Assignment 1 Simulation

By Alex Mezodi

Table of Contents

[Software Design](#Software)……………………………………………………………………………………….……………………………..pg. 4

[Simulation Testing](#Simulation)………………………………………………………………………………………………...….……………. pg. 5

[Code of my program](#Code)………………………………………………………………………………………………………………. pg. 9

Introduction

In this section I will be shortly describing real world autonomous robots I have learnt about in this course. In the second lecture which was about Mobility, we have watched a short video about the NAO robots. I picked this one to talk about first, as my individual project is about the robot Pepper, a robot from the same company as NAO. NAO is smaller than Pepper; about the size if a little child and can be programmed to talk, dance and is usable for almost any sensual applications. Pepper is even more impressive; she is around 5 foot tall and instead of having legs like NAO she uses wheels to assist her in precise movement. Since my project is about movement I know a few things about this. Pepper has lasers and sonars located at the bottom of her frame and using a provided algorithm can map an environment and move around out. The movement in the wheels is recorded so she knows where she is on the map. She is fully autonomous; can rotate her hips, arms including shoulders, elbows, fingers, as well as her head.



Yet another interesting and commercial robot we learn about in Lecture 7 Navigation was the Roomba. These robots are used to clean flat homes; they are essentially robot hoovers. They work very slowly, however they don’t omit much noise. In the lectures we talked about robots which use this algorithm to know where they have already been. This is useful for the Roomba as it records where it already cleaned the floor. They essentially use a combination of wall following and random bounce algorithm. This means they follow walls and go in a random direction until they hit an obstacle or a spot they already cleaned. They use their infrared sensors to detect obstacles instead of something like sonar. The older Roombas used to be completely random with movement, but the newer ones take pictures using Visual SLAM to know where they are in the room.

C:\Users\amezod\Downloads\Bio-inspired_Big_Dog_quadruped_robot_is_being_developed_as_a_mule_that_can_traverse_difficult_terrain.tiffThe final robot I wanted to mention that I found interesting personally was Big Dog by Boston Dynamics. By the name, it is easy to imagine what this robot looks like. It is a huge robot dog that was designed to accompany soldiers in terrain that was hard to access with robots and vehicles with wheels. It essentially carries the soldiers gear. There are around 50 sensors located on Big Dog that assist the robot with acceleration, motion, force of joints, actuators, temperature and many more. Unfortunately as I was doing my own research for this assignment I read that the Big Dog project that started originally in 2005 was discontinued in 2015. The reason for the discontinuation was that the robot was too noisy. I have watched videos of Big Dog and the movement wasn’t that precise either in my opinion. Especially when the robot received the hit, it would just tumble over. I can only imagine how difficult it would be to correct a robot which weighs a 110kg. Not to mention that it could carry 150kg which is more than the actual robot’s weight. Working with Pepper as I mentioned above really showed me how Robots are still taking their baby steps even in 2018 especially when it comes to navigation.

Software Design – Flow Chart



Simulation Testing

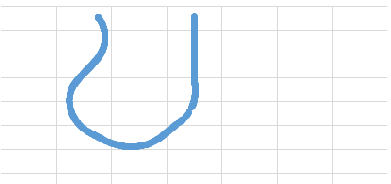
All of the following data and graphs generated data were obtained by running the simulation. Around every 20ms when the update function ran (please refer to the code below), we not only would make sure that the robot is sensing the distances of the obstacles around the robot and navigates around the map effectively, but also write a line of data readings into a text file titled data.txt.

These data readings would include trajectory information (where the robot is on the map), information about the speed of each of the two wheels, as well as odometry readings concerning the sonar and laser sensors.

Trajectory – below you can see the trajectory of the robot which is the blue line. I thought it would be interesting to include this trajectory together with the map graph generated from the *laser* readings. We get the position of the robot by simply invoking the getX( ) and getY( ) functions on the robot object. I will talk about how to obtain the laser readings later.

In the second assignment due to the nature of the calculation and how Microsoft Excel works, it was impossible to combine this trajectory map below and the laser readings map. Not only that, but result columns had to be edited, flipped with the x,y results and the end result graph rotated to make sense of it. Hence why I have to apologise for the low resolution images, however there is nothing which can be done about it. I witnessed multiple people reporting this error in Excel.

In the image below we see the robot starting next to the left wall (or the right wall from this perspective) and self-correcting itself a little bit due to the self-correction algorithm. Then the robot takes in the pillars rotation and self-correcting right. Finally after we grabbed the ball the robot turns leftwards into the goal.



