David Dallamore

P15188966

Pioneer robot simulation

Featuring Edge Following, Wander, Avoid, and Maps

Contents

[2. Foreword 3](#_Toc506506676)

[3. Introduction 3](#_Toc506506677)

[4. Behaviour Control Architecture 3](#_Toc506506678)

[5. Finite State Machine 4](#_Toc506506679)

[6. PID Controller 4](#_Toc506506680)

[7. Developmental Challenges and Solutions 5](#_Toc506506681)

[8. Testing 5](#_Toc506506682)

[9. Results 6](#_Toc506506683)

[9.1. Obstacle Avoidance 6](#_Toc506506684)

[9.2. Edge Following 6](#_Toc506506685)

[9.3. Wander 7](#_Toc506506686)

[9.4. Mapping 7](#_Toc506506687)

[10. Conclusion 7](#_Toc506506688)

[11. Appendix 8](#_Toc506506689)

[11.1 Finite State Machine 8](#_Toc506506690)

[11.2 Map Collection 9](#_Toc506506691)

[11.3 Edge Follow Outside the Box 10](#_Toc506506692)

[11.4 Successful Mapping including the robot’s path 10](#_Toc506506693)

[11.5 Main.cpp 11](#_Toc506506694)

[11.6 Avoid.h 12](#_Toc506506695)

[11.7 Avoid.cpp 13](#_Toc506506696)

[11.8 Follow.h 14](#_Toc506506697)

[11.9 Follow.cpp 15](#_Toc506506698)

[11.10 Map.h 16](#_Toc506506699)

[11.11 Map.cpp 16](#_Toc506506700)

[11.12 Wander.h 17](#_Toc506506701)

[11.13 Wander.cpp 18](#_Toc506506702)

[Figure 1 - Subsumption Implementation 3](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506703)

[Figure 2 - C++ switch statement 4](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506704)

[Figure 3 - Incorrect Mapping Attempt 5](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506705)

[Figure 4 - Correct Map Attempt 5](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506706)

[Figure 5 - Avoid / Follow interleaving 6](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506707)

[Figure 6 – Wander Pattern 7](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506708)

[Appendix 1 - Finite State Machine Diagram 8](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506709)

[Appendix 2 - Test Map Collection 9](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506710)

[Appendix 3 - Edge Follow outside the box 10](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506711)

[Appendix 4 - Mapping including robot's path 10](https://demontfortuniversity-my.sharepoint.com/personal/p15188966_my365_dmu_ac_uk/Documents/Robotics/Essay.docx#_Toc506506712)

# Foreword

The beginnings of this project were written in collaboration with Sandeep Kainth (P15203704) and Shivangi Karia (P15204042). Once the group had a minimally functional Edge Follow in place each member began to work on their project individually.

# Introduction

The aim of this project is the creation of a Pioneer Robot simulation which can:

* Edge follows at 1.0m from the obstacle
* Avoid obstacles within 0.5m
* Wander randomly if no obstacles detected within 1.5m
* Map the global location of any obstacles found

The project uses Microsoft Visual Studio 2017, written in C++ using ARIA (Advanced Robot Interface for Applications). The simulation makes extensive use of the robot’s sonar system to detect obstacles and uses MobileSim for all development and testing.

# Behaviour Control Architecture

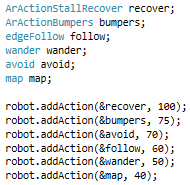


Figure 1 - Subsumption Implementation

A key aspect of the design of this simulation is the Subsumption model used to switch between each behaviour when needed. Figure 1 shows this implementation, the top section is the declaration of each behaviour, while the bottom section is each behaviour added to the robot. The number on the right-hand side is the priority; recover is given the highest priority because work cannot be completed while the robot is disabled.

The three-movement behaviours are arranged in this way because avoiding obstacles must be done to allow further edge following. Wander is only done while no other movement is being carried out, and mapping is a background process.

A major advantage of using this architecture is the redundancy of conflicts; a higher priority behaviour is given control no matter what the lower priority is doing. Another advantage is the simplicity of incremental implementation, the behaviours only need to be added once they are ready, and all other behaviours will function independently.

