Mobile Robotics Coursework 2

p15194468

Sam Fuller

2018

Table of Contents

[Intro 2](#_Toc506484570)

[Design 2](#_Toc506484571)

[Architecture 2](#_Toc506484572)

[Behaviours 2](#_Toc506484573)

[Avoid Obstacle 2](#_Toc506484574)

[Wandering 2](#_Toc506484575)

[Edge Following 2](#_Toc506484576)

[Route Planning 2](#_Toc506484577)

[Testing 2](#_Toc506484578)

[Conclusion 2](#_Toc506484579)

[Bibliography 2](#_Toc506484580)

[Code 2](#_Toc506484581)

# Introduction

This report will explain the implementation of a robot controller using the Aria mobile robotics library. The controller will contain four behaviours, one to avoid obstacles, one to edge follow, one to wander, and one to plan a path through a given map. The behaviours will be implemented and tested with evidence and explanation in this report.

# Design

## Behaviours

### Avoid Obstacle

The avoid obstacle behaviour stops the robot from colliding with any object, a potentially damaging action. To ensure this never occurs, the behaviour repeatedly checks the forward sonar distances, if any returns a value smaller than 500mm the robot slows down or stops and then turns. If the obstacle is directly ahead the robot will reverse at -25mm/s giving it time to turn away from the corner before crashing. If the obstacle is to the side of the robot the speed is only decreased to 100mm/s, this gives the robot time to manoeuvre away from the object before a collision. In both cases the robot will turn away from the object at a rate of 25 degrees.

Avoid obstacle is the most important behaviour, because it is the only one that protects the robot, therefore the code utilises a pointer to a suppression bool given to every behaviour after instantiation. Only avoid obstacle changes the value of this bool, when true other behaviours will not activate. Cooldown is used so that for 20 iterations after an object is detected all other behaviours are suppressed, this gives the robot time to ensure it has turned away from the obstacle before another behaviour takes control.

Stucktime is used to ensure the robot does not become permanently stuck, if the robot is facing an acute corner the behaviour will try to turn it in to the corner. This is caused by the robot wanting to turn away from the nearest object, which in that particular case is on both of its sides. As the robot continues to oscillate against a corner stucktime will slowly increase until it reaches a critical point where the robot is made to turn a full 180 degrees, hopefully pointing the robot in the direction where there is no obstacle.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Behaviours | Test | Expected result | Actual result | Comments |
| Avoid Obstacle | Facing the robot towards a flat wall <500mm | The robot should turn away from the wall | The robot turns, facing towards the wall but at a significant angle. | Not the ideal result but an acceptable result, the avoid obstacle behaviour will be called again later and prevent a future crash. |
| Avoid Obstacle  forward | Facing the robot towards a wall >1500mm | The robot should approache the wall, when it gets within 500mm should turn away and continue | The robot approaches the wall, when it gets within 500mm it turns slightly away and continues, this happens a few times and the robot continues almost parallel to the wall | This is a good result, the robot did not collide with the wall, and made minimal corrections in order not to crash. |
| Avoid Obstacle  forward | Placed in mine.map | The robot should move around the map, approach a wall and turn away. | The robot navigates the map with surprising similarity to what is expected of edge following. | Another good result, so long as the robot never crashes the behaviour works. |
| Avoid Obstacle | Placing the robot between a locker and a wall on mine.map | The robot should head parallel to the wall half way between the wall and locker. | The robot continues very slowly and quickly oscillating between the locker and wall. | The robot was slower than expected, however made its way out of the small gap. |
| Avoid Obstacle | Placing the robot in an acute corner | The robot should stay in the corner for approx. 2 seconds and then turn around. | The robot oscillates in the corner for a few seconds and then turns 180 away from the corner. | This took much longer than expected to turn, the timer has been adjusted to speed it up. |

### Wandering

When no object is detected within 1500mm, the robot will wander randomly until an object is found. The wander behaviour travels a random distance between 500mm and 1500mm at a speed of 200mm/s before turning a random angle between -140 and 140 degrees.

The distance is calculated using a random number, whilst the distance has not been reached, the robot continues to travel forward. The distance travelled in the iteration is then subtracted from the remaining distance. The distance travelled is calculated using the current speed and the time of an iteration (assumed to be 0.1s). When the remaining distance becomes lower than 0, the end of that leg has been reached, the remaining distance is reset to a new random number and the robot turns by a different randomly generated number.