Speed – to obtain the speed graph we simply paste the values for the variables we used to save the wheel velocity. We simply update the values for speed1 and speed2 variables then call robot.setVel2 (speed1, speed2).

In this second assignment, the wheels fluctuate a bit more which is to be expected. We start our wheel speed in different states – this is due to the correction algorithm which keeps the robot close to the left initial wall. The difference between speed 60 mm/s and 70mm/s in most of the times negligible anyways. A lot of the times it just results an overall same speed in both wheels.

After that we have two major turns where the robot is turning right in through the first pillars and self corrects hence the wiggly lines in the middle of the graph. The second pair of pillars is taken in with a big turn resulting in the left wheel going up to a 120 mm/s and the right wheel staying at a 90 mm/s.

After we see the ball we can view a constant fluctuation of both wheels. This is due to the algorithm inside the chase function which self corrects similarly to the left wall algorithm but is even more precise. Since the ball is on the right side after the last pillars, this means again that the robot is turning right.

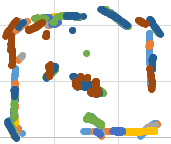
Finally, since our robot overturns to grab the ball, we let it go towards the end goal to the left. This results in the right wheel going up to a speed of 60mm/s and the left only a meek 40 mm/s. Both wheels halt to a 0 mm/s when we see a wall in front of us.

For a more accurate and technical understanding please refer to the code below.

Map readings – we use the same formula to calculate both the laser and sonar readings separately. The only difference is using different functions to access different parts of the ArLaser and ArRobot’s sonar readings framework.

There are three general steps to obtain map readings. We have to calculate the range of the obstacle from the robot (sensor), rotate this data according to the theta of the robot itself, and finally add the position of the robot on this local frame.

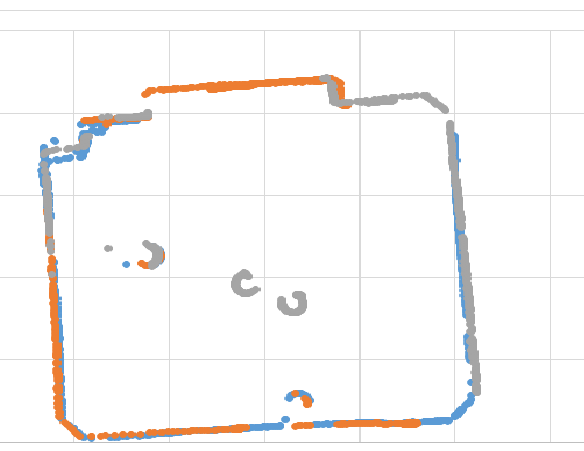
Sonar readings – the sonar readings are extremely imprecise as we have been warned by the graduate lab assistants. As you can see on the image below, they pick up all sorts of readings even outside the box the robot is running in. I have spent a great deal of time trying to filter these results – I included code to omit results which are under 0 or over 7000 millimetres away from the robot’s position. Unfortunately this Microsoft Office Excel cannot work with and provides atrocious results as random lines. What we can do is omit the lines outside this box.



After multiple tries and inspecting the code this is the best the sonar can provide. It picks up the obstacles inside the box pretty well, but the walls are missing as the sonar picks up obstacles outside the box sometimes resulting in results being omitted. The robot’s path is not hindered by this though as can be viewed so it is relatively safe to use sonar. Laser is still the recommended route as you’ll be able to read below.

Laser readings – the laser readings in this real robot assignment proved to be much more precise than the sonar readings. In the first assignment, they provided pretty similar results but as you can see this is a much clearer map than the sonar which picks up all sorts of readings even outside the arena itself.

There are some parts of the map which are not visible here – I will discuss them here. The obstacles are missing parts. This is due to the fact the robot doesn’t actually see them at all. Not with the front and side lasers. This is not a major problem as we can easily fill in the gaps, and it does not hinder the robots path anyways. The end box is missing its left side. This is because the robot is coming diagonally in to that area, so the North West side of the box is covering that part of the end box. The other side (right side) of the end box is visible since the robot’s diagonal right laser can see that part easily before it stops in the goal.