# Finite State Machine

**Switch** (state) {

**Case** idle:

Detect obstacles

**Case** left:

Turn left code

**Case** right:

Turn right code

Figure 2 - C++ switch statement

Avoid, Wander and Follow all make extensive use of switch statements, which is an important aspect of the control design. A switch statement allows different aspects of a given behaviour to modelled and accessed when needed. Figure 2 shows the basic skeleton of the switch statement used in the Avoid Behaviour. Each part of the behaviour is modelled as a state in the statement. In this example, the idle statement is the default state and is in control of when the behaviour is used. In Avoid, this is when an obstacle is detected within 500mm. It then detects the location of the obstacles relative to the robot and decides to turn left or right.

This style of switch statement is used in each behaviour to create Finite State Machine. Each behaviour can work simultaneously, but only one state in each behaviour can be active at any one time. The image Appendix 1 is a graphical representation of the entire Finite State Machine switch system used in this project. It is important to note the priority, which is used to give one behaviour control because its function is more important to the success of the robot. For example, it is more important for the robot to follow an edge than it is to wander randomly.

# PID Controller

A Reactive Control is a direct coupling of perception and action and if done well, results in the timely robotic response to a dynamic, unstructured environment. There is multiple examples of a reactive system including fuzzy logic based, neural network, Potential Fields. This project uses the PID system, which stands for Proportional, Integral, and Derivative.

The PID Control System is very well studied and has stood the test of time. A primary task behind the control is the monitoring of errors from the desired state. It is the responsibility of the proportional section to get a parameter back to a pre-established set point. The Integral section aims to reduce long-term errors, which the Derivative monitor the slope of errors of time. When tuned correctly this results in a system which efficiently returns to the desired state as quickly as possible.

This project uses the PID controller in the Edge-Following behaviour. The pre-established set point is the distance the robot must be from the wall, 1.0m. When the robot strays too close to, or too far from, the wall the error will build. It is the responsibility of the PID controller to gradually bring this error back to zero, therefore keeping the robot travelling in a straight line.

# Developmental Challenges and Solutions

There were many struggles faced during the development of this project. A very early one is the understanding of the Pioneer robot’s design. The sonars are designed to encompass the perimeter of the robot, with 0 degrees being the front, minus values to the right and positive to the left. This was very counter-intuitive, and a constant tripping point.

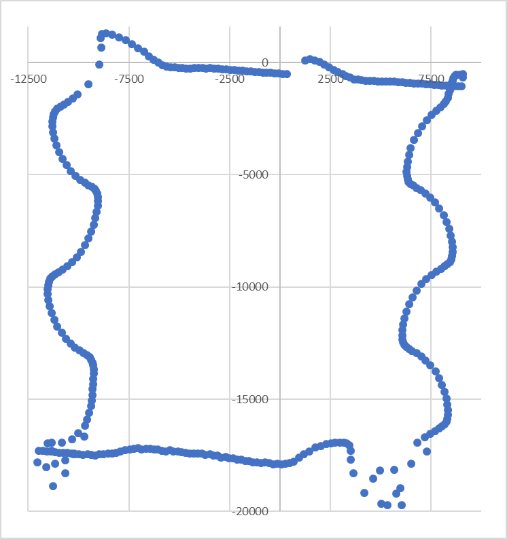


Figure 3 - Incorrect Mapping Attempt

A much more prominent problem with the project was the implementation of mapping. The earliest attempts returned the correct answer when given a set of numbers with a known result. However, when given the dynamic input of a running robot, it would always result in a map of wiggly lines.

Through testing included several methods of obtaining the sonar’s closest value and limiting the angle to be very slim. This was inspired by the very odd readings seen in the corners of the square.

During this testing, it became apparent that it must be the fault of the mathematics in use. Running through the code line by line and using many print statements to find the answers returned at every available point eventually provided a solution. The culprit was the code used to obtain the object’s theta value from the robot. The fully working map program utilises a function which can optionally provide a theta from the heading to the detected point. The result is rotated by 90 degrees clockwise to the original, but this is expected from the rotate method.

