Autonomous Navigation
Project Report

Harman Kumar
2013CS10224
Computer Science
Mob: 9013026503

Aniket Bajpai
2013CS102225
Computer Science
Mob: 9717310691

Supervisor:-
Dr. Subhashis Banerjee
Professor
Department of CSE
IIT Delhi

Contents

1 Overview

The goal of the project was to build an obstacle detection and path planning unit of an autonomous robot, which would work reliably for a short distance range (5m-10m).

A significant part of our time went into setting up the sensors (Camera, IMU and wheel encoders). The steps we took to set up the robot explained later in more detail.

The next step was to test the accuracy of the sensor readings, the latency between issue and execution of commands and making sure that the controls of the robot work properly. This was done by making the bot navigate using ArUco markers.

The obstacle detection module was implemented next. This module was fine tuned to work well for a floor that consisted of white tiles. After getting a good idea of which region in the image is free space, we found out the trajectory the robot should follow.

The testing of the above module was done next. In this part, we set up a test path which was clear except for a few added obstacles. The path also required the robot to take a turn. The robot was able to successfully navigate from the start to the goal state, demonstrating that our method worked reasonably well with obstacles and turns in the control environment.

The source code can be found on:

https://github.com/harmankumar/AutoNav

Some Videos of how the bot performs can be found on:

https://drive.google.com/open?id=0B16DvzSVHFJvdzVhT3ZwbXZmNjQ

2 Setting Up the Robot

2.1 Taking Images Using Point Grey Camera

The libraries for capturing images using Point Grey were available only in C. Since the robot controller code was python based, we had to create a module for communication between C and python to directly control the camera from the bot controller. For this purpose, we decided to use Inter-Process communication using shared memory between the Point Grey C library and our bot controller. We took care to make the module robust for any general communication between C/C++ and python modules, as this would also be of use to us later while integrating other components such as the global path planner into our bot controller code.

2.2 Getting IMU Readings from mobile phone

We decided to use a mobile phone IMU to get accurate orientation measurements for the bot. A mobile phone was used as it already supplied noise corrected and filtered values from the IMU, so we did not have to deal with raw IMU data. An Android app was created to obtain these precise IMU readings from the phone. The magnetic field readings were used to get an accurate estimate of the yaw angle. The angle so obtained had an error of around 2 ° - 3 °. A UDP socket connection was established between the phone and the bot to send the calculated orientation values continuously, with low latency. IMU readings were being transmitted to the bot at a rate of 100 Hz.

2.3 Registering Wheel Encoder Ticks

There was a need to get wheel encoder ticks to get a measure of the distance moved by the bot. This distance when coupled with the orientation obtained in the previous part helped in localization of the bot. The library code of the bot did not give any function to get the wheel encoder ticks, so we had to implement and test this ourselves. Encoder ticks were recorded on the arduino from the wheel encoder pulses. Serial communication was used between the bot and arduino to get the encoder tick values on the bot.

As wheel odometry is a core functionality of the bot, we decided to include it in the API of the bot. Hence, we integrated our module into the library code of the bot so that wheel encoder readings can be directly obtained in the future via a function call to a bot library function.

3 Calibrating Robot motion using ArUco Markers

The next step was to test the accuracy of the sensor readings, and the latency between issue and execution of commands to measure how precisely the robot motion takes place before we make it autonomous. This was done by placing several ArUco markers in the lab, and allowing the robot to track these markers for navigation. After its distance to the marker falls below a threshold, it turns until it detects the next marker. It then navigates by tracking the next marker. This process is continued until the last marker, where it stops.

The outcome was that that the robot was able to navigate till the goal state from the start and we were able to establish the fact that we can properly control the robot. After the completion of this experiment, we had a good idea of the response time of the bot, and the translation and rotation errors involved in the movement of the bot.

4 Free Space Detection

Our next step was to make the bot autonomous, now that we have all the sensor readings and we know how to control the bot.

The first step towards this was to segment out the free space from the obstacles in the image taken by the camera.

The following are the some assumptions we made that enabled us to got an extremely good estimate of free space in front of the robot

- Tiles are white in color
- All obstacles are placed on the ground (there are no overhanging obstacles)
- The ground is flat (to get an estimate of the depth of the obstacles)

The above assumptions also enable us to get an estimate of the depth of obstacles since we know the height of the camera and by finding out what tile the obstacle occupies, we can get an approximate handle it's depth.

