# Zumo robot documentation

Metropolia University of Applied Sciences

Bachelor of Engineering

Information technology, Smart systems

Project documentation

15 December 2017

Metropolia

| | |
|---|---|
| Project participants<br>Title<br><br>Number of Pages<br>Date | Kristofer Knutsen, Michael Bell, Anh Phan<br>Zumo robot documentation<br><br>29 Pages<br>15 December 2017 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information technology |
| Course | Smart systems |
| Instructors | Joseph Hotchkiss<br>Keijo Länsikunnas<br>Timo Leinonen |

The goal for this project was to program a robot that can compete in a line following race. The project required competitors to create the code used for the robot in C language and apply it to the Zumo robot using PSoc creator. The project was undertaken by a group of 3 and given free rein to program the robot how we saw fit. This documentation will recount how finished code was reached, the separate branches the team members took and an evaluation of how we feel the end product performed.

| | |
|---|---|
| Keywords | If/Else, PSoC, robot, PID |

## Contents

List of Abbreviations

1 Introduction 1

2 Software and applications used for the project 1

2.1 PSoC Creator 4.1 1
2.2 PSoC Programmer 3.26.0 2
2.3 PuTTY 3
2.4 Zotero 4

3 Race rules and requirements 5

3.1 The line-following race 5
3.1.1 The robot must complete the track 5
3.1.2 The robot must drive to and stop at the start line and await the infrared signal to begin the race 6
3.1.3 The robot must stop on the second finish line and remain there 6
3.1.4 The robot must follow the line using its infrared sensors only 6
3.2 The zumo battle 7

4 The Zumo robot 8

5 The code 9

5.1 The C programming language 9
5.2 Sensor reading 9
5.3 If / Else code 10
5.3.1 Code function 10
5.3.2 Code conditions 10
5.3.3 Turns 10
5.3.4 Stop/Break 12
5.3.5 Forward 13
5.3.6 Loops 13
5.4 The Zumo wrestling code 13

6 The PD controller 15

6.1 PD controllers 15

6.1.1 P controller 15
6.1.2 Challenges of the P Controller 17
6.1.3 The PD Controller 18
6.1.4 The challenges of a PD controller 20
6.1.5 PD Controller Conclusion 20

7 Conclusion 21

7.1 Race day, line following race 21
7.2 Race day, zumo wrestling 21

8 Final words 22

References 23

## List of Abbreviations

| | |
|---|---|
| **PSoC** | Programmable system-on-chip |
| **SoC** | System-on-chip |
| **IDE** | Integrated Design Environment |
| **Gui** | Graphical User Interface |
| **Repo** | Repository |
| **Git** | GitHub |
| **Cap** | Capacitor |
| **Res** | Resistance |
| **P-control** | Proportional control |
| **SSH** | Secure shell |
| **Telnet** | Protocol to provide a interactive text-oriented communication facility using a virtual terminal connection |
| **FM0+** | Portfolio of energy efficient flexible microcontrollers |
| **USB** | Universal serial bus |
| **PID** | Proportional Integral Derivative |

# 1 Introduction

This report will cover the steps that were used to get from receiving the robot to finishing a line-following race and zumo wrestling competition. We will start by covering what applications were used for this project, race goals and requirements, code used for each project, PID controller, and conclusion.

# 2 Software and applications used for the project

## 2.1 PSoC Creator 4.1

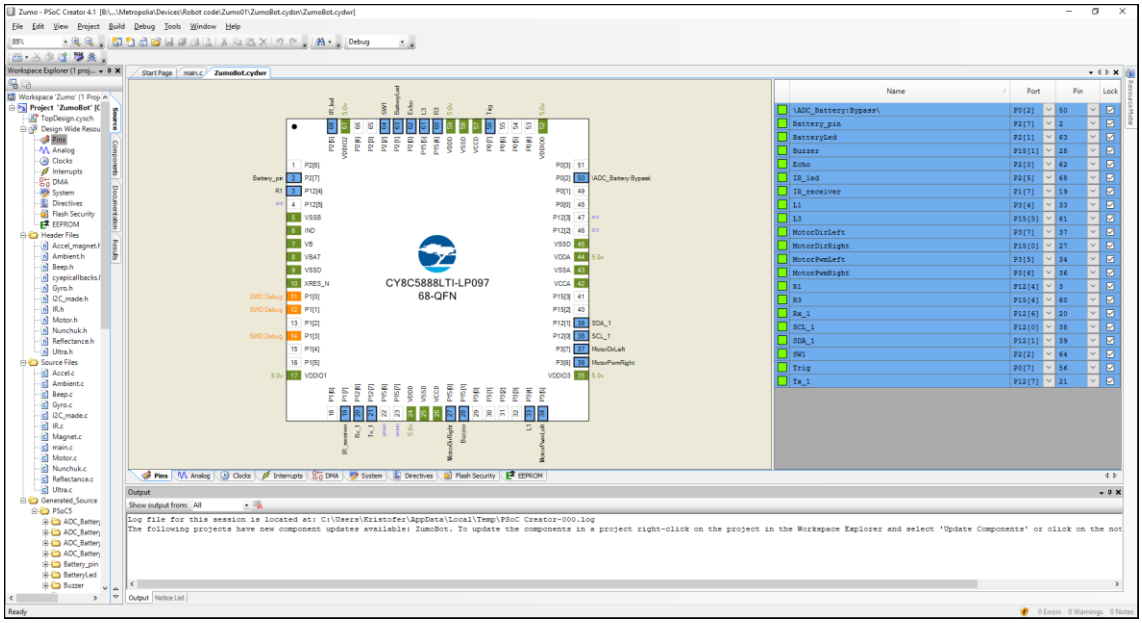



Figure 1. PSoC Creator software

> **PSoC Creator** is an Integrated Design Environment (IDE) that enables concurrent hardware and firmware editing, compiling and debugging of PSoC systems. Applications are created using schematic capture and over 150 pre-verified, production-ready peripheral Components. (1)

The **PSoC Creator** was used as the compiler for the code for the robot as well as the main coding environment.

