Programming the Autonomous Navigation and Response of a QBot 2

Autonomous Systems | MANU2206

19/06/2024

--------- Authors ---------

Yusri Hayat	S3842117
Kyle Arnold	S3895440
Jay Dickson	S3719855

Table of Contents

Problem Statement

The aim of this project is to create a program that will enable the QBot 2 robot to travel and interact with an unfamiliar environment on its own while following predetermined rules and guidelines. In addition to mapping its surroundings and avoiding obstacles, the robot must be able to recognise and move towards a target. On the day of the demonstration this was a pink plushie octopus. It must also be able to obey traffic signals, which include slowing down at yellow signals and stopping at red ones. All while avoiding obstacles and mapping the environment. The robot has one minute to complete the demonstration.

Introduction

In the rapidly evolving field of robotics, the ability to autonomously navigate and interact with dynamic environments is paramount. This project focuses on developing a Simulink model for the QBot 2 robot, enabling it to autonomously map an unknown environment, avoid obstacles, obey traffic signals, and approach a specified target. The QBot 2 must perform these tasks under strict constraints, including responding to coloured signals, including slowing down at yellow, stopping at red, and approaching a pink plushie octopus without collision. All expected to be performed within a one-minute demonstration time frame. This report details the design, implementation, and performance of the robot's navigation system, highlighting the integration of depth sensing, bump detection, and colour recognition to achieve the project's objectives.

Background

In recent years, the automation of various operations has driven the development of autonomous systems and their associated programming. The Quanser Qbot 2 facilitates robot autonomy by utilising its advanced onboard features. This robot can be connected and controlled remotely. The Qbot2 is equipped with two bi-directional wheels, enabling it to turn and pivot on the spot with ease.

Additionally, the QBot 2 features functional bumper sensors on its front half, allowing it to detect direct collisions. It also comes with a Kinect that includes both an RGB camera and two depth cameras. The RGB camera provides video feedback, while the depth cameras enable the robot to gauge the distance to objects in its field of view. This combination of cameras allows the QBot 2 to react to different objects and accurately determine their proximity, enhancing its ability to navigate and interact with its environment.

Quanser integrates seamlessly with Simulink, a graphical block-based programming application used for the simulation and analysis of dynamic systems, extending the capabilities of MATLAB. Simulink is widely utilised in control systems and robotics due to its ability to run simulated frequency responses without altering the physical system, and to verify all aspects of an autonomous system, from perception to motion.

With the Quanser Interactive Labs for MATLAB, the QBot 2 can be operated directly from Simulink, leveraging all the tools provided by both Simulink and MATLAB. This toolbox ensures smooth integration and communication, enhancing the functionality and ease of use of the QBot 2 within the Simulink environment.

The primary blocks used when integrating and utilising the QBot 2's systems include the HIL Initialize block, which sets up everything related to Quanser lab blocks and products. Additionally, the HIL Read and Write blocks are employed to send and receive information to and from the QBot 2, such as wheel speeds, durations, and data from QBot 2 encoders like location or bumper signals.

Furthermore, the Kinect has its own set of blocks, including initialization to configure the Kinect, as well as Kinect Get Image and Kinect Get Depth blocks to acquire image and depth information from the cameras. These capabilities enable the detection of objects of interest in the field of view, prompting the QBot 2 to stop, slow down, or track the object. The depth measurements provided can determine how far away objects are, instructing the robot to move or avoid obstacles accordingly.

Model Design

Overview

Our robot employs a system that combines depth sensing, bump detection, and colour recognition to navigate its environment and respond to various stimuli.

The primary obstacle avoidance mechanism leverages a Depth Sensor that the robot uses to identify objects in front of it. Supplementing the depth sensor are the bump sensors that detect physical contact. This secondary system is crucial for detecting objects that the depth sensor might miss, especially those approaching from the sides or within the minimum detection range.

In addition to obstacle avoidance, our robot features a colour detection system. The system responds to three specific colours, each triggering a different behaviour.

The complete integration of these systems then ensures that our robot can navigate the environment while responding to colour triggers and avoiding obstacles. These key systems are outlined in detail in the next sections.

Obstacle Avoidance

Depth Sensor

Our obstacle avoidance system relied primarily on the Kinect Depth Sensor. This sensor captures a 640 x 480 depth-mapped image, the depth measurements are valid from 0.5m to 6m's per the specification. We then take a horizontal centre slice of this image and filter out values below 0.5m as they are not valid under the sensor's specifications. In using the central slice, we ensure objects directly in front of the robot are captured.