The wandering behaviour is surrounded by an if statement that checks suppression, if avoid obstacle is suppressing other behaviours, none of the wandering code will run.

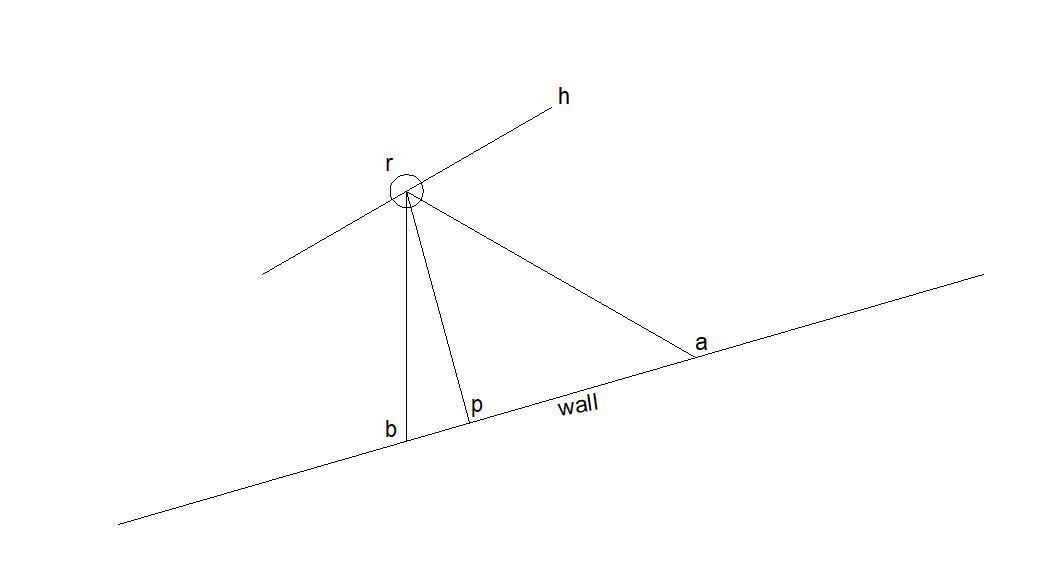
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Behaviours | Test | Expected result | Actual result | Comments |
| Wander | Letting the robot roam on mine.map | The robot should move forward and then turn randomly, this action should be repeated until a collision | The robot moves around map unexpectedly until it collides with a wall. | The collision only happens because avoid obstacle is off, this result is ideal. |

### Edge Following

Edge following calculates the distance from a side wall and attempts to maintain a 1000mm distance from the wall. When a wall is found the distance will not be exactly 1000mm, therefore an error needs to be calculated and from that a correct angle of approach towards the distance.

The algorithm used to determine the error takes the distance away from the wall and subtracts 1000mm, this gives a measurement of how far the robot is from where it needs to be. The error in distance is mapped to an angle that will point the robot towards the correct distance. The desired angle is worked out using the formula: desired angle = (error / 500) \* 45, this formula effectively maps an error of -500mm to 500mm on to -45**°** to 45**°**. Next the angle relative to the wall is calculated, this is then subtracted from the desired angle to give a delta angle. The delta angle represents the angle the robot must turn by to match the desired angle.

The distance and angle are calculated as follows:



r is the robot

ra is the distance of sensor 6

rb is the distance of sensor 9

arb therefore is 80**°**

rp is the distance between the robot and the wall, this is perpendicular to the wall

rpa and rpb are therefore 90**°**

h is the heading of the robot

hrp is therefore the heading of the robot relative to the wall

arh is 50**°**

cosine rule: a2 = b2 + c2 – 2bccos(A)

ab2 = ar2 + br2 - 2⋅ar⋅br⋅cos(80)

sine rule: sin(A) / a = sin(B) / b

sin(bar) / br = sin(arb) / ab

sin(bar) = br⋅(sin(60) / ab)

bar = par

sin(bar) = sin(par)

sine rule: sin(A) / a = sin(B) / b

rp / sin(par) = ar / sin(rpa) sin(rpa) = sin(90) = 1

rp = sin(par)⋅ar

rp is the distance between the robot and the wall.

par = sin-1(br⋅(sin(60) / ab))

A + B + C = 180

arp + par + rpa = 180

arp = 180 – par – 90

arp = 90 – par

hrp = arp + 50

the angle difference between the wall and the heading is equal to hrp – 90.

