COSC343 Assignment 1 - Robots on a Chessboard

Group 12: Alexis Barltrop, Meaghan Bradshaw, Thomas Youngson, & Natalie Grattan.

Algorithms

Firstly, we split the task down into different components in order to create smaller and more manageable tasks. The task was already split into two stages, making it easier to divide the work up. We identified the main tasks: making the robot move a set distance, sensing black tiles and playing a sound to indicate so, counting those tiles as it goes over a number of them, making the robot turn a specified number of degrees and sensing an object using sonar. If we managed to get these tasks correctly implemented, it would then simply require a main method and some logic statements to complete all the assigned tasks in the correct order. However, in practise we encountered various issues with both our understanding of what the task was asking us to do and the robot not responding as expected to our programs.

COSC343Assignment1Library

We created a library file containing a number of functions: moveDistance(), turnClockwise(), turnAntiClockwise(), correctDirection(), turnNinetyCorrectly(), and a number of boolean functions: isWhite(), isBlack(), isGrey(), isBlackOrWhite(). This allowed us to keep every method we needed in a convenient place and add an #include "COSC343Assignment1Library.c" statement in our assignment file.

moveDistance

The *moveDistance* algorithm was designed to make the robot move forward a certain number of centimeters. It takes two parameters: the distance desired and the motor speed. Originally, the algorithm only took the one parameter: distance. However, we added in motor speed so that you could test the robot efficiently at a variety of different speeds. For example, we found that keeping the robot on the black and white squares required a slower speed than the robot going towards the tower on the finish square (though it had to be slowed down when using the sonar detector). We used the motor encoder to ensure the robot's movement relied upon the number of ticks of the wheel. In order to move a specific distance using the motor encoder, we needed to calculate how far the robot moved per encoder tick. After one rotation, the robot should move forward by the circumference of the wheel (approximately 17.6 cm). We divide the distance inputted by the circumference of the wheel, then multiply this by 360 (full rotation of a wheel) for the number of ticks the wheels should move. Using the motor encoder is a more accurate way of measuring the distance rather than just setting the power for the motor of each wheel. Factors such as battery can influence the power of the motors and because our program needs to work on any robot at any level of battery power, the motor encoder was a superior way of moving a particular distance.

turnClockwise and *turnAntiClockwise*

The turns were split into two functions: clockwise and anticlockwise. We could have done one function, however two functions made the code look tidier. The *turnClockwise* and *turnAntiClockwise* algorithms were designed to make the robot turn a certain number of degrees. They both take one parameter: the number of degrees you wish the robot to turn by. We made use of the motor encoder again for the same reasons as in the *moveDistance* function, so that the accuracy of the turn was not as influenced by factors we cannot control. In a quarter turn, we found that for each wheel there are 160 ticks in a 90-degree turn. The algorithm converts the input parameter specifying the number of degrees to the number of ticks required for that rotation. The rotation is performed by moving the left wheel forward and the right wheel backwards. For anticlockwise, we move the right wheel forward and the left wheel backwards.

correctDirection

This function takes no parameters and is void so does not return any outputs. The purpose of this function was to find the direction of the nearest object using the sonar detector and have the robot turn in this direction. The algorithm works by rotating the robot 360 degrees in 30 degree steps. At each step, the robot uses the sonar sensor to sense for an object nearby. We then use an array (distanceArray) to store all of the distances. Then, we find the minimum distance (the closest object) by running a for loop and comparing every item in the array to find the minimum distance. The variable 'location' stores the index of the minimum distance. We then run another for loop to calculate the direction the robot needs to turn in order to be facing the closest object using an array to store all the possible directions. We then use 'location' to find the direction of the closest object and use *turnClockwise* to program the robot to turn this way. We created this function so that the robot would not drive off in the wrong distance and then sense the wall or one of the shelves as being the tower. We also needed the process of finding the correct distance to be a separate function from the main method, in case we needed to call it more than once. We realised early on that we could not distinguish between objects, meaning it would be very important for the robot to find the location of the correct object.