## 2.2 PSoC Programmer 3.26.0

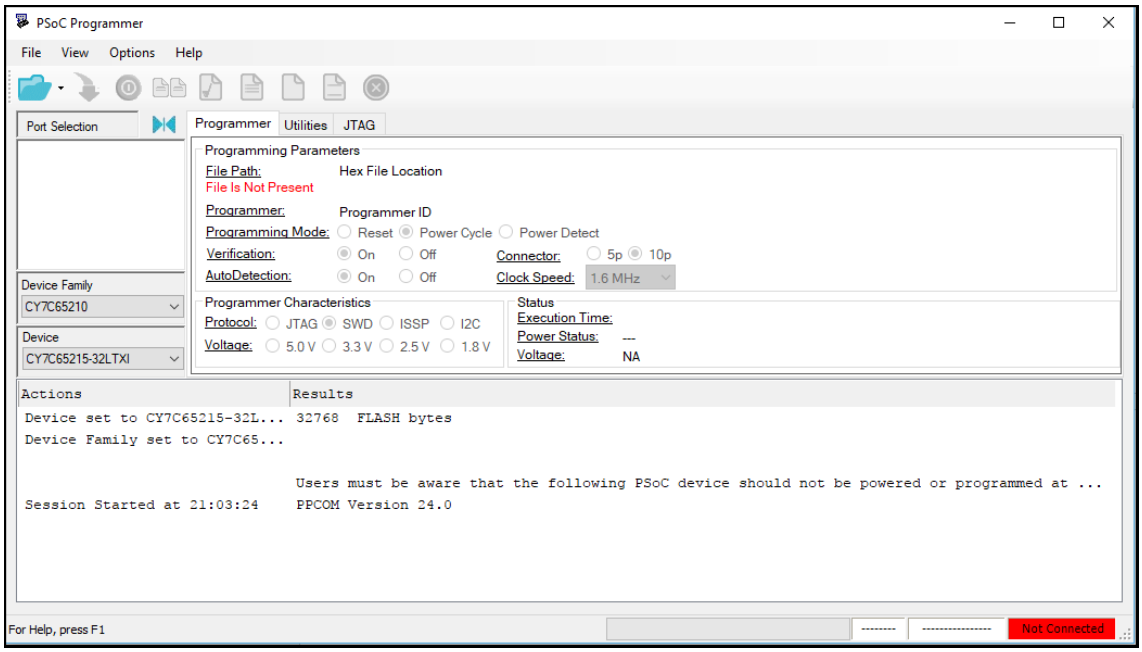



Figure 2. PSoC Programmer

> **PSoC Programmer** is a simple GUI that enables you to program Cypress's programmable devices. This is a single tool which supports the programming of Cypress's MCUs including PSoC® and FM0+ devices, and USB Type-C and Power Delivery devices. PSoC Programmer also provides integrated support in both the PSoC Creator and PSoC Designer IDEs. (2)

The PSoC Programmer was only used to update the firmware of the PSoC, if the firmware on the chip was outdated. Turned out the project chip needed update so the programmer was indeed used.

## 2.3 PuTTY

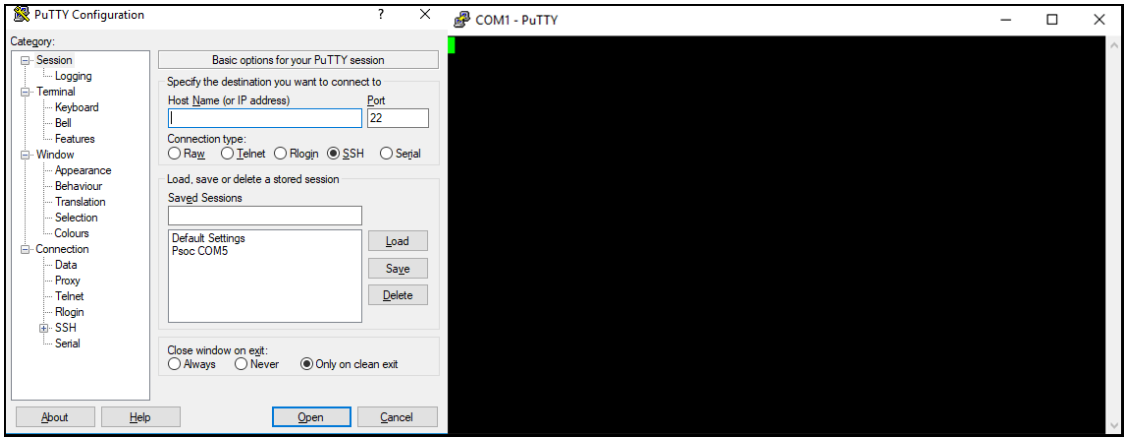



Figure 3. PuTTY configuration and terminal

> PuTTY is a free implementation of SSH and Telnet for Windows and UNIX platforms, along with a terminal emulator.