Bump Sensor

Our secondary avoidance system employs the robot's bump sensors to detect physical contact in its forward half. These sensors can differentiate between front, left, and right collisions, which is essential when the depth sensor fails to detect objects due to its limited vertical field of view. Although extending the depth sensor's field of view was considered, our tests demonstrated it to be unnecessary. Additionally, collisions may occur if an object enters the robot's path from the side, bypassing the 0.5-metre detection range.

Response

These two systems integrate to offer a robust obstacle avoidance solution. The depth sensor is configured to detect objects within a 0.6-meter range. When either sensor is triggered, the robot will rotate clockwise in place. This uniform directional response is designed to prevent a feedback loop, where the robot alternates its turns due to consecutive triggers from the bumper and depth sensor, causing it to repeatedly react to the same obstacle. By spinning in the same direction, the robot avoids this oscillating behaviour and navigates obstacles more effectively.

Colour Detection

Sensors and Data Processing

To achieve our goal of having the robot respond to various colour triggers and perform specific tasks—such as stopping, slowing down, and approaching—we required a versatile detection system that could be quickly adjusted to recognise any given colours, as the triggers were not predetermined before the demonstration.

We opted to use the HSV (Hue, Saturation, Value) colour space instead of the RGB colour space for our detection system. HSV proved to be much easier to calibrate and allowed us to define a range for each colour. This made colour detection more reliable across different lighting conditions and helped to ignore subtle variations in colour.

Our colour detection system relies on the Kinect Image Sensor, which outputs 640 x 480-pixel RGB images. To optimise processing, we first downsample the image by cropping out the top 100 pixels and the bottom 160 pixels, resulting in an image with a resolution of 640 x 220. We then further downsample the image by retaining only every fourth pixel in the vertical and horizontal directions. This results in a final image resolution of 160 x 55pixels. This cropping ensures that only the area directly in front of the robot is processed, excluding unnecessary information from outside the maze.

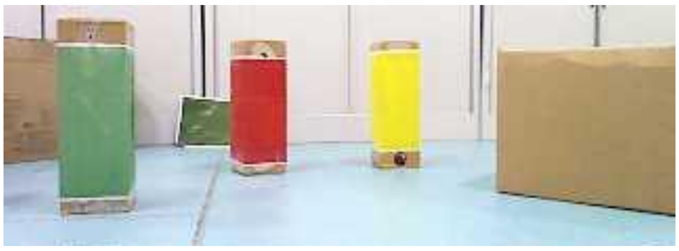

Figure 1: Example of the robot's viewport

Next, we segment this image using predefined HSV ranges to create binary masks for each colour. For red, two masks are generated to account for the high and low hue ranges found in red and red-orange, respectively.

These segmented images are then processed to calculate the total number of pixels that match the specified colour criteria. Additionally, we determine the horizontal centroid of the matched pixels, which can be used for object tracking.

This approach ensures that our robot can reliably detect and respond to colour triggers in real-time, adjusting to a variety of scenarios and lighting conditions.

Colour Response

The robot responds to three different colours, each triggering a specific behaviour. The first task is to determine the most prevalent colour within the captured image. The quantities of each colour are compared, and the most dominant colour is identified. If the quantity of the dominant colour exceeds a threshold of 100 pixels, the robot reacts accordingly. This threshold helps filter out small sets of colours that are likely false positives and adjusts the distance at which the robot responds to colours.

- **Stop Colour**: When the stop colour is detected and exceeds the pixel threshold, the robot’s speed is set to zero, halting its movement.
- **Slow Colour**: When the slow colour is detected and exceeds the threshold, all the robot’s speeds are reduced by half while still allowing it to continue object avoidance.
- **Approach Colour**: The approach colour has the robot move towards it. The robot will stop when the quantity of pixels exceeds a threshold, indicating it is close enough to the object. To keep the colour centred, the robot calculates the horizontal centroid of the approach colour and adjusts its direction accordingly.

Calibration

We developed a calibration script to test our segmentation ranges effectively. The process begins by running the model and passing the pre-processed image into a Video Compressed Display Block. We then position the coloured markers in front of the robot under various lighting conditions to capture a diverse set of images. These images are saved for further processing by our calibration script.

