**046213 – Mobile Robots – Lab2 Report**

Lahav Barak ID 312148307

Yaron Halle ID 066077033

**Part 1 Helper Methods**

* 1. Configuration Sampling

We implemented the function **sample\_configuration**. The function takes the map parameters and the desired dimensions of the output matrix. The function first scans for all the free pixels in the map (using **NumPy.argwhere** function) and then calculates these pixel's bounding box. The integer indices are later converted to real-world coordinates using the **map2pose\_coordinates** function. Finally, the (x,y) pairs are sampled according to the computed x and y ranges boundaries of the available free space in the map.

* 1. Collision Check

The function **collision\_check** was implemented for this the purpose of asserting that a specific robot's configuration is entirely collision-free. The function takes once again all the map's parameters, and the robot's configuration represented as a . To avoid computation of specific azimuthal orientation of the robot described by the , we take the approach of computing a bounding-box that will bound all the possible postures of the robot in respect to . To that end, we use 2 times the wheelbase length as the length of the robot's bounding square. Given the coordinates of the robot, we compute the bounding box that surrounds it. We then convert the real-world coordinates back to map integer indices using the **pose2map\_coordinates** function. Lastly, we use NumPy's **any()** function to efficiently query if there's any True pixels (i.e. obstacle) inside the projected robot's bounding box in the map array.

* 1. Control Sampling

The function **sample\_control\_inputs** was implemented. The function samples possible control commands for the robot. Recall that the command vector is comprised of the velocity and the steering angle. For the speed command we chose 0.5-1 [m/s] as the dynamic range to draw from. Regarding the steering angle, we used the RVIZ display to retrieve the laser beam and the left-front wheel quaternion angles to deduce the minimal and maximal steering angles. Recall that the steering angle is defined in the robot's local frame. We use the laser beam's orientation as a proxy for the robot's local frame axis, and by subtracting the orientation angle of the laser beam and left-front wheel (at extreme left-hand and right-hand turning scenarios) we can compute the steering angle. From our calculations (after using Quaternion to Euler angles web calculators), we deduce that the dynamical range of the valid steering angle resides between (-0.3) 🡪 0.3 [rad].

* 1. Forward Simulation of the Model

We implemented an Ackerman car steering model in the function **forward\_simulation\_of\_kinematic\_model**. The function takes a given robot's configuration represented as , the corresponding time of that configuration, the velocity and steering angle commands and the time duration to exert these commands. Since every car transition needs to be collision-free checked, we propagate and store the the robot's configuration (due to the commands) throughout the time duration of the control commands. That is, besides the final resultant configuration, we also keep track of the intermediate configurations leading to the final configuration at the end of time duration. We do that, so we can supply our *edge* collision check function with a complete transition of the car between two considered configurations to be connected in the graph. We implemented the standard Ackerman car model of the form:

Where:

* is the velocity command,
* is the steering angle command,
* is the car's wheelbase (extracted from the XACRO file),

And the state propagation is computed in the following way:

**Part 3 Kino-RRT Trajectory**

The Kinodynamic-RRT was implemented. The main function is **create\_kino\_rrt\_traj** which takes a map\_file as in input and sequentially invokes the Kino-RRT search (implemented in function **kino\_rrt**) between consecutive segments that form a complete lap. Every time the Kino-RRT search is invoked, it is inputted with the last segment final configuration as its start time and configuration. In addition, the Kino-RRT is inputted with the x and y ranges such that the sampled configurations will only be drawn only in the rectangle defined by the starting and finishing mid-points. This mitigates the number of needed random samples by focusing the samples only to a relevant area in the map.

Once the Kino-RRT is completed, its configurations are standardized into 0.5 [sec] time steps using **SciPy's interp1d** interpolating function. The interpolation is needed since the Kino-RRT nodes' times stamps are random. We want to the locations of the path to be adjusted for the robot's actual velocity (according to our experience from Lab1 submission).

After we hold an interpolated path, we aggregate it to previously planned path so far using NumPy's vstack function.

We'll now dive into describing the main Kino-RRT search in function **kino\_rrt**.

**kino\_rrt** function synopsis:

The RRT search takes the start and goal configurations and the corresponding time of the starting configuration. The function takes the x and y ranges to be used for sampling new positions. The function also takes all the necessary map parameters.