PuTTY was used to get visual representation from the sensors from the robot. The tool was connected to the robot via USB and the terminal screen would display the information that was printed, by the printf command in the code that was made in PSoC Creator

## 2.4 Zotero

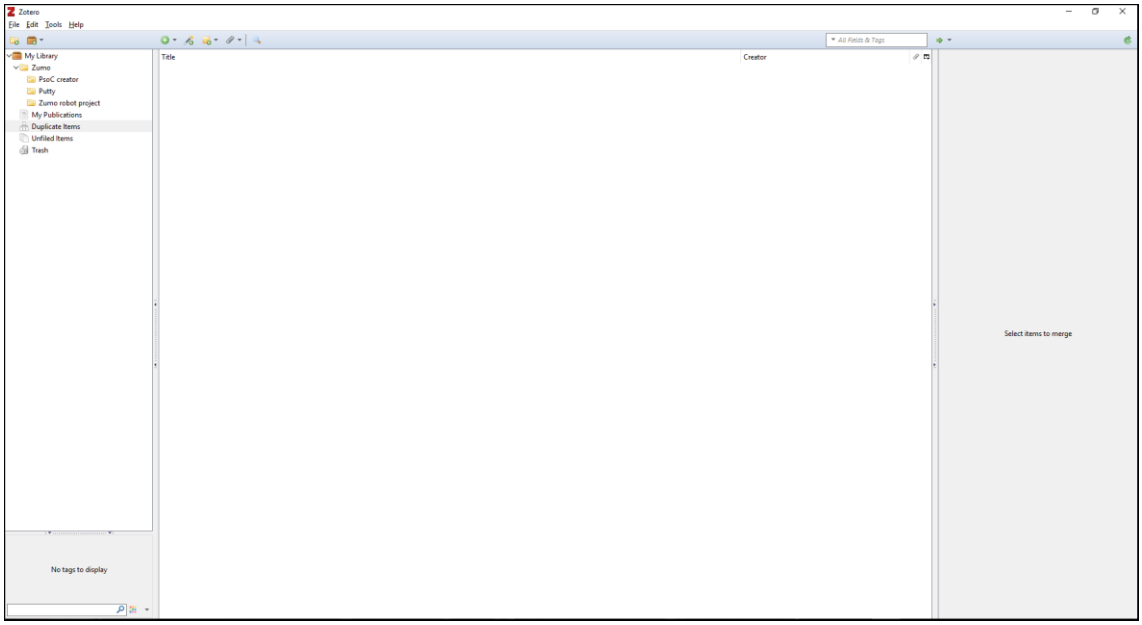
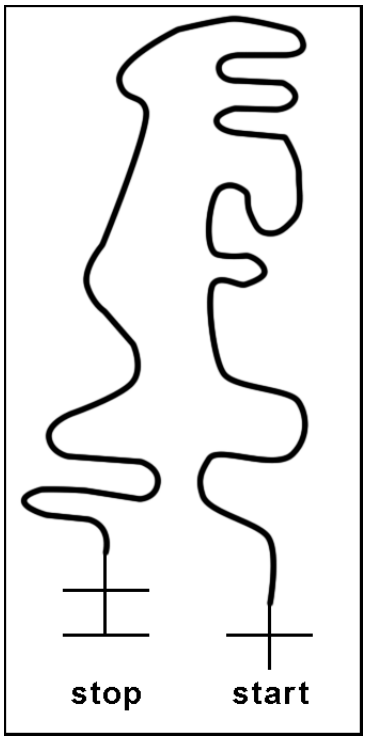



Figure 4. Zotero desktop application

Zotero is a fantastic research tool that allows you to save any website or document with one click and save it into a library. It has a browser extension which has the one-click feature and then it is saved to your Zotero desktop application. It can also be linked to multiple computers so working on different workstations is no problem. After all things are saved it can compile all your links and documentation and save it to a bibliography format for your report in alphabetical order, so it is very helpful for all research work.

# 3 Race rules and requirements

## 3.1 The line-following race




The task was to create a robot code that could allow it to compete in a line racing competition, not only finishing the track but also by following the set rules in order to avoid disqualification. These rules are outlined below:

- The robot must complete the track from start to finish
- The robot must drive to and stop at the start line and await the infrared signal to begin the race
- The robot must stop on the second finish line and remain there

Figure 5. Line-following track

- The robot must follow the line using its infrared sensors only
- The robot must always stay on the main line, if he loses the line it is automatically disqualified

These requirements created the parameters for the code and as such will be explained in more detail to provide a deeper understanding of the challenges faced in creating the code.

### 3.1.1 The robot must complete the track

The first objective set was to get the robot to finish the track in its entirety. Should the robot fail to remain on, or return to, the track then it becomes disqualified and loses the race automatically.

### 3.1.2 The robot must drive to and stop at the start line and await the infrared signal to begin the race

The race begins with the robot driving to the set start point and awaiting the signal to begin racing. Should the robot fail to stop at the line or fail to start after the signal has been sent, it becomes an automatic fail.

### 3.1.3 The robot must stop on the second finish line and remain there

At the end of the track there are two lines parallel to each other and perpendicular to the track. The robot is required to pass the first of these lines and then stop when it reaches the second one. If it stops at the first or drives over the second line, then the robot fails the race.

### 3.1.4 The robot must follow the line using its infrared sensors only

Given the secrecy of the final version of the track this rule was the simplest to adhere to as only a robot capable of functioning on its own accord would be able to successfully complete the track. This requires the robot to take input from the infrared sensors and use the data to steer itself quickly and accurately.