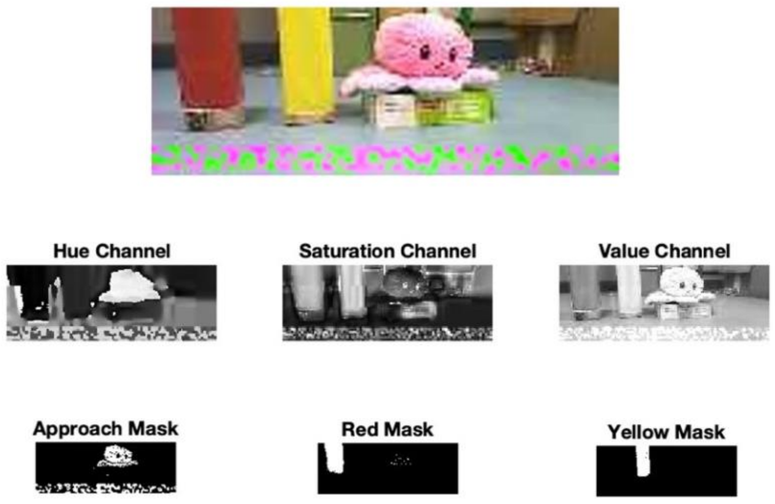

Figure 2: Colour Calibration Workflow

The script applies a binary mask to each image based on the defined HSV ranges and then displays several figures: the original image, the Hue, the Value, the Saturation, and the segmentation result. By analysing these figures, we can sample the HSV values and fine-tune the ranges for each captured image. This iterative process continues until we achieve viable segmentation across all images with a minimal number of false positives.

Table 1: Calibrated HSV Values used in the Demonstration. Note the two hue ranges for red.

COLOUR	VALUE	LOW	HIGH
RED	H	0|300	20|360
	S	150	240
	V	120	200
YELLOW	H	35	60
	S	180	230
	V	210	255
PINK	H	250	360
	S	20	180
	V	220	255

Path Finding

Navigation

The pathfinding system for the robot incorporates object avoidance and colour detection modules. By default, the robot moves forward at a speed of 0.3 m/s. When an obstacle is detected, the robot halts and spins clockwise until the obstacle is no longer detected. This results in a strong rightward navigational bias, often causing the robot to hug the left wall and move in a clockwise circle. While this isn't inherently problematic, it can lead to the robot repeating the same path and neglecting other potential paths.

Decision Making

Priority System

The robot uses a priority system to decide what input to act on first. This system allows the robot to respond effectively and manage its inputs.

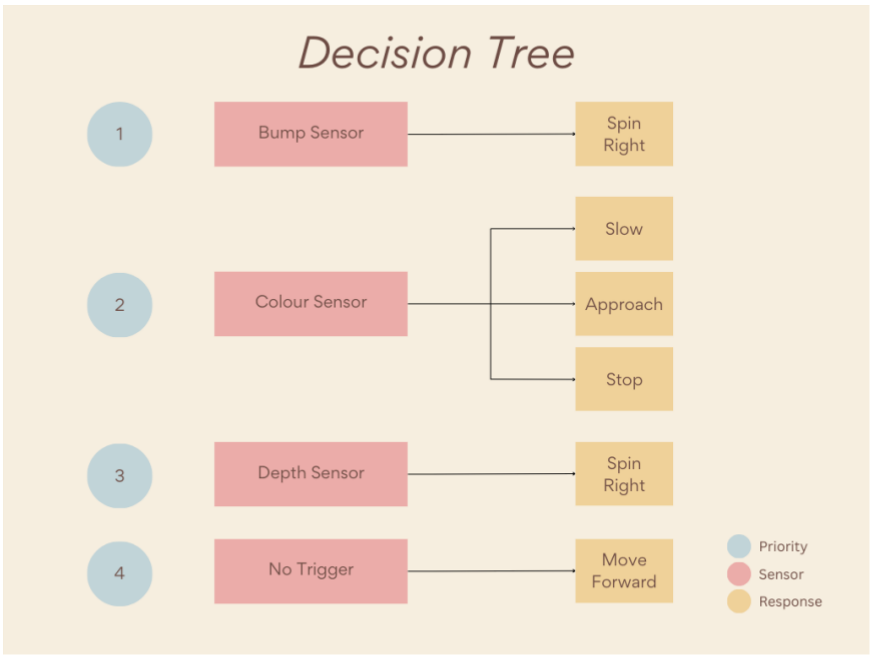

Figure 3: The Decision Tree showing the priority given to each system/sensor and the response

We use a status system to determine the actions the robot will take. Each module—Depth, Bump, and Colour—outputs a flag indicating its activity status. Additionally, the colour sensor outputs a special flag for the slow condition. This flag operates differently from the others by allowing the programme to continue functioning normally but reduces all speed values by half. This ensures the system continues tracking and avoiding obstacles, including those associated with the slow colour.