Code of my program –

#include <stdio.h>

#include <stdlib.h>

#include "Aria.h"

#include <list>

#include <iostream>

#include <math.h>

//Sensors and robot things, file readings variable

ArRobot robot;

ArSensorReading \*sonarReadingLeft, \*sonarReadingFrontLeft, \*sonarReadingFrontRight, \*sonarReadingRight;

ArLaser \*myLaser;

ArGripper \*myGripper = NULL;

bool gotTheBall = false;

FILE \*fpData;

//variables for distance and speed and angles

//distance is in milimeters

//speed is in milimeters per second

int distance\_620mm=620, distance\_580mm=580, distance\_1800mm=1800, distance\_200mm=200, distance\_600mm=600, distance\_1400mm=1400;

int speed\_70mmps=70, speed\_60mmps=60, speed\_0mmps=0, speed\_90mmps=90, speed\_120mmps=120, speed\_40mmps=40;

int angle\_n1\_degrees=-1, angle\_1\_degrees=1, angle\_n85\_degrees=-85, angle\_85\_degrees=85, angle\_n90\_degrees=-90, angle\_90\_degrees=90, angle\_40\_degrees=40, angle\_45\_degrees=45, angle\_n40\_degrees=-40, angle\_n45\_degrees=-45;

//variables for speed of the wheels, laser distance

double leftWheel = 100, rightWheel = 100, angle1, angle2, angle3, angle4, angle5, distFront, distRight, distLeft, distRightDiag, distLeftDiag;

//variables to save sonar map data and laser map data

double sonarX, sonarY;

double distLeftX, distLeftY, distFrontX, distFrontY, distRightX, distRightY;

int pi\_angle = 180 \* M\_PI;

// The camera (Cannon VC-C4).

ArVCC4 vcc4(&robot);

// ACTS, for tracking blobs of color.

ArACTS\_1\_2 acts;

// This is the chase function that follows a ball

// Uses program called ACTS to identify blobs

// The blobs trained are orange particles on the camera

// Robot also grabs the ball with the ArGripper

class Chase

{

public:

// Constructor.

Chase(ArRobot \*robot, ArACTS\_1\_2 \*acts);

// Destructor.

~Chase(void);

// The chase action.

void ChaseAction();

// Height and width of pixels from frame-grabber.

enum {

WIDTH = 160,

HEIGHT = 120

};

protected:

ArRobot \*myRobot;

ArACTS\_1\_2 \*myActs;

int myChannel;

};

// Constructor: Initialize the chase action.

Chase::Chase(ArRobot \*robot, ArACTS\_1\_2 \*acts)

{

myRobot = robot;

myActs = acts;

myChannel = 1;

}

// Destructor.

Chase::~Chase(void) {}

// The chase action.

void Chase::ChaseAction()

{

ArACTSBlob blob;

ArACTSBlob largestBlob;

int numberOfBlobs;

int blobArea = 10;

double xRel, yRel;

bool flag = false;

numberOfBlobs = myActs->getNumBlobs(myChannel);

// Get largest blob.

if (numberOfBlobs != 0)

{

for (int i = 0; i < numberOfBlobs; i++)

{

myActs->getBlob(myChannel, i + 1, &blob);

if (blob.getArea() > blobArea)

{

flag = true;

blobArea = blob.getArea();

largestBlob = blob;

}

}

}

if (flag == true)

{

//Slow robot down for precision

leftWheel = speed\_40mmps, rightWheel = speed\_40mmps;

//Open/Close gripper depending on whether there is an item inbetween the inner gripper

if (myGripper->getBreakBeamState() == 0) {

myGripper->gripperDeploy();

}

if (myGripper->getBreakBeamState() == 1 || myGripper->getBreakBeamState() == 2) {

myGripper->gripperStore();

gotTheBall = true;

std::cout << "got the ball \n";

}

// Determine where the largest blob's center of gravity is relative to the center of the camera and adjust xRel.

xRel = (double)(largestBlob.getXCG() - WIDTH / 2.0) / (double)WIDTH - 0.20;

yRel = (double)(largestBlob.getYCG() - HEIGHT / 2.0) / (double)HEIGHT;

// Set the heading and velocity for the robot.

if (ArMath::fabs(xRel) < .1)

{

myRobot->setDeltaHeading(0);

}

else

{

if (ArMath::fabs(xRel) <= 1)

myRobot->setDeltaHeading(-xRel \* 5);

else if (-xRel > 0)

myRobot->setDeltaHeading(5);

else

myRobot->setDeltaHeading(-5);

}

}

}

// Use the Chase class defined above to declare an object named chase.