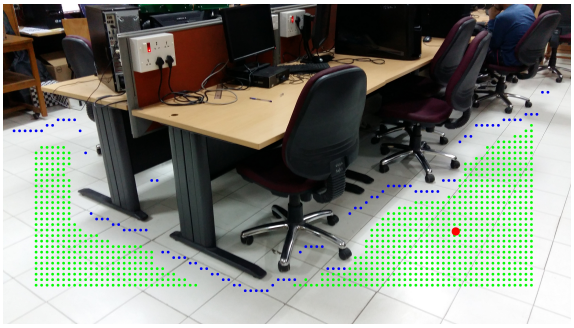
Following is the pseudocode for the algorithm we used for floor detection and boundary detection from image:

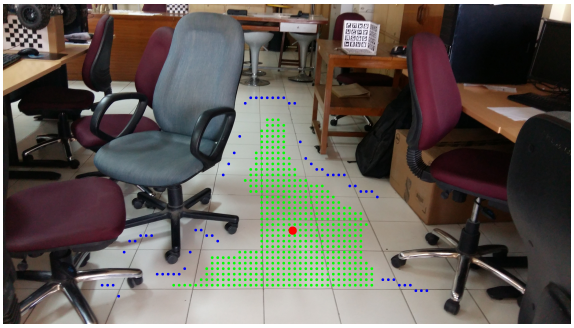
```
Floor Detection Algorithm
Input  : Floor Image
Output: Set of high, low confidence points
HighConfidenceSet, LowConfidenceSet = {}
Filter Image using median, bilateral filter
Transform image to HSV color space                          ▷ intensity invariance
for each point in image:
    if point.H > H_thres and point.S < S_thres:             ▷ floor segmentation
        Mark point as positive
    else
        Mark point as negative
Slide a window of size L over the image with stride S_L     ▷ Large window
If (num positive points) / (num total points) > L_thres
    Add points to LowConfidenceSet
Slide a window of size S over the image with stride S_S     ▷ Small window
If (num positive points) / (num total points) > S_thres
    Add points to HighConfidenceSet
```

```
Obstacle Boundary Detection Algorithm
Input  : LowConfidenceSet
Output: Set of boundary points
BoundarySet = {}
for point in LowConfidenceSet:
    Transform point to bot co-ordinate system
Define zero and infinity lines in image                     ▷ remove noise at extremities
Discard points in LowConfidenceSet below zero and beyond infinity
Sort points in lowConfidenceSet by x(ascending), then y(descending)
for x in lowConfidenceSet:
    for y in xSet:
        yDel = yCurr - yPrev                                ▷ Spacing between consecutive points
        if yDel > SpacingThres                              ▷ Large Obstacle Detected
            Add (x, yCurr) to BoundarySet
            break
    Add (x, Inf) to BoundarySet
return BoundarySet
```

ALGORITHM EXPLANATION

Image is first filtered and transformed to the HSV color space, to reduce sensitivity to intensity variation. Points are considered to be marked as floor points only if their H, S values satisfy particular conditions. Only points which occur above threshold value in a window are added to the floor region. This helps in removal of stray cluster of points. Image is first divided into small regions, and floor point count in each region is computed. This helps in reusing values for subsequent computations using dynamic programming.
In practice, good results were obtained with a small window size of 300 x 300 pixels, and a large window size of 500 x 500 pixels. The threshold used for floor point ratio in a window was 0.05.
The low confidence region gives almost all of the floor region in the image. This also includes small pockets in various places, and hence all of it is not suitable for the bot to move. However, the boundaries of this region can be used to detect the boundaries of obstacles. A larger window size ensures that small pockets are eliminated, and the region obtained has a comfortable distance from obstacle boundaries.
As explained in the previous paragraph, the set of low confidence points are used to find the obstacle boundaries. To find these boundaries, the points are first transformed to the co-ordinate system of the bot. This causes the obstacle horizon to be at y=infinity. Now, to detect boundary at given x co-ordinate, floor point with furthest y is found until a large gap is encountered in the floor (signifies obstacle). This point is considered to be in the boundary.





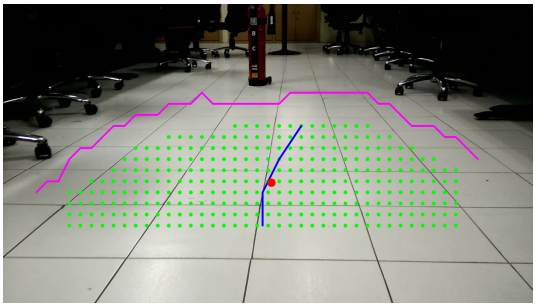
The green region in image denotes the connected component found in high confidence region of floor. The blue points are boundary points. The red point is the target point where the bot aims to move from current position.