Justifications

It was decided that the bump system should always be given priority. If the bump sensor is triggered, it indicates that the depth sensor has failed to detect an object, putting us at risk of a significant collision. Following this, we wanted to ensure that the goals were given the utmost importance. The robot should stop or slow down as quickly as possible when necessary. Additionally, we wanted to ensure that the approach colour is not lost once detected, so anything less than a physical collision will not stop it from tracking the object. Finally, to navigate around obstacles effectively, we allow the depth sensor to take over if no other pressing triggers are present. If no triggers are active, the robot will simply continue forward until another trigger presents itself.

Architecture

The colour detection system performs extensive pre-processing before passing its output to the status controller. While pre-processing is common among all systems, the colour detection system handles a significant number of tasks such as tracking, detecting, slowing down, stopping, and calculating pixel areas. Consequently, it undertakes the most processing before outputting its velocity and status.

We also retain a remnant of the old control system, which involved issuing string-based commands to the control module. This approach allowed us to clearly separate decision-making from the translation of those decisions into velocity values. It also facilitated troubleshooting, as we could easily display the robot's actions at any given time. However, after integrating the bump and object detection modules, we determined that they would not benefit much from this approach, so we abandoned the old system in favour of a simpler one that passes velocity values directly to the controller. Since the colour control system was already built around the command system, we made only minor modifications to integrate it with the other modules.

Optimisations

A significant consideration was improving the speed at which the robot could respond to stimuli. Before optimisation, we noticed several sub-optimal behaviours:

- The robot would overshoot the target during tracking and get stuck in a back-and-forth loop.
- The robot would turn for longer than required when the bump sensors were triggered.
- The robot would get closer to objects than expected due to not reacting quickly enough to the depth information.
- The robot would sometimes overturn when reacting to depth information, resulting in odd and inconsistent headings.

To address these issues, we implemented several optimisations:

- **Reduced Speeds**: Initially, we lowered the robot's speeds to minimise the effects of slow reactions. While this provided some improvement, it was not an ideal solution as we now performed the tasks at a sluggish pace.
- **Cropped Camera Feed**: We cropped the colour detection camera feed to remove extraneous information that hindered the robot's performance. This significantly improved processing efficiency.
- **Adjusted Sensor Sample Rates**: We adjusted the sensor sample rates to ensure the robot acted on the most current information, enhancing its responsiveness.
- **Optimised Image Processing**: We consolidated the colour processing by using a single Kinect Get Image block instead of separate blocks for each colour subsystem.

These optimisations allowed us to restore the robot to its maximum speeds while ensuring it could perform all tasks effectively and accurately.

Demonstration Results

Performance

During our demonstration, the robot completed all tasks successfully. It entered the maze and initially navigated towards the left, making a full circuit around the maze. When the stop marker (red) was placed in front of it, the robot stopped as expected. Following this, the slow marker (yellow) was positioned quite close to the robot, triggering it to slow down but also to immediately navigate around the marker. In subsequent attempts, placing the marker further away better demonstrated the slow-down effect and successful object avoidance.

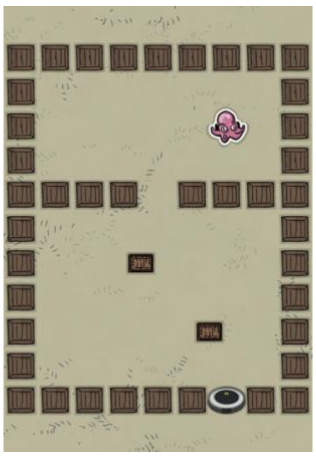

Figure 4: Map of the Maze, showing the Target (Pink Octopus) and the Robot at the start point

The robot bumped into a central obstacle once, which activated the bump sensors, allowing the robot to navigate out of the situation. Eventually, the robot found the exit and entered the zone containing the target (a pink octopus). Initially, the robot faced the target at an obtuse angle and did not lock on immediately. It circled the target once before aligning itself from a distance and adjusting its heading to centre its approach path. The robot then navigated directly to the target and stopped just short of it as required.

