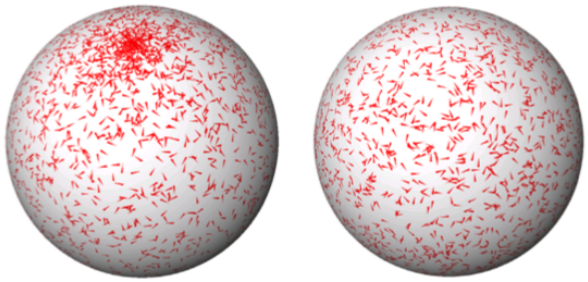
Sampling-Based Motion Planning

Deep Patel and Akash Desai

November 18, 2018

1 Part A: Sampling-based Motion Planning for a Rigid Body in SE(3)

1.1 Sampling



(a) Naïve sampling (b) Uniform sampling

Figure 1: Naive sampling vs Uniform sampling of SO(3)

To define SE3 we need to represent the a body's positions and orientation. There are 3 possible ways of defining orientation: 1) Rotation Matrix, 2) Euler Angles, and 3) Unit Quaternions. To represent configurations in SE(3), it requires at least 6 parameters: namely, 3 for describing the position in the 3-dimensional space (x,y,z) and 3 for describing the orientation of the rigid body, which can be the Euler angles: yaw, pitch, and roll. Using Euler angles becomes difficult because the order of the angles matters and also many combinations of Euler angles can describe the same orientation, causing fundamental ambiguity. Because of this, we decided not to use the minimum number of parameters and chose to represent the orientation of the rigid body using Quaternions, which consist of 4 parameter. In order to the represent the position of a body-fixed frame relative to a stationary world frame we used the standard Cartesian coordinate system, defining them as x, y, and z. To parametrize the rotation in

three dimensions, in Quaternions, any orientation is achieved by a single rotation, theta, about a single axis angle v = (vx, vy,vz). The unit quaternion is shown by: $Q{=}(w,x,y,z){=}(cos(\theta/2),v_x sin(\theta/2),v_y sin(\theta/2),v_z sin(\theta/2))$. Some nice features of Quaternions include that they are compact & efficient, easy to renormalize, and have other simple features that are directly useful in path-planning tasks like sampling, distance metric, and interpolation. We are sampling collision-free configurations in the C-space by uniformly sampling x, y, and z coordinates between the boundaries of the walls of the environment.

Further, to ensure that the orientations are also uniformly random, we use the Algorithm 2 in James Kuffner's paper. [2] After generating each state, we use a collision checking library called PQP to see if the sample is a valid configuration. This is done by using a triangular mesh of both the environment world and the rigid body, which is a piano in this case, and using PQP to check that these meshes do not intersect at any point for the certain configuration of the piano. If a collision occurs, then the sample is discard and a new one is regenerated until a collision-free sample is found. In this way, we are ensuring uniform at random, collision-free configurations in the free-space of the configuration space.

1.2 Distance-Function

When choosing a distance function, it is crucial to ensure that it is efficient and accurate, because it is called by the algorithm many times to find nearest neighbors. By accurate, I mean, it should be as close to the minimum swept volume in the C-space, because this minimizes the chance of collision with obstacles and maximizes the likelihood of discovering collision-free paths. [2] Because finding the exact swept volume would be computationally inefficient, we decided to use a distance metric that is a weighted combination of the translational and rotational distances. Lets define the configuration to be $q = (X, R) \in SE(3)$, where X is the translational component and R is the rotational component, then we defined the distance, ρ between two configurations as in the equation below, which is the weighted sum of the translational and rotational components.

$$\rho(q0, q1) = w_t||X_0X_1|| + w_r f(R_0, R_1)$$

For the translational component, we used a standard Euclidean L2-norm distance because this is the actual distance a path would take between configurations in SE(3). For the rotational component, we used an approximate metric that uses the dot product between two unit quaternions. This is explained by Kuffner in his paper on effective sampling, where he explains that the arccosine of the the scalar inner product of two 4D unit quaternions is directly proportional to the length of the geodesic path on the 4D unit sphere.[2] He also accounts for the fact that two quaternions, namely any Q_1 and Q_2, represent the same rotation if $Q_1 = -Q_2$, by taking the absolute value of the inner product, which effectively take the negation of one of the quaternions. We chose the weights for the rotational and translational motion as a function of the