magicFunction

This function used the mathematical absolute equation y = 1 - (abs((float)(SensorValue[colorSensor] - xShift) / slope)). The equation uses the value of the colour sensor in order to figure out what colour tile the robot is on and returns a value based on this, which is then implemented in the robot's path correction on the black and white tiles. The *PathCorrectionAlongBlackAndWhiteTiles* task calls to this function to make an adjustment to the speed of each wheel as the robot moves over the black and white tiles. The function returns a value between -1 and 1 depending on whether the robot is on the grey tiles or the black and white tiles. When the robot is on the grey tiles, the function will return a positive value, depending on how far the robot is on a grey tile. When the robot is on the black and white tiles, the function will return a negative value.

turnNinetyCorrectly

This function takes no parameters and does not return any outputs. However, it calls to the *turnClockwise* function and gives input to this function. The purpose of the *turnNinetyCorrectly* was so that after the robot had performed task 1, it would turn more accurately towards the tower because prior to this function, we had problems with the robot turning too far towards the elevator and having the robot go towards the wall instead of the tower. The function does this by using the equation from the *magicFunction* to determine if the robot is on the grey line. If the robot is on the grey line, it should rotate slightly more but slightly less if on the black or white tile.

TileCount

We created another task called *TileCount*. This was so that the *TileCount* could be running at the same time as the *PathCorrectionAlongBlackAndWhiteTiles* task which performs the error correcting for movement along the chess board. The function of this task is to keep track of the colour of the tiles the robot is on and to increment the count when the robot first senses a black tile. To do this, we used boolean variables and statements. We created two boolean variables called *isTileWhite* and *isTileBlack* and set both to false to begin with. Then in *COSC343Assignment1Library* we created boolean functions, which return the colour sensor values for the grey tiles, black tiles and white tiles. Therefore in our *TileCount* task, we could just refer to these rather than have to write out the full if statement every single time we wanted to refer to this condition. We then set the booleans to check if the robot is on a white tile. If both booleans are false, then this must mean the tile is black so we set the booleans to reflect this, play the sound and increase the count. Also, in our main method file we used two constants for the black and white thresholds (reflectance on black and white tiles) so that we did not keep needing to update these throughout the file.

sonarTargetFinding

This task was designed to run the *correctDirection* function. While the touch sensors were 0 (meaning they were not touching anything, such as the tower) the *correctDirection* function was ran and then the robot moves forward until it connects with the tower.

PathCorrectionAlongBlackAndWhiteTiles

When trying to accomplish path correction, we tried many different approaches. We also had the path correction as a separate function. However, in our final approach we found that having it as a task was the best way because then we could have it running at the same time as the main method. The basic theory behind this algorithm was that it would follow the grey line to the left of the black and white tiles. We set the colour sensor to reflective mode, but used a wide range in order to adjust for different light levels. We had three variables: baseSpeed, biasB and biasC. These were each set to specific speeds. Finding the right combination of speeds for these three variables was very important in order to get the robot functioning correctly. Also, the correct range for the white and black thresholds was very important. This is because the colours of the tiles that the robot senses change based on the light, so an overcast day might make the tiles seem more grey according to the colour sensor. With these variables, the algorithm sets

the speed of each motor. For the left motor, the speed is baseSpeed + biasB*diff (this is the result of *magicFunction*). For the right motor, the speed is baseSpeed - biasC*diff. If the result of *magicFunction* is positive, the robot moves to the right and if it is negative, the robot moves to the left. The bias is added so the left wheel is more sensitive to the function, which means the robot spends more time on the black and white tiles rather than the grey.

Main

Our main method basically involved joining all of our functions and starting the tasks we designed in order to complete the assignment specifications. Firstly, the robot moved from the start tile to the line of black and white tiles, using the *moveDistance* method. Then, it did a 90 degree turn to face the elevator, using the *turnClockwise* method. The robot then moved along the tiles, counting 15 black tiles, using the *TileCount* task and *PathCorrectionAlongBlackAndWhiteTiles* task. It then turned to face the tower, using the *turnNinetyCorrectly* method. The robot then moved forward a certain distance so it could be in range of the tower for the sonar to start detecting, using the *moveDistance* method. The sonar then detects the tower and moves towards it, using the *SonarTargetFinding* task which makes use of the *correctDirection* method. The robot then pushes the tower off the finish square and moves to sit on this square, using the *moveDistance* method.

