Autonomous Vehicles Final Report

Candidate Number: 181395

Introduction to QBot 2

In this project, we explored the capabilities and functions of the Quanser QBot 2e, an open-architecture autonomous ground robot that is equipped with sensors (seen in figure 1) and a computer vision system to detect obstacles [1]. It is programmed and powered by substantial courseware provided to us and accessible through MATLAB and Simulink. The QBot 2e was designed for teaching and is suited to indoor lab environments so it is ideal for this project. However, due to the COVID-19 pandemic, students were unable to attend the university in person and were provided with an alternative remote-working solution. Quanser Interactive Labs allowed the students to examine and research the QBot 2e remotely, without having the robot at hand.

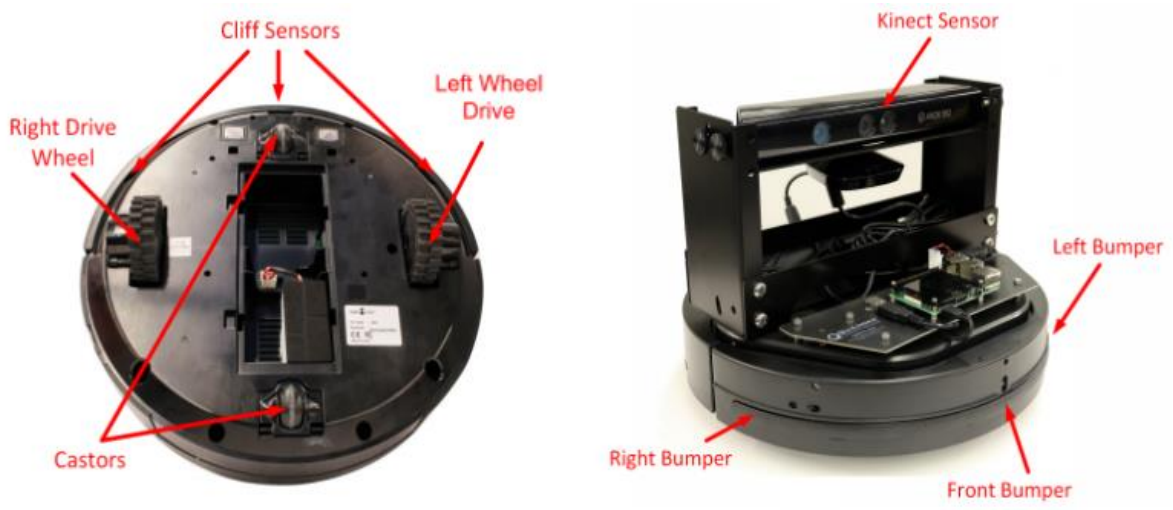


Figure 1: Main QBot 2e Hardware Components[2]

Figure 1 shows the main components in the QBot 2e, including two drive wheels on the left and right which are independently programmed to drive forwards or backwards. They are in a differential drive configuration which is common in robotics due to its simplicity in movement. The wheels movements are measured by encoders while the angle of the robot is measured through estimation with an integrated gyro. The two castors at the front and back stabilize the robot while the main sensor is a depth sensor known as a Microsoft Kinect which outputs both colour image frames and depth. Other sensors on the robot include the bump sensors, used to detect, or avoid obstacles as well as the cliff sensors used to detect when there is a large drop off on the ground. We can see the QBot 2e with all of its components in the workspace provided by Quanser Interactive Labs in figure 2.

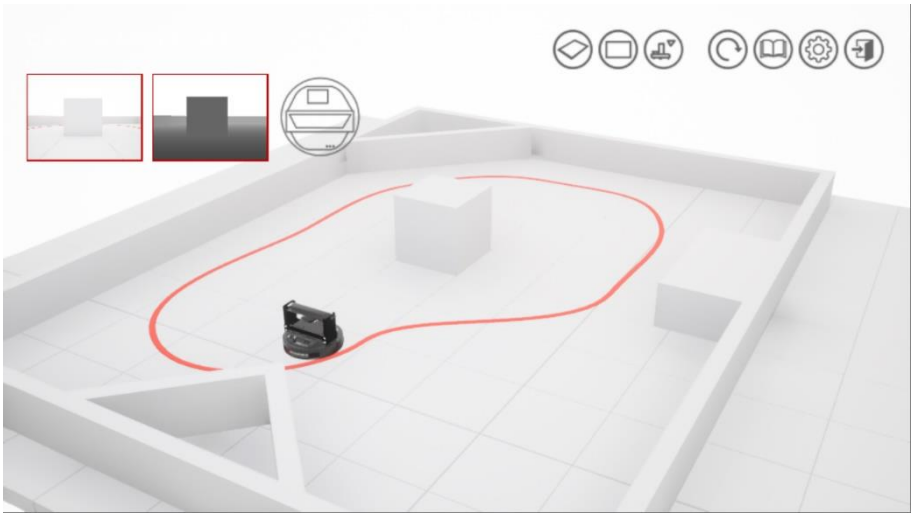


Figure 2: Quanser Interactive Labs, QBot 2e Workspace

Figure 2 shows a screenshot of the QBot 2e workspace from the software provided by Quanser called Interactive Labs. The platform allowed us to simulate the actions that the robot would take in person, in response to the programme run in Simulink. The QBot 2e workspace contains multiple obstacles for the robot to detect or avoid as well as a path used for path planning. The gridded lines seen on the floor represent squares of 50x50cm, which enables us to measure distances for future experiments. In addition, multiple orientations are provided to view the simulation from a profile or top view as well as a view constantly following the robot. These views allow better observation for certain experiments. The two types of images from the Kinect sensor, colour (RGB) and depth are seen in the top left while an image indicating when the bump sensors are active is also seen in the top left of the screenshot.

The QUARC software used for the QBot 2e integrates with MATLAB and Simulink to communicate with the robot in real-time. A variety of QUARC blocks are used in Simulink throughout this project to complete different experiments with the first experiment using a block diagram seen in figure 3.

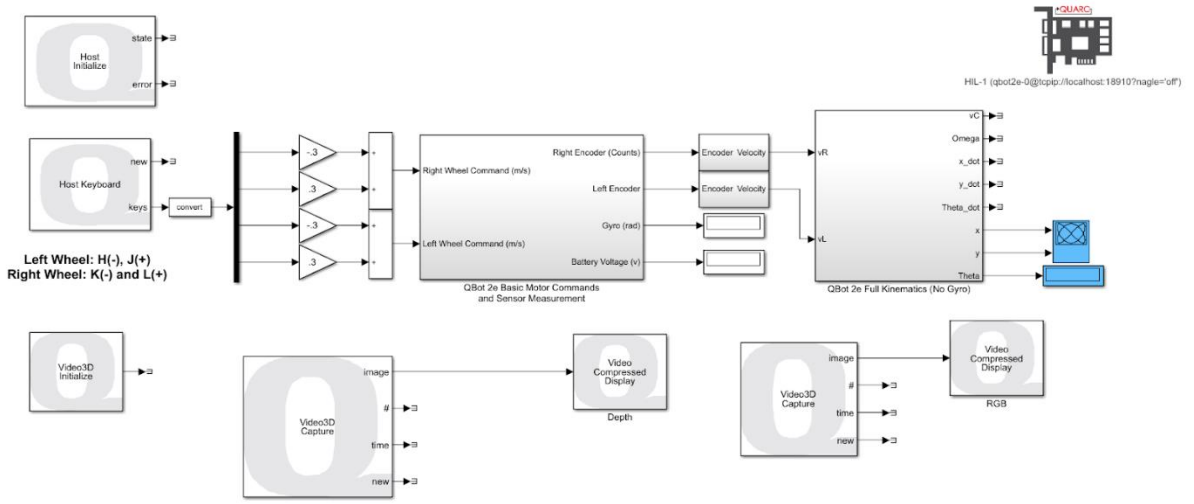


Figure 3: Snapshot of the QBot2e_Keyboard_Teleop_Wheel.mdl model

When running the Simulink block diagram, we could command the left and right wheels of the robot independently using a computer keyboard while also observing live depth and RGB images likewise to the displays in the Interactive Labs workspace. The movement commands of the robot could be described as follows:

- If J and L are pushed together, the vehicle moves straight forward. If H and K are pushed together, the vehicle moves straight backwards.
- If J is pushed, the vehicle spins clockwise about the axis of the right wheel. It spins anticlockwise if H is pushed. The same goes when K is pushed, clockwise spinning about the axis of the left wheel and anticlockwise if L is pushed.
- If J and K are pushed, the vehicle spins clockwise about the origin of the vehicle and anticlockwise if H and L are pushed.

Once all movement configurations were known, this system allowed for easy movement around the workspace. However, there are pros and cons to this system, as we realized after completing the next experiment.

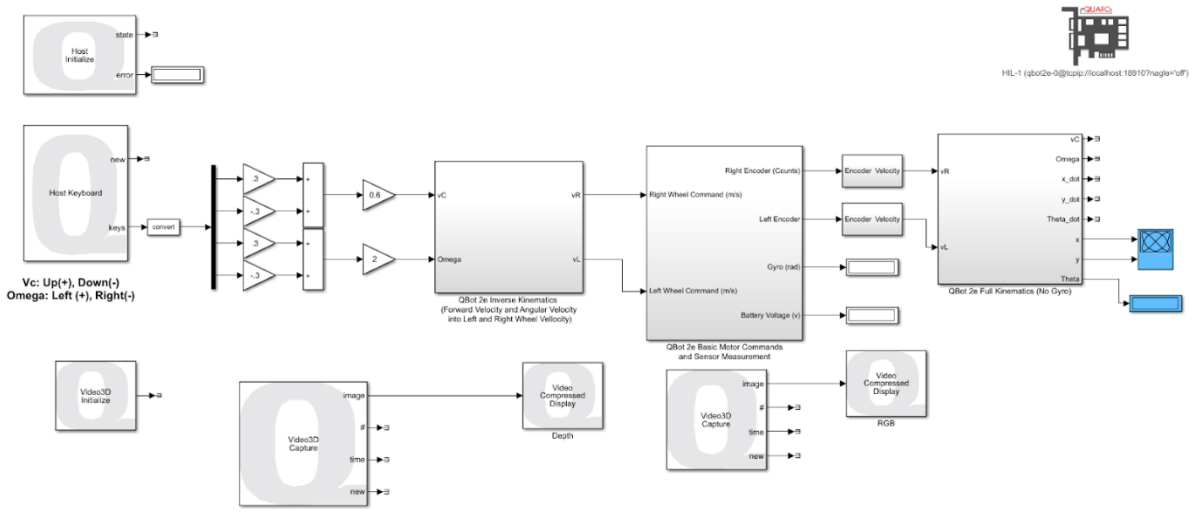


Figure 4: Snapshot of the QBot2e_Keyboard_Teleop_Normal.mdl model

Figure 4 shows the block diagram that when run, enabled us to move the robot using the computer keyboard but this time with the arrow keys. This is the normal vehicle drive mode and is very common, used in cars for example. The up arrow moves the robot forward, the down arrow backwards and the left and right arrows move the robot left and right respectively.