## 3.2 The zumo battle

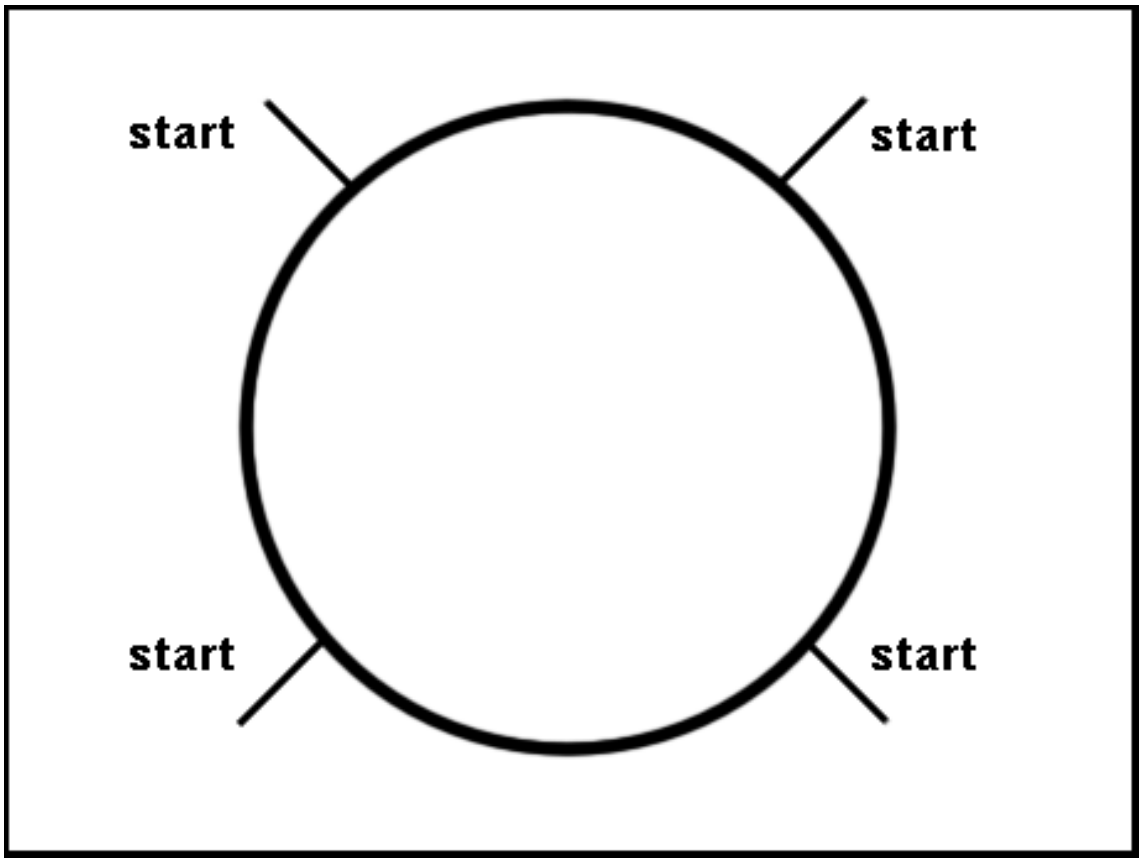



Figure 6. Zumo battle track

The zumo battle was quite different from the line following race. This was purely made for entertainment purposes. The goal here was to be the last robot standing in the ring. Four robots compete each time against each other. The robot is placed on the start line and it starts to drive automatically to the ring edge and stop there. Then a remote is pressed and all robots go in at the same time. The objective was to push other robots out of the ring.

# 4 The Zumo robot

The robot provided for the task is a Zumo brand robot that has been merged with a Cypress microcontroller on the chassis which allows it to receive commands through PSoC creator and programmer programs on our laptops. The robot has two motors, two treads and six infrared sensors. The left motor controls the left tread by moving it forwards or backwards and the right motor does the same for its side. Moving the left tread forward at a greater speed than the right allows the robot to make a right turn and vice versa. While six sensors were available, l1,2,3 and r1,2,3, the only ones required were the innermost (l1, r1) and outermost (l3, r3) and as such the others were left unused throughout the programming.

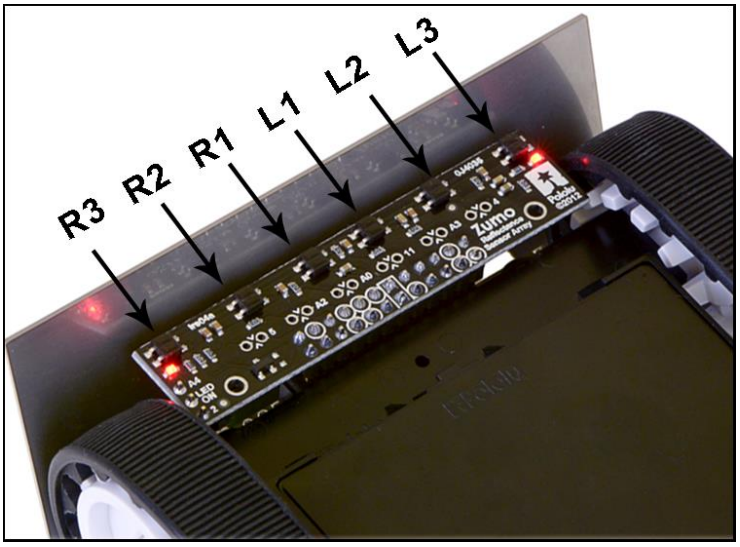



Figure 7. Zumo robot IR sensor array

# 5 The code

## 5.1 The C programming language

The use of C language was one of the requirements for completing the project. C is one of the most commonly used programming languages and is also one of the earliest that is still in use today. There were several benefits in using C to create the project's code. The portability of the language, meaning it can be compiled on any platform, the flexibility, it can be used in a very wide range of things, and its reliability (its longevity stands as testament to this). C is also a precursor and influencer of several other notable programming languages, including java and C++, which means a fundamental understanding of C affords a degree of insight into the languages that followed it.

## 5.2 Sensor reading

The sensors are the tools that was needed to steer the robot left and right. The IR sensor is based on two components, the IR transmitter and IR receiver. The transmitter sends an infrared light down to the surface and the receiver measures the reflectance of the obstacle material. In this case the black values of the line and the white paper the line is on.

The blackest value the sensor detects is 23999 and that is full black (when the sensor is directly in centre of the black line). That is true for all sensors.

The white values are different for each sensor and has to be calibrated for each sensor, but they varied from 4400-6400 depending on the sensor.

## 5.3 If / Else code

### 5.3.1 Code function

The If/Else code was based on pre-written conditions that might occur during the robot's manoeuvre throughout the course, using the sensor readings. It was created by trial and error and by calibrating and adjusting the black/white threshold.