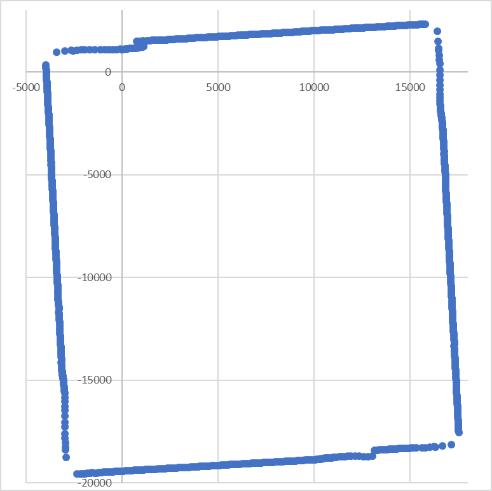


Figure 4 - Correct Map Attempt

During testing, it became apparent that the follow and avoid methods were contradictory to each other. This resulted in the robot getting stuck between both behaviours and stopping all functions. Initial thoughts were that there is a gap between a less than or more than comparison in the two behaviours. Further attempting to fine tune these threads were not successfully, the faulty code was eventually found to be in the edge-follow thread.

The problematic code was an if statement which attempts to limit the size of the error in the PID controller if the error were greater than 500mm in either direction, then the thread would switch to an idle state. This had the result of the following thread staying in an idle state while the robot sat in a position out of bounds of all other threads.

# Testing

Once each thread reached an acceptable working standard, they were testing again their specification.

The Obstacle Avoidance behaviour is expected to detect any obstacles within 0.5m of the robot and avoid them. The implementation of this resulted in the robot detecting the obstacle and turning using a delta heading of 135 degrees. The thread also attempts to turn away from the obstacle by detecting if it is on the left or right-hand side and spinning away from it.

The specification of the edge following is to follow a detected edge at a constant distance of 1m at the robot’s maximum speed of 0.2m/s. It is also important for the edge follow to cease when there is no edge left to follow. This was implement using the PID controller discussed above.

The Wander thread is specified to move forward for a random distance between 0.5m and 1.5m, once this distance is completed it must select a random angle between -140 degrees and 140 degrees and turn to it. This behaviour must only be active when no obstacle is detected within 1.5m of the robot. The distance limit was implement using an “if statement”, with a switch statement inside designed to control the random number generation and execution.

The mapping behaviour is a background process, and will map the global position of any obstacle that comes within 3500mm. This limit was imposed to limit the potential of false readings.

# Results

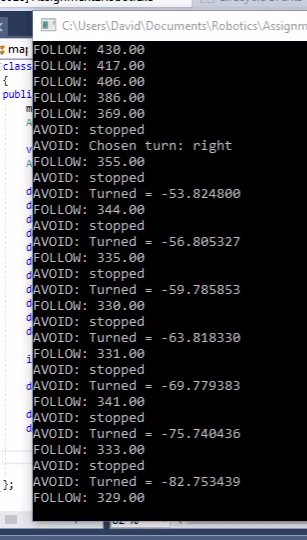


Figure 5 - Avoid / Follow interleaving

## 9.1. Obstacle Avoidance

Running the obstacle avoidance tests found that the avoidance action was trigger while the follow was still partially in control. As seen in Figure 5, both threads could briefly take control in the same period. The structure of the project means that Avoid took the most control because it has a higher priority, therefore resulting in the action being completed before handing fully back to follow or to wander. Further investigation into the code found that the problem was the same piece of code discussed in the Developmental Challenges section, the small if statement to limit the size of the error was resulting in the follow behaviour having no lower bound. However, the result is a fully functional avoid strategy, even if the path to getting there is slightly messier than originally intended.

## 9.2. Edge Following

Edge Following perform very well, nearly exactly as intended. Inside of any shape, the thread worked perfectly. The robot stayed within an acceptable error of the set point throughout its journey. Unfortunately, when it is placed on the outside of the square map, it performs drastically worse. Appendix 3 highlights the issues the robot has with edge follow outside of the box. The outside 90-degree angles cause the robot to lose the edge, which triggers wandering. During development, it was intended that the large sonar range (-100 to 100 in total) would be enough to catch objects just behind the robot to either side. However, this seems not to be the case, a potential solution would be to extend the angles further and increase, and distance from the robot within which follow will be triggered.

The testing also revealed that the robot struggles to edge follow with the obstacle on its right-hand side. Most of the development was done using the large square map with the robot travelling clockwise around it. An attempt to fix this was the introduction of an if statement with made the heading negative is the obstacle being following is on the right. However, this resulted in the robot no longer being able to turn corners, instead of turning into them triggering an avoid response.