The normal vehicle drive method works well as only one input is needed at a time to either move straight forwards or backwards or rotate about its origin compared to two keys for the wheel drive mode. In addition, the robot can simultaneously move forwards and at an angle, allowing for larger turns and more degrees of movement. These factors are seen in the XY figures shown in figures 5 and 6.

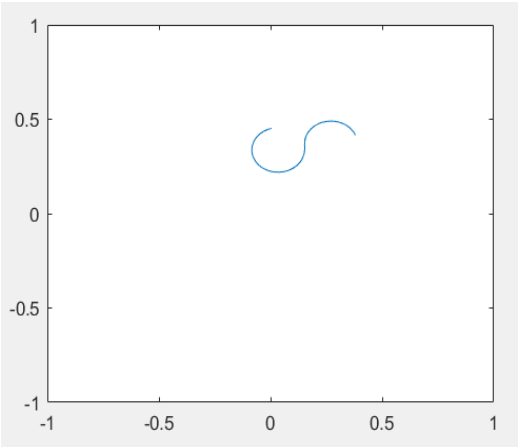


Figure 5: Wheel Drive Mode XY Figure

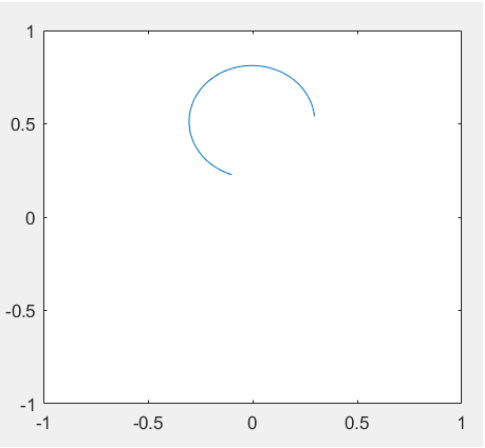


Figure 6: Normal Vehicle Drive Mode XY Figure

These figures display a map of the trajectory of the robot and we can see that with one combination of inputs, the robot can make sharper turns when adopting the wheel drive mode. To mimic this movement pattern with the normal drive mode, the forward input will need to be pushed on and off while the left or right arrow is pushed to achieve a tighter turn. However, this pattern would look similar but would not be a perfectly circular path which may not be beneficial in some circumstances. The same is true if we want to achieve a larger turn for the wheel drive mode, we could push the left or right wheel on and off to mimic this pattern but could not achieve a truly circular path. The drive mode the user chooses should depend on particular functions needed from the robot, therefore no mode is better than the other.

Kinematics and Locomotion of QBot 2

Kinematics describes the motion of bodies/points while ignoring the forces that cause them to move; typically focussing on the position, velocity and acceleration of said body and their relationship between each other. Understanding kinematics, the QBot2 can be manipulated in many different ways - altering speed, acceleration, position and direction to perform different tasks.

The purpose of this experiment is to study the basic motion behaviour of the Quanser QBot2 robot. The topics covered will be:

1) Differential Drive kinematics

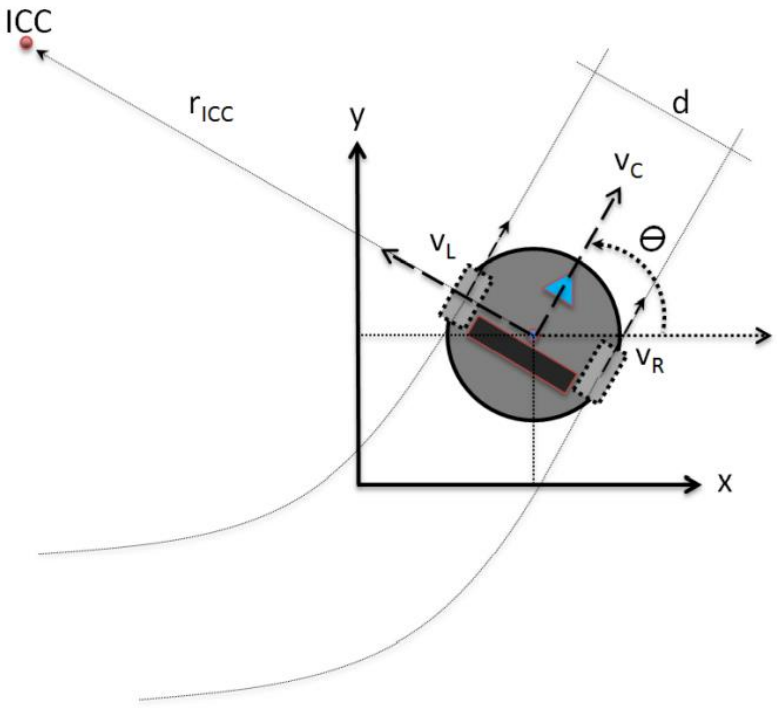
2) Forward and inverse kinematics

3) Odometric localization and Dead Reckoning

1. Differential Drive Kinematics

The QBot 2 is driven coaxially by a set of 2 wheels, controlled by high-performance DC motors with encoders and drop sensors. [3] Each wheel is driven independently and thus can be driven at different speeds and in opposite directions simultaneously making the robot very manoeuvrable.

The image below shows the QBot2's mobile platform reference frame followed by their definitions.



ICC = Instantaneous Centre of Curvature

r_{ICC} = distance from centre of robot chassis

v_C = Linear velocity of the centre of robot chassis

v_L = Linear velocity of the left wheel

v_R = Linear velocity of the right wheel

ω_C = $\dot{\theta}$ = Angular velocity for centre of chassis

Figure 7: Quanser QBot2 Mobile Platform Frame Definitions [3]

Differential drive kinematics is mathematical relationship that maps the motion of the wheels to the overall movement of the robot chassis; and thus is responsible for the predictability of the robots motion. This principle is the foundation of all mobile robot control. [3]

Kinematic equations

$$r_{ICC} = \frac{d}{2}\frac{(v_R+v_L)}{(v_R-v_L)} \qquad \text{eq 1}$$

$$v_C = \dot{\theta} r_{ICC} = \frac{v_R+v_L}{2} \qquad \text{eq 2}$$

$v_L = \dot{\theta}(r_{ICC} - \frac{d}{2})$ eq 3

$v_R = \dot{\theta}(r_{ICC} - \frac{d}{2})$ eq 4

$\omega_C = \dot{\theta} = \frac{v_R - v_L}{d}$ eq 5

1.1 Wheel Command Scenarios

This experiment will make use of the "QBot 2_Diff_Drive_Kinematics.mdl" Simulink model shown below:

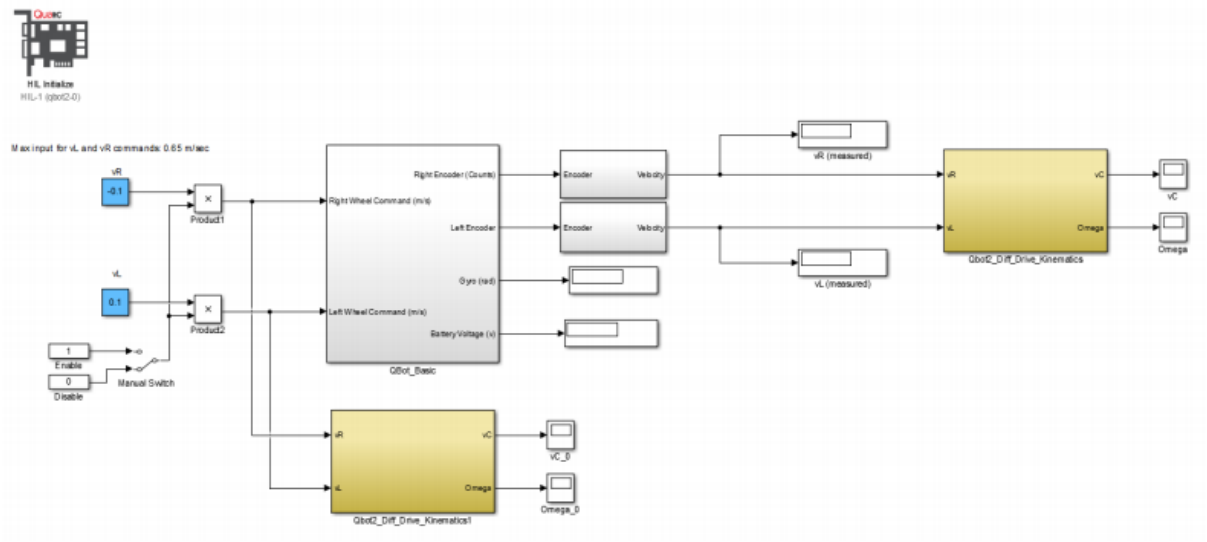


Figure 8: Controller model "QBot 2_Diff_Drive_Kinematics.mdl" [3]

1.1.2

Setting the left and right wheel velocities to the same value (0.1 m/s in this instance) causes the QBot2 to travel in a straight line with constant velocity, i.e. when $v_L = v_R$; $\omega_C = 0$ and $r_{ICC} = \infty$. This is because when the robot is travelling on a straight trajectory there is no angular velocity and the robot is travelling at its maximum speed in the given direction for the assigned wheel speeds.

This can also be explored mathematically, looking at equation 1:

$$r_{ICC} = \frac{d}{2}\frac{(v_R + v_L)}{(v_R - v_L)} = \frac{d}{2}\frac{(0.1+0.1)}{(0.1-0.1)} = \frac{d}{2}\frac{(0.2)}{(0)}$$

This sum is mathematically impossible and thus $r_{ICC} = \infty$

looking at equation 5:

$$\omega_C = \dot{\theta} = \frac{v_R - v_L}{d} = \frac{0.1 - 0.1}{d} = \frac{0}{d} = 0$$

The relationship between r_{ICC} and ω_C is that they're inversely proportional to each other.

1.1.3

Setting the right wheel velocity to -0.1 and keeping the left one at 0.1 (same speed, opposite direction) the robot spins at constant angular acceleration about ICC. As the robot experiences no displacement, $r_{ICC} = 0$ and due to the constant rotation, $\omega_C = \infty$. Like in 1.1.2, this can be equated using equations 1 and 5. The relationship is inversely proportional as found in 1.1.2.