Euclidean distance, because we found previous research has suggested that as planning queries become harder, the rotational component becomes less important, so we defined "harder" as being far away translationally. The function we came up with, after tweaking the coefficient based on results from many trials, is given below, where d_t is the Euclidean distance between the two configurations and we found that setting c = 0.2 gave a good amount of weight on the rotation, so that at short distances, the closest neighbor is skewed towards ones that requires less rotation because that becomes more important. The function goes to 0 as d_t goes to infinity and tends to c when d_t tends to 0.

$$w_r = c * \frac{1}{1 + d_t}$$

This metric seems appropriate for finding good neighborhoods because it approximately minimizes the swept volume in the configuration space, ensuring that we maximize the likelihood of finding collision-free paths to when finding neighborhoods.

1.3 Local-Planner and Collision-Checking

In practice, paths between configurations should be smooth, without erratic changes rotation. To ensure this, we designed our local planner to find discrete intermediate configurations between any two configurations of interest that uniformly discretizes the translational distance at a constant step size and uses the intermediate quaternion orientations between the two configurations so as to ensure rotation along the minor arc of the 3D sphere, which is discussed in [2]. The interpolation discussed in the paper gives the intermediate orientation between two points. To achieve all the intermediate orientations, we used a nested approach, where we find the middle orientation and then use that to find the quarter way and 3/4 way orientation and so on. We chose the step size to be around 0.8 times the size of the piano in the shortest length, which we measured from the Gazebo visualization of the world to be approximately 0.5 meters. This seemed like a good choice because there must be some overlap of the piano between neighboring configurations in the path because if not, an obstacle in between two neighboring configurations might be missed. We chose 0.8 instead of just exactly the size of the piano so that we can account for the shape of the piano, which is not symmetric and so that corners and such are not missed along the path, so the final step size choice was 0.4 meters. We could have decreased this step size even more, but this seemed more than sufficient and decreasing it would just add to the computational effort.

Now that we have this local path, we must collision check it to see if the path is valid and collision-free in the C-space. To collision-check, we use a collision detection library called PQP, which takes two models of objects in the form of triangular meshes and checks collision between all of the triangles of the models. To be accurate, this requires a fine representation of the model with very small triangles, which was provided for the rigid body, the piano, and the environment bounds and its corresponding obstacles. By collision checking every

configuration in the discrete path obtained by the local planner, we checked if the path between two configurations was obstacle-free and assigned an edge between the sample configurations if it in fact was collision-free.

1.4 Implementation of PRM and its variants

We implemented two varients of the PRM algorithm: the k-nearest-neighbors version, where we set k to be a constant, and the PRM* version where $k(n) \geq e(1 + \frac{1}{d})log(n)$. In Figure 2 and 3 you can see examples of roadmaps built with the PRM algorithm with k=5 nearest neighbors and the PRM* algorithm, respectively, both with 100 samples. As you can see the PRM* graph is much more dense than the PRM graph because k above that threshold. You can also see that there might clearly be some disconnected components in the PRM graph, which is not so much the case in the PRM* graph.

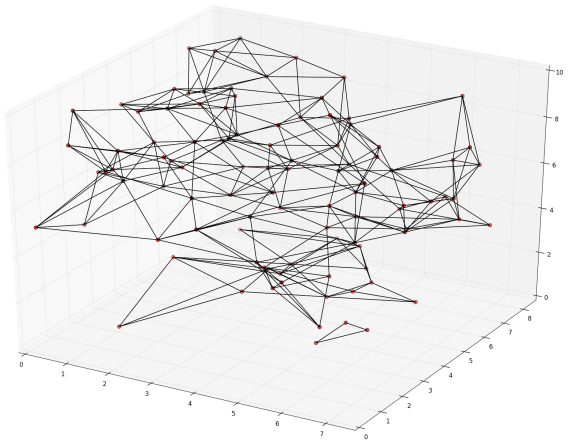


Figure 2: PRM graph with k=5

1.5 Sampling Initial and Goal Configurations

To sample these random starting and ending configurations, we used the same process as sampling configurations in the C-space to build the PRM graph, but while restricting the value of z to 0.4 (for the piano center, because the initial pose z is set to 0.3 in the models/models.sdf file, but this led to collision with the ground, so we increased it by a little), so that the piano legs would be

