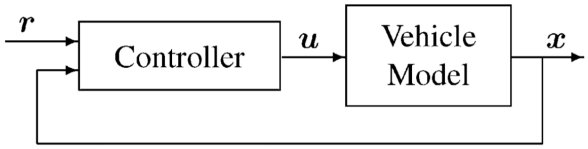
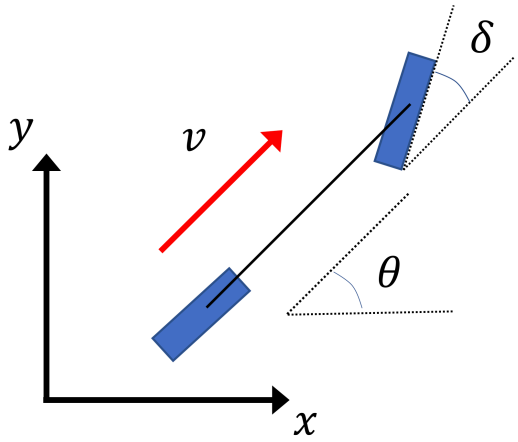
Ben Groelke
Final Project Report
ITCS – 8152

Background Info/Robot Model

For my project, I implemented the vehicle motion planning algorithm CL-RRT. The main idea behind CL-RRT is that the samples are generated in the controller input space, $\boldsymbol{r}$, of a lower level controller that is closed loop stable.



CL-RRT is especially beneficial as a motion planning algorithm for vehicles with complex, and unstable dynamics, as a stabilizing control signal automatically causes the vehicle to track the reference. The kinematic single track car model was used as the robot model, in addition to controller dynamics. A diagram for the kinematic single track model is shown below.



The state space for the model used in this project, which includes controller dynamics are described by equations.

$$\dot{x} = v\cos\theta$$
$$\dot{y} = v\sin\theta$$
$$\dot{\delta} = \frac{v}{L}\tan\delta$$
$$\dot{v} = a$$
$$\dot{a} = \frac{1}{T_d}(a_c - a)$$

Each node in the tree contains the vehicles configuration and controller input at that configuration. As stated before, sampled in the controller input space $\boldsymbol{r} = [x \ y \ v_{cmd}]^T$. However, $v_{cmd}$is predetermined as a function of distance, having a ramp up, coast, and ramp down phase where the ramp down phase brings the vehicle to a complete stop once arriving at a sample, $\boldsymbol{s}$. This is shown in the figure below. v_{max} was set to 12 m/s in this project.

