Autonomous Mapping with Turtlebots using SLAM in ROS

Daniel Heideman

1 Summary

This report covers the test code written to get a turtlebot robot to autonomously map an unknown space using SLAM algorithms in Robot Operating System (ROS). Issues that were challenging or not well documented on the ROS wiki are described here. The tests include code that reduces the map to a more manageable size, finds frontier boundary points of interest, generates a cost map of known open areas' proximity to frontiers, and determines the best position in the map for maximizing collected data.

2 Big Picture

The ultimate goal of the use of turtlebots and ROS in Dr. Cortes's and Dr. Martinez's labs is to test multi-robot control systems with real robots. The current goal is to get multiple turtlebots to work together using SLAM algorithms to autonomously map out a room and create one master map from which the turtlebots can all navigate. A SLAM localization algorithm already exists for ROS, so our work will focus on combining the data into a master copy and making the turtlebots autonomously map an unknown space.

A TurtleBot 2, with Microsoft Kinect 3d sensor and Kobuki base

3 My Contributions

My work on the turtlebots is involved in determining where the robot should be to best improve its map of the surroundings. The algorithms have been tested in Matlab before implementing the algorithms in the ROS environment, as Matlab is much easier and more forgiving than ROS and C. When the algorithms work well enough to reliably determine the robot's ideal position in its map, these will be ported to ROS for further testing.

The approach I have been following for autonomous mapping is a frontier-based search. By defining frontiers as boundaries between known open and unknown regions, a map of points of interest to be mapped can be generated. I am looking into algorithms that can efficiently locate these frontiers and decide where in the map the most information gain of these frontier regions can be gained.

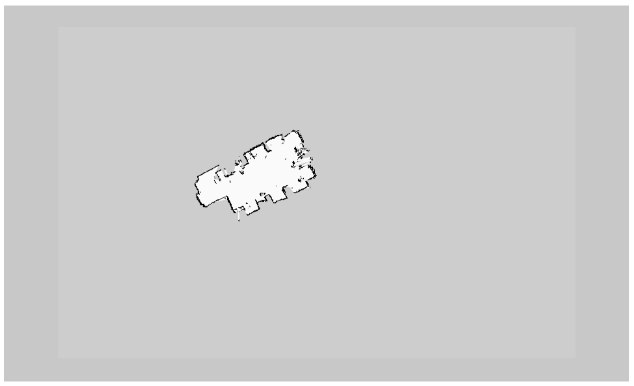
4 Methodology

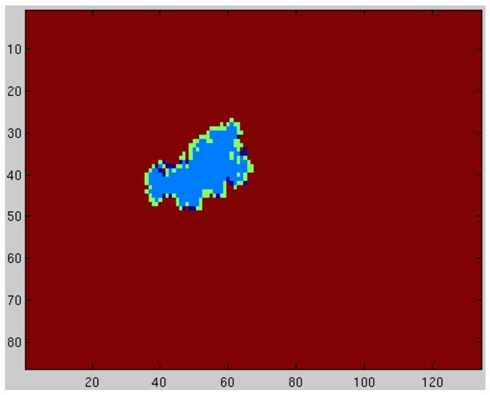
This section contains a list and high-level descriptions of test programs completed during Fall Quarter 2014.

4.1 Minimizing Map Data

The current method of locating all frontiers involves looking at every value in the map to determine if it is a frontier. The maps are initially about 500x800 values, necessitating 400,000 evaluations. To reduce the time required to locate the frontiers, the map is split into a grid of robot-sized regions of 6x6 values, thereby reducing the number of evaluations to around 11,000. If the region is above a certain threshold in unknown or known obstacle spaces, it will be registered as such. Otherwise, it will be recorded as a known open space.

This approach has the added benefit of making each value of the simplified map into a region the robot can fit into. Thus each open space can fit the robot, so the robot can be told to move there without further analysis in determining it can fit. This simplifies the later step of determining the best place from which to map the frontiers.


