343 Assignment 1 Group Report

Group 15: Maaha Ahmad (3759740)
Magdeline Huang (2824402)
Jack MacCormick (2148113)
Hayden McAlister (1663787).

1 | Strategy

Representing the environment: Our task was to write a program capable of autonomously driving a robot across the floor of the Otago University's Owheo building, and locating a "tower" within a subset of this environment. The floor of the Owheo building is displayed below *(fig1)*, it can be summarised as a set of non-uniformly tesselating black and white tiles. We abstracted the physical environment into a two dimensional Cartesian grid, where the black tiles form the grid vertices, starting where black tile 1 is *[0,0]*,. Simplifying the representation like this allowed our robot's position to be stored internally as a set of integer coordinates *[x,y]* which represent one black tile within the 15x8 layout. Returning the real world black square number is then a two variable algebraic formula explained more below.

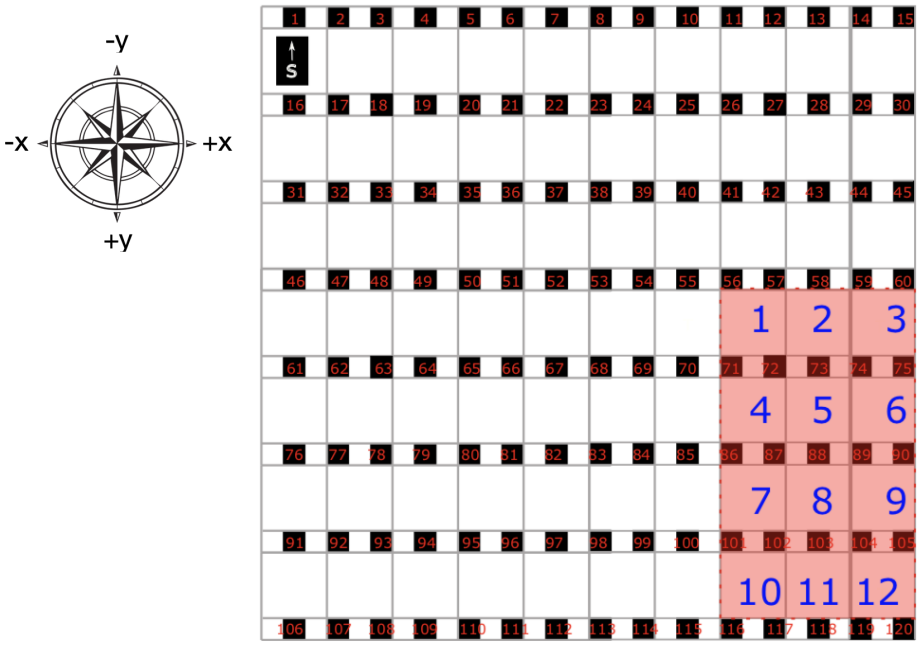


Figure 1

We decided to invert the y-axis, such that moving away from the Origin increases the value of Y, consistent with graphical representation in computing. This did mean that the robot initially faces towards *Y = 0* in it's starting orientation.