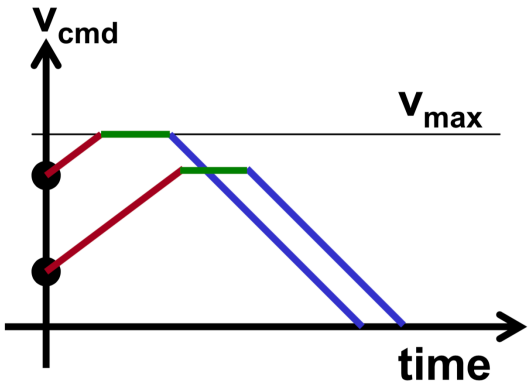
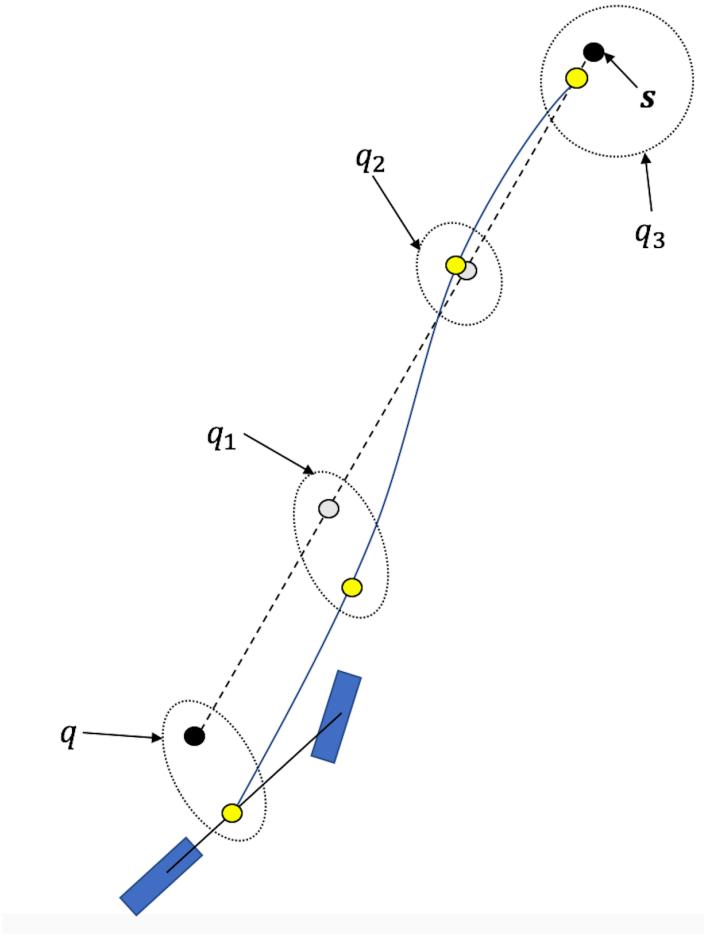


Figure 1. source: found here

Since v_{cmd} is a predetermined function of position, samples can just be composed of x and y position, $\boldsymbol{s} = [s_x \ \ s_y]^T$. In the figure below, the dotted ovals, $q_1 \ldots q_3$ represent nodes. The yellow dots show the x and y position of the vehicle and the black and grey dots show the commanded x and y position of the vehicle at that configuration. The scenario depicts the vehicle performing a successful connecting from a configuration from existing node q to q_3. Nodes q_1 and q_2 are added as intermediate nodes



Implementing CL-RRT

Algorithm 1 was used to build the tree, which is similar to the `Connect` function for RRT, which was covered in the lecture on RRT.

Algorithm 1 Expand_tree()

1: Take a sample $\boldsymbol{s}$ for input to controller.
2: Sort the nodes in the tree using heuristics.
3: **for** each node q in the tree, in the sorted order **do**
4: Form a reference command to the controller, by connecting the controller input at q and the sample $\boldsymbol{s}$.
5: Use the reference command and propagate from q until vehicle stops. Obtain a trajectory $\boldsymbol{x}(t)$, $t \in [t_1, t_2]$.
6: Add intermediate nodes q_i on the propagated trajectory.
7: **if** $\boldsymbol{x}(t) \in \mathcal{X}_{\text{free}}(t)$, $\forall t \in [t_1, t_2]$ **then**
8: Add sample and intermediate nodes to tree. Break.
9: **else if** all intermediate nodes are feasible **then**
10: Add intermediate nodes to tree and mark them unsafe. Break.
11: **end if**
12: **end for**
13: **for** each newly added node q **do**
14: Form a reference command to the controller, by connecting the controller input at q and the goal location.
15: Use the reference command and propagate to obtain trajectory $\boldsymbol{x}(t)$, $t \in [t_3, t_4]$.
16: **if** $\boldsymbol{x}(t) \in \mathcal{X}_{\text{free}}(t)$, $\forall t \in [t_3, t_4]$ **then**
17: Add the goal node to tree.
18: Set cost of the propagated trajectory as an upper bound C_{UB} of cost-to-go at q.
19: **end if**
20: **end for**

Figure 2. Source: https://ieeexplore.ieee.org/document/5175292/

Samples $\boldsymbol{s}$ generated using a reference position, (x_0, y_0) and heading, θ_0. where η_r and η_θ are random variables with standard Gaussian distributions.