Problems

The first problem that we ran into was understanding what the brief required of us. This is where having a group was particularly useful because discussing the brief as a group meant that if one person did not understand a part of the brief, it was likely someone else did and they could explain it. Another common problem that arose for all of our coding was finding the right speed and the right places to use wait functions. We found that when the robot was going at full speed (100) it would veer to the left. However, if it was going 99 this was not an issue. Working in a group, we ran into the usual coding issue of having multiple different files and we were working on multiple computers. This meant the code could get messy, with many different versions around. We solved this issue by using Google Drive to upload all our files so we could easily transfer them from different computers.

turnClockwise & turnAntiClockwise

The main problem encountered when creating the *turnClockwise* and *turnAntiClockwise* functions was trying to find how many ticks were in a 90 degree turn. At first, there was confusion because of division using integers was not giving us the results we anticipated. This was easily fixed by converting the numbers to a float before the division took place. Another issue was figuring out how exactly the turn would occur. At first, the turn had only one wheel going forward but this did not work for turning around on the spot. So the function was rewritten to have one wheel going forward and one wheel going backwards. This allowed the robot to turn on the spot. The function then worked smoothly and the next problems came not with the function itself, but implementing it correctly in the main method with the wait statements in the correct places.

correctDirection

The main difficulty with this task was the fact there were other objects around. When we initially implemented this algorithm, we found that sometimes the turn would be slightly off so the robot would be facing towards the elevator more than the tower. Thus, the robot would often go towards the elevator and detect the elevator as the nearest object. This made us realise that we had to get the turn as accurate as possible, but this was difficult as our algorithm for the robot moving along the black and white tiles meant that the robot did not move in a straight line. Therefore, we introduced the *turnNinetyCorrectly* function, which reused the mathematical function from path correction. This meant that if the robot was on the grey line, it should rotate more and less if it was on the black and white tiles. Once we did this, the robot was always roughly facing towards the tower.

TileCount & PathCorrectionAlongBlackAndWhiteTiles

One of the biggest challenges we overcame while working on this assignment was getting the robot to travel over the 15 black tiles and not divert off path. One of our initial approaches was to not have error correction at all. We tried to get the turn off the start tile as accurate as possible, so the robot would simply just travel in a straight line over the tiles and count each black one. However, no matter how accurate the turn was, the robot would still divert at some point.

The next idea we had was for the robot to check if it was on the grey, speckled tiles (using the light sensor). If the robot was on the speckled tiles, it would mean it was off path. However, the issue we had with this was the the range of the speckled tiles was 11-52 of light reflectivity and the range of the grey lines between the black and white squares falls into this range. This meant when we tried to get the robot to correct by sensing for a speckled tile, it would see the grey line between the black and white squares as another speckled tile, so would correct then also when we did not need it to. This meant it would not count the black squares properly and would continue to divert from them.

To solve this error we tried to get the robot to move forward when it encountered a square in the range of 10-60 and then check again for the colour. The logic behind this was that if it moved forward slightly, if it had been the grey line between the white and black squares that sent the robot into this loop, it would exit the loop because it would now be on a white or black square and therefore would not enter the error correcting process (we created a weave where the robot would turn the opposite way to the direction it was currently going in). This did not go quite to plan and made the code quite complex and was still not working perfectly.

The next approach we tried (and one we used in our final algorithm) was actually using the grey lines between the black and white squares. We used an absolute mathematical function to calculate how far on the grey the robot was. Based on this, we altered the speed of the motors and how far the robot would turn. This kept the robot on path. At first, we found that the robot would get about 4 black squares in and then would go off path and begin to spin in a circle. To fix this, we altered the coordinates in the function and also the particular combinations of speeds for the variables in the *PathCorrectionAlongBlackAndWhiteTiles* task. Once we found the right combination, the robot completed the first part of the task correctly.