## 9.3. Wander

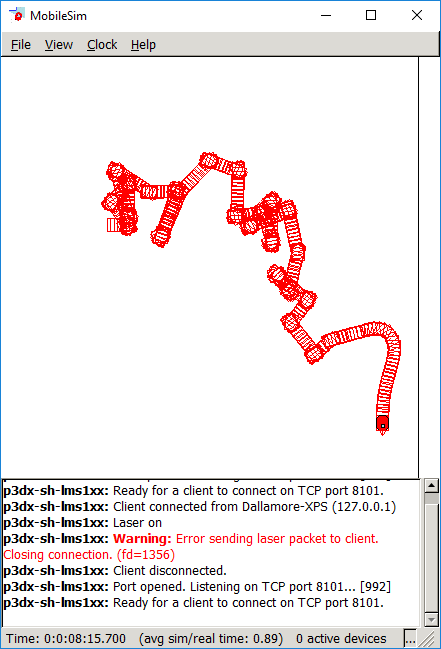


Figure 6 – Wander Pattern

The Wander thread may be one of the most successful threads; it performed exactly as expected. Figure 6 shows an example of the simulation beginning with the robot in the middle of nowhere, with nothing in Wander’s range of 1500mm. The robot proceeds to randomly select a distance to travel between 500mm and 1500mm, once it reaches this distance it chooses a random angle to turn between -140 and 140 degrees, once this turn is completed, it starts again with a distance. This can be interrupted at any point if the sonar finds and obstacle within 1500mm where it will then begin to edge follow as seen in the image.

## 9.4. Mapping

The result of the mapping thread is a great success. Appendix 4 shows the results of edge following the inner perimeter of the large square map (blue) with the addition of the robot’s path taken (red). The resulting map is very precise and an excellent representation of the real thing.

# Conclusion

The testing has proven the project to success overall; the robot can successfully Wander, Avoid, and Map obstacles. Edge Follow works well but would greatly benefit from further development; it may only need small modifications to aid in navigating corners correctly. However, it still successfully follows edges clockwise.

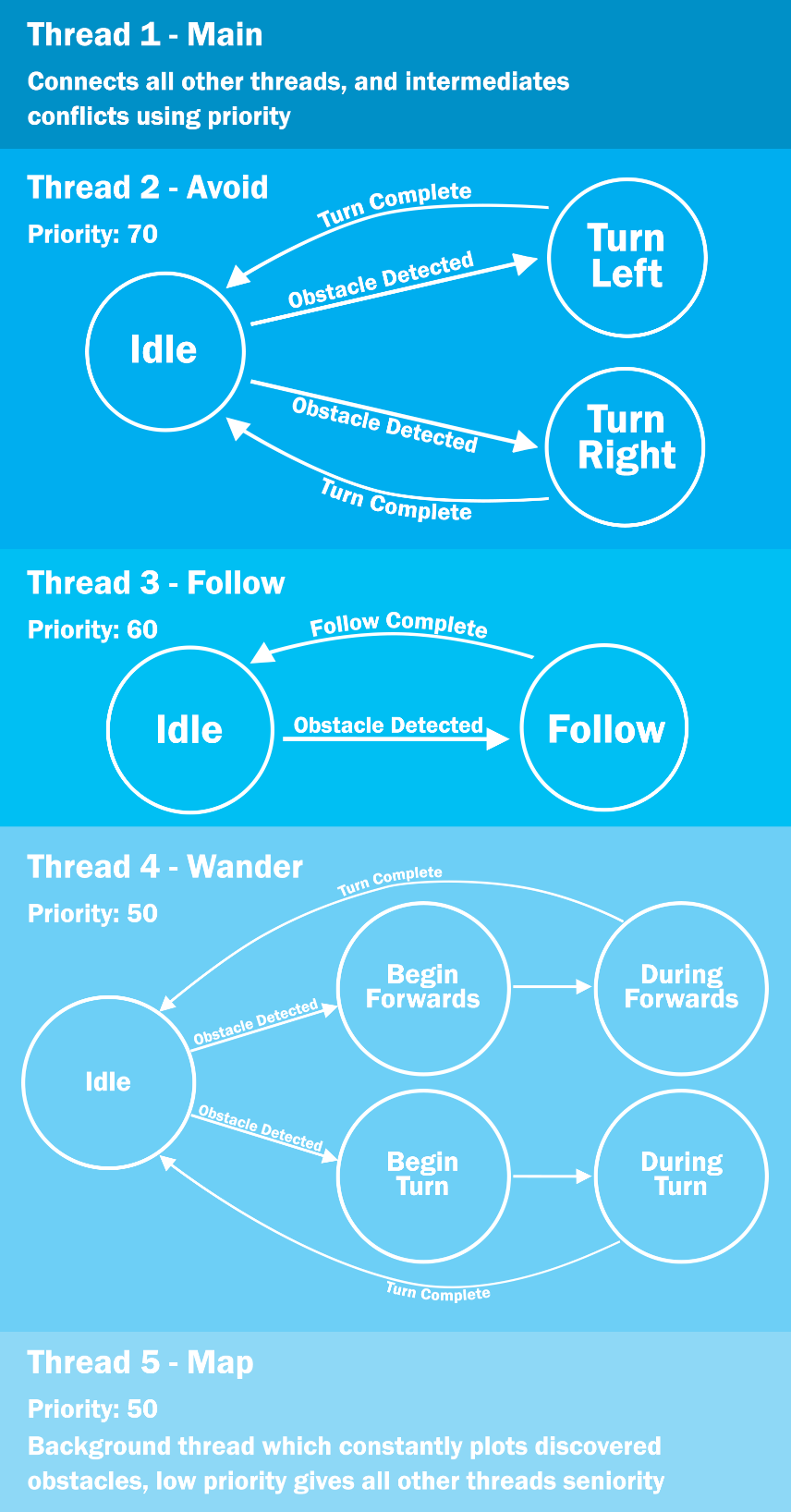
There are two extra features this project would benefit from, the first being live visualisation of the mapping. This was briefly attempted using SFML early on. However it became more important to have a fully working mapping feature first, before having the ability to visualise it in real time. The project still has SFML set as dependencies as well as the beginnings of the code effort commented out in main.cpp.

