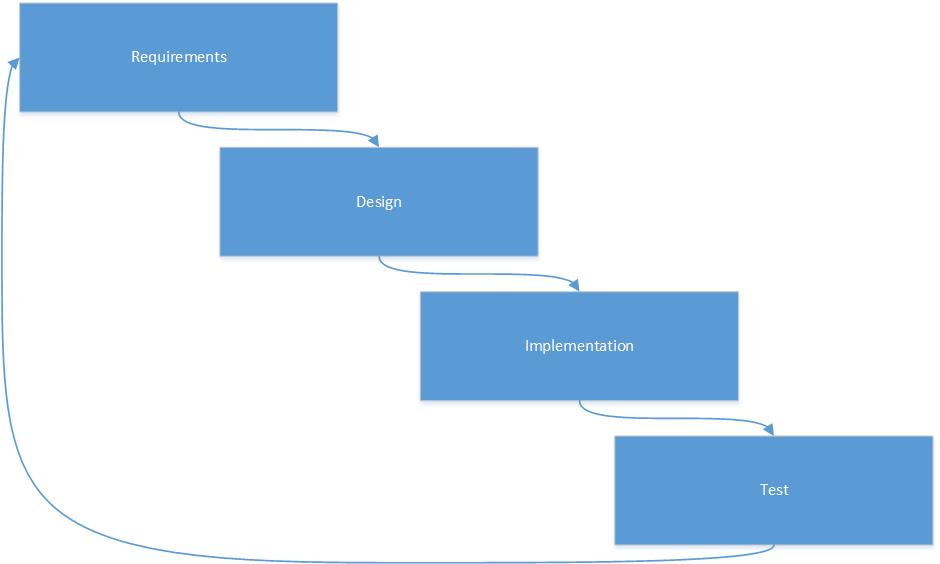
Identified problems:

|  |  |  |
| --- | --- | --- |
| Subject | Problem | Proposed solution |
| Language | Find an OOP language that runs on arm/PC | Python |

# Proposed solution strategy

We plan to use our own version of an iterative deterministic process development system, which mainly focuses on the phases of design, implementation and testing of code.

The development of the robot and code takes foothold in the identified project requirements, where the design of the solution to a requirement is first discussed thoroughly, before taken into the implementation phase.

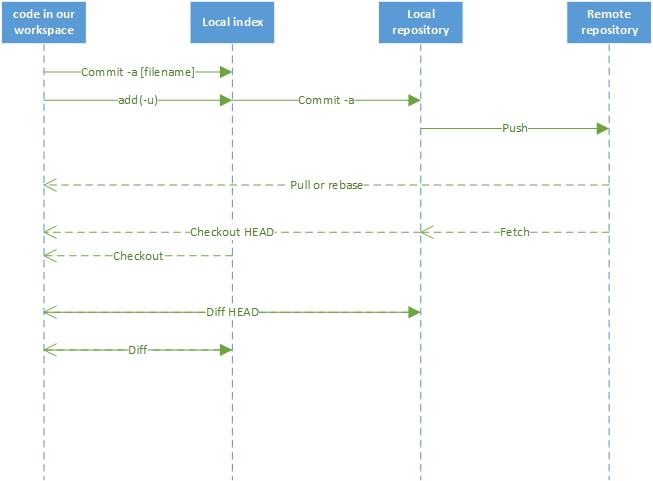


The development process is structured around maintaining a weekly ‘week description’ which summarizes and documents all project activities of the week. This week description will keep track of project, and code development as well as time spent on different problems, in order to backtrack and pinpoint badly selected choices.

## Git

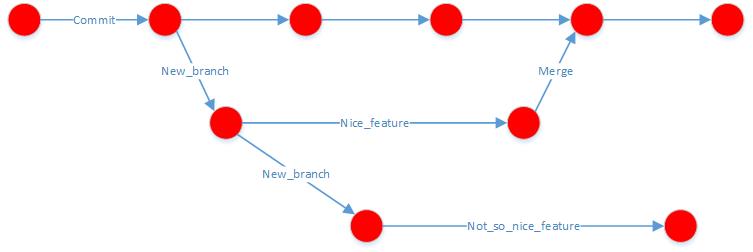
The software solutions to a given problem will be written in Python. All group members will keep their own local workspace synchronized with an up-to-date remote repository, through the use of Git.

Git is a distributed revision control system that allows us to develop project software, without the need to maintain a connection to a common network. Git will let us create a remote repository to contain project code, and provides an excellent version control system, to make sure local and remote code is up-to-date.



### Branches

One of the most efficient Git functions we plan to use is branching. Branching allows us to create a new branch of code that will extend a current branch (most likely the master branch). This means that if any developer wants to enhance the current code, they can do so by creating a new branch and code their new feature. If this new feature suits the group of developers as a whole – it can be merged into the current master branch.



### Issues

Throughout the development process, we plan to use the Git build-in feature of ‘Issues’ that allows a user to create an issue based on current code, or missing code. Every developer can see, follow and contribute to open issues by creating a branch and open a ‘pull request’ to be reviewed by other developers, and eventually merged into the master branch.

## Resources

### Hardware

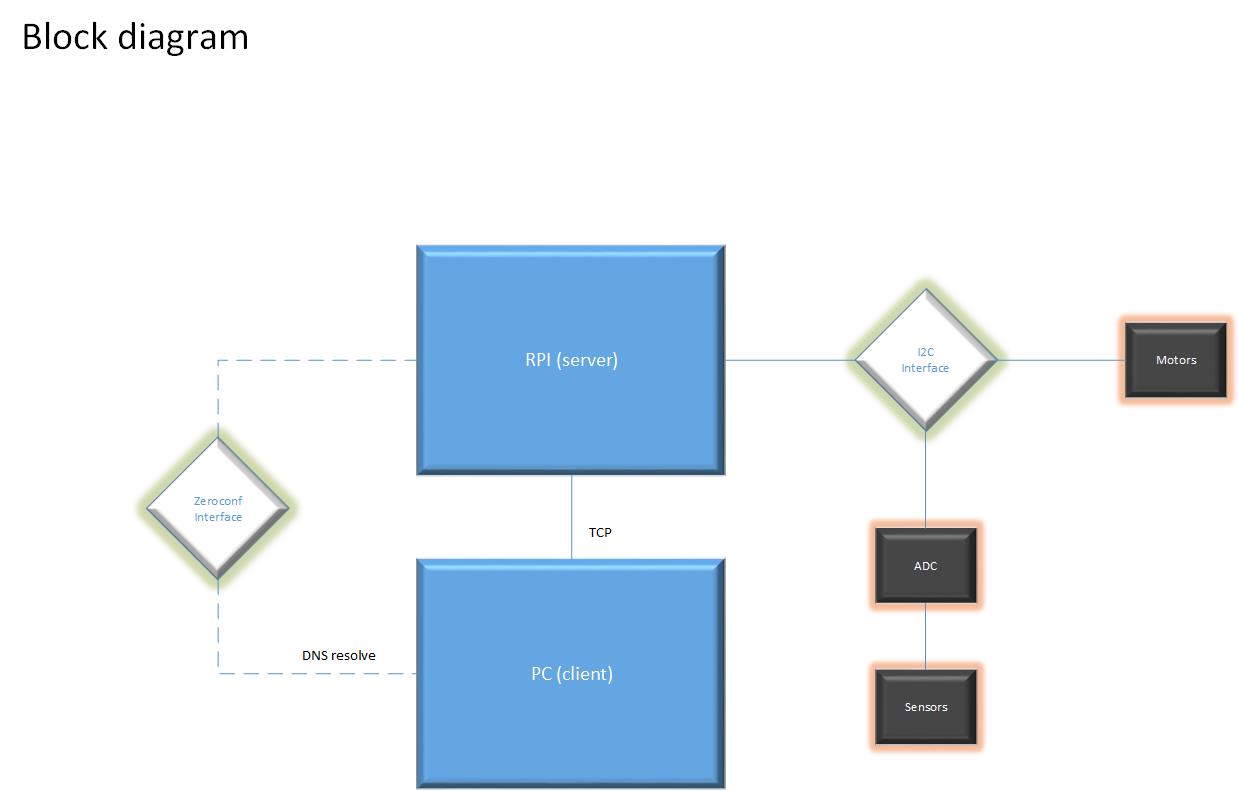
* Robot platform with DC motors
* Robot platform with Stepper motors
* Motor controllers for DC motor
* Motor controllers for stepper motor
* Ultrasonic sensors
* IR sensors
* AD7998
* Mice
* Raspberry PI´s
* WIFI USB adapter
* I2C expansion board
* Voltage regulator
* Battery pack
* Laptops
* Router

### Software

* Eclipse
* Github
* Dropbox
* Google drive

# Problem solution

Figure XX will show an overall block diagram



The robot involves two main functions. The ability to map the maze and the ability to over and over go through the maze to a selected target using the shortest path.

We saw the need for a state machine to handle which action we were in the midst of performing with an extra state where we could wait for instructions.

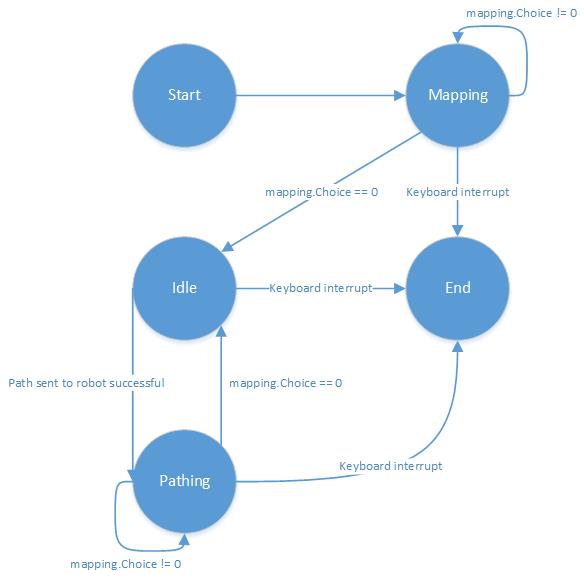
We divided the robot into three states:

1. Idle
2. Mapping
3. Pathing

In idle the robot would be able to receive requests and respond with data.

In Mapping the robot would traverse the maze and build a data structure representing the maze for future transmission to the PC.

In “Pathing” the robot would go through a path received from the PC and stop at the target cell. Figure XX will show a state machine diagram



We wanted to run the two main functions in threads. This would give us the ability to check if a thread was alive and evaluate it instead of using critical zones.

We designed the threads so when they had finished their tasks they would self-terminate and in the end be joined together with the main process.

## Navigation

### Design process

The design of a navigation system, is a very fundamental requirement of the project, and one of the issues that we addressed first. A navigation system requires the collaboration of several hardware parts, including multiple sensors and motors. The sensors provides the basis of the ability to auto-correct the robot, and as such, it is an important choice to consider. The ability to auto-correct implies the capability of detecting heading displacement and distance travelled.

In the very beginning of the design discussion of the navigation system, we had to primary consideration to provide this capability – wheel encoder- and optic-mice odometry. Wheel encoders were considered but not investigated or implemented further; however mice odometry were given a lot of our time and were also implemented to some extend, before discarding it as a solution. It turned out that the math behind using the optic mice as sensors were too complex – at least with the timeframe that we had given ourselves[[1]](#footnote-1). Instead the solution fell upon using Infra-red(IR)-sensors to provide the feedback for auto-correction.

The robot navigates using three infra-red sensors, which are mounted on each side, and on the front. These sensors tell the robot about its current surroundings, but not where in the mace it is. The two stepper motors will adjust their state based on the sensor input, thus the correlation between the sensors and motors provides the foundation of the navigation through the mace.

Robot to sensor/motor communication takes place using the Inter-Integrated Circuit serial communication protocol over the Raspberry Pi. One of the very first issues we were facing, was choosing the RPi baud rate on the I2C bus. If this value was either too low or too high, we would receive complete rubbish from both motors. Such basic motor data as actual- and target position would turn out to make no sense, which ultimately would make it impossible to implement the navigation system, since we need to the robots actual- and target position at all times. The issue was fixed by adjusting the baud rate based on trial and error[[2]](#footnote-2)

The infra-red sensors need a certain amount of time to convert the analogue input to digital output for the robot to react on. This means that this time is a limiting factor, with regards to how many times a second it can react. This has been a huge concern for us throughout the project, because of the fact that we wanted to use PID to autocorrect the robot. The more valid readings we can receive each second, the better. The average time it takes before we can read valid converted analogue input is 38 ms, so if we want to take an average of a few samples, it becomes very time consuming. As explained in the theory section of the sensors earlier, the alternative is the ultrasonic sensors, however these require even more time due to provide a valid reading. In terms of using a PID feedback controller to auto correct, the obvious choice was then to use the IR-sensors to provide the feedback to react on

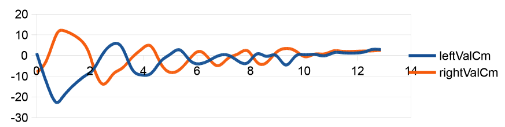
#### PID controller algorithm

The PID auto-correcting algorithm is the heart of the navigation system. As described in the theory section, PID is a feedback controller algorithm that uses feedback (sensor readings in our case) to determine its own algorithm output. Our idea of using this form of auto correcting is its brilliant ability of creating a damped auto-correction curve. Alternately the movement curve of the robot would be similar to an oscillating curve, where the robot overshoots every time it tries to keep the heading.

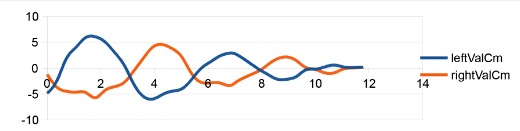
In order to get the desired damped curve, we needed to calibrate what’s called the gain factors of the output of the algorithm. PID operates with three types of errors: **current** error, **accumulated** errors and **predicted future errors**. Each error has got a constant multiplied on it (the gain factor). By changing these gain factors, it is possible to create various forms of damped curves that fit the need of the developer. It is possible to use the so called  **Ziegler–Nichols tuning method** to tune the gain factors; however we used the trial-and-error approach.