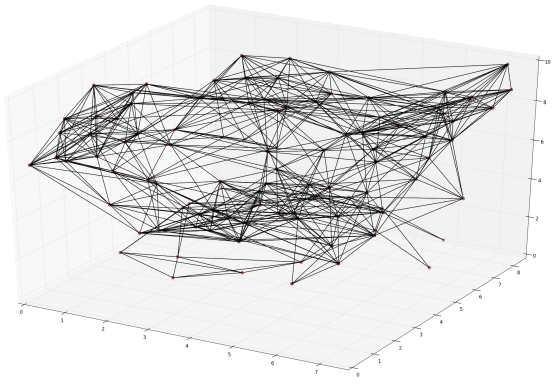


Figure 3: PRM* graph

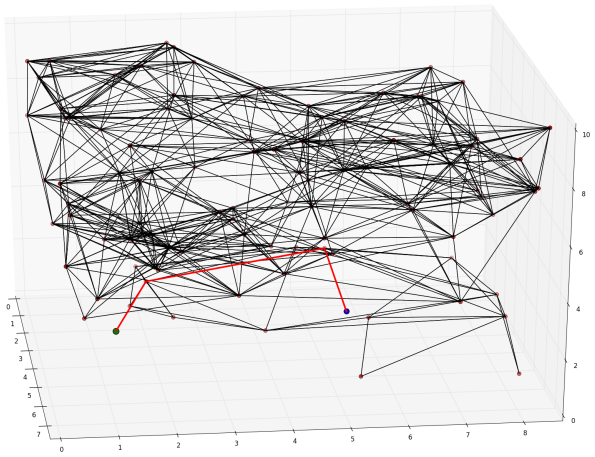
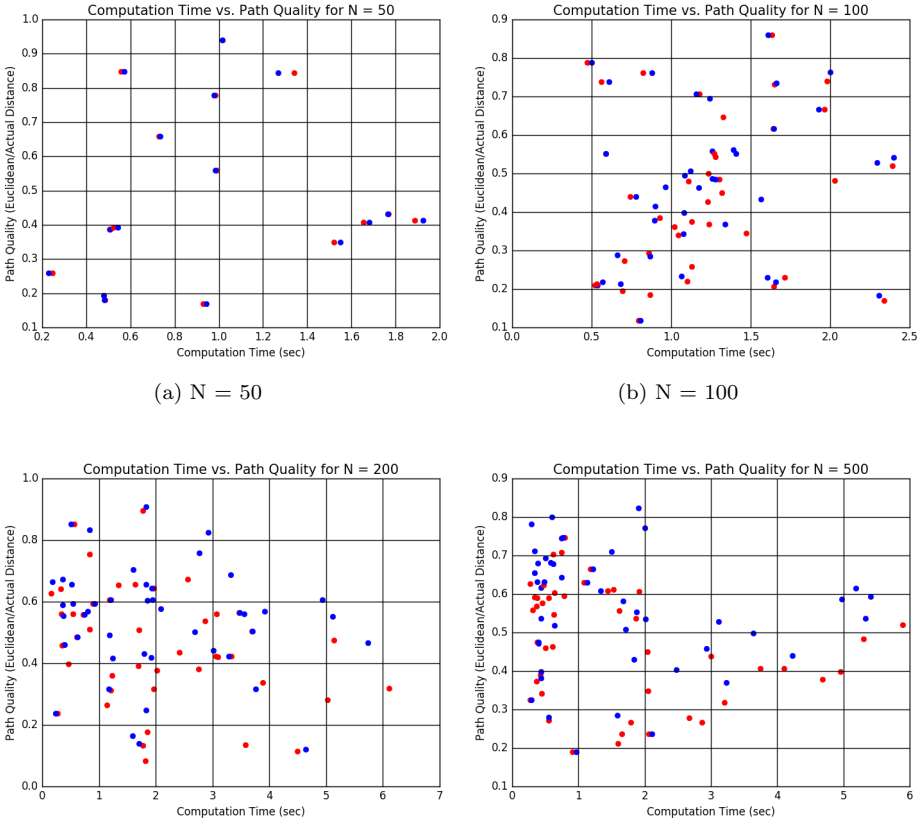