The second feature would be the use of Random Sampling and Consensus Algorithm or RANSAC. It would allow the data points of the obstacles to be mapped as straight lines instead of dots resulting in a more accurate map overall. The effects of a RANSAC would be much more prominent in a live visualisation than in an Excel Scattergram.

The lack of these features is acceptable as, without them, the robot is still fully functional. The project has successfully implemented multiple behaviours based on a finite state machine and can switch between them as and when is needed.

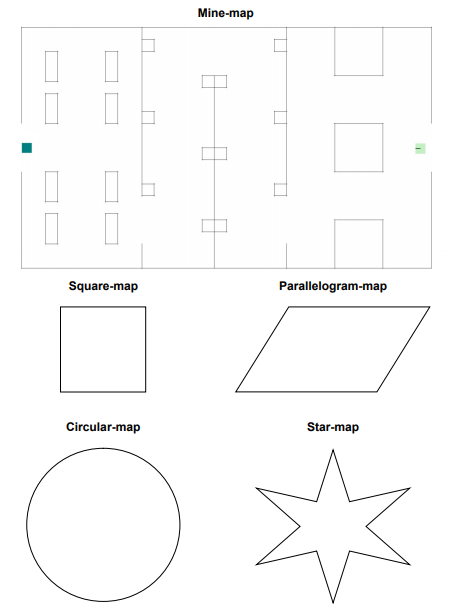
# Appendix

## 11.1 Finite State Machine



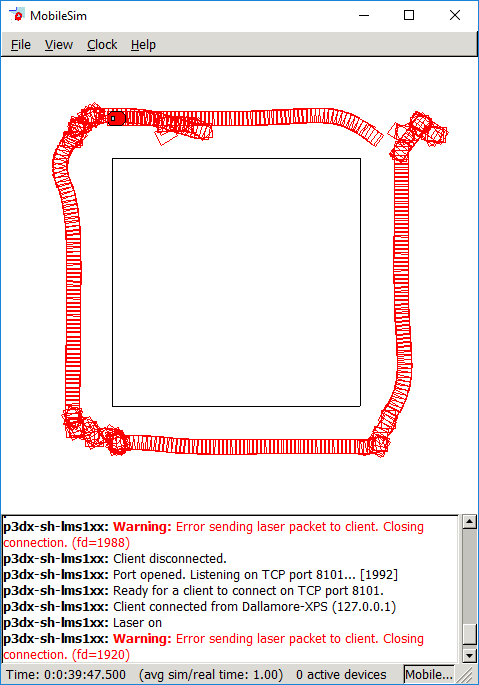
Appendix 1 - Finite State Machine Diagram