Edge following also calculates two RSME scores, one using all errors collected, and one using the last ten errors collected. This gives an accurate RSME of both long term and short-term calculations.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Behaviours | Test | Expected result | Actual result | Comments |
| Edge following | The robot is placed near a straight wall | The robot should turn and ‘lock on’ to the wall and follow at 1000mm | The robot turns and follows the wall, calculated distance is 1000, however the closest sensor reads nearer 900. | I believe the error to be within the calculation of the distance and including the robots radius in the sonar ranges. |
| Edge following | The robot is placed on a wall on square.map | The robot should find the new wall before colliding with it, and then follow it | The robot does similarly to the previous test until it reaches a corner, and makes a sharp turn and corrects itself to the new wall | This is acceptable but could be improved, future tests will indicated if an improvement is needed. |
| Edge following | Robot is made to edge follow in to a sharp corner on parallelogram | The robot should find the new wall before colliding with it, and then follow it | The robot does not find the new wall and continues on into the corner and gets trapped. | A fix has been made in order to turn away form the wall its following before reaching the corner |
| Edge following | Second test similar to previous | The robot should find the new wall before colliding with it, and then follow it | The robot now finds the new wall and edge follows that after turning the corner | The fix works, an additional test has been run similar to the second test, to ensure it performs similarly on square.map |
| Edge following | The robot is placed in star.map | unknown | The robot occasionally skips over points and stays in the center and sometimes follows the edge down the point. |  |

### Route Planning

Route planning is unfinished and therefore does not work, however the code written has been left in the program and has no affect on the ArActionDesired. Currently the code loads the map file into an array of coordinates of all the lines, transforms these lines into smaller numbers to fit on to a 70x24 grid (where each cell is half the length of the robot). The code should then mark the cells on the map as filled if a line passes through, this does not work but has been attempted.

Ideally if the map was created correctly, the function would then assess each cell to ensure that it has a free cell one either side vertically and horizontally, to ensure the robot has enough room to fit through. A route would then be calculated using A\* routing, this would then be followed by the robot in the fire action.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Behaviours | Test | Expected result | Actual result | Comments |
| Route planning | Ensure the code transforms the numbers to fit on a 70\*24 grid | An output file with the correct numbers in | An output file has been made and a selection of numbers have been checked to ensure the calculation works |  |
| Route planning | Ensure the code plots the coords on the map | An output file with a map resembling mine.map | An output file with a map, has some features of mine.map, however is not complete | This code does not work, multiple fixes have been attempted, and given more time could be made to work. |

## Architecture

The robot uses subsumption to calculate the ideal heading and speed. This allows for multiple behaviours to control the robot at any one time. However, avoid obstacle suppresses all other behaviours when an obstacle is detected, meaning aria only subsumes the avoid obstacle behaviour. Subsumption and suppression is ideal for the kind of robot that performs multiple behaviours at once. For example, both edge following and path planning can run at the same time, meaning the robot will attempt to get to the end of the map, but without getting too close to any edge. As a result of subsumption, behaviours can be more simple and be only coded for cases in their scope.

# Bibliography

Aria, 2016. *ARIA Developers Reverence Manual.* [Online]   
Available at: http://robots.mobilerobots.com/docs/api/ARIA/2.9.1/docs/index.html  
[Accessed 9 January 2018].

Joyce, D. E., 2013. *Summary of Trigonometric Formulas.* [Online]   
Available at: http://www2.clarku.edu/~djoyce/trig/formulas.html  
[Accessed 10 January 2018].

Simpson, J., Jacobsen, C. L. & Jadud, M. C., 2006. Mobile Robot Control The Subsumption Architecture and occam-pi. *Communicating Process Architectures,* pp. 225-236.

tutorials Point, n.d. *C++ Multi-Dimensional Arrays.* [Online]   
Available at: https://www.tutorialspoint.com/cplusplus/cpp\_multi\_dimensional\_arrays.htm  
[Accessed 13 January 2018].