$$\begin{bmatrix} s_x \\ s_y \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \end{bmatrix} + r \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \text{ with } \begin{cases} r = \sigma_r |n_r| + r_0 \\ \theta = \sigma_\theta n_\theta + \theta_0 \end{cases}$$

After a sample was generated, nodes were sorted via Dubins distance in ascending order. Connections between the nodes and the sample were then attempted in the sorted order. If a connection was successful, three nodes were added to the tree; one node was the sample and two nodes along the trajectory from the start node to the sample were added to the tree. Each node in the tree contained the states $\boldsymbol{x}$ and the input, $\boldsymbol{r}$. After a successful connection between a node and sample was made, connections were attempted between the added nodes and the goal configuration, which was essentially an x and y position. A successful connection was one

that was collision free. Collision was detected by representing the vehicle as a rectangle, and representing the environment with 20cmX20cm cells, each belonging to either obstacles or free space. If the rectangle that represented the vehicle intersected a cell that was an obstacle, it was considered that a collision had occurred. The commanded velocity was determined by a function of estimated distance to go, implementing this was one of the more challenging aspects of this project. Steering angle geometries was another challenging aspect. In addition to finding complete paths to the goal configuration, costs-to-go were calculated for each tail of parent nodes. These costs-to-go were representative if distance, so the smallest cost-to-go corresponds to the shortest distance. Shortest control input ($\boldsymbol{r}$) paths are shown in the results as a bold black line. Samples are indicated with stars, and state trajectories are indicated with circles. In the results shown nodes added earlier to the tree are represented by cooler colors. Here are some preliminary results:

Code Written

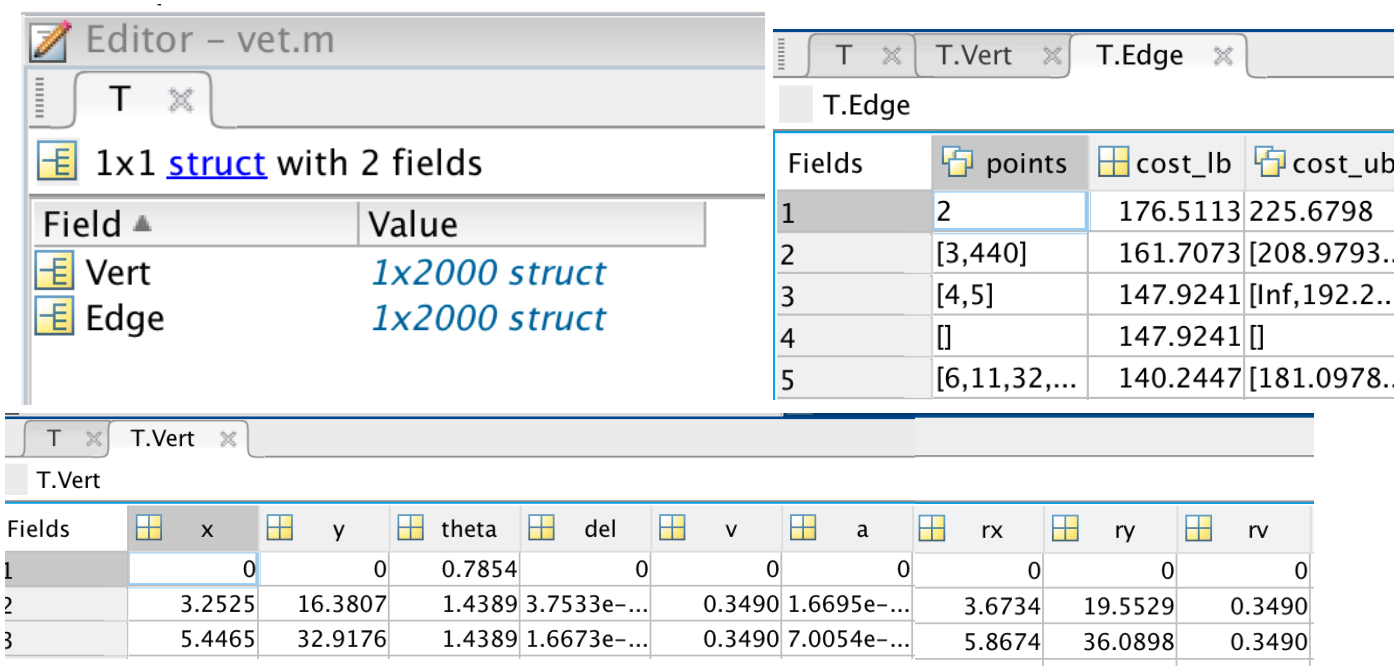
Here are the functions I wrote in Matlab to implement CL-RRT