Chase chase(&robot, &acts);

void update(void);

ArGlobalFunctor updateCB(&update);

void update(void)

{

// Set the velocity of the wheels.

robot.setVel2(leftWheel, rightWheel);

// Get sonar readings.

sonarReadingLeft = robot.getSonarReading(0);

//0 is left sonar

sonarReadingFrontLeft = robot.getSonarReading(3);

//3 is front (littlebit to left) sonar

sonarReadingFrontRight = robot.getSonarReading(4);

//4 is front (littlebit to right) sonar

sonarReadingRight = robot.getSonarReading(7);

//7 is right sonar

// Get laser reading. Self explanatory left right front and diagonal lasers.

distFront = myLaser->currentReadingPolar(angle\_n1\_degrees, angle\_1\_degrees, &angle1);

distRight = myLaser->currentReadingPolar(angle\_n90\_degrees, angle\_n85\_degrees, &angle2);

distLeft = myLaser->currentReadingPolar(angle\_85\_degrees, angle\_90\_degrees, &angle3);

distRightDiag = myLaser->currentReadingPolar(angle\_n45\_degrees, angle\_n40\_degrees, &angle4);

distLeftDiag = myLaser->currentReadingPolar(angle\_40\_degrees, angle\_45\_degrees, &angle5);

//---------------------------------- MAIN CODE FOR NAVIGATION

//sense the left wall and stay close to it (IDEAL distance: 600)

//[if left wall is certain distance]AND[we are not turning]

if (distLeft > distance\_620mm && !firstPairReached) {

leftWheel = speed\_60mmps, rightWheel = speed\_70mmps;

}

if (distLeft < distance\_580mm && !firstPairReached) {

leftWheel = speed\_70mmps, rightWheel = speed\_60mmps;

}

//turn the robot right when you see pillars

if (distRight < distance\_1800mm && distRight > distance\_200mm) {

firstPairReached = true;

leftWheel = speed\_120mmps, rightWheel = speed\_90mmps;

}

//maximum left and right diagonal distance from robot to pillar is 1400

//if robot is already turning, and see the first two columns, go straight again

if (firstPairReached && distRightDiag < distance\_1400mm) {

leftWheel = speed\_70mmps; rightWheel = speed\_70mmps;

}

//  Call chase the ball function, if we don't already have the ball

if (!gotTheBall) {

chase.ChaseAction();

}

// If we do have the ball in our gripper turn north towards the goal

// If the front wall is closer than 700mms, stop the wheels

else {

leftWheel = speed\_40mmps, rightWheel = speed\_60mmps;

}

if (distFront < distance\_600mm ) {

leftWheel = speed\_0mmps, rightWheel = speed\_0mmps;

myGripper->gripperDeploy();

}

//------------------------CALCULATIONS MAP READINGS

//Laser calculations

//all 3 steps to save processing power

distLeftX = (distLeft \* cos(angle3 / pi\_angle) + myLaser->getSensorPositionX()) \* cos(robot.getTh() / pi\_angle) - (distLeft  \* sin(angle3 / pi\_angle) + myLaser->getSensorPositionY()) \*sin(robot.getTh() / pi\_angle) + robot.getX();

distLeftY = (distLeft  \* cos(angle3 / pi\_angle) + myLaser->getSensorPositionX()) \* sin(robot.getTh() / pi\_angle) + (distLeft  \* sin(angle3 / pi\_angle) + myLaser->getSensorPositionY()) \*cos(robot.getTh() / pi\_angle) + robot.getY();

distFrontX = (distFront \* cos(angle1 / pi\_angle) + myLaser->getSensorPositionX()) \* cos(robot.getTh() / pi\_angle) - (distFront \* sin(angle1 / pi\_angle) + myLaser->getSensorPositionY()) \*sin(robot.getTh() / pi\_angle) + robot.getX();

distFrontY = (distFront \* cos(angle1 / pi\_angle) + myLaser->getSensorPositionX()) \* sin(robot.getTh() / pi\_angle) + (distFront \* sin(angle1 / pi\_angle) + myLaser->getSensorPositionY()) \*cos(robot.getTh() / pi\_angle) + robot.getY();

distRightX = (distRight \* cos(angle2 / pi\_angle) + myLaser->getSensorPositionX()) \* cos(robot.getTh() / pi\_angle) - (distRight \* sin(angle2 / pi\_angle) + myLaser->getSensorPositionY()) \*sin(robot.getTh() / pi\_angle) + robot.getX();

distRightY = (distRight \* cos(angle2 / pi\_angle) + myLaser->getSensorPositionX()) \* sin(robot.getTh() / pi\_angle) + (distRight \* sin(angle2 / pi\_angle) + myLaser->getSensorPositionY()) \*cos(robot.getTh() / pi\_angle) + robot.getY();