# Code

## main

#include "Aria.h"

#include "wandering.h"

#include "avoidobstacle.h"

#include "routeplanning.h"

#include "follow.h"

static bool Suppression = false;

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

avoidobstacle avoidobstacle;

avoidobstacle.setup(&Suppression);

routeplanning routeplanning;

follow follow;

follow.setup(&Suppression);

wandering wandering;

wandering.setup(&Suppression);

robot.addAction(&avoidobstacle, 75);

robot.addAction(&routeplanning, 50);

robot.addAction(&follow, 50);

robot.addAction(&wandering, 25);

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

robot.waitForRunExit();

Aria::exit(0);

}

## avoidobstacle

### Header

class avoidobstacle : public ArAction // Class action inherits from ArAction

{

public:

avoidobstacle(); // Constructor

void setup(bool \* SuppressionIn);

virtual ~avoidobstacle() {} // Destructor

virtual ArActionDesired \* fire(ArActionDesired d); // Body of the action

ArActionDesired desiredState; // Holds state of the robot that we wish to action

protected:

int critDist; // Speed of the robot in mm/s

int cooldown; //time before control is given back to other behaviours

int stucktime; //time the robot has been stuck for

};

C++

#include <iostream>

#include <stdlib.h>

#include <Aria.h>

#include "avoidobstacle.h"

avoidobstacle::avoidobstacle() : ArAction("Avoid Obstacle")

{

critDist = 500;

}

bool \*avoidSuppression;

void avoidobstacle::setup(bool \*SuppressionIn) {

avoidSuppression = SuppressionIn;

}

// Body of action

ArActionDesired \* avoidobstacle::fire(ArActionDesired d)

{

desiredState.reset(); // reset the desired state (must be done)

//desiredState.setVel(myRobot->getVel() + 10);

int closestObject = myRobot->getClosestSonarRange(-40, 40); //get the sonars from front

int closestSonar = myRobot->getClosestSonarNumber(-40, 40); //get the sonar number

if (closestObject < critDist) //if an object is within critical distance

{

\*avoidSuppression = true; //suppress all other behaviours

cooldown = 20;

stucktime+=10; //increase the amount of time the robot has been stuck

if (stucktime < 200) {

if (myRobot->getClosestSonarRange(-10, 10) <= (critDist / 2)) {

desiredState.setVel(-25); //if the object is directly ahead reverse

}

else {

desiredState.setVel(100); //else just slow down

}

if (closestSonar <= 3) {

myRobot->setDeltaHeading(-25); //if the closest object is to the left turn right

}

else if (closestSonar >= 3) {

myRobot->setDeltaHeading(25); //if the closest object is to the right turn left

}

}

else {

myRobot->setDeltaHeading(180); //if the robot has been stuck for a long time turn around

stucktime = 0;

}

system("CLS"); //output

printf("Current State: AvoidObstacle - active\n");

printf("Nearest Object: %i\n", myRobot->getClosestSonarRange(-179, 179));

printf("x: %f y: %f th: %f vel: %f\n", myRobot->getX(), myRobot->getY(), myRobot->getTh(), myRobot->getVel());

}

else {

if (cooldown <= 0) {

\*avoidSuppression = false; //unsuppress all other behaviours

stucktime = 0;

}

else {

cooldown--; //robot has just come out of being stuck (dont unsuppress other behaviours until the robot has had time to turn and back away)

stucktime > 0 ? stucktime-=2 : false ;

system("CLS"); //output

printf("Current State: AvoidObstacle - cooldown\n");

printf("Nearest Object: %i\n", myRobot->getClosestSonarRange(-179, 179));

printf("x: %f y: %f th: %f vel: %f\n", myRobot->getX(), myRobot->getY(), myRobot->getTh(), myRobot->getVel());

}

}

return &desiredState; // give the desired state to the robot for actioning

}

## wandering

### Header

class wandering : public ArAction // Class action inherits from ArAction

{

public:

wandering(); // Constructor

void setup(bool \* SuppressionIn);

virtual ~wandering() {} // Destructor

virtual ArActionDesired \* fire(ArActionDesired d); // Body of the action

ArActionDesired desiredState; // Holds state of the robot that we wish to action

protected:

int targetSpeed; //the target speed of the robot (200mm/s)

int currentSpeed; //the current speed of the robot

int maxDist; //the distance from an object before the robot wanders

int remainingDist; //the distance before the next turn

int lastDist; //the distance of the current 'leg'

int lastTurn; //the angle of the last turn

};