Output = Kp\*error + Ki\*integral + Kd\*derivative

We tried to tune Kp, Ki and Kd such that the robot overshoots a maximum of two-three times before settling around the set-point of the maze[[3]](#footnote-3)



**Graph 1: Kp=4,1 Ki=0 and Kd=0,2**

****

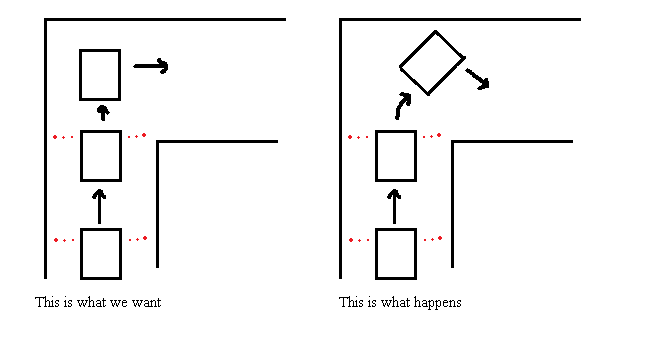
**Graph 2: Kp=1, Ki=0, Kd=0,3**

The two above graphs shows the auto-correction curve (centimeters as a function of time (s))based on two sets of PID gain factors. The curve of graph 1 obviously has too big of an amplitude in the beginning – this would result in wall collision between the robot and walls. Graph 2 has got half the amplitude of graph 1 and only a couple of overshoots before settling – this set of PID gain factors turned out to be suitable for our navigation system.

#### Development process - considerations and issues

In the process of designing the navigation system, we experienced lots and lots of issues; both with hardware as well as the implementation of the code to make the hardware work. The whole idea of the development strategy (as described in chapter blabla) was to avoid a stochastic development process, but instead use a deterministic approach. This should culminate in a well-thought and documented process. As a result of this strategy, we often explored two or more possible solution in parallel, which is also the case in the decision of auto-correcting choice.

An issue that we experienced was when transitioning from driving in a corridor, to detecting a missing wall. The frequency of our program main loop is relatively high to make the PID auto correction sensitive, however this high frequency also means that we react several time on the same input and that the sensors will need a certain amount of readings before the output reflects a no-wall case for the robot to react on. This issue has a devastating impact on the robot, as it does PID auto-correction before entering a turn; instead of just stopping to initialize the turn. The result of this is a slightly displaced robot in every turn.

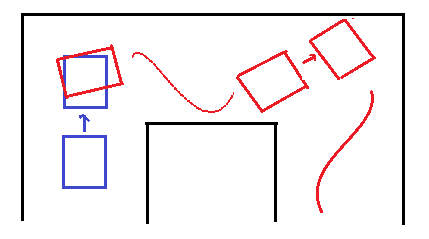


The obvious solution to this very annoying problem (but one we implemented rather late) was to make sure that no invalid readings would be returned from the implemented sensor read function. If only valid readings are returned, no false PID will be performed, which hopefully results in the robot being parallel to the side walls when entering a turn[[4]](#footnote-4)

#### One-cell PID only

As described earlier in this chapter, the PID controller auto corrects the robot based on sensor feedback from two side walls; our implementation does not support single-wall PID. That fact combined with the turn90 issue described above, proved to be a really devastating combination.

A scenario where only one cell is available for use, to auto-correct after a turn, is not rarely seen – and if we drive out of that turn with a wrong angle, we (most likely) won’t be able to regain correct heading before smashing into a wall (even though we fixed the turn90 PID error, the robot still sometimes came out of a turn in a wrong angle).



After countless of failed tries to navigate the robot through the maze, we began to investigate alternative ways of auto-correcting

#### Iterative approach

At a certain point in the attempt to make the PID feedback controller work, we almost gave up, since the PID auto-correction performance was way too sporadic and unpredictable. The other main way of approach since the beginning, but something we hadn’t spared much of a thought, was the iterative way of making the robot drive in the mace. We really didn’t like this strategy, because it would require the robot to make a short stop each time a new heading needs to be calculated; this can potentially mean several stops per cell. Another reason to disregard this approach is the fact that one of our basic robot requirements says that the robot must move in a continuous manner.

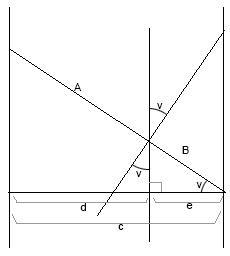
However since the PID feedback controller turned out to cause a lot of problems, we were forced to spend some time investigating this new iterative approach

In the iterative approach, the robot uses trigonometric calculations to determine its placement in the maze, with regards to angle and heading (and travelled position is possible too). This is a more simple way of auto correcting. The general idea in this approach is:

* Read sensor samples
* Use trigonometry to calculate displacement with regards to set point
* Use calculated to displacement angle to determine new heading
* Drive to new position using the new heading

A simple loop is all that’s needed to simulate this behavior – the only challenging part (for some) is implementing the mathematical equations required to calculate a new heading.

We spend 1 day to come up with the math for the robot to be (somewhat) functional[[5]](#footnote-5). It’s based on calculating two angles, created by the beam of the front sensor and two side sensors. Eventually, these two angles tell something about the displacement of the robot.



One additional day was spent to decide if this new iterative approach was a possible solution to the problem of auto-correcting. Even though it did kind of work in the beginning, it turned out that it was quite a challenge to distinguish some scenarios over others. Our implementation was based on driving across the middle of the cell, each time iteration was done – which means that if it did not, it would misjudge its current heading and drive the wrong way in the next iteration. Instead of spending more time on fixing this, we went back to the original idea of continuous movement and PID auto-correcting (turned out to be a good choice)[[6]](#footnote-6)

#### Back to PID

Hardware failures, and abnormal robot behavior was frequently (very frequently) happening throughout the design process of the navigation system. It turned out that some of the weird behaviors were caused by one of the IR-sensors being turned upside down. This resulted in a maximum sensors reading distance of 30 cm. which again caused false auto-correcting and failure to detect missing walls. The genuine reason for returning to PID (beside that the iterative approach was causing us some headache) was the idea that this sensor problem might have been an influential factor in the PID not working properly in the first place.

#### Final PID adjustments

The solution of returning to the PID design, proved to be a really good decision. Remounting the IR-sensor, to face to correct way improved the auto-correction of the robot enormously. The following additional adjustments to improve auto-correction have been made throughout the last weeks of the project:

* Adjusting steps to make turns (90- and 180 degree turns) more accurate
* Adjusting main loop frequency for sensitiveness
* Mounting two motors that are actually in sync

After mounting and implementing the code for the last adjustments, the navigation system works great.

#### Conclusion

The combination of IR-sensors and PID feedback controlling to auto-correct the robot proved quite a big challenge to design and implement. Despite countless of different failed design ideas we managed to produce a solution that works satisfying.

### Implementation

This section documents the implementation of the navigation system on a higher level of abstraction. The implementation of the chosen hardware parts (IR-sensors and stepper motors), for use in the navigation system, are enclosed in the appendices[[7]](#footnote-7).

As mentioned earlier, the navigation is based on sensor readings from three Infra-Red sensors. These readings provide the foundation of the navigation, since decisions are taken and executed based on these.

A total of 7 Python modules are interacting in order to navigate through a maze; each are either responsible for a particular function, or encapsulating important data for

use of the other modules.



The control flow of the RobotNavigator module is as follows:

1. Read samples from all three sensors
   1. Check if samples are valid, else read again
2. Get current robot surroundings, based on the sensors input
3. Decide if the robot is driving in a corridor
   1. If yes: do PID auto-correcting
   2. Else get the number of steps driven so far and current surroundings and pass them to the mapping module in order to make a turn decision
4. Execute decision

A detailed sequence diagram showing the a single loop of the RobotNavigator module is enclosed in the appendices[[8]](#footnote-8)

### Test

Testing was split into two parts; white-box- and black-box testing. Through the implementation of code, white-box testing was used on different code-blocks to check if output was as expected. The different parts of the navigation system were then black-box tested individually, before assembling it and doing an overall black-box test of the whole system.

**Testing the parts:**

1. We did a blackbox test of the IR-sensors, where we tested the raw input of the sensors, as well as the converted input to distance in centimeters. Furthermore, we did a test, where all three sensors (mounted on the robot) were placed in the maze, and checked for errors[[9]](#footnote-9)
2. The testing of the motors was a two-step process – one to test the actual communication between the RPi and the motor controllers, and one to test for valid return data (data that made sense). This were done by adjusting the baud rate of the I2C bus, and by making so called ‘Python decorators’ to check the status of the motors[[10]](#footnote-10)

**Overall test:**

After integrating the different parts, the navigation system was tested, as whole. This was done by adjusting the maze so that the robot could move in a straight line down a maze corridor. Graphs were produced based on the outputof the feedback controller algorithm, to check if the robot behavior were satisfying[[11]](#footnote-11)

## Mapping

### Design

When faced with the objective to map a maze we first needed to generalize the specifications of a maze. The maze we were given was constructed of square cells side-by-side in a 2d coordinate system.

The square cells are divided either by an opening or a wall.

In 2.2.2 it was specified that the exam maze could held loops and dead-ends but not “islands” defining cells with walls in all directions.

We tried to outline the actions a robot would need to make in order to traverse the maze. The robot would need to turn, drive straight, make U-turns in dead-ends and have some sort of memory in order to avoid getting stuck in loops in the maze.

#### Position

In order to map a maze the robot needed to be able to reposition itself inside the rows and columns of the maze, as well as a way of detecting whether a cell has been previously examined.

To reposition itself the robot needed a way of knowing the range it had traversed in a given direction. We decided to use steps from the stepper motors as our approach towards odometry. This would work since our PID algorithm would help keep the robot in the middle of the path, and thereby the counted steps traversed would come near the straight line between two points in a row or column.

#### Coordinate system

In order to explore the maze, and remember where the robot had been, a data structure for the underlying coordinate system was needed. Inside the data structure would be a table with cells, each cell occupied by a description of the walls surrounding it.

When we had settled on a way of position the robot and a data structure to hold the cells of the maze, we moved our focus to a procedure of exploring the maze.

#### Mapping algorithm

We started at the blackboard by drawing a maze similar to the current layout of the test maze, and tried to identify movements to successfully complete the mapping. We saw that a maze consisted of four scenarios:

1. Corridors  
   The robot changes position in either rows or columns
2. Corners  
   The robot changes position in both rows and columns
3. Crossroads  
   The robot has to make a choice between different possibilities
4. Dead-ends  
   The robot can only make a U-turn

Of the four scenarios we started with the first. Since we already had auto-correction and a way of counting steps implemented, we decided to only make mapping actions in the other three scenarios.

Scenario 2 and 4 had a similarity in that only two possibilities existed in both: “stop” or “continue”. The difference was the option of “continuing”.

“Continue” in scenario 2 meant “turn” however in scenario 4, it meant “U-turn”

Since differences were minor these scenarios were classified as similar.

Scenario 3 stood out as representing the points where the path would branch out in different directions.

We decided to use “Depth first search” to explore these branches. This algorithm normally suffers from infinite loops if faced with a loop, however since we had a strong coordinate system with memory capabilities underneath this would not be a problem.

We specified our priority for the algorithm as following:

1. Straight
2. Right
3. Left
4. U-turn

Since scenario 3 was defined a situation with lack of auto-correction we felt that the priority of decisions could be neglected since both outcomes would result in insecurities regarding actual position and thereby accumulating error until next scenario 1.

#### Robot / PC - strategy

When the robot had traversed the maze and build up a maze model it would need a way of being transported to the PC application.

Since we already had a working WIFI connection to the robot we decided the PC should be able to request the maze over network when the robot had finished mapping the maze.

We then looked at the different possibilities of transporting our maze model data in a way that were standardized in both ends. Python offered pickles but since we had not decided the language of the PC application yet we decided to use JSON objects.

At the PC we explored the capabilities of Python doing the GUI and found that the QT library was sufficient in providing an easy “model-view-controller” patterned approach.

Briefly we had thoughts off using Java for the development of the PC application but using the combination of Python and QT kept us in a mono-linguistic development phase which eased our focus.

An application was developed in Python with a GUI and functionality for requesting mazes from the robot and displaying them. See Appendix XX for application screenshots of central functionality.

We wanted our robot and PC to communicate without too much setup, so instead of having to configure options as IP in the network, we wanted a more automatic approach to the infrastructure. We looked at making our own application protocol for establishing the TCP interactions but eventually a solution based on Zero-Configuration was implemented.

We wanted a simple communication platform, so we decided to use “request-response” with the PC being the client, and the robot being the server.

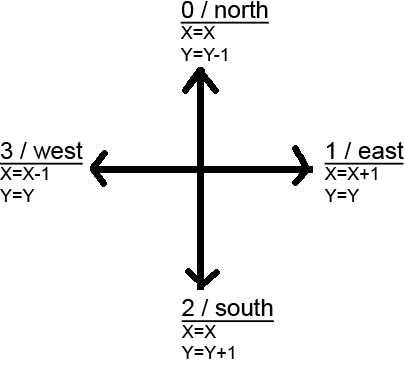
Looking at our state machine's three states, we saw the risk of deadlocks since the robot simultaneously ran a TCP server taking requests for a map, while the robot was building the map.

We solved this by running the “Mapping” state and “Pathing” state inside a thread. By doing this we could check and see if the thread was running, before trying to obtain the map data. This solution was preferred due to simplicity compared to having a variable used to change the state inside a critical zone.

### Implementation

We started by creating the coordinate system and orientation concepts to move within.

In order to keep track of direction we decided to use cardinal directions as a global reference.

In FIGURE X underneath we show how we define the directions. 

By using these specifications we can also specify the rules of the cells in the maze.

A cell can be one of many situations that all can be generalized into a binary pattern using the defined directions.