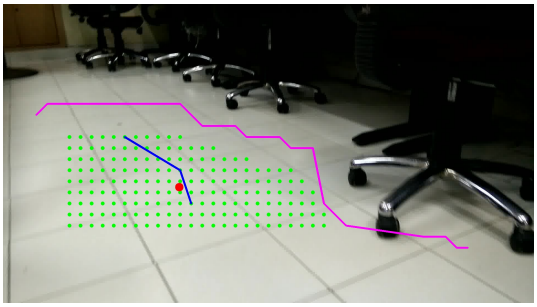
5 Trajectory Planning

At this stage, we have detected the free space in front of the robot. Now, the robot needs to move in the free area making sure that it does not collide with an obstacle. Only the high confidence region of the floor is considered for computing trajectory, as the bot will always be a safe distance from obstacles in this way. After obtaining this 'modified' free space, we estimate the trajectory as follows:

```
1  Path Finding Algorithm
   Input  : HighConfidenceSet
   Output: TargetPoint, trajectoryCurve
2  Trajectory = {}
3  Convert the points in the HighConfidenceSet to a graph representation
4  Detect connected components in this free space graph
5  Find the target connected component based on heuristic (currently size)
6  Find optimal target point in connected component (Current: Max sum of distances from
     boundary points for component)
7  Sort points in HighConfidenceSet by y(descending), then x(ascending)
8  for y in HighConfidenceSet:
9        for x in ySet:
10            Choose point with optimal x
11           (Max sum of distances from boundary points)
12            Add point to Trajectory
13 Fit a spline through the points in Trajectory to get trajectoryCurve.
14 return targetPoint, trajectoryCurve
```

A target point is computed as the point in the high confidence region which is located furthest from the boundary. The bot changes its orientation at each time step to move towards the target point. The trajectory is calculated in a similar manner by finding the point at each y co-ordinate which is located at a maximum distance from the boundary. A curve is then fitted through these points to obtain the trajectory. When the bot detects insufficient free space, it has reached close to an obstacle. The action that we currently take is to stop the bot and wait for further instructions from other modules.





The magenta lines in the above images depict the detected boundary in the image. The red point signifies the target direction that the bot will move. The green region is the connected component detected in the high confidence floor region.The blue curve denotes the ideal trajectory taken by the bot.

Although the above approach works very well for obstacle avoidance, there is one caveat. The largest connected component might not be the best place for the bot to go. This is merely a heuristic that was used for the time being, once the long range path planner is integrated, we will have a better sense of which direction to go and will have no need of this heuristic.

6 Integration with other modules

Since the long range path controller and SLAM modules written in C++, there was a need to establish inter process communication between C++ and python. We did not have to put in any effort for this part since a module that allowed communication between C++ and python had already been written earlier and we only have to make a few changes in that to allow for communication between the large scale path planner and the small scale path planner (our module). This integration can be done quickly once the long range path planner is ready.

7 Testing the local path planner and Results

After several tests and collisions, we were able to build a feedback loop which ran every 0.2 seconds (Obstacle Detection + Trajectory Estimation), the fact that the frequency of execution of this loop was so high compensated for the fact that at times, the bot might make sub optimal moves and also the translation and rotation error.

One caveat in our module is that once the bot is out of free space to move on, it stops. An alternate strategy might have been to move around the obstacle until free space is encountered again, or to backtrack and explore a better route. But this problem would vanish once the long range path planner is integrated into the system as it would give the bot an alternate path to traverse.

The testing of the above module was done next in which we set up a path with obstacles and turn laid out, the robot was able to navigate itself from the start to the goal state.

The videos having the results can seen at the link provided in the overview.

8 Future Work

8.1 Making the Obstacle Detection Unit Generic:

At this point we are exploiting the texture, color and symmetry of the floor. With a few tweaks we can make it work for any kind of floor that is uniform in nature but the next step is to handle the case when the texture and color of the floor changes in the trajectory of the robot.

8.2 Integrating the Global Path Planner:

As of now, the robot is capable of navigating in small environments by avoiding obstacles. In order to enable the bot to navigate itself over large distances, there is a need to have a module that will give a rough estimation of the trajectory to follow in order to reach the goal location. While traversing this path, the short range path planner will make sure to avoid obstacles and do localization based on wheel odometry and IMU.