## 11.2 Map Collection



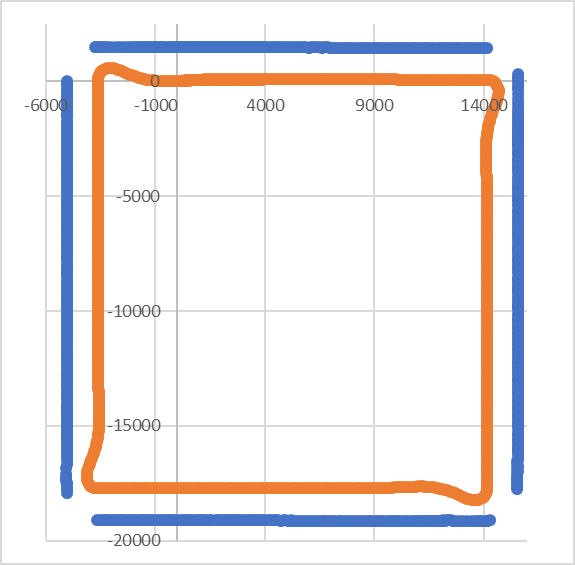
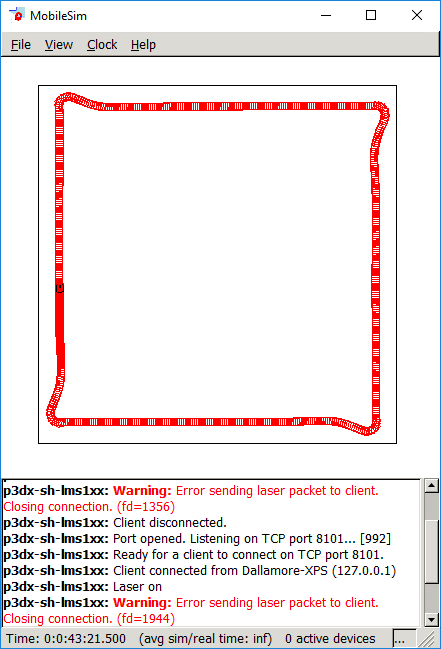
Appendix 2 - Test Map Collection

## 11.3 Edge Follow Outside the Box



Appendix 3 - Edge Follow outside the box

## 11.4 Successful Mapping including the robot’s path



Appendix 4 - Mapping including robot's path

## 11.5 Main.cpp

#include "Aria.h"

#include "follow.h"

#include "wander.h"

#include "avoid.h"

#include "map.h"

#include <SFML/Graphics.hpp>

int main(int argc, char \*\*argv)

{

Aria::init();

ArArgumentParser argParser(&argc, argv);

argParser.loadDefaultArguments();

ArRobot robot;

ArRobotConnector robotConnector(&argParser, &robot);

ArLaserConnector laserConnector(&argParser, &robot, &robotConnector);

// Always try to connect to the first laser:

argParser.addDefaultArgument("-connectLaser");

if(!robotConnector.connectRobot())

{

ArLog::log(ArLog::Terse, "Could not connect to the robot.");

if(argParser.checkHelpAndWarnUnparsed())

{

// -help not given, just exit.

Aria::logOptions();

Aria::exit(1);

}

}

// Trigger argument parsing

if (!Aria::parseArgs() || !argParser.checkHelpAndWarnUnparsed())

{

Aria::logOptions();

Aria::exit(1);

}

ArKeyHandler keyHandler;

Aria::setKeyHandler(&keyHandler);

robot.attachKeyHandler(&keyHandler);

puts("Press Escape to exit.");

ArSonarDevice sonar;

robot.addRangeDevice(&sonar);

robot.runAsync(true);

// try to connect to laser. if fail, warn but continue, using sonar only

if(!laserConnector.connectLasers())

{

ArLog::log(ArLog::Normal, "Warning: unable to connect to requested lasers, will wander using robot sonar only.");

}

// turn on the motors

robot.enableMotors();

// add a set of actions that combine together to effect the wander behavior

ArActionStallRecover recover;

ArActionBumpers bumpers;

edgeFollow follow;

wander wander;

avoid avoid;

map map;

robot.addAction(&recover, 100);

robot.addAction(&bumpers, 75);

robot.addAction(&avoid, 70);

robot.addAction(&follow, 60);

robot.addAction(&wander, 50);

robot.addAction(&map, 40);

