MEM 800: Mobile Robotics II - Final Report

Treasure Hunt

Alex Alspach
Sean Mason

Date:	4/24/2012
Professor:	M. Ani Hsieh
TA:	James Worcester

Abstract

Using the SRV-1 robot with a camera, a 2.6m square maze was to be navigated and “treasure” locations were to be found. The robot was successfully able to navigate the maze using an A* path planner and a piecewise PD controller with only one collision. All of the treasures were found in the correct locations using color segmentation code that compared the position of the centroids for different color blobs. In addition, one nonexistent treasure was incorrectly found due to colors detected in the environment outside of the maze.

Introduction

For this final project, the objective was to have a robot autonomously navigate through a maze and search for treasure. The treasure locations should then be mapped to the location at which they were detected by the robot. The robot in use is a modified version of the SRV-1 Robot that includes a camera. Communication is established through a wireless network and commands are sent via a MATLAB programming environment. The treasures consisted of two squares of different colors stacked vertically (figure 1).

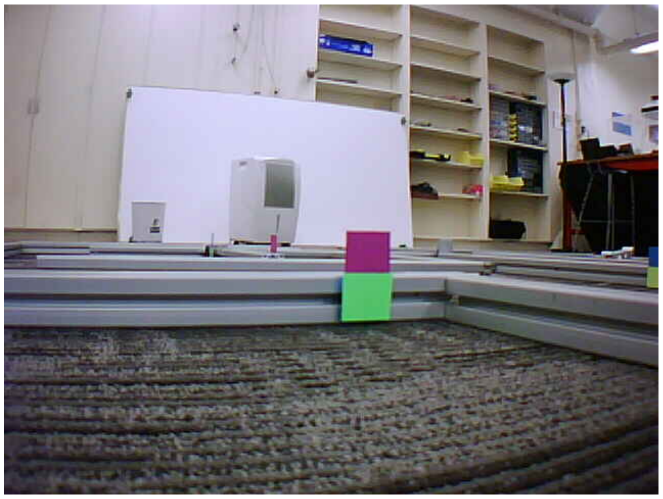

Figure 1: Example of a treasure image. Images similar to this were provided beforehand for calibration purposes.

The maze was built using 80/20 beams and a PNG format map was provided (figure 2).

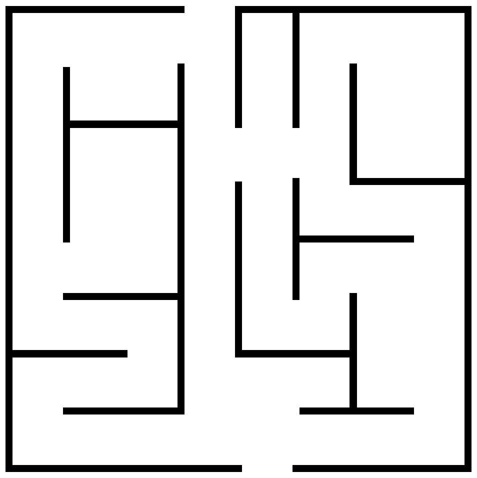

Figure 2: Maze image representing the actual maze that measured 2.6m square.

Methodology

The arena in which the robot was required to locate treasure (bi-colored rectangular targets) was explored room-by-room. In each of these rooms, a hunting algorithm was executed to determine the existence of treasure in each room.

Path Planning

The maze, with a PNG format map provided along with the project solicitation, was divided into multiple rooms and areas where treasure was likely to be found. From the start point to the first room and from room to room, an A* algorithm is used to plan a list of waypoints that the robot can follow.

The maze PNG is first transformed into an occupancy matrix with ones (1) signifying obstacles and zeros (0) being free navigable space. This map is then padded using a 2-D convolution method with a kernel of ones then saturated back to a matrix of just binary elements. This padding of the matrix is executed to provide a new map with thick obstacles and narrow corridors. This causes the path planner to be constrained to planning only in the centers of rooms and hallways, leaving as much space as possible between the robot and the walls. The A* path planning algorithm is tuned to provide the shortest, most-direct path from start to goal locations by weighing each pixel according to euclidean distance from the goal and from the previously analyzed pixel. The dense path produced is then down-sampled for smoother path following without corner cutting and transformed from pixel coordinates to floor coordinates (figure 3).