- `Main` – The main function which the user runs. The user sets the initial configuration, which gets initialized as to root node, and also the sampling distribution parameters
  - `createDriveMap*` – The drive map is a discretized grid of 20cmX20cm cells that are either represented with a 1 if it contains an obstacle or a 0 if it is free. I used * because for the results shown I used `createDriveMapU` and `createDriveMapMaze`. `createDriveMapU` places a 'u' shaped obstacle, `createDriveMapMaze` creates a simple maze-like obstacle.
  - `buildRRT` – initializes robot to the user set configuration. Initializes loop to generate samples and extend the tree
    - `randomSample` – Generates a random sample with a Gaussian distribution. User set parameters σ_r and σ_θ, determine the radial and angular distribution from some reference point (as in anchor point, not $\boldsymbol{r}$), (x, y, θ).
    - `extendRRT` – Extends tree all the way to sample. If no collision occurs, the node is added to the tree, as well as a certain number (1-3) intermediate nodes. The intermediate nodes are needed for tree growth as samples avoid connection with leaf nodes, since the vehicle comes to a complete stop at all leaf nodes. `extendRRT` attempts to connect to the closest 20 nodes, if no collision free path is found, the sample is thrown out.
      - `sortClosest` – This function is used to sort the nodes of the tree based on minimum Dubins path from the sample, in ascending order.
      - `formRefNew` – Forms the reference path (a straight line) connecting the controller input, $\boldsymbol{r}$, of some node to the sample $\boldsymbol{s}$
      - `stateTrajectoryNew` – Propagates the states via the vehicle model, using the reference path drawn from $\boldsymbol{r}$ to $\boldsymbol{s}$ to calculated δ_c. a_c was determined by v_{cmd} which, as previously state, was a function of distance.
      - `noCollision` – Checks state trajectory for collision. State trajectory is given for the center point on the rear axle of the vehicle. `noCollision` adds the width

and length of the vehicle by boxing in the vehicle with 4 points. The 4 points are then checked for if they enter obstacle space of the drivability map.

- `vet` – Vets the tree by finding the lowest upper-bound cost and committing to that path. Plots the path for the user to examine the correctness of path. Used to generate results shown at the end of report

My project was entirely written in Matlab. To store the tree, I used Matlab's struct data type which, I believe, is essentially Matlab's version of a classes in traditional object oriented programming. The entire tree was stored in variable 'T', All of the vertices and edges of the tree were stored in 'T.Vert' and 'T.Edge', respectively. The i^{th} vertex and edge was stored in 'T.Vert(i)' and 'T.Edge(i)', respectively. This data structure is shown in the figure below



