Demonstrating Color Detection and Obstacle Avoidance with Simon

Justin Meinecke

Zhenlu Song

Table of contents

Background and Description of Problem

Simon is a game in which four colored panels light up in a pattern, which the player repeats by touching those panels in order. This pattern grows more complex over time. We plan to represent the game space as a set of four colored squares taped to the floor. The bounds of the game space will be delineated by a colored tape rectangle on the floor.

The Turtlebot we are using for this project has a webcam taped to it in a downward-facing position in order to detect colored tape and an Asta 3D depth camera. The robot also has a known map available to it which was constructed in an assignment earlier in this course. The robot will be able to use SLAM to find its position in that map and therefore the playing area which is a small portion of the known map.

Turtlebot will receive commands from another computer on the same wireless network in the form of a list:

- [green, blue, red, yellow, red, blue, …]

Only one finite list will be sent at a time to eliminate confusion. Each one of these colors will correspond to a goal state Turtlebot recorded when it found each color so when commanded to go to ‘blue’ Turtlebot will move to its corresponding location.

In later phases we will add obstacles to the playing area that we believe the Turtlebot will navigate around automatically.

The nature of this research project is experimental. As a result, we experienced difficulties stemming from our lack of knowledge in this field. Limitations and difficulties we experienced will be addressed later.

Main research objective: To get turtlebot to respond to a list of colors by checking for the locations of colored pads (green, red, blue, and yellow) in its playing area and subsequently visit those pads in the order the colors appear in the list.

Methodology

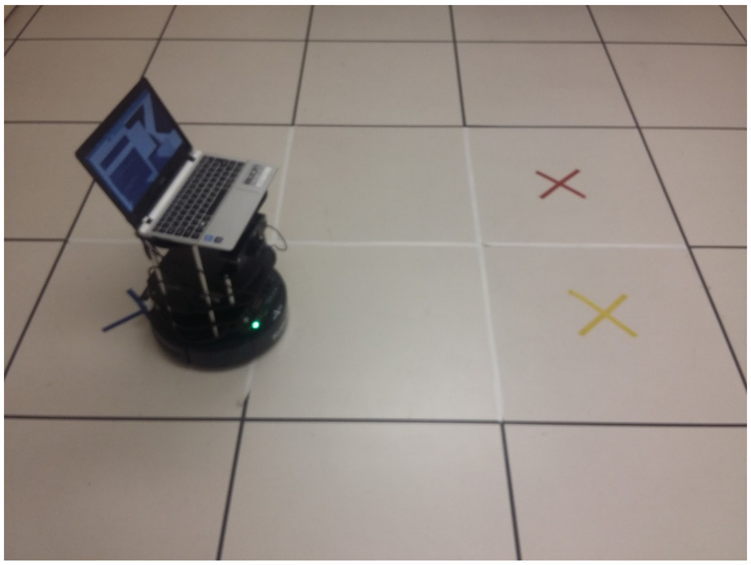
1. Setup

Requirements:

In order to begin this experiment we need to make a version of simon that the Turtlebot can play. The requirements for doing so are:

- The turtlebot has to be able to access the colors it needs to get to
- There has to be a well defined playing area
- The turtlebot has to be able to see the colors
- The turtlebot has to know which colors to press in what order

1.1. P lay Area



The environment we were conducting this experiment in already had black-bordered tiles. We took advantage of that and just used white tape to cover the tile lines on the interior of the playing area.

- The colored X's are the same as the colored pads on the Simon game.
- The Turtlebot will be able to see the tape with the downward pointing webcam
- The turtlebot drives over the X's in order to trigger them in the same way pressing a pad

1.2. The Turtlebot

In order to start working with the Turtlebot we had to use this command:

roslaunch turtlebot_bringup minimal.launch --screen

To use the Creative Lab's webcam:

roslaunch usb_cam usb_cam-creative.launch

I found this package on github at this URL:

https://github.com/bosch-ros-pkg/usb_cam.git

We had to make filters to detect colors (described in part 2.) By using the range detector program we could quickly and easily find the hsv range of the blue, green, red, and yellow tape as well as the black lines between the tiles.

Rosrun simon_project range_detector_ros -f hsv

This is the primary node for this project.