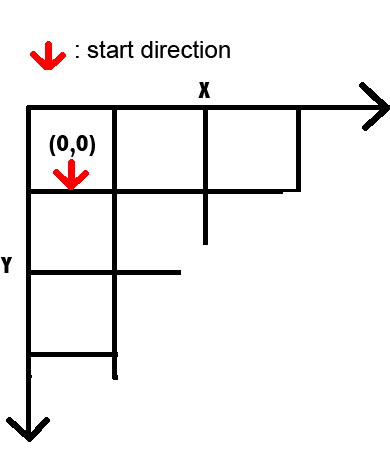
By using binary representations where 1 defines a wall and 0 an opening we can clockwise define a cell by the following list:

[north, east, south, west]

For example a cell with an opening north and west would be:

[0,1,1,0]

When placed in the maze the start direction of the robot is south and its start position is (0,0)

)

When mapping the robot goes through the flow described in Appendix XX

It starts by sampling the IR-sensors and parses the sample to a function that identifies which directions are blocked by a wall. The result of this evaluation is locally defined [left, right, straight] where 1 represents a wall and zero represents and opening.

We use this value to identify the different scenarios previously described.

When scenario 2 to 4 is experienced the walls list is passed to the mapping object that returns a locally defined choice with the exception that the mapping object also can return 0 recognized as “finished mapping”. If “0” the Mapping thread will set an event which will stop its while loop and in the end terminate itself.

The Mapping object will take care of making a local choice of direction based on current position and direction.

It does so by implementing a stack. Items are added to the stack at all corners, dead-ends and crossroads. At every iteration the Mapping object goes through the process described in Appendix XX

When the Mapping object adds to the maze table it writes an integer into the cells. The integer only has 4 bits used being the binary representation of the globally oriented walls at the current cell.

Example:

A cell with 4 walls north and east and openings south and west will be [1,1,0,0] and in the table it will be saved as an int with the same binary pattern: 0b1100 or 12.

Saving the walls list as an int simplifies the data structure and makes it easier to transfer to the PC as a JSON dictionary object.

In the Mapping object we used a simple procedure to translate local directions into global directions:

FIGURE X

For all the following it the result is wrapped to be within the boundaries of 0 to 3 according to the global directions IS IT REALLY ZERO TO THREE?

* Straight is current direction
* Left is current direction - 1
  + if < 0 then 3
* Right is current direction + 1
  + if > 3 then 0
* U-turn is current direction - 2
  + if < 0 then += 3

#### Robot / PC communication

When we implemented the TCP server at the robot we wanted it to be flexible and future proof.

Based on the chosen “request-response” model the PC application transmitted a JSON object with a key called “message”.

We chose to use “callback” functionality by implementing a dictionary inside the TCP server. in this dictionary you could add your own function and a message value it should react on.

Our TCP server receives and strips the JSON objects and looks for the key:“message”. it then takes the value from that key and tries to look it up in its dictionary of callbacks.

If it finds a callback it will run it with the value of the key:”params” also in the JSON object.

When the robot starts it registers its own callback functions at the tcp server. It registers functions to take care of sending maze models, receiving paths and sending current positions.

When asking for a maze our JSON object would look like this:

***{'message':"maze"}***

### Test

The mapping part was developed in a close cycle of testing and implementing.

We blackbox tested the mapping related functionality in two ways

1. Tests where the robot part could be simulated in a virtual machine
2. Tests that involved the robot in the actual maze

Using virtual environment we tested the parts involving communication between the robot and the PC. These blackbox tests can be seen in TESTCASE-1.

Using the actual robot we blackbox tested the ability to map a maze as seen in TESTCASE-3.

We also verified that the robot was able to write a logfile while driving.

During the process we used regressive testing every time we needed to fuse big parts into the main branch. Here we tested to see if new functionality had broken the software at placed previously been debugged or even new bugs that could have been created.

We whitebox tested our classes using main methods that were developed to test functionality. We used these to test the classes themselves every time code had been rewritten or reorganized. This process resembled unit-testing however in a more manual approach.

During the endphase we used destructive testing in order to reveal missed bugs etc. This proved a good idea since many parts had been left untouched while focus had been shifted during the process. Destructive testing revealed the loose ends otherwise forgotten.

From the tests we concluded that our mapping was functional and worked in the tested scenarios on mazes of different layouts and sizes. However our autocorrection from time to time served problems in such a way that in rare occasions we would leave a turn with such an angle that steps would be lost and the robot would lose its exact position in the coordinate system.

## Path finding

### Design

The robot had a main task of being able to go from a point to another point in the maze by following the fastest path.

Since our robot only used 90 degree turns, the fastest path would not only be lesser traversed cells but also lesser turns.

#### Defining the graph

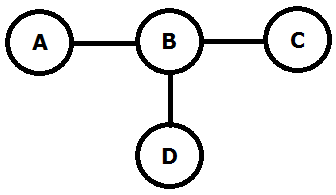
We started out early by planning to use flood fill algorithm. We wanted to use flood fill algorithm because it at the same time provided a way of traversing a maze creating a map as well as finding the fastest route between two points.

After sometime we realized that we couldn’t include the turn cost with the flood fill algorithm which we had to discard due to this and start finding a new algorithm for our path finding challenge. Another reason was also that we wanted to separate the process of mapping and path finding to the level of the robot doing the mapping and the PC doing the path finding.

Later in the process when the mapping process had spawned the specification of the maze model SEE REFERENCE we began researching the different ways of making path finding on the data structure in the maze model.

A critical point in most path finding algorithms is the use of a graph.

We focused on a way of translating the data of our maze model into usable basis for creating a graph.

For solving the problem of designing relations between each cell of the maze with a cost we looked into parent -> neighbor relation. In the illustration (FIGURE XX) the cost between the cells will be explained. In the case we will use cell A B C and D.

They are connected in the following way A to B (AB), B to C (BC), and D to B (DB). The cost between AB, and DB, will always be the same, in this example let’s call it 10. But the cost from BC will change depending on the previous node was AB or DB.

The reason for this cost different is, turns takes 3 times longer for the robot to execute then driving straight thru a cell. In this example if the robot is coming from BD the cost will change form 10 between BC to 30. This value will be used for the algorithm to find the fastest path.

The graph is implemented into a coordinate system where each note will have its own X, Y coordinate. To determinate whether the notes are a turn note or a drive straight note, we are looking on the change in X, Y coordinates. If only one of the coordinate’s changes, e.g. X to X1 and Y stays the same, we know that it a straight thru turn, but if both X and Y changes to X1 and Y1 we know it’s a turn and the cost will be adjusted. Using this data structure in theory look very easy to read and detecting errors.

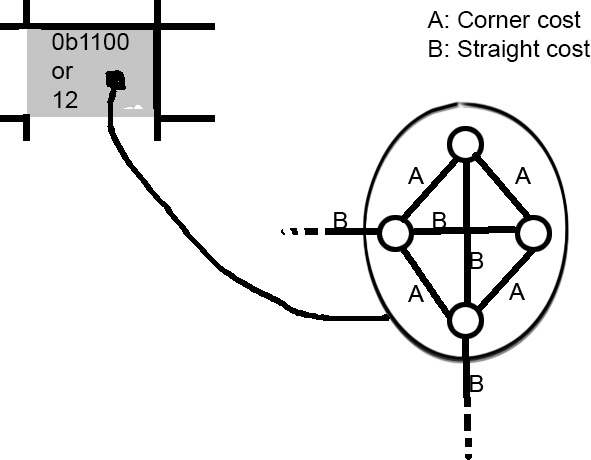
We started to investigate A-star which looked promising as a solution to the path finding issue since it had a heuristic function that would lower the search arena and thereby have a much faster time finding the path.

We started implementing A-star and it was able to operate on our graph using the turn cost process using parent-neighbor comparison but then we encountered a new error where we did not get the shortest path sometimes.

This error was unknown and still not explained why it happened. We tried to isolate the error by flipping off the heuristic function as well as trying different A-star implementations as well as increase the turn cost to 20 times the straight cost.

Since we were running late according to the milestone plan we decided to focus on creating a graph from scratch that already incorporated the turn cost differences.

After sometime we came up with the idea that creating 4 micro nodes (north, east, south, west connected as Figure XX shows) in every cell it would be possible to detect if we were turning and giving it an extra cost.



Every cell has at least one micro node that is connected to a neighboring cluster of micro nodes, the cost of the travel from those nodes would be a straight cost. The turn cost is only inside the cell with the micro cells within, north- south and east-west is the only ones with a straight cost but its bidirectional.

The next step after creating the new graph was to implement the Dijkstra algorithm to traverse it.

Our whole project has regarded the maze as undefined in size and thereby scalable. So we wanted to use Dijkstra to find specific paths and not pre-calculated all existing fastest paths from all locations to all locations. Thereby we chose to use CPU power over ram which makes sense since we run the path finding path on a PC.

#### Robot / pc strategy

We wanted the PC to transmit the path back to the robot over our established networking platform. At the robot we wanted to be able to receive a path with instructions and traverse the map using the instructions.

First we tried to send steps with each position in the final path. But this showed insufficient since the step counter often was imprecise. This would lead to premature instructions where the robot would execute a choice leading to wall since it thought it had reached the opening.

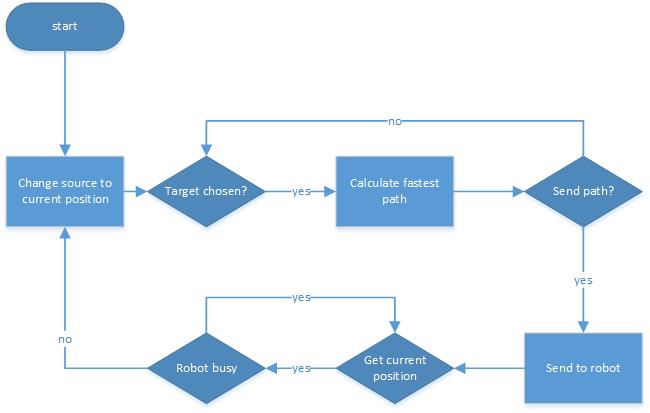
Our solution was a mix of the ability to react on scenarios where we broke off from driving in corridors together with the ability to drive to a certain position inside a corridor.

This solution meant that the robot would receive choices from the path at every dilemma (turn, dead-end or crossroad) and only in the last segment of the path the instructions would include steps. This would solve the problem of being able to drive to a point in the middle of a corridor where no dilemma existed.

To solve the issue with imprecise steps we decided that during the “Pathing” process the robot would traverse a known map and thereby loading its current position from the path would be good enough.

We made the path finding and pathing modules around the idea that the PC would acquire the current position of the robot and use this position as the start of the path.

In practice this meant a workflow described in following figure:



### Implementation

Graph:

Inside a structure of nested for loops we evaluated the content on an inspected cell in the maze model table.

The item in each cell was an int containing a bit pattern describing the walls surrounding it. So in order to interpret it we had to loop through all four cardinal directions. We checked for walls by using bit operators checking if 1 shifted in to the current inspected direction would match a with the content of the cell.

***for d in range(4):***

***tmp=1***

***if not (walls &(1<<(3-d))):***

***tmp=0***

***self.nodes[x][y][d]=Node(x,y,tmp,d,dillemma)***

***self.graph[self.nodes[x][y][d]]=[]***

Using the code above we created micro nodes for each existing direction containing a 0 if not facing a wall inside a cell.

Since our graph would be bidirectional the edges would be doubled.

When creating edges we used the following two approaches:

**For connections between micro nodes:**

*for i in all directions*

*for j in all directions*

*cost = straight cost*

*if absolute difference between i and j is not 2*

*cost = turn cost*

*append node i and j to eachother with cost*

**for connections between micro node clusters:**

*for all cells in table*

*if node(x,y) facing east and node(x+1,y) facing west exists with openings*

*append eachother to eachother in graph with straight cost*

*if node(x,y) facing south and node(x,y+1) facing north exists with openings*

*append eachother to eachother in graph with straight cost*

When the graph is created we use a standard implementation of Dijkstra to find the shortest path from source to target. see REFERENCE NEEDED (Wikipedia Dijkstra) for pseudocode.

### Test

When we tested the path finding part we could again divide the process into two testing with virtual robot and actual robot.

We black box tested the path finding algorithm using a virtual environment which can be seen in TESTCASE-4. We tested with a test maze and went through the test many times to check that the end target was the current position in the next test.

We used white box testing to check if the networking setup worked using a main class. Here we tested to see if the event handlers worked as predicted by seeing if the returned values matched what we had set in the event handling methods.

We tested regressively and destructively when implementing new functionality marked by points when we merged into master branch.

Here we checked if the merging spawned new critical bugs or revived “dead” bugs. To solve the issue we used the built-in debug mode of eclipse/py-dev.

To test the ability to drive a given path we used blackbox testing on the actual root driving a defined smaller maze to reduce testing time. This scenario can be observed in TESTCASE-5.

We have concluded that our robot is capable of receiving a path and drive to the destination using only its navigational and “pathing” capabilities.