### 5.3.2 Code conditions

The conditions for the main loop were simple and easy to implement but hard to calibrate. The code had 6 turn conditions (3 for left side turns and 3 for right side turns) 1 forward condition and a stopping condition

### 5.3.3 Turns

```
//Mega hard left
else if (ref.l3 > BLACKL3 && ref.l1 > WHITEL1 && ref.r1 > WHITER1 &&
ref.r3 > WHITER3)
            {
             tankturnleft(TANKTURNSPEED,HARD_TURN_TIME);
            }
```

This is the first turn condition.

BLACKL3 = The black set value for L3 sensor

WHITEL1 = The white set value for L1 sensor

WHITER1 = The white set value for R1 sensor

WHITER3 = The white set value for R3 sensor

This translates to if the left-most sensor sees black and rest sees white (robot is losing the track) it will take a hard turn to the left. The tankturnleft function is a coded function that says that when an input to the function is made it will turn the left wheel backwards and right wheel forwards, making a tank turn

TANKTURNSPEED is a definition for the speed input for the motors for this function, in this case it is set to 255 which is max speed.

HARD_TURN_TIME is a time command, or a delay in milliseconds, how long the speed pulse is set to the motor. In this case it was set to 65 milliseconds.

```
//hard left
else if (ref.l3 > BLACKL3 && ref.l1 > BLACKL1 && ref.r1 > BLACKR1 &&
ref.r3 < WHITER3)
            {
             tankturnleft(TANKTURNSPEED,MOVETIME);
            }
```

This is the second turn condition.

BLACKL3 = The black set value for L3 sensor

BLACKL1 = The black set value for L1 sensor

BLACKR1 = The black set value for R1 sensor

WHITER3 = The white set value for R3 sensor

This translates to if the sensors sense a sharp left turn (the first 3 sensors are black and right-most is white) it will take a sharp left turn

MOVETIME is a time command, or a delay in milliseconds, how long the speed pulse is set to the motor. In this case it was set to 1 milliseconds.

```
//left turn
else if (ref.l1 > BLACKL1 && ref.r1 < BLACKR1)
            {
             motor_turn(OTHERTURNSPEED,TURNSPEED,MOVETIME);
            }
```

This is the third turn condition.

BLACKL1 = The black set value for L1 sensor

BLACKR1 = The black set value for R1 sensor

This translates to if the left middle sensor sees greater than the set black value but the right middle sensors sees less than set black value it will turn to the right. So just as the robot is about to lose the line it will turn a little bit.

Motor_turn function was a predefined function in the robot library. It has the input for the left motor power, right motor power, and time delay.

OTHERTURNSPEED was a defined power number for the robot, here set to 0

TURNSPEED was a defined power number for the robot, here set to 255

Those 3 turn functions were mirrored to suit the right turns also. Everything was mirrored and all the power and delay values remained the same for left and right turns.

### 5.3.4 Stop/Break

```
//Final stop loop
if (ref.l3 > BLACKL32 && ref.l1 > BLACKL12 && ref.r1 > BLACKR12 &&
ref.r3 > BLACKR32)
            {
             motor_stop(); or break;
            }
```

This is the stop or break loop function. This is used to break out of loops if all sensors see black value, or in the last case of the code it will stop the motor if sensors see all black. The loop functions will be described in a different segment of the report.

### 5.3.5 Forward

Here things were straight forward. There were 1 forward condition used.

```
//forward
    else motor_forward(MAXSPEED,MOVETIME);
```

This translates to if none of the turn and stopping conditions are true it will go straight forward.

MAXSPEED was a defined power number for the robot, here set to 255

### 5.3.6 Loops

For this code the “for” loop function was used. 3 loops were implemented in the code for different part of the track. 1 for the start of the race, one for the actual race and one for stopping.

For the first segment the loop wold start with an IR signal and robot would slowly drive towards the black line, and when all the sensors see black it will break the loop and stop the motor and wait for a new IR signal. After second IR signal is received the robot goes forward over the black line and enters a new “for” loop, which has all the forward and turn functions as mentioned above. When the sensors see all black in that loop then the loop breaks and the robot will move forward over the second black line and enter the third and final loop. When the sensors in that final loop see all black, it means the robot has finished the race track, it will stop the motors and the code is complete.

## 5.4 The Zumo wrestling code

This code was built on the if/else code. It uses the same functions as the line following code but with few modifications.

It had the same start as the line following code. It will drive to the edge of the ring and wait for an IR signal. Then after the signal is received the robot goes forward and enters a “for” loop. The wrestling code only had 4 functions, 2 turns, 1 forward and 1 backwards.

```
//If all sensors are black
if (ref.l3 > BLACKL3Z && ref.l1 > BLACKL1Z && ref.r1 > BLACKR1Z &&
ref.r3 > BLACKR3Z)
           {
            motor_backward(200,500);
            tankturnleft(255,350);
           }
```

This translates to if all sensors see greater than set black value the robot will go full reverse for 500 milliseconds @200 power and then do a left tank turn for 350 milliseconds @full power (255)

```
//If Left (L3) sensor is black & rest is white
else if (ref.l3 > BLACKL3Z && ref.l1 < WHITEL1Z && ref.r1 < WHITER1Z
&& ref.r3 < WHITER3Z)
           {
            motor_backward(200,100);
            tankturnright (255,200);
           }
```

This translates to if the left-most sensor sees black but rest sees white it will reverse for 100 milliseconds @200 speed and make a right tank turn for 200 milliseconds @full power (255)

This will ensure that the robot will not turn out of the ring and the setting on the tank turn will make the robot turn about 60° so it will drive in a triangle in the circle.

Same turn was made for the right side with all the same parameters.

```
//Go forward if all is white

else motor_forward (255,1);
```

If none of the turn parameters are true the motor will go forward @full power (255) at 1 millisecond.

