Abstract

The main goal of this lab was to write a motion planner that would traverse a path through the cones as described in the lab. A secondary goal was to be able to monitor that progress.

To complete the first goal, a new program, the executive, was written and the navigator was edited. The two programs were to work in conjunction to be able to determine the path the robot was to traverse and to determine the point which the robot would aim to follow. This breaks down into the path planning and pure-pursuit problems. Path planning was addressed by discretizing the theoretical field and running A\* on a variety of points. Pure pursuit was addressed by determining the goal point in the executive and then the twist commands in the navigator.

The secondary goal was rather simple. First, the completely path was determined in the executive, so a path message was advertised that the GUI would listen to. With that, the GUI would draw the path. The GUI would also listen the goal point from the executive and draw a line from the robot to the goal point. In retrospect, a better form of display would be a marker, such as an X, only at the goal point.

The only real data that can be gathered is the determined path. The pure pursuit is a mess.

Code Implementation

My executive (ros\_executer.cpp) is broken down into three pieces.

The first is a Node class. This acts as the point in the discretization on my map. It holds a “vector<double> location” that holds the global position of the point in meters. It holds a weight, which was used purely to determine whether or not my planner should use that point or not in the path. If the point was theoretically on a cone, the points weight was set to INFINITY (a math.h value). Otherwise, it was 0.0. It holds a g and an f, used to hold a double that would then represent the g and f values for that point during A\*. It holds a vector<int> myIndex which holds the x and y indexes for that node. A thing to note is that my map (addressed later) holds *rows* of vector<double>, so accessing the xth and yth node is done by *nodes[y][x]*, not *nodes[x][y]*. It holds a vector<int> parentIndex, which is assigned during A\* for tracking back the optimal path.

The second is a Map class. This is where all of the path creation is done. It holds a vector< vector<Node> > nodes which are all the nodes that represent the space to move in. it holds a vector<Node\*> path which holds a *backwards* ordered set of Node pointers representing the path after A\* is done on the map for two nodes. It holds ints depth, width, resolution, and nodeCount which determine the dimensions in meters, points per meter, and total number of nodes for the map. The other variables are used in A\* only and are held as class member variables in order to reduce time by eliminating their reallocation for successive runs through A\*.

My code has a ton of comments, so look for them if there is any confusion.

The constructor sets up the class by setting all the nodes in *nodes*. The map is a representation of the robotics lab floor, 4 meters width and 8 meters long, with the robot at (0, 0) on the center left of the grid. X is along the long side and y along the width. Each node has its location and and indexes set accordingly. Indexes start from the top left, were y increases down and x increases right. Then, the nodes that are on cones are set to have a weight of infinity and the path is reset.

setHeavyNodes() sets the nodes in the class to have a weight of infinity. They represent the cones.

printMap() prints the map out, both it’s global location and it’s (x, y) indexes.

printPath() prints the current path set. As the traversal is backwards, it is stored backwards, so it is printed backwards.

printNeighbors() was a helper function I made for debugging. It printed the vector<Node\*> neighbors of the class to see if the right neighbors were gotten.

aStar() runs the A\* algorithm from the *start* node to the *end* node passed. This is the algorithm. For reference, I used the one on wikipedia:

1. vector<Node\*> closedSet is reset
2. vector<Node\*> openSet is rest
3. Node\* start is pushed to openSet, as the first variable to explore
4. Reset the g and f values of *nodes* to infinity
5. Set the g and f value of the start to 0
6. While opeenSet is not empty
   1. Set the Node\* *current* to the one with the lowest f value in the openSet
   2. If the *current* node is the *end* node, you’re done, and run setPath to set the found path. Then return true to signify it was a success
   3. Remove *current* from the openSet
   4. Put *current* in the closedSet
   5. Set the *neighbors* vector to that of *currents*  neighbors
   6. For all of the neighbors of current
      1. If the neighbor is in the closed set, move on to the next neighbor, or if it has a weight of infinity, move on to the next neighbor
      2. Set the tentative g value to be the distance between the current node and neighbor, plus the g value of the current node
      3. If the neighbor is not in the openset, put it, or if the tentative g value that has been found is larger than it’s past one, move on
      4. Set the neighbor’s members, so the parent index to be the current node’s, the g value to be the tentative g value, and the f value to be its (now new) g value and the heuristic value of the neighbor for that.