Figure 4: Initial and Goal Nodes with path

resting on the ground at the start and goal configurations. We set the initial orientation to be only random along the z-axis, which essentially means it can must be sitting on the ground, but can be facing any direction. We did this my randomly sampling the q_3 component of the quaternion, setting q_1 and q_2 to 0 and finding the scalar, q_0 using the constraint equation. Again, we ensured these configurations are collision by using the PQP Collision Detection library. You can see in Figure 4, the initial node (colored blue) and goal node (colored green) are both at z = 0.4, which is approximately the ground of the environment. To see that they are indeed collision free, they can be checked in Gazebo.

1.6 Results and Analysis



(a) N = 50

(b) N = 100

(c) N = 200

(d) N = 500

Figure 5: Computation Time vs. Path Quality of 50 query pairs of random start and goals using PRM with k=5 (in red) and PRM* (in blue) for various numbers of samples, N = 50(a), 100(b), 200(c), 500(d), and 1000(e) samples

After computing the solutions to the path planning problem for the 50 start

Table 1. Average Path Quality and Average Solution Time for PRM & PRM* for different roadmap sizes

	PRM	PRM	**PRM***	PRM*
N	***APQ [0,1]***	***AST (sec)***	***APQ [0,1]***	***AST (sec)***
50	0.278	1.071	0.281	1.110
100	0.311	1.377	0.318	1.384
200	0.306	1.784	0.312	1.882
500	0.357	1.839	0.394	1.942
1000	0.365	2.039	0.413	2.493

and goal states for the k-nearest variant of PRM and PRM*, we can see many interesting trends that support previous research and that support the optimality of PRM*, which was first introduced by Sertac Karaman and Emilio Frazzoli in their paper in 2011. [1] As the number of sample increase, you can see that the computation time decreases while the path quality increases slightly. For path quality, we divided the Euclidean distance between the the start and goal locations and divided it by the actual length of the path found as this is a good measure of how far away it is from that. For computation time we just considered the time it took to find the path, not considering the time taken to build the graph itself, as this can be done offline. We performed these queries to find solutions for the 50 start and goal locations on different sized graphs to see the impact that it has on path quality and computation time. We can see in Figure 5, the graphs of this data plotted as a scatter plot for the 50 different start and goal configurations. As you can see, the graph for N=50 shows significantly fewer points than the others. We found that this is because with too few nodes in the roadmap, the start and/or goal configurations have a harder time being able to connect to neighboring nodes in the roadmap that are collision-free, in which case, no solution can be found for that query pair and thus it has been eliminated form the graph. We looked for collision-free paths to points in the roadmap by discretizing the path to the neighboring node and using PQP to collision check it, similar to the method used for collision checking paths between configurations in our local planner. This fact showed us that a minimum number of samples must be used in order to cover the sample space well and account for all possible start and goal locations being able to connect to the roadmap. Further, in Table 1 you can see for different sizes of samples in the roadmap, the average path quality and average first time to find a solution.