This loop will repeat with no stopping so it will keep going in the ring until it is manually turned off or batteries run out. The reason for low delay (1ms) for the forward part is so the sensor reading will be as fast as possible

# 6 The PD controller

After gaining a grasp on the relationship between the robot and the code through creating an "IF ELSE" script that could complete the track and complete all the requirements, work began on creating a PD controller.

## 6.1 PD controllers

PD controllers stand for Proportional Derivative Controllers and is two thirds of the commonly used PID (Proportional Integral Derivative Controller) found throughout various engineering works and projects. Each part of the name refers to a specific function found within the controller to make it an effective loop system that can intake data and use it to adjust the output to a desired level. Proportional control and derivative control are briefly discussed, in terms of the line following robot, below.

Proportional Control

The Proportional is the part of the control loop that gauges exactly how far from the desired target (the line in this case) the robot is and as a result how much correction is required to self-correct.

Derivative Control

Unlike Proportional, which takes its measurements solely on a moment by moment basis, the Derivative uses the previous and current data readings to predict where the line will be next, smoothing out the oscillations that come from the P only controller.

### 6.1.1 P controller

The first stage of building the PD controller was to establish a functioning Proportional control before adding a Derivative controller to optimize the efficiency of the robot on the track.

The theory behind a Proportional control is that by creating a maximum and a minimum for the data readings and then having a maximum and minimum for the speed (in this case the minimum speed would be 0, the maximum would initially be 255 which is the fastest the motors can go) it would become possible to create a formula in which the error would be turned into a value which could in turn be multiplied with the maximum possible correction to steer the robot onto the correct path. See fig 8 below for the standard formulas of a P controller.

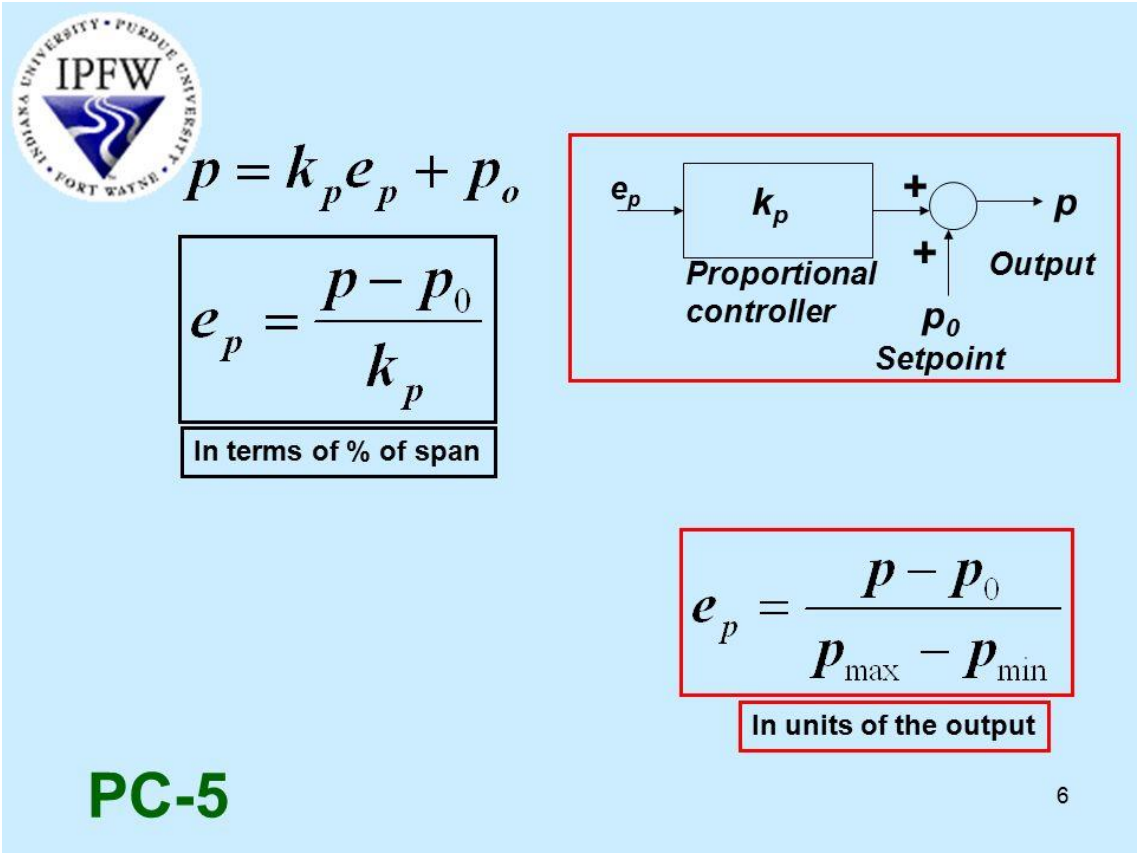



Figure 8. Theory behind P control formula

In the case of the project, the error values were initially set to an average of 5 black readings ($P_{max}$) for the minimum error and the average of 5 white values ($p_{min)}$ for the maximum error.

By subtracting the actual reading ($P_0$) from the targeted value ($P_{max}$) it was possible to create a proportion of error by dividing the remaining value by the difference in the minimum and maximum, or the fullest possible error value, which could then be given as the value of $E_1$. $E_1$ represented a proportion of how much the robot was correctly on the track. It was then a matter of converting this number, in our case a percentage, into a value that could be multiplied by the maximum possible speed. This meant if the $E_1$ value was 30%, for example, it would then require a 70% correction in motor speed for whichever engine needed to make the turn. This value could be achieved by subtracting $E_1$ from 100 and the result would then be multiplied by the maximum turn speed as well as another value, K. K is a constant value in the formula, it changes according to how well the robot performs and in what way it fails to stay on the track. A lower K value would prevent the robot from turning sharply enough to keep on the line, a higher K value would create too sharp a turn and result in a robot that oscillated heavily as it went over the track, slowing it down.