Videos

Object Tracking and Approach (33 Seconds)

Navigation and Colour Response (3 Minutes, 4 Seconds)

Mapping

Due to a technical error, we did not manage to save the mapping output for the demonstration. We had verified that the mapping was working in previous sessions. But are unable to include it here.

Conclusions

Overall, the robot successfully met all the requirements. Its primary subsystems - Colour Detection and Object Avoidance - performed well, enabling it to locate the target within a few minutes. However, its navigation system could benefit from improvements, as it currently relies mostly on chance to find the target and explore the environment. We also did not achieve consistent wheel calibration. Our attempts at calibration showed inconsistency between runs, even with the same robot.

Improvements

Implementing a navigation algorithm that counts the number of object detections on the left and right sides of the robot could enhance its pathing by focusing on areas with fewer obstacles. Additionally, we intended to use path mapping to avoid revisiting already explored spaces, but we did not manage to implement this feature.

A better wheel calibration system is needed to ensure more predictable movement. We also attempted to implement a system that uses the heading from the encoder. But we encountered unexpected behaviour when trying to use a continuous system aimed at aligning the robot with the eight primary compass directions.

We also noted an issue with the robot’s ability to avoid objects once it began tracking the target. The robot was blind to objects in its path once it had recognised the target colour. Although the bump sensors helped manage this, it would be better to modify the code to allow both systems to operate more seamlessly together.

Statement of Authorship

- I/we hold a photocopy/copy of this work which can be produced if the original is lost/damaged.
- This work is my/our original work and no part of it has been copied from any other student's work or from any other source except where due acknowledgement is made.
- No part of this work has been written for me/us by any other person except where such collaboration has been authorised by the lecturer/teacher.
- I/we have not previously submitted this work for any other course/unit.
- This work may be reproduced, communicated, compared and archived for the purpose of detecting plagiarism.
- I/we give permission for a copy of my/our marked work to be retained by the school for review and comparison, including review by external examiners.
- I/we understand that plagiarism is the presentation of the work, ideas or creation of another person as though it is my/our own. It is a form of cheating and is a very serious academic offence that may lead to expulsion from the University. Plagiarised material can be drawn from, and presented in, written, graphic and visual form, including electronic data, and oral presentations. Plagiarism occurs when the origin of the material used is not appropriately cited.
- Plagiarism includes the act of assisting or allowing another person to plagiarise or to copy my/our work.

Appendix

Colour Detection Functions

```
function Iout = Reduce_Resolution(Iin)
    topCut = 100;
    bottomCut = 160;

    % Get the size of the input image
    [m, n, ~] = size(Iin);

    % Cut off the specified number of rows from the top and bottom
    IinCropped = Iin((topCut + 1):(m - bottomCut), :, :);

    % Get the size of the cropped image
    [mCropped, nCropped, ~] = size(IinCropped);

    % Reduce the resolution by selecting every 4th pixel
    Iout = IinCropped(1:4:mCropped, 1:4:nCropped, :);
end
```

Function 1: Resolution Reduction including image cropping. Cuts the top and bottom pixels off the image and then samples every fourth pixel.

```
function J = Segmented(I, hmin1, hmax1, hmin2, hmax2, smin, smax, vmin, vmax)
    % Convert the image to HSV color space
    hsvImage = rgb2hsv(I);

    % Extract individual HSV channels
    H = hsvImage(:,:,1) * 360; % Scale H to 0-360 for easier range specification
    S = hsvImage(:,:,2) * 255; % Scale S to 0-255
    V = hsvImage(:,:,3) * 255; % Scale V to 0-255

    % Create a binary mask based on the first set of specified hue thresholds
    mask1 = (H >= hmin1 & H <= hmax1) & ...
            (S >= smin & S <= smax) & ...
            (V >= vmin & V <= vmax);

    % Create a binary mask based on the second set of specified hue thresholds
    mask2 = (H >= hmin2 & H <= hmax2) & ...
            (S >= smin & S <= smax) & ...
            (V >= vmin & V <= vmax);
    % Combine the two masks
    combinedMask = mask1 | mask2;

    % Create the output binary image
    J = uint8(combinedMask) * 255;
end
```