Rosrun simon_project simon_the_turtle.py

1.3. Introducing the Second Computer

Not Implemented

1.4. Building and Using the Map

1.4.1. Building

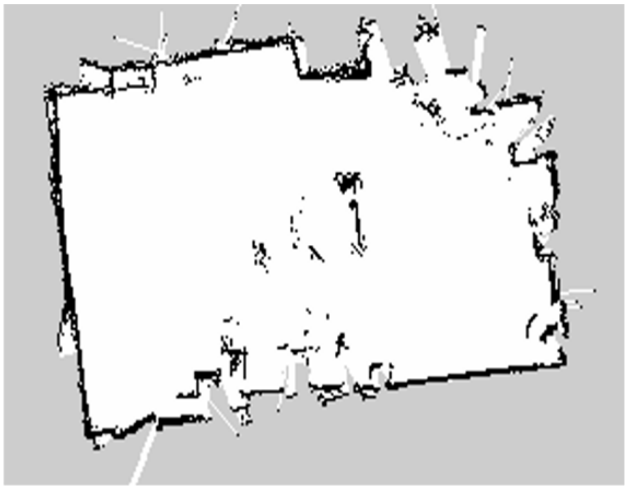
This is the sequence of commands we used to start the mapping process. The Turtlebot is teleoperated and moved around the room to build a map using the keyboard as a controller.

```
roslaunch turtlebot_navigation gmapping_demo.launch --screen
roslaunch turtlebot_teleop keyboard_teleop.launch --screen
roslaunch turtlebot_rviz_launchers view_navigation.launch --screen
```

The map is then saved to a location of our choice with:

```
rosrun map_server map_saver -f /path/to/your_map
```

1.4.2. Using



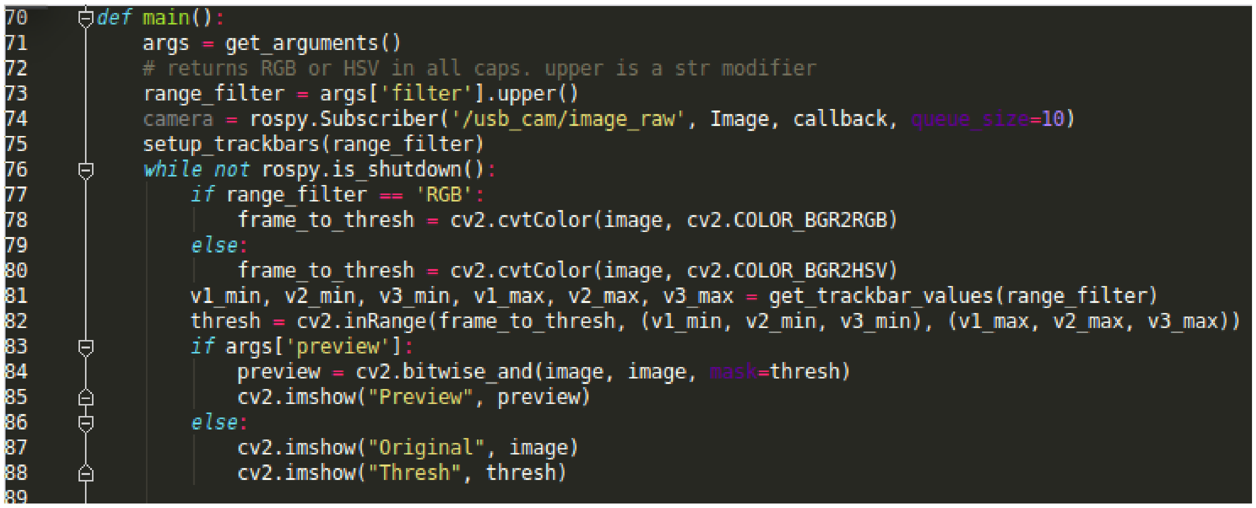
The map above is the result of the mapping process described in section 1.4.1. In order to use the map we built, we had to run the following command:

```
roslaunch turtlebot_navigation amcl_demo.launch map_file:=/path/to/your_map.yaml
```

This brings up an RVIZ window in which you can publish a goal to the Turtlebot for it to navigate toward autonomously as a proof of concept. We can also publish these same goal points to the Turtlebot with custom code.