# Conclusion

## Process assessment

Requirement traceability matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | Test cases | | | | |
|  |  | TC1 | TC2 | TC3 | TC4 | TC5 |
| Requirements | R1 |  |  | X |  |  |
| R2 | X |  |  |  |  |
| R3 | X |  |  |  |  |
| R4 |  |  |  | X |  |
| R5 |  |  |  | X |  |
| R6 |  |  |  |  | X |
| R7 |  |  | X |  |  |
| R8 |  | X |  |  |  |

# Bibliography

Internet sites

<http://en.wikipedia.org/wiki/Brushed_DC_electric_motor>

<http://da.wikipedia.org/wiki/Step-motor>

Datasheet:

<http://www.analog.com/en/analog-to-digital-converters/ad-converters/ad7998/products/product.html>

<http://www.sharpsma.com/webfm_send/1205>

<http://www.avrcard.com/Documents/datasheets/tmc222_datasheet_v105.pdf>

# Glossary

|  |  |
| --- | --- |
| PC Glossary | |
| PID controller | A proportional-integral-derivative controller |
| TCP | Transmission Control Protocol |
| DFS | Depth first search |
| IR | Inferred |
| Git | Github |
| RPi | Raspberry Pi |
| TC | Test Case |
| DHCP | Dynamic Host Configuration Protocol |
| DNS | Domain Name System |
| Zeroconf | Zero Configuration Networking |
| CD | Compact Disc |
| IP | Internet Protocol |
| LAN | Local Area Network |
| mDNS | multicast Domain Name System |
| DNS -SD | Domain Name System Service Discovery |
| GUI | Graphical user interface |
| JSON | JavaScript Object Notation |
| pathing | Term used to describe the path finding mode |

# Appendix

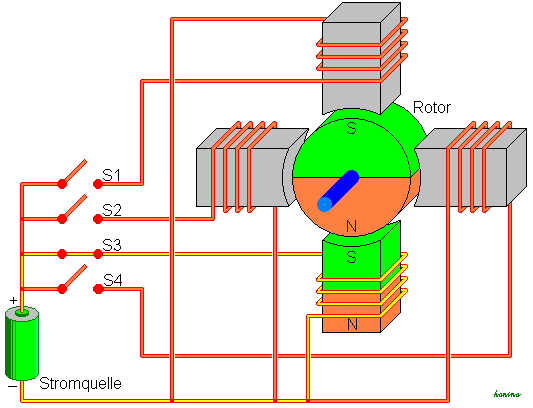
1. Milestone plan

missing

1. Theory

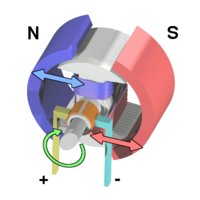
### **Motor**

#### Stepper Motor

A stepper motor consists of a minimum of 4 iron cores. These iron cores are entangled in copper wire and by supplying electricity to these core a magnetic field is created. This is also called a Coil-magnet setup. In the middle of all the cores a shaft is placed. This shaft is magnetic and therefore has South and a North pole. When adding electricity to one of the coils in the shaft will align its North pole with the coils South pole. By alternating between which coil gets power the shaft will start tuning. By only tuning one coil on at the time the shaft will preformed a Full-Step rotation. This will make the shaft turn 90 degrees each time the next coil is activated. Turning on 2 coils, 1 coil, 2 coils, and 1 coil etc. the motor will generate a Haft-Step rotation and the shaft will turn 45 degrees on each impulse. This makes the increments much finer when the shaft is rotation and therefore more accurate. All these different combination of turning on coils is called StepModes.

To control this switching of electricity impulses to each coil, a motor controller is needed. The motor controller controls which coil needs power when, to perform the requested rotation.

#### DC Motor

A DC motor uses almost the same setup internally as a stepper motor, but the magnet-shaft and coils are switch and there are in most cases only 2 Coils. This means that the magnets are on the "outside" and the coil is on the "inside". (See the illustration). A DC motor receives direct current, and uses brushes located on the shaft to switch the polarities. The switching of poles happens internally inside the DC motor

and therefore in most cases a motor controller is not needed to make it work.

Due to the lack of a motor controller most DC motor, is more responsive then a stepper motor and also accelerate faster.

### Auto correction input

#### Odometry

Odometry is the collection of data from actuators, encoders or sensors, in order to estimate a position relative to the starting position.

#### Mouse

The idea of using mice odometry to estimate position is an idea we have had since the beginning of the project. Two mice are used to input relative coordinate-changes to our RPi whenever a displacement of the robot has happened. Several mathematical calculations are done based on the given x,y coordinate pair, and ultimately an length and angle of the movement is calculated. These two values are then used to attempt to estimate the change in position over time, and fed to a PID controller- calculation-algorithm to minimize heading errors, respectively**[[12]](#footnote-12)**.

#### Wheel Encoders

Another alternative to using mouse was the idea of using wheel encoders to determinate distance and direction. Setup on the robot is done by mounting gray-code Ref til et appendix med et gray code wheel to each wheel, with an sensor to read the bit-pattern. When the wheels are turning the sensors will recognize changes in the bit pattern and a position of the wheels can be determinate.

#### Sensors

The robot must be able to determine distances to the surrounding walls, in order to maintain a heading and know when to make a turn. Two primary options are given in this context; Infra-red sensors or Ultrasonic-sensors (or a combination of both). Either choice will work on a robot, but have different impacts on how the robot will function when running a maze. We have to look at the two sensor specifications and the robot behavior requirements, to make a satisfying engineerical choice

* **IR Sensor**

The infra-red sensor (which in our case is a GPD120x Sharp sensor) uses triangulation and CCD array to calculate a distance to objects. It works by emitting pulses of infra-red light, which in case of a present object, will be reflected and caught by the detector. This will create a triangle between the emitter, detector and point of reflection. The distance of the reflecting object will determine the angle in this triangle, which is then used by the CCD array to calculate a distance to the object.

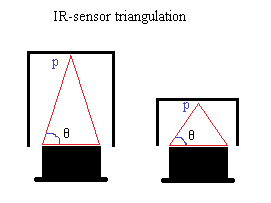


Figure 1: Shows the IR-sensor triangulation, where ‘p’ is the point of reflection and ‘theta’ is the angle created

The sensor output is non-linear with respect to the distance, so it is not insignificant where the sensors are mounted on the robot. Based on the graph of the analog voltage as a function of the distance, one will have to make decision of an optimal reading interval**[[13]](#footnote-13)**.

* **Ultrasonic sensor**

The ultrasonic sensor works differently from the IR-sensor, by emitting inaudible sound to detect surrounding objects, instead of light. We know that sound travels at a certain speed in air, so by keeping track of time elapsed since the sound was emitted and until the echo is detected, a very accurate distance can be calculated.

The sound emitted by the ultrasonic sensors spread out radially, which can cause problems, if several ultrasonic sensors are used, and not placed properly on the robot. If the sensors are pinging too close to each other, one sensor may read another sensors echo as its own. Furthermore materials and object angles might cause other problems such as lost echoes, or ghost echoes.

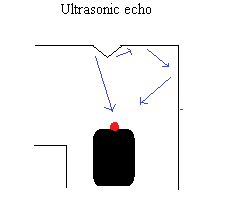


Figure 2: Shows ultrasonic sensor echoes. One echo reflects fine on the wall, the other turns into a ghost echo as a result of the angle on the wall

### Auto correction

#### PID

A Proportional – Integral – Derivative controller is a feedback controller, this means that it uses feedback response to determinate its output. A PID Controller calculates an error value in form of a difference from an input to a reference value.

Readouts from the Process block are added to the Setpoint value. From this sum an error will occur, it can be positive or negative. This error is past into the 3 blocks, P, I, and D block. All these values from the outputs are summed together after they have been multiplied with their individually gain factors Kp,Ki and Kd. The sum is then past into the Process. This loop is repeated until the Possess is terminated. (Look below for illustration)

|  |  |  |
| --- | --- | --- |
| P = Current error | I = The sum of previously errors | D = Estimate around future errors |



#### Iterative

The iterative solution consists of moving cell by cell and doing trigonometry calculations. The calculations is made when the robot have moved to its next target no calculations is made on ongoing movement. The robot moves from its current position to the middle of the next cell. First when it has reached its target the robot will first turn so it will be facing straight down its path, the degrees it needs to turn is calculated with help of the width of cell and the length of the distance read from the sensors + the distance from the sensors to the middle of the robot. Secondly the distance it moves each time is calculated with help of trigonometry by knowing the distance of a whole cell and the distance from half of the width – new distance from sensor + the distance from sensor to the middle of the robot. Finally it will find the degrees it needs to turn to be facing its next target; this is done by taking acos((half of the width – (new distance from sensor + the distance from sensor to the middle of the robot))/( the distance it needs to move)). There are two different scenarios depending on if the robots movement last going to the left or right.

### Path finding

Path finding is the process of finding a way between a source and a target. In computer science path finding is normally associated with finding the optimized path from A to B in terms of cost.

Path finding algorithms operate in a system of nodes connected by edges.

For easy understanding a node can be a station, an edge can be a transport route and the cost can be the time it takes to get from station A to station B. The system is described as a graph.

In such a system terminal cost from station A to station B can vary after the chosen path. You can choose to take route that goes near the harbour for fresh air but will end up with extra minutes used on the bus if the fastest route dictates not getting near the harbour.

In path finding the relationship between nodes is often described as parent <-> child.

In such a way the cost between a parent and a child describes the cost to move on in the process. Where the cost between a child and its parent can describe the memory of how much it cost to get to your present position.[[14]](#footnote-14)

A lot of different algorithms for path finding exist.

Important characteristics of algorithms can be divided in two.

1. Greedy:  
   Greedy algorithms that doesn't remember cost. They look ahead and make choices based on distance to target as well as cost to nodes.
2. Informed:  
   Informed algorithms which takes the cost until present position into account when making choices about future travel. On top of that they can have the characteristics of greedy algorithms with heuristics and cost to next coming nodes.

#### Dijkstra

Dijkstra finds the minimum cost to all nodes from its origin, which can be done for all nodes as its origin.

Dijkstra algorithm is often used for routing as it searches all paths.

If negative edges are used it's not possible to guaranteed the shortest path.

Dijkstra works by defining a source node and a target node. The source node sets a cost to 0 and all else nodes get a cost of infinity.

Dijkstra uses a priority queue to hold info on cost to nodes from source node.

The priority queue is prioritized with minimum cost as highest priority.

Dijkstra iterates through nodes starting with the source node and expanding with its neighbouring nodes.

While iterating it decides between investigating a neighbouring node or going back and investigating another branch defined in previous iterations.

Reconstruction of a path is based on each node having a pointer to its previous node.

At the initialization all nodes have their previous node set to null.

In this way a path can be reconstructed in the end by starting with the target node and iterating through previous nodes until previous node is set to null.

Dijkstra is mainly used for situations where the knowledge of cost to all nodes is needed. This is due to Dijkstra lacking direction in its search.

#### A-star

A-star is an extension to Dijkstra’s algorithm.

It depends on functionality inherited by Dijkstra in its memory implementation.

However it extends functionality by incorporating a heuristic function.

A heuristic function is what gives direction. Its purpose is to give an idea about whether you are nearing the target or moving farther away

Cost in A-star is a combination of heuristics function and cost until now plus cost to next.

H (n) is the heuristic cost from node (n) to the destination.

G is the cost so far + the cost the reach the neighbour.

F is the sum of them.

The heuristic can control the behaviour of A\*.

If the heuristic is 0 then only G counts and A\* will then become Dijkstra which will find the shortest path but explore too many options.

The lower the H is compared to G, the slower A-star gets because it has to search in a larger area.

If the heuristic is too high compared to G then only H matters and A\* becomes Best-First-Search.

If the heuristic is greater than g then you will not always find the shortest path but it may run faster.

If the H is in equilibrium with G then A\* will only follow the best path and never search in other areas this will make A\* work perfect and run really fast.

The shortest path is guaranteed with A\* if the heuristic function is less than or equal to g. In this scenario A-star will use less memory than Dijkstra and run faster. But in order to get their knowledge of the graph and a good heuristic function derived from it must be created.

#### Flood fill

Flood fill can be used to scan an area and check for connections between nodes or for path finding. By flooding the area from a point (start node) to all other nodes you will be able to map the whole area. With path finding you can find each neighbour to the current node and mark it as example where you start will be node number 1 every node that can be connected to this node will be node number 2 then you will investigate node number 2 and mark all possible connections to those as number 3 and continue till you find them all then by numbering you will be able to find the cheapest path to your chosen target and use that path to follow

### Communication

#### Zeroconf

zeroconf or Zero-Configuration describes a process where the IP and port of an application can be registered and resolved automatically by nodes in a network.

In a network of nodes configured by DHCP you get an IP address but it doesn’t supply you with a way of resolving a specific service on a specific host, a task which is normally carried out by a DNS server.

Zero-Configuration solves this gap in a LAN by providing a service structure where servers can register a service and clients can browse for specific services.

Zero-Configuration is an umbrella-term describing many different technologies as mDNS, DNS-SD etc.

Major operating systems implement a library to take care of all these different approaches to resolve local services. Mac OSX uses “Bonjour”, Linux typically uses “Avahi”.

A typical Zero-Configuration setup would be a network printer. The printer would announce and register its service (printing) on a network by using a service name “printer”, a domain “.local” and a service type “\_ipp.\_tcp”.

Service types are defined by the DNS-SD organization and can be seen here:

<http://www.dns-sd.org/ServiceTypes.html>

A client on the network would then be able to use a Zero-Configuration browser either as a complete application or embedded inside an application, to check which services are available on the network. By doing so the client would see a service named “printer.local” with the service type “\_ipp.\_tcp” identifying a network printer.

### Mapping

#### Depth first search

Depth first search (DFS) is used for traversing or searching graph. it marks one node as its start/root and then it explores one of the branches all the way till it can’t go any further then it will start backtracking till it finds a unexplored branch and goes all the way to the end and will repeat this till it have explored every possibility the order which it choses it priority to search is chosen by the user of the DFS. You can go left right and straight in which ever you chose to priorities you set.

#### Graph

A graph refers to a set of nodes (cells) with edges (connection between nodes) connecting.

An undirected graph is a graph where there is no difference between the directions for the edges meaning that from node A to node B is the same, as from node B to node A.

A directed graph is a graph where a edges are not the same meaning that it got a direction from node A to node B, is not the same edge as from node B to node A and its possible to go one way only.

An edge can be weighted meaning it will have a cost (distance/time/price).

The most used ways to store a graph are adjacency matrix and adjacency list.

Adjacency matrix is a matrix that is representing which nodes of a graph are connected together with other nodes. The matrix is n x n where n = nodes. A matrix structure provides faster access but is consuming a lot of memory.

Adjacency list is a list of unordered collection of a graph one list for each node in the graph, each list shows a set of the nodes neighbours. List structures takes more time to access then a matrix but most cases uses less memory.

If the concern is memory then with a dense graph a matrix would be better than a list, but if it’s a sparse graph the list is better option as less memory is wasted.