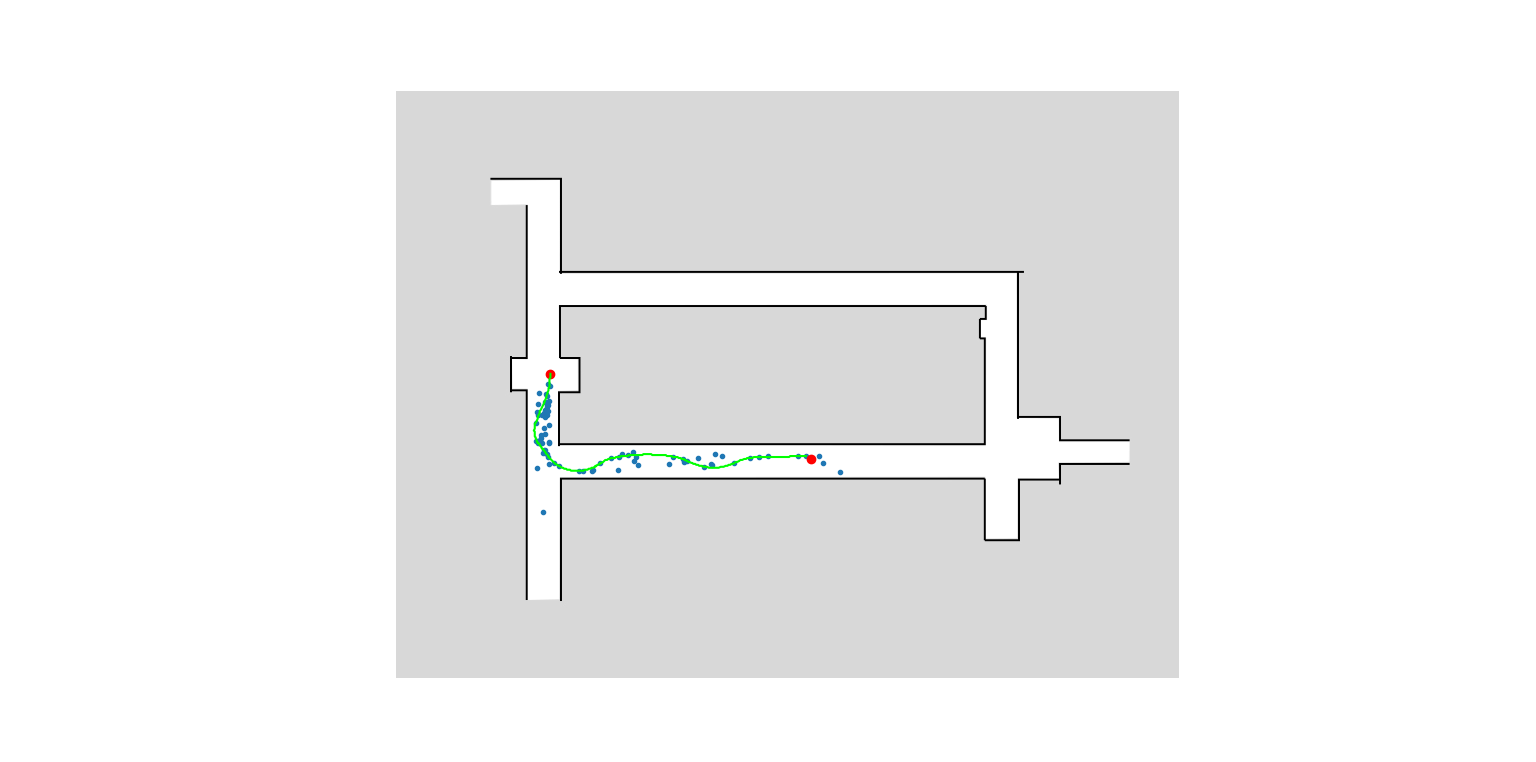
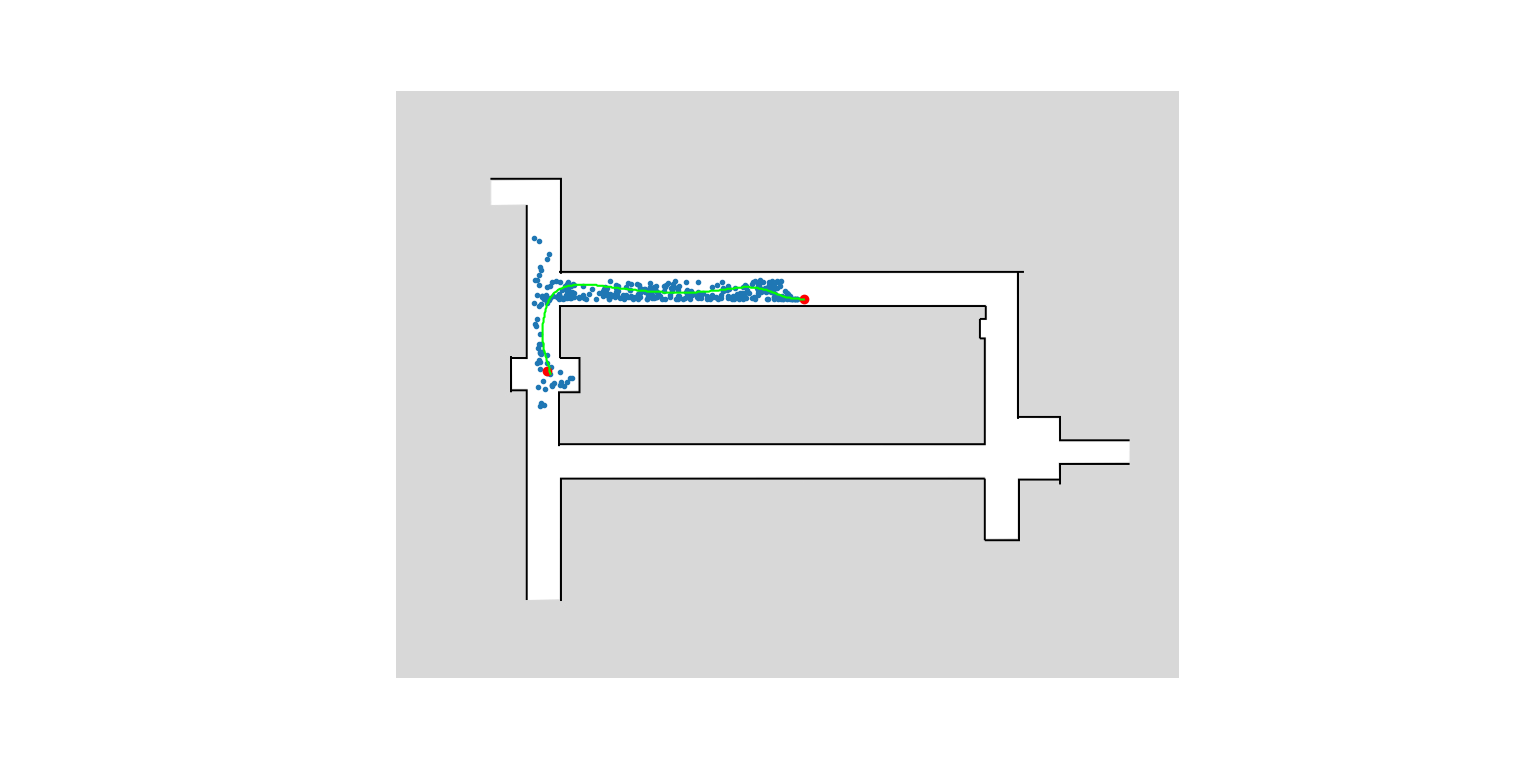
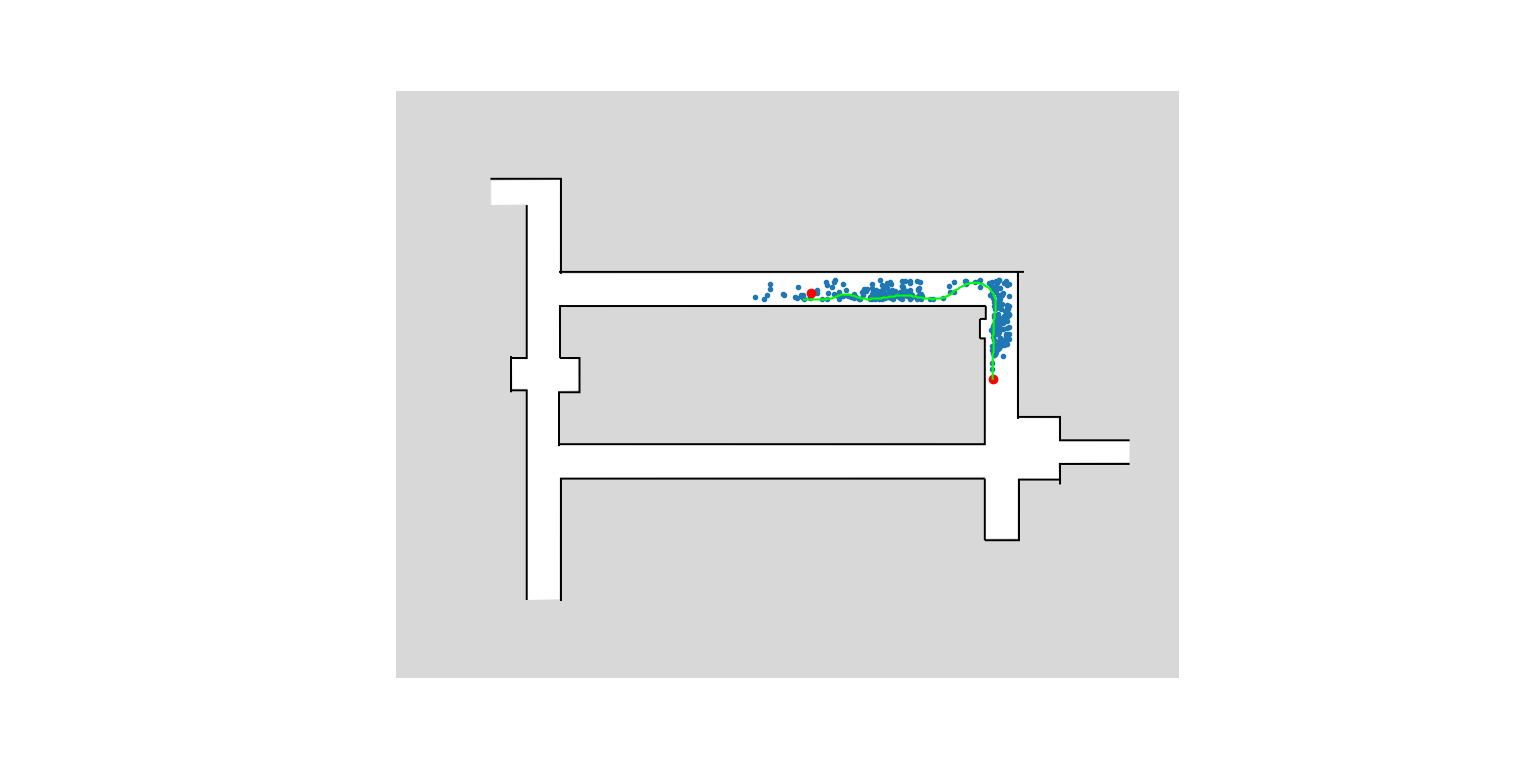
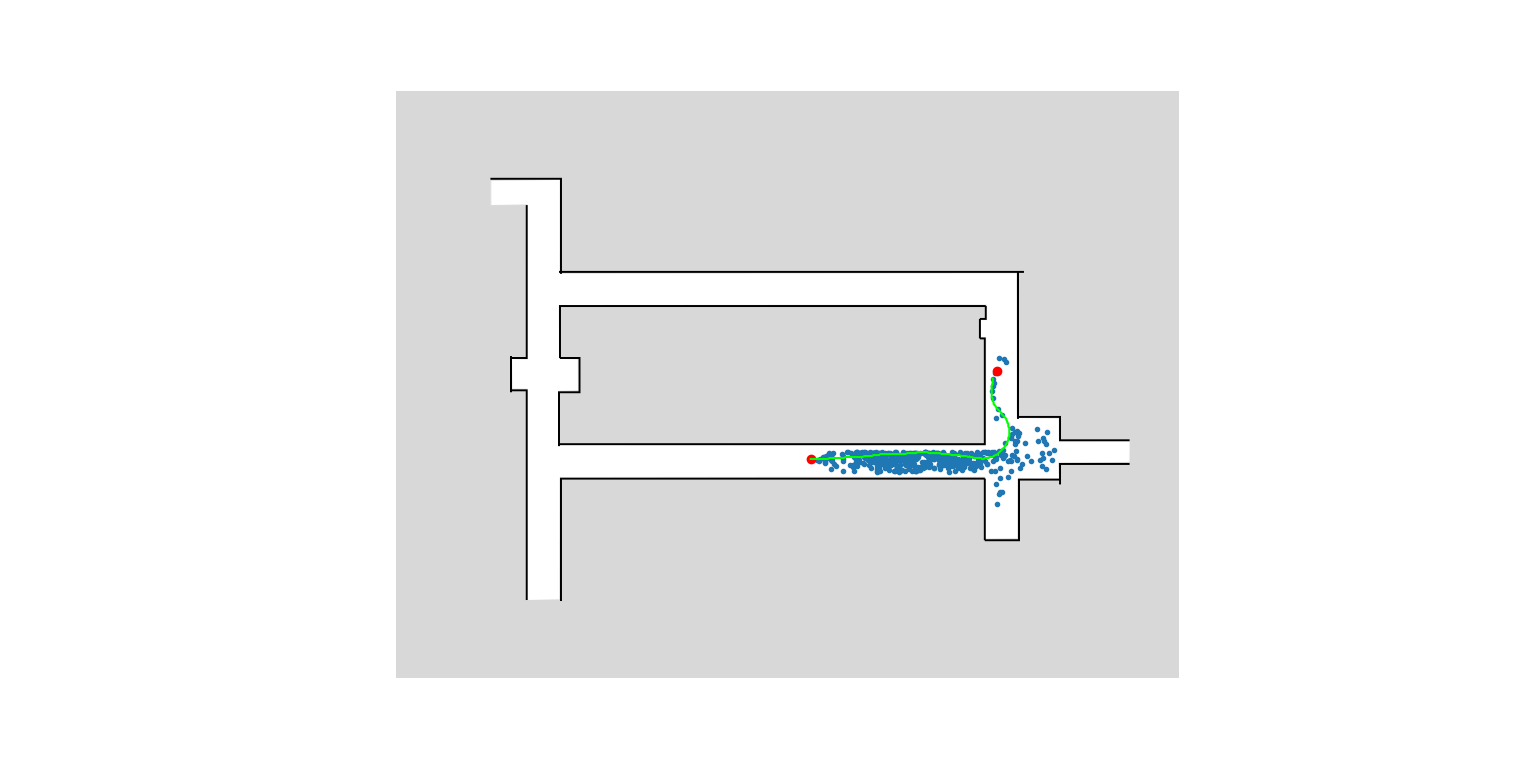
The search is initialized by instantiating a search tree by calling the **RRTTree()** class constructor. This class is a very simple one, comprised of a list of vertices and edges. Each vertex has a unique id and an edge is defined by its starting vertex id and finishing vertex id. An important class method is **get\_nearest\_state**. Given a new considered configuration (state), this method scans all the nodes comprised in the RRT tree and computes the mutual distances (using helper function named **compute\_distance** which returns the Euclidean distance between the two (x,y) coordinates of each pair of configurations). The function returns the tree node that was found to be the "closest" to the newly considered configuration.