1.1.4

When r_{ICC} is negative the robot is moving backwards

1.1.5

When ω_C is negative the robot is spinning anticlockwise

1.2 Forward and Inverse Kinematics

Forward kinematics are used to determine the linear and angular velocity of the robot in the world coordinate frame given the robots wheel speeds. Inverse kinematics is used to determine the wheel commands needed for the robot to follow a specific path at a specific speed. [3]

1.2.1 Forward kinematics

This experiment will make use of the "QBot 2_Forward_Kinematics.mdl" Simulink model shown below:

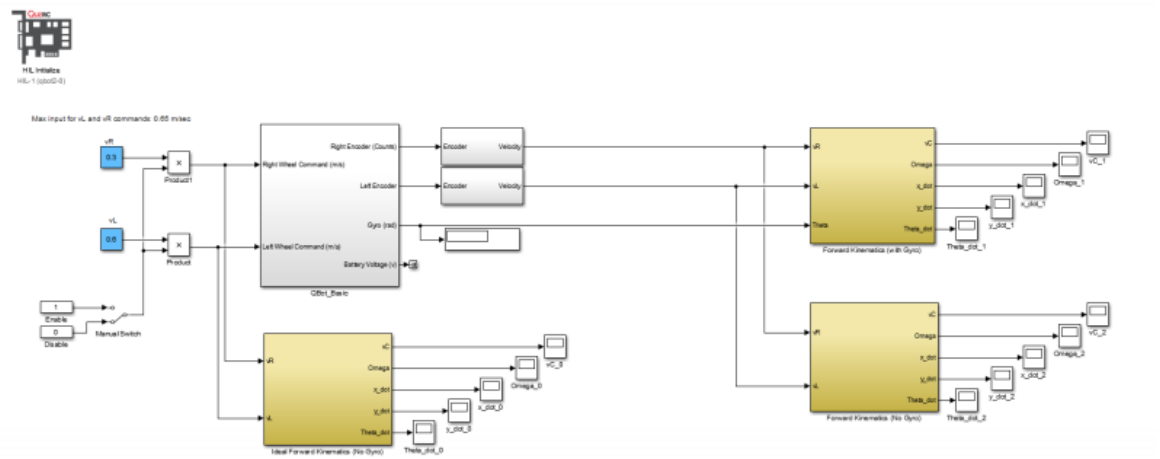


Figure 9: Controller model "QBot 2_Forward_Kinematics.mdl"

1.2.1.2

Setting $v_R = 0.3\frac{m}{s}$ and $v_L = 0.6\frac{m}{s}$ the robot moves clockwise about a point of radius r.

Comparing the ideal and measured linear and angular velocities in the world coordinate frame we see:

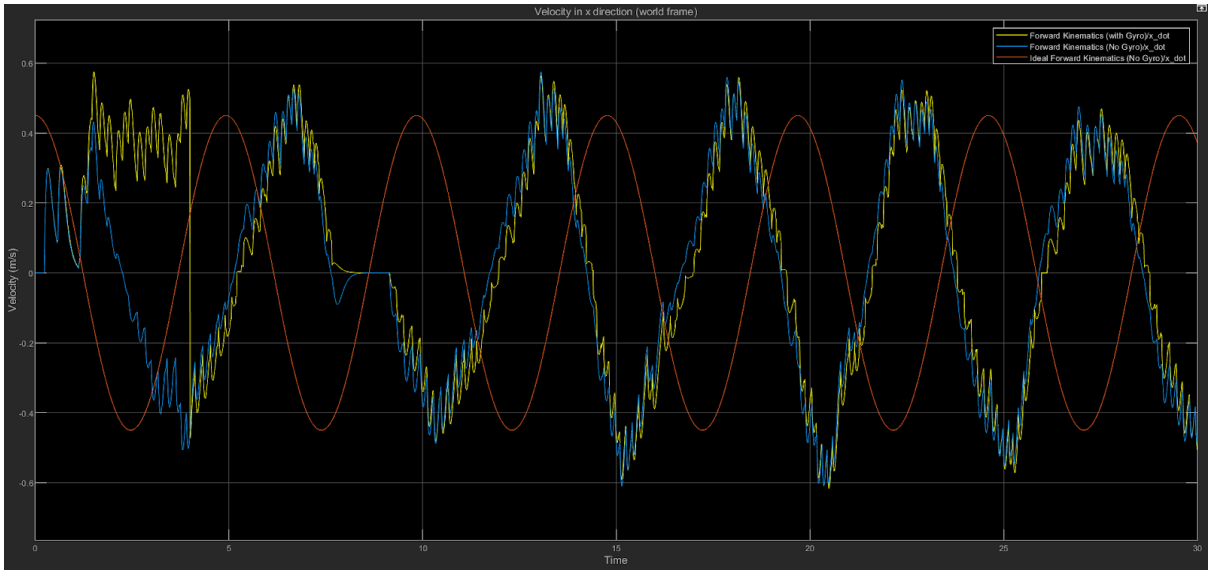


Figure 10: Ideal velocity in x direction vs measured with/without gyro

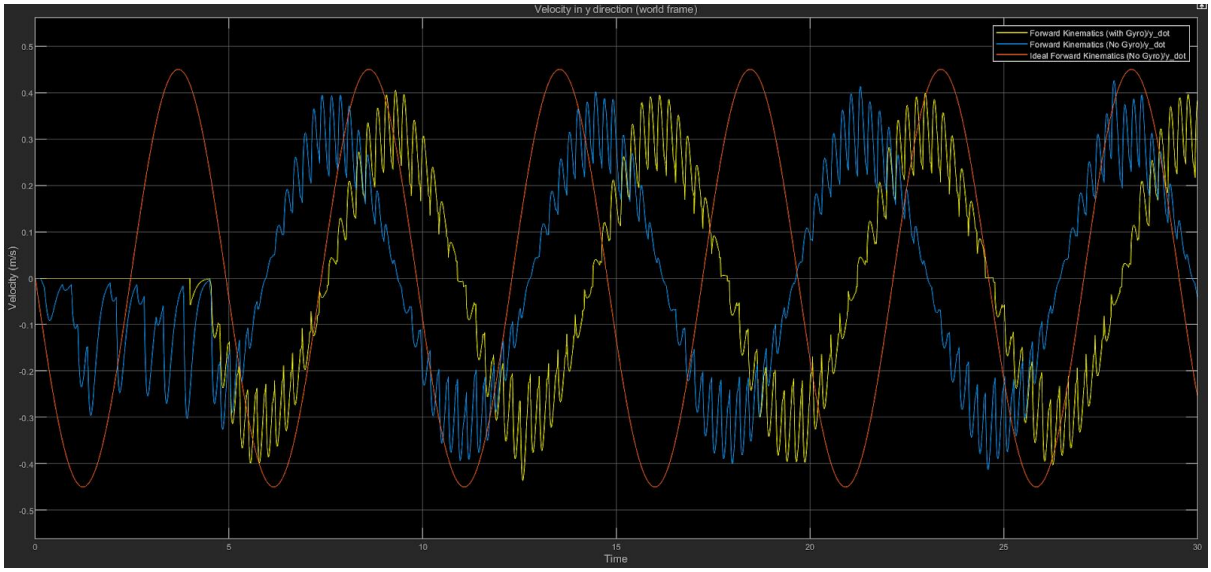


Figure 11: Ideal velocity in y direction vs measured with/without gyro

The velocities observed in the x and y directions seem to follow the ideal velocity overall quite accurately. However, there does seem to be a lot of noise generated especially in the y direction which is typically unwanted. I am uncertain to what extent the noise generated and it being out of phase with the ideal line affected QBot2's movement.

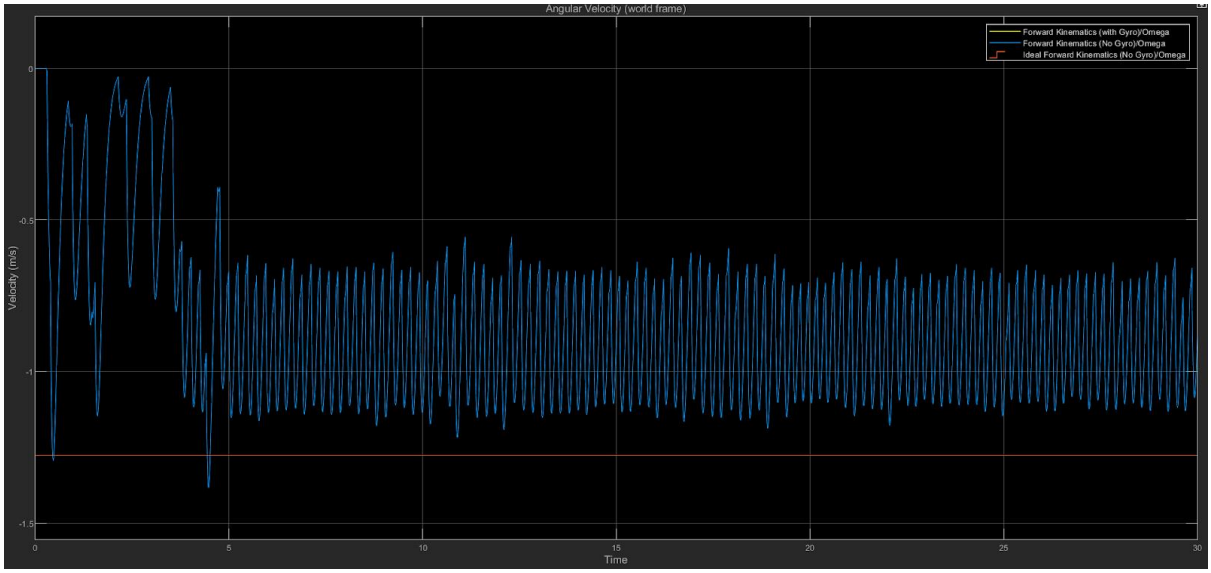


Figure 12: Ideal angular velocity vs forward kinematics, no gyro

Unlike the linear velocity comparisons, the angular velocity seems to generate a lot more noise, with its mean value resting at a potentially significant distance from its intended value. The angular velocity observed is quite a bit slower that the ideal speed.

1.2.1.3

When the right wheel is set to travel at twice the speed of the left wheel ($v_R = 2v_L$) the robot moves in a circular anticlockwise motion about a point of radius r. Increasing the value of v_L, (in comparison to v_R) produces a smaller radius r. Lowering the value of v_L, while $v_R < v_L$ creates anticlockwise

pathways with increasing values of r. If $v_R = v_L$, r becomes 0 and the robot goes in a straight line; and if $v_R > v_L$ the robot begins to move clockwise and the reverse of the correlations mentioned earlier become true.

1.2.1.4

To compute the required constant wheel speeds to generate a trajectory with a constant turning rate of $\omega_C = 0.1\, rad/s$ and a constant turning radius of r_{ICC} = 1m equations 3 and 4 can be used:

$$v_L = \dot{\theta}(r_{ICC} - \frac{d}{2}) \qquad \text{eq 3}$$

$$v_R = \dot{\theta}(r_{ICC} - \frac{d}{2}) \qquad \text{eq 4}$$

d = distance between centre of wheels = 0.235m [1]

inputting the values and equating:

$$v_L = 0.08824\ \frac{m}{s}$$

$$v_R = 0.11175 \frac{m}{2}$$

The robot travels along a circular path along a radius of r = 1m about ICC in an anticlockwise direction. Direction can be changed by swapping the values of v_L and v_R.