2. Color Detection

The process we used to filter the images



```
70  def main():
71      args = get_arguments()
72      # returns RGB or HSV in all caps. upper is a str modifier
73      range_filter = args['filter'].upper()
74      camera = rospy.Subscriber('/usb_cam/image_raw', Image, callback, queue_size=10)
75      setup_trackbars(range_filter)
76      while not rospy.is_shutdown():
77          if range_filter == 'RGB':
78              frame_to_thresh = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
79          else:
80              frame_to_thresh = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
81          v1_min, v2_min, v3_min, v1_max, v2_max, v3_max = get_trackbar_values(range_filter)
82          thresh = cv2.inRange(frame_to_thresh, (v1_min, v2_min, v3_min), (v1_max, v2_max, v3_max))
83          if args['preview']:
84              preview = cv2.bitwise_and(image, image, mask=thresh)
85              cv2.imshow("Preview", preview)
86          else:
87              cv2.imshow("Original", image)
88              cv2.imshow("Thresh", thresh)
89
```

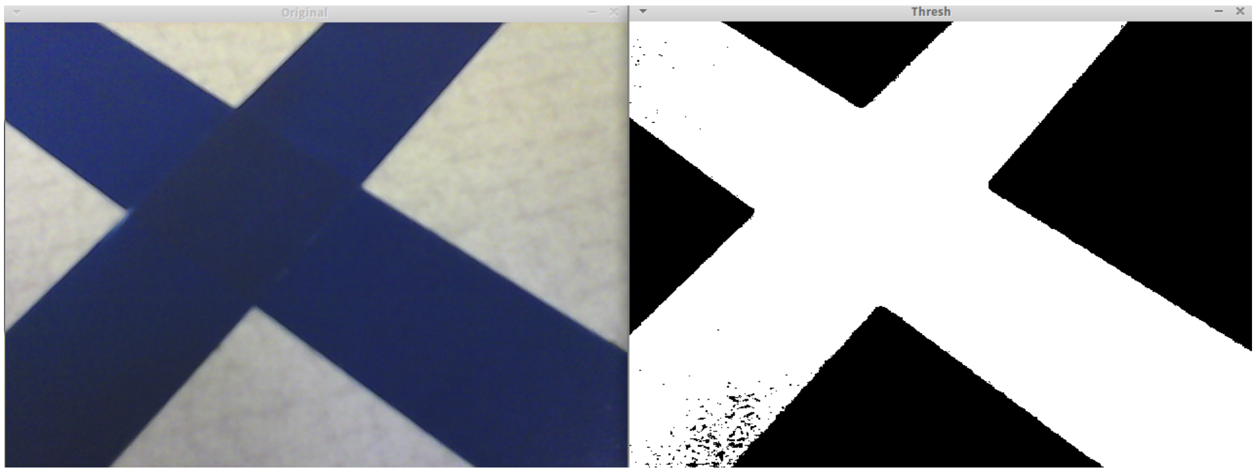
Line 74: Subscribe to usb camera topic to receive images

Line 80: Convert the image from the webcam to HSV values

Line 81: Define a range for filtering out everything but the values between the min and max values

Line 82: Apply the values from the previous step. Everything outside the range is now black

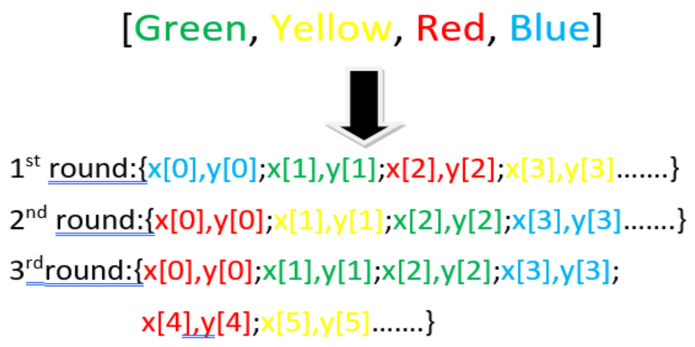
Line 87, 88: Displays the original image from the webcam and the filtered image Respectively. Below you can see the result of that process for the color blue. Original on the left and filtered on the right.



3. Navigation

Based on the map we have built before; the position of color panels' locations can also be defined. If we want to develop a ROS program that will allow the robot to navigate to those four locations, we first need to know what are the (x, y) coordinate of these locations onto the map. This is what the color detect will do. We will let the turtle bot do a free movement on the map area. During that time, the turtle bot use additional mounted webcam, point down, detected all four color panels. Every color panel successfully detected the turtle bot will publish odometry information, the (x, y) coordinate positions of color panels, over ROS.

