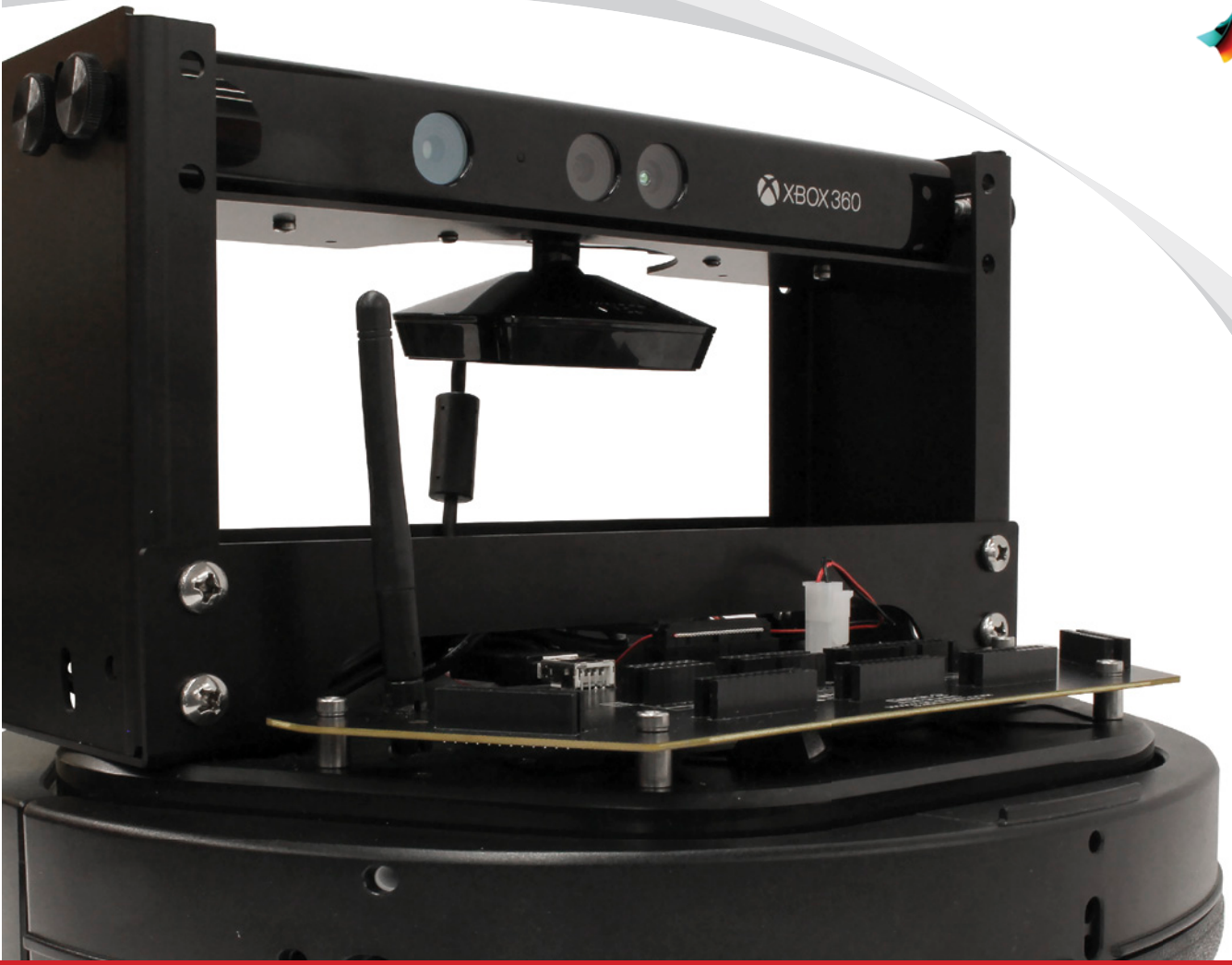
QUANSER
INNOVATE · EDUCATE

STUDENT WORKBOOK

QBot 2 for QUARC

Developed by:
Amir Haddadi, Ph.D., Quanser
Peter Martin, M.A.Sc., Quanser
Cameron Fulford, M.A.Sc., Quanser

Quanser educational solutions
are powered by:
MATLAB
SIMULINK



CAPTIVATE. MOTIVATE. GRADUATE.

EXPERIMENT 0: INTRODUCTION TO QBOT 2 FOR QUARC

The objective of this introductory exercise is to explore the hardware and software related to Quanser QBot 2 Mobile Platform. You will learn how the QBot 2 is actuated, what types of sensors are available, and how you can communicate with the device in order to send commands and receive sensory data.

Topics Covered

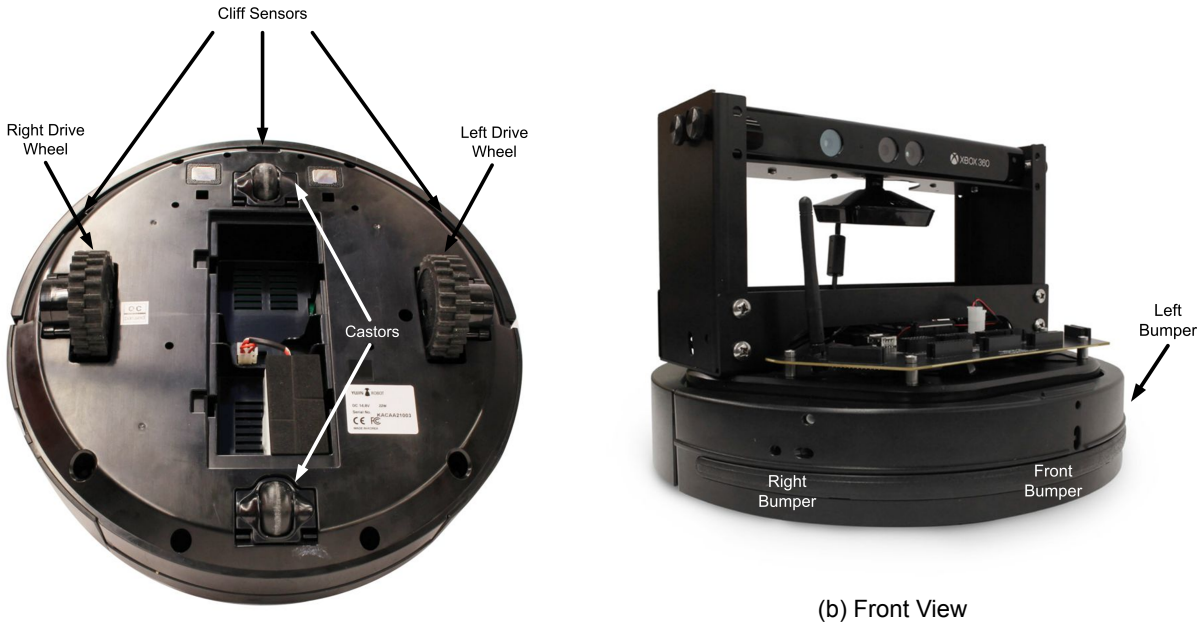
- QBot 2 Hardware Components
- QBot 2 Software and Communication

1 Background

The purpose of this lab is to get you started with the Quanser QBot 2 Mobile Platform and familiarize you with the basic concepts related to the product including sensors, actuators, and the QBot 2software components.

1.1 QBot 2 Main Hardware Components

The Quanser QBot 2 Mobile Platform consists of two central drive wheels mounted on a common axis that bisects the robot as shown in Figure 1.1 a. This drive configuration is known as differential drive. Castors at the front and back of the robot stabilize the platform without compromising movement. The two drive wheels are independently driven forward and backward in order to actuate the robot. This approach to mobile robot wheel geometry is very common due to its simplicity and maneuverability.



(a) Bottom View

(b) Front View

Figure 1.1: Main QBot 2 Hardware Components

The motion of each wheel is measured using encoders, and the robot's orientation, or yaw angle, are estimated using the integrated gyro. For more information on the Kinematics of the QBot 2, and how you can generate wheel commands to achieve specific motion trajectories, refer to the Forward/Inverse and Differential Kinematics laboratory experiments. You will also learn how the measured sensory information is used for odometric localization.

In addition to the encoders and gyro, the QBot 2 comes with a Microsoft Kinect for vision, shown in Figure 1.1 b, that outputs color image frames (RGB) as well as depth information. You can process RGB and depth data for various purposes including visual inspection, 2D and 3D occupancy grid mapping, visual odometry, etc. For more information on these concepts and more, refer to the Computer Vision laboratory experiments.

The QBot 2 also comes with integrated bump sensors (left, right and central), and cliff sensors (left, right, and central) as shown in Figure 1.1 a. These sensors can be used in a control algorithm to avoid obstacles, or prevent damage to the robot.

1.2 QBot 2 Software and Communication

The QBot 2 for QUARC leverages Quanser QUARC Rapid Control Prototyping software which seamlessly integrates with MATLAB/Simulink Software to provide real-time communication and interfacing to the components of the QBot 2. QUARC extends the code generation capabilities of Simulink to the QBot 2 as an external real-time target. Using QUARC, you can rapidly prototype any algorithm and quickly evaluate it on the device.

To communicate with the QBot 2, the following QUARC blocks are used:

1. Hardware In the Loop (HIL) Initialize block: The HIL Initialize block configures the drivers and hardware interface for the QBot 2
2. HIL Read/Write: The HIL Read/Write blocks are used to read sensory data and drive the motors
3. Kinect Initialize: Used to initialize the Kinect sensor. Maximum frame-rate and resolution are set in this block.
4. Kinect Get Image: Captures RGB data from the Kinect sensor.
5. Kinect Get Depth: Captures depth data from the Kinect sensor.
6. Display Image: Transmits the input data (RGB or depth) from QBot 2 to the PC and displays them on the monitor.

Other than the afore-mentioned blocks, the "Host Initialize" block can be used to make use of external input devices such as a keyboard (Host Keyboard) or joystick (Host Game Controller). These blocks can be seen in the supplied model for the In-Lab portion of this laboratory experiment.

2 In-Lab Exercise

2.1 Wheel Drive Mode

In this experiment, you will command the left and right wheels independently and observe the motion of the robot and vision data from the Kinect sensor. The supplied model for this part of the lab is called "QBot2_Keyboard_Teleop_Wheel.mdl", shown in Figure Figure 2.1. In this model, we use the HIL Initialize block to configure the interface options for the QBot 2, Kinect Initialize for the Kinect sensor, and Host Initialize for the keyboard interface. If you double click on the QBot2_Basic subsystem, and then QBot2_IO_Basic, you will find HIL_Write and HIL_Read blocks used to drive motors and read from the sensors. Take time to explore the model. You can right-click on the blocks and select help to find more useful information regarding each block.

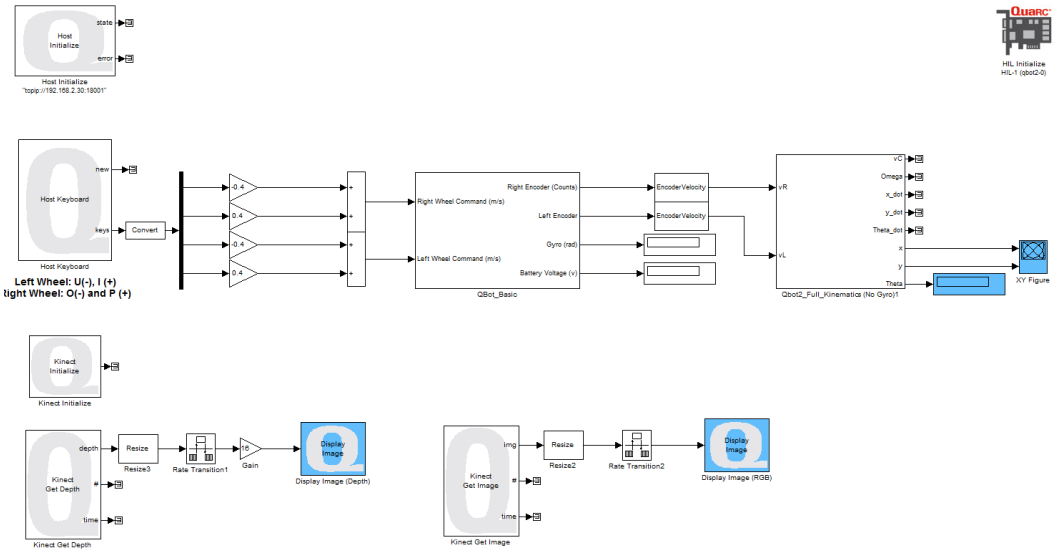


Figure 2.1: Snapshot of the ``QBot2_Keyboard_Teleop_Wheel.mdl" model.

The Kinect Get Depth and Kinect Get Image blocks are used to read the depth and RGB data from the Kinect sensors. We then use the "Display Image" blocks to display these images.

The keyboard controls for this experiments are as follows:

- Left wheel command: U (-) and I (+)
- Right wheel command: O (-) and P (+)

After turning on the device, and connecting to the GSAH wifi network, follow these steps:

1. Compile the supplied model and run it.
2. Double-click on the XY Figure, Display Image (Depth) and Display Image (RGB) blocks.
3. Use the keyboard to command the robot.
4. Describe how the motion of the left/right wheels relate to the actual motion of the robot (forward/backward motion, left/right turn, etc.)
5. Stop the model.

2.2 Normal Vehicle Drive Mode

In this experiment, you will drive the robot in a conventional manner where the robot commands correspond to "move forward/backward" and "turn left/right", similar to the way you would control any vehicle. The controller model for this exercise, shown in Figure Figure 2.2, is called "QBot2_Keyboard_Teleop_Normal.mdl". The QUARC blocks used in this model are similar to the ones described above.

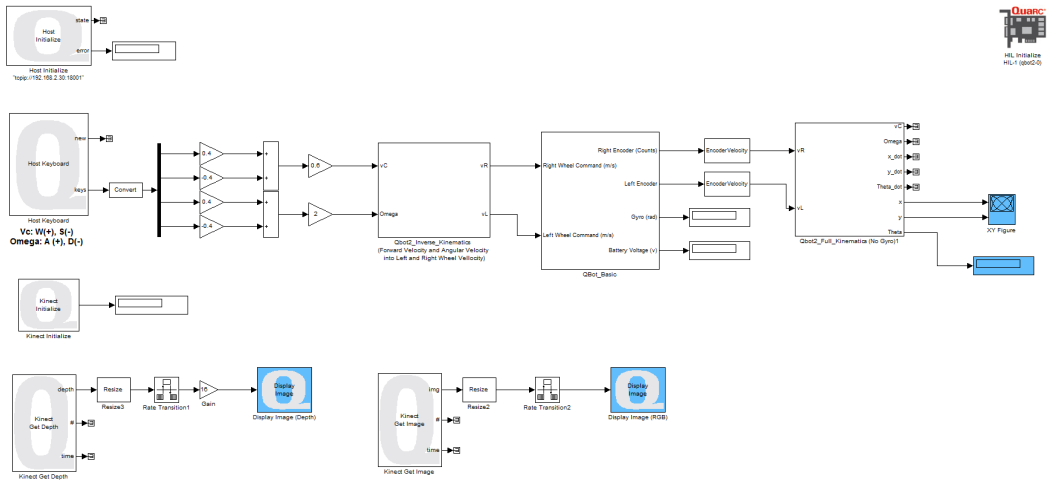


Figure 2.2: Snapshot of the ``QBot2_Keyboard_Teleop_Normal.mdl" model.

Two customized blocks are also used that apply the Inverse Kinematics and Forward Kinematics models of the device, the details of which are presented in the Kinematics laboratory experiment. Keyboard controls for this model are as follows:

- Linear velocity command:W (+) and S (-)
- Angular velocity command: A (+) and D (-)

After turning on the device and connecting to the GSAH wifi network, following these steps to run the model:

1. Open the supplied model, compile the model and run it.
2. Double-click on the XY Figure, Display Image (Depth) and Display Image (RGB).
3. Use the keyboard keys (W, S, A, D) to command the robot. W to move forward, S to move backward, A to turn left, and D to turn right.
4. Observe the RGB and Depth images, as well as the XY Figure, and report your observations.
5. Describe the benefits of controlling the vehicle in normal mode.
6. Stop the model.

Quanser Inc.
119 Spy Court
Markham, Ontario
L3R 5H6
Canada
info@quanser.com
Phone: 1-905-940-3575
Fax: 1-905-940-3576

Printed in Markham, Ontario.

For more information on the solutions Quanser Inc. offers, please visit the web site at:
http://www.quanser.com

EXPERIMENT 1: LOCOMOTION AND KINEMATICS

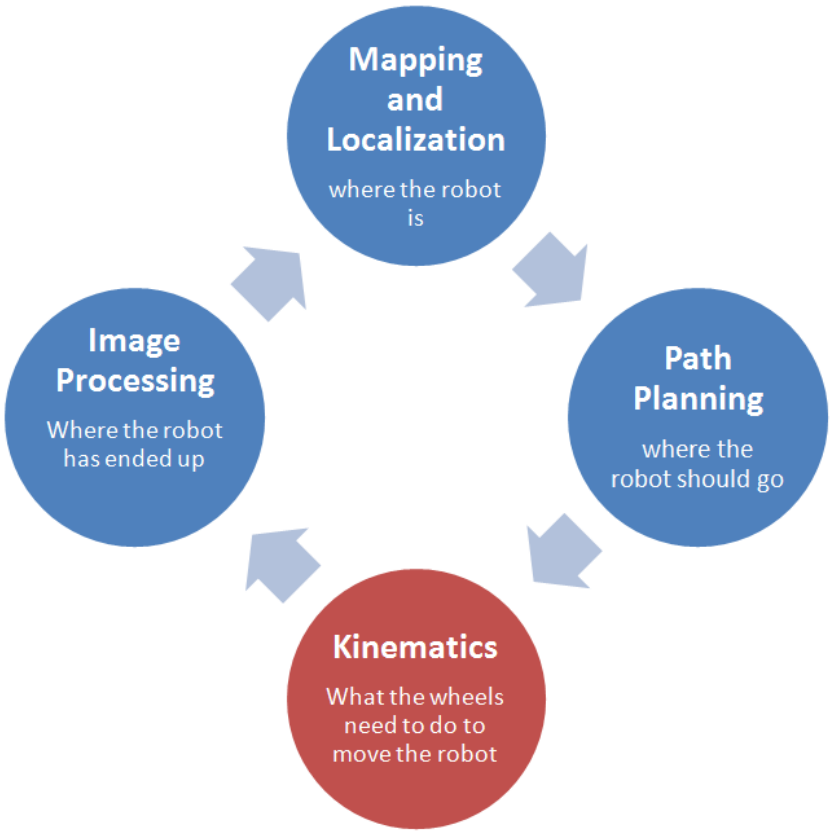
The purpose of this experiment is to study the basic motion behaviour of the Quanser QBot 2 Mobile Platform. The following topics will be studied in this experiment.