Function 2: Image Segmentation. Applies HSV ranges to calculate binary masks for valid pixels. These masks are then applied to the image to identify pixels of interest. Note this function shows the inclusion of a secondary hue range to allow for the lower hues found in red-orange colours.

```
function [xc,A] = Compute_Stats(J)
I = double(J)/255;
[m,n] = size(I);
A = sum(sum(I));
relative_no_pixels_in_column = sum(I)/A;
column_numbers = 1:n;
xc = sum(column_numbers.*relative_no_pixels_in_column);
```

Function 3: Calculation of Pixel Area (A) and Centroid (xc). Adds up valid pixels and calculates the horizontal centre for use in tracking.

```
function [Command, M, Colour] = DecideCommand(xc_stop, A_Stop, xc_Approach,
A_Approach, xc_Slow, A_Slow, Alimits, xcLow, xcHigh)

    % Define Colour Areas
    ColourAreas = [A_Stop, A_Approach, A_Slow];

    % Get the maximum area and its index
    [M, I] = max(ColourAreas);

    % Default Command and Colour
    Command = "Path";
    Colour = "None";

    % Determine Colour based on the maximum area
    if M > 20
        if I == 1
            Colour = "StopColour";
        elseif I == 2
            Colour = "TrackedColour";
        elseif I == 3
            Colour = "SlowColour";
        end
    else
        Colour = "None";
        Command = "Path";
    end

    % Control logic based on area limits
    if M > Alimits(2) && Colour == "TrackedColour"
        Command = "Stop";

    elseif M > Alimits(1)

        % Decide what to do based on Colour and xc value
        if Colour == "TrackedColour"
            Command = Track(xc_Approach, xcLow, xcHigh);
        elseif Colour == "StopColour"
            Command = "Stop";
        elseif Colour == "SlowColour"
            Command = "Slow";
        end
    end

end

function Command = Track(xc, xcLow, xcHigh)
    if xc<xcLow
            Command = "SRig";
        elseif xc>xcHigh
            Command = "SLef";
        else
            Command = "Forw";
    end
end
```

Function 4: Colour Control Script including Tracking and the original string based command system.

```
function [cv, av, Status] = Controller(Command, V_A, Omega)

    Status = 1;
    if Command == "Spin"
        av = Omega; cv = 0;
    elseif Command == "Stop"
        cv = 0; av = 0;
    elseif Command == "Forw"
```

```
        cv = V_A; av = 0;
    elseif Command == "Slow"
        Status = 2;
        cv = 0; av = 0;
    elseif Command == "Left"
        av = -Omega; cv = 0.3;
    elseif Command == "Righ"
        av = Omega; cv =0.3;
     elseif Command == "SLef"
        av = -Omega; cv =0;
    elseif Command == "SRig"
        av = Omega; cv =0;
    elseif Command == "Path"
        Status = 0;
        cv = 0; av = 0;
    else
        cv = 0; av = 0;
    end
end
```

Function 5: Colour Command to Velocity. Reads the command and converts it to actual velocity values. Also sets status for Colour System.

Object Avoidance Functions

```
function [x, y] = Local_2D_Point_Cloud(D)
    % Determine the number of rows in the depth matrix D
    num_rows = size(D, 1);

    % Calculate the index of the middle row
    middle_row_index = ceil(num_rows / 2);

    % Extract the middle row of the depth matrix D
    x = double(D(middle_row_index, :)) / 1000;

    % Filter out depths less than 0.5 meters
    x(x < 0.5) = 0;

    % Generate the angle array for horizontal field of view
    a = linspace(28.413, -28.499, 640) * pi / 180;

    % Calculate the y coordinates using the depth values and angles
    y = x .* tan(-a);
```

```
end
```

Function 6: Depth Data extraction. Calculates a 2D point cloud along a single horizontal plane.

```
function [cv,av, Status] = Controller(x, y, th, Omega)

r = sqrt(x.^2+y.^2);
r = r(x>0);

if min(r)<th
 av = -Omega; cv = 0;
 Status = 1;
else
 Status = 0;
 av = 0; cv = 0.3;
end

end
```

Function 7: Respond to Depth Data. Calculates distances to objects and determines a velocity output. Also sets status for Depth System.

```
function Status  = Controller(Bump_R,Bump_C,Bump_L)

if Bump_R || Bump_C || Bump_L
    Status = 1;
else
    Status = 0;
end
```

Function 8: Responds to Bump Sensor Data. Determines whether a bump has occurred. Status here is independent from the final status system. It simply outputs whether or not a bump has occurred.

```
function [cv, av, Status] = fcn(Bump, Omega)

if Bump == 1
    av = -Omega; cv = 0;
    Status = 1;
else
    Status = 0;
    av = 0; cv = 0;
end
```

Function 9: Outlines Response to a Bump Detection. Also sets status for Bump Detection System.