After fixing this, the next issue we ran into was when the robot would travel along the black and white squares, it did not count every black tile. We soon realised that this was because due to our algorithm and the sequence of boolean statements we had used, the robot needed to go over a white tile before it would add to the black tile count and play the sound for a black tile. This should not have been an issue, but after taking colour readings of the problem tiles, we realised the robot would be reading the white tiles prior to the problem black tiles as grey due to the darker lighting. We therefore made our white threshold range wider so it would include these tiles and this fixed the problem.

correctDirection & sonarTargetFinding

The first approach to get the sonar function to work was the robot detecting at three angles, 60 degrees forward, 60 degrees to the left and 60 degrees to the right. However, this did not work all the time. Therefore, we changed this so the robot would be detecting in a 360 angle so it would detect in a full circle. Another problem with the sonar function was using our pre-existing turn functions (*turnClockwise* and *turnAntiClockwise*). The turn function worked perfectly for the turn onto the start tile at the beginning and for the 90 degree turn (though we added the *turnNinetyCorrectly* function so that the the robot turned more if it was on the grey line) but it did not work as well for the 360 degree scan. Therefore, we had to create a variable called ‘rotationVariable’, which added a few degrees to where the sonar sensor was scanning.

Overall, our robot completed the tasks that were assigned. The important principle was testing the robot frequently when we made any changes to the code so that we could pinpoint where the errors were occurring. We made our code as clear and tidy as possible while still fulfilling the assignment briefs and worked together well as a team.

Appendix

COSC343AssignmentLibrary.c

```
/*    Moves the robot forward a specified distance at a specified motor speed.
*     Each full rotation of the wheel moves the robot roughly 17.6 cm.
*     Motor encoders monitor the rotations and hence the distance to move.
*     @param distanceCM - the distance in cm to move.
*     @param motorSpeed - the motor speed, 0 - 100.Negative for reverse.
*     @author Alexis Barltrop
*/
void moveDistance(int distanceCM,int motorSpeed){
      float distancePerRotation = 17.6; // distance in cm.
      float noRotations = ((float)distanceCM)/(distancePerRotation);
      nMotorEncoder[motorB] = 0;
      nMotorEncoder[rightMotor] = 0;
      while (abs(nMotorEncoder[leftMotor])<(noRotations*360)){
            motor[leftMotor] = motorSpeed;
            motor[rightMotor] = motorSpeed;
      }
      motor[leftMotor] = 0;
      motor[rightMotor] = 0;
}

/*    Rotates the robot clockwise.
*     Wheels move in opposite direction so that the robot turns on the spot.
*     Number of ticks per degree callibrated with 90 degree turn.
*     Has related function turnAntiClockwise.
*     @param turndegree - the number of degrees to rotate the robot sideways
*     @author Alexis Barltrop
*/
void turnClockwise(int turndegree){
      float ticksPerDegree = ((float)160)/90;
      int noEncoderTicks = ((float)turndegree)*ticksPerDegree;
      nMotorEncoder[leftMotor] = 0;
      nMotorEncoder[rightMotor] = 0;


while(nMotorEncoder[leftMotor]<noEncoderTicks&&nMotorEncoder[rightMotor]>-noEnc
oderTicks){
            motor[leftMotor] = 50;
            motor[rightMotor]= -50;
      }