In the figure above, the numbers in the "Fields" represent the i^{th} node. The root node is always located in field 1. Nine values were stored in each vertex they are listed here in the order they our shown at the bottom of the figure above, from left to right: 1) x position, 2) y position, 3) heading (θ), 4) steering angle (δ), 5) longitudinal velocity, 6) longitudinal acceleration, 7) the commanded x position, 8) the commanded y position, and 9) the commanded velocity. So, for example, if I wanted to access the steering angle of the 3^{rd} node, I know it would be stored in 'T.Vert(3).del'. For the edges (shown in the upper right of the figure above), the "Field" number corresponds to the vertex at which the tail of the edge connects to. The "points" value stores the index of the vertex of the child node which the edge connects to. Nodes with multiple edges will have multiple values in "points" . "cost_lb" stands for the lower bound cost (cost being distance to go) and is calculated by the Euclidean distance from T.Vert(i).x and T.Vert(i).y to the goal configuration (heading was not considered as part of the goal configuration, but

could be enforced via batch sampling, however the results from this project do not include batch sampling). The upper bound cost is initially set to infinity for all nodes while building the tree when a complete connection has not been made between the root and goal node. Once a complete path from the root node and goal has been established, the upper-bound cost for nodes which connect to the completed path, is then set to the distance along the path from node to node.

Results

Results shown the stars are the samples and the circles are the nodes. The lines color change from cold to hot as samples continue.

U obstacle:

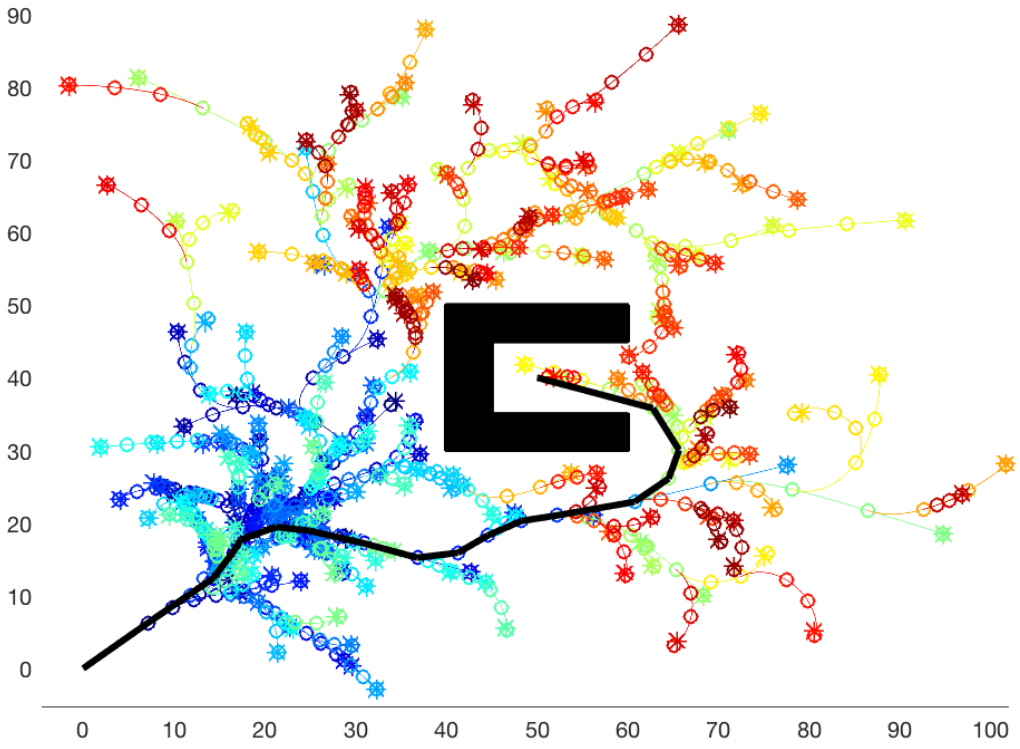


Figure 3. 300 samples were taken, σ_r= 20, σ_θ=π/10, r_0=20 for the first 150 samples and r_0 = 40 for the second 150 samples. Goal configuration is at (50,40);