Topics Covered

- Differential Drive Kinematics
- Forward and Inverse Kinematics
- Odometric Localization and Dead Reckoning

Prerequisites

- The QBot 2 has been setup and tested. See the QBot 2 Quick Start Guide for details.
- You have access to the QBot 2 User Manual.
- You are familiar with the basics of Matlab® and Simulink®.



Kinematics is used to determine the appropriate actuator commands for the robot

DIFFERENTIAL DRIVE KINEMATICS

The Quanser QBot 2 Mobile Platform uses a drive mechanism known as differential drive. It consists of two central drive wheels mounted on a common axis that bisects the robot. Castors at the front and back of the robot stabilize the platform without compromising movement. Each drive wheel can be independently driven forward and backward, to actuate different motion from the robotic base. This approach to mobile robot wheel geometry is very common due to it's simplicity and maneuverability.

Differential drive kinematics is the mathematical relationship that maps the independent motion of the wheels to the overall movement of the robot chassis. This fundamental topic is the foundation of all mobile robot control, in that it is chiefly responsible for the predictable mobility of the robot. In this laboratory you will investigate the differential drive kinematics of the Quanser QBot 2 Mobile Platform.

Topics Covered

- Differential drive mechanism of the QBot 2
- Derive the kinematics model of the QBot 2 differential drive system

1 Background

The QBot 2 is driven by a set of two coaxial wheels. These wheels are actuated using high-performance DC motors with encoders and drop sensors. To determine the relationship between the independent motion of the two wheels and the motion of the overall robot, we begin by modeling the motion of the robot about a common point.

Let the radius of the wheels be denoted by r, and the wheel rotational speed be denoted by ω_L and ω_R for the left and right wheel respectively. The linear speed of the two wheels along the ground is then given by the following equations:

$$v_L = \omega_L r \tag{1.1}$$

$$v_R = \omega_R r \tag{1.2}$$

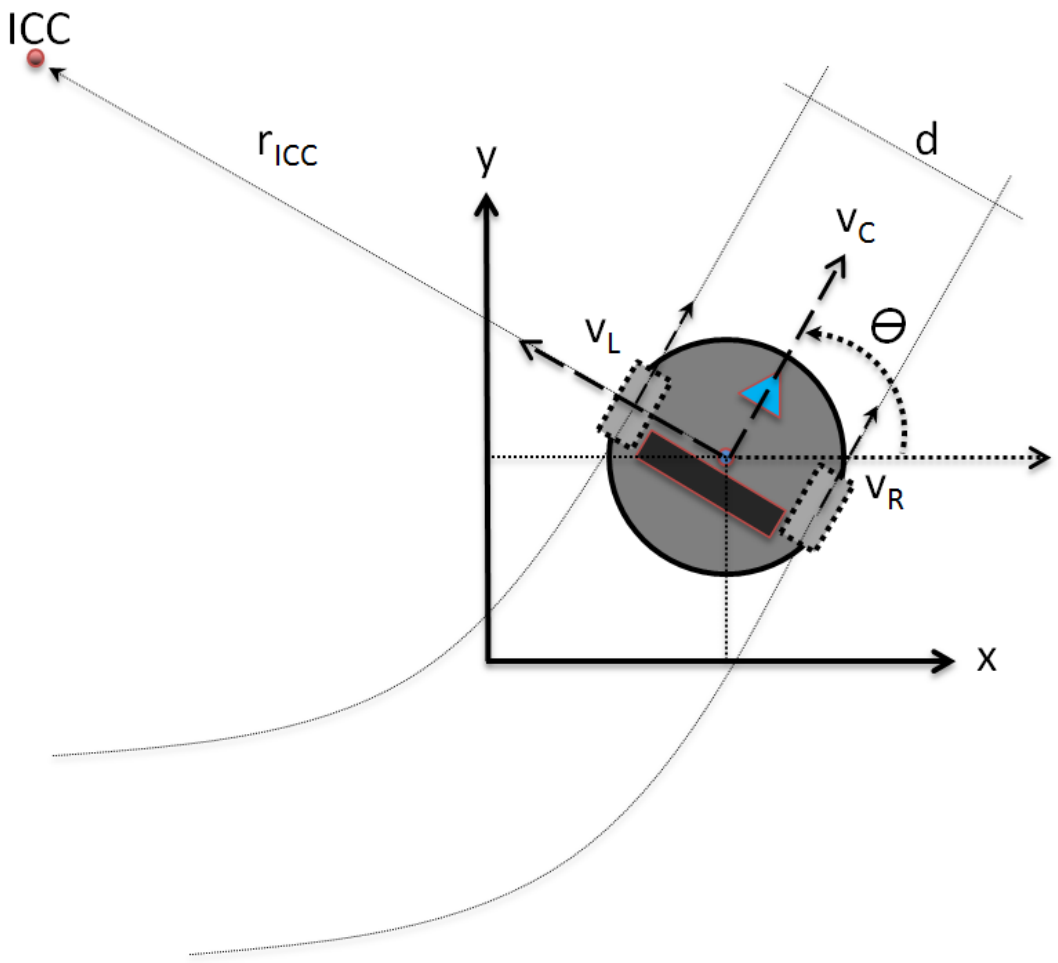


Figure 1.1: Quanser QBot 2 Mobile Platform Reference Frame Definitions

Assuming there is no wheel slippage, the QBot 2 can move along the horizontal plane in straight or curved trajectory, as well as spin on a spot, by varying the relative speed between the left and right wheels.

Since we are assuming that the wheels are not subject to slip, the motion of the wheels are constrained to move along their forward and backward directions. This, together with the inherent constraint that is imposed by the robot chassis coupling the two wheels together, means that all robot chassis rotations must be about a point that lies along the common wheel axis. For example, if only one of the two wheels rotates, the robot would rotate (pivot) about the non-moving wheel. On the other hand, if both wheels rotate at the same speed, the robot rotates about a point infinitely far from the robot. This center of rotation is known as the *Instantaneous Center of Curvature* (ICC).

1.1 Kinematic Model

Let r_{ICC} be the distance measured from the center of the robot chassis, which is halfway between the left and right wheels, to the ICC. If d is the distance between the left and right wheels, θ is the heading angle of the robot, and v_C is the (forward/backward) speed of the robot chassis center, the motion of the QBot 2 chassis can be summarized in the following equations:

$$v_C = \dot{\theta}\, r_{ICC} \tag{1.3}$$

$$v_L = \dot{\theta}\left(r_{ICC} - \frac{d}{2}\right) \tag{1.4}$$

$$v_R = \dot{\theta}\left(r_{ICC} + \frac{d}{2}\right) \tag{1.5}$$

Notice that v_C, v_L and v_R are all defined along the same axis, which lies in the forward/backward direction of the chassis. Given the wheel speed, v_L and v_R, the robot speed, v_C, the angular rate, $\omega_C = \dot{\theta}$, and the distance from ICC, r_{ICC}, we can relate the motion of the wheels to the motion of the robot using the following kinematic model for the differential drive system:

$$v_C = \frac{v_R + v_L}{2} \tag{1.6}$$

$$\omega_C = \dot{\theta} = \frac{v_R - v_L}{d} \tag{1.7}$$

$$r_{ICC} = \frac{d}{2}\frac{(v_R + v_L)}{(v_R - v_L)} \tag{1.8}$$

2 In-Lab Exercise

2.1 Wheel Command Scenarios

The Simulink model for this exercise is "QBot 2_Diff_Drive_Kinematics.mdl" the snapshot of which shown in Figure Figure 2.1.

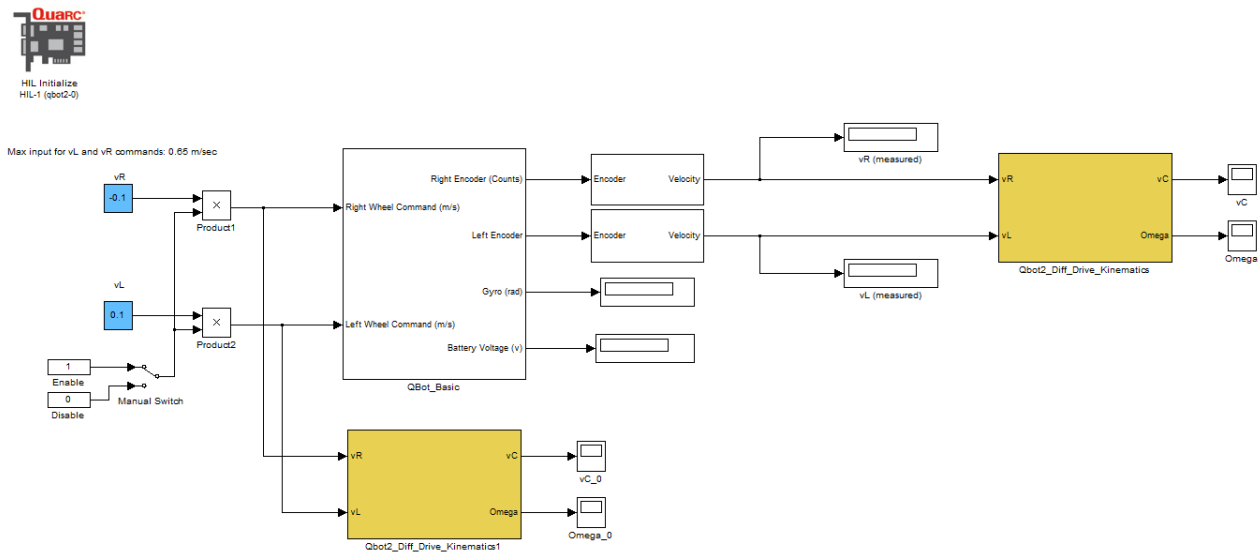


Figure 2.1: Snapshot of the controller model "QBot 2_Diff_Drive_Kinematics.mdl"

The " QBot 2_Diff_Drive_Kinematics" block, shown in yellow, receives the left and right wheel velocities as input and computes the forward and angular velocities. Compile and run the model, then follow the procedure outlined below. Make observations on the motion of the robot, and answer the associated questions.

1. Wait 5 seconds until the QBot 2 has fully initialized, and then enable the movement of the robot using the manual switch shown in Figure 2.1.
2. Set the left and right wheel velocity set points, highlighted with blue, to 0.1 m/s. Run the model and observe the linear and angular velocities. What are the values of r_{ICC} and ω_C when the left and right wheels are moving at the same speed (i.e. $v_L = v_R$)? What do your results indicate about the relationship between r_{ICC} and ω_C?
3. Change the right wheel velocity to -0.1 m/s and keep the left wheel velocity set point at 0.1 m/s. What is the value of r_{ICC} when the left and right wheels are moving at the same speed but in the opposite direction (i.e. $v_L = -v_R$)? What do these results indicate? Does the relationship identified earlier hold?
4. What does it mean when r_{ICC} is negative?
5. What does it mean when ω_C is negative?

FORWARD AND INVERSE KINEMATICS

The objective of this exercise is to investigate the forward and inverse kinematics of the Quanser QBot 2 Mobile Platform. Forward kinematics is used to determine the linear and angular velocity of the robot in the world coordinate frame given robot's wheel speeds. Inverse kinematics on the other hand, is used to determine the wheel commands needed for the robot to follow a specific path at a specific speed. Inverse kinematics is an essential tool for mobile robotics as it bridges the gap between a navigation and path planning module, and actual robot locomotion.

Topics Covered

- Forward kinematics model of the QBot 2
- Inverse kinematics model of the QBot 2

1 Background

A typical kinematics model that computes the robot chassis speed, v_C, and turning rate, ω_C, from the wheel speed, v_R and v_L, with wheel separation distance, d, for a robot with differential drive system like the QBot 2 is given by:

$$v_C = \frac{1}{2}(v_R + v_L) \tag{1.1}$$

$$\omega_C = \dot{\theta} = \frac{1}{d}(v_R - v_L) \tag{1.2}$$

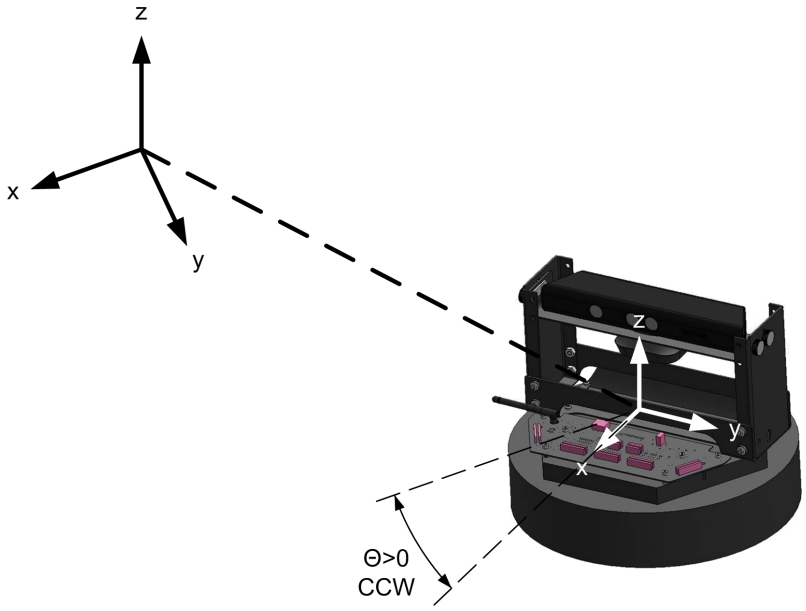


Figure 1.1: Kinematics is used to determine the appropriate actuator commands for the robot

Implicit in the derivation of the above kinematics model is the use of a local frame of reference. In other words, the chassis speed, v_C, is expressed in the forward/backward (heading) direction of the robot chassis and not the global frame that would be used in a map of the environment. Since the robot chassis heading changes when the angular rate is non-zero, $\omega_C = \dot{\theta}$, we need to apply a transformation to the differential drive kinematics model in order to compute the robot chassis motion with respect to the global reference frame. For a robot with a heading, θ, the transformation required is the following rotation matrix:

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

This transformation maps motion expressed with respect to the robot chassis local frame to the corresponding motion in the global frame.

The corresponding inverse mapping is given as follows:

$$R^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.4}$$

1.1 Forward Kinematic Model

We define a state vector, S, as the position, x and y, and the heading, θ, of the robot chassis. Its definition and rate of change are given as follows:

$$S = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}, \qquad \dot{S} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$$

The x and y axes lie in the "ground" plane that the robot primarily travels in. The heading, θ, is measured about the vertical z axis, which is defined as positive pointing upwards. The heading is zero, ($\theta = 0$), when the robot chassis' forward direction aligns with the global x axis. The rate of change of the states can be expressed in terms of the robot chassis speed, v_C, and angular rate, ω_C, as follows:

$$\dot{S} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = R \begin{bmatrix} v_C \\ 0 \\ \omega_C \end{bmatrix} = \begin{bmatrix} v_C \cos\theta \\ v_C \sin\theta \\ \omega_C \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(v_R + v_L)\cos\theta \\ \frac{1}{2}(v_R + v_L)\sin\theta \\ \frac{1}{d}(v_R - v_L) \end{bmatrix} \tag{1.5}$$

Equation 1.5 represents the forward kinematics model for the QBot 2 that computes the linear speed, ($\dot{x}$ and $\dot{y}$), and turning rate, (ω_C), of the robot chassis given its heading, (θ), and wheel speed, (v_R and v_L).

Similarly, the position of the Instantaneous Center of Curvature (ICC) in space, (x_{ICC} and y_{ICC}), expressed with respect to the global reference frame can be obtained as follows:

$$\begin{bmatrix} x_{ICC} \\ y_{ICC} \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} + R \begin{bmatrix} 0 \\ r_{ICC} \\ 0 \end{bmatrix} = \begin{bmatrix} x - r_{ICC}\sin\theta \\ y + r_{ICC}\cos\theta \\ 0 \end{bmatrix} = \begin{bmatrix} x - \frac{d}{2}\frac{(v_R+v_L)}{(v_R-v_L)}\sin\theta \\ y + \frac{d}{2}\frac{(v_R+v_L)}{(v_R-v_L)}\cos\theta \\ 0 \end{bmatrix} \tag{1.6}$$

This can be useful for path planning or obstacle avoidance algorithms.

1.2 Inverse Kinematic Model

As mentioned in the Background section, if you want the robot to follow a certain path or speed, you need to send appropriate wheel commands to the robot. The inverse kinematics model computes the required wheel speed to obtain a desired robot chassis speed v_C, and angular rate ω_C. It is obtained by solving Equation 1.1 and Equation 1.2 together for the wheel speed v_R and v_L and is given as follows:

$$\begin{bmatrix} v_R \\ v_L \end{bmatrix} = \begin{bmatrix} v_C + \frac{1}{2}d\,\omega_C \\ v_C - \frac{1}{2}d\,\omega_C \end{bmatrix} \tag{1.7}$$

2 In-Lab Exercise

2.1 Forward Kinematics

The controller model for this exercise, shown in Figure 2.1, is called "QBot 2_Forward_Kinematics.mdl".

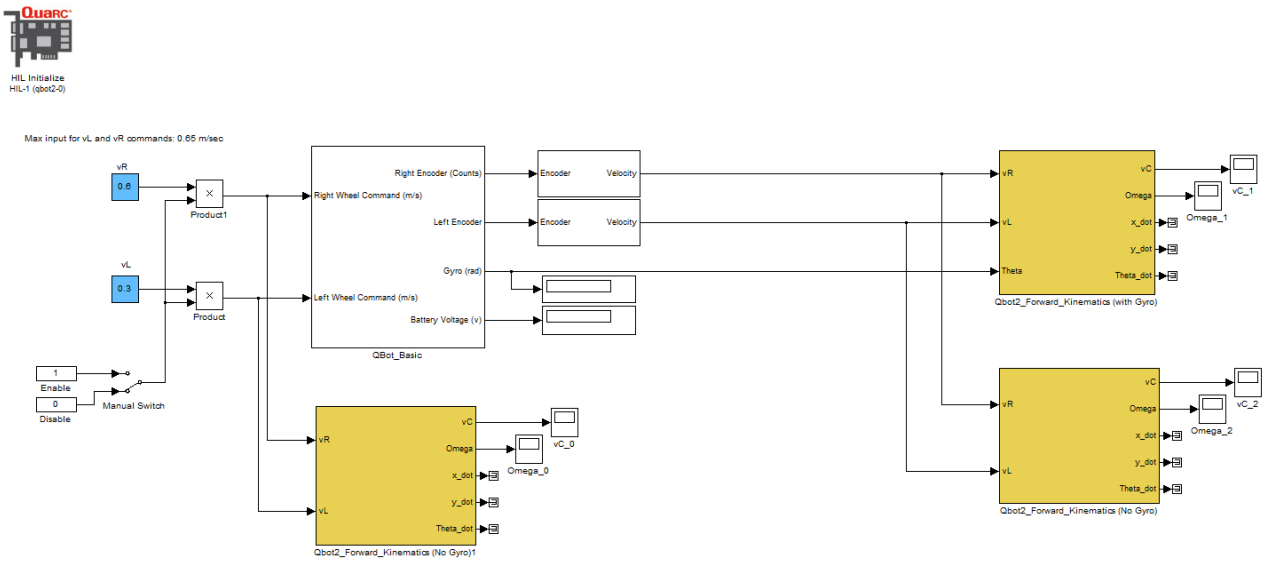


Figure 2.1: Snapshot of the controller model "QBot 2_Forward_Kinematics.mdl"

The "QBot 2_Forward_Kinematics" block, shown in yellow, receives the left and right wheel velocities as inputs and computes the linear velocities in the world coordinate frame. The "QBot 2_Forward_Kinematics" block is used on both commanded wheel velocities giving us the "ideal" robot speed, as well as on the measured wheel velocities, resulting in the actual measured robot velocity.