original map



frontier map of simplified map

4.2 Locating Frontier Regions

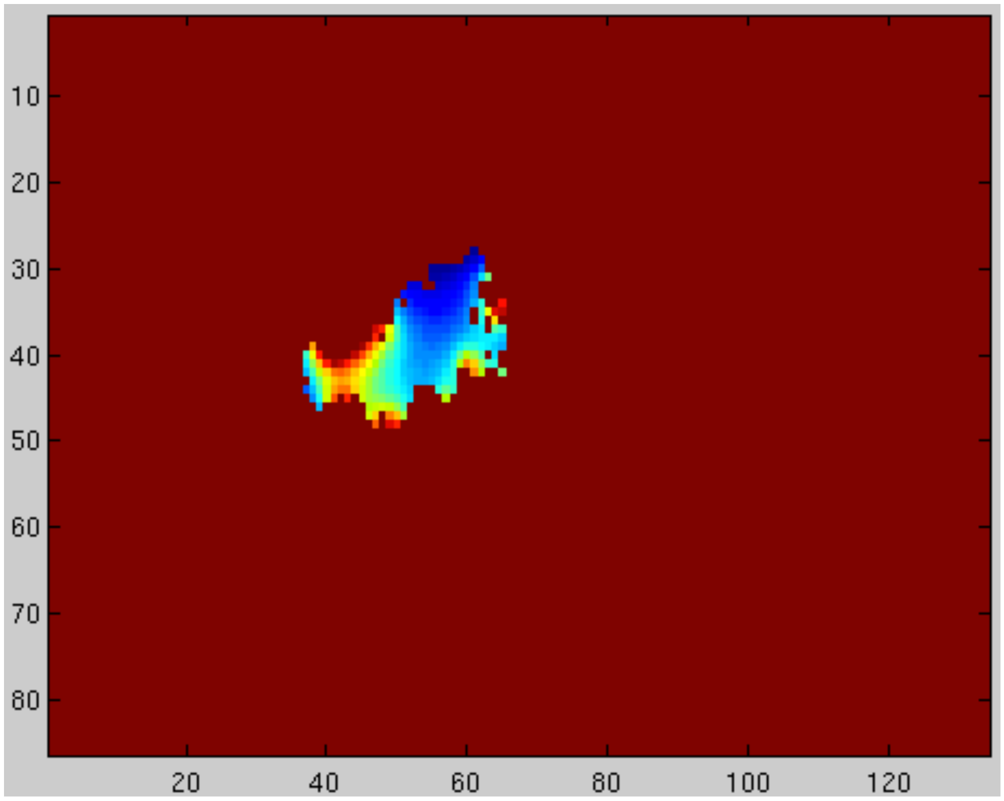
To locate frontiers, a brute-force approach is used where each value in the map is searched. If it is a known open space, the spaces above and beside it are analyzed. If one or more of them are unknown regions, then the space is labeled as a frontier.

In the map above, the known obstacles are in a light green, known open areas are in light blue, and identified frontiers are in dark blue. The surrounding red is unknown. The simplified map is useful because it significantly decreases the amount of locations that must be checked.

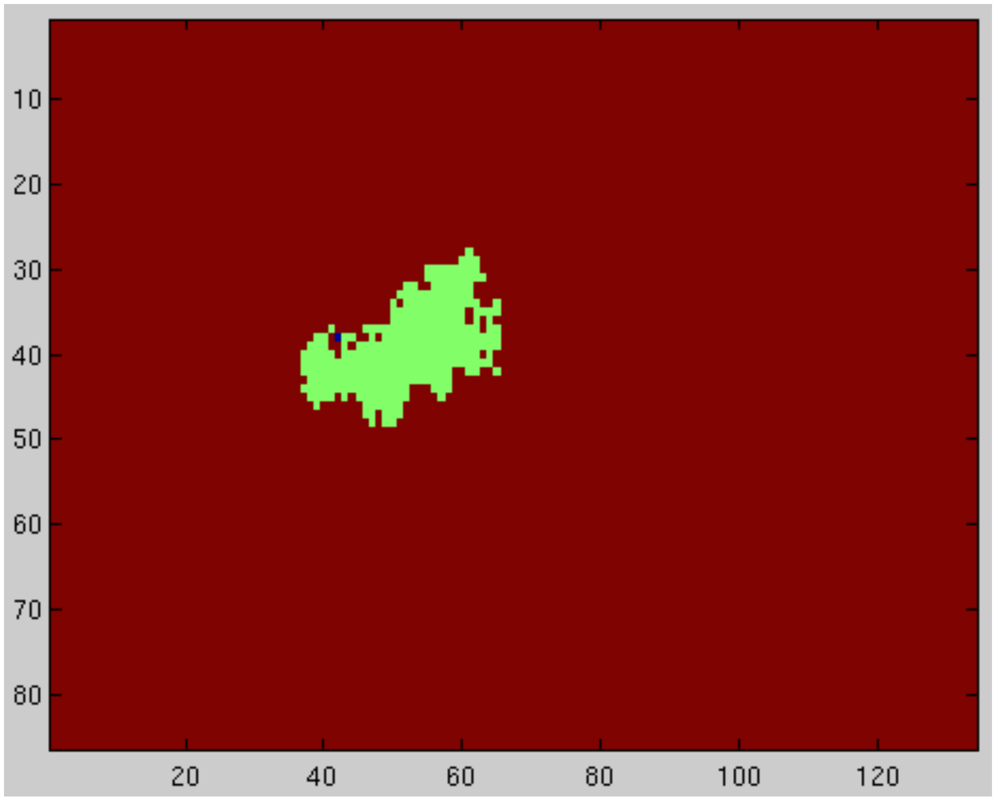
4.3 Generating Cost Map

A cost map is generated based on the inverse of the distance of each frontier from each known open space. Locations close to a lot of frontiers thus have very high costs, whereas those in the middle of a known space will be worth much less.