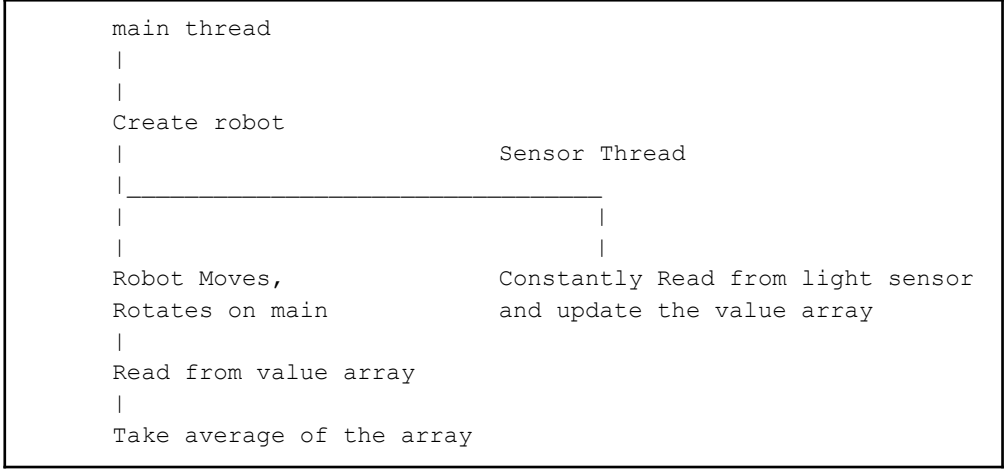
Reaching the search space: In order to reach the search space (denoted as the red area on *fig1*), we decided to program the robot to travel along the x-axis (black squares 1 to 11) then "downwards" along the y-axis (black square 11 to 56). This aligns the robot with the first column of the search space Col(A) = {1,4,7,10}. Column-wise search was selected, as searching 3 columns is faster than searching 4 rows, and requires fewer movements. Furthermore, the tower is within the larger voids between columns, not between rows markers, so the robot's direction of travel makes it more likely to cross paths with the tower.

Searching for tower: Exploring the columns left to right, from outside edge Col(A), centre Col(B) and back to outside edge Col(C), meant we could search each column without risk of accidentally detecting anything outside of the current column, and risk interpreting it as within our current column. Starting with column A, we know there is nothing to the robot's right, and nothing to it's left when in column C, as both of these form the outer perimeter of the searchable region.

2 | Algorithms

We designed our implementation around two class type objects ***Robot*** and ***BlackSquareSensor***. This allowed us to use a threading implementation, such that multiple processes can run concurrently.
This diagram describes the thread implementation. As can be seen in the diagram, since the *value array* is accessed by both threads, we have a *value_array_lock* so we can't run into race conditions, where reading and writing occur simultaneously. The small amount of time overhead from this is negligible compared to the risk of crashing due to thread problems.

Table 1: Threading Implementation



The internal storage of the robot's position within the environment is done in two parts, raw position and facing, as the robot's orientation determines how it can interact with the environment (ie, it cannot "see" behind itself, or drive sideways).
Firstly, the *direction* the robot is facing is stored as a two long array, of mutually exclusive values (0||∓1), representing a faced direction along the Cartesian grid, ie [0,+1] represents facing positive X, along parallel Y gridlines. This prevents us from having non-right angle facings, reducing the complexity of the task.
Position is a two long array of integer values *[X,Y]* which represents a vertex on the Cartesian grid, as outlined above, starting at [0,0].
As *direction* is only ever one of the set *{[0,+1],[+1,0],[0,-1],[-1,0]}*, the product of *direction*(number of move calls)* is a translation of *n* units in one Cartesian direction. P*osition* is thus the result of the previous *position*, increased by the number of *move()* calls multiplied by the value of the *direction* array. I.e. at position [0,0], facing positive X *[0,1]*, moving 6 squares, the robot's internally stored position changes to *[0,0] + 6*[1,0] = [6,0]*.

Translating the Cartesian position value back into Black Square numbers is implemented through the formula: $(x + 1) * (y * 15)$ such that when the robot is at coordinate *(0,0)* it's over black square 1, and at *(3,2)*, the second tile of the third row, it's at black square 34. Through a similar calculation we can convert from Cartesian coordinates to the blue numbers of the search space that the tower may be on.
The position of the robot, and functions to translate it are stored outside of the robot class itself, and accessed by the robot class as necessary.

Classes Implemented

1. **Robot class**

   Creating a *Robot* class allowed us to abstract a lot of the details and parameters for the robot, and thus enabled us to program an abstract method for finding the tower, without having to constantly add checks for events like crossing black squares.