Compile and run the model, then follow the procedure outlined below. Make observations on the motion of the robot, and answer the associated questions.

1. Wait 5 seconds until the QBot 2 has fully initialized, and then enable the movement of the robot using the manual switch shown in Figure 2.1.
2. Set the left and right wheel velocity set points, highlighted with blue, to 0.6 m/s and 0.3 m/s accordingly. Run the model and observe the ideal and measured linear and angular velocities in the world coordinate frame ($\dot{x}$, $\dot{y}$ and $\dot{\theta}$).
3. What is the shape of the robot trajectory when the right wheel is commanded to travel at twice the speed of the left wheel (i.e. $v_R = 2v_L$)? Generate a x-y plot of the resulting robot chassis trajectory. Comment on the effect of changing the value of v_L on the robot chassis trajectory.
4. Compute the required constant wheel speeds v_R and v_L to generate a trajectory with a constant turning rate of $\omega_C = 0.1\,rad/s$ and a constant turning radius of $r_{ICC} = 1\,m$. Implement the wheel speed command on the robot chassis and record the resulting chassis trajectory. Compare the desired turning rate and radius to the measured turning rate and radius of the robot chassis.

2.2 Inverse Kinematics

The QUARC model for this exercise is called "QBot 2_Inverse_Kinematics.mdl" and is shown in Figure 2.2. In this model, the Inverse Kinematics block for the QBot 2 is shown in yellow and the input commands for v_C and ω_C are highlighted in blue.

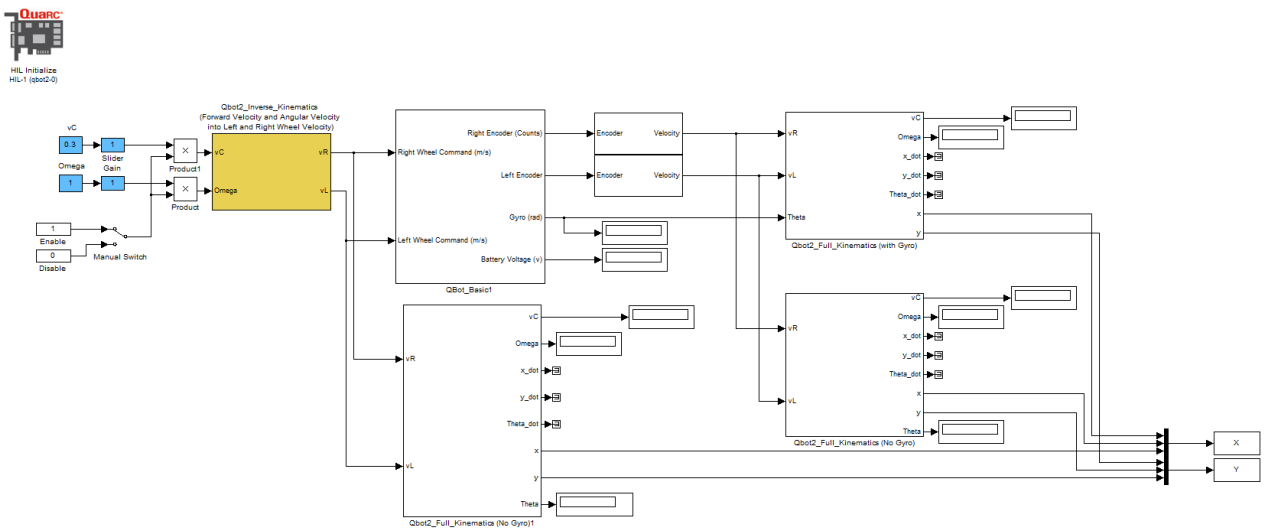


Figure 2.2: Snapshot of the controller model "QBot 2_Inverse_Kinematics.mdl"

Open the supplied Inverse Kinematics controller model, compile it and go through the following steps. For each step observe the motion of the robot chassis and answer the associated questions.

1. Wait 5 seconds until the QBot 2 has fully initialized, and then enable the movement of the robot using the manual switch shown in Figure 2.2.
2. Set the desired forward speed $v_C = 0.1$ m/s and the desired turning rate ω_C = 0.1 rad/s.
3. Run the model.
4. Record and observe the corresponding wheel speed commands v_R and v_L. Compare them with the wheel speeds computed in the Forward Kinematics exercise.
5. Set the desired forward speed $v_C = 0$ m/s and the desired turning rate $\omega_C = 0.2$ rad/s and run the model again. Observe the behaviour of the robot as well as the measured v_L and v_R signals.

ODOMETRIC LOCALIZATION AND DEAD RECKONING

The objective of this exercise is to explore the concept of odometric localization as applying to the Quanser QBot 2 Mobile Platform.

Topics Covered

- Equations of motion for odometry
- Accumulated errors

1 Background

Odometric Localization, also known as Dead Reckoning, is the estimation of a robot's position and orientation (pose) based on the measured or estimated motion of the robot. In the case of the Quanser QBot 2 Mobile Platform, the procedure for odometric localization involves estimating the wheel speeds, ($v_R(t)$ and $v_L(t)$), based on encoder data or the integrated gyro. The forward kinematics model is then applied to estimate the robot chassis' linear speed, $v_C(t)$, and angular rate, $\omega_C(t)$. The data is then integrated over time starting from a known initial location, ($x(0)$ and $y(0)$), and heading, $\theta(0)$, to obtain an estimate of the robot chassis' pose.

This approach to localization is the most basic methodology used in mobile robotics, but is still routinely applied in industrial robotics applications that do not require high-fidelity location estimation, or as a redundant backup system for validation and error detection.

1.1 Equations of Motion

Given the robot chassis state vector, $S(t)$, and its rate of change, $\dot{S}(t)$, expressed in the global inertial frame, the robot pose at time, t, can be computed as follows:

$$S(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} = \int_0^t \dot{S}(t)\, dt \tag{1.1}$$

For the QBot 2, given the wheel separation, d, the heading, θ, and the wheel speed, v_R and v_L, the equations of motion for odometric localization are given by:

$$S(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} = \int_0^t \begin{bmatrix} \frac{1}{2}(v_R(t) + v_L(t)) \cos\theta(t) \\ \frac{1}{2}(v_R(t) + v_L(t)) \sin\theta(t) \\ \frac{1}{d}(v_R(t) - v_L(t)) \end{bmatrix} dt \tag{1.2}$$

In MATLAB/Simulink, it is easy to use the built-in integration blocks to solve the odometric calculations. However, for low-level languages, we need to employ other integration methods. For example, for a small time step, δt, the above QBot 2 equations of motion can be approximated as a first order Taylor series expansion:

$$S(t+\delta t) = S(t) + \dot{S}(t)\delta t = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix} + \begin{bmatrix} \frac{1}{2}(v_R(t) + v_L(t)) \cos\theta(t) \\ \frac{1}{2}(v_R(t) + v_L(t)) \sin\theta(t) \\ \frac{1}{d}(v_R(t) - v_L(t)) \end{bmatrix} \delta t \tag{1.3}$$

2 In-Lab Exercise

2.1 Trajectory Errors

The controller model for this exercise, shown in Figure Figure 2.1, is called "QBot 2_Odometric_Localization.mdl".

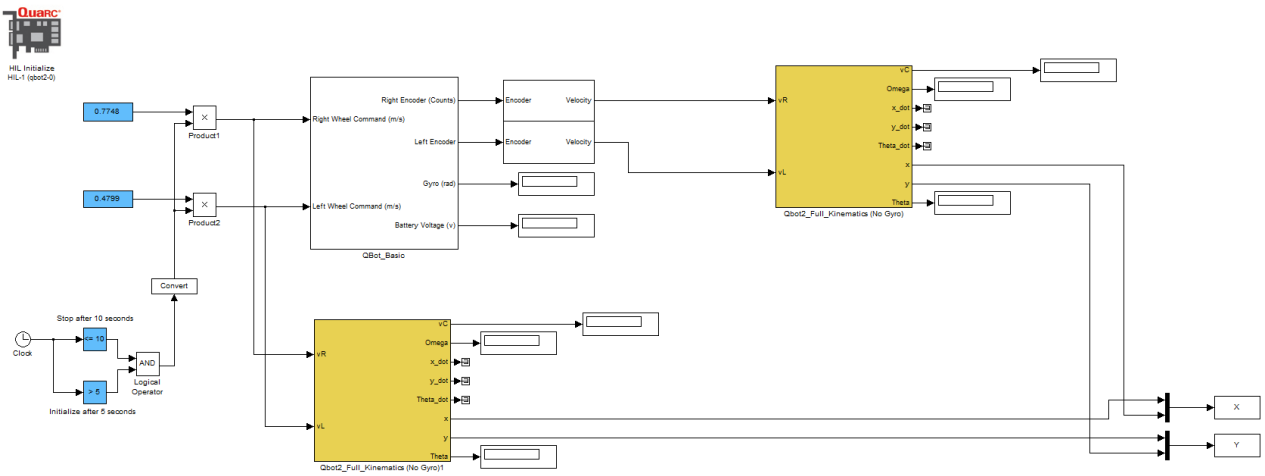


Figure 2.1: Snapshot of the controller model "QBot 2_Odometric_Localization.mdl"

The supplied model includes an open-loop trajectory controller for the QBot 2. The specified path consists of the following segments:

- Rotate in a large circle with a radius of 0.5 m
- Stop when the robot has returned to the initial position

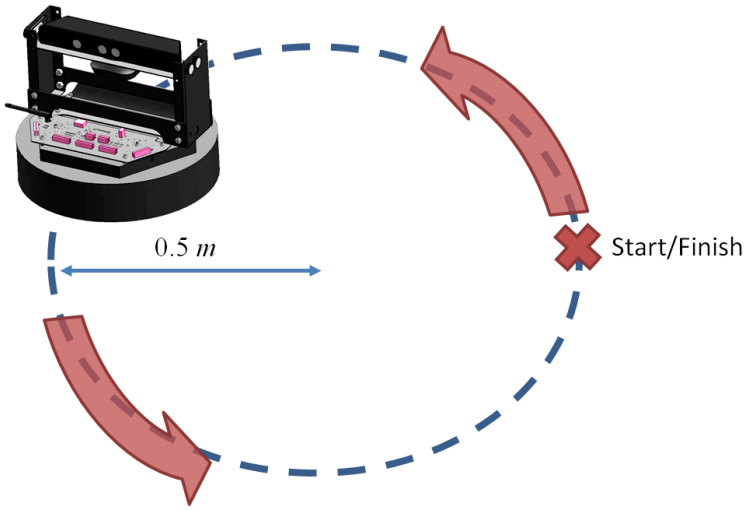


Figure 2.2: Path following controller desired path

At the end of the specified path, the QBot 2 is expected to have returned to approximately the starting point. Compile and run the model, then follow the procedure outlined below. Make observations on the motion of the robot, and answer the associated questions.

Note: It is helpful to mark the starting location and heading prior to running the supplied controller model.

1. After 5 seconds, the QBot 2 will have fully initialized, and will begin to move.
2. Measure and record the final position and heading of the robot chassis with respect to the starting position and heading.

   Note: A MATLAB function called "plotXY.m" has been provided to generate an appropriate plot for the path analysis.

   - What is the error between the desired end-point of the robot, and the actual final x-y position? Recall, the x axis is in the forward/backward direction of the robot and the y axis is in the left/right direction.
   - What is the heading error?
3. Determine the theoretical right wheel and left wheel commands, and time required, for the QBot to travel in a straight line 5 meters in length. Enter your values into the appropriate fields in the controller model, indicated in blue and shown in Figure 2.2.
4. Repeat steps 1 and 2, and record your new results. Modify the wheel commands and time values until the robot is able to reach the correct position.
5. Identify and discuss the different factors that contribute to the path/trajectory tracking error. Specifically, note the error reported by the controller model and the error based on direct physical measurements.

Quanser Inc.
119 Spy Court
Markham, Ontario
L3R 5H6
Canada
info@quanser.com
Phone: 1-905-940-3575
Fax: 1-905-940-3576

Printed in Markham, Ontario.

For more information on the solutions Quanser Inc. offers, please visit the web site at:
http://www.quanser.com

EXPERIMENT 2: MAPPING AND LOCALIZATION

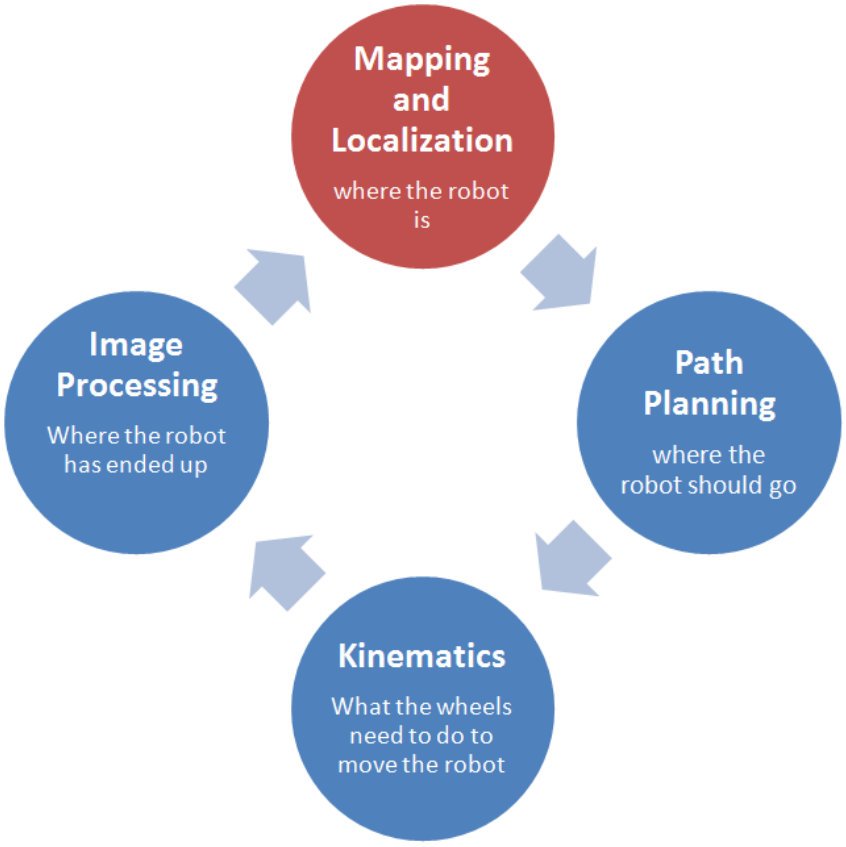
The purpose of this experiment is to create algorithms to map the environment around the Quanser QBot 2 Mobile Platform, and localize the robot inside that environment. The following topics will be studied in this experiment.

Topics Covered

- Occupancy Grid Mapping
- Particle Filtering

Prerequisites

- The QBot 2 has been setup and tested. See the QBot 2 Quick Start Guide for details.
- You have access to the QBot 2 User Manual.
- You are familiar with the basics of Matlab® and Simulink®.



Mapping and localization is used to determine where the robot is

OCCUPANCY GRID MAPPING

In this lab you will learn how to use the on-board sensors of the Quanser QBot 2 Mobile Platform to autonomously build a map of the robotic environment through directed exploration.

Topics Covered

- Gathering and interpreting depth data from the Kinect sensor mounted on the QBot 2
- Autonomous 2D mapping of the surrounding environment QBot 2

1 Background

The Quanser QBot 2 Mobile Platform comes with a Microsoft Kinect sensor that has the ability to generate a depth map of the environment. This information, along with the location and orientation of the robot chassis, can be used for autonomous map building. To generate a 2D map, we will use the planar data received from the Kinect depth image.

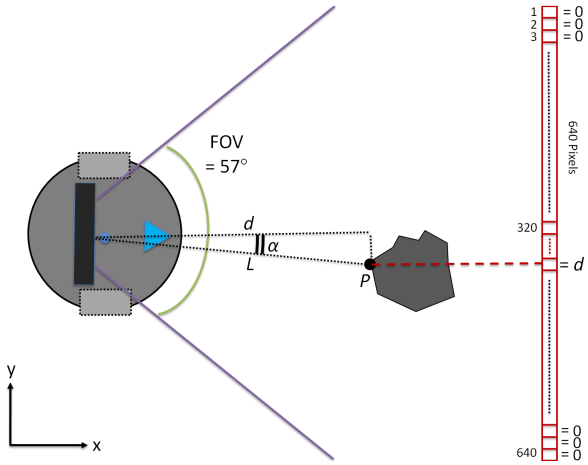


Figure 1.1: Depth data received from Kinect sensor mounted on the Quanser QBot 2 Mobile Platform.

As is the case with any sensor, the Kinect has some limitations:

- Range: The Kinect sensor mounted on the QBot 2 has the ability to determine the distance to an object located between 0.5 m and 5 m. If an object is closer than 0.5 m or further than 5 m, the data is invalid and it will be zeroed down in the software. To map objects beyond this range, the robot should be moved accordingly, or other types of sensors can be used.
- Field of View: The horizontal field-of-view of the Kinect is limited to 57 °. Therefore, to map the entire 360 °, the robot should be rotated accordingly.

Depth data received from the Kinect sensor in QUARC is represented as a 480 $\times$ 640 depth image. For simplicity, we will only use one row of the depth image data for 2D mapping (each row includes 640 pixels) as illustrated in Figure 1.1.

The value of each pixel represents the distance (in millimeters) from the camera to the object, and should be mapped to the corresponding (x, y) point in the world coordinate frame. For example, assume we want to map the point *P*, shown in Figure 1.1, represented by the 400th pixel of the centre row to the world coordinate frame. The value of this pixel received from the Kinect sensor is d, which is the distance of the point *P* to the camera plane. Therefore $P_x = d$ mm. To calculate P_y the angle α is required as $P_y = L\sin(\alpha) = d\tan(\alpha)$.

The angle α can be calculated based on the distance from the desired point to the centre of the row (in pixels), and the horizontal FOV of the Kinect sensor as follows:

$$\alpha = (400 - 320) \times 57/640 = 7.12°,$$

where 320 refers to the central pixel of the sensor data. Therefore, we have $P_y = d\tan(11.6)$, or $P = (d, d\times\tan(7.12))$ in the local coordinate frame of the QBot 2. Using odometric data, we can map the point *P* in the world coordinate frame if we know the pose of the robot, $(x,\ y,\ \theta)$, where θ is the QBot 2 heading. Because odometric data initializes to zero when you start the robot, the origin of the map that you create is based on the initial location of the QBot 2.