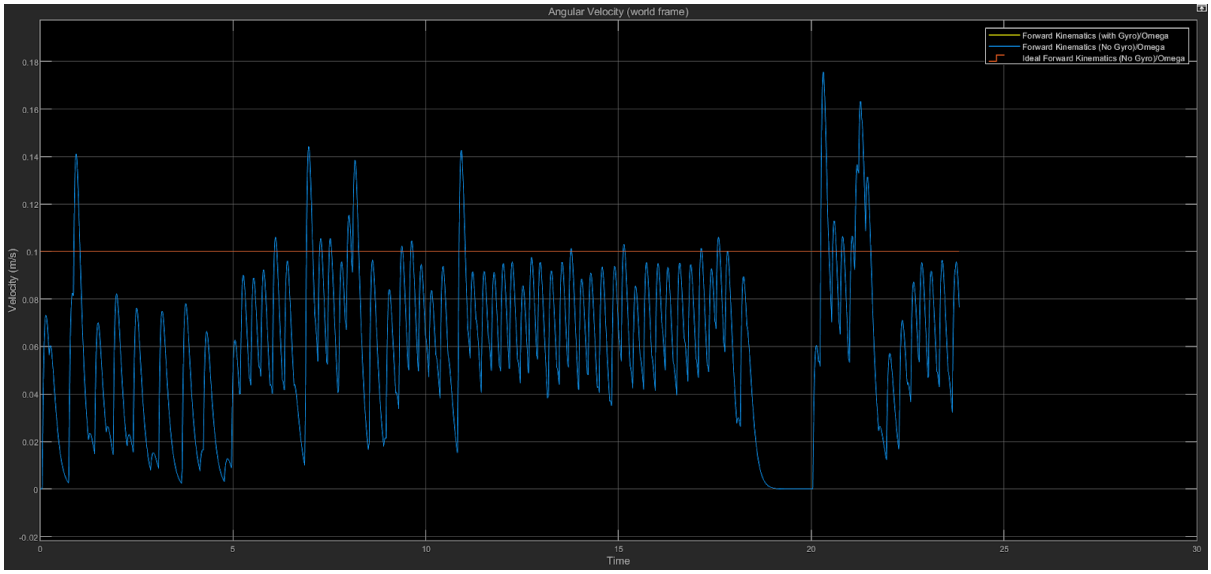


Figure 13: Ideal angular velocity vs forward kinematics, no gyro

The angular velocity is slightly slower than desired and seems to give slightly more unstable results than in other experiments.

1.2.2 Inverse Kinematics

This experiment will make use of the “QBot 2_Inverse_Kinematics.mdl” Simulink model shown below:

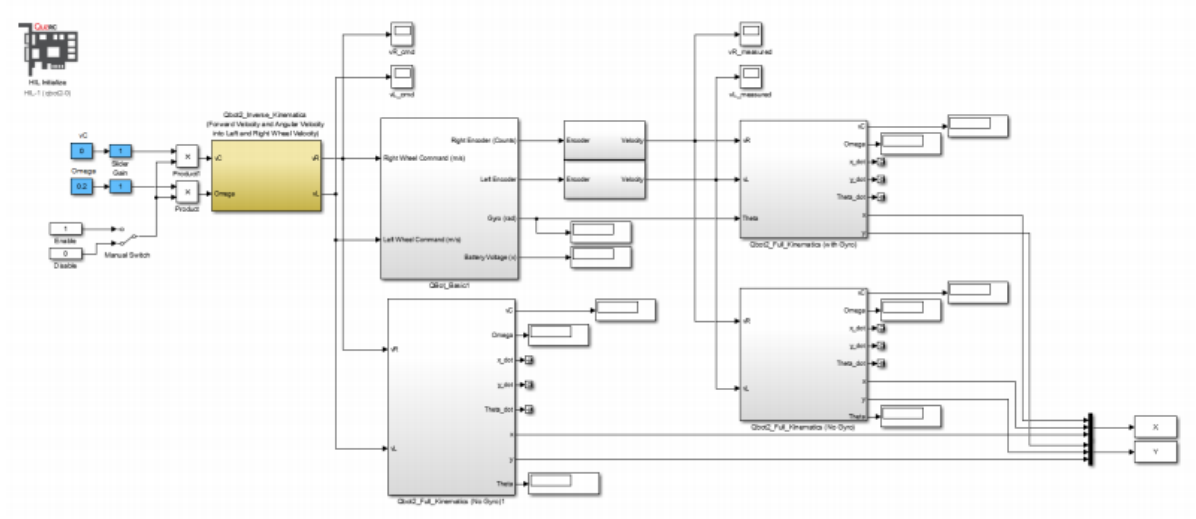


Figure 14: Controller model “QBot 2_Inverse_Kinematics.mdl” [3]

1.2.2.4

Setting v_C = 0.1 m/s and $\omega_C = 0.1\ rad/s$ and comparing the ideal results to the observed ones gives us these graphs:

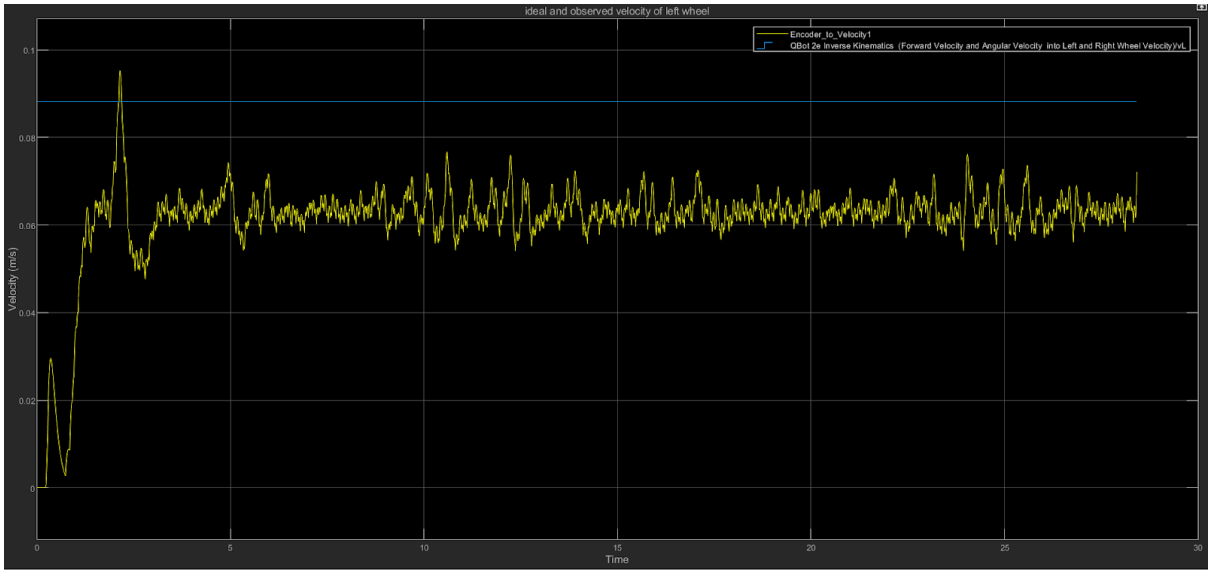


Figure 15: ideal and measured velocity of left wheel

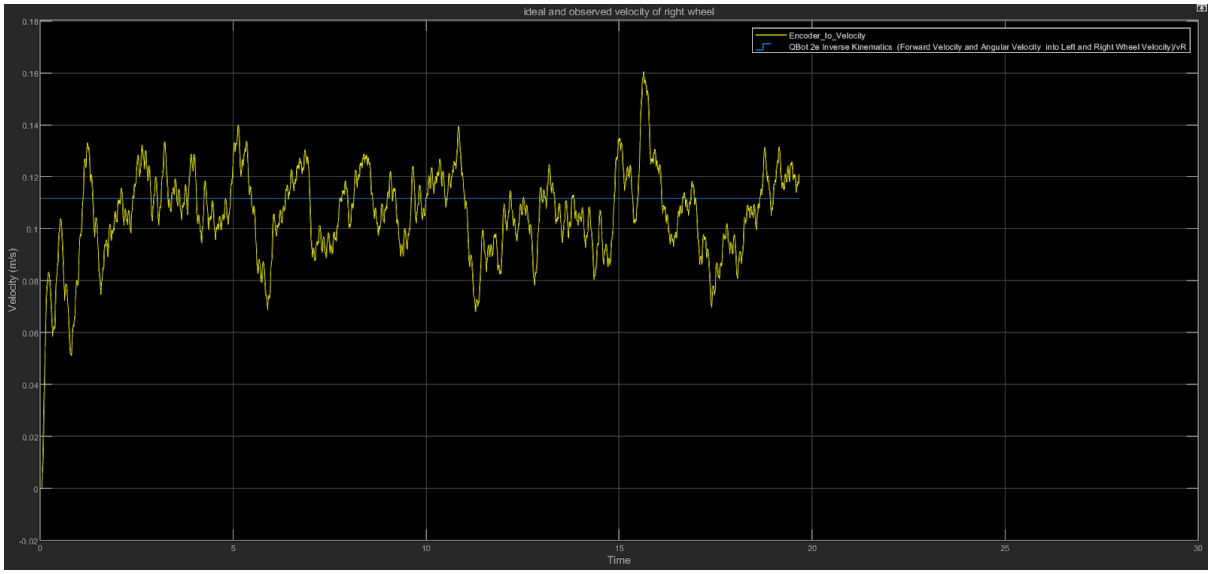


Figure 16: ideal and measured velocity of right wheel

The left wheel seems more stable but settles at a lower speed than the ideal while the right wheel seems less stable but seems to have an average speed closer to the ideal. Comparing them to the forward kinematic graphs; when the robot was on a rotating trajectory, the velocities oscillated on the graph producing a sin wave. Though the robot also turns for this experiment about a point of radius r, I believe that because the graph does not oscillate because the robot has constant angular velocity meaning that the wheels would have to have constant velocity also.