// Open a file to store data about robot coordinates, speed and obstacle / map calculations

fpData = fopen("Data.txt", "a");

if (fpData == NULL)

std::cout << "File cannot be opened." << std::endl;

fprintf(fpData, "%.2f; %.2f; %.2f; %.2f; %.2f; %.2f; %.2f; %.2f; %.2f; %.2f;", robot.getX(), robot.getY(), leftWheel, rightWheel, distLeftX, distLeftY, distFrontX, distFrontY, distRightX, distRightY);

//for every 2 sonars (from sonar0 to sonar7) save map data

for (int i = 0; i<8; i = i++)

{

sonarX =

//Calculating X according to the formula (3 steps in notes)

/\*Step 1 xP->\*/ (robot.getSonarReading(i)->getRange() \* cos(robot.getSonarReading(i)->getSensorTh() / pi\_angle) + robot.getSonarReading(i)->getSensorX())

/\*Step 2 cosRob->\*/\* cos(robot.getTh() / pi\_angle) -

/\*Step 1 yP->\*/ (robot.getSonarReading(i)->getRange() \* sin(robot.getSonarReading(i)->getSensorTh() / pi\_angle) + robot.getSonarReading(i)->getSensorY())

/\*Step 2 sinRob->\*/ \*sin(robot.getTh() / pi\_angle)

/\*Step 3Dist->\*/ + robot.getX();

fprintf(fpData, " %.2f; ", sonarX);

sonarY =

//Calculating Y according to the formula (3 steps in notes)

/\*Step 1 xP->\*/ (robot.getSonarReading(i)->getRange() \* cos(robot.getSonarReading(i)->getSensorTh() / pi\_angle) + robot.getSonarReading(i)->getSensorX())

/\*Step 2 sinRob->\*/\* sin(robot.getTh() / pi\_angle) +

/\*Step 1 yP->\*/ (robot.getSonarReading(i)->getRange() \* sin(robot.getSonarReading(i)->getSensorTh() / pi\_angle) + robot.getSonarReading(i)->getSensorY())

/\*Step 2 cosRob->\*/ \*cos(robot.getTh() / pi\_angle)

/\*Step 3 Dist->\*/ + robot.getY();

fprintf(fpData, " %.2f; ", sonarY);

}

fprintf(fpData, "\n");

fclose(fpData); // Close the file.

}

// --------------------------MAIN FUNCTION

int main(int argc, char \*\*argv)

{

// Initialisation

Aria::init();

// Open a connection to ACTS.

acts.openPort(&robot);

// Initialize the camera.

vcc4.init();

// Wait for a little while.

ArUtil::sleep(2000);

ArArgumentParser argParser(&argc, argv);

argParser.loadDefaultArguments();

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);

}

}

//Add Gripper

myGripper = new ArGripper(&robot);

// Trigger argument parsing

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

{

Aria::logOptions();

Aria::exit(1);

}

// Add sonar.

ArSonarDevice sonar;

robot.addRangeDevice(&sonar);

// Connect laser.

if (!laserConnector.connectLasers())

{

ArLog::log(ArLog::Terse, "Could not connect to configured laser.");

Aria::logOptions();

Aria::exit(1);

}

myLaser = robot.findLaser(1);

// Left these in below though I am not using them

// Code for setting the robot’s position to north for instance

// seems to glitch out the robot and we can’t change wheel-speed after

ArPose space(3800, 3500, 180); // Initial robot's odometry.

robot.moveTo(space); //Moves the robot's idea of its position to this position.

// Tilt the camera down 45 degrees to make it find the ball easier.

// Only 45 degrees seems to work. Camera doesn’t seem to move lower/higher

vcc4.tilt(-45);

ArUtil::sleep(1000);

// turn on the motors, turn off amigobot sounds

robot.enableMotors();

robot.addUserTask("update", 50, &updateCB);

//robot.setCycleTime(100);

robot.runAsync(true);

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

robot.waitForRunExit();

Aria::exit(0);

}