### C++

#include <iostream>

#include <stdlib.h>

#include <Aria.h>

#include "wandering.h"

// Implementation

// Constructor

wandering::wandering() : ArAction("wandering")

{

maxDist = 1700;

targetSpeed = 200;

remainingDist = (rand() % 1000) + 500; //in mm

}

//pointer to the suppression boolean

bool \*wanderSuppression;

void wandering::setup(bool \*SuppressionIn) {

wanderSuppression = SuppressionIn;

}

ArActionDesired \* wandering::fire(ArActionDesired d)

{

desiredState.reset();

currentSpeed = myRobot->getVel();

//if this behaviour is not being suppressed

if (\*wanderSuppression == false) {

//if the closest sonar reports higher than the max distance

if (myRobot->getClosestSonarRange(-179, 179) > maxDist) {

//if the remaining distance has been reached

if (remainingDist < 0) {

lastTurn = (rand() % 280) - 140; //create a new turn angle

myRobot->setDeltaHeading(lastTurn); //turn by that angle

lastDist = remainingDist = (rand() % 1000) + 500; //create a new remainingDist (and set lastDist too)

system("CLS"); //outputs

printf("Current State: Turning\n");

printf("Nearest Object: %i\n", myRobot->getClosestSonarRange(-179, 179));

printf("x: %f y: %f th: %f vel: %f\n", myRobot->getX(), myRobot->getY(), myRobot->getTh(), myRobot->getVel());

printf("Total Distance to Travel: %i Distance Until Next Turn: %i\n", lastDist, remainingDist);

printf("Last Turn was %i degrees", lastTurn);

}

else {

remainingDist = remainingDist - (currentSpeed / 10); //remove from the remaining distance the distance traveled by the robot (d = s \* t) time is 1/10s

system("CLS"); //outputs

printf("Current State: Wander - Straight\n");

printf("Nearest Object: %i\n", myRobot->getClosestSonarRange(-179, 179));

printf("x: %f y: %f th: %f vel: %f\n", myRobot->getX(), myRobot->getY(), myRobot->getTh(), myRobot->getVel());

printf("Total Distance to Travel: %i Distance Until Next Turn: %i\n", lastDist, remainingDist);

printf("Last Turn was %i degrees", lastTurn);

}

}

if (currentSpeed < targetSpeed) //whilst below target speed

desiredState.setVel(currentSpeed + 10); //speed up

;

}

return &desiredState;

}

## follow

### Header

// Signatures

class follow : public ArAction // Class action inherits from ArAction

{

public:

follow(); // Constructor

void setup(bool \* SuppressionIn);

virtual ~follow() {} // Destructor

virtual ArActionDesired \* fire(ArActionDesired d); // Body of the action

ArActionDesired desiredState; // Holds state of the robot that we wish to action

protected:

int speed; // target speed of the robot in mm/s

double deltaHeading;

//following distances (so they can be edited if needed)

double followDist;

double maxFollowDist;

double minFollowDist;

//variables for calculating the RMSE and storing previous Errors

int walltimer;

double RMSE10[10];

double RMSEall[10000];

double RMSE10ans;

double RMSEallans;

//variables used in calculating and storing angles

double desiredAngle;

double actualAngle;

double deltaAngle;

//variables for storing sensors, distance, and errors relating to the left

double leftFront;

double leftBack;

double leftDist;

double leftError;

//variables for storing sensors, distance, and errors relating to the left

double rightFront;

double rightBack;

double rightDist;

double rightError;

};

### C++

#include <iostream>

#include <stdlib.h>

#include <Aria.h>

#include "follow.h"

// Implementation

// Constructor

follow::follow() : ArAction("Follow Edge")

{

speed = 200; // Set the robots speed to 50 mm/s. 200 is top speed

deltaHeading = 0; // Straight line

followDist = 1000;

maxFollowDist = 1500;

minFollowDist = 500;

walltimer = 0;

}

double distance(double ar, double br) {

double ab = sqrt(ar \* ar + br \* br - 2\*ar\*br\*cos(80 \* M\_PI/180));

double sinbar = br \* (sin(M\_PI / 3) / ab);

double sincar = sinbar;

double rc = sincar \* ar;

return rc;