Setting the values to v_C = 0 m/s and $\omega_C = 0.2\ rad/s$ and running the model again

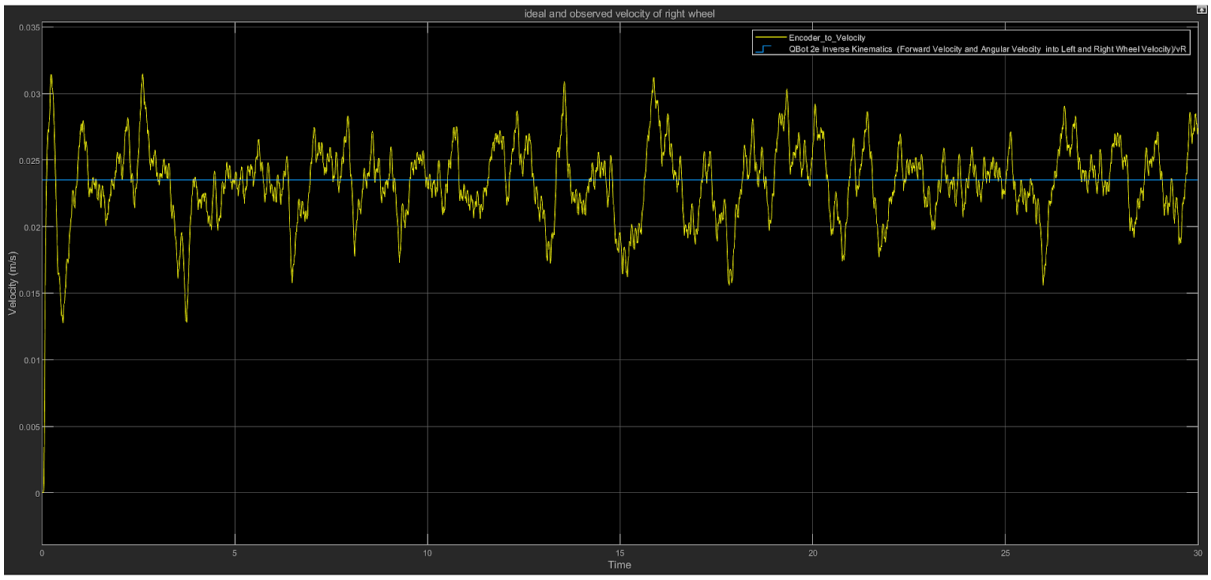


Figure 17: ideal and measured velocity of right wheel

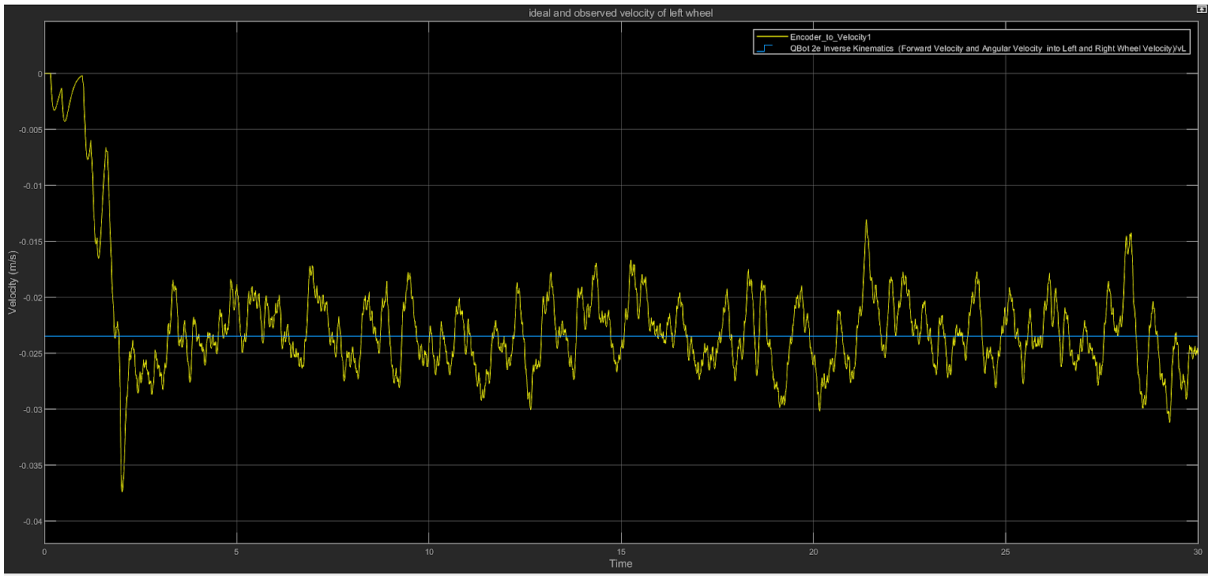


Figure 18: ideal and measured velocity of left wheel

As v_C = 0 m/s the robot does not move linearly and rotates about the ICC. The speeds of the wheels are roughly averaging at the ideal speeds but it doesn't oscillate in the way that the forward kinematics graph displayed, I believe the reason for this is that the angular velocity is still a constant.

1.3 Odometric localization and dead reckoning

1.3.1 Trajectory errors

This experiment will make use of the "QBot 2_Odometric_Localization.mdl" Simulink model shown below:

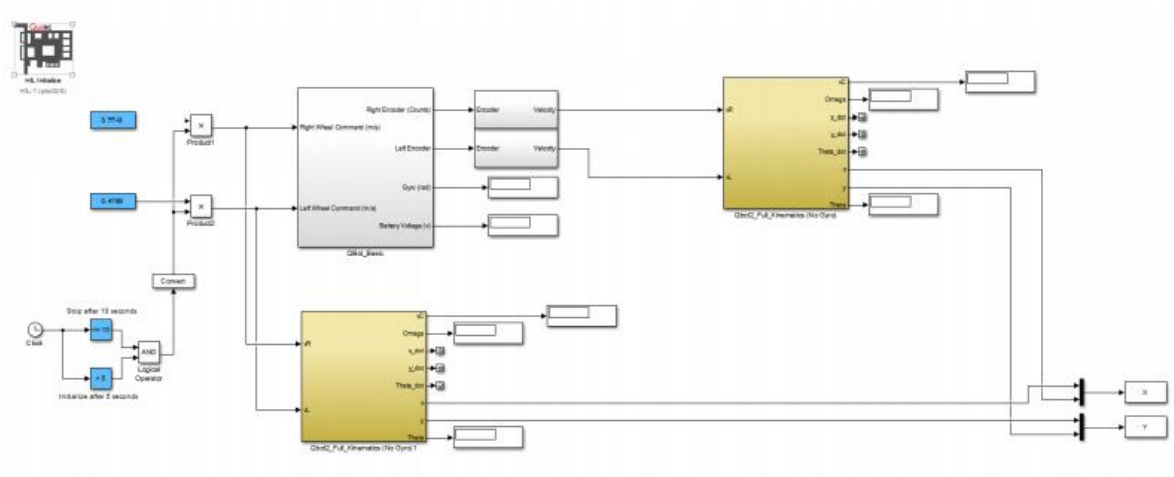


Figure 19: Controller model "QBot 2_Odometric_Localization.mdl" [3]

1.3.1.2

This controller attempts to make the robot perform a circle of 0.5m, while using the plotXY.m code provided to track the trajectory.

This is the graph that was produced:

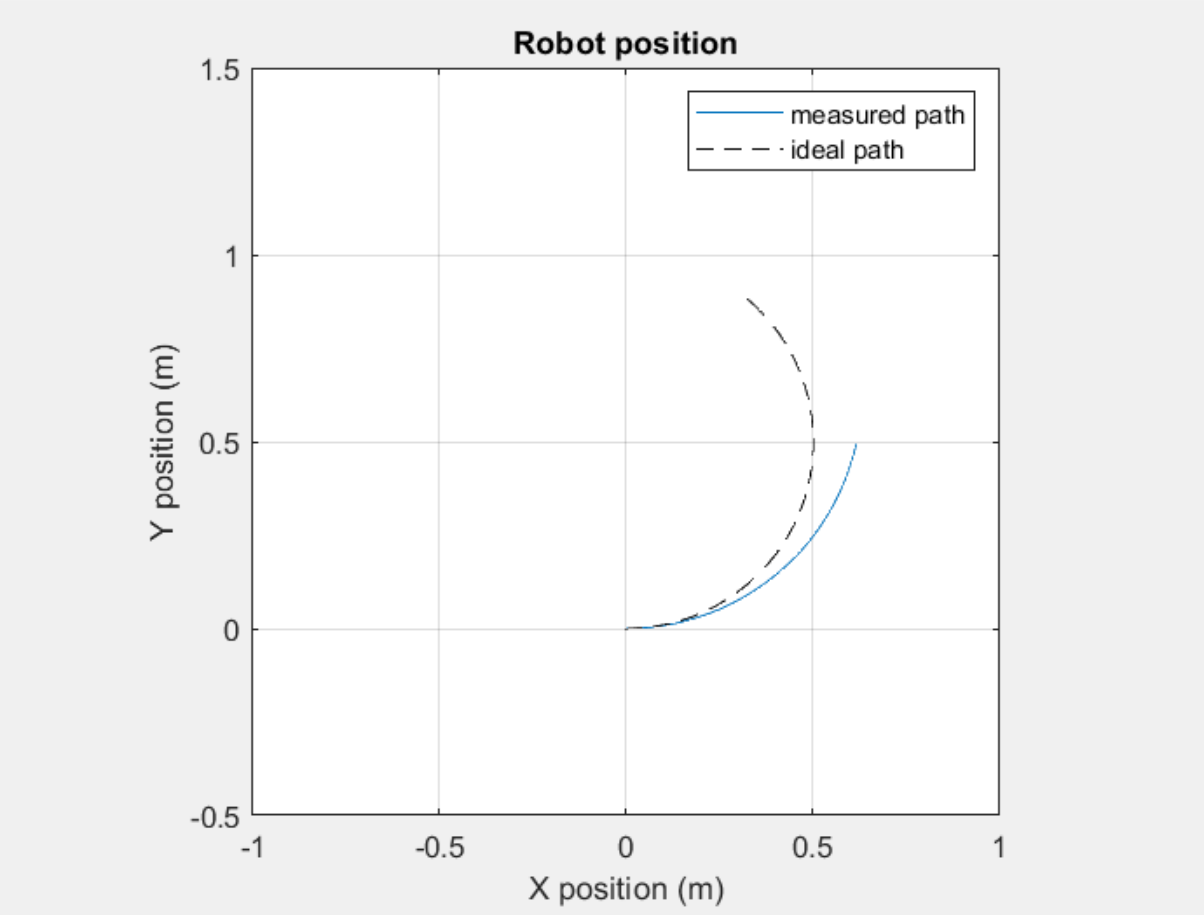


Figure 20: ideal and measured path

Uncertain on how to track the whole process, but you can see that if the ideal path had completed, it would have made a perfect circle of 0.5m radius as intended. The measured trajectory however is slightly off.

1.3.1.5

Incorrect wheel speed can contribute to pathing errors, so it is important to have them right.

Mapping and Localisation

The next experiment for the QBot 2 was to test its mapping and localisation abilities. In this experiment, we used algorithms to map the QBots environment and localize the robot inside that environment. The methods used to accomplish this were occupancy grid mapping and particle filtering.

Occupancy Grid Mapping

An occupancy grid map is a representation of the environment and consists of grid cells relating to specific points in the environment and storing information as to whether that cell is occupied or not, which means whether or not there is an obstacle at that location (black is occupied, white is unoccupied). In theory, sensors could just detect where an obstacle is and make the corresponding cell in the map occupied. However, in reality, it is more difficult as sensors have some uncertainty associated with them due to inaccuracy and noise, therefore we can never be certain that there is an obstacle at a particular point or if a cell is occupied or not. Every cell has a probability based on how likely there is an obstacle and the occupancy grid map is a grid of these cells. The map generated is a map that has the highest probability of matching the real-life environment.