### 6.1.2 Challenges of the P Controller

After it was introduced as the next step following a successful “IF ELSE” code the first step was understanding how the p controller functioned. Whilst the principle of the p controller was understandable given time it proved difficult to turn into a code that functioned as intended. The first major hurdle was translating the given formula into an algorithm that C could compile. After a degree of discussion, it was decided to deviate from the formula presented to us and instead base the code on the understanding of how a p controller should function. First came the code that established the error value, this was formed using the same formula in the bottom right of fig. However once the e became established, the code diverged from the presented formula in a couple of ways. First, instead of using a constant (k) the code subtracted the decimal value that the error came as and subtracted it from 1 to get the value that the robot was missing from the line. That value then multiplied with the top speed in order to establish how much the robot needed to turn and using which motor.

The code was used for the left and right motors separately which resulted in the robot only following the code for the motor that came first in the compiler. As a result, the robot initially only turned left or right, depending on the code format, and so was unable to complete the track. After some further discussion, the code was condensed into just two lines. By running the left and right motor in the same line of code without one taking priority over the other in any way other than intended the robot successfully completed the track on its first test run. However, the max speed was set to 50, one fifth of its actual max speed. And so while it was successful in completing the track, any attempts to raise the speed to a competitive level resulted in the robot being unable to max adjustments to the motors in time to stay on the track.
After consultation with Joe, Mr.robot, we were made aware that the use of the K constant would help the robot turn more effectively at higher speeds. However our initial efforts to incorporate K into the existing code proved fruitless as the robot failed to run.
Taking what had become slightly more than a basic understanding of a P controller into mind an attempt began to start a fresh code that utilised the K constant from the start, rather than attempting to retroactively fit it into an existing code. This was met with mixed results but ultimately led to a robot that used K and could follow the line fairly quickly on several attempts on the test track. The need for tuning the constant as well recalibrating the sensors became apparent and while it worked better with the constant included it was far from race ready.

Before that was fully realized a collaborative effort with a member of another team resulted in the introduction of the Derivative Controller to the code and so work began on making a functioning PD controller

### 6.1.3 The PD Controller

As the Proportional controller was covered in .2a & b this section will focus more heavily on the Derivative Controller. A P only controller creates a robot that constantly needs corrections due to its own output. It doesn't have the ability to predict where the line will go next and so often overshoots first one direction and then the other in its attempts to gain the ideal position. This was something that became a problem even during the development of the P

controller and the solution was to introduce the next step, the Derivative portion of the PD controller. Below is an example of a full PID controller, ignoring the I section will give an example of how the P and D work in tandem before creating the appropriate output. The “I” segment was deemed unnecessary in this project.

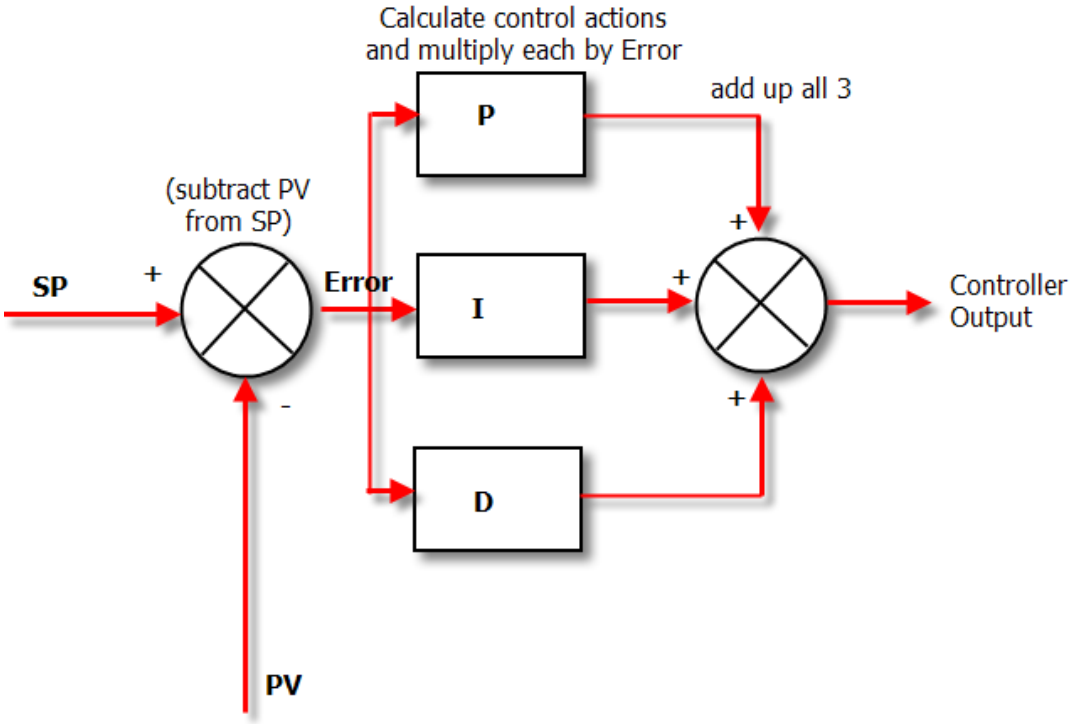



Figure 9. The flow chart of PID controller

As previously mentioned, the D controller attempts to predict the position of the line using the former and current errors and determining how much the robot will be off the line and adjusting things accordingly. This is useful for reigning in the P control and preventing it from overshooting constantly. A higher $K_d$ results in a smoother performance, $K_d$ being the constant in the Derivative controller. By focusing on the difference between the error and previous error the D is able to ensure that any dramatic increases or decreases to the speed that potentially overshoot the ideal positioning are reduced slightly. This frees the P controller up to have a higher gain and as a result increase the overall speed whilst not causing such extreme oscillations.