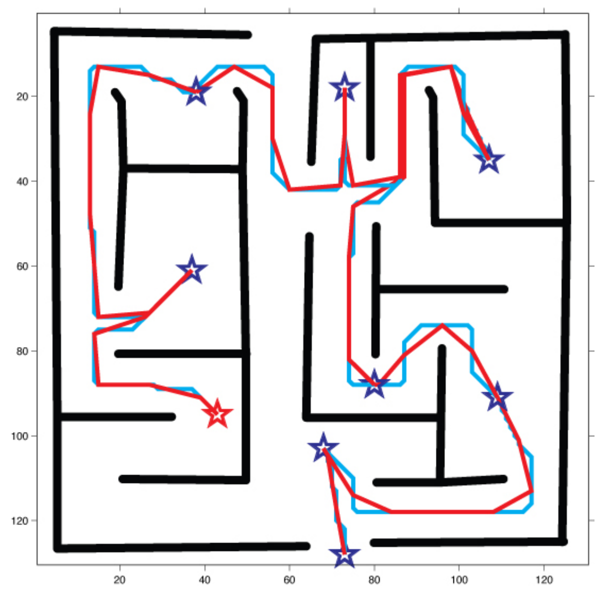

Figure 3. Map obtained with colored blobs at each arena wall-corner and overhead camera data. The A* path between each of the goals is also plotted with the original path in cyan and the downsampled path in red.

Navigation

In order to smoothly navigate through the list of waypoints created by A*, a piecewise controller was implemented. The controller was piecewise so that the velocity at which the robot traveled would remain constant unless a large turn was necessary. The first part of the controller is for when the error between the current orientation of the robot and the angle from the robot to the goal is less than some tolerance. In this mode, the driving speed is fixed and PD control is implemented on the angular velocity. The derivative gain was put in place to compensate for oscillations caused by overshoot observed in early testing. The second mode of the controller is used for large angular error. In this mode, the robot moves very slow and turns very fast. This was very useful when cornering. Again, a PD controller was used on the angular velocity.

Pseudo Code

```
for each waypoint
    Tol = 2.5 cm
    angleLimit = 60 degrees
    goalRobPose = way-point
    err = inf;
    alphaOld = 0;

    while err > Tol
         get robot pose
        dx = goalRobPoseX - robPoseX;
        dy = goalRobPoseY - robPoseY;
        theta = robPoseOrientation;

        phi = mod(atan2(dy,dx),2*pi)
        alpha = angleDifference(phi,theta)

        if abs(alpha) > angleLimit
            v = 4;
            Kalpha = 16;
            omega = Kalpha*alpha+KDiff*(alpha-alphaOld);
        else
            v = 16;
            Kalpha = 20;
            omega = Kalpha*alpha+KDiff*(alpha-alphaOld);
        end
        leftWheelSpeed = (v-1/2*omega);
        rightWheelSpeed = (v+1/2*omega);

        saturate motor commands
        send drive command
        compute euclidean error

        alphaOld = alpha;
    end
end
```

Treasure Hunting

A treasure is defined as a bi-color rectangle containing two equal-sized squares of different colors aligned with one above the other (figure 4). The possible treasure color combinations include, with the top color first, magenta and green, blue and yellow, blue and green, green and blue, and magenta and yellow.

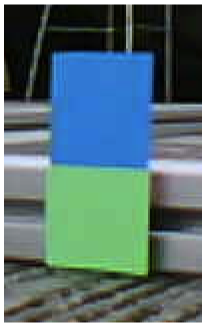
Figure 4: SRV camera image of one possible treasure.

The treasure hunting code is an extension of previously demonstrated color segmentation code. Instead of looking for just one color, the algorithm is run once for each of four colors. In pixels, centroidal location and area are output along with a matrix locating the colored pixels for each of the four colors. The centroid locations are used to determine horizontal alignment of the colors, and therefore existence of a treasure.

The color segmentation algorithm runs through each pixel on a YUV color space image and records a one (1) where the color of the pixel falls within a certain range of calibrated average values for the U and V chrominance channels. If these criteria are not satisfied, the value of the pixel is set to zero (0). This acceptable range about the mean value is also determined through calibration and is the variance in U and V for a specific color over a sample of treasure images from the SRV camera.