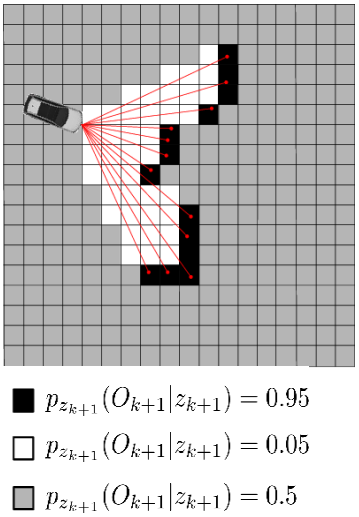


Figure 21: Occupancy Grid Map With Occupancy Probabilities

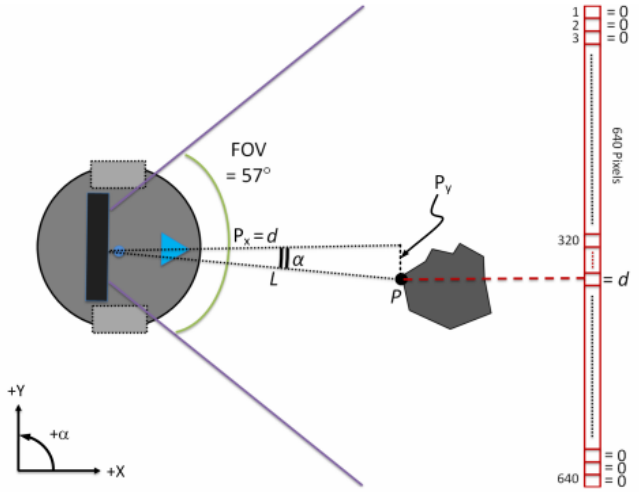


Figure.22: Depth Data Received From Kinect Sensor

In our experiments, we generated a 2D map using just one row of 640 pixels on the Kinect sensor rather than using all the 480x640 pixels. Each pixel represents a distance or depth from the camera to the object. The row of pixels we used was in line with the camera height, if we used a different row then a different depth reading will be recorded due to the difference in angle from the horizontal. If the row is lower, the sensor could measure the distance to the floor instead of an obstacle or if higher, the sensor

may not detect an object even if it is hitting one as it will be reading measurements from the sky or a ceiling. Therefore, it is important to use the horizontal row for this experiment.

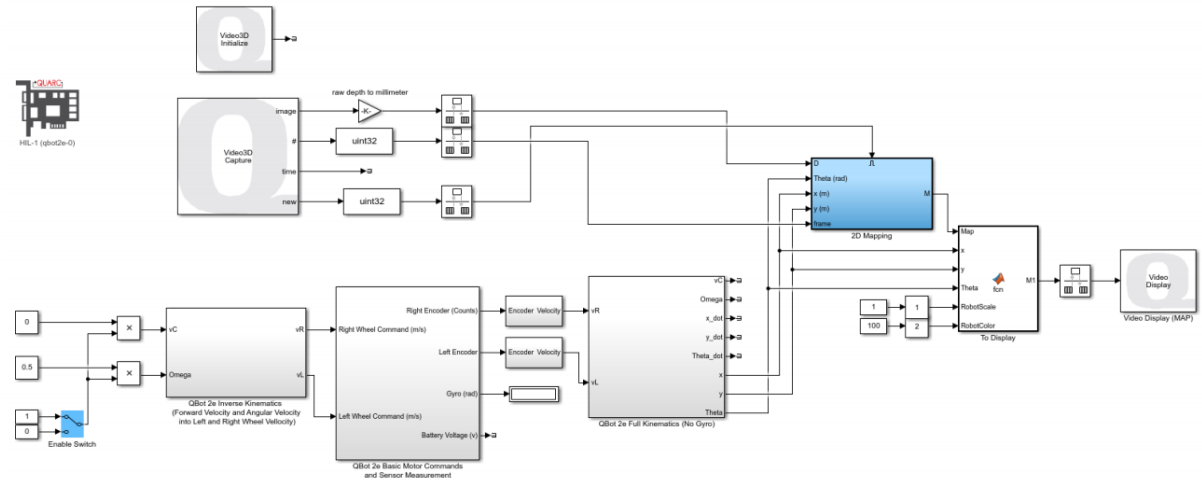


Figure 23: Snapshot of the controller model QBot2e_2D_Mapping_Manual.mdl

Firstly, after moving the robot 1 metre away from the middle block seen in the simulator (using the gridded lines on the workspace floor) and setting angular and linear velocities to zero (making the robot stationary), we ran the Simulink model to show a video display window. This window shows the map of the robot's surroundings, shown in figure 24

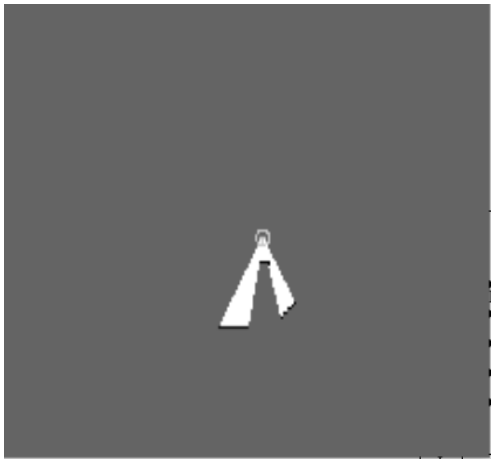


Figure. 24: Map Video Display, Robot 1m Away from Obstacle

The figure shows a light grey circle representing the QBot 2, while the black dots represent obstacles in the field of view of the camera attached to the robot. This field of view is shown as the white area in the figure and the dark grey area is any unknown space outside the field of view or behind obstacles. To observe how the display changes depending on the distance from the obstacle, we moved the robot to variable distances seen in figures 25 to 27.

Figure. 25: Map Video Display, Robot 1m Away from Obstacle

Figure. 26: Map Video Display, Robot 1.5m Away from Obstacle

Figure. 27: Map Video Display, Robot 1m Away from Obstacle

We can see that the further the obstacle is from the robot, there are fewer unknown grey areas and we can see more of the map due to the obstacle not blocking most of the field of view of the camera. But, to observe the entire map, we needed to rotate the robot so the field of view is effectively 360 degrees rather than 57 degrees. By setting the angular rate, $\omega = 0.3$, the robot rotated twice at a constant rate and a map was displayed, this time showcasing the entire workspace of the simulator. To test the robot's accuracy, we can compare the map displayed in MATLAB to the workspace from the simulator.

Figure 28: Map Video Display, Robot rotated 360 deg

Figure 29: Top View of Snapshot of QBot 2e Workspace

When comparing these two images, we can see the map produced by the robot is very similar to the workspace except for the grey areas behind obstacles that the robot cannot see. To eliminate this grey area, we moved the robot to another point in the workspace to view more of the area, as seen in figure 30.

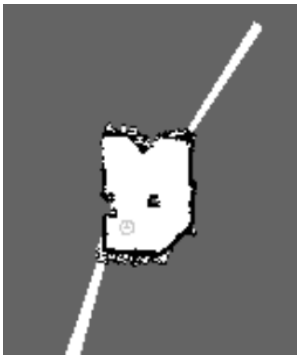


Figure 30: Map Video Display rotated 360 deg at two points

While the entire workspace can now be seen on the map, many errors can now be observed. There is a "clear space" seen outside the boundaries which is an issue with the Kinect sensor where the sensor reads an infinite distance when the obstacle is less than 0.5m away. Another error is the noise associated with the sensor which is the reason for the black and white areas outside of the map. As the robot moved linearly, the distances calculated had less accuracy but when the robot stopped and just rotated the map was again accurate.

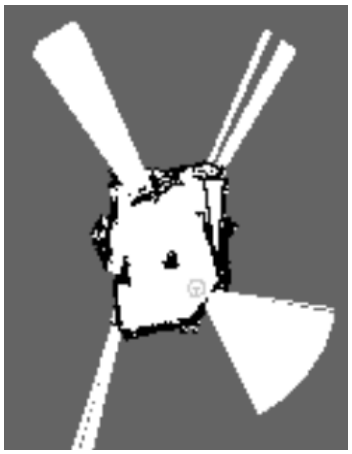


Figure 31: Map Video Display Showing Offset As Robot Rotates

Figure 32: Map Video Display Showing Robot Moving Past Boundary

As the robot moved over time, there was an offset between the real and expected positions of the robot. After a lot of movement within the environment, we rotated the robot at a particular point and each rotation led to a slightly rotated map seen in figure 31. This may also be due to lag associated with the sensor not being in line with the robot's rotation. Another error is when the robot hits the wall, displaying the map seen in figure 32. Despite not moving past the boundary in the workspace, the robot carries on past the boundary in the generated map.

Particle Filtering

Particle filtering is a method of localization, used to estimate the position and orientation of the robot within the environment. Since we have produced a map of the surroundings using occupancy grid mapping in the former experiment, in this experiment the robot initially knows the environment but does not know its pose within that map. To find the pose, discrete poses (particles) are generated and randomly distributed across the map since the robot has an equal chance of being anywhere on the map and facing any direction. Particles that closest resembles the robot sensor readings are saved and they converge towards possible locations in the map. As the robot moves, Odemetric readings (velocity, angle etc) help narrow down possible locations so we should see particles converge towards a specific location.

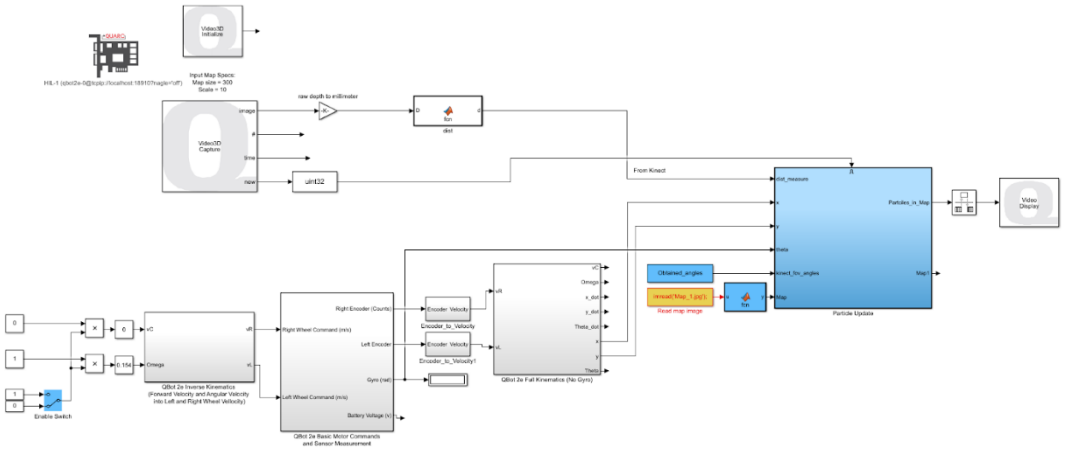


Figure 33: Snapshot of the controller model QBot2e_Particle_Filtering.mdl

When running this simulation, the enable switch is off and the robot does not move so only sensor readings can be used and no odometry data. To see where particles are distributed over time we can change the simulation time to stop after a certain time.