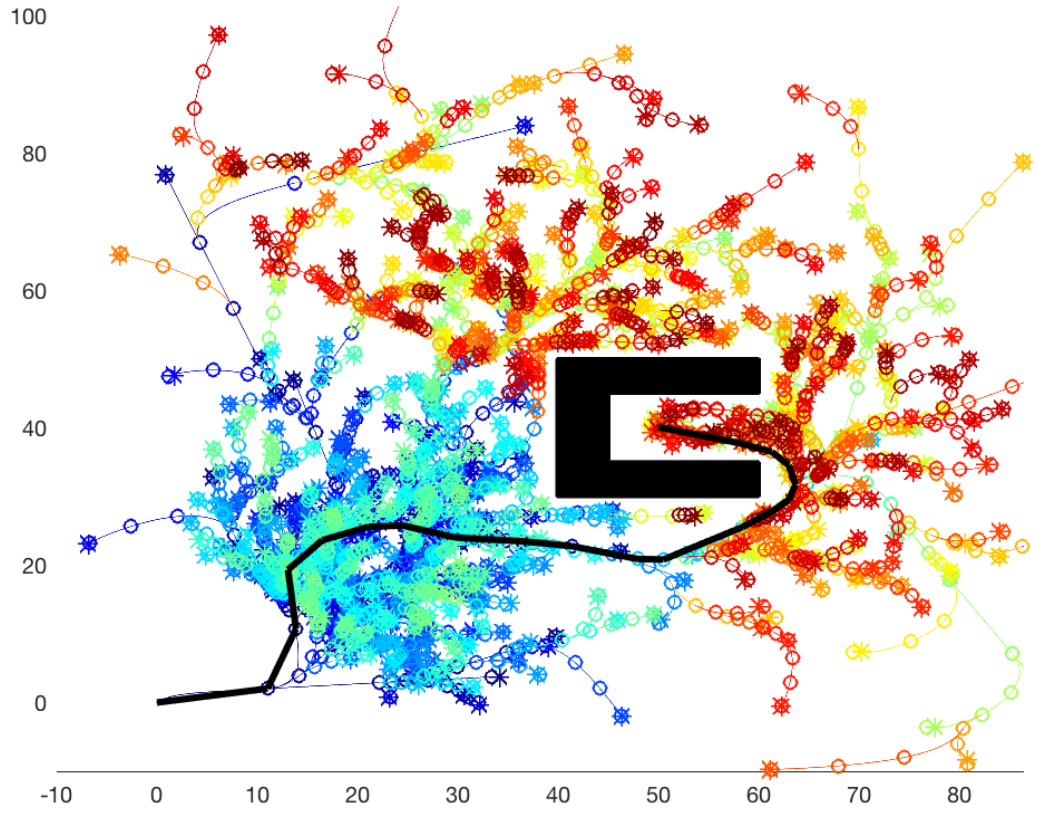


Figure 4 1000 samples were taken, σ_r= 20, σ_θ=π/10, r_0=20 for the first 500 samples and r_0 = 40 for the second 500 samples. Goal configuration is at (50,40);

Maze:

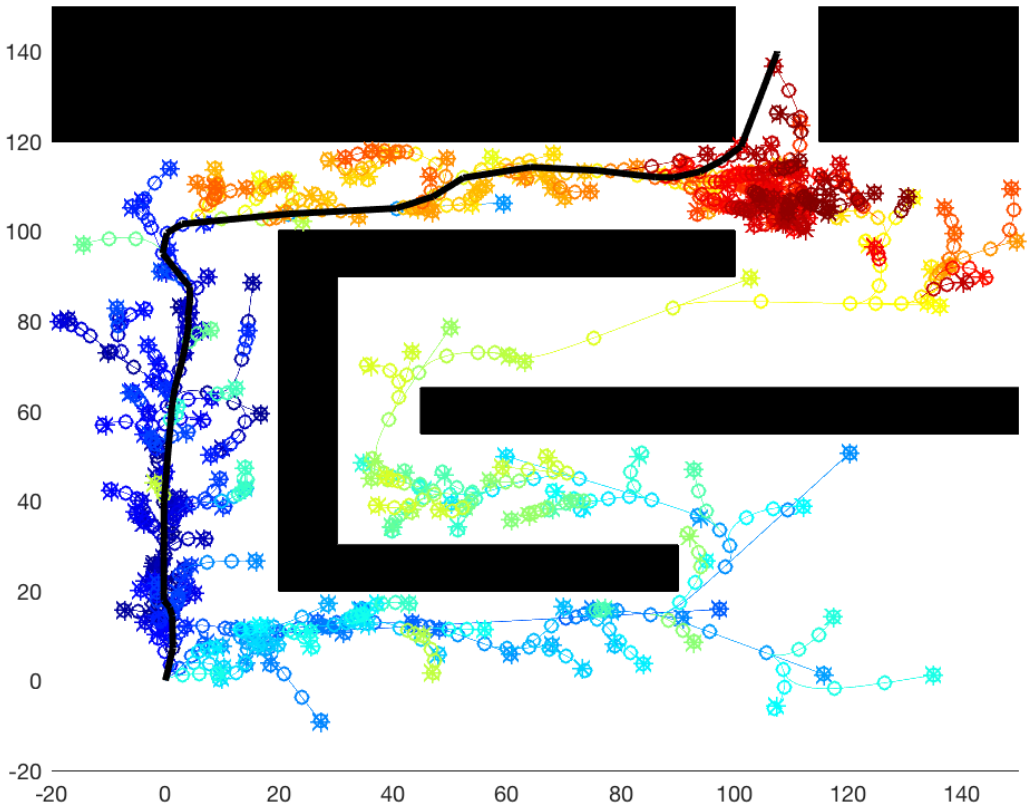


Figure 5. In this simulation 5 regions were sampled to emphasize exploration. 500 samples were taken in all. Goal configuration is at (107.5, 140)