As the robot moves around a 2D space, all of the points can be mapped to the world coordinate frame. These points can then be used to generate an occupancy grid map of that area collectively.

2 In-Lab Exercise

2.1 Autonomous Mapping

In this exercise, you will perform autonomous occupancy grid map generation for the QBot 2. Occupancy grid mapping is required for the robot to become aware of surrounding obstacles. The controller model for this exercise, shown in Figure 2.1, is called "QBot 2_2D_Mapping.mdl".

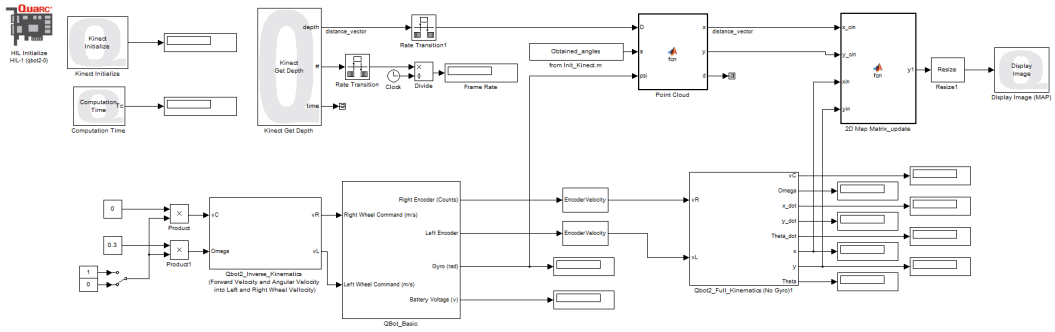


Figure 2.1: Snapshot of the controller model "QBot 2_2D_Mapping.mdl"

Before performing the experiment, make sure the Kinect sensor is flat so that it is parallel with the ground plane and not tilted. The resulting vector is used in the QUARC models to match the depth data to real-world coordinates. Then open the "QBot 2_2D_Mapping.mdl" model, compile it and go through the following exercises.

1. Wait 5 seconds until the QBot 2 has fully initialized, and then enable the movement of the robot using the manual switch shown in Figure 2.1. This is an important step, as it allows the Kinect sensor to fully initialize before the movement starts.
2. Set the left and right wheel speed set-points to zero. Run the model, and look at the video viewer window showing the generated map. Put an object in front of the robot (make sure it is right in front of the Kinect sensor, both horizontally and vertically) and 1 m away. Try to find the object in the map.

   Note: The default scale is set to 10 $\times$ (1 m will show as 10 units on the map).

3. What would happen if you used other rows of the depth image for mapping?
4. Move the object further away from the sensor (e.g. 1.5 m and 2 m) and follow the object on the map. Try moving the object closer to the sensor. Describe your observations and explain the behaviour of the mapping data. Based on your observations, what do the white and gray areas and black dots represent on the map?
5. Stop the model.
6. Configure four objects surrounding the QBot 2 as in Figure 2.2. Now set v_L to 0.2 and v_R to -0.2. Put the robot in a known initial configuration (facing the x axis on your map, shown in Figure 2.2) and run the model. Allow the robot to rotate twice and then stop the model. Compare the generated map to the actual environment and examine any errors.

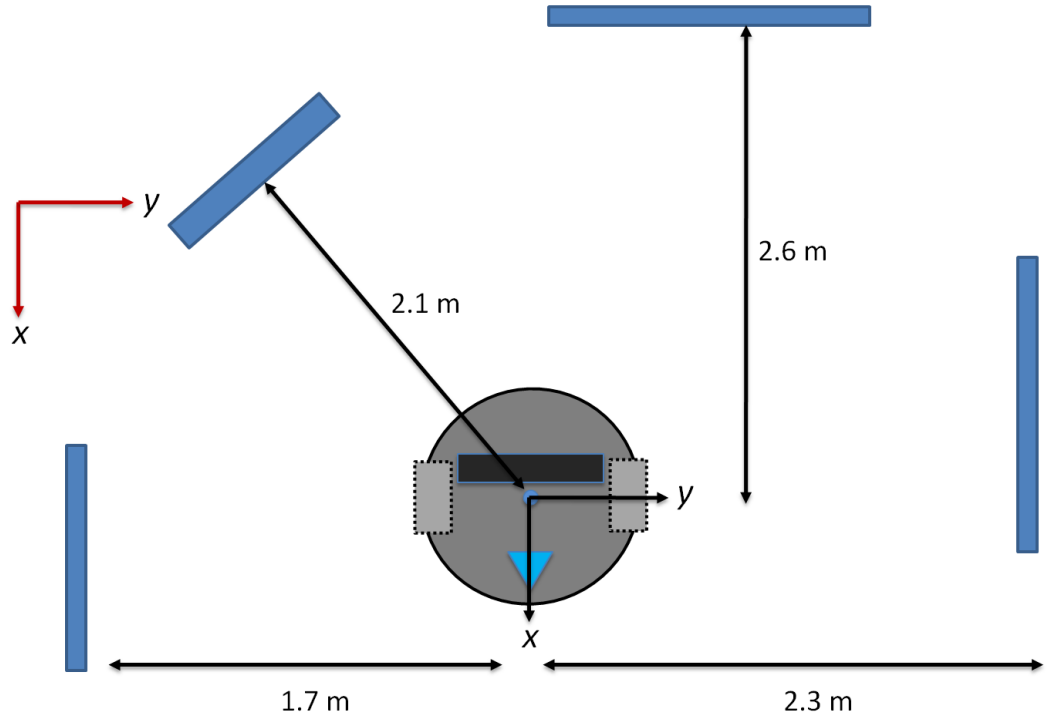


Figure 2.2: Configuration of the objects surrounding the QBot 2 for autonomous mapping.

7. Right click on the image (from Display Image block) and click "save to workspace". Name the variable as "Map_1". Go to the MATLAB workspace, find the Map_1 workspace variable, right click on it and save it to Map1.mat file for future reference.
8. What do you think are the potential sources of error in your measurements?
9. Explain how you could generate a 3D map of the environment (Vertical FOV of Kinect is 43 °).
10. Tilt the Kinect sensor down to the maximum angle allowed which is $43/2 = 21.5$ degrees. Which row of the depth data should be used so that the occupancy grid map is not tilted?

Figure 2.3: Relative FOV of the tilted Kinect sensor

11. Change the parameter "LINE" in the "Point Cloud" embedded m-function to your desired row. Compile and run the program to confirm your findings.
12. Stop the model, and tilt the Kinect sensor back to the original pose.

ROBOT LOCALIZATION USING PARTICLE FILTERING

In this Lab you will learn how to use the depth and vision sensors of the Quanser QBot 2 Mobile Platform to accurately localize the robot.

Topics Covered

- Basic Knowledge of Particle Filtering
- Robot Localization using Particles

1 Background

Particle Filters are very versatile algorithms that can be used to "track" variables of interest as they evolve over time. For the purposes of robot localization, particle filtering is used to track the pose (position and orientation) of the QBot 2 in a given 2D map using the Microsoft Kinect sensor depth data as *sensory information*.

This algorithm first creates and randomly initializes multiple copies of the variable set, known as *particles*. For our experiment, each particle is essentially a copy of the QBot 2 including its position and orientation. Each particle is associated with a weight that signifies the accuracy of that specific particle's location. An estimate of the variable of interest can be obtained using the weighted sum of all particles. The particle filtering algorithm is iterative, with two fundamental phases as follows:

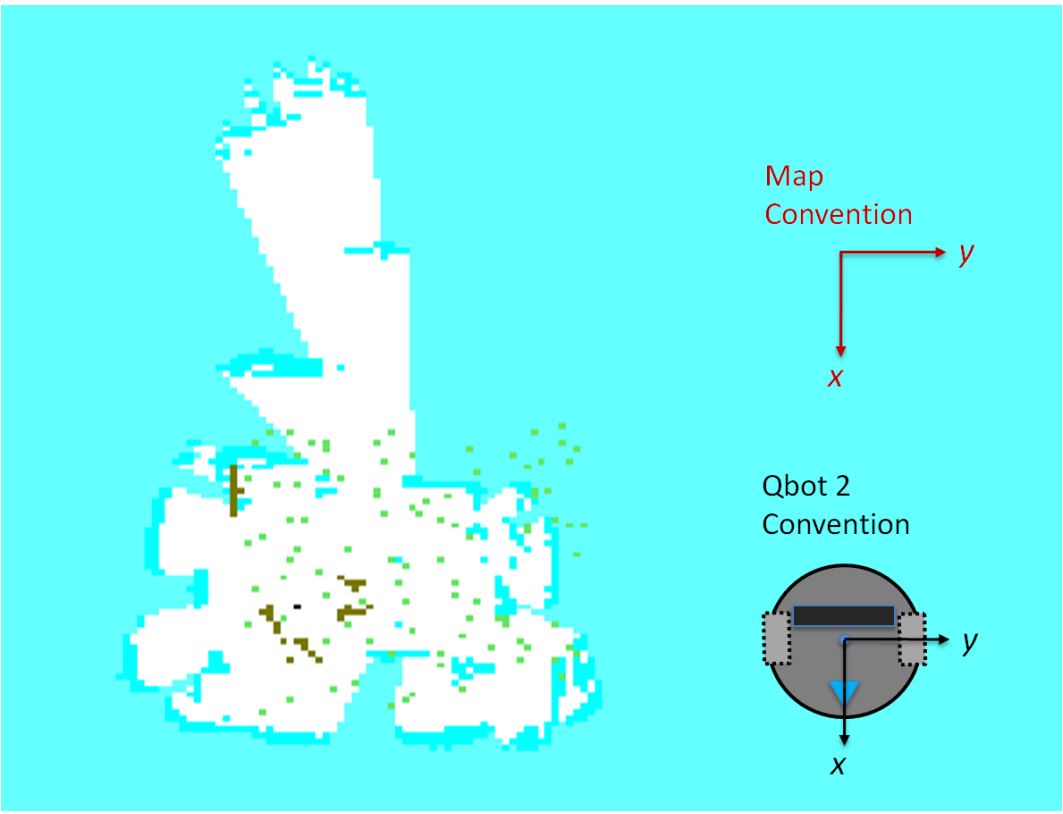


Figure 1.1: Example of the particle filtering algorithm used for localization of the QBot 2. the green dots represent the initial values for the particles, the red dots show the particles updated in real-time, and the black dot shows odometric data localization

- Prediction: In the prediction stage, the location of each particle is predicted according to the existing data and additional random noise to simulate the effect of sensory noise.
- Update: Then the weights of each particle are then updated based on the latest sensory information available.

At the end of the two stages the particles are evaluated, and the ones with small weights are eliminated. There are several particle filtering variations that utilize alternative methods to predict the particle locations and update the gains. In this experiment, we have kept the algorithm as simple as possible to convey the fundamental concept. The following describes the main steps in the algorithm used in this experiment:

1. Initialize N particles that contain position and orientation variables. Then randomly initialize the particles in a 2D map based on the size of the area around the robot.

2. Initialize the weights, W_i, where $i \in \{1, 2, .., N\} = 1/N$.
3. Move the QBot 2 robot by along both axis by Δ_x and Δ_y.
4. Apply the same motion (Δ_x and Δ_y) to all particles with added noise and update all particles.
5. Determine what features the robot would see (*sensory information*) if it had the pose of each particle in the given 2D map.
6. Compare the actual sensory data from QBot 2 with each particle's and determine the error.
7. Update the weights of all particles based on the error. The closer a particle's data is to the QBot 2 sensory information, the higher the weight.
8. Continue iterating through steps 3-7 until the particles converge on a single location.

Every time the robot moves to a new location, the particles will spread as they are actuated in a similar way, and errors are added to their location. Their locations will then converge after a few samples as the weights are adjusted. The sensory data received from Kinect can be used either in the raw format (individual pixels) or in the processed form (features such as edges or corners).

Figure 1.1 shows an example of a particle filtering algorithm for localization of the QBot 2. Here the green dots represent the initial particles, and the red dots show the particles after the robot moves for a few seconds. The black dot is the location of the data based on pure encoder measurements. It is clear that the particles are converging to the actual location of the robot within the map.

2 In-Lab Exercise

In this exercise, you will become familiar with particle filtering, an iterative algorithm used for localization. The controller model for this exercise, shown in Figure 2.1, is called "QBot 2_Particle_Filtering.mdl".

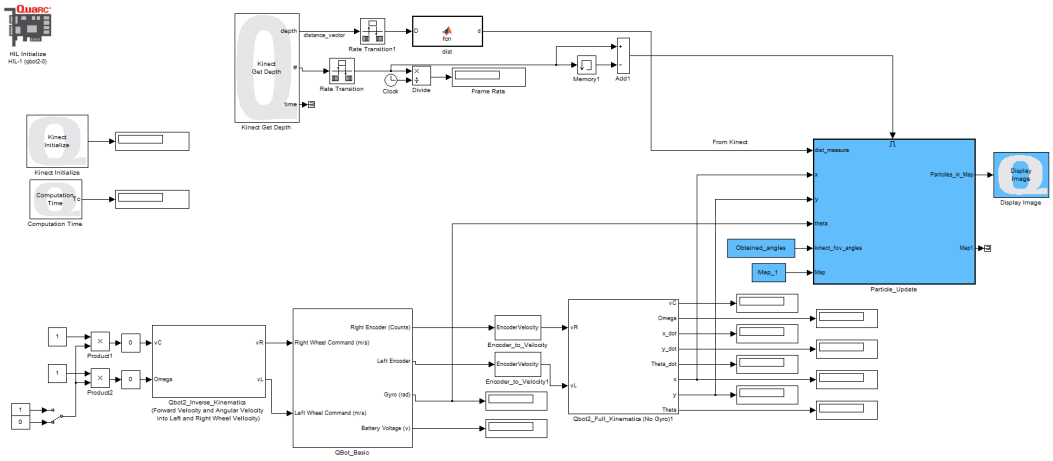


Figure 2.1: Snapshot of the controller model "QBot 2_Particle_Filtering.mdl"

Before the experiment, you will need to load an existing map of the environment by right clicking on the .mat file (e.g. "Map_1.mat") and selecting load.

Make sure the robot's surroundings match the map you will be using. Put the QBot 2 in an initial location in the space, making sure the axes of the QBot 2 match of those of the map you have loaded as shown in Figure 1.1. Then go through the following steps.

1. Wait 5 seconds until the QBot 2 has fully initialized, and then enable the movement of the robot using the manual switch shown in Figure 2.1. This is an important step to insure that the sensor has fully initialized.
2. Make sure the QBot 2 is facing towards the x axis of the environment map as shown in Figure 1.1. The robot can be anywhere within your mapped region, but make sure it is no closer to any objects than 0.5 m so that you get valid non-zero data from the Kinect sensor. To better understand this concept, try placing the robot about 0.5 m away from the origin of the map (the initial location of the QBot 2 when you created the map). In this case the odometric data will not represent the exact location of the QBot 2 (odometric data is always reset to zero when you start the robot, which is the origin of the map).
3. Compile and run "QBot 2_Particle_Filtering.mdl". Observe the initial particles in green, and their real-time location in red as well as the QBot 2 odometric location in black (which initially should be zero).
4. Wait for several seconds and observe the behavior of the particles (red dots).
5. Start rotating the robot slowly using the motor command slider gains and observe the particles as they update their location. Give the algorithm a few seconds to converge.
6. Now move the robot slowly within the map boundaries. Describe your observations.
7. Why do you think the particles tend to converge to multiple locations during the experiment?
8. Without blocking Kinect sensor, approach the QBot 2 from behind and pick it up. Ask a friend to drive the robot so that the odometric data changes (by about 0.5 m) without the robot moving, then put the robot on the same location. Wait for the particles (red dots) to move and converge, and observe the black dot as well as the red dots. Explain the results of your observations.
9. Pick up the QBot 2 and manually move it 0.5 m closer to an obstacle. Wait for the particles to move and observe the black dot and the red dots. Compare your results to the previous case.

10. Double click on the "Particle_Update" block in the "QBot 2_Particle_Filtering.mdl" and open up the "particle_filter" MATLAB function. Try to match the 8 steps described in Section 1 with the code.

11. Change the number of the particles in the function to 10, compile the model, and run through the above steps (steps 2 to 8). Observe the behaviour of the algorithm with 10, 20, and 50 particles and compare your results to the previous cases.

12. Pick the most successful number of the particles from the previous step. Change the Gaussian noise distributions: "sigma_measure", "sigma_move", and "sigma_rotate", that represent the noise of sensor measurements by multiplying them by 2 and 0.5. Compare your results to the previous cases.

13. Can you think of other methods to modify the algorithm? Discuss your thoughts.

Quanser Inc.
119 Spy Court
Markham, Ontario
L3R 5H6
Canada
info@quanser.com
Phone: 1-905-940-3575
Fax: 1-905-940-3576

Printed in Markham, Ontario.

For more information on the solutions Quanser Inc. offers, please visit the web site at:
http://www.quanser.com

EXPERIMENT 3: COMPUTER VISION AND VISION-GUIDED CONTROL

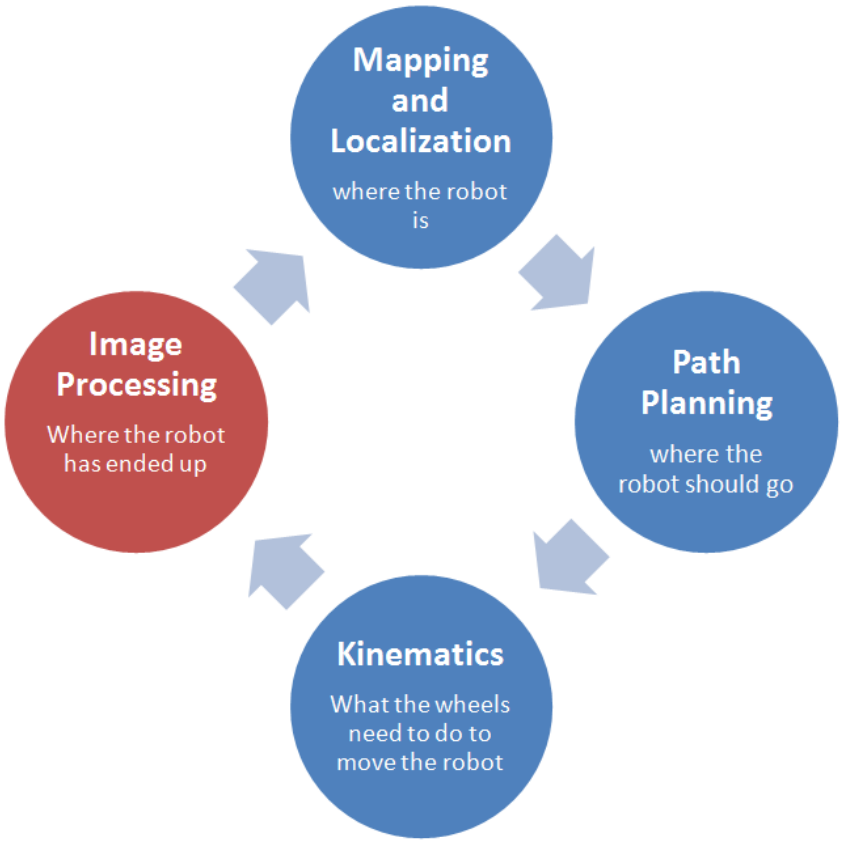
The objective of this experiment is to study computer vision for robotic applications using the QBot 2. The following topics will be studies in this laboratory.