Calibration

To calibrate the upper and lower bounds of acceptable pixel color associated with the blob color, a frame is captured from the camera for analysis. On this image, the user selects a polygon containing the color of interest and the pixels inside this polygon are used to determine the mean and variance of the values for the selected color in the YUV color space. The mean and variance are calculated during one scan through the pixels of interest. This is possible by using a running estimated average of the values in both the U and V channels for use in calculating the U and V variance for each pixel. This estimation method saves us from iterating through the pixels more than once and provides a great estimate of mean pixel variance. The Y channel is ignored as it theoretically should only change with changes of light. Changes of color are realized in the U and V channels.

Pseudo code:

```
averageU = 0
runningAverageU =  0
runningVarianceU = [ ]
averageV = 0
runningAverageV =  0
runningVarianceV = [ ]

numberPixels = 1

For every pixel within the polygon selected

        averageU = averageU + U
        averageV = averageV + V

        runningAverageU = averageU / numberPixels
        runningAverageV = averageV / numberPixels

        runningVarianceU(numberPixels) = ( U - runningAverageU )^2
        runningVarianceV(numberPixels) = ( V - runningAverageV )^2

        numberPixels = numberPixels + 1
end

averageU = averageU / numberPixels
averageV = averageV / numberPixels

varianceU = mean(runningVarianceU)
varianceV = mean(runningVarianceV)
```

A program was written so that mean and variance data could be logged for the four treasure colors over multiple sample images. This program gave the user the ability to select two polygons in each picture and specify which color each selection contained. For any number of sample images, the color, mean and variance were saved but not combined.

When all calibration data had been collected, a second program was used to combine the data for each color into a U and V mean and variance (4 values) for each color. This algorithm, for each of the four colors, outputs the mean of the means and the minimum variance out of all trials for the U and V channels.

These values for the mean and the variance in the U and V channels are used to set bounds for considering whether or not a pixel is considered the color of interest. These bounds are specified as follows:

```
upper_bound_U  = average_U + varianceWeight*sqrt(variance_U) + blurring;
lower_bound_U  = average_U -  varianceWeight*sqrt(variance_U)  - blurring;

upper_bound_V  = average_V + varianceWeight*sqrt(variance_V) + blurring;
lower_bound_V  = average_V -  varianceWeight*sqrt(variance_V)  - blurring;
```

where both *varianceWeight* and *blurring* are tuned for each color to provide robust and clean color segmentation.

Segmentation

For each image from the SRV camera, the segmentation algorithm is run four times - once for each of the four colors. For each color, the corresponding averages and variances are used to segment out the desired color. A matrix, with the pixel locations of color of interest (marked as ones), the color's centroidal location and and number of pixels, or area of the color, are output. After the four iterations, we have four binary matrices, four areas and four centroid locations.

Pseudo code for one color:

```
Convert YUV color image to binary image
for each pixel = 1
    area = area+1
    xPos = xPos+1
    yPos = yPos+1
end

xCentroid = xPos/area
yCentroid = yPos/area
```

The binary matrices are multiplied by a value, one through four, assigned to each color. This produces four matrices that, when combined together (overwriting overlaps), can be displayed to the user as an image with the appropriate colors for the segmented blobs. This image also includes the centroid of each color (figure 5).