1 sec: 3 sec: 5 sec: 10 sec:

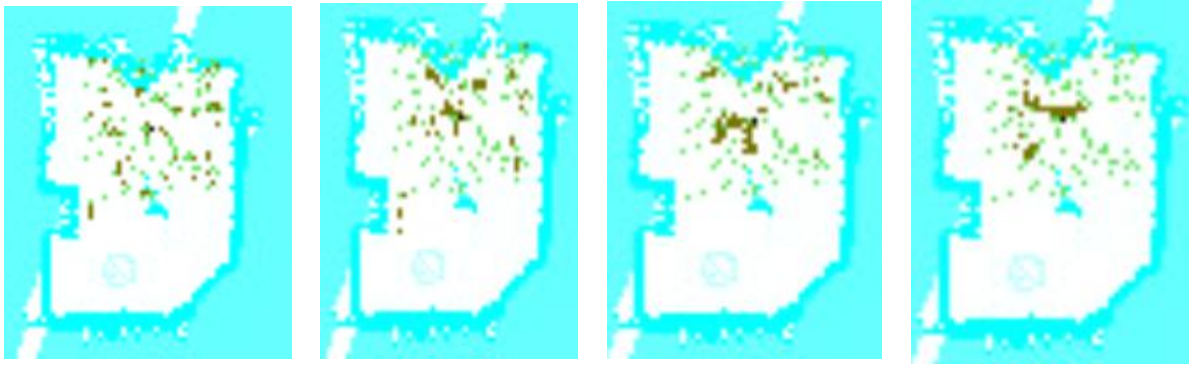


Figure 34: Screenshots of Particle Filters After Different simulation times

In these displays we see green dots that represent the initial values for the particles at t = 0 while the red dots show the particles updated in real-time, changing every second. The black dot shows odemetric data localization. We see the red dots moving over time, gathering around certain locations in the map until converging to the centre around the black dot. This shows how the algorithm for the programme works where the location of each particle is predicted according to the existing data and additional random noise to simulate the effect of sensory noise then the weights of each particle are updated based on the latest sensory information available. In this case, the robot is looking at the block in the middle of the workspace so the weighted particles need to have simulated sensor readings matching the robot's sensor readings. Since the robot is facing the block, it is clear for the particles to gather around points where the block can be observed from that distance. However, if the robot is initially facing just a wall, for example, the location is harder to find.

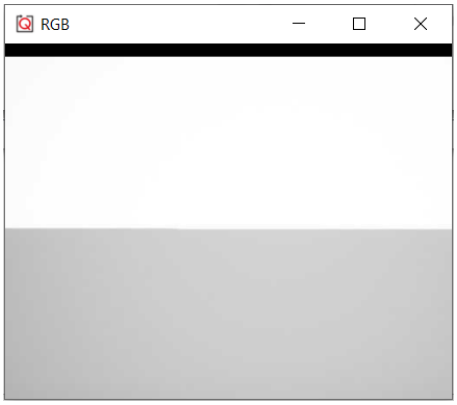


Figure 35: RGB Video Display With Camera Aimed At Wall

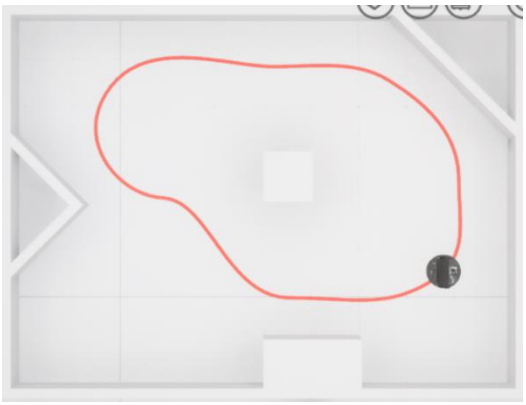


Figure 36: Robot Facing Wall in Quanser Workspace

Figure 35 shows what the robot sees when in the location shown in figure 36 Since the robot's field of view encompasses just a wall, there should be many locations the robot could be at across the map, opposed to if it was facing a corner. These locations are shown in figure 37.

Figure 37: Estimated Location of Robot

Figure 38: Particle Filter Estimating Location of Robot

However, there are issues with the algorithm, particularly with respect to our situation with remote learning and simulations. The particles initially distributed only cover the bottom half of the map so the robot must be in that region when the simulation starts for the localization to work. This is shown in figure 38 where the predicted location of the robot is not at the real location.

Furthermore, the black dot will always start at the origin so if we move the robot before the simulation starts, the algorithm thinks the robot is at the origin rather than its new position. This is the same situation as in a real lab environment if someone picked the robot up and put it somewhere else. The black dot will represent where the robot was rather than where it is now. As the robot then moves, the particle filter will show the black dot mimicking the robot's movement but at a different location.

The above examples show the particle filter initially distributing 100 particles. The following examples show the filter distributing 10, 20, 50, 200 and 500 particles respectively.

Figure 39: Screenshots of particle filters after converging, with 10, 20, 50, 200 and 500 particles respectively from left to right

Figure 40: Screenshots of particle filters after converging, with 10, 20, 50, 200 and 500 particles respectively from left to right

The number of particles in each simulation affected the accuracy of the filter with the 10, 20 and 50 particle filters having too few particles in to get accurate localization as particles were not distributed well. The filter with 200 particles was more accurate than those before as expected while the 500 particle filter did not converge even after waiting 500 seconds, both of these took longer to converge and run the simulation than the default filter with 100 particles. This is a common characteristic with particle filters as more computational resources are required to assign weights to more particles. It was decided that the accuracy for these filters was not significantly better than the filter with 100 particles but took much longer to run and converge, therefore this filter was the best option for this exercise.

The next test was to change the Gaussian noise distribution to see how that affected the filter with 100 particles. We first halved the sigma_measure, sigma_move, and sigma_rotate variables then doubled them and observed the effects.

Figure 41: Particle Filter with Gaussian Noise Distribution Halved

Figure 42: Particle Filter with Default Gaussian Noise Distribution

Figure 43: Particle Filter with Gaussian Noise Distribution Doubled

Each simulation was run for 50 seconds to allow plenty of time to converge. All filters estimate the location of the robot to be in roughly the same location however, we found that the lower the noise, the smaller the area of the converged particles were. When the noise is halved, the particles are densely packed in one location so the algorithm is more confident of the robot being at that location. This is opposed to the filter when noise is doubled, there is very little confidence of the location as many locations are predicted. Ideally, we would prefer to have very little noise like in figure 41 but in the real world, the noise distribution likely reflects that in figures 42 and 43.

Computer Vision

3. Computer vison and vision-guided control

Computer vision typically consists of processes and algorithms that interpret images by identifying attributes such as lines, corners, specific colours, and object locations. Using these techniques, digital image processing can be used for a wide array of applications from mapping to navigation to facial recognition or dynamic path planning. [3]

This exercise objective covers: 1) Image Thresholding

2) Edge Detection

3) Blob Analysis

3.1 Image thresholding

Image thresholding is a technique used to isolate specific parameters from an image i.e. specific colour or brightness level.

This example uses the following controller model "QBot2_Image_Processing_Color_Thresholding.mdl" :

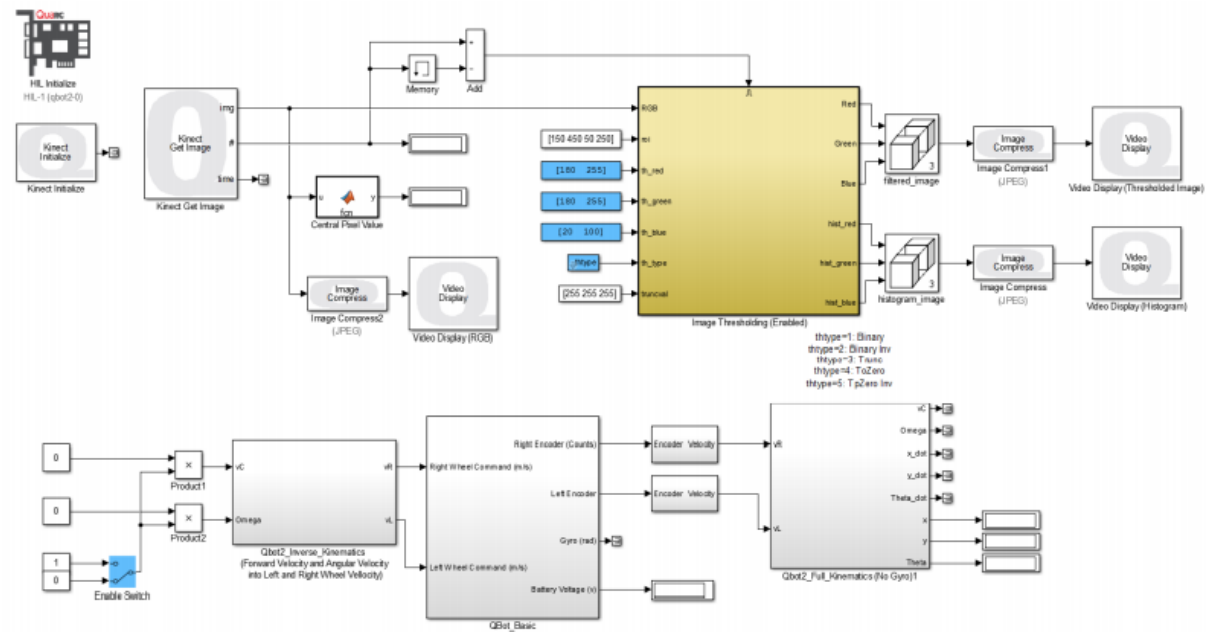


Figure 44: Controller model "QBot2_Image_Processing_Color_Thresholding.mdl" [3]

The QBot2 robot is fitted with an RGB camera that is used to detect colours. The camera can be used to produce a histogram which represents the brightness of the primary colours in an image. When the camera is facing a white surface, the primary colours detected should be somewhat equal (excluding shadows and reflections etc) as shown below:

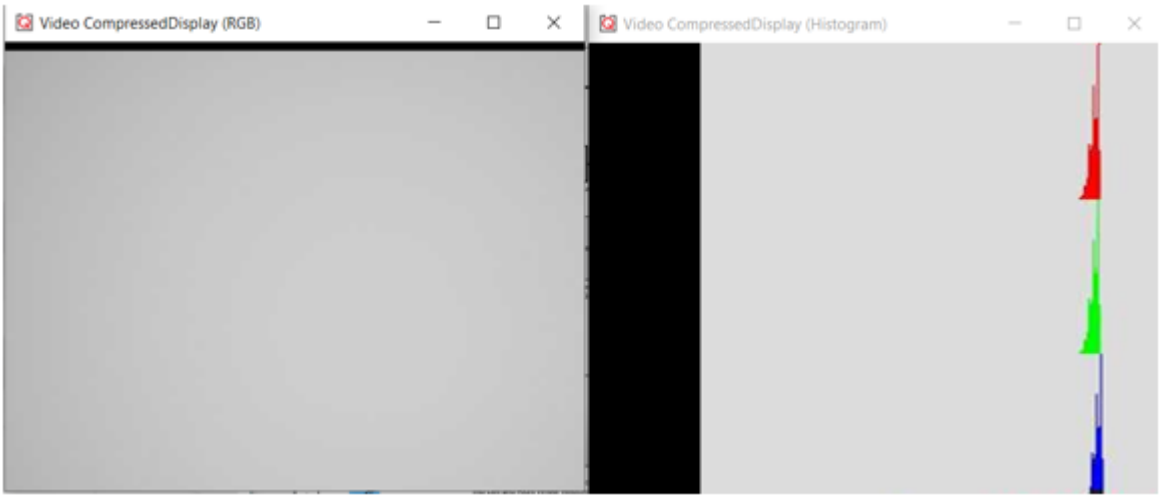


Figure 45: RBG camera facing white surface and histogram

As you can see the histogram has detected almost identical levels of red, green and blue when looking at a white surface. Each colour has a threshold value and the values of the camera can be tweaked to isolate certain colours/shades/brightness.