In the map below, more blue areas have lower cost, and more red areas, aside from the dark red unknown region, have higher costs. The frontiers and obstacles are not shown, but it can be seen that the upper right area of the map is well known with few frontiers, whereas the left side has many more frontiers and is thus more interesting and valuable to search.



cost map of the simplified map



best position map

4.4 Determining Best Position

Currently the best position is determined by locating the space with the best cost on the cost map. As all the spaces with a cost value are known open spaces, and all known open spaces are can fit a robot, this best position is known to be a space that the robot can inhabit that is close to many frontiers. However, this does not take into account whether the robot can get to the spot, as such would be handled by the path-finding algorithm built into the ROS SLAM package.

5 Tips

The reason I started off using Matlab instead of ROS is because Matlab has much better methods of displaying results and errors. It is very hard to program something like an autonomous mapping program from scratch because there is so much that can go wrong, from incorrect syntax to bad logic, and unless the results from each step are displayed it can be very hard to determine what problem has occurred.

I believe it is much easier to test these algorithms in an environment such as Matlab because the outputs are all coordinates in a map that cannot be easily visualized without being displayed in an image. As I did not wish to take the time to figure out how to generate and display an image in C, I used an environment in which I already knew how to do so. For testing algorithms, I would encourage testing first in an environment such as Matlab first, then porting it to the final application.

6 Pitfalls

Locating frontiers and generating cost maps the way I use require calculating a value for each space in the map, which can take a very long time if more complicated analysis must be performed. For instance, one method for generating a cost map that I tried involved searching each open space's line of sight in all directions for frontiers, rather than looking at distance alone. The method took far too long to run through all 400,000 data points on the original map, to the point that it never generated an answer. A cost map generated in this way would be more useful in this application, but at the moment would take too much time to be useful in a real-time autonomous mapping application. In this way the cost maps are limited by the amount of data that must be processed and the time it takes to process each space.

To partially fix this, I limited the amount of data that each method had to deal with by splitting the original map into larger regions, decreasing the resolution of the map. This had the added benefit of solving the problem of figuring out where the robot could physically occupy, as each new region was large enough to hold a turtlebot, assuming it was entirely open. Decreasing the resolution of the map may have rather bad side effects on performance, but testing with the actual turtlebots in different areas will be needed to determine if this is a problem.

7 Conclusions & Future Directions

These tests show a simple method by which a location on a map can be found that provides new data for a SLAM mapping algorithm. It should be possible to port this to ROS and the turtlebots to create a simple autonomous mapping node. However, this node would not be particularly efficient, and may tell the turtlebot to move to an area that is impossible to access. It is in these areas that further work will be needed to create a useful autonomous mapping node.

In determining the best position based on the cost map, other factors that are not currently considered must be taken into account. Considering accessibility to the region and distance from the robot would help, but the most important part would be improving the cost map.

The cost map method currently used can be improved significantly by changing it to evaluate what can be seen from each space, rather than using only distance. As discussed in the Pitfalls section, a preliminary test program took far too long to process, but simpler methods can be used to estimate the number of frontiers that would be visible from each location.

The next step after improving these algorithms would be to rewrite them in C++ as ROS nodes to test mapping in a physical, real-time environment. The ROS gmapping node and AMCL node would be run to provide the SLAM mapping and for moving the turtlebot to the coordinates the autonomous mapping node determines are best, respectively. The autonomous mapping node would have to watch the gmapping and AMCL nodes to determine when to perform each task, such as waiting until the turtlebot is in position before spinning in place to map the area or waiting until the map is sufficiently updated before moving to the next location.

To make a proper autonomous SLAM program, the robot must also revisit known areas to localize itself occasionally. While similar to a line-of-sight cost map method, it would be looking for proximity to known obstacles and would be completely separate from the mapping methods. As SLAM cannot both localize and map well at the same time, they must take turns. A value of localization certainty would trigger the switches between modes. However, without being given certainties of whether an area is open or occupied, I currently have no idea how to approach this latter part.

Our final goal involves the implementation of mostly-autonomous turtlebots using the master only for communication, not for receiving commands. With this setup, the turtlebots would individually run their own autonomous mapping nodes, but when they wanted to know their position they might ask the master, or consult the other turtlebots. The master would also tell each turtlebot what areas it is responsible for mapping and combine the maps into one global map. A deployment algorithm may be used to split up the global map between turtlebots, with each turtlebot performing only minimal overlap between regions while mapping.