Decision Making Functions

```
function [CV, VA] = StatusManager(COL_CV, COL_VA, ColourStatus, PATH_CV,
PATH_VA, PathStatus, BUMP_CV, BUMP_VA, BumpStatus)

    if BumpStatus == 1
        CV = BUMP_CV; VA = BUMP_VA;
    elseif ColourStatus == 2
        CV = PATH_CV * 0.5; VA = PATH_VA * 0.5;
    elseif ColourStatus == 1
        CV = COL_CV; VA = COL_VA;
    elseif PathStatus == 1
        CV = PATH_CV; VA = PATH_VA;
    else
        CV = 0.3; VA = 0;
    end
end
```

Function 10: Robot Controller. Determines which subsystem to prioritise based on Status Flags. Also manages the unique slow flag **ColourStatus = 2**.

Helper Functions

```
function [vR,vL] = Center2RL(vC , Omega)
d=0.235;
vR = vC + Omega*d/2;
vL= vC - Omega*d/2;
vR = min(vR,0.3);
vL = min(vL,0.3);
end
```

Function 11: Converts angular velocity and central velocity values to left and right wheel speeds.

```
% Read the input image
I = imread('PinkOcto2.BMP');
imshow(I);

% Convert to HSV and display channels
hsvImage = rgb2hsv(I);
H = hsvImage(:,:,1) * 360; % Scale H to 0-360
S = hsvImage(:,:,2) * 255; % Scale S to 0-255
V = hsvImage(:,:,3) * 255; % Scale V to 0-255
```

```
figure;
subplot(1,3,1); imshow(H, []); title('Hue Channel');
subplot(1,3,2); imshow(S, []); title('Saturation Channel');
subplot(1,3,3); imshow(V, []); title('Value Channel');

% HSV ranges for Red, Green, and Yellow (to be updated based on your calibration
hmin_red1 = 300; hmax_red1 = 360;
hmin_red2 = 0; hmax_red2 = 20;
smin_red = 150; smax_red = 240;
vmin_red = 120; vmax_red = 240;

hmin_green = 250; hmax_green = 360;
smin_green = 20; smax_green = 180;
vmin_green = 220; vmax_green = 255;

hmin_yellow = 30; hmax_yellow = 65;
smin_yellow = 180; smax_yellow = 230;
vmin_yellow = 210; vmax_yellow = 255;

% Detect Red Color
redMask1 = Segmented(I, hmin_red1, hmax_red1, smin_red, smax_red, vmin_red,
vmax_red);
redMask2 = Segmented(I, hmin_red2, hmax_red2, smin_red, smax_red, vmin_red,
vmax_red);
redMask = max(redMask1, redMask2);
figure; imshow(redMask); title('Red Mask');

% Detect Approach Color
greenMask = Segmented(I, hmin_green, hmax_green, smin_green, smax_green,
vmin_green, vmax_green);
figure; imshow(greenMask); title('Approach Mask');

% Detect Yellow Color
yellowMask = Segmented(I, hmin_yellow, hmax_yellow, smin_yellow, smax_yellow,
vmin_yellow, vmax_yellow);
figure; imshow(yellowMask); title('Yellow Mask');

% Step 4: Function to segment colors based on the calibrated HSV ranges
function J = Segmented(I, hmin, hmax, smin, smax, vmin, vmax)
    hsvImage = rgb2hsv(I);
    H = hsvImage(:,:,1) * 360; % Scale H to 0-360
    S = hsvImage(:,:,2) * 255; % Scale S to 0-255
    V = hsvImage(:,:,3) * 255; % Scale V to 0-255

    mask = (H >= hmin & H <= hmax) & ...
           (S >= smin & S <= smax) & ...
           (V >= vmin & V <= vmax);

    J = uint8(mask) * 255;
end
```
Function 12: Calibration Script. Takes an image and applies all the HSV Colour Detection rules that are applied while the robot is running. Then outputs useful figures that can be used to determine how well the robot is segmenting the images. From there the values can be tweaked.