2 Part B. Sampling-based Motion Planning for a Physically Simulated System

2.1 Kinodynamic RRT Algorithm

RRT is a sampling based tree planner. It can be applied to problems with dynamics, just like the Ackermann vehicle, by sampling controls in a kinody-

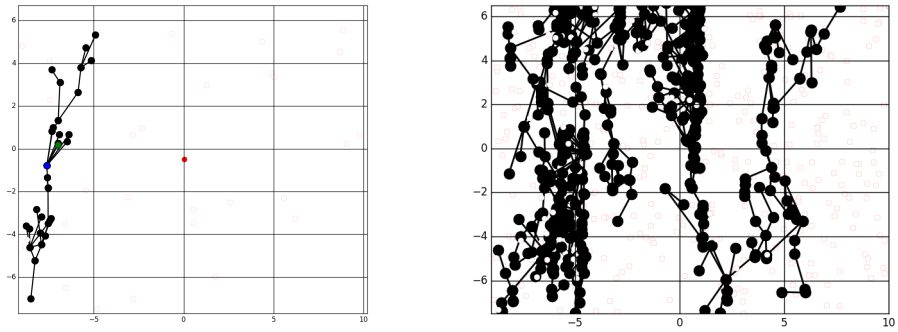
namic fashion. The high level overview of RRT is:

- Randomly sample a configuration in the state space
- Propagate a random control with a random duration from the closest part of the tree from that node
- Choose the control that resulted in a state closest to the node and add it to the tree
- Repeat until the goal is reached

For our project we used a Ackermann vehicle with a physics engine. The steering of the vehicle is similar to that of a tricycle. In order to build a tree, we start at a an initial pose and twist in the world(start node). 1) Then we randomly sample an x_{rand} which corresponds to a x,y position on the map. 2) From our tree we then find an x_{close}, which the closest node in the tree to the newly sampled x_{rand}. 4) Then we randomly sample [u_{rand}, t_{rand}] k times, where u_{rand} = [u1, u2] is a randomly sampled control; here, u_1 = speed (m/s) and u_2 = steering angle (radians). t_{rand} = [time] is randomly sampled time in seconds. This way we get random controls for random duration of propagation. After propagating the controls in gazebo we would get Ackermann vehicle's current model-state in order to get the most accurate representation of the position that resulted from the controls. After propagating we would set state back to the original pose and twist and continue to propagate the rest of k controls. The state(with randomly propagated control) x_{new} closest to the x_{close} is added to our tree with parent being x_{close} while the rest of the controls and x_{rand} is discarded. We decided to store the pose and twist of each tree node because the same controls with the same conditions led to varying states due to the physics engine. We defined the goal state to be a region around the around the goal position. When a node is added to the graph within this region, we backtrack to the find the pose and twist that led to the current pose and twist in our goal region. Additionally we know that this algorithm is probabilistic ally complete, because if there is a way to get from start to goal, then with enough iteration we will eventually get there. One of the areas where RRT lacks is tight spaces, however the maze have openly spaced obstacles. Also the algorithm can be made asymptotically complete by using the best-near selection method.

2.2 Discovering Solution Trajectories in Gazebo

Figure 6 is an example of a tree that was built in the Gazebo world that reaches the goal region. The trajectory is then found by tracing back from the node that reached the tree to the start position by accessing the parent of each node along the path until the start is reached.



(a) Expanding the tree with random controls

(b) Solution trajectory

Figure 6: Kinodynamic RRT Algorithm in action finding a solution trajectory one the left and the trajectory reaching the goal on the right

2.3 Choosing a Distance Function that achieves Voronoi Bias

We achieved Voronoi bias with our kinodynamic RRT algorithm by randomly sampling a node in the space, x_{rand}, finding the closest node in the tree to x_{rand}, x_{close}, and randomly sampling many controls from that state and taking the many final states from these random propagation of controls and choosing the one that is closest to x_{rand} to set as x_{new}. As we are uniformly sampling x_{rand}, at every iteration, we are choosing the final state that was achieved by propagating closest to x_{rand}, which grows the tree in uniformly random directions because x_{rand} is chosen randomly. This ensures that the tree is exploring the unexplored regions, because the probability of x_{rand} being in an unexplored direction is higher than being in an already explored direction because of uniformly random sampling. Thus the exploration strategy has an interesting property, this is characterized by the Voronoi bias. At each iteration, the probability that a node is selected is proportional to the volume of its Voronoi region. Thus the samples are biased towards the largest Voronoi region causing the tree to rapidly explore thus getting the name Rapidly-Exploring Random Trees. During our experimentation we noticed that keeping the number of sampled controls low resulted in not enough diversity of final states and in turn the tree was not really propogating towards x_{rand} at each iteration. Thus, we decided to increase this parameter to around 5-7 controls at each iteration which increased the probability of being closer to x_{rand}, and thus giving us a balance between computation time and rapid growth of our tree towards unexplored regions. Anything higher resulted in higher computation time without much added benefit of getting to the goal faster.

