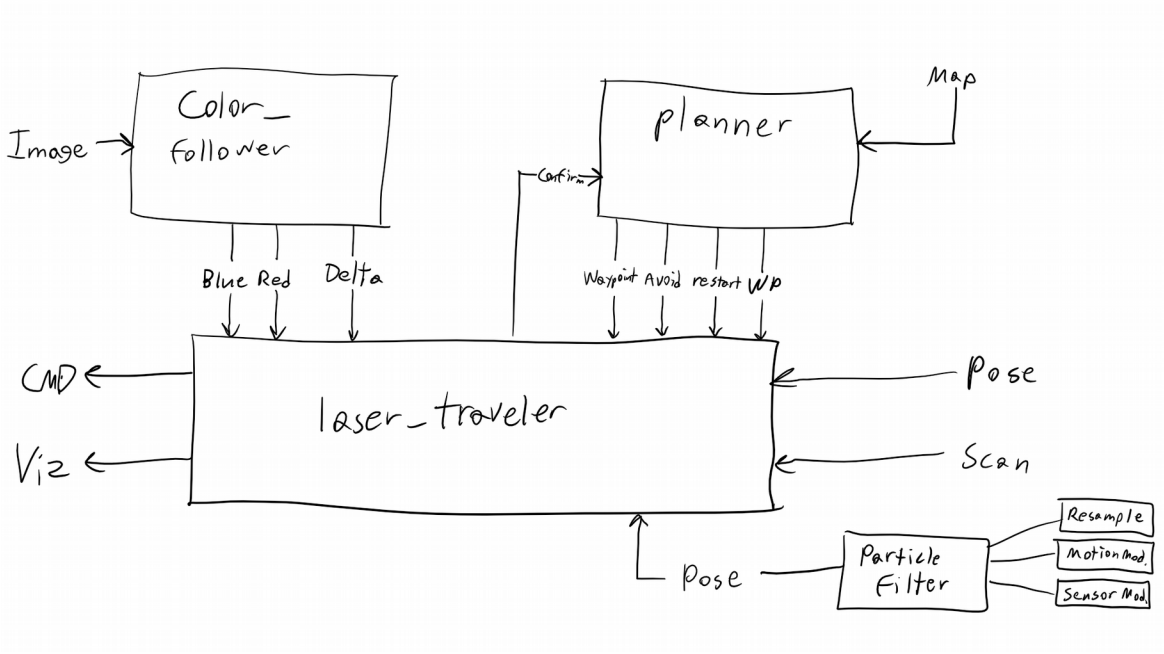
Team DDK – Daniel Luncasu-Rolea, Drew Clark, Kyle Phan
EE P 545 – The Self Driving Car: Introduction To Air For Mobile Robots
Final Project: Localization with Particle Filters
Due: December 10 2018

1. System Model



The system consists of four main segments:

- The planner, which parses the waypoints and then assembles a path for the robot to take, assigning the path to the robot as the robot moves along the path.
- The particle filter from the previous lab, which provides localization info (a pose).
- The "laser_traveler," which is a significantly expanded version of the laser_wanderer from lab1, serving the purpose of integrating sensor data and then deciding how to move the robot
- The color sensing system, which observes the camera feed and advises the robot which direction to move in order to avoid red squares and visit blue squares

The data flow consists of sensor information being funneled into the laser_traveler, which serves as an information processing core and decides when to listen to the color sensing system, when to follow the path, when the robot is about to hit a wall, and which waypoint to follow next. The rest of the modules take on a helper role, and as such the robot does not always pay attention to the info provided by a particular model.

a. **planner.py**

Input Topics:

CONFIRM_TOPIC – laser_traveler.py will specify which waypoint it is currently requesting
MAP_TOPIC – Used to set up the map and pathfinding

Output Topics:

WAYPOINT_TOPIC – The current waypoint, given to laser_traveler.py
WAYPOINTS_TOPIC, START_TOPIC, AVOID_TOPIC –Waypoints, used for visualization
RESTART_TOPIC – A topic used to specify if laser_traveler.py should restart the plan
WP_TOPIC – A topic used to determine whether the waypoint is a primary waypoint or a secondary (pathfinding) waypoint