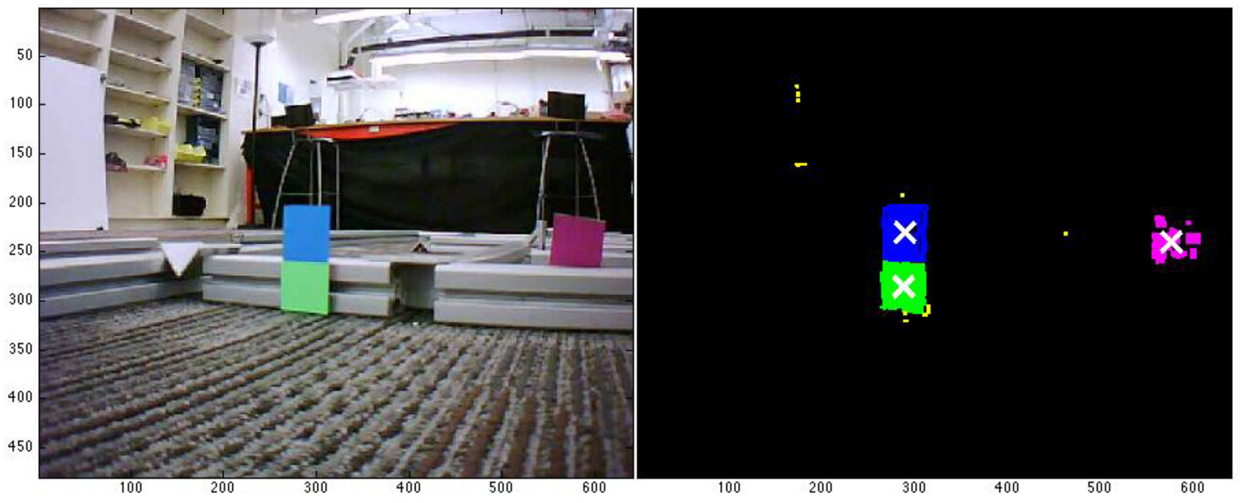

Figure 5: Visual output of color segmentation and centroid location compared to SRV camera image

The area of each color is used only to determine whether to segmented area is large enough to be considered part of the bi-colored targets. The centroid locations are compared to determine if two colors are aligned, within tolerance, on a vertical axis. If so, the pair of colors found are logged along with the room in which they were found (the robot's current location).

Logging

This hunting phase of the algorithm is only invoked at each A* goal, or selected room for searching. When the robot reaches the room, it stops then spins about one and a half turns in nine seconds.

Throughout this spin, every iteration captures a camera image from the SRV and segments out the four colors. If the centroids align for a given pair, the treasure description and room location are added to a list. An assumption being made when using this algorithm is that there will not be two of the same treasure (e.g. two with blue above green) in the same room. This allows the robot to capture as many images as possible while spinning, many in which it will detect the treasure, supporting its existence in the room. This process is completed in each room.

When the robot has reached its last goal and finished all searches, the list of treasure found is produced. Inherent to the algorithm, the list contains many duplicates that can easily be stripped out. The list of unique treasure may contain multiple different treasure in the same room or the same type of treasure in two different rooms.

Results

The robot was subjected to a few test scenarios in order to test algorithm robustness. The robot was able to successfully navigate to all of the rooms from various starting locations. Also, the robot was able to successfully find and locate all of the treasures throughout the map. One error that was observed in the camera segmentation code was the mapping of an extra treasure that did not exist in the environment. We believe that this error was most likely caused by objects in the background that our code mapped to yellow.

As the robot made its way through the list of waypoints, it still exhibited a small overshoot when correcting for the angle differences. The result of this was that it did not stay centered in the hallways and occasionally had to backtrack to reach missed waypoints. Despite the overshoot, there was only one instance where the robot contacted a maze wall and required human intervention to be freed. The different start positions confirmed the robustness of the path planning code.

Lessons Learned

If global knowledge is to be relied on, the map must be as realistic as possible. Being away for three weeks allowed the maze to be knocked around and ultimately removed and replaced before we had our first test. While the robot's trajectory resembled what was expected, it was not without collision with the arena. Finally, after attempting to move beam-walls to match the map, we instead aimed to create a map that matched the arena's current state. Orange markers were placed at every vertex of the arena, half at a time, and the MATLAB figures were saved. After stripping out the grids and numbers, the dots from the overhead position camera were connected at the walls and a PNG format map was produced. As this map matched even the crooked walls in the arena, we had much better luck navigating when A* used this new map to plan.

A second lesson was learned when the robot was stuck against a wall. When the robot contacts the wall, it has the potential to become immobilized and may remain so indefinitely. One way to fix this issue is to introduce an integral gain that will grow over time if the robot is not moving. This will allow the robot to eventually free itself in this situation.

A final lesson learned pertains to the matter of simply getting the right start to a coding project, it is always beneficial to save working versions of code. We found ourselves saving copies under new names and relying on Dropbox to restore anything we may lose. In any future project, Git or SVN versioning software should not be avoided.