8 Referenced Code

8.1 simplifymap.m

```
function [outputmap] = simplifymap(inputmap)
%
% Make the map into a smaller map by grouping values into regions the 'bot can fit
in
%
% Size of regions: 0.30m x 0.30m
% Resolution of original map: 0.05m
%

occupiedthreshold = .20;  %5% occupied => region is occupied
unknownthreshold = .60;   %60% unknown => region is unknown


[height, width] = size(inputmap);
resolution = 0.05;
regionsize = 0.30;

regionlength = regionsize/resolution;
ouputmap = zeros(ceil(width/regionlength),ceil(height/regionlength));

for xx=1:ceil(width/regionlength)
  for yy=1:ceil(height/regionlength)
    open=0;
    occupied=0;
    unknown=0;
    total=0;
    for xi=1:regionlength+1
      for yi=1:regionlength+1
        x=int16((xx-1)*regionlength+xi);
        y=int16((yy-1)*regionlength+yi);
        if (x<=width && y<=height)
          total=total+1;
          if inputmap(y,x)==205  %unknown
            unknown=unknown+1;
          end
          if inputmap(y,x)==254  %open
            open=open+1;
          end
          if inputmap(y,x)==0    %occupied
            occupied=occupied+1;
          end
        end
      end
    end
    percentopen=open/total;
    percentoccupied=occupied/total;
    percentunknown=unknown/total;

    if percentoccupied>=occupiedthreshold
      outputmap(yy,xx)=0;
    elseif percentunknown>=unknownthreshold
      outputmap(yy,xx)=205;
    else
      outputmap(yy,xx)=254;
    end

  end
end
```

8.2 locatefrontiers.m

```
function frontiers=locatefrontiers(imgstrname)
% Locates frontiers in img, displays img with frontier points denoted.
%Returns coordinates of every frontier as (x,y)
%
[img,map]=imread(imgstrname);
img=simplifymap(img);
newimg = zeros(size(img));
[height, width] = size(img);
frontiers = zeros(1,2);
%knownboundaries = zeros(1,2);
index=1;
%kbindex=1;

for x=2:(width-1)
   for y=2:(height-1)
      if img(y,x) == 254
         if checkfrontier2(img,y,x) == 1
            newimg(y,x)= 0;
            frontiers(index,:) = [x,y];
            index = index+1;
         else
            newimg(y,x)= 16;
         end
      else
         if img(y,x) == 0
            newimg(y,x)=32;
         else
            newimg(y,x)= 64;
         end
      end
   end
end

newimg(1,:)=64;
newimg(height,:)=64;
newimg(:,1)=64;
newimg(:,width)=64;
%colormap(map);
image(newimg);
Number_of_frontier_pts = index-1

end
```

```
function isfrontier = checkfrontier(img,x,y)
isfrontier = 0;
for i=-1:1
    for j=-1:1
        if img(x+i,y+j) == 205
            isfrontier = 1;
        end
    end
end

end

function isfrontier = checkfrontier2(img,x,y)
isfrontier = 0;
for i=-1:1
    for j=-1:1
        if (i*j==0)
            if img(x+i,y+j) == 205
                isfrontier = 1;
            end
        end
    end
end

end
```

8.3 costmap.m

```
function newimg=costmap(frontiers,imgstrname)
% Builds cost map for each open known space,  Σ 1/dist to all frontiers
% Returns cost map image
%
[img,map]=imread(imgstrname);
img=simplifymap(img);
newimg = zeros(size(img));
[height, width] = size(img);
index=1;

for x=1:(width)
   for y=1:(height)
      if img(y,x) == 254
         newimg(y,x)= findcost(x,y,frontiers);
      else
         newimg(y,x)=-1;
      end
   end
end
maxcost=max(max(newimg));
mincost=min(min(newimg(newimg~=-1)));
newimg(newimg~=-1)=100.*(newimg(newimg~=-1)-mincost)./(maxcost-mincost);
newimg(newimg==-1)=255;

%colormap(map);
image(int16(newimg));

end

function cost = findcost(x,y,frontiers)
numfrontiervalues = size(frontiers,1);
xs=x.*ones(numfrontiervalues,1);
ys=y.*ones(numfrontiervalues,1);
cost = sum(1./sqrt((frontiers(:,1)-xs).^2+(frontiers(:,2)-ys).^2+1));
end
```

8.4 testfrontiers.m

```
close ALL
img = 'theoffice.pgm';
imshow(imread(img),[0,255]);

figure
frontiers=locatefrontiers(img);
%figure
%plot(frontiers(:,1),frontiers(:,2).*-1,'o')
%axis([0,800,-512,0])

figure
costimg=costmap(frontiers,img);

figure
findbestpos(costimg);


%figure
%shortrangecostmap(frontiers,20,img);

%figure
%knowncostmap(100,'theoffice.pgm');

%image(I);
```

8.4 Reference Material

“Comparative Study of Algorithms for Fast Frontier based Area Exploration and Slam for Mobile Robots”