### 6.1.4 The challenges of a PD controller

The reasoning behind creating the PD controller before finalizing a working P controller was that, frankly, it might make it work properly with both aspects functioning at the same time. Whilst the code did result in a robot that was more capable of following the line it did have several setbacks.

The first issue discovered was that while the p control allowed the robot to follow the track, introducing the derivative algorithm required the code to be rectified slightly to fit both. As such the p control became initially less capable and the robot's performance suffered as a result. The issue was with redefining the $K_p$ to help the robot turn more sharply and try and work out the effect the $K_d$ had on the final performance.

The second, and larger, issue was understanding how to translate the theory of a Derivative Control into an actual usable code. This proved to be a lot more difficult than the P control as it was introduced to an already functioning algorithm and so incorporating it smoothly was initially complicated so not only did the P element suffer but it became difficult to ensure the Derivative would function as intended as well. This proved to be the problem that prevented the code from becoming fully realised. All varieties of the PD controller were, ultimately, incapable of completing the track satisfactorily. As the code became refined to fit in just a handful of lines whilst following the line more accurately, it suffered on the sharper turns and despite several efforts to refine the $K_p$ and $K_d$ constants over several lessons, it became apparent that the code was not capable of finishing the track in it's current state and time was short.

### 6.1.5 PD Controller Conclusion

In conclusion, the PD controller proved to have much more potential than the "IF ELSE" code that had been established already. Attempting to utilise this format in a Zumo robot provided a great deal of insight into just how the two separate elements function and interact. It also allowed excellent practice in converting an algorithm into the C programming language as well as giving practice in condensing lines of code into as few as possible.

However the complications with the code, the tuning parameters and ultimately the lack of time meant the final version of the code never came to fruition. Attempting to work out the kinks and bugs was deemed too risky as the “IF ELSE” code was capable of finishing the track but incapable of doing so quickly and thus needed more focus as it was more of a guarantee.

# 7 Conclusion

## 7.1 Race day, line following race

The race was at Metropolia on the 13.12.2017.

First we needed to calibrate the sensors for the new track. The measurements were dramatically different from our test track, because the test track had become very dirty over time which made our white values much higher than on a clean track.

When doing test runs we noticed immediately that the old values would not work on that track so calibration was required. After some calibration the robot finished the race, following all the set requirements for the course.

## 7.2 Race day, zumo wrestling

The zumo wrestling competition was after the line following race. The robot did very well and ended going in the final battle. But in the final battle the robot ran out of power so it ended up in 2$^{nd}$ place.

## 8 Final words

All of the authors of this document agree on that that this experience was very demanding but also very enlightening and educational. Not only did we get basic understanding for C programming, robotics, electrical engineering, but also an insight to the smart system industry and what that major will have to offer, if we decide to select that one next semester.

## References


1. 0J4214.1200.jpg (800×581) [Internet]. [cited 2017 Dec 15]. Available from: https://a.pololu-files.com/picture/0J4214.1200.jpg?fcb59377e1a393539427c563da1f9e1a
2. 3pi Spinning Line Follower example [Internet]. Pololu Forum. [cited 2017 Nov 29]. Available from: https://forum.pololu.com/t/3pi-spinning-line-follower-example/1156
3. am06-pid_16Sep06.pdf [Internet]. [cited 2017 Nov 22]. Available from: http://www.cds.caltech.edu/~murray/books/AM08/pdf/am06-pid_16Sep06.pdf
4. am06-pid_16Sep06.pdf [Internet]. [cited 2017 Nov 22]. Available from: http://www.cds.caltech.edu/~murray/books/AM08/pdf/am06-pid_16Sep06.pdf
5. Download PuTTY - a free SSH and telnet client for Windows [Internet]. [cited 2017 Dec 15]. Available from: http://www.putty.org/
6. download.pdf [Internet]. [cited 2017 Dec 14]. Available from: http://www.cypress.com/file/44826/download
7. High Performance Line Follower Robot [Internet]. Instructables.com. [cited 2017 Nov 29]. Available from: http://www.instructables.com/id/High-performance-Line-follower-Robot/
8. Line Following Robot With Basic PD (Proportional-Derivative) Control [Internet]. Instructables.com. [cited 2017 Nov 23]. Available from: http://www.instructables.com/id/Basic-PD-Proportional-Derivative-Line-Follower/
9. PID C++ implementation [Internet]. Gist. [cited 2017 Nov 29]. Available from: https://gist.github.com/bradley219/5373998
10. Mahadev Gopalakrishnan. Pid control for line follwoers [Internet]. Education presented at; 02:45:46 UTC [cited 2017 Nov 29]. Available from: https://www.slideshare.net/MahadevGopalakrishnan/pid-control-for-line-follwoers-22317416
11. Mahadev Gopalakrishnan. Pid control for line follwoers [Internet]. Education presented at; 02:45:46 UTC [cited 2017 Nov 29]. Available from: https://www.slideshare.net/MahadevGopalakrishnan/pid-control-for-line-follwoers-22317416
12. PID Controller - ControlTheoryPro.com [Internet]. [cited 2017 Nov 29]. Available from: http://wikis.controltheorypro.com/PID_Controller

3. Flownex SE. PID Controller Tuning [Internet]. [cited 2017 Nov 28]. Available from: https://www.youtube.com/watch?v=zkWUFOajrVk

14. troubleshooting_translator_issues [Zotero Documentation] [Internet]. [cited 2017 Dec 14]. Available from: https://www.zotero.org/support/troubleshooting_translator_issues

15. Zotero | Downloads [Internet]. [cited 2017 Nov 22]. Available from: https://www.zotero.org/