2.4 Greedy Kinodynamic RRT

In order to make the propagation process greedier we used a heuristic in our distance function and also increased the probability of a random sample falling in the goal region by sampling it exactly at the goal with a certain probability. In the non-greedy version we pick the x_{new} closest to x_{close} by choosing the smallest f(x)= g(x) where the g(x) is the distance from the x_{close} to the x_{new}. In the greedy version we decided to add f(x) = g(x) + h(x). The h(x) stands for the distance between the x_{new} and the x_{rand}. Thus, we add the nodes that are least in g(x) and h(x) making it propagate more towards the goal region. Additionally, we found that once the vehicle was closer to the goal region, as we were randomly sampling the map it was harder to sample towards the goal region and thus sampling more towards the goal region we found that we were able to grow the tree quicker. However as we were in a maze we could not sample towards the goal region at a really high probability because the vehicle would get trapped in a corner. We accounted for this by keeping the probability moderately low when the vehicle was far away and once it got to within a certain distance of the goal, we increased the probability of sampling the goal. This really sped up the process, by extending the tree in the area that is close to the goal region and not wasting computational effort by growing it in other unnecessary areas.

2.5 Results and Analysis

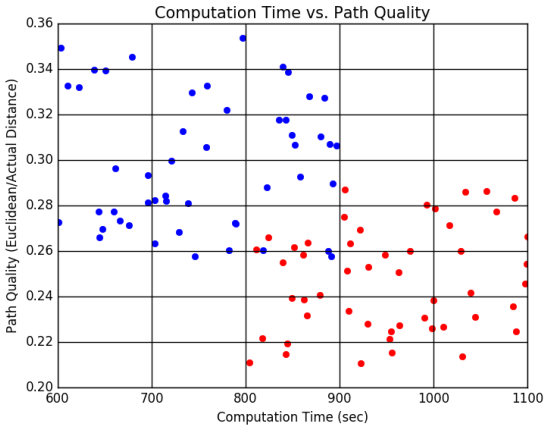


Figure 7: Kinodynamic RRT Path Quality vs Computation Time

Running the RRT algorithm was not time efficient. Because of the randomness of the algorithm, the communication between the ROS controller and Gazebo acted as a blocker in our system, making the entire process slow. For instance if a random time was sampled to be 2 secs then the car would in the

simulation for 2 seconds and give us a feedback. This multiplied by n iterations created a major roadblock to getting fast solution trajectories to the goal. The mean path quality for RRT and greedy RRT was .25 and .30, respectively. The mean computational time for RRT and greedy RRT was 970 and 750 seconds, respectively. The standard deviation for path quality RRT and RRT greedy was .024 and .928 respectively. The standard deviation for computational time for RRT and RRT greedy is 83.17 and 70.50 respectively.

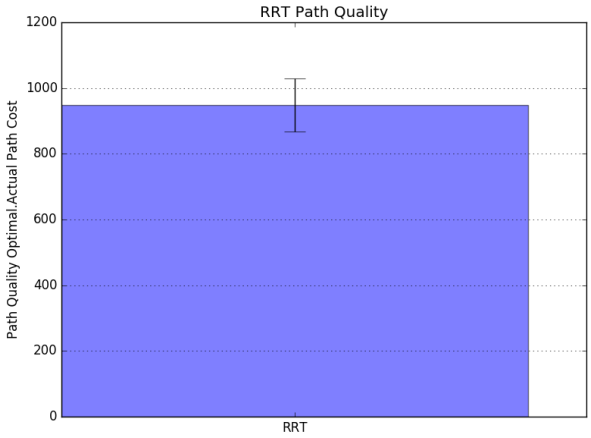


Figure 8: RRT Path Quality