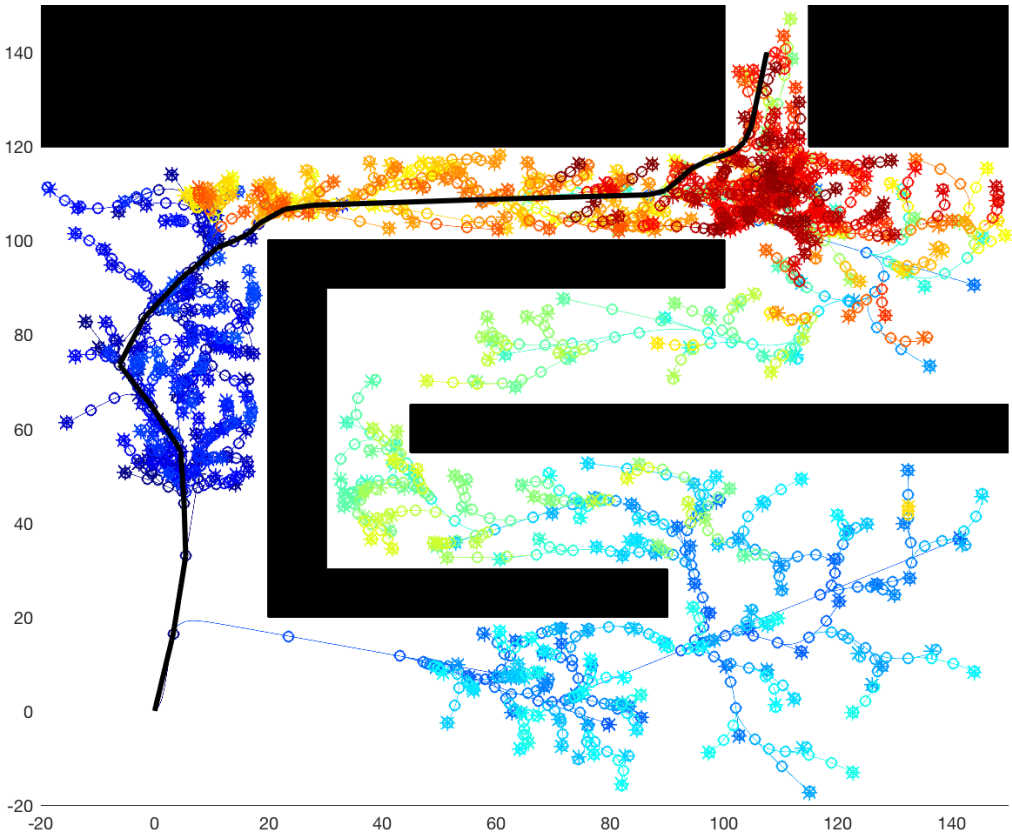


Figure 6. In this simulation, 5 regions were sampled to emphasize exploration. 1000 samples were taken in all. Even still there is a lack of nodes around point (107.5,140)

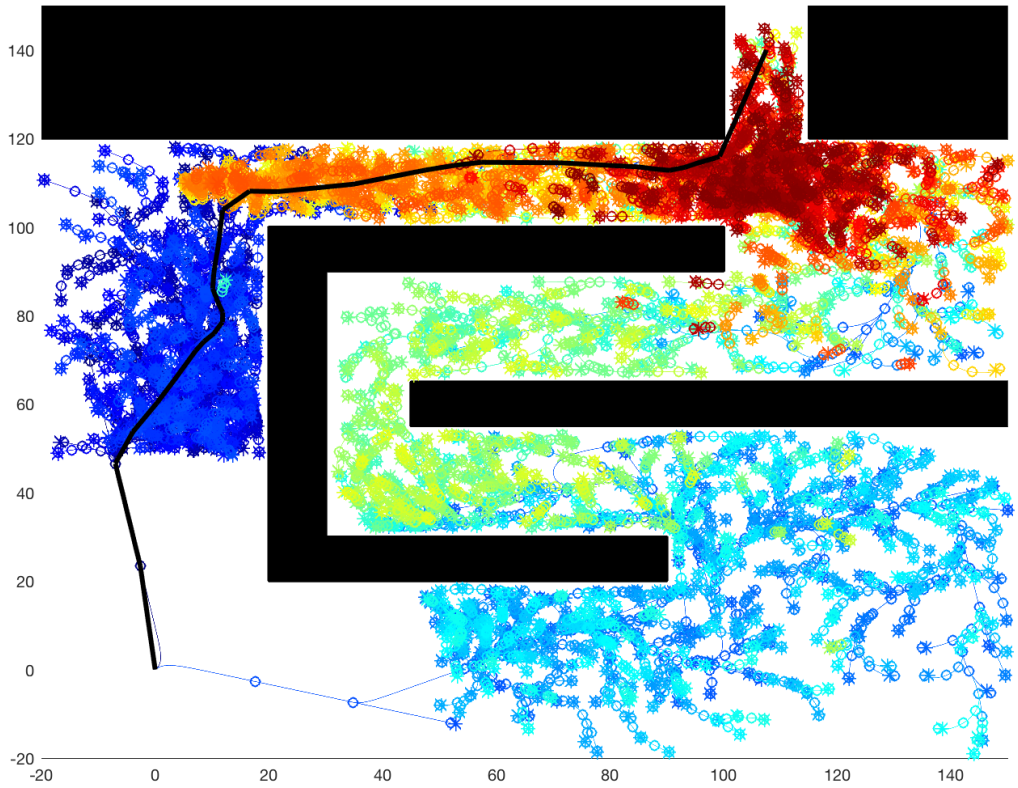


Figure 7 In this simulation, 5 regions were sampled to emphasize exploration. 5000 samples were taken in all. Even still there is a lack of nodes around point (107.5,140)