   We decided to restrict the basic movement options to a) moving forward (*robot.move)* and b) turning only at right angles (*robot.rotate)*. This reduces the possible points of failure that could be induced by more complex logic, such as diagonal shifts. While saying this, we do have more complex methods that allow for correction, however these are siloed into separate callable functions, limiting them to only come into play when needed, rather than being embedded within all movements.

Table 2: Components of the robot	
left_motor	robot's reference to it's left wheel motor
right_motor	robot's reference to it's right wheel motor
tank	robot's reference to both the left and right wheel motors
touch_sensor	robot's reference to the impact detection bar at the front
ultrasonic_sensor	Range sensor that functions as the "eyes" of the robot
color_sensor	The colour sensor at the base of the robot, faces the ground
sound	Plays noise
lcd	LCD display screen
btn	Listener for if a button is pressed, can be subset to listen for specific buttons
black_square_sensor	Our subclass

   a. ***move***: This was implemented to move the robot forward like a tank (dual wheel drive) until it hits a black square, at which point, the position of the robot will be updated using the cartesian state [x,y]. This is implemented in the following way:
      i. While the reflectivity array's average is above our threshold for "black", drive is continued.
      ii. Once the average value drops below threshold, the movement is halted, and the robot report's it's position.
      iii. A correction is made to the angle of exit, to account for our angle of arrival.

   b. ***correction***: This method does a correction for any deviation after travelling between squares. It assumes that the robot did not deviate too far between black squares which means that the robot is still roughly orthogonal to the square. This also assumes that the squares are perfectly square. The method is implemented as follows:
      i. Find the angle left and right until the robot is off the square
      ii. Take the midpoint
      iii. Correct the orientation of the robot, with a tank-like turn, by the value calculated

   This should put the robot on a lesser but opposite angle compared to the one of the last trip, asymptotically cancelling the angle the robot is off by. We chose reflective asymptotical correction, as simple perpendicular correction does not correct for the physical deviation off centre the robot may have developed, but only the angle of approach, meaning the robot could slowly drift more and more sideways, into the void without noticing until it's already happened.

This behaviour difference is illustrated in *fig2*, below. The parameter that will be passed to this method is the speed at which the robot turns in degrees per second.

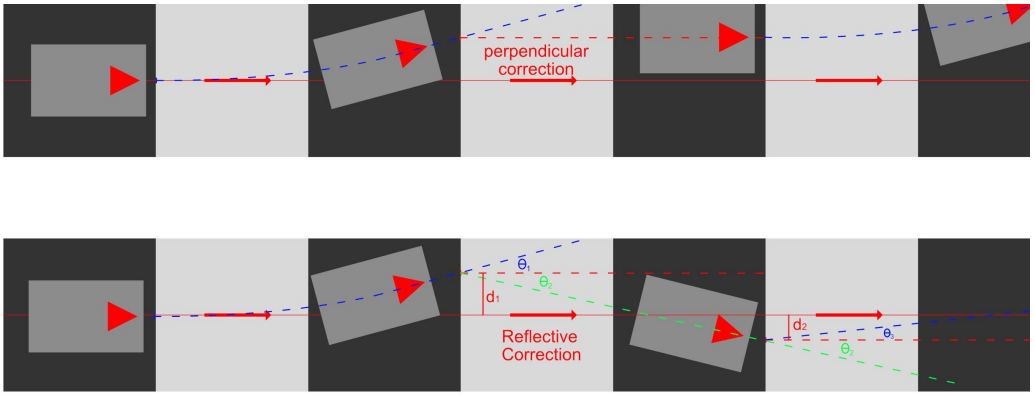


Figure 2

c. ***rotate:*** This method rotates at a multiple of 90 degrees on an axis like a tank, and updates the direction vector. We are defining positive rotation as turning clockwise. The parameters that are passed to this method are the angle the robot rotates through(as a multiple of 90 degrees) and the speed it rotates around.

d. ***report_black_square:*** Reports the black square we are on, determined from position coordinates.