Topics Covered

- Image Processing Techniques
- Reasoning and Motion Planning
- Visual Odometry

Prerequisites

- The QBot 2 has been setup and tested. See the QBot 2 Quick Start Guide for details.
- You have access to the QBot 2 User Manual.
- You are familiar with the basics of Matlab® and Simulink®.



Vision and image processing is used to detect properties of the environment around a robot

IMAGE PROCESSING

The field of digital image processing is chiefly concerned with the processing of digital images using computers. It normally consists of the development of processes and algorithms whose inputs and outputs are images, and extract attributes from images such as lines, corners, specific colors, and object locations. For the field of robotics, image processing is often used for navigation and mapping, but can also be used for more advanced topics including facial recognition, dynamic path planning, etc.

The objective of this exercise is to explore some useful image processing techniques using the Quanser QBot 2 Mobile Platform.

Topics Covered

- Image Thresholding
- Edge Detection
- Blob Analysis

1 Background

Visual sensing plays a key role in robotic applications. Mobile robots use visual feedback to build an internal representation of the environment, which is used in the decision-making process of the robot for motion control. Figure 1.1 describes a typical vision-based robotic application which consists of the following steps

1. **Perception**, which includes image acquisition and processing
2. **Localization and Path Planning**
3. **Motion Control**, which includes kinematics and motion control

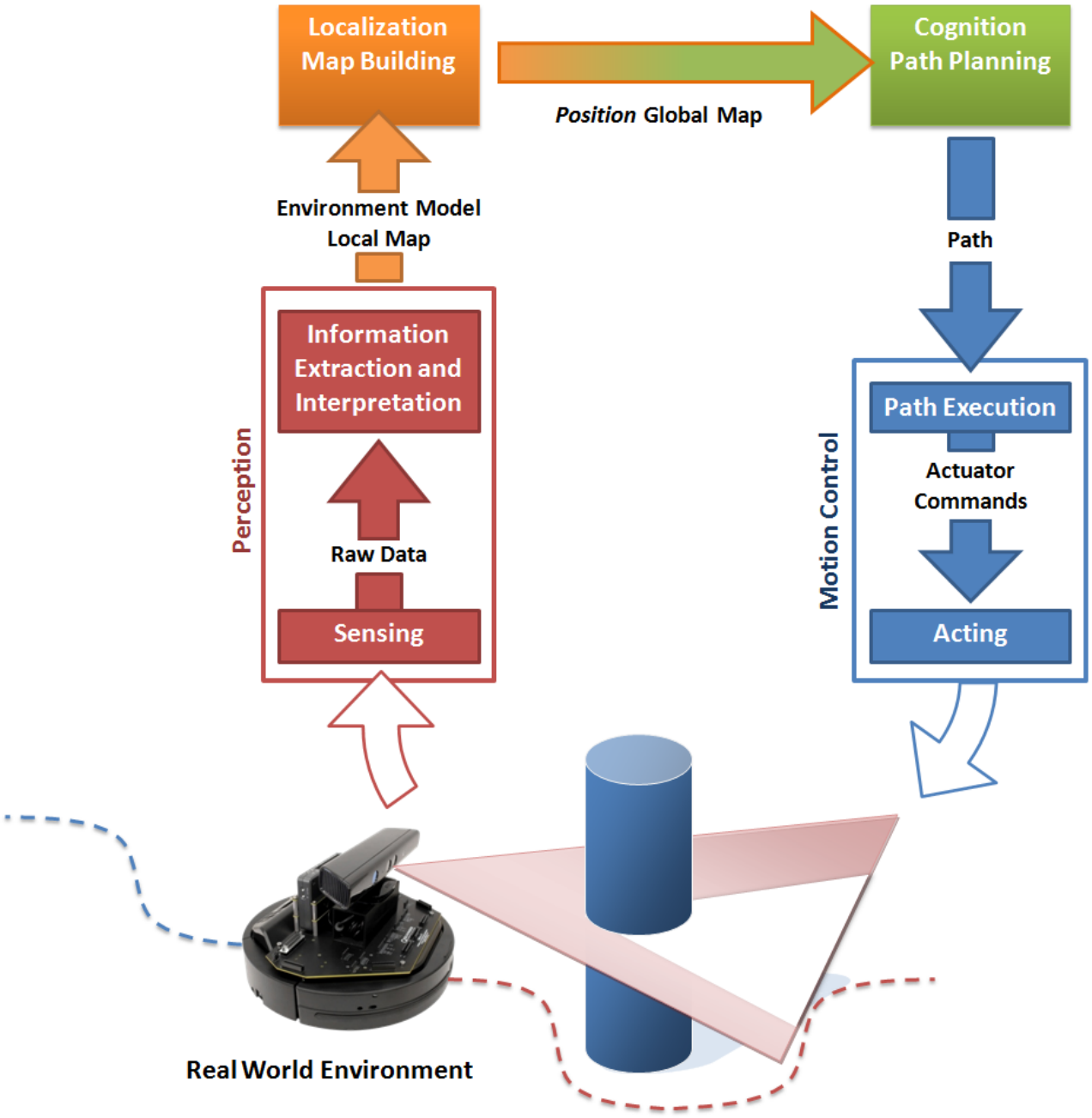


Figure 1.1: Vision-based robotic applications.

The following sections briefly describes the individual components of the block diagram:

1.1 Perception: Image Acquisition and Processing

A digital image can be defined as a two-dimensional function, $f(x, y)$, where x and y are spatial coordinates, and the amplitude of f at any pair of coordinates (x, y) may be a scalar or a three-element vector. When the scalar represents a value proportional to the energy of the visual spectrum of the electro magnetic (EM) field at the coordinate (x, y), the image is called a gray-scale image. On the other hand, when the vector amplitude represents the energy of red, green, and blue colours in the visible EM spectrum, the image is called a colour image or RGB (Red-Green-Blue).

In digital image acquisition, photo sensors are arranged in a 2-D array where each photo sensor indicates a point in the discrete spatial coordinate and is called a picture element or pixel. The electrical signals from photo sensors are digitized using a quantizer to produce a digital image which can be stored in a memory chip. Therefore, when working with images in this laboratory, you will be dealing with $m \times n$ (for gray-scale images) or $m \times n \times 3$ for colour images, where m is the numbers of pixels in a row (number of columns) and n is the number of the pixels in a column (number of rows).

Once an image has been acquired, it can be processed. Image processing fundamentally involves the manipulation of digital images using a computer. Image processing techniques are widely used for visual-based control of mobile vehicles. The following sub-sections briefly describe selected image processing techniques covered in the QBot 2 experiments.

1.1.1 Image Thresholding

Thresholding is an operation that is often used to isolate specific colours or brightness levels in an image. In general, a spatial domain process like thresholding is denoted by the following expression:

$$h(x, y) = T\big(f(x, y)\big), \tag{1.1}$$

where $f(x, y)$ is the input image, $h(x, y)$ is the processed image, and T is the thresholding operation that can be implemented on the image pixels in one of the following ways:

- Binary thresholding:
$$T\big(f(x, y)\big) = \begin{cases} 255 & \text{if } th_1 \leq f(x, y) \leq th_2 \\ 0 & \text{otherwise} \end{cases}$$
- Binary thresholding inverted:
$$T\big(f(x, y)\big) = \begin{cases} 0 & \text{if } th_1 \leq f(x, y) \leq th_2 \\ 255 & \text{otherwise} \end{cases}$$
- Truncate to a value
$$T\big(f(x, y)\big) = \begin{cases} trunc & \text{if } th_1 \leq f(x, y) \leq th_2 \\ f(x, y) & \text{otherwise} \end{cases}$$
- Threshold to zero:
$$T\big(f(x, y)\big) = \begin{cases} f(x, y) & \text{if } th_1 \leq f(x, y) \leq th_2 \\ 0 & \text{otherwise} \end{cases}$$
- Threshold to zero, inverted:
$$T\big(f(x, y)\big) = \begin{cases} 0 & \text{if } th_1 \leq f(x, y) \leq th_2 \\ f(x, y) & \text{otherwise} \end{cases}$$

where the threshold range is defined with $[th_1 th_2]$ and $trunc$ denotes an arbitrary level of truncation. The above equations can be used for gray-scale image thresholding directly. For colour image thresholding, similar functions can be applied to each channel of the image. For example, binary thresholding for RGB images can be written as:

$$T\big(f(x,y)\big) = \begin{cases} f(x,y) & \text{if range-condition } = true \\ 0 & \text{otherwise} \end{cases} \tag{1.2}$$

in which range-condition can be written as the logical AND of three conditions, $th_{r,1} \leq f(x,y) \leq th_{r,2}$, $th_{g,1} \leq f_g(x,y) \leq th_{g,2}$ and $th_{b,1} \leq f_b(x,y) \leq th_{b,2}$ where r, g and b subscripts denote the red, green and blue channels, respectively.

1.1.2 Edge Detection

An edge is a set of similar, and connected pixels that lie on the boundary between two regions. Edge detection is performed by convolving an input image with gradient masks. The convolution process involves computing the sum of products of mask (also called window or kernel) coefficients with the gray levels contained in the region encompassed by the mask. For example, Figure 1.2 shows a 3x3 moving mask on an image, where the mask center $w_{0,0}$ coincides with the image location at (x,y). The following equation describes how the processed image, $h(x,y)$, is calculated based on the gray value of the original image $f(x,y)$ at (x,y) using the convolution procedure when the mask size is $(2p+1) \times (2q+1)$.

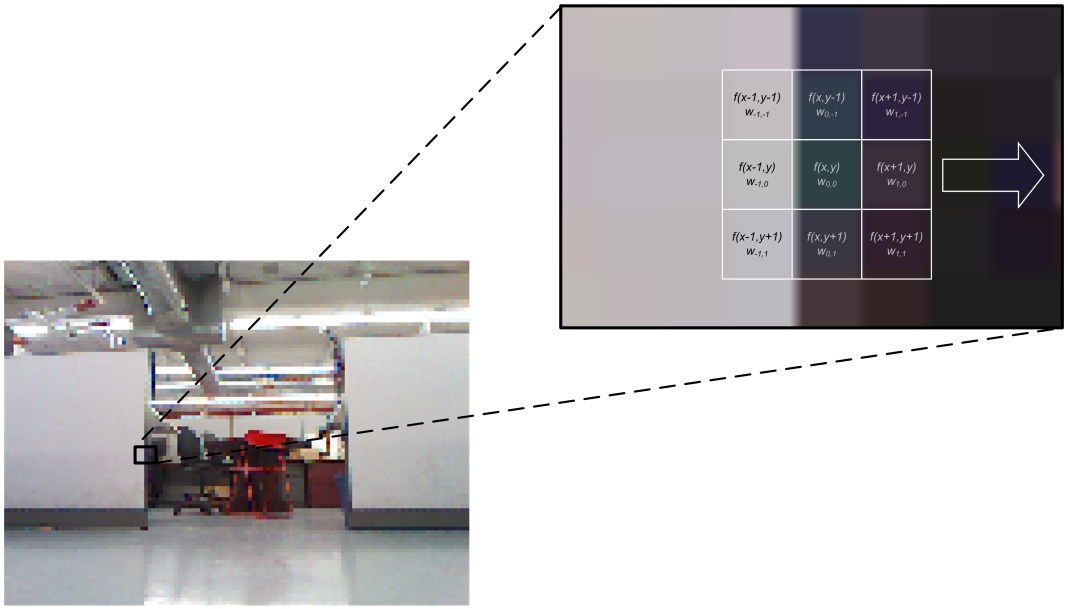


Figure 1.2: Mask operation in image processing.

$$h(x,y) = \sum_{-p \leq i \leq p} \sum_{-q \leq j < \leq q} w_(i,j) \times f(x+i, y+j) \tag{1.3}$$

The edge detection procedure employs horizontal, w_x, and vertical, w_y, gradient masks to determine image gradients $G_x(x,y)$, $G_y(x,y)$ in x and y directions, respectively. The overall gradient image is defined by the following:

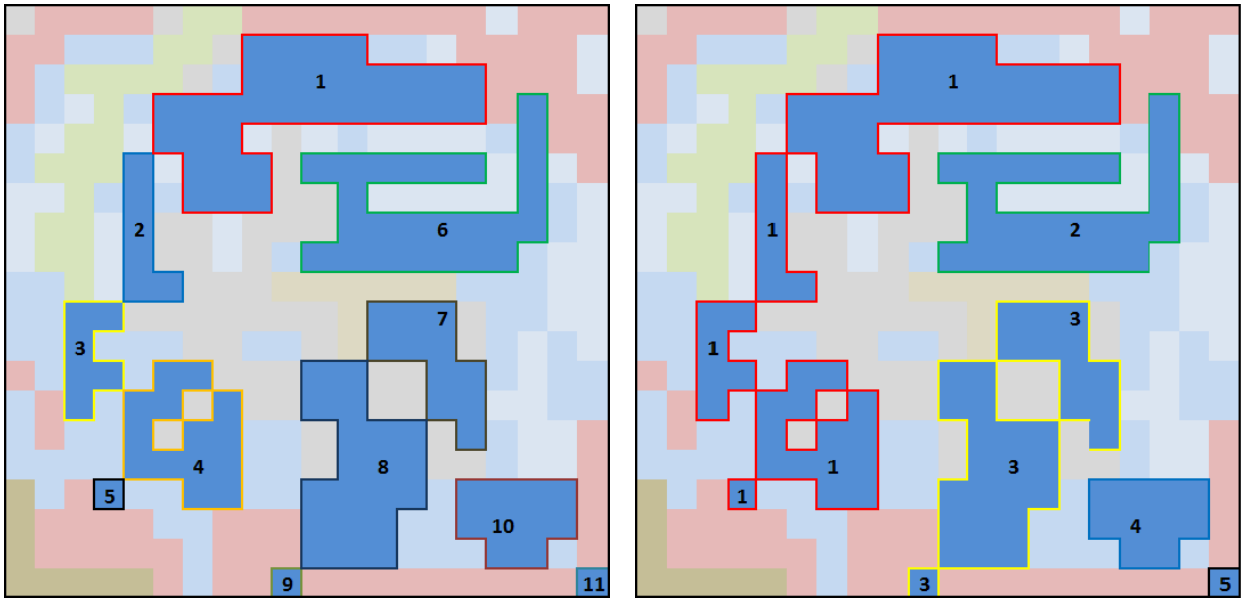
$$G(x,y) = \sqrt{G_x^2(x,y) + G_y^2(x,y)}, \tag{1.4}$$

where $G_x = conv\big(f(x,y), w_x\big)$ and $G_x = conv\big(f(x,y), w_y\big)$. Different implementations exist for gradient masks which determine the type of the edge detection algorithm. For instance, *Sobel* edge detection method uses the following

gradient masks of size 3x3.

$$w_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, w_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{1.5}$$

1.1.3 Blob Analysis



(a) 4-Connectivity (b) 8-Connectivity

Figure 1.3: Example of blob analysis connectivity types when finding blue coloured blobs.

Blob analysis involves the segmentation of images based on connected components (blobs), and analysis of various blob properties, e.g., area and centroid of the blobs. The shape of the connected components may vary based on the type of connectivity, which is commonly a 4 or 8-connected type. A 4-connected type refers to pixels that are neighbors to every pixel that touches one of their edges. These pixels are connected horizontally and vertically. 8-connected pixels are neighbors to every pixel that touches one of their edges or corners. These pixels are connected horizontally, vertically, and diagonally. Figure 1.3 visualizes the 4 and 8-connectivity concepts in digital images.

2 In-Lab Exercise

2.1 Image Thresholding

The first controller model that is used for this lab is "QBot2_Image_Thresholding.mdl" shown in Figure 2.3.

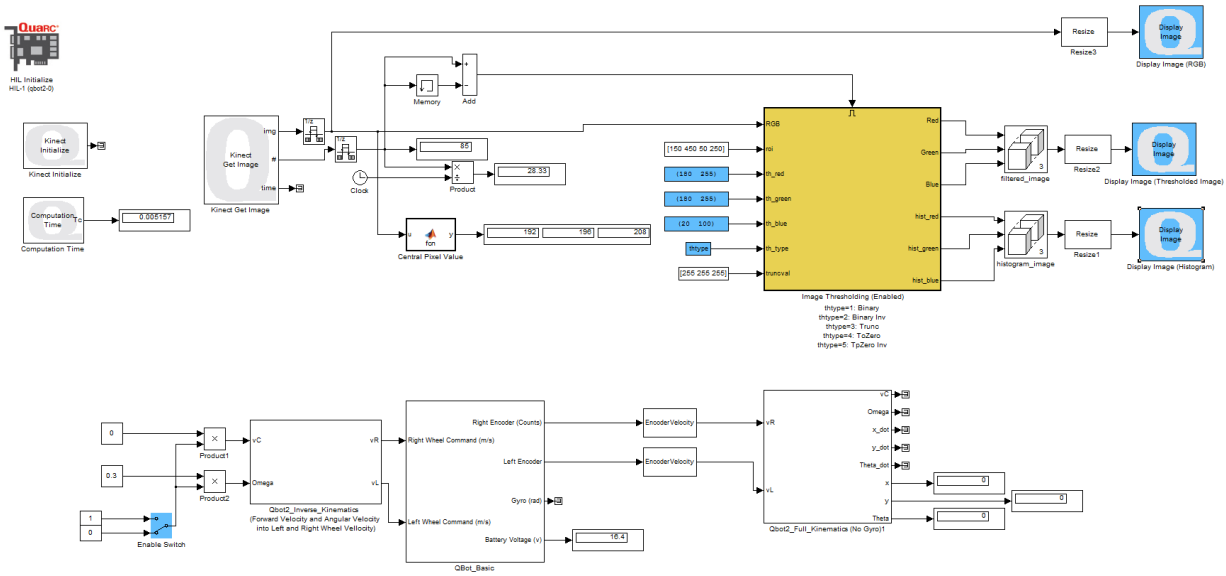


Figure 2.1: Snapshot of the controller model "QBot2_Image_Thresholding.mdl"

The *Image Thresholding* block, in yellow, implements the thresholding algorithm using the different methods described in the Background section. This block receives the frame, region of interest (roi), thresholds on all three color channels (th_red, th_green, th_blue), the method (th_type) and the truncation values (thuncval) required for the truncation approach. This block outputs the processed image through RGB channels as well as the histograms.

Follow the procedure outlined below. Make observations about the effect of the algorithms on the generated images, and answer the associated questions.