      motor[leftMotor] = 0;
      motor[rightMotor] = 0;
}
```

```
/*    Rotates the robot anticlockwise.
*     See related function turnClockwise.
*     @param turndegree - the number of degrees to rotate the robot
*     @author Alexis Barltrop
*/
void turnAntiClockwise(int turndegree){
      float ticksPerDegree = 160/90;
      int noEncoderTicks = ((float)turndegree) / ticksPerDegree;
      nMotorEncoder[leftMotor] = 0;
      nMotorEncoder[rightMotor] = 0;
      //     setMotorSyncEncoder(leftTouchSensor,     rightTouchSensor,     1,
noEncoderTicks, -1);


while(nMotorEncoder[rightMotor]<noEncoderTicks&&nMotorEncoder[leftMotor]>-noEnc
oderTicks){
            motor[leftMotor] = -50;
            motor[rightMotor]= 50;
      }
      motor[leftMotor] = 0;
      motor[rightMotor] = 0;
}

/*    Correct Direction Method for Sonar
*     Method for Sonar Detection Direction Correction.
*     Rotates robot in steps of 30 degrees, taking a distance measurement
*     with the sonar sensor at each step.
*     Rotates to face the closest object based on these measurements.
*/
void correctDirection(){
      // Creates array containing distance measurments.
      int rotate = 0;
      int count = 0;
      int step = 30;
      int size = 360/step;
      int distanceArray[12];
      while (rotate < 360){
            turnClockwise(step);
            wait1Msec(100);
            distanceArray[count]= SensorValue[sonarSensor];
            wait1Msec(100);
            rotate =rotate+step;
            count++;
      }
      turnClockwise(step);

      // Finds location of minimum distance in array.
      int minimumDistance;
```

```
	int location = 0;
	minimumDistance = distanceArray[0];
	for ( int i = 1 ; i <size ; i++ ){
		if ( distanceArray[i] < minimumDistance ){
			minimumDistance = distanceArray[i];
			location = i;
		}
	}

	// Creates array containing turning directions.
	int directionArray[12];
	for (int i = 0; i<(size); i++) {
		directionArray[i] = 0 + (i*step);
	}

	// Turns to face closest object.
	int rotationError = 10;
	turnClockwise(directionArray[location]+rotationError);
}

/**
* Mathematical equation used for line correction.
* See comments for task PathCorrectionAlongBlackAndWhiteTiles.
* Equation is an absolute function designed to have a maximum of one,
* and to intercept the x-axis at the WHITETHRESHOLD and BLACKTHRESHOLD.
* @author Alexis Barltrop
*/
float magicFunction(){
	int xShift = (BLACKTHRESHOLD+WHITETHRESHOLD)/2;
	int slope = xShift-BLACKTHRESHOLD;
	float y = 1 - (abs((float)(SensorValue[colorSensor] - xShift) / slope));
	return y;
}

/**
* Used to turn 90 degrees after tile count in part one is complete.
* Reuses the mathematical function from path correction to determine if,
* and how much the robot is on the grey line.
* Should rotate slighty more if on grey, less if on black or white tile.
*/
void turnNinetyCorrectly(){
	turnClockwise(90+30*magicFunction());
}

/**
* Following functions return true if sensor is reading values corresponding to
the relevant tile.
```

```
* WHITETHRESHOLD and BLACKTHRESHOLD are declared before main task in program
file.
*/
bool isWhite(){
      return (SensorValue[colorSensor]> WHITETHRESHOLD);
}

bool isBlack(){
      return (SensorValue[colorSensor]< BLACKTHRESHOLD);
}

bool isGrey(){
      return      (SensorValue(colorSensor)      >      BLACKTHRESHOLD      &&
SensorValue(colorSensor) < WHITETHRESHOLD);
}

bool isBlackOrWhite(){
      return      (SensorValue(colorSensor)      <      BLACKTHRESHOLD      ||
SensorValue(colorSensor) > WHITETHRESHOLD);
}
```