}

double angle(double ar, double br) {

double ab = sqrt(ar \* ar + br \* br - 2\*ar\*br\*cos(80 \* M\_PI / 180));

double sinbar = br \* (sin(M\_PI/3)/ ab);

double bar = asin(sinbar) \* (180 / M\_PI);

double arc = 90 - bar;

return -(arc - 50);

}

//pointer to the suppression boolean

bool \*followSuppression;

void follow::setup(bool \*SuppressionIn) {

followSuppression = SuppressionIn;

}

// Body of action

ArActionDesired \* follow::fire(ArActionDesired d)

{

desiredState.reset(); // reset the desired state (must be done)

//if behaviour is not being suppressed

if (\*followSuppression == false) {

leftFront = myRobot->getSonarRange(1) + myRobot->getRobotRadius(); //get the data from the sonars

leftBack = myRobot->getSonarRange(14) + myRobot->getRobotRadius();

leftDist = distance(leftFront, leftBack); //calculate distance from the wall

if ((leftDist < maxFollowDist)) {

leftError = leftDist - followDist; //error is calculated from the distances

desiredState.setVel(speed);

}

else {

leftError = 100000; //number that will never be reached to signal no wall found in range

}

rightFront = myRobot->getSonarRange(6) + myRobot->getRobotRadius(); //same as above but for the right

rightBack = myRobot->getSonarRange(9) + myRobot->getRobotRadius();

rightDist = distance(rightFront, rightBack);

if ((rightDist < maxFollowDist)) {

rightError = rightDist - followDist;

desiredState.setVel(speed);

}

else {

rightError = 100000;

}

if (abs(leftError) < abs(rightError) && (leftError != 100000 || rightError != 100000)) { //for left edge following

desiredAngle = (leftError / 500) \* 45; //calculate the desired angle from the error

if (myRobot->getSonarRange(3) < (maxFollowDist+1000)) {

desiredAngle = desiredAngle - (((maxFollowDist + 1000) - myRobot->getSonarRange(3)) / 1000) \* 45; //scale the desired angle if there is a wall ahead

}

actualAngle = angle(leftFront, leftBack); //calculate the actual angle

deltaAngle = desiredAngle - actualAngle; //delta angle calculated

myRobot->setDeltaHeading(deltaAngle); //angle set

RMSE10 [walltimer % 10] = leftError; //store the error in the RMSE arrays

RMSEall [walltimer] = leftError;

RMSE10ans = 0;

if (walltimer > 10) { //calculate the RMSE for the last 10 errors

for (int i = 0; i < 10; i++) {

RMSE10ans += (RMSE10[i] \* RMSE10[i]);

}

RMSE10ans /= 10;

RMSE10ans = sqrt(RMSE10ans);

}

RMSEallans = 0;

for (int i = 0; i <= walltimer; i++) { //calculate the RSME for all errors

RMSEallans += (RMSEall[i] \* RMSEall[i]);

}

RMSEallans /= walltimer;

RMSEallans = sqrt(RMSEallans);

walltimer++;

system("CLS"); //outputs

printf("Current State: Following Wall - left\n");

printf("Nearest Object: %i\n", myRobot->getClosestSonarRange(-179, 179));

printf("x: %f y: %f th: %f vel: %f\n", myRobot->getX(), myRobot->getY(), myRobot->getTh(), myRobot->getVel());

printf("RMSE: %f RMSE of last 10: %f", RMSEallans, RMSE10ans);

}

else if (abs(leftError) >= abs(rightError) && (leftError != 100000 || rightError != 100000)) { //for right edge following (same as above but for the right)

desiredAngle = (rightError / 500) \* 45;

if (myRobot->getSonarRange(4) < (maxFollowDist + 1000)) {

desiredAngle = desiredAngle - (((maxFollowDist + 1000) - myRobot->getSonarRange(4)) / 1000) \* 45;

}

actualAngle = angle(rightFront, rightBack);

deltaAngle = actualAngle - desiredAngle;

myRobot->setDeltaHeading(deltaAngle);

walltimer++;

RMSE10[walltimer % 10] = rightError;

RMSEall[walltimer] = rightError;

RMSE10ans = 0;

if (walltimer > 10) {

for (int i = 0; i < 10; i++) {

RMSE10ans += (RMSE10[i] \* RMSE10[i]);