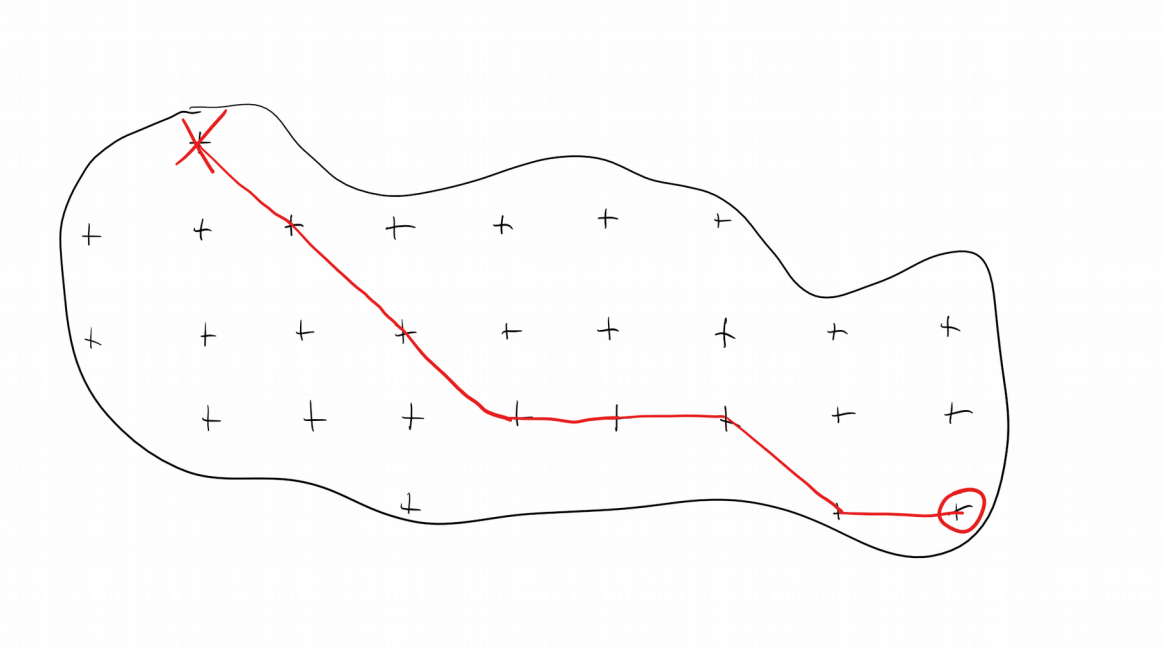
Planner consists of two main parts, a pathfinder which runs in the beginning, and a waypoint assigner which runs constantly in order to guide the car along the path. Rather than using PID, we decided that the most computationally efficient way to guide the car along a line without missing waypoints is to give the car a wide margin of error, such that we can be sure the car will stay within the area the entire time. We added a parameter which is a "large" waypoint radius, which tells the car the distance it needs to approach a non-essential waypoint before switching to the next. This creates a wide tunnel where the car only needs to pass directly through the absolutely essential waypoints.

The final plan consists of an array of waypoints, a few of which are the "primary" waypoints specified by the lab spec, the rest of which are "secondary" waypoints which the car need not pass through perfectly, but merely follow as a general guide.

The planner also serves the auxilary function of parsing the waypoints (and thus making them available to the laser_traveler as well) and also detecting whether or not the path needs to be restarted, based on the initialpose feature of Rviz.

The pathfinder operates at the beginning, first dividing the map into a grid of node classes, which are connected to their 8 adjacent nodes, and then running an A* algorithm to find a path from start to finish. The A* algorithm uses a heuristic based on 3 components:

- Distance from the next waypoint
- Closeness to the nearest red waypoint
- Difference in theta from the last two waypoints (preferring straight paths)



This combination of heuristics minimizes turning while also allowing the robot to navigate past walls thicker than the node spacing. We did not use a Halton sequence or pseudorandom node arrangement because we wanted a deterministic setup (keeping the parameters the same) and because our grid spacing approached 0.5m due to the high speed of the A* algorithm. We also included several methods of optimizing link creation, such as randomizing the order of the node array and limiting the amount of connections to 8.

The second part of the planner assigns nodes along a path based on an index **requested by the MPC algorithm**. We gave laser_traveler.py itself full control over the decision whether or not the car passed a waypoint based on 1) the fact that the laser_traveler integrates data from all sensors, including particle filtering, laser scans, and color information, and 2) because we want our latency to be as low as possible.

b. laser_traveler.py

Input Topics:

BLUE_TOPIC – From color_follower.py, determines if robot sees a blue WP
RED_TOPIC – From color_follower.py, determines if robot sees a red WP
DELTA_TOPIC – From color_follower.py, determines delta decided by camera processing
SCAN_TOPIC – Used to receive laser scans
AVOID_TOPIC – From planner.py, list containing all waypoints to avoid
WAYPOINT_TOPIC – From planner.py, current waypoint to follow
RESTART_TOPIC– From planner.py, determines whether to restart the plan
WP_TOPIC– From planner.py, determines whether this is a primary waypoint (pay close attention) or a secondary waypoint (go roughly towards it)
POSE_TOPIC – Inferred pose from ParticleFilter

Output Topics:

VIZ_TOPIC – Used for visualization
CONFIRM_TOPIC – Used to request a specific waypoint index, increased automatically when the robot arrives at a waypoint
CMD_TOPIC – Used to send commands to the motor controller

Laser_traveler was built on our original laser_wanderer module from lab1 using the MPC technique to create trajectory rollouts and assign costs to each rollout in order to determine the next control.