//sf::RenderWindow window(sf::VideoMode(600, 600), "SFML works!");

//sf::CircleShape shape(1);

//shape.setFillColor(sf::Color::Green);

//shape.setPosition(200, 400);

////double x = map.xs;

//while (window.isOpen())

//{

// sf::Event event;

// while (window.pollEvent(event))

// {

// if (event.type == sf::Event::Closed)

// window.close();

// }

// window.clear();

// window.draw(shape);

// window.display();

//}

// wait for robot task loop to end before exiting the program

robot.waitForRunExit();

Aria::exit(0);

}

## 11.6 Avoid.h

class avoid : public ArAction

{

public:

avoid();

ArActionDesired \* fire(ArActionDesired d);

virtual ~avoid() {}

ArActionDesired desiredState;

protected:

double leftSonar;

double rightSonar;

double distance;

enum STATE {

idle,

left,

right

};

int state;

double proximity;

double heading;

double theta;

double objectTheta;

};

## 11.7 Avoid.cpp

#include <iostream>

#include <stdlib.h>

#include <Aria.h>

#include "avoid.h"

avoid::avoid() : ArAction("avoid dat crash boi") {

state = idle;

proximity = 510;

heading = 135;

}

ArActionDesired \* avoid::fire(ArActionDesired d) {

desiredState.reset();

distance = myRobot->checkRangeDevicesCurrentPolar(-80, 80, &objectTheta);

theta = myRobot->getTh();

if (distance < proximity) {

desiredState.setVel(0);

printf("AVOID: stopped\n");

switch (state) {

case idle:

if (objectTheta > 0) {

state = right;

printf("AVOID: Chosen turn: right\n");

}

else {

state = left;

printf("AVOID: Chosen turn: left\n");

}

break;

case right:

if (theta >= -heading - 5 && theta <= -heading + 5) {

printf("AVOID: TURN COMPLETE\n");

state = idle;

}

else {

printf("AVOID: Turned = %f\n", myRobot->getTh());

desiredState.setDeltaHeading(-heading);

}

break;

case left:

if (theta >= heading - 5 && theta <= heading + 5) {

printf("AVOID: TURN COMPLETE\n");

state = idle;

}

else {

printf("AVOID: Turned = %f\n", myRobot->getTh());

desiredState.setDeltaHeading(heading);

}

break;

default:

break;

}

}

return &desiredState;

}

## 11.8 Follow.h

class edgeFollow : public ArAction

{

public:

edgeFollow();

ArActionDesired \* fire(ArActionDesired d);

virtual ~edgeFollow() {}

ArActionDesired desiredState;

protected:

int speed;

enum state {

idle,

following

};

int state;

double setPoint;

double error;

double output;

double pGain;

double iGain;

double dGain;

double pOut;

double iOut;

double dOut;

bool leftOrRight;//L=0, R=1

double errorHistory;

double prevError;

double distance;

double leftSonar;

double rightSonar;

};

## 11.9 Follow.cpp

#include <iostream>

#include <stdlib.h>

#include <Aria.h>

#include "follow.h"

edgeFollow::edgeFollow() : ArAction("Stickin' to the straight and narrow") {

speed = 200;

setPoint = 1000;

pGain = 0.03; //0.03

iGain = 0.00005; //0.00005

dGain = 1.3; //1.3

}

ArActionDesired \* edgeFollow::fire(ArActionDesired d) {

desiredState.reset();

leftSonar = myRobot->getClosestSonarRange(-10, 100);

rightSonar = myRobot->getClosestSonarRange(-100, 10);

if (leftSonar <= rightSonar) {

distance = leftSonar;

leftOrRight = 0;

}

else {

distance = rightSonar;

leftOrRight = 1;

}

//printf("distance = %f\n", distance);

switch (state) {

case idle:

if (distance <= 1500) {

state = following;

}

else {

break;

}

case following:

prevError = error;

error = distance - setPoint;

if (error >= 4000) {

error = prevError;

}

//if (error > 500 || error <= -500) {

// state = idle;

// break;

//}

printf("FOLLOW: %.2f\n", distance);

pOut = pGain \* error;

iOut = iGain \* errorHistory;

dOut = dGain \* (error - prevError);

output = dOut + pOut + iOut;

errorHistory = errorHistory + error;