The start configuration is seeded into the tree search using the method function **add\_vertex**. Then, the algorithm main while-loop starts comprised of the following steps:

1. Repeatedly sampling a new configuration . Note that we omit the argument since the nearest-neighbor search is performed only using the (x,y) coordinates (in our experiments we trialed a version of the nearest-neighbor that does take the argument as part of the distance metrics, we noticed that good results can be achieved without it).
2. Looking for the nearest neighbor among the RRT's tree nodes using the **get\_nearest\_state** method.
3. Sampling control commands and time duration of exerting these commands using our previously described **sample\_control\_inputs** function.
4. Propagating the dynamics starting from the nearest state while invoking the sampled commands during the sampled time duration. The state propagation is computed using our previously described **forward\_simulation\_of\_kinematic\_model**. The function outputs mid\_state, a sequence of configurations sampled every 0.1 [sec] from the nearest state towards the new derived state.
5. Collision Check: Taking the sequence of previously computed configurations, they are tested for collision clearance using **is\_edge\_collision\_free** function. This function for-loops over all intermediate configurations and checks for collision clearance using our previously described **collision\_check** function. If the "edge" (the state transition) intersect an obstacle, the for-loop continues by going back to stage step (1).
6. If the edge is collision-free, the new state is formed by taking the last sequence of the previously computed intermediate configurations. Lastly, the new state and its corresponding edge are added to the tree using the **add\_vertex** and **add\_edge** functions.
7. Checking if goal configuration was reached by calling the **are\_configs\_close** function. The function takes the new state and the goal state and checks whether new state resides "close enough" to the goal configuration. Conditions for reaching the goal configuration's vicinity:
   1. The Euclidean distance between the two (x,y) coordinates of the two configurations is smaller than **0.5 [m]**.
   2. The absolute azimuthal difference between the parameters of the two configurations is smaller than **3 [deg] ~ 0.052 [rad]**.
8. Once the goal's vicinity is reached, the while-loop is terminated and the path is backtracked from the last node towards the start node using the edges connections.