}

RMSE10ans /= 10;

RMSE10ans = sqrt(RMSE10ans);

}

RMSEallans = 0;

for (int i = 0; i <= walltimer; i++) {

RMSEallans += (RMSEall[i] \* RMSEall[i]);

}

RMSEallans /= walltimer;

RMSEallans = sqrt(RMSEallans);

walltimer++;

system("CLS");

printf("Current State: Following Wall - right\n");

printf("Nearest Object: %i\n", myRobot->getClosestSonarRange(-179, 179));

printf("x: %f y: %f th: %f vel: %f\n", myRobot->getX(), myRobot->getY(), myRobot->getTh(), myRobot->getVel());

printf("RMSE: %f RMSE of last 10: %f", RMSEallans, RMSE10ans);

}

else { //for no edge following

desiredAngle = 0;

actualAngle = 0;

deltaAngle = 0;

}

// Set the heading change of the robot

}

return &desiredState; // give the desired state to the robot for actioning

}

## routeplanning

### Header

class routeplanning : public ArAction // Class action inherits from ArAction

{

public:

routeplanning(); // Constructor

virtual ~routeplanning() {} // Destructor

virtual ArActionDesired \* fire(ArActionDesired d); // Body of the action

ArActionDesired desiredState; // Holds state of the robot that we wish to action

protected:

int currentcoord; //the current coord for an input loop

int size; //the amount of variables in coords[] (will vary from map to map)

int coords[500]; //an array for storing all map line coords

int map[70][42]; //an array for storing the map and route (-1 for a wall, 0+ for the route)

};

### C++

#include <iostream>

#include <stdlib.h>

#include <Aria.h>

#include "avoidobstacle.h"

avoidobstacle::avoidobstacle() : ArAction("Avoid Obstacle")

{

critDist = 500;

}

bool \*avoidSuppression;

void avoidobstacle::setup(bool \*SuppressionIn) {

avoidSuppression = SuppressionIn;

}

// Body of action

ArActionDesired \* avoidobstacle::fire(ArActionDesired d)

{

desiredState.reset(); // reset the desired state (must be done)

//desiredState.setVel(myRobot->getVel() + 10);

int closestObject = myRobot->getClosestSonarRange(-40, 40); //get the sonars from front

int closestSonar = myRobot->getClosestSonarNumber(-40, 40); //get the sonar number

if (closestObject < critDist) //if an object is within critical distance

{

\*avoidSuppression = true; //suppress all other behaviours

cooldown = 20;

stucktime+=10; //increase the amount of time the robot has been stuck

if (stucktime < 200) {

if (myRobot->getClosestSonarRange(-10, 10) <= (critDist / 2)) {

desiredState.setVel(-25); //if the object is directly ahead reverse

}

else {

desiredState.setVel(100); //else just slow down

}

if (closestSonar <= 3) {

myRobot->setDeltaHeading(-25); //if the closest object is to the left turn right

}

else if (closestSonar >= 3) {

myRobot->setDeltaHeading(25); //if the closest object is to the right turn left

}

}

else {

myRobot->setDeltaHeading(180); //if the robot has been stuck for a long time turn around

stucktime = 0;

}

system("CLS"); //output

printf("Current State: AvoidObstacle - active\n");

printf("Nearest Object: %i\n", myRobot->getClosestSonarRange(-179, 179));

printf("x: %f y: %f th: %f vel: %f\n", myRobot->getX(), myRobot->getY(), myRobot->getTh(), myRobot->getVel());

}

else {

if (cooldown <= 0) {

\*avoidSuppression = false; //unsuppress all other behaviours

stucktime = 0;

}

else {

cooldown--; //robot has just come out of being stuck (dont unsuppress other behaviours until the robot has had time to turn and back away)

stucktime > 0 ? stucktime-=2 : false ;

system("CLS"); //output

printf("Current State: AvoidObstacle - cooldown\n");

printf("Nearest Object: %i\n", myRobot->getClosestSonarRange(-179, 179));

printf("x: %f y: %f th: %f vel: %f\n", myRobot->getX(), myRobot->getY(), myRobot->getTh(), myRobot->getVel());

}

}

return &desiredState; // give the desired state to the robot for actioning

}