The main difference in laser_traveler is that it subscribes to our planner and color follower. Instead of wandering around aimlessly, laser_traveler gets the next goal pose from the planner and assigns a cost to all steering angle proportional to the difference between the pose angle and final heading of the pose corresponding to each steering angle. As with laser_wanderer, running into an object gets assigned maximum cost, and if there are no viable paths, the robot will go into reverse and seek out the most open path it can find.
However, the algorithm now also adds a cost to areas close to red waypoints, as instructed by the list published by the planner. This way, the robot avoids red waypoints even while trying to navigate toward the next step in the path.

Once a goal pose is reached, the laser_traveler publishes a confirmation to tell the planner to send the next goal pose. Goal conditions are determined by the following state process:

```
if distance_to_waypoint < color_seeking_distance:
     if color_is_visible:
          // Follow the color until it's invisible
     else
          // Go toward the center till minimum radius
```

Laser_traveler is the module that publishes our controls. It uses the controls determined by the former laser_wanderer routine unless a color is visible and a primary waypoint is within seeking range according to the ParticleFilter's inferred pose. The module subscribes to the blue_topic, red_topic, and delta_topic from the color_follower , and uses the delta controls given by color_follower when the proper conditions are met, as shown in the process above.

We tested a redundant feature to make the robot avoid red squares via color sensing, however the pathfinder already avoids it, and the MPC already applies increased costs to areas near red waypoints. So this feature is largely redundant and marginalized in terms of use.

c. **color_follower.py**

Input Topics:

IMAGE_TOPIC – Subscribes to camera images and processes data

Output Topics:

BLUE_TOPIC – Specifies if a blue waypoint is seen
RED_TOPIC – Specifies if a red waypoint is seen
DELTA_TOPIC – Used to specify the steering angle the robot should take based on images

Color_follower subscribes to the images published by the RGB-D camera to look for red or blue squares and provide an appropriate steering angle to either drive over the blue squares or avoid the red squares.

When an image is received it converts the message to OpenCV, reduced the image size by 0.5 and converts the image to HSV. Using the HSV image, thresholds are compared to the image to look for the colors blue and red to create a binary image for each color. If either color exits, the contours of the binary image are found and sorted by contour area. Starting with the largest area, the number of vertices are calculated and if there are at least three vertices and a minimum area is met the contour is determined to be a box and the center of the box is calculated and passed back and none of the other (smaller area) contours are looked at.

If a blue box exists, the goal is to center the blue box in the x direction in order to ensure the car drives over the box. The left and right most extreme points of the box are calculated and if the center of the image is within these points then a delta of 0 is passed back to ensure the car continues going straight. If not, a steering angle proportional to the pixel error (from center of box to center of image) is passed back to turn the car toward the box.

If a red box exists, the goal is to avoid the red box. In this case, if the center of the image is within the left or right most extreme points (plus a buffer) then a steering angle is given to avoid the red box, otherwise it is directed to go straight. If both a blue and red box exits, the one closest in the image (largest value of y for each center point) is used.

Finally, the delta is scaled such that the farther away the box (lower y value for center point), the smaller the delta. This is to make the car drive smoother and avoid hard turns when a box is detected with enough time.

d. ParticleFilter.py

Input Topics:

This class receives data from the localization classes from the previous lab

Output Topics:

POSE_TOPIC – Outputs inferred pose for robot localization

We do not use any of the other topics published by this class

ParticleFilter was inherited from the previous lab. There were no intentional changes made to it whatsoever as the existing behavior proved sufficient for our purposes. However, we did have a bug with our motion model which we inherited from the previous lab, where our velocity values were integers and therefore being cropped to 0. This caused our robot to always believe it is stationary, making localization unreliable.

Fixing this made it apparent that localization was not a large concern for our robot, as color_follower.py (see above) would be used for precision tasks, and planner.py (see above) would give general direction.

2. Project Design

For this project it was decided to use and modify our existing MPC and Particle Filter modules from labs 1 and 2. On top of these algorithms, a planner was necessary to generate and publish a plan to navigate the map, and a camera module was needed to utilize the RGB-camera to ensure that we crossed the blue squares and avoided red squares along the way.

The particle filter was unmodified from lab2 besides a minor bugfix explained above. This algorithm provides us with localization capability.

The planner generates a list of intermediate waypoints between the main waypoints (blue squares) and publishes this information so that our car is always traveling towards goal pose.

The MPC was modified from laser_wanderer to adjust the cost based on poses from the planner. High costs were still assigned to running into objects, but the baseline was changed from zero delta to prefer deltas that approach the goal pose. This was all implemented into laser_traveler, which is the main module that publishes the controls.

When within a certain radius of a main waypoint (blue-square) and a blue square is detected in our image processing our color_follower takes over for the MPC and generates appropriate deltas to make the blue square in the center of the image.

3. Demo

Time Slot: 8:10pm-8:22pm.

Captain: Daniel Luncasu-Rolea.