1. Open the supplied model, QBot2_Image_Thresholding, and make sure the manual switch (Enable switch) is set to zero. Compile the model and run it.
2. Double click on the three *Display Image blocks* in your model.
3. Find or create a coloured sheet of paper, and cut out a 10cm x 10cm square.
4. Bring the colored sheet close to the QBot 2, and use the RGB image to estimate the RGB values of the paper. To do so, change the mouse function to "Data Cursor" as shown in Figure 2.2. Select a point and move it to different locations on the RGB image, and note the RGB values for a few points on the card. Tilt the card in various orientations and observe the changes in the measured colours. Then define a range for the RGB values that represent the colour of the card.
5. Inspect the "Display Image (Histogram)" window, and observe the changes in the output when you bring objects with different colors, including the colored piece you just made, in front of the robot. Can the histogram help you find the [min max] ranges (the histogram is scaled by a factor of 10)? Save a snap-shot of the image in each case, and discuss the results.
6. Using the range that you found, set the [min max] values of the three threshold parameters, th_red, th_green and th_blue.
7. If the filtering does not seem good enough, try tuning the threshold values until the coloured piece of paper is completely filtered out.

8. Double-click on the "th_type" variable, and select the various options one-by-one, observing what each method does to the results. Save a snap-shot in each case, and discuss the results.

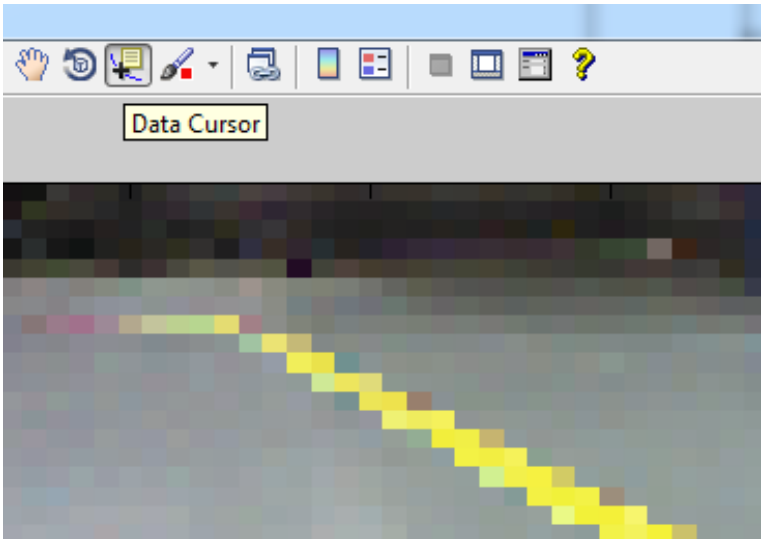


Figure 2.2: Data cursor mouse function to get RGB data from the image.

2.2 Edge Detection

The next controller model for this lab is "QBot2_Edge_Detection.mdl" shown in Figure Figure 2.3.

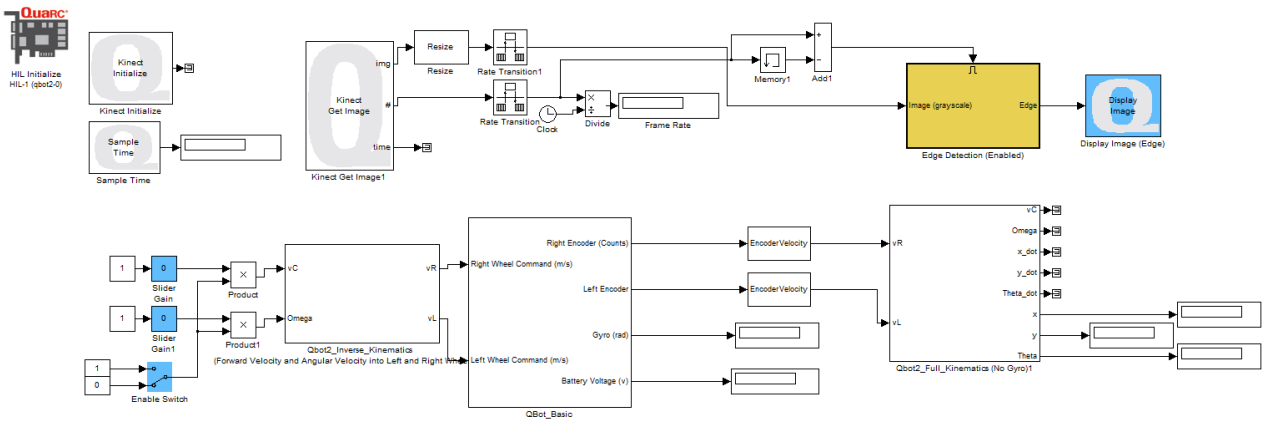


Figure 2.3: Snapshot of the controller model "QBot2_Edge_Detection.mdl"

The "Edge_Detection" subsystem, shown in yellow, processes the image using the mask convolution technique described in the Background section. Double-click on this block, open the embedded mathscript and find the masks and convolution functions used in the code.

Follow the procedure outlined below. Make observations about the effect of the algorithms on the generated images, and answer the associated questions.

1. Make sure the manual switch (Enable switch) as well as all the sliding gains (shown in blue) are set to zero.
2. Double click on the "Display Image (Edge)" block (also in blue) to open up the figure window.
3. Compile the model and run it once it is downloaded to the target.

4. Look at the resulting image showing the edges. Toggle the enable switch, and slowly change the slider gain related to "Omega". Observe the effect on the generated image, and record your observations.
5. Toggle the manual switch to disable the algorithm, and stop the model.
6. Double-click on the "Edge Detection" block to open the embedded mathscript. Comment out the MATLAB convolution functions and implement your own convolution function based on the masks and convolution functions in the Background section (Equation 1.3). Compile the updated code and go through the steps above to access the comparative performance of your solution.
7. What would need to be changed if you were asked to implement a different Edge Detection technique?
8. Describe how you would extract the edges out of the image.

2.3 Blob Detection

The model for this part of the lab is called "QBot2_Blob_Tracking.mdl" shown in Figure Figure 2.4.

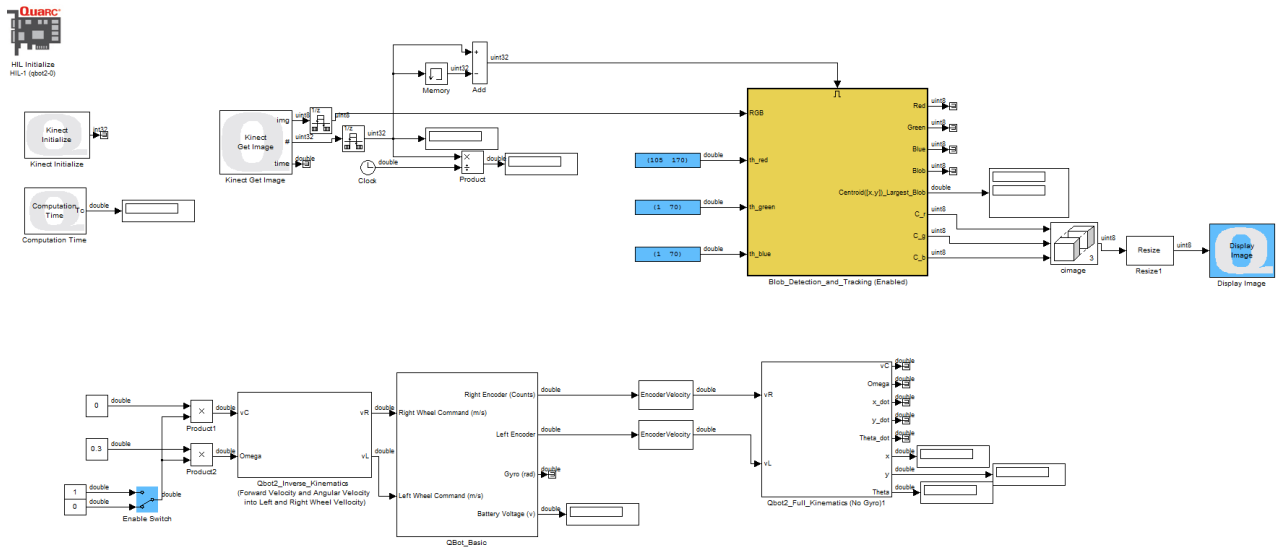


Figure 2.4: Snapshot of the controller model "QBot2_Blob_Tracking.mdl"

The yellow subsystem called "Blob_Detection_and_Tracking" includes a) a thresholding algorithm on all three color channels, b) a blob filter (8-connectivity default value) and c) a third block that finds the centroid of the largest blob.

Follow the procedure outlined below. Make observations about the effect of the algorithms and answer the associated questions.

1. Set the [min max] values of the three threshold parameters (highlighted with blue) that you found in the first part of the lab.
2. Make sure that the manual switch (Enable Switch) is set to zero.
3. Double-click on the "Display Image" block.
4. Compile and run the model and look at the Display Image window.
5. Bring your colored piece in front of the robot and see if it is completely detected by the robot. Move if back and force, up and down and look at the "Blob Centroid" display. If the object is not fully detected as a blob, tune the [min max] threshold values until your object is fully detected as one blob.
6. Explain how the blob centroids could be used to plan the motion of the robot.

REASONING AND MOTION PLANNING

The reasoning and motion planning stage of a vision-based robotic application uses different image processing algorithms in order to generate appropriate motion commands for the robot. The objective of this exercise is to explore how these methods can be implemented on the Quanser QBot 2 Mobile Platform.

Topics Covered

- Reasoning based on the image features
- Motion Planning

1 Background

The goal of this lab is to use a set of image processing algorithms to interpret the environment around the robot, and create appropriate motion commands for the robot. As an example, you will learn how the QBot 2 can follow a line on the floor.

1.1 Reasoning Based on Image Features

Image features, such as blob centroids, edges, corners, etc. are utilized primarily in robotics to make reasoned statements as to the environment around the robot, and an appropriate course of action. In order to create an algorithm that can make appropriate decisions, you will often need to remove much of the detail from the data that is provided by the image capture device in order to create clear judgments about the state of the robot and the environment. This act of using available data to create conditional statements is the basis for much of artificial intelligence.

Reasoning is the highest level of vision-based motion planning, extending several lower-level image processing techniques. In this laboratory, we will use blob centroids as "facts" about the environment that indicate where the next goal for the robot is, and generate motion commands based on "rules" for appropriate robot motion.

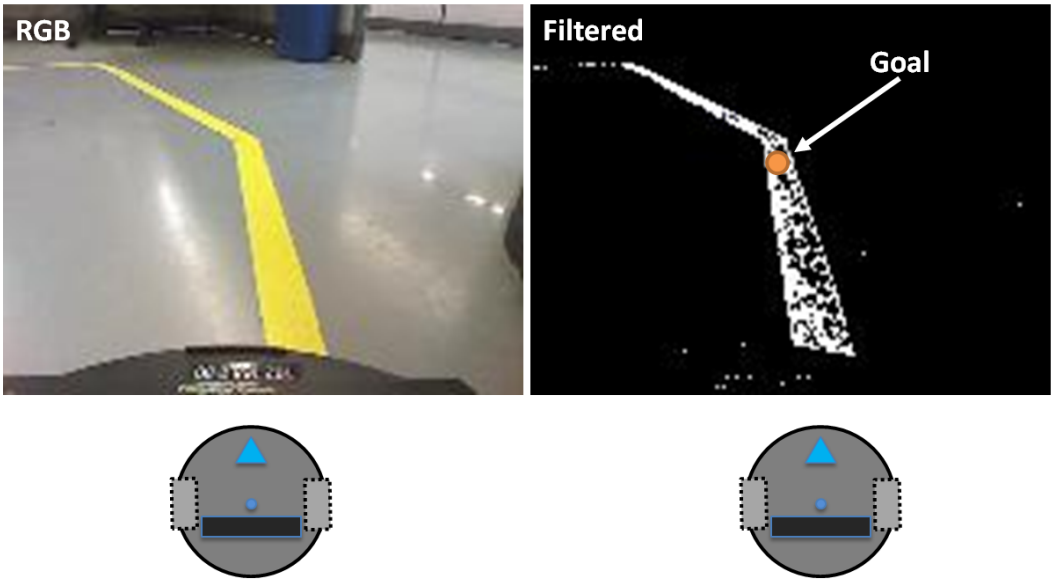


Figure 1.1: Goal and current location of the robot base in a line following scenario.

1.2 Motion Planning

Motion planning generally involves the creation of motion commands for the robot based on a series of goal positions, or way-points. For this laboratory experiment, the motion planning algorithm uses the centre of the current blob as the next goal for the position of the robot (command or set-point), as well as the current location of the robot to create the motion path. Given that the Kinect sensor is mounted on top of the robot, and can see directly in front of the robot (when the Kinect is tilted down), we can always assume that the current location of the robot is a fixed distance below the last row of the image in the image coordinate frame. Figure Figure 1.1 shows the goal (blob centroid) as well as the current location of the robot (below the last row of the image) in the image coordinate frame.

2 In-Lab Exercise

The model for this part of the lab is "QBot2_Line_Following.mdl" shown in Figure 2.1. This model uses image processing algorithms, explicitly blob filtering, in order to find a line, locate the goal, and controls the robot to reach the target.

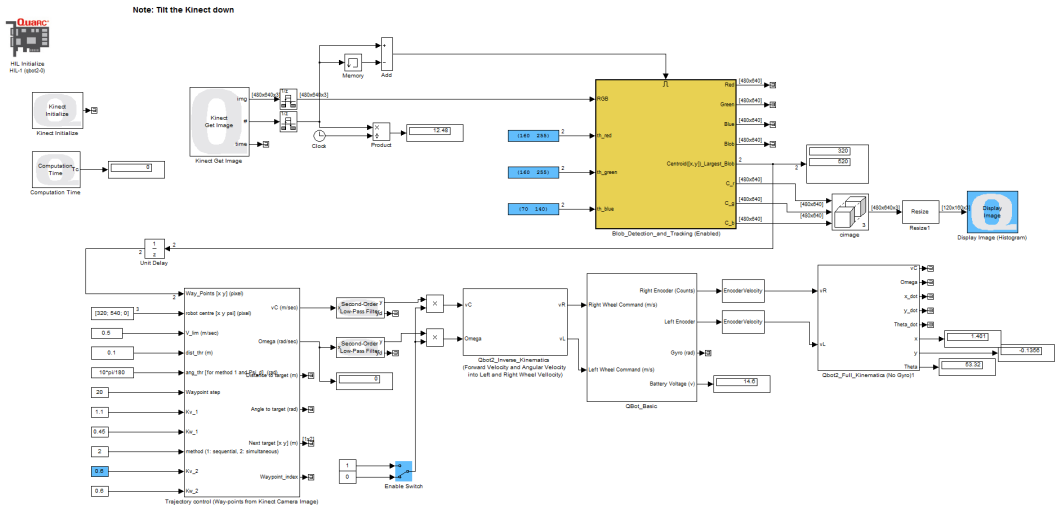


Figure 2.1: Snapshot of the controller model "QBot2_Line_Following.mdl"

Note: In this lab you need to tilt down the Kinect sensor so that the robot can see right in front of its chassis.

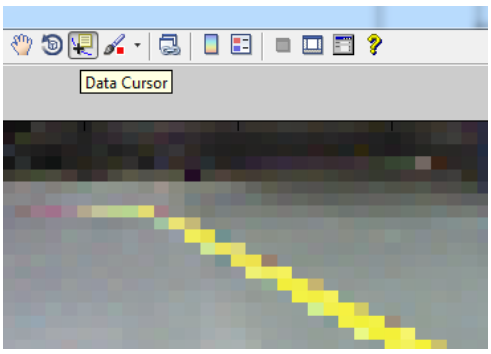


Figure 2.2: Data cursor mouse function to get RGB data from the image.

1. Open the supplied model, QBot2_Line_Following.mdl, and make sure the manual switch (Enable switch) is set to zero. Compile the model and run it.
2. Double click on the two "Display Image" blocks in your model.
3. Pick a color tape of your choice, and tape in the floor to make a line for the robot to follow as shown in .
4. Put the robot on the floor right in front of your start point of the line. Tune the [min max] values of the three threshold values, th_red, th_green and th_blue, highlighted with blue, while looking at the "Display Image" window so that the line is clearly filtered in the resulting image. To achieve good results change the mouse function to "Data Cursor" as shown in Figure 2.2 and click on different points on the RGB image to estimate the RGB range of values for this piece. Select a point and move it to different locations on the RGB image and note the RGB values for a few points on the card in different angles. Then find a range for the RGB values that represent this card. Once the tuning is successful, enable the manual switch and see if the robot can follow the line. You can change the Kv_2 gain which is the control gain of the robot. The larger this value, the faster the robot will move. Note that by increasing Kv_2, you might make the controller unstable.

VISUAL ODOMETRY

Visual odometry is an approach that is used to estimate the motion of a robot using input from the vision sensors alone. To estimate the motion of the robot from vision data, one needs to capture images from a camera mounted on the robot, and estimate how different points of interest move from one image to another. This approach is called Structure from Motion, or SFM.

Therefore, the first step to visual odometry involves finding corresponding points in two images taken from one scene, called the Correspondence problem. In order to get absolute scale measurements for the motion, stereo cameras are usually required. Therefore, since in this lab we will only have access to a single camera we won't be able to determine the absolute scale.

Topics Covered

- Correspondence Problem
- Structure From Motion and Visual Odometry

1 Background

1.1 Correspondence Problem

The first step in visual odometry is to find corresponding points in two images taken from a common scene, called the "Correspondence Problem". Given two images of a single scene taken from different perspectives, we need to identify the same object points in both images and recover their 3D position in the scene, (x, y, z). The natural assumption in this case is that the two images do not differ much and that we will be able to find similar features in both.

There are two approaches to finding corresponding points in two images:

- Area-based methods: consider a small window in one image and look for the most similar window in the other. Several different algorithms are used to measure the similarity including "Sum of Absolute Differences (SAD)" and "Sum of Squared Differences".
- Feature-based methods: extract features from the images such as Edges, Corners, Blobs, etc. and try to find the corresponding features. The same similarity measures can apply to this approach.