e. ***report_tower:*** Report the blue number of where the tower is

f. ***check_next:*** Perform a check of the space between current position and next black square. This is very similar to the move function, but has checks for sonar proximity and touch. The parameter passed to this method is the speed at which we travel over the next space. This method returns True if the *touch* sensor is activated during the journey, or if the *sonar* sensor is triggered, and False otherwise.

g. ***check_next_number***: subtype of *check_next*, simply allows for multiple *move* methods to be called by one dynamic function.

h. ***display_text:*** Display some text on the LCD Display. The parameters passed to this method are the string to display and the font to use.

i. ***move_back***: Moves the robot back onto the last black square and does a correction. Useful for if we've driven into the void, and would otherwise never see another black tile again, and would be completely desynchronised from our internal position value

j. ***move_number***: subtype of *robot.move*, simply allows for multiple *move* methods to be called by one dynamic function.

2. BlackSquareSensor class

We created a *BlackSquareSensor* class which handles taking input from colour_sensor, and determining if we are over a black tile. Separating this process into its' own class enabled us to utilise a threading implementation, which allowed the robot to continually sense its environment on this thread, while the other governed the actuators.

BlackSquareSensor is divided into the following attributes:

Table 3: List of BlackSquareSensor attributes

VALUE_LIST	The list of values that have been read
VALUE_LIST_LOCK	A *Lock* object from the *Threading* class, so we can be sure we don't run into race conditions when writing/taking averages
CONSTANT_READ	A Boolean to check if we are to be constantly reading (*Thread* termination condition)
CURRENT_INDEX	The current index into the *VALUE_LIST* array that we are writing to
THRESHOLD	The threshold that must be surpassed to be on a white square
SENSOR	The sensor to read from
THREAD	A reference to the thread

The following is the implementation the general strategy of getting to, and then checking the search space, as described in abstract above:

1. Use *robot.move* and *robot.rotate* to get to tile 11 (0,10) and then to tile 56 (10,3).
2. Use *robot.check_next* to progress down column A of the search space (red area on *fig1*)
3. Rotate and move the robot to check column B, from below, starting at tile 113, moving upwards
4. Rotate and move the robot to check column C from above, starting at tile 60, moving downwards

3 | Problems

These were the problems we faced during development:

Problems	Solution
Wheels can be inconsistent with each other, which leads to the robot turning a wide arc when it thinks it's driving a straight line, as one wheel may rotate faster/has more grip than the other. This also affects spot turns, leading to non-90° turns	We added a correction that allows us to evaluate and adjust for the angle which we may be off from, when leaving a black tile. Rather than adjust to come out of the black tile at a perpendicular angle, we decided to do reflective angles, which progressively decrease, such that our deviation asymptotically approaches 0°. Without doing this, we could eventually drift off the side of the tile grid, into the void.
Grey lines having a reflectivity close to that of the black tiles. This meant we could not use instantaneous readings, as the grey lines would terminate robot drive methods.	We created an array of readings, to which reflectivity readings were stored, the average of which was returned, and used to determine if we were over a black square. This approach also made our sensor/check method robust to noise from tile texture and shadows on the tiles, as this noise would not draw the average down low enough to affect robot termination.
Touch sensors being non-effective for this problem. As we were not allowed to push the tower out of the square, the travel velocity of the robot, the elasticity of both the tower and the “spring” loaded touch sensor, meant that the touch sensor would either not fire long enough to trigger, or bounce the tower away from us.	As such, we changed our detection method to be using the sonar sensor, rather than using the touch sensor. To avoid too much excess noise from interfering with the sonar sensor, we kept the distance threshold fairly low, which also allows for higher accuracy.
Sonar sensor was not detecting the tower even when it was close to it because the light reflecting from the tower was bouncing away from the sensor by a wide angle.	We made the robot move left and right by small increments where it was constantly detecting the tower and we increased the distance for detection.