When the Simon game has started, the sequence of color panels' position will be assigned to a sequence of {x[i], y[i]} coordinates, which start from x[0], y[0]. A " int i=0; while (i<N); i++" loop will order the turtle bot to visit each position one by one. These coordinates will be used to define a navigation mission that we submit to the robot's navigation stack to execute it. The good news is that any turtle bot on ROS runs the move_base navigation stack which allow the robot to find a path towards a goal and execute the path following while avoiding obstacles. We also need to let the robot to turn. It starts by declaring a Twist message to send velocity commands and a declaration of tf transform listener to listen and capture the transformation between the odom frame and the base_footprint frame. Then change the angles to radians and then start publishing topics according to the right angles until the robot reaches a certain angle.



[Green, Yellow, Red, Blue]

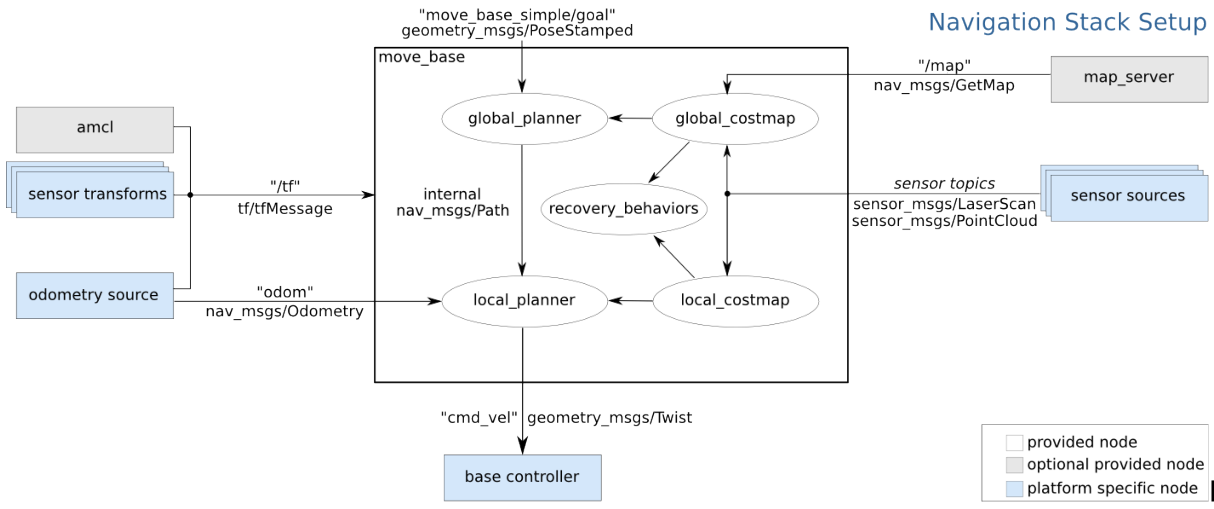
1st round:{x[0],y[0];x[1],y[1];x[2],y[2];x[3],y[3].......}

2nd round:{x[0],y[0];x[1],y[1];x[2],y[2];x[3],y[3].......}

3rd round:{x[0],y[0];x[1],y[1];x[2],y[2];x[3],y[3];
x[4],y[4];x[5],y[5].......}

3.1. Obstacle Avoidance

The navigation stack of ROS has a MoveBaseAction action server that receives navigation goals request, and then finds a global path from the robot location to the goal location through the Global Path Planner, and once a path is found, it executes the path while avoiding obstacle through the Local Path Planner until it reaches the destination or fails to do so for any reason (like unexpected obstacle found on path).



In

Figure. 999 we can see the role of global_planner package. If we take a look at one of its parameter, ~<name>/use_dijkstra (bool, default: true). If true, use dijkstra's algorithm. Otherwise, it will switch to A* algorithm.