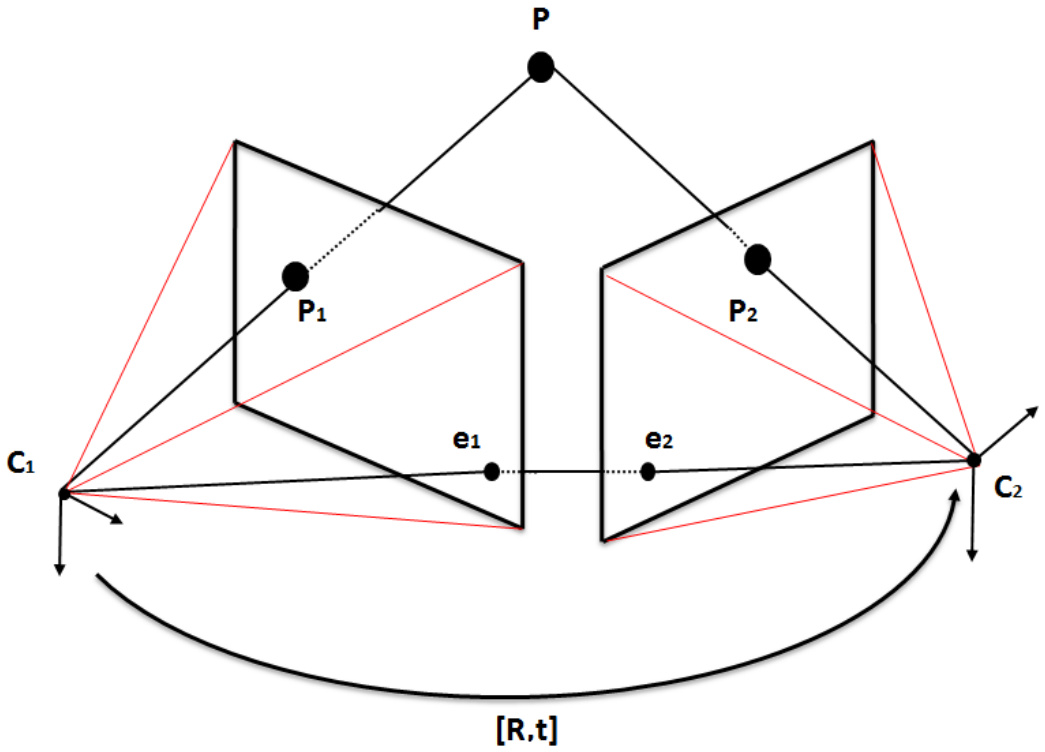


Figure 1.1: Epipolar geometry for two camera configurations.

In this lab, we use the more stable feature-based technique and use corners as an example feature.

1.2 Structure From Motion (SFM) and Visual Odometry

After finding the corresponding points, we need to estimate the motion of the camera. Figure 1.1 shows two images taken from one camera in two different poses (C_1 and C_2) where the camera is rotated with the rotation matrix R and translated with translation vector t from pose C_1 to the pose C_2. Assume that we have identified the corresponding points P_1 and P_2, both representing the point P in the space as shown in Figure 1.1.

If the camera is calibrated, with calibration matrix A, the relationship between the point P and its projections, P1 and P2 on C1 and C2 respectively is

$$\lambda_1 P_1 = AP \text{ for the } C_1 \text{ case, and } \lambda_2 P_2 = A[R|t]P \text{ for the } C_2 \text{ case.} \tag{1.1}$$

where $[R|t]$ shows a Rotation by matrix R and translation by vector t. Equation 1.1 shows the relationship between the projected points $P_1 = [u_1, v_1, 1]^T$ and $P_2 = [u_2, v_2, 1]$ (2D) and the actual point $P = [x, y, z]^T$ in space (3D). Therefore, replacing $P = \lambda_1 A^{-1} \times P_1$ from the C_1 case into the C_2 case, we have

$$\lambda_2 P_2 = A[R|t]P = ARA^{-1}P_1 + At \tag{1.2}$$

which represent the $e_1 e_2$ line, shown in Figure 1.1, called *epipolar* line. After some manipulation, we derive the following equation

$$P_2^T \, E \, P_1 = 0, E = [t] \times R \tag{1.3}$$

where $[t]\times$ is defined to be

$$t = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}.$$

Matrix E is called the "essential matrix" and plays an important role in finding structure from motion, and equation Equation 1.3 is defined as the "epipolar constraint". In this lab we compute the essential matrix using a *Singular Value Decomposition (SVD)*-based method called the *eight-point* algorithm.

After finding the essential matrix, all we need is to decompose the matrix into the rotation R and translation t. The above routine should be repeated every time we have small movements in the robot (and camera). Finally, the motion of the robot can be estimated by numerically integrating over these small movements.

2 In-Lab Exercise

The supplied controller model is called "QBot2_Image_Proc_SFM.mdl" as shown in Figure 2.1. We use this model to find the relative rotation of the robot using visual data from the Kinect sensor.

Note: Since finding absolute movements requires stereo camera or using other means, we will only find the relative rotation, R, in this lab.

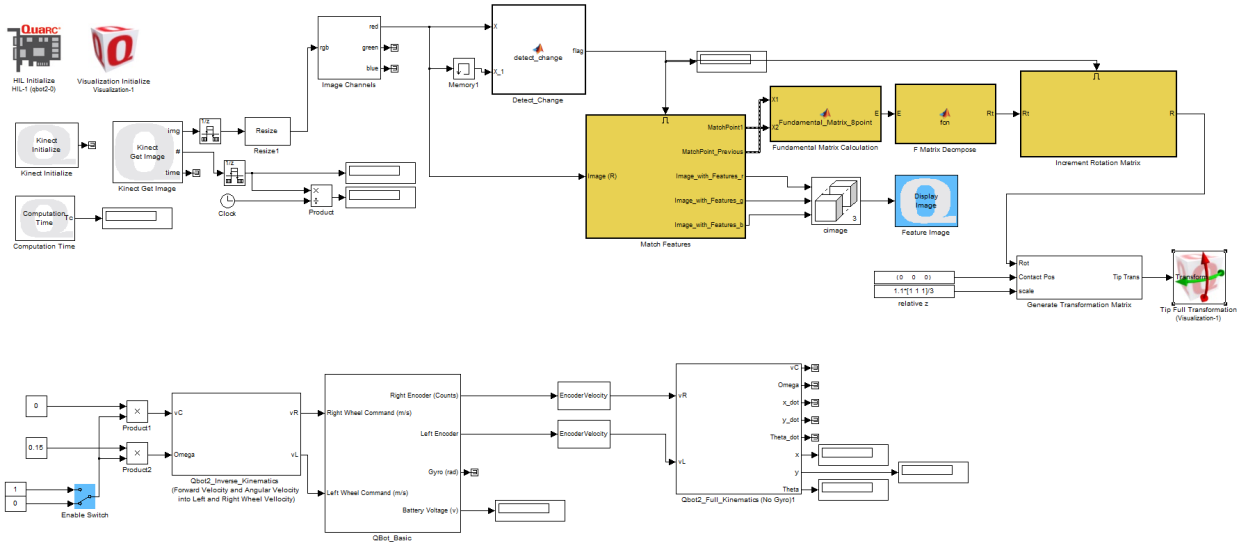


Figure 2.1: Snapshot of the controller model "QBot2_Image_Proc_SFM.mdl"

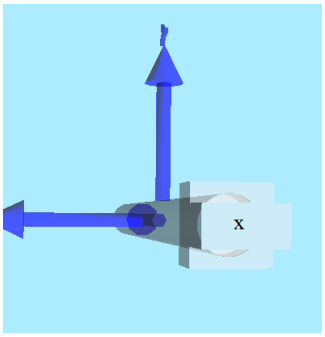


Figure 2.2: Visualization of the device axes. x axis is highlighted with a white rod in the visualization.

The steps needed for Visual Odometry, as outlined in the Background section, are implemented in this model. After getting the RGB image from the Kinect sensor, one channel of the image (Red) is used to find the feature points of interest, corners, and match them with those from the previously captured video frame. The block diagram "Match Features" finds the corresponding feature points in both images and sends them out as X1 and X2. These two vector of points are used in the "Essential Matrix Calculation" to calculate the essential matrix "E". Then the rotation matrix at each time, Rt, is calculated from the essential matrix using the SVD-based method called the eight-point algorithm. Finally Rt is integrated numerically over time such that the matrix R is calculated.

1. Open the supplied model, QBot2_Image_Proc_SFM.mdl, in MATLAB. Make sure the manual switch (Enable Switch) is set to zero.
2. Compile the model and run it. A visualization window will open, similar to the one in Figure 2.2. The x axis arrow in this visualization shows the robot forward direction (you are looking behind the robot and the x axis is facing forward). Double-click on the "Feature Image" window. This image shows the features in the current

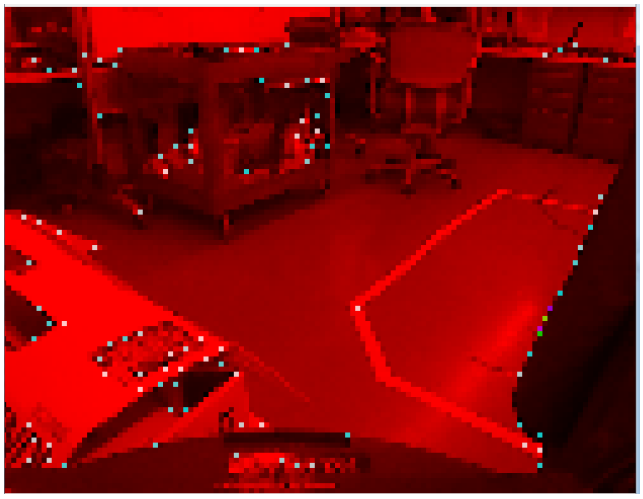


Figure 2.3: Detected features shown as dots on the screen.

frame as well as the previously captured frame as shown in Figure 2.3. Initially these point perfectly match since there is no motion involved.

3. Manually rotate the QBot 2 for about 90 degrees. Your visualization shows the rotation of the robot according to the calculated rotation matrix, R, based on the vision data.
4. Continue rotating the robot in different directions. You will see that the visualized x axis drifts over time. Explain the possible causes. What happens if the environment has no features at all; e.g. a white background?
5. Open up the "Match_Features" block and double click on the "Match_Points" MATLAB function. Describe the algorithm that the "matchFeatures" function uses to find matching points between two vectors.

Quanser Inc.
119 Spy Court
Markham, Ontario
L3R 5H6
Canada
info@quanser.com
Phone: 1-905-940-3575
Fax: 1-905-940-3576

Printed in Markham, Ontario.

For more information on the solutions Quanser Inc. offers, please visit the web site at:
http://www.quanser.com

EXPERIMENT 4: PATH PLANNING

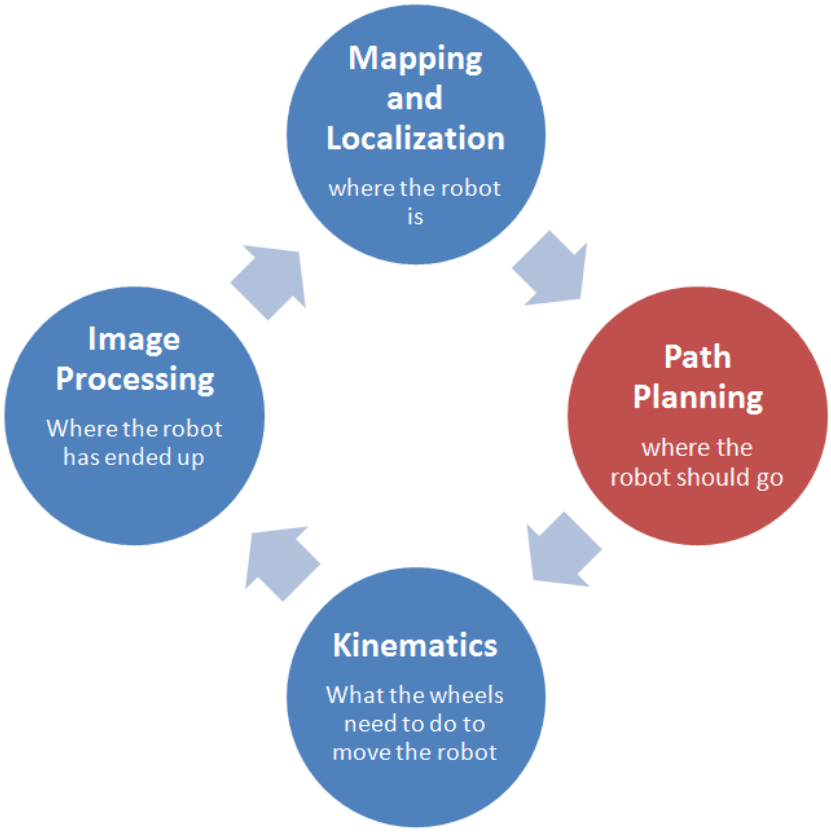
The purpose of this experiment is to create algorithms for the QBot 2 to navigate safely and avoid obstacles. The following topics will be studied in this experiment.

Topics Covered

- Different motion planning approaches including deliberate, reactive and hybrid methods.
- Different path planning algorithms such as A* and potential fields.

Prerequisites

- The QBot 2 has been setup and tested. See the QBot 2 Quick Start Guide for details.
- You have access to the QBot 2 User Manual.
- You are familiar with the basics of Matlab® and Simulink®.



Path planning is used to determine where the robot should go

PATH PLANNING

Motion planing plays a key role in autonomous mobile robot navigation. It chiefly involves interpreting the goals specified for the robot, along with the current mapping and navigational data in order to create a path to a target location. In this lab, you will learn how to design and implement path planning algorithms in order to navigate and avoid obstacles in a mapped environment. You will learn two important algorithms for path planning.

Topics Covered

- Potential fields
- A*

You will also learn how to implement motion control for QBot 2 using closed-loop control.

1 Background

The Quanser QBot 2 Mobile Platform comes with a Microsoft Kinect sensor that has the ability to generate a depth map of the environment. This information, along with the location and orientation of the robot chassis, can be used for autonomous map building. The 2D map generated using this data will be processed to locate various obstacles in the map and create the "occupancy grid map" of the environment. In this lab we only focus on the path planning portion of this process.

Generally, global path planning is performed in robot's *configuration space*, where the robot is considered as a point object and the obstacle dimensions are increased by the maximum centroid-to-periphery dimension of the robot (in the case of the QBot 2, by the 18 cm radius of the body). Path planning in the configuration space provides a safety margin between the way points and the obstacle grids. An example of an occupancy grid map is shown in Figure 1.1 where the workspace is represented in pixels.

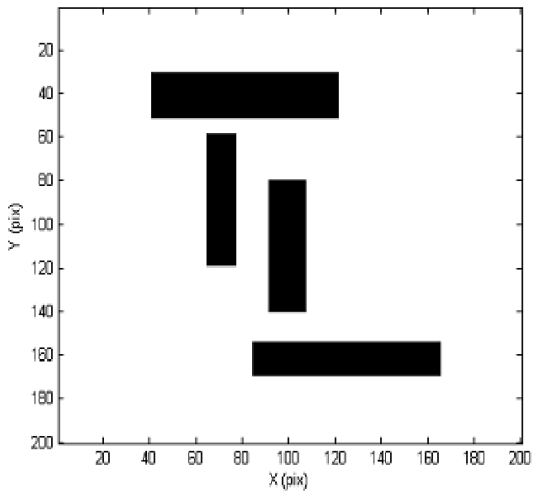


Figure 1.1: An example of an occupancy grid map where "pix" represents the pixels in the map.

Global path planning methods search for the connected grid components between the robot and target. There exist several techniques for global path planning; this laboratory will only describe the following two methods: Potential Field, and A* Diagram.

1.1 Potential Field

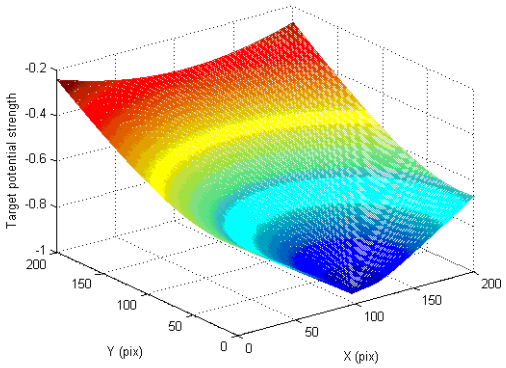
In this approach, the target exhibits an attractive potential field (U_{target}) and the obstacles exhibit repulsive potential fields (U_{obs_i}). The resultant potential field is produced by summing the attractive and repulsive potential fields as follows:

$$U = U_{target} + \sum U_{obs_i} \tag{1.1}$$

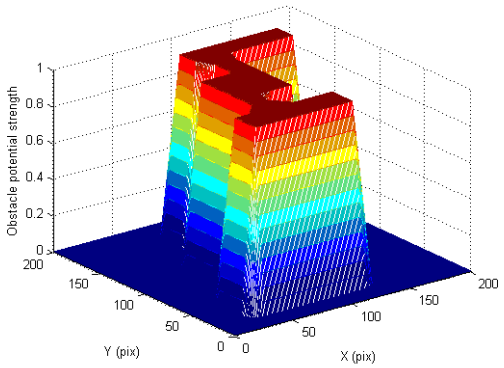
The net potential field is used to produce an artificial force equivalent to the gradient of the potential field with a negative sign as follows

$$F = -\nabla U \tag{1.2}$$

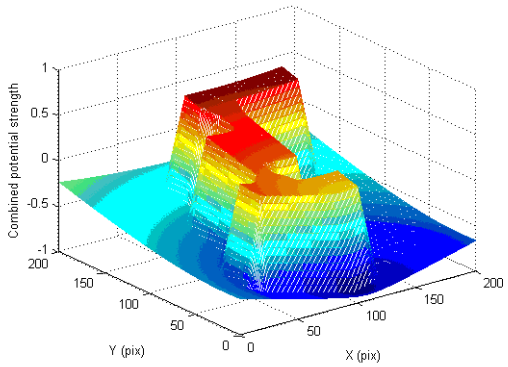
The motion is then executed by following the direction of the created force. To understand this, the potential fields are shown in Figure 1.2.



(a) Target potential field



(b) Obstacle potential field



(c) Combined potential field

Figure 1.2: Visualized potential fields for path planning

The target and obstacle potential field can be defined based on the distances to the target and the obstacles as follows

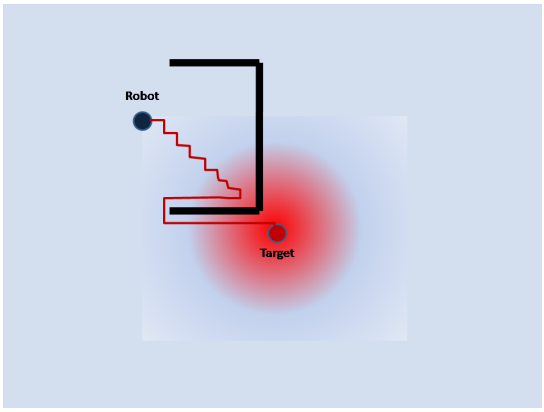
$$U_{target} = \frac{dist_{tar}(x,y)}{dist_{tar,th}} - 1 \tag{1.3}$$

$$U_{obs_i} = \begin{cases} 0 & \text{if } dist_{obs_i} > dis_{obs_i,th} \\ 1 - \frac{dist_{obs_i}(x,y)}{dist_{obs_i,th}} - 1 & \text{otherwise} \end{cases} \tag{1.4}$$

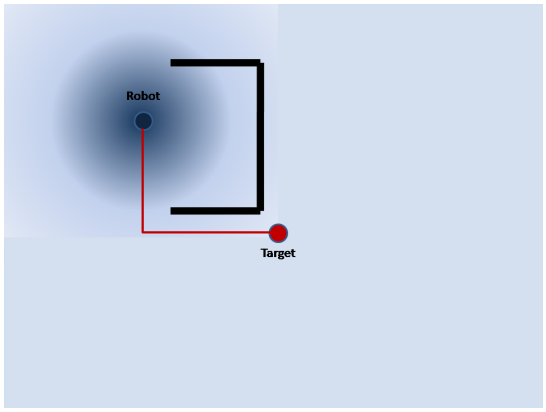
where $dist_{tar}(x,y)$ is the distance to the target (x,y) location of the workspace, $dist_{tar,th}$ is a normalization factor set to the diagonal distance of the workspace, $dist_{obs_i}$ is the distance to the i^{th} obstacle point from (x,y) location of the workspace and $dis_{obs_i,th}$ is the radial boundary of the obstacle potential field. A common problem of the potential field method is that the robot is trapped in a back-and-forth oscillatory motion when attractive and repulsive forces work in a straight line. To overcome this problem, a wall-following approach can be deployed where the robot follows the normal vector to the repulsive force. Tuning the threshold values will also help get better performance out of the algorithm.