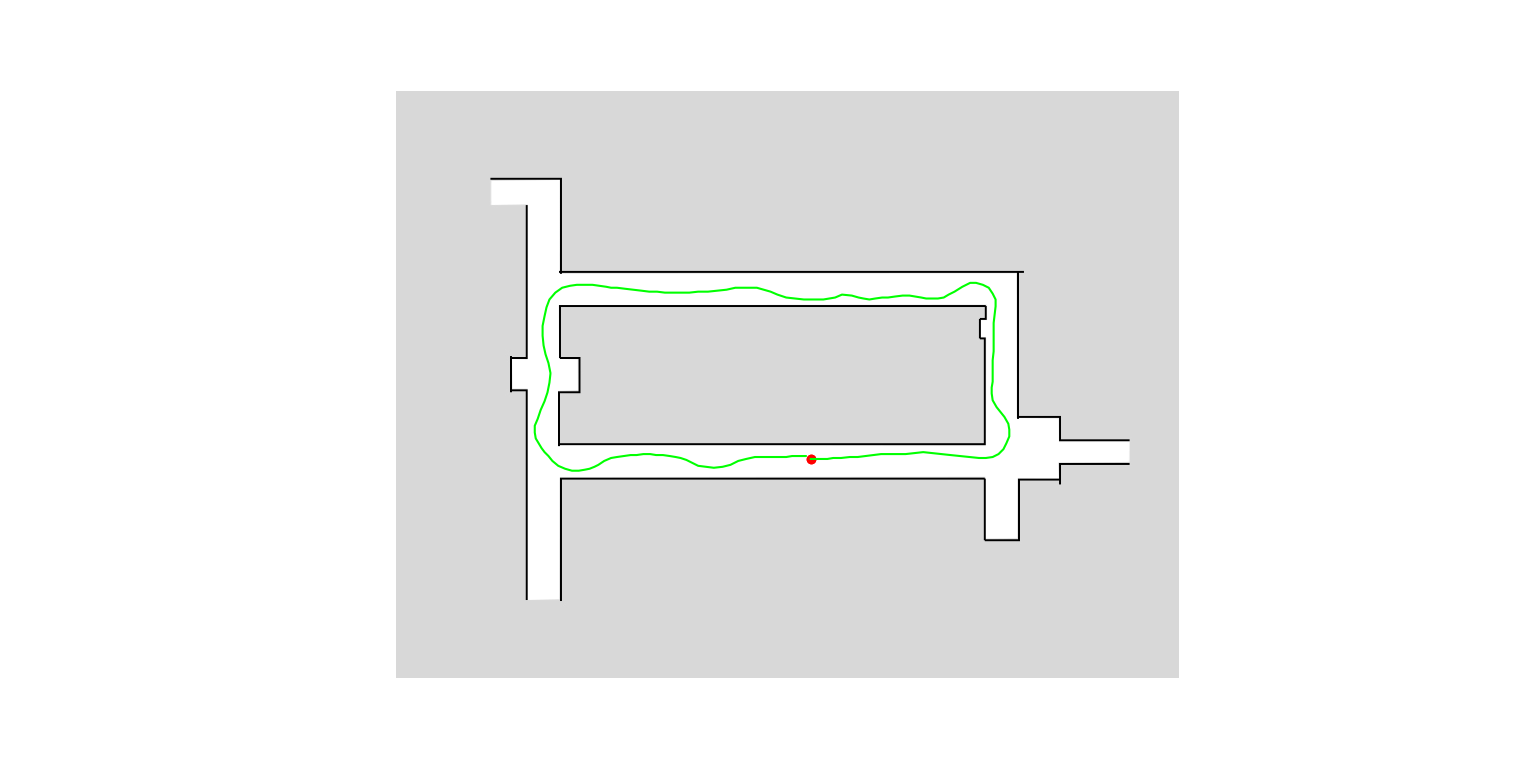
Experiments

We executed the Kino-RRT search for the **levine** map. We first depict the intermediate paths planned for each segment. Each figure displays the starting and finishing mid-points (points), the RRT's tree configurations (blue points) and the computed path (green curve). The paths are aggregation of all the intermediate configurations while propagating the every path node in the tree towards its successor.

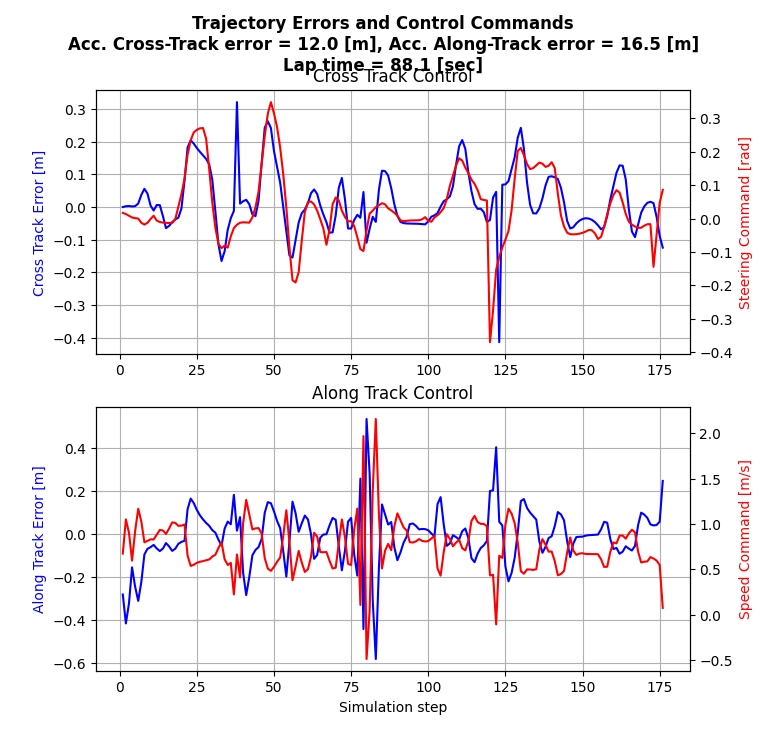


The above graphs well demonstrate how Kino-RRT takes into account the true dynamics of the Ackerman model, producing smooth and realistic transition curves.

And by connecting all the green curves together we get the complete lap planned trajectory:



We fed the generated trajectory file to our optimal controller of lab1 (which was previously described in detail in lab1 submission report). The results are depicted below:



The above figure depicts the control effort of both the cross and along tracks. The graphs depict both the error and the control commands (steering angle and speed command). We can see that good control was achieved with accumulated cross track error and along track errors of 12 [m] and 16.5 [m] respectively. It's easy to see that the average errors (both along and cross track) are merely zero. The total lap times was 88.1 [sec].

In addition, we provide the following figure comparing the robot's actual trajectory vs. the desired trajectory (supplied in kino\_rrt\_traj.npy). As we can see, the two curves (the green and red) almost perfectly overlap.