AssignmentFile.c

```
#pragma config(Sensor, S1,     leftTouchSensor,    sensorEV3_Touch)
#pragma config(Sensor, S2,     rightTouchSensor,     sensorEV3_Touch)
#pragma config(Sensor, S3,     colorSensor,    sensorEV3_Color)
#pragma config(Sensor, S4,     sonarSensor,    sensorEV3_Ultrasonic)
#pragma  config(Motor,   motorB,               leftMotor,       tmotorEV3_Large,
PIDControl, driveLeft, encoder)
#pragma  config(Motor,   motorC,               rightMotor,      tmotorEV3_Large,
PIDControl, driveRight, encoder)
//*!!Code   automatically   generated   by   'ROBOTC'   configuration   wizard
!!*//

/*    Constants contain threshold values for reflectance on black and white
tiles. */
#define BLACKTHRESHOLD 13
#define WHITETHRESHOLD 53

/*    Include the assignment library with a number of useful functions. */
#include "COSC343Assignment1Library.c"

/*    Global variable to keep track of the number of tiles the robot has gone
over. */
int tiles = 1;

/*  TileCount Task
```

```
*           Task to count the number of black tiles the robot goes over
*   in the lobby. Uses booleans to detect tiles correctly.
*     @author Meaghan Bradshaw
*/
task TileCount{
      bool isTileWhite = false;
      bool isTileBlack = false;
      // While loop to keep track of what colour tile we are on.
      while (tiles<16){
            // If we're on a white tile, set the booleans to reflect so.
            if(isWhite()){
                  isTileWhite =true;
                  isTileBlack = false;
                  } else {
                  isTileWhite = false;
            }
            // If both varibles are false enter if to determine whether we're
on a black tile.
            if (!isTileWhite&&!isTileBlack){
                  //  If  we're  on  a  black  tile  that  we  haven't  detected
previously, add to the count and reset booleans.
                  if(isBlack()) {
                        isTileBlack = true;
                        playSoundFile("Boing.rsf");
                        wait1Msec(200);
                        isTileWhite = false;
                        tiles++;
                  }
            }
      }
}

/*    Path Correction Algorithim
*     Designed to make the robot follow the grey line to the left of the black
and white tiles.
*     Color Sensor used on reflectance mode.
*     Mathematical absolute equation used to make a adjustment to the speed of
each wheel as the robot moves between tiles.
*     On grey tiles the function returns a value between 0 and 1 depending on
how far the tile is on the grey.
*     This causes the robot to move to the right.
*     On black or white tiles the funciton returns a value between 0 and -1
depending how far off the grey it is.
*     This causes the robot to move towards the left.
*     Thus the robot follows the left hand edge of the black and white tiles.
*     A bias is added to each wheel which causes the robot to swerve onto the
black and white tiles more often.
*     @author Alexis Barltrop
```

```
*/
task PathCorrectionAlongBlackAndWhiteTiles{
	int baseSpeed = 20;
	int biasB = 35;
	int biasC = 20;
	while (tiles < 16) {
		float diff = magicFunction();
		motor[leftMotor] = baseSpeed + biasB*diff;
		motor[rightMotor] = baseSpeed - biasC*diff;
	}
}

/*	Target Finding Task
*		Adjust the robot to face the closest target and moves forward.
*	Repeats this until bumpers hit something.
*/
task SonarTargetFinding{
	while(SensorValue[leftTouchSensor]==0&&SensorValue[rightTouchSensor]==0){
		correctDirection();
		moveDistance(50,100);
	}
}




/*  Main
*		Steps to carry out to complete the specifications of the
assignment.
*		1. Move from the start tile to the line of black and white tiles.
*		2. Do a 90 degree turn to face the elevator.
*		3. Move along tiles and count 15 black tiles.
*		4. Turn to face the tower and finish tile.
*		5. Move foward a distance to get within range
*		6. Detect the tower and move toward it
*		7. Push the tower off the finish tile and sit on it
*/
task main {

	// Step 1
	setSoundVolume(100);
	playSoundFile("Hello.rsf");
	wait1Msec(500);
	moveDistance(20,40);
	wait1Msec(500);

	// Step 2
	turnClockwise(90);
```

```
	wait1Msec(500);

	// Step 3
	startTask(TileCount);
	startTask(PathCorrectionAlongBlackAndWhiteTiles);

	// Stops everything once 15 tiles counted.
	bool tileLoop = true;
	while (tileLoop) {
		if (!(tiles < 16)) {
			stopTask(TileCount);
			stopTask(PathCorrectionAlongBlackAndWhiteTiles);
			motor[leftMotor] = 0;
			motor[rightMotor] = 0;
			tileLoop = false;
		}
	}

	// Yay 15 tiles!
	wait1Msec(500);
	playSound(soundUpwardTones);
	wait1Msec(500);

	//Step 4
	turnNinetyCorrectly();

	//Step 5
	moveDistance(200,100);

	// Step 6
	// Robot stops when it hits the tower.
	startTask(SonarTargetFinding);
	bool sonarLoop = true;
	while (sonarLoop) {
		if	(SensorValue[leftTouchSensor]	==	1	||
SensorValue[rightTouchSensor] == 1) {
			stopTask(SonarTargetFinding);
			motor[leftMotor] = 0;
			motor[rightMotor] = 0;
			sonarLoop =false;
		}
	}
	wait1Msec(1000);

	// Step 7
	// Final Push.
	moveDistance(30, 100);
	playSoundFile("Good job.rsf");
```

```
      wait1Msec(1000);

} // End Main.
```