//if (leftOrRight == 1){

// output = -output;

//}

desiredState.setDeltaHeading(output);

desiredState.setVel(speed);

state = idle;

break;

default:

break;

}

return &desiredState;

}

## 11.10 Map.h

class map : public ArAction

{

public:

map();

ArActionDesired \* fire(ArActionDesired d);

virtual ~map() {}

ArActionDesired desiredState;

int count;

double r;

double objectTheta;

double robotX;

double robotY;

double robotTh;

double radius;

double objectX;

double objectY;

double xs;

double ys;

};

## 11.11 Map.cpp

#include <fstream>

#include <iostream>

#include <stdlib.h>

#include <vector>

#include <Aria.h>

#include "map.h"

map::map() : ArAction("Cartographer all up in here") {

//Know values for testing, answers should be 13.3, 5.85

//robotX = 12.6;

//robotY = 4.3;

//robotTh = -0.2;

//r = 1.1;

//thetaS = 1.35;

//radius = 0.6;

count = 0;

}

ArActionDesired \* map::fire(ArActionDesired d) {

//C++ cos/sin uses radians

//ArMath cos/sin uses degrees

desiredState.reset();

r = (myRobot->checkRangeDevicesCurrentPolar(-90, 90, &objectTheta));

if (r <= 3500 && count >= 5) {

count = 0;

robotX = myRobot->getX();

robotY = myRobot->getY();

robotTh = -myRobot->getTh();

radius = myRobot->getRobotRadius();

//Step One - find x and y

objectX = ArMath::cos(objectTheta) \* (r + radius);

objectY = ArMath::sin(objectTheta) \* (r + radius);

//Step Two - rotate to global coordinates

ArMath::pointRotate(&objectX, &objectY, robotTh);

//Step Three - translate to global coordinates

xs = robotX + objectX;

ys = robotY + objectY;

//Print to CSV file

std::ofstream printToCsv;

printToCsv.open("map.csv", std::ios::app);

printToCsv << xs << "," << ys << "," << robotX << "," << robotY << "\n";

}

count++;

return &desiredState;

}

## 11.12 Wander.h

class wander : public ArAction

{

public:

wander();

ArActionDesired \* fire(ArActionDesired d);

virtual ~wander() {}

ArActionDesired desiredState;

protected:

enum state {beginForwards,

duringForwards,

beginTurn,

duringTurn};

int state;

double range;

double heading;

double distance;

double travelled;

double angle;

int speed;

double theta;

double beginForwardsX;

double beginForwardsY;

};

## 11.13 Wander.cpp

#include <iostream>

#include <stdlib.h>

#include <Aria.h>

#include "wander.h"

wander::wander() : ArAction("Wander around") {

speed = 200;

heading = 10;

state = beginForwards;

distance = 1;

travelled = 0;

angle = 0;

}

ArActionDesired \* wander::fire(ArActionDesired d) {

desiredState.reset();

range = myRobot->checkRangeDevicesCurrentPolar(-90, 90);

theta = myRobot->getTh();

if (range > 1500) {

if (travelled > distance) {

state = beginTurn;

travelled = 0;

speed = 0;

printf("WANDER: FORWARDS COMPLETE\n");

}

switch (state) {

case beginForwards:

distance = (rand() % 1000 + 500);

printf("WANDER: Chosen distance: %f\n", distance);

beginForwardsX = myRobot->getX();

beginForwardsY = myRobot->getY();

heading = 0;

speed = 200;

state = duringForwards;

break;

case duringForwards:

travelled = sqrt(pow(myRobot->getX() - beginForwardsX, 2) + pow(myRobot->getY() - beginForwardsY, 2));

printf("WANDER: Travelled: %f\n", travelled);

break;

case beginTurn:

angle = rand() % 280 + -140;

printf("WANDER: Chosen turn: %f\n", angle);

state = duringTurn;

break;

case duringTurn:

if (theta >= angle - 5 && theta <= angle + 5) {

printf("WANDER: TURN COMPLETE\n");

state = beginForwards;

}

else {

printf("WANDER: Turned = %f\n", myRobot->getTh());

desiredState.setHeading(angle);

}

break;

default:

break;

}

}

desiredState.setVel(speed);

return &desiredState;

}