1. Test cases

|  |  |
| --- | --- |
| **TC1: Receive and display a maze in a GUI** | |
| **Tested by:** | Team 3 |
| **Purpose:** | Verify that the PC can request and receive the maze map and display it graphically |
| **Functional requirements:** | R2, R3 |
| **Preconditions:** | The robot has mapped a maze and is in idle state.  The PC application is open and has identified and connected to the robot. |
| **Test sequence:** | 1. We click the get maze button 2. We wait for the robot to respond 3. The robot responds with a maze 4. The GUI opens a new window and displays the maze graphically. 5. The shown maze is compard with the aschii output from the robot. |
| **Description of the expected result:** | The PC is able to request and receive the map from the robot and displays a map identical to the one visible in aschii at the robot |
| **Result of the test:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:mazeReceivedTC1.pngNIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:mazeSolvdTerminalTC1.png  We checked the aschii output from the robot in the terminal and found that it was identical to the maze shown in the graphical window.  (Both figures are inverted to save ink) |
| **Testers comments:** | By looking at the terminal screenshot you can recognize the content of a maze cell describing its surrounding walls. |

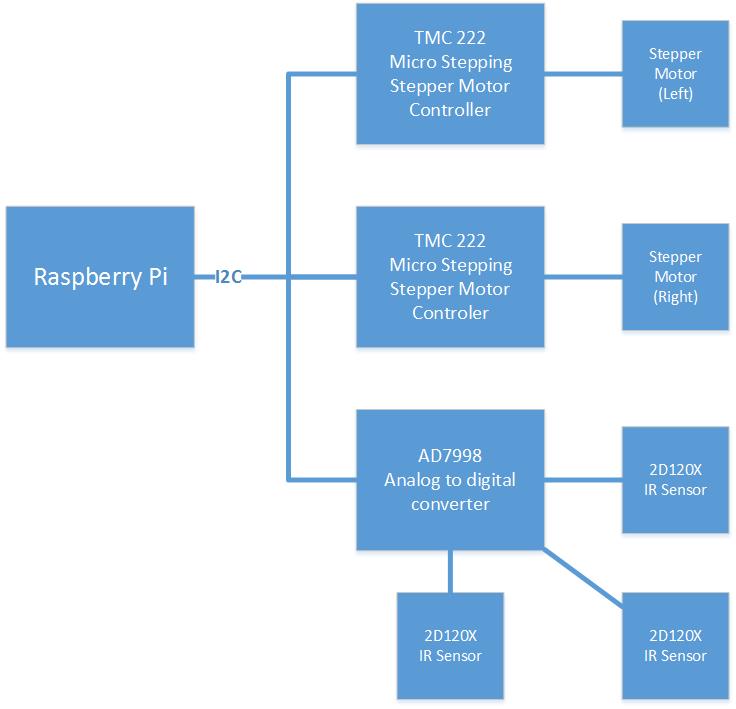
|  |  |
| --- | --- |
| **TC2: Autocorrection** | |
| **Tested by:** | Team 3 |
| **Purpose:** | See if the robot is able to correct itself in a corridor with an angled start direction |
| **Functional requirements:** | R8 |
| **Preconditions:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:robotPidStartingPosition.jpg  The robot is placed as the above figure shows. Facing angled towards the corridor.  The corridor is created with a length of 1.5 meters. |
| **Test sequence:** | 1. The robot is put into mapping mode 2. The robot will use PID autocorrection to align in the middle of the hall. 3. The program will be stopped by keyboard interruption. |
| **Description of the expected result:** | The robot has aligned center in the corridor with few overshoots. |
| **Result of the test:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:PID_KURVER_PG1_IG0_DG0.2.png  The robot aligned center after 4 overshoots over 10 seconds with overshoots maxing out at 5cm from the center of the corridor. |
| **Testers comments:** | None |

|  |  |
| --- | --- |
| **TC3: Map a maze** | |
| **Tested by:** | Team 3 |
| **Purpose:** | Verify that the robot is able to map a maze |
| **Functional requirements:** | R1, R7 |
| **Preconditions:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:robotStartPosition.jpg  The robot is able to autocorrect and is placed in in the maze at the position above facing south. |
| **Test sequence:** | 1. The robot is put into mapping mode 2. The robot will set out from its start position and map the maze exploring first the longest path and then backtracing and exploring sidepaths. 3. The robot will in the end output its maze in aschii in the terminal 4. The maze is verified according to the physical layout of the maze |
| **Description of the expected result:** | We expected the robot to first go straight and only turn if straight was not an option and start explore sidepaths when faced with a u-turn. |
| **Result of the test:** | We saw the robot exploring the maze according to our expectations and in the end it gave us a maze representation fitting with the pysical layout of the maze. |
| **Testers comments:** | In some tries autocorrect made the robot leave a sidepath in an angled fashion making it hard for it to identify travelled distance, however result stayed true to the physical layout. |

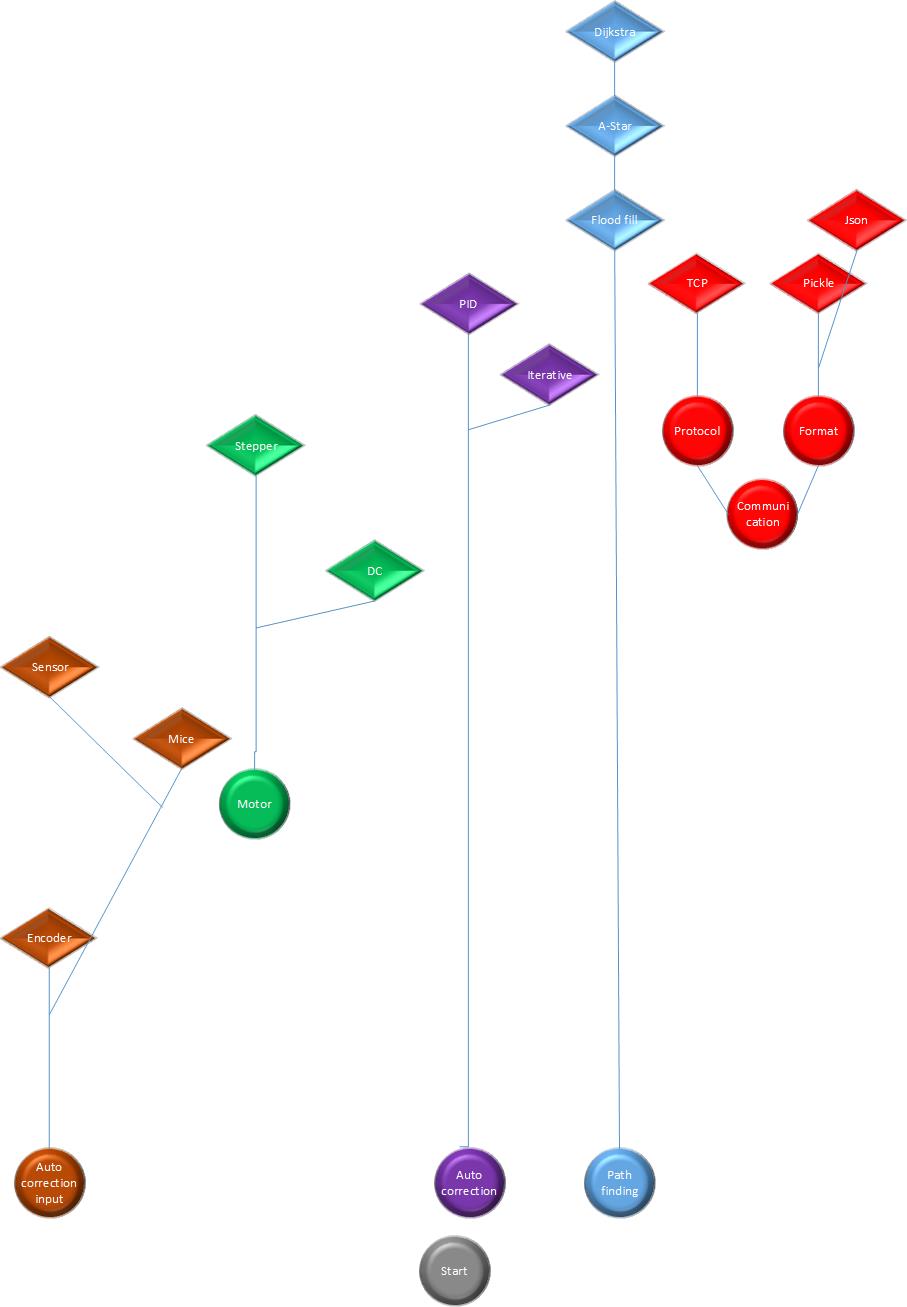
|  |  |
| --- | --- |
| **TC4: Fastest path** | |
| **Tested by:** | Team 3 |
| **Purpose:** | Verify that the fastest path algorithm finds the fastest path with fewest turns. And that the PC is able to send the maze. |
| **Functional requirements:** | R4, R5 |
| **Preconditions:** | Either a robot or a fictional test server is running a tcp server able to send a maze and respond with ”received, ok” when given a path over network.  The PC application is opened and waiting for a robot to register its zeroconf service. |
| **Test sequence:** | 1. We started a virtual server that responded with a known maze structure. The server was running in a virtual machine. 2. The gui application pops up and asks if we want to connect to the robot. 3. We press the get maze button 4. The server recognizes the request and responds with a maze model using JSON NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:fastest path:terminalSendPath.png 5. The GUI opens a view containing a graphical representation of the maze NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:fastest path:guiReceivedPath.png 6. We click the ”select and make path” button an chooses a target destination 7. We verify that it is the shortest path and send it to the virtual robot server 8. The Server receives the path |
| **Description of the expected result:** | We expect that our fastest path algorithm will find the fastest path will less turns and be able to successfully send this path to the robot |
| **Result of the test:** | NIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:fastest path:terminalReceivedPath.pngNIHILDISK:Users:johannes:Google Drive:itsem4project:rapport:Diagrammer:JPG:fastest path:calculatedFastestPath.png  We verified that the PC application was able to calculate the fastest path using a given target point and we could see using the colored circles scattered in the maze which cells the search algorithm had inspected.  We also verified that the virtual server was able to receive the path and reply that transmission was succesfull. |
| **Testers comments:** | The colored dots in the fastest path GUI figure shows which cells the algorithm had inspected colored after when the cell was inspected. Red = early, green = late. |

|  |  |
| --- | --- |
| **TC5: Drive a path** | |
| **Tested by:** | Team 3 |
| **Purpose:** | To comfirm that the robot recives the right path and follow the instructions correctly towards the target. |
| **Functional requirements:** | R6 |
| **Preconditions:** | That an user have sent a path |
| **Test sequence:** | 1. The robot have successfully received the path 2. The robot drives the received path using the instructions within. 3. When the robot have no instructions left it goes to idle state. 4. We request the current position from the robot and verify it as our chosen target position. |
| **Description of the expected result:** | We expect the robot to be able to receive the path and drive to its target destination. |
| **Result of the test:** | The robot drove until the target destination and reported a correct position back equal to our chosen target destination. |
| **Testers comments:** | none |