1.2 A* algorithm

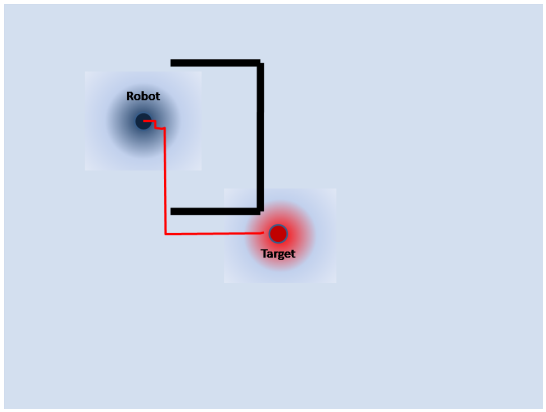
A* is the one of the most popular choices for path planning, because it's fairly simple and flexible. This algorithm considers the map as a two-dimensional grid with each tile being a node or a vertex. A* is based on graph search methods and works by visiting vertices (nodes) in a graph starting with the current position of the robot all the way to the goal. The key to the algorithm is identifying the appropriate successor node at each step.



(a) A* using path cost, g(n), exclusively



(b) A* using heuristic cost, h(n), exclusively



(c) A* complete

Figure 1.3: A* algorithm components and complete algorithm.

Given the information regarding the goal node, the current node, and the obstacle nodes, we can make an educated guess to find the best next node and add it to the list. A* uses a heuristic algorithm to guide the search while ensuring that it will compute a path with minimum cost. A* calculates the distance (also called the cost) between the current location in the workspace, or the current node, and the target location. It then evaluates all the adjacent nodes that are open (i.e., not an obstacle nor already visited) for the expected distance or the heuristic estimated cost from them to the target, also called the heuristic cost, h(n). It also determines the cost to move from the current node to the next node, called the path cost, g(n). Therefore, the total cost to get to the target node, f(n) = h(n) + g(n), is calculated for each successor node and the node with the smallest cost is chosen as the next point.

Using either the path cost g(n) or heuristic cost h(n) exclusively will result on non-optimized paths as shown in Figure 1.3. Figure 1.3 a and Figure 1.3 b show the result of using only g(n) and h(n) respectively. Together, an optimized path from the current node to the target node can be achieved as shown in Figure 1.3 c.

2 In-Lab Exercise

This exercise consists of the following four steps

1. Creating an occupancy grid map around an obstacle using "Qbot2_2D_Mapping_Keyboard.mdl".
2. Processing the created map to detect the obstacle using "Process_Map.m" MATLAB code.
3. Running different path planning algorithms using "Path_Planning_Map.m" MATLAB code.
4. Performing robot motion using "Qbot2_Path_Planning'_Motion_Control.mdl" model.

2.1 Setting up the environment and creating an occupancy grid map

You will have to first perform an occupancy grid map generation so that the QBot 2 is aware of its surrounding obstacles. The controller model for the mapping, shown in Figure 2.1, is called "QBot 2_2D_Mapping_Keyboard.mdl".

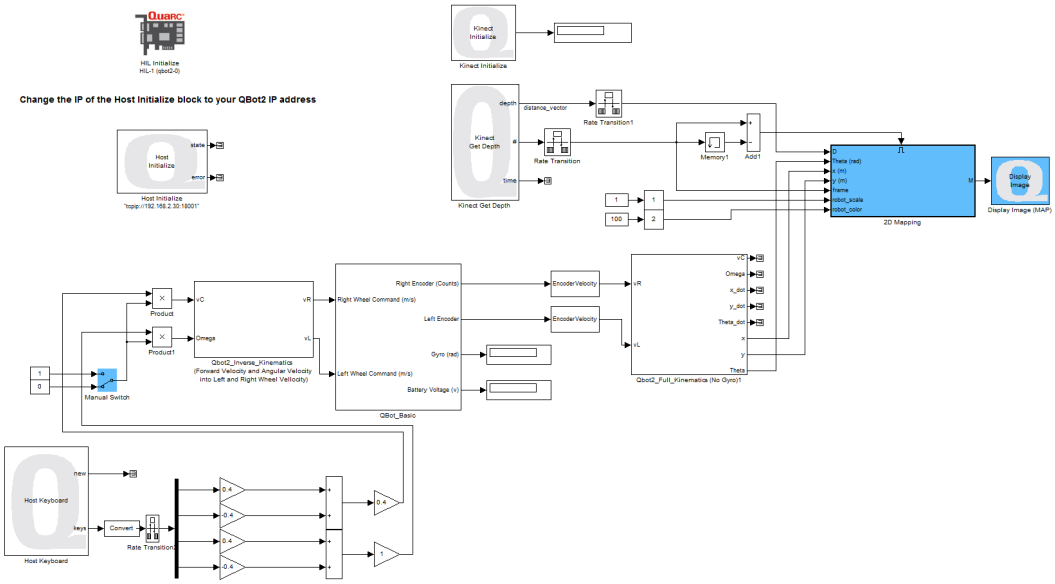


Figure 2.1: Snapshot of the controller model "QBot 2_2D_Mapping_Keyboard.mdl"

Before performing the experiment, make sure the Kinect sensor is flat so that it is parallel with the ground plane and not tilted. The resulting vector is used in the QUARC models to match the depth data to real-world coordinates. Then open the "QBot 2_2D_Mapping_Keyboard.mdl" model; make sure you change the IP address to your robot's IP address in QUARC-Preferences menu option. Also **change the IP address of the Host Controller block to your robot's IP address**, compile the model and go through the following exercises.

1. Find a zero pose for your QBot 2 where you have at least a 2.5m$\times$2.5m free space around it. Mark this initial position and orientation of the the QBot 2 as reference for the next steps. Find a box of at least 0.5m$\times$0.5m$\times$0.5m and place it about 1m in front of the robot as in Figure 2.2. Put the robot in a known initial configuration (Pose 1), shown in Figure 2.2), run the model and enable the manual switch once the robot is ready. Use the keyboards (W and S for linear motion; A and D for rotation) to move the robot around the obstacle, to poses 2-4, and then move it back to the initial start point. At each pose try to rotate the robot 360 degrees around itself so it can find all the free space around the obstacles.

   Note: Make sure the robot always stays at least 1 meter away from the obstacle as in Figure 2.2. Do not drive the robot closer to the obstacle.

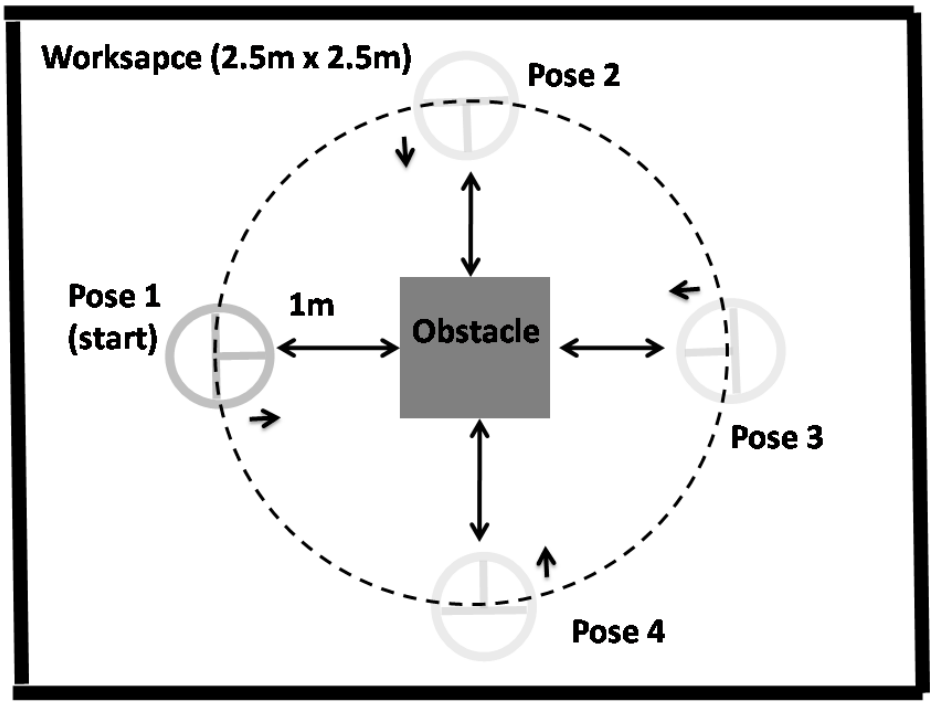


Figure 2.2: Configuration of the robot and obstacle for occupancy grid mapping.

2. When the robot is back to the initial pose, you should clearly see the box in the created map similar to Figure 2.3.

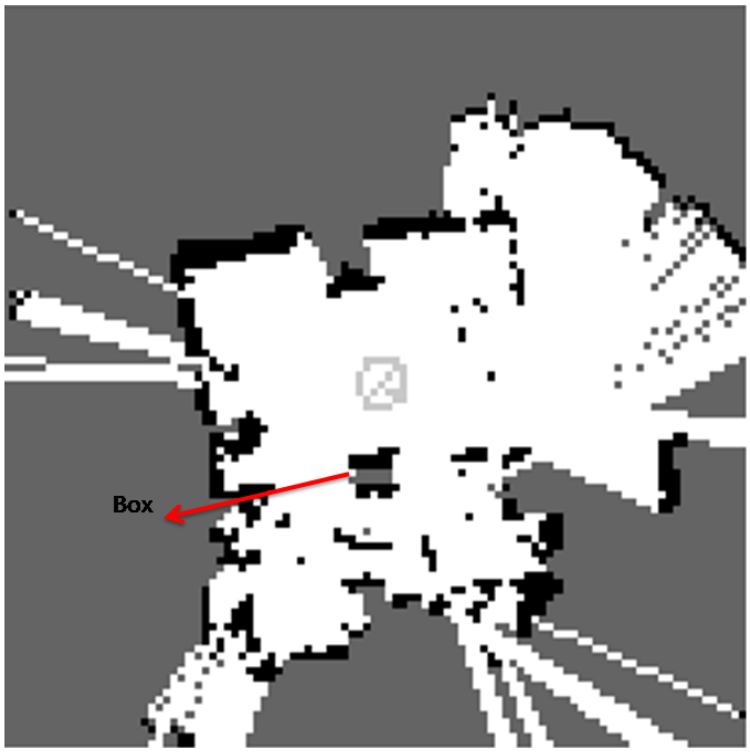


Figure 2.3: Configuration of the robot and obstacle for occupancy grid mapping required for path planning.

3. Right click on the map figure and select save to workspace. Save it as "Map_obs". Go to the MATLAB workspace, find this item, right click on it and save it as "Map_obs.mat" in the same folder as the code is for future reference.

2.2 Path planning and motion execution

The next step is to process the map and find the coordinate of the obstacle in the map. This step ignores errors and imperfections and at the same time adds margins to the found obstacle. For the purpose of this lab, this pre-process MATLAB code (called Process_Map.m) is designed to use blob analysis and ignores larger blobs which reprisent for background areas.

1. Run the MATLAB function Process_Map.m. It creates a simplified occupancy grid map (look for x and y axes directions - x axis shows the direction of the initial pose of the robot) as in Figure 2.4 (only black rectangles at this point) and make sure it can find the obstacle and display it. It then asks you to click on the robot initial position as well as the target position. You should select (0,0) for the initial position since you start off from the centre of the map.

   Note: The resulting simplified map may appear to be rotated 90 degrees counter clockwise compared to the initial map you saved before; this is is because of the definition of initial axes direction.

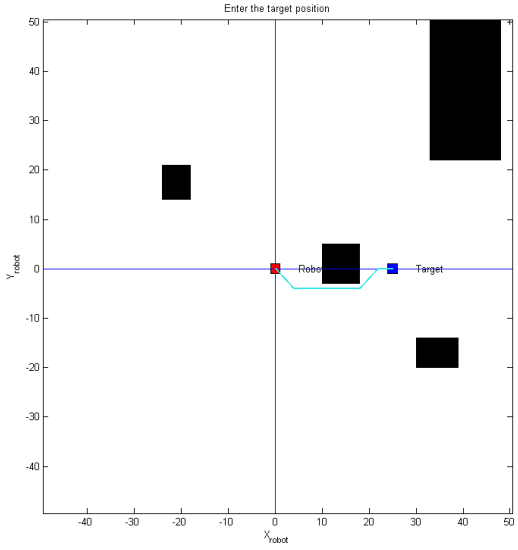


Figure 2.4: Path, start and target points for the robot shown in the simplified occupancy grid map using the A* algorithm.

2. Without closing this figure, open MATLAB function called "Path_Planning_MAP.m" and look for "Method options:" in the code (line 10). Uncomment the "method = POTENTIAL_FIELD;" line and make sure the rest of the methods are commented out.

3. Run this MATLAB code and observe the path from start to the end (as in Figure 2.4).

2.3 Motion Execution

After the path is found, manually move the robot back to the starting point and make sure you maintain the initial orientation as marked in the first step. The x axis is the direction of the QBot 2 at this pose and the y axis can be found using the right hand rule (on the left side of the robot when standing behind it and looking forward) . By looking at the path plotted in the previous step (note the x and y axis values), try to estimate the motion of the robot (note

the scale of 10: 1m shows as 10 in the figure). If your estimated motion is not colliding with the obstacle, then go through the following path to execute the motion.

1. If the path in the resulting figure completely avoids the obstacles in the simplified map, now you can execute the motion. For this, Open the model called "Qbot2_Path_Planning_Motion_Control.mdl" shown in Figure 2.5. This model utilizes the path created in the previous step and controls the robot to move along this path. This

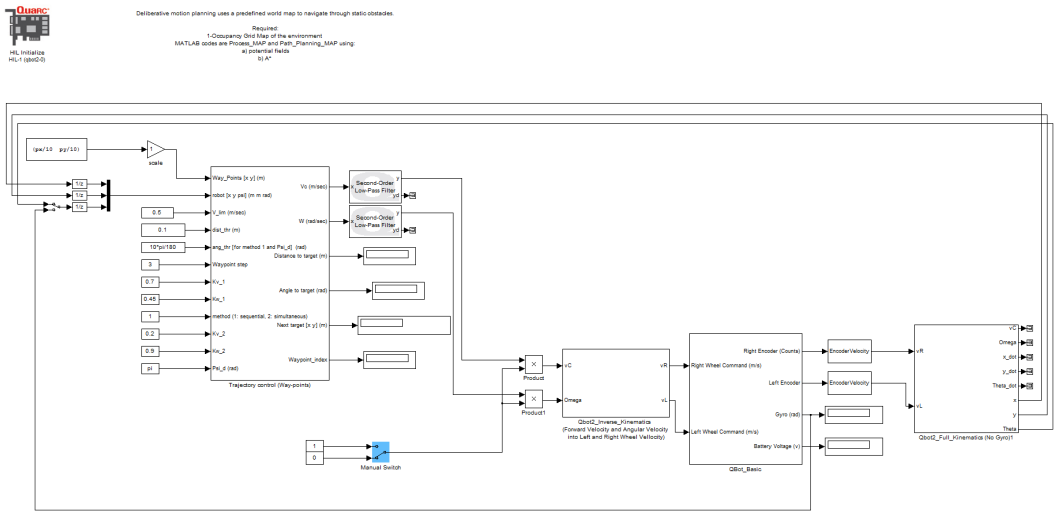


Figure 2.5: Snapshot of the controller model "QBot 2_Path_Planning_Motion_Control.mdl"

model gets the target way points generated by the path planning algorithm as vectors and sends them one by one to the robot. Then it will compare the way points with the current position of the robot and uses a proportional controller in a feedback loop to control the pose of the robot. Take a moment and explore the model, particularly the feedback and the trajectory control block. Then go through the following steps to execute this motion.

2. Make sure that the Manual Switch is disabled. Compile this model and run it.
3. Enable the manual switch and wait for the robot to move. Observe the behaviour of the robot and compare the motion with the path that is found. Once the robot arrives at the target, disable the manual
4. Go back to the first step on Subsection 2.2 and this time use the A* method to find the path. Run through all the steps above and observe robot motion.
5. Open up the "potential_fields.m" as well as the "A_Star.m" MATLAB function, explain the algorithms used in both, and compare it with the discussion in the Background section.

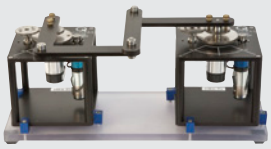
Quanser Inc.
119 Spy Court
Markham, Ontario
L3R 5H6
Canada
info@quanser.com
Phone: 1-905-940-3575
Fax: 1-905-940-3576

Printed in Markham, Ontario.

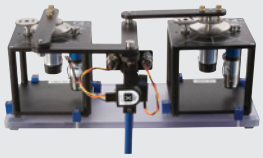
For more information on the solutions Quanser Inc. offers, please visit the web site at:
http://www.quanser.com

Solutions for teaching and research in robotics and autonomous systems

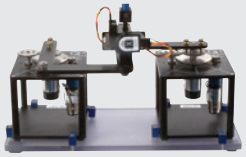
2 DOF Robot



2 DOF Gantry

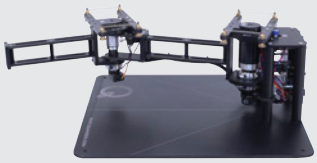


2 DOF Inverted Pendulum

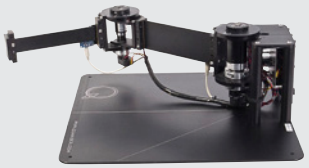


2 DOF Planar Robot

2 DOF Serial Flexible Joint

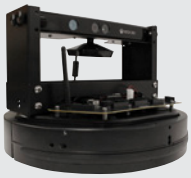


2 DOF Serial Flexible Link



6 DOF Denso Open Architecture Robot

QBot 2



QBall 2

Unmanned Vehicle Systems Lab

With Quanser robotic systems, you can introduce control concepts related to stationary and mobile robotics, from vibration analysis, resonance and planar position control to sensors, computer, vision-guided control to unmanned systems control. All of the experiments/platforms are compatible with MATLAB®/Simulink®.

QUANSER
INNOVATE·EDUCATE

INFO@QUANSER.COM +1-905-940-3575 QUANSER.COM

Solutions for teaching and research. Made in Canada.