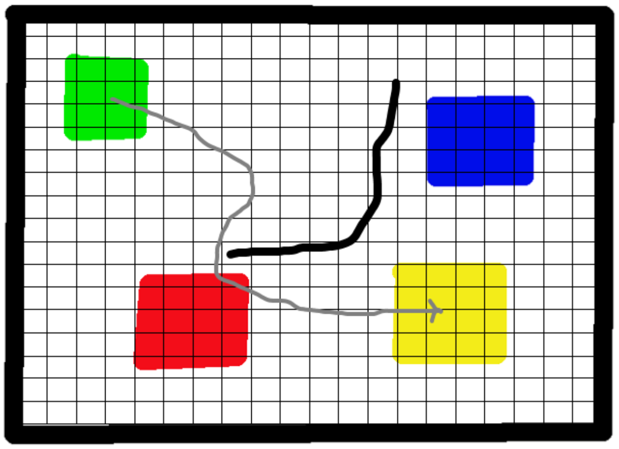
3.2. Path Planning Algorithm

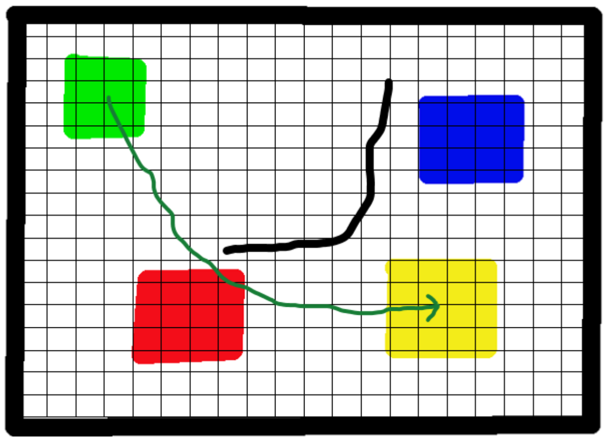
Dijkstra’s algorithm works by visiting vertices in the graph starting with the object’s starting point. It then repeatedly looks over the closest not-yet-examined vertex, adding its vertices to the set of vertices to be examined. It expands outwards from the starting point until it reaches the goal. Dijkstra’s algorithm is guaranteed to find a shortest path from the starting point to the goal, if none of the edges have a negative cost.

The Greedy Best-First-Search algorithm works in an equivalent way, except that it has some estimate (called a heuristic) of how far from the goal any vertex is. Instead of selecting the vertex closest to the starting point, it selects the vertex closest to the goal. Greedy Best-First-Search is not guaranteed to find a shortest path. However, it runs much quicker than Dijkstra’s algorithm because it uses the heuristic function to guide its way towards the goal very quickly.

For example, if the goal is to the south of the starting position, Greedy Best-First-Search will tend to focus on paths that lead southwards.

However, both have a good performance in a map without obstacles, and the calculated shortest path is basically a straight line. When we implant obstacles on the map, the result will show their pro and con:





The Figures are the processes of Greedy Best-First-Search, on the left, and Dijkstra, on the right. The trouble is that Greedy Best-First-Search is "greedy" and tries to move towards the goal even if it's not the right path. Since it only considers the cost to get to the goal and ignores the cost of the path so far, it keeps going even if the path it's on has become really long. A* was developed in 1968 to combine heuristic approaches like Greedy Best-First-Search and formal approaches like Dijsktra's algorithm. It's a little unusual in that heuristic approaches usually give you an approximate way to solve problems without guaranteeing that you get the best answer.

A* is like Dijkstra's algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. The secret to A* algorithm's success is that it combines the pieces of information that Dijkstra's algorithm uses (favoring vertices that are close to the starting point) and information that Greedy Best-First-Search uses (favoring vertices that are close to the goal). In the standard terminology used when talking about A*, g(n) represents the exact cost of the path from the starting point to any vertex n, and h(n) represents the heuristic estimated cost from vertex n to the goal. In the above diagrams, the yellow (h) represents vertices far from the goal and teal (g) represents vertices far from the starting point. A* balances the two as it moves from the starting point to the goal. Each time through the main loop, it examines the vertex n that has the lowest f(n) = g(n) + h(n).

Limitations

Justin Meinecke: Because of my lack of knowledge of OpenCV, most of my time was spent learning that. So much in fact that I had hardly any time to work on much else. I underestimated what I could get done in the short amount of time I had left to work on this project. As a result, this project is far from complete. However, I do believe that given more time I could complete this project.

Zhenlu Song: This is a very interesting class for me. For all the knowledge I have learned is how to use ros combined with programming language(C++, python) to control a robot. It gives me a clear ideal on how those real world robots been designed. However, due to my programming skill shortage, this class really give me a hard time. Almost everything in this class is new to me, but this is a valuable challenge to myself. The final project is based on real world situation. I start from a blank paper and keeping adding the new knowledge I have learned from this class and research. I have read a lot of research papers about ROS， SLAM，coding, color detection, robot language, etc. Even though, I could not completely finish this project, the knowledge I have learned is real.