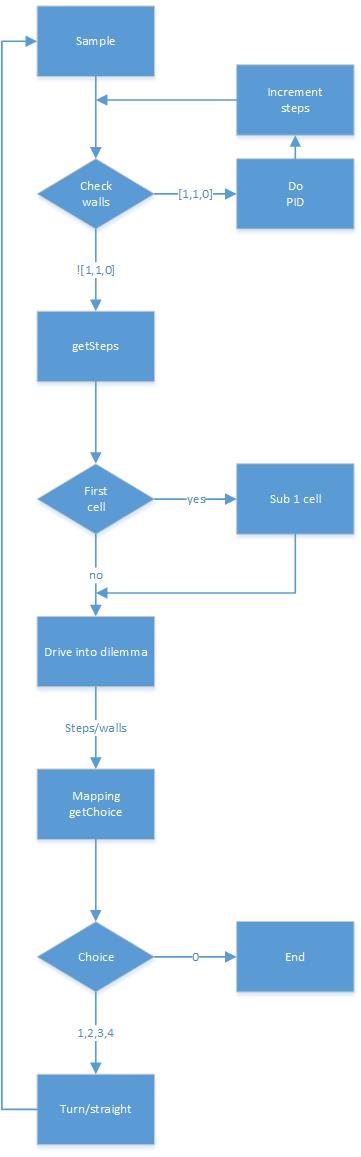
1. Hardware diagram



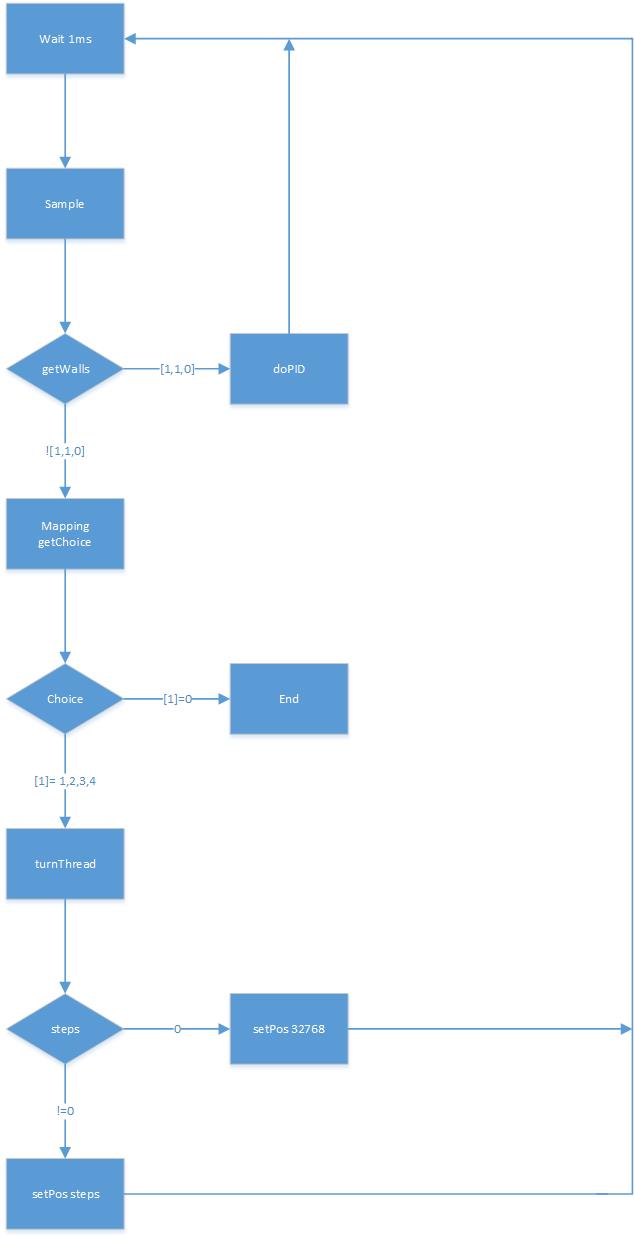
1. Process diagram



1. Mapping logic flowchart



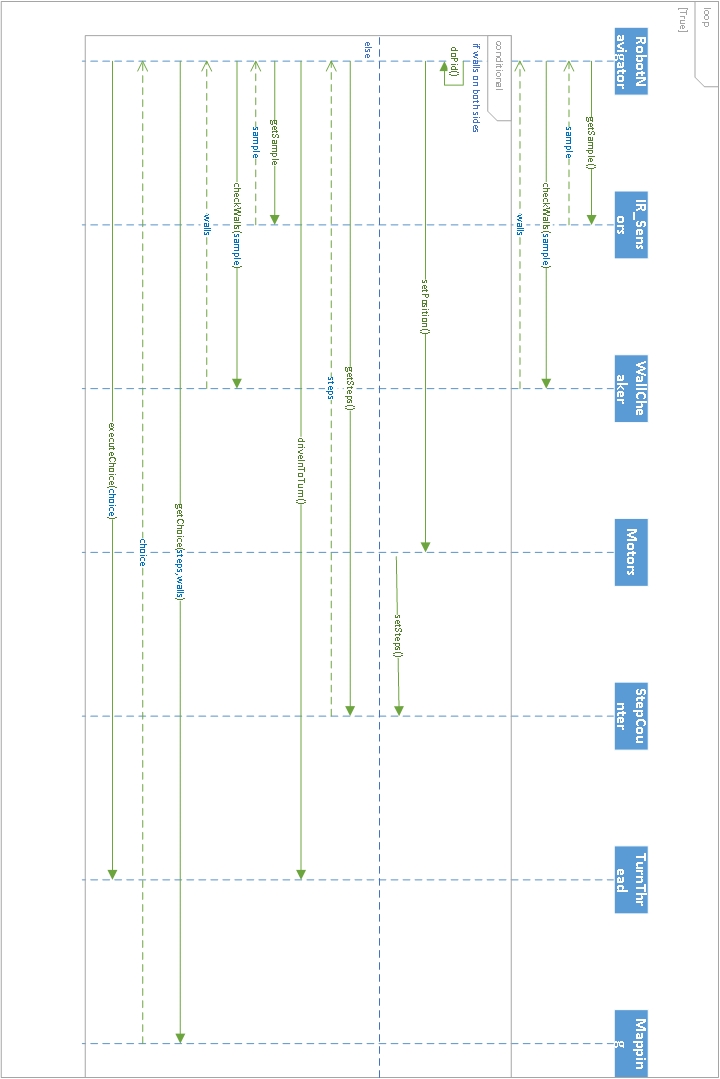
1. Mapping flowchart I dunno



1. Workflow diagram



3. Sequence diagram



8. Motor controller documentation and tests (MISSING!)
9. Navigationsystem tests

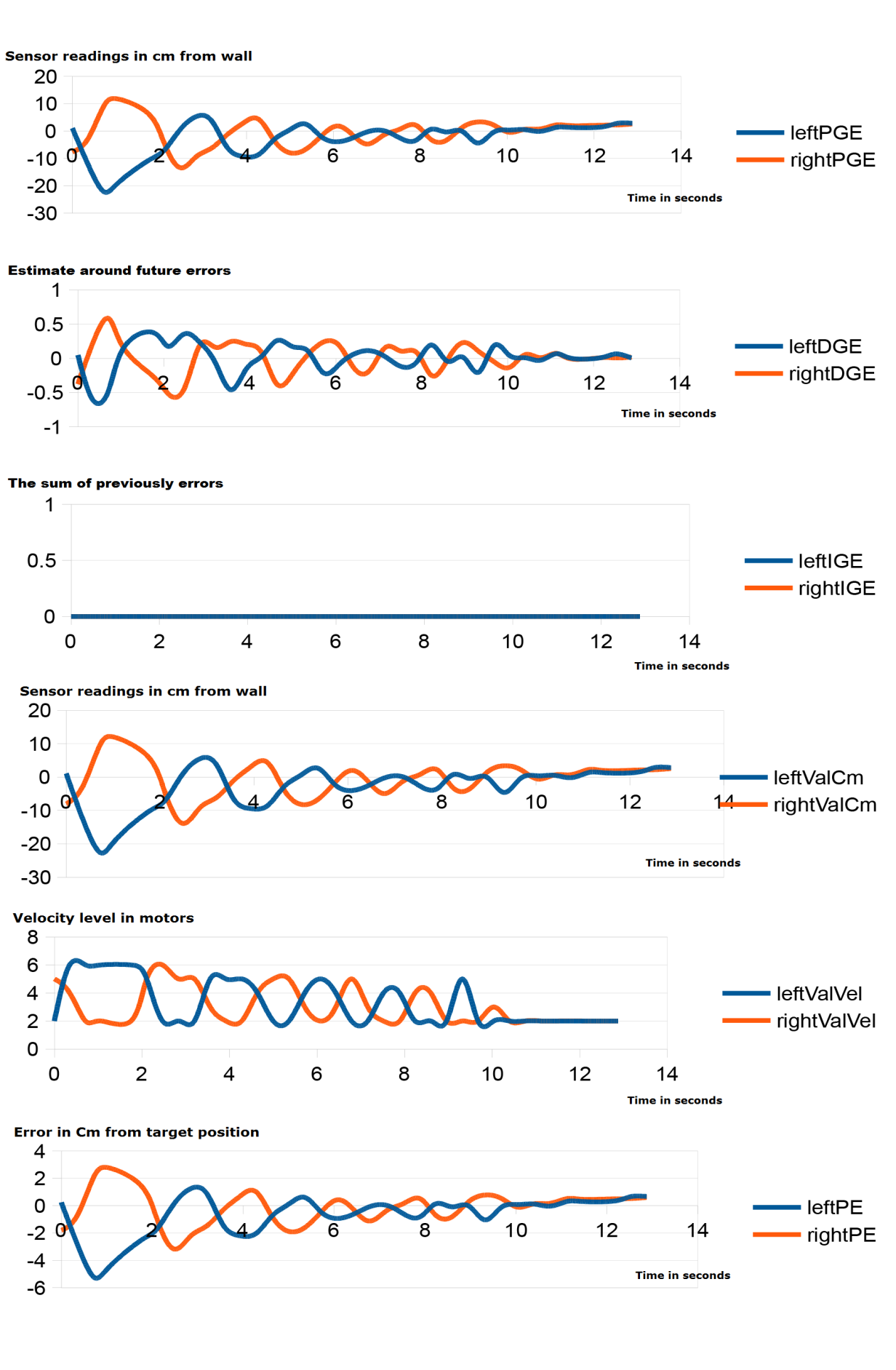
The following graphs shows the testing of the complete navigationsystem, where sensors, motors and PID work together.

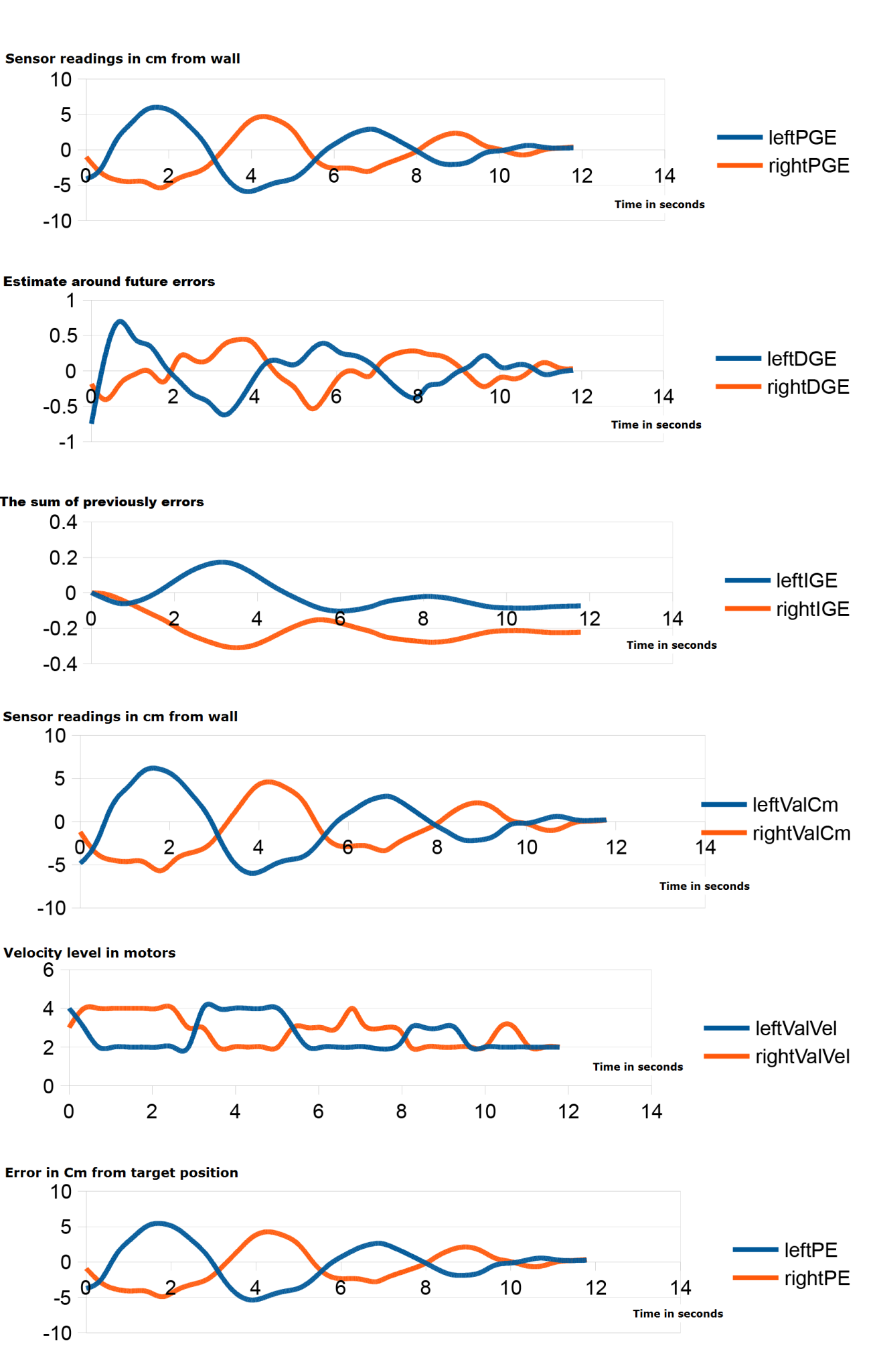
The most important graph, and the one we’re most interested in, is graph number 4, that shows the output of the PID feedback controller algorithm, in distances to the walls, given in centimeters.

The idea of PID is to create a damped curve from the output of the algorithm, where only a few overshoots happen. As a rule of thumb, we want as few overshoots as possible.

* The first set of graphs shows output with gain factors Kp=4,1 Ki=0 and Kd=0,2
* The second set of graphs shows the output with gain factors Kp=1 Ki=0 and Kd=0,3

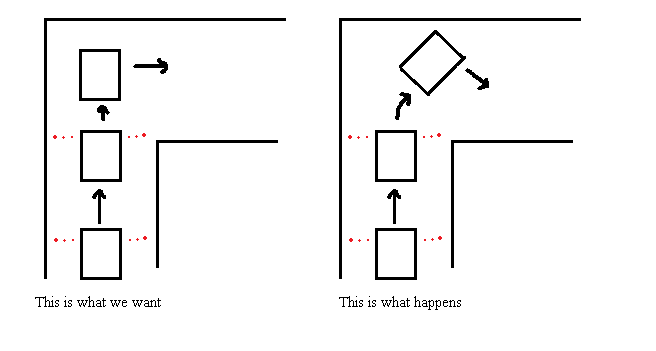
The second set of graphs shows overall better robot behavior, since the amplitude of themaximum overshoot is half (10) to the corresponding graph on the first set of graphs





1. PID turn90 error

Whenever we need to turn, we experience some unexpected behaviour from our robot. The robot increases its velocity rapidly, before realizing that it should initiate a 90 degree turn. We have spent lots of hours to figure out what’s wrong, and have finally been able to pinpoint the error.



There are several things that causes the above to happen, and there are several fixes to it – however some fixes will have an impact on other functionalities on the robot, and we don’t want this to happen. In fact, the above happens as a result of a variable change, that fixed another issue – a classical example of one fix that causes another issue.

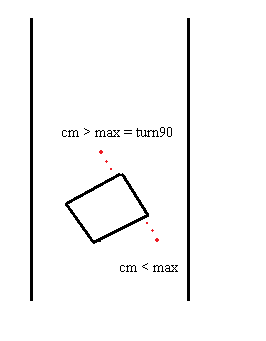
* The left picture shows that the robot detect a missing wall (right distance above threshold = 28cm), and initiates a turn sequence.
* The right picture shows that the robot doesn’t detect a missing wall (right distance below threshold = 28cm), and take it as a sign of being way out of the course, hence increasing the velocity on the left wheel.

The problem is that we take 3 samples before calculating the average measurement, which can cause the following scenario to happen:

1. Measurement 1: Right sensor distance = 15 cm.
2. Measurement 2: Right sensor distance = 15 cm.
3. Measurement 3: Right sensor distance = 50 cm.