setCurrentToLowestFInOpenSet() sets the Node\* *current* to the Node\* with the lowest value in the openSet.

removeNodeFromOpenSet() removes the Node\* passed from the openSet.

setPath() takes two Node\* and follows the path backwards from the currentNode to the startNode and records them in the vector<Node\*> path (member).

setNodeNeighbors() sets *neighbors* to be the neighbors of the passed node. It is used in A\*. It handles edge cases. Details in comments.

isInClosedSet() returns a bool on whether the passed Node\* is in the closedSet. True for when it is, false when not.

isNotInOpenSet() returns a bool on whether the passed Node\* is in the openSet. False for when it is, true when not.

distBetweenTwoNodes() returns the Euclidian distance between two passed Node\*.

h() is my heuristic function, which just has an embedded distBetweenTwoNodes(). Created for readability.

That’s my Map class. Check comments for details.

The third part is my Path class. It has a Map map that can be used to find paths from one point to the other. It has a vector<Node\*> waypoints that holds all the nodes that will connect the masterPath together. It has a vector<Node\*> masterPath which is every node along the traversing path that is touched. It has a vector<double> position that holds the current mu of the robot. It has a vector<double> sigma that holds the diagonals of the current sigma. It has a vector<double> currentGoal that holds the current goal of the robot in its local frame.

The constructor sets the waypoints that mapPath() will then connect together. It resets the position, sigma, masterPath,, and currenGoal. It sets the subscribers and publishers.

printPath() prints out the masterPath, which is saved in order.

printNodeIndex() prints the indexes of the Node\* passed.

printWaypointsIndex() prints the indexes of all the waypoints.

mapPath() sets the vector<Node\*> masterPath to a connected set of path between each successive waypoints using A\*. A\* is run on each successive set of waypoints, and that vector<Node\*> path of Map is pushed onto masterPath.

handle\_pose() saves the PoseWithCovariance message from the localizer to position and sigma.

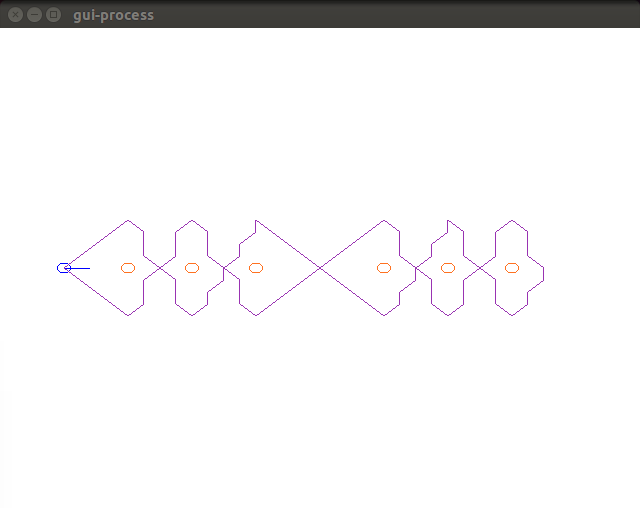
Publish\_pose() is meant to publish the currentGoal, the first 4 steps of pure pursuit, to the navigator. It seems to fail horribly. It’s meant to set the goal point to the closest node on the masterPath and a little further along the path.

The main first creates a Path, then sets the masterPath for that Path, then continuously publishes the masterPath to the GUI and the currentGoal to the navigator.

Data Analysis

This is my path.

Boom.



Here it is with everything running. The localizer just went straight off the field.