Path Planning

The final experiment involving the Qbot 2e was the path planning. This is used to determine where the robot should go. This is done by first setting up the occupancy grid map, which reveals where the obstacles are to the robot. There are many different methods for path planning, however, in this experiment we only looked at two methods: Potential Field and the A* algorithm.

Potential field

This works by the target exhibiting an attractive potential field and the obstacles exhibiting a repulsive potential field. This means that the robot will be attracted towards the target whilst avoiding the repulsive obstacles. Adding together the attractive and repulsive potential fields gives the resultant potential field. This resultant potential field, or net potential field is used to produce an artificial force equivalent to the gradient of the potential field.

A* algorithm

The A* algorithm is a popular choice for path planning applications, as it's simple and flexible. The way it works is that it considers the map as a two dimensional grid, with each square of the grid being a node or a vertex. With the goal node specified, it uses graph search methods to work its way towards the goal. It visits vertices or nodes, identifying the appropriate successor node at each step. It uses a

combination of heuristic cost, h(n) and path cost, g(n). The heuristic cost is the expected distance from the available adjacent nodes to the target node (available nodes not being obstacles or previously visited nodes). The path cost is the cost of moving from the current node to the next node. Using just heuristic or path cost alone is not optimal, with a combination of the two providing an optimized path.

1. Creating an occupancy grid map

In order to create the occupancy grid map, we had to move the robot to 4 different poses around the obstacle, making sure to stay at least 1m away from the obstacle. At each pose, the robot was rotated 360° in order to create a clearer image.

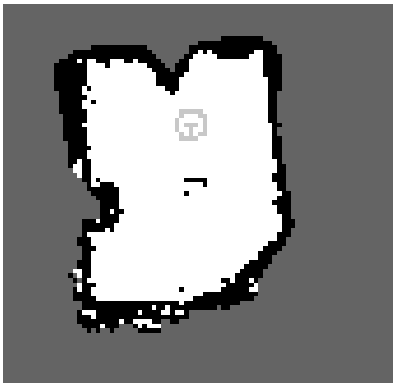


Figure 46: map video display of robot workspace with obstacle

This image was then imported into the MATLAB workspace and converted into a mat file.

2. Path Planning and Motion Execution

In this next step, we processed this map image to find the coordinates of the obstacle in the map. It was supposed to have created a simplified occupancy grid map, however we encountered some problems with the provided MATLAB script ‘process_MAP’. There seemed to be an undefined variable ‘C’ which we could not find a solution for. This meant that the script didn’t run properly, producing an empty occupancy grid map. At this stage, we were unable to even set the position of the robot and the target, however after tweaking the code it allowed us to set them.
We then ran the script ‘Path_Planning_MAP’, while making sure to run the potential field method. This produced the following map:

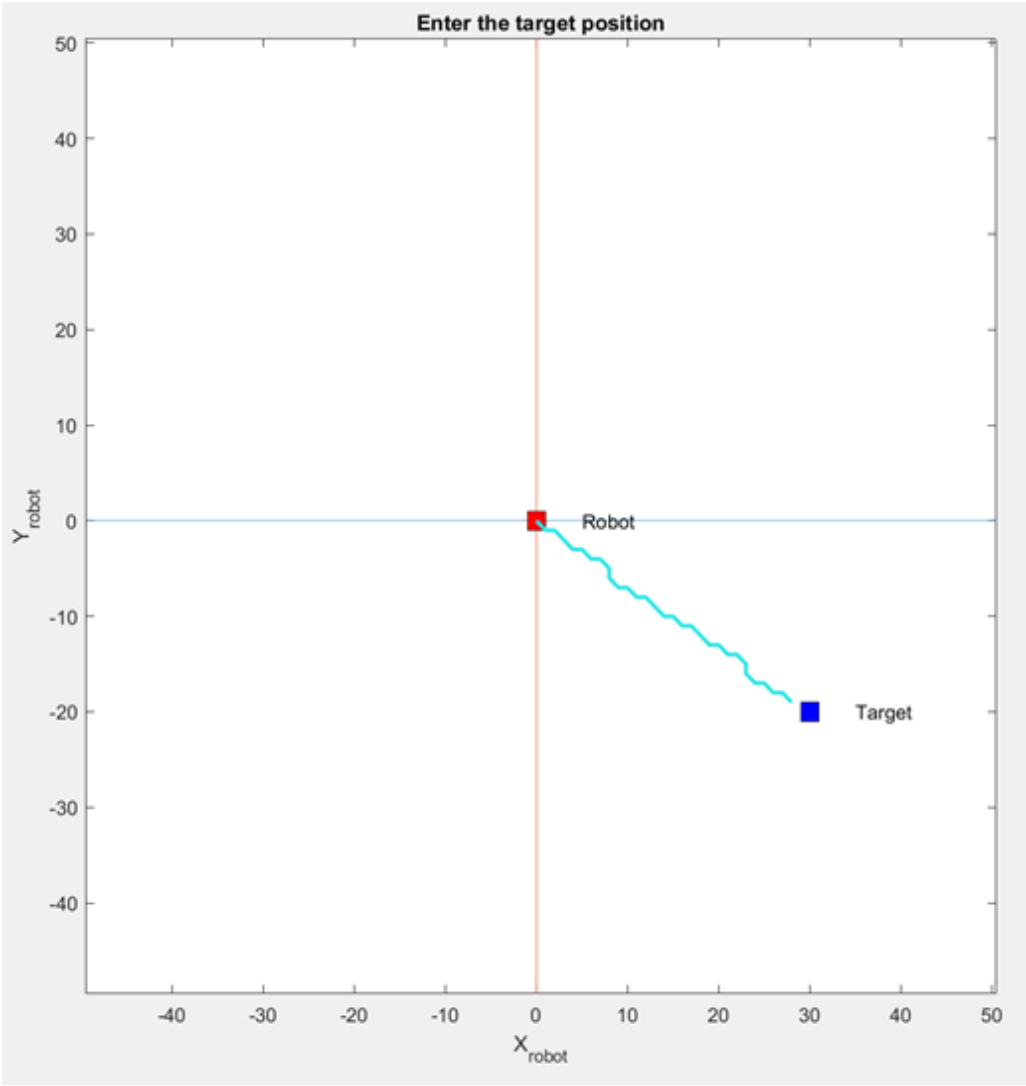


Figure 47: Potential field method of path planning

As figure… shows, the path from the robot to the target isn't the smoothest with the potential field method. It is not a straight line to the target, with a very stuttered pathway.

The next method is the A* algorithm. This produced the following plot:

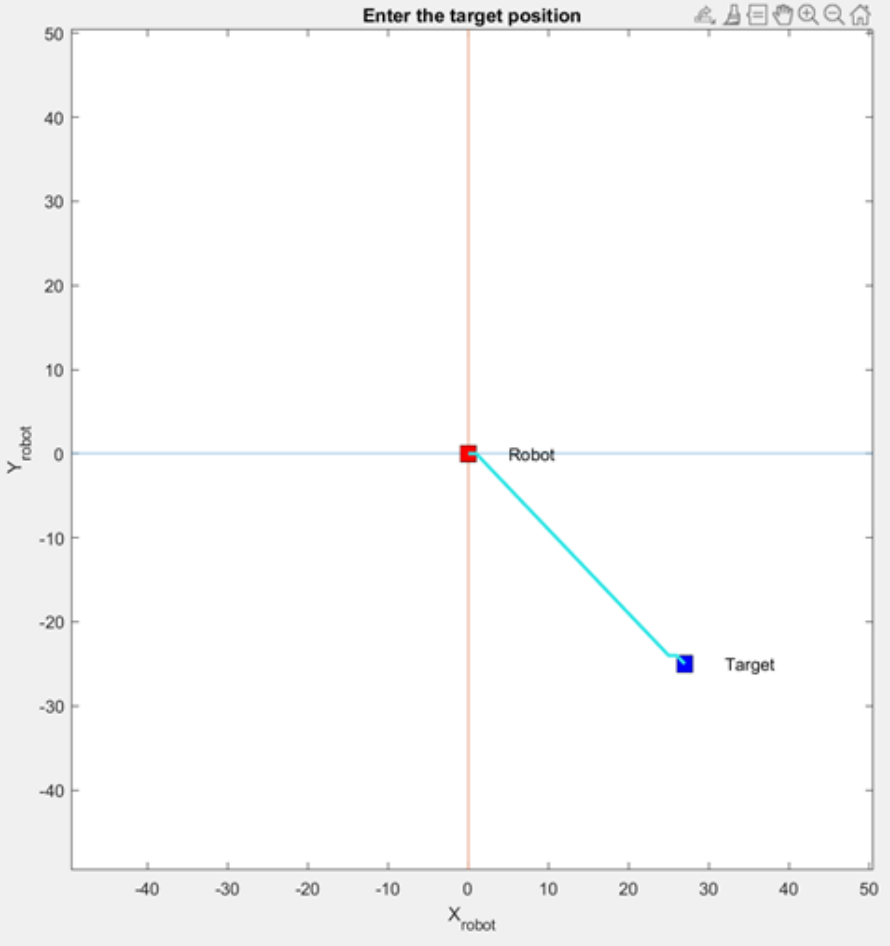


Figure 48: A algorithm method of path planning*

References

[1] - Quanser, “High-Performance Autonomous Ground Robot for Indoor Labs”, 2021 [online] https://www.quanser.com/products/qbot-2e/ [Accessed 22/04/21]

[2] - Amir Haddidi, Peter Martin, Cameron Fulford - Quanser, *“Student Workbook Qbot 2 for QUARC”*, 2017. Pg 2

[3] - P. Martin, C. Fulford and A. Haddadi, “Student Workbook for QBot 2,” Quanser Inc., 2017. [Online]. Available: https://www.made-for-science.com/de/quanser/?df=made-for-science-quanser-qbot-2-quarc-coursewarestud-matlab.pdf. [Accessed 2 April 2021].