* Measurement average for right sensor is then equal to 26,66 cm. This value is below the threshold of 28 cm, and as a result it thinks theres still a wall on the right side, hence doing PID instead of turning. A value of 26,66 cm will result in increasing the left wheel velocity to the maxmium velocity in its category, which will bring the robot out of course

An obvious solution would be to decrease the threshold – but this will just create another issue. The threshold needs to be high, or else the robot will do false 90 degree turns:

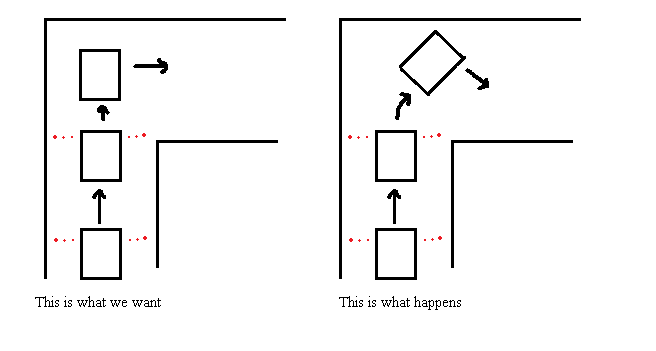


In fact the threshold of 28 cm was set exactly to prevent that.

The solution will be to constrain the maximum value that a three measurement sample can take or make a turn90 depend on another separate threshold, to keep it separate from the above scenario.

1. PID turn90 error revision 2

This paper is a follow-up on the problem reported last week concerning turning errors. Since last week, we have tried several things to solve the problem, where our robot would do PID correction instead of turning 90 degrees.



We described that taking an average reading of 3 inputs of a sensor, could cause a problem when a wall suddenly was missing. If a series of reading was 15, 15, 50, that would create an average measurement of 26,6. As the right figure above shows, this measurement would result in doing PID correction with a very high velocity, instead of initiating a turn.

It turns out that this is not entirely true. In fact, the real problem is that the robot doesn’t drive fast enough for the sensors to instantly detect the missing wall. Ideally we would like the sensor reading to be 65 cm (max distance) as soon as a wall is missing. However, this is not possible – at least not with the velocity that we use. What happens is that it takes a couple of readings before the sensor input reflects the state of the physical maze, when transitioning from wall to no wall.

This basically means that we receive 3-5 readings that gradually grow towards the maximum distance of 65 cm. If we use these readings we will always enter a turn, out of our course, due to PID correction of the robot.

The figure on the next page shows a series of readings, right before- and right after a missing wall has been detected.

Looking at the above figure, it’s obvious that we need to take some precaution when we enter a turn. The scenario can easily be compared to that of a button being pushed down without a de-bouncer – mechanical noise will create false output, similarly the solution is to “de-bounce” the sensor input before accepting it as a valid reading.

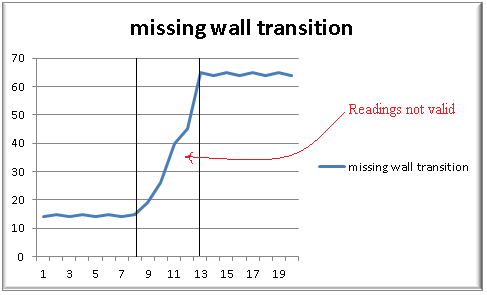
**This flow chart shows what we did before:**



**This flow chart shows the de-bouncing of the input:**



The reading is now based on a ”lastSample – newSample” comparison, where the initial last sample is set to the setPoint of the maze and “n” is some arbitrary value that indicates that a measurement is too different from the previous one for the robot to be driving in a corridor



**Conclusion:**

The new de-bounced version of our reading function works very well. Depending on the arbitrary value of n, the robot will do little to no PID correction on turns – which is what we wanted to achieve.

1. Mice odometry documentation

The idea of using mice odometry to estimate position is an idea we have had since the beginning of the project. We use two mice to input relative coordinate changes to our RPi whenever a displacement of the robot has happened. Several calculations are then done based on the given x,y coordinates – ultimately an angle and length of the movement is calculated. These two values are then used by a PID controller calculation algorithm to attempt to minimize heading errors.

Problem breakdown

 are left mouse input

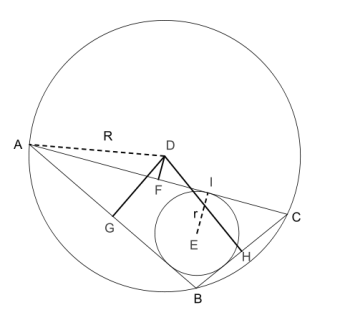
 are right mouse input

  is the angle between the intern x-axis and the tangent of the arc that the mouse traverses.

**** is the length of the arc that the mouse traverses

 , 

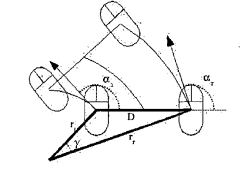
Above values can be calculated from our mice input.

Carnot’s theorem says that: in triangle ABC, the sum of the distances from center D to the middle of the trianglesides, are equal to the sum of the two circles radii.

DF + DG + DH = R + r,\ 

Equivalent equations in our case: 

The angle y can be calculated. The radii cant – but equations are given by:

,  og  (y can be calculated)

Above equations are substituted into Carnots theorem:

 og giver:



There are no unknown variables in this equations, so delta angle of the robot can be calculated!

If  er equal, the angle will always be 0.

Proportional Integral Derivative

Succesfully calculating our current heading (angle), we use this value as our process variable in a PID controlling algorithm, to make sure we stay on the desired heading. A change in  means our robot is slightly out of course, and the value will be saved as P (current error). The values of I (integral) and D derivative) will also be calculated based on the current error.

Having calculated all three values, these are combined with three preset gain terms – the proportional gain, the integral gain and the derivative gain – to get an output value. This way the PID controller keeps track of current error, accumulated errors, and estimated future errors.

previous\_error = 0

integral = 0

start:

error = setpoint - measured\_value

integral = integral + error\*dt

derivative = (error - previous\_error)/dt

output = Kp\*error + Ki\*integral + Kd\*derivative

previous\_error = error

wait(dt)

goto start[[15]](#footnote-15)

Calibrating the mice

Implementation

Being that our RPi runs with an embedded linux OS, creating an input event to detect mouse movement is rather simple. We have a simple Python module, that reads input from two mice in a loop, creates a 2x2 matrix and pass it to another module that then calculates the internal angle displacement on the two mice.

devices = map(InputDevice,(“/dev/input/event4”,”/dev/input/event5”))

def run(self)

while True:

r,w,x = select(devices, [], [])

for fd in r:

for event in devices[fd].read()

input = event

Calculations.calculateAngle(input)

The calculateAngle() will use the equations explained earlier, to calculate angles and length. This is done tens- or hundreds of times each second, depending on the speed of the robot. This is a lot of calculations that have to be done fast, in order for us to make the robot to keep its current heading.

We were aware early in the process, that certain algebraic operations, such as sin, cos, tan are slow and demanding, so we made a LookUpTable to store all possible lengths and angles of a single mouse input. In linux, mouse event input is stored in a single signed byte. A 255x255 python List stores a python Tuple, containing angle and length of a give input, in each entry. Whenever a mouse event is detected, the (x,y)-pair is used to make a TableLookUp to get the associated angle and length.The table is created upon startup and stored in memory. Hopefully, this approach should safe us small amount of time:

**Speed test using the python module timeit:**

calcAngLen = timeit(calcAngLen(25,25))

createTable = timeit(initAngLenTable)

getAngLen = timeit(getAngLen(25,25)

0.0040 s

2.9353 s

1.0967\*

The first line of code calculates angle and length based on (25,25)-coordinates

The second line of code creates a 255x255 “List” containing (angle,length)-pairs

The third line of code makes a TableLookUp and fetches the angle and length

The test shows that a considerable amount of time can be saved by creating the table in advance, and then perform lookups whenever a mouse event occurs.

1. IR-sensors documentation and tests

The robot must be able to determine distances to the surrounding walls, in order to maintain a heading and know when to make a turn. Two primary options are given in this context; Infra-red sensors or Ultrasonic-sensors (or a combination of both). Either choice will work on a robot, but have different impacts on how the robot will function when running a maze. We have to look at the two sensor specifications and the robot behavior requirements, to make a satisfying engineerical choice.

**IR Sensor**

The infra-red sensor (which in our case is a GPD120x Sharp sensor) uses triangulation and CCD array to calculate a distance to objects. It works by emitting pulses of infra-red light, which in case of a present object, will be reflected and caught by the detector. This will create a triangle between the emitter, detector and point of reflection. The distance of the reflecting object will determine the angle in this triangle, which is then used by the CCD array to calculate a distance to the object.

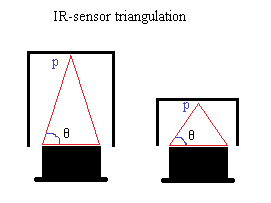


Figure 1: Shows the IR-sensor triangulation, where ‘p’ is the point of reflection and ‘theta’ is the angle created

The sensor output is non-linear with respect to the distance, so it is not insignificant where the sensors are mounted on the robot. Based on the graph of the analog voltage as a function of the distance, one will have to make decision of an optimal reading interval.

**Ultrasonic sensor**

The ultrasonic sensor works differently from the IR-sensor, by emitting inaudible sound to detect surrounding objects, instead of light. We know that sound travels at a certain speed in air, so by keeping track of time elapsed since the sound was emitted and until the echo is detected, a very accurate distance can be calculated.

The sound emitted by the ultrasonic sensors spread out radially, which can cause problems, if several ultrasonic sensors are used, and not placed properly on the robot. If the sensors are pinging too close to each other, one sensor may read another sensors echo as its own. Furthermore materials and object angles might cause other problems such as lost echoes, or ghost echoes.

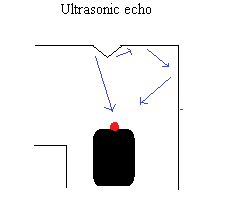


Figure 2: Shows ultrasonic sensor echoes. One echo reflects fine on the wall, the other turns into a ghost echo as a result of the angle on the wall

**IR- vs Ultrasonic sensor**

Both sensors are fine at detecting the distance from the robot to an object. Our main argument of choice is when mounting multiple sensors on the robot, comes into play. We need two sensors beaming on each side on the robot, as well as one beaming in a frontal direction. Hence, our choice of using only IR-sensors, are based on the fact that they have a narrow beam width, which (hopefully) won’t interfere with each other. We’re concerned that by using Ultrasonic sensors, the echoes will mess up the reading on the other sensors – especially because we want to maintain a continuous robot movement, where we need to ping, basically all the time, to determine the distances to the wall and maintain a heading.

**Implementation**

Reading from the IR-sensors is relatively simple. In Python, we use the System Management Bus (SMBus) to communicate via I2C with the DA7997 chip. The DA7997 chip is an 8-channel, 12-bit ADC ,that converts the analog sharp sensor output, to digital. The converted output can be read directly from a certain register within the chip, with a few commands:

SMBus.write\_byte(slaveAddress, (0x00 | 0x08 << 4))

SMBus.read\_i2c\_block\_data(slaveAddress)

The first line tells the DA7997 chips “pointer register” to point at the 0x00 register, and convert from channel 0x08.

The second line tells the DA7997 chip that we want to read the content of whatever register is pointed to

This seem very simple – and it is. The procedure when reading the sensor output is always the same:

* Make sure the pointer register points to the register you want to read/write to (4 LSBS’s)
* Make sure the pointer register contains the channel you want to convert from (4 MSB’s)
* Read data

**Reading from multiple sensors**

Reading from multiple sensors, can be done in two ways. Either in a loop, where you loop through n sensors (max 8), tell them to start converting and read the result of the “conversionResultRegister”, or by setting up the “ConfigurationRegister”. As the name suggests, the Configuration register contains some bits for selecting the operation mode, but also reserves a byte for reading a sequence of channels. This way, the chip switches between the channels by itself, and simplifies the programming code.

Ideally, we would like to use the automatic swithing solution, but we encountered some problems when trying to code it. We got wrong output from the sensors, and not all three sensors were read from. We switched to reading in a loop instead - that worked perfectly, and we will stick to this for a while. However, we would like to compare the two solutions through tests, in the upcoming weeks, to determine the best solution.

**Using the converted analog output**

When a reading command is done, we receive 2 bytes, where the 12 LSB’s are measurement data

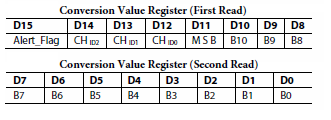
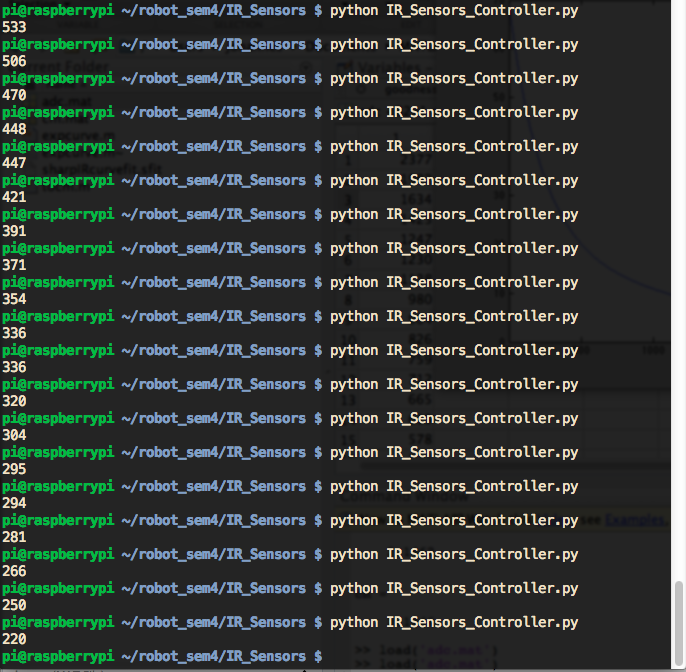
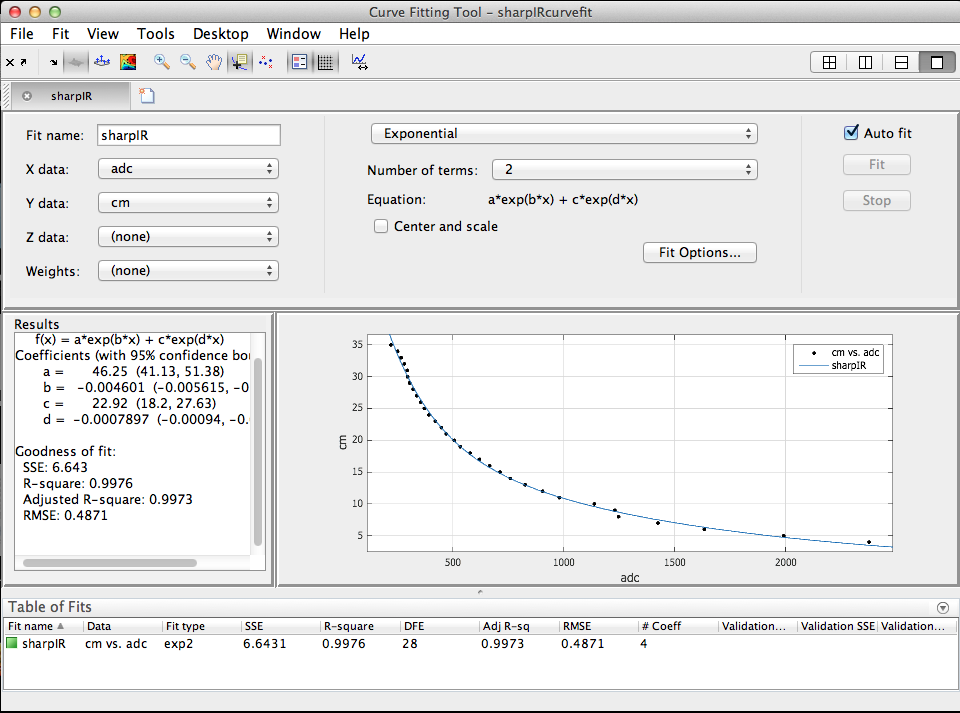


Figure 3: Conversion register

This gives us a maximum decimal value of 4095. This value is gonna be converted to a distance in centimeters. We did this by making 100 sample measurements with a 200 ms delay, for each distance in a range from 3-35 cm, to some object. Each measurement was divided by 100 to get the average measurement.



This data was entered in Excel and then imported into Matlab (CFTool) to get an approximated graph, and the equation of this graph.



The equation that describes the graph is given by:

a =   46.25

b =   -0.004601

c =   22.92

d =  -0.0007897

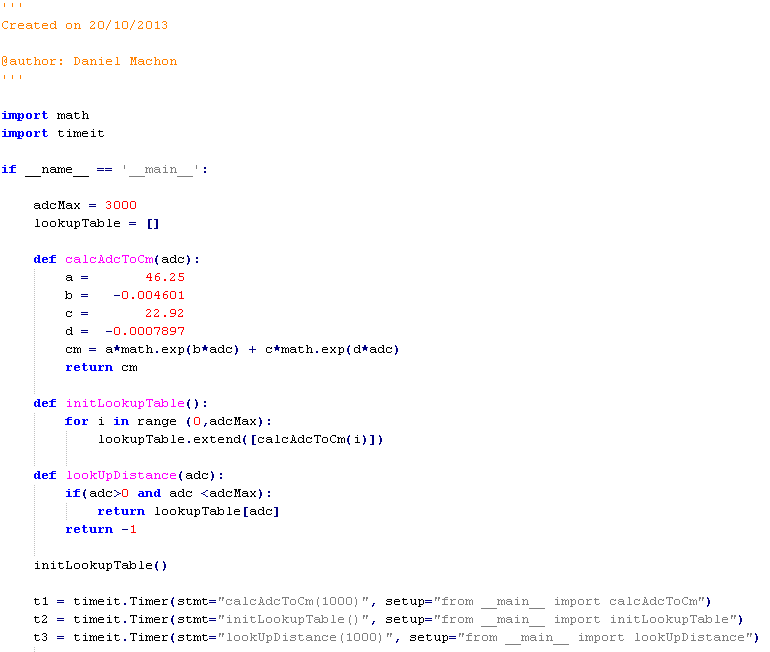
Distance(cm) = a\*exp(b\*adc) + c\*exp(d\*adc)

Whenever receiving data from sensors, the distance from that sensor to a wall, can be calculated from entering that data into the above equation.

**Calculating at runtime vs. LookupTable**

Sometimes it can be expensive to make calculations at runtime – especially when using Python, that are not that efficient at number-crunching. We already learned, from our mice odometry code, that making a lookup-table that indexes operations, can sometime save a lot of time. Similarly we want to find out if the same applies to our equation that describes the graph on the previous page.

For this purpose we use the Python Timeit module, that measures the CPU time used for a given method.



T1: Use the calcAdcToCm() method to convert the value 1000 to a distance in centimeters

T2: Create a Lookup-table of converted values from 0 to 3000

T3: Use the Lookup-table to look up the converted value of decimal 1000

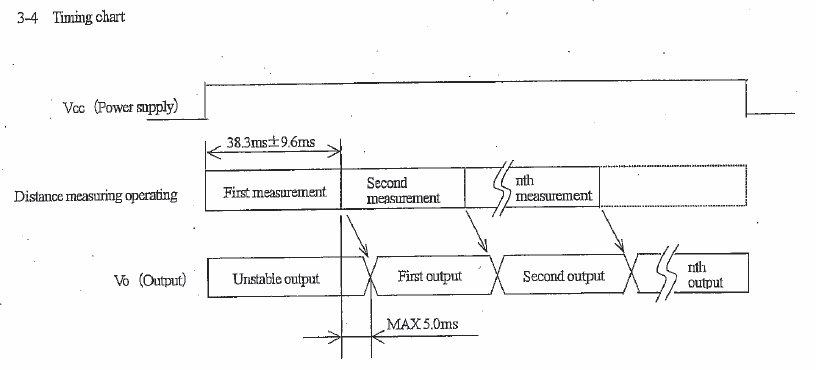
The results of the timeit tests are as follows:

|  |  |
| --- | --- |
| calcAdcToCm(1000) | s |
| initLookupTable() | s |
| lookUpDistance(1000) | s |

The test shows that making a lookup instead of calculating at runtime, is slightly faster. We actually expected it to be MUCH faster to make a lookup, but it seems as Python is fairly efficient in this case. In our **mice odometry lookup test,** where we did the same test, only with sin, cos and arctan calculation, it was more than 400 times faster to make a lookup instead of making a calculation.

Despite only giving a slight time usage decrease, we’re gonna stick with the lookup-table. We’re aware that this will cost us a 36 ms. creation overhead time and a small amount of memory.

**Sensor sampling rate**

****

**Lidt om hvor mange beregninger vi laver før vi tager gennemsnittet. Vi er begrænset at tiden da det tager omkring 40ms per måling.**

1. RPi-TMC222 baud rate Documentation

In order to maintain a continuous movement with the robot, we need to make sure the actual position of the robot, never reaches the target position. This is accomplished by reading certain registers in the TMC222 chip. Our getFullStatus2() method provides us with this information, in the form of:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address | ActPos. | TagPos. | SecPos. | SecPos. | NA |

During our testing of getting this data, we have discovered huge variations of the information we receive, as well as complete useless return data. We suspected that this could be the result of wrong/inefficient baud rate of the I2C bus.

We have conducted three tests that show 40 status readings of the TMC222 chip, while the robot moves towards a target position. In each reading, the I2C baud rate is set differently. Also in each reading, the target position is set to 32000 micro-steps, followed by 25 intervals of 2500 micro-steps. Readings are done in a 1 second interval.

**First test: Baud rate = 100000**

**Expectation: We would expect ActPos to increase from 0 to 32000, and then increasing by 2500 until it overflows the TagPos limit of  and starts over. Also Address, SecPost, and NA should always stay the same (0xE0, 0x00, 0xFC and 0xFF).**

**Results:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address | ActPos. | TagPos. | SecPos. | SecPos. | NA |

0xe0 32896 64896 0x80 0xfe 0xff

0xe0 34996 48895 0xff 0xff 0xff

0xe0 37859 64896 0x80 0xfc 0xff

0xe0 41094 64896 0x80 0xfc 0xff

0xe0 44453 64896 0x80 0xff 0xff

0xe0 48318 64896 0x80 0xff 0xff

0xe0 50565 64896 0x80 0xfc 0xff

0xe0 53171 64896 0x80 0xfe 0xff

0xe0 56534 64896 0xff 0xff 0xff

0xe0 59893 64896 0x80 0xfc 0xff

0xe0 63630 64896 0xff 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 64896 64896 0x80 0xfc 0xff

0xe0 64896 64896 0x80 0xfc 0xff

0xe0 64896 64896 0x80 0xfe 0xff

0xe0 64896 64896 0x80 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 37000 37000 0x80 0xfe 0xff

0xe0 39628 39628 0x80 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 44500 44500 0x80 0xfc 0xff

0xe0 47000 56319 0xff 0xff 0xff

0xe0 49628 49628 0x80 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 54500 54500 0x80 0xfc 0xff

0xe0 57000 57000 0x80 0xff 0xff

0xe0 59628 59628 0x80 0xfe 0xff

0xe0 62128 62128 0x80 0xfc 0xff

0xe0 64500 64500 0x80 0xfe 0xff

0xe0 34232 34268 0xff 0xff 0xff

0xe0 36860 36860 0x80 0xfc 0xff

0xe0 39360 39360 0x80 0xfc 0xff

0xe0 41860 41860 0x80 0xfe 0xff

0xe0 44232 38655 0xff 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 49360 49360 0x80 0xfc 0xff

0xe0 51860 51860 0x80 0xfe 0xff

0xe0 54232 43519 0xff 0xff 0xff

0xf0 65535 65535 0xff 0xff 0xff

0xe0 59360 59360 0x80 0xfe 0xff

**The test shows that all received data is complete rubbish and in no way useable to achieve a continuous movement.**

**Second test: Baud rate = 375000**

**Expectation: We would expect ActPos to increase from 0 to 32000, and then increasing by 2500 until it overflows the TagPos limit of  and starts over. Also Address, SecPost, and NA should always stay the same (0xE0, 0x00, 0xFC and 0xFF).**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Address | ActPos. | TagPos. | SecPos. | SecPos. | NA |

0xe0 0 32000 0x0 0xfc 0xff

0xe0 2222 32000 0x0 0xfc 0xff

0xe0 4951 32000 0x0 0xfc 0xff

0xe0 8179 32000 0x0 0xfc 0xff

0xe0 11656 32000 0x0 0xfc 0xff

0xe0 15384 32000 0x0 0xfc 0xff

0xe0 17622 32000 0x0 0xfc 0xff

0xe0 20350 32000 0x0 0xfc 0xff

0xe0 23578 32000 0x0 0xfc 0xff

0xe0 27056 32000 0x0 0xfc 0xff

0xe0 30783 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 32000 32000 0x0 0xfc 0xff

0xe0 34500 34500 0x0 0xfc 0xff

0xe0 37000 37000 0x0 0xfc 0xff

0xe0 39500 39500 0x0 0xfc 0xff

0xe0 42000 42000 0x0 0xfc 0xff

0xe0 44500 44500 0x0 0xfc 0xff

0xe0 47000 47000 0x0 0xfc 0xff

0xe0 49500 49500 0x0 0xfc 0xff

0xe0 52000 52000 0x0 0xfc 0xff

0xe0 54500 54500 0x0 0xfc 0xff

0xe0 57000 57000 0x0 0xfc 0xff

0xe0 59500 59500 0x0 0xfc 0xff

0xe0 62000 62000 0x0 0xfc 0xff

0xe0 64500 64500 0x0 0xfc 0xff **Overflow!**

0xe0 1464 1464 0x0 0xfc 0xff

0xe0 3964 3964 0x0 0xfc 0xff

0xe0 6464 6464 0x0 0xfc 0xff

0xe0 8964 8964 0x0 0xfc 0xff

0xe0 11464 11464 0x0 0xfc 0xff

0xe0 13964 13964 0x0 0xfc 0xff

0xe0 16464 16464 0x0 0xfc 0xff

0xe0 18964 18964 0x0 0xfc 0xff

0xe0 21464 21464 0x0 0xfc 0xff

0xe0 23964 23964 0x0 0xfc 0xff

0xe0 26464 26464 0x0 0xfc 0xff

**The test shows that with a baudrate of 375000, the robot behaves as expected, and returns correct information.**

In conclusion, we can derive that the serial communication between the robot and our raspberry pi, won’t work with a low baud rate. My idea is that with a low baud rate, we cannot read the data from the TMC222 registers fast enough, and as a result the register overflow and return corrupt information.

1. Mice odometry documentation and choice of discarding it, is enclosed in the appendices [↑](#footnote-ref-1)
2. The full baud rate documentation is enclosed in the appendices [↑](#footnote-ref-2)
3. Full documentation on PID gain factor tuning and graphs are enclosed in the appendices [↑](#footnote-ref-3)
4. A more adequate explanation of this problem, as well as its solution is enclosed in the appendices: **PID\_Turn90 error + solution** [↑](#footnote-ref-4)
5. The math behind the iterative approach is enclosed in the appendices [↑](#footnote-ref-5)
6. We still have the half-working branch of the iterative development code on http://www.github.com/sloev/robot\_sem4 [↑](#footnote-ref-6)
7. Appendix this and appendix that [↑](#footnote-ref-7)
8. Appendix x [↑](#footnote-ref-8)
9. The complete test of the IR-sensors is enclosed in the appendices [↑](#footnote-ref-9)
10. The complete test of the motors is enclosed in the appendices [↑](#footnote-ref-10)
11. Graphs based on PID output are enclosed in the appendices [↑](#footnote-ref-11)
12. The complete mice odometry documentation is enclosed in the appendices [↑](#footnote-ref-12)
13. Full voltage output documentation is enclosed in the appendices [↑](#footnote-ref-13)
14. <http://cs.stackexchange.com/questions/553/how-do-common-pathfinding-algorithms-compare-to-human-process> [↑](#footnote-ref-14)
15. <http://en.wikipedia.org/wiki/PID_controller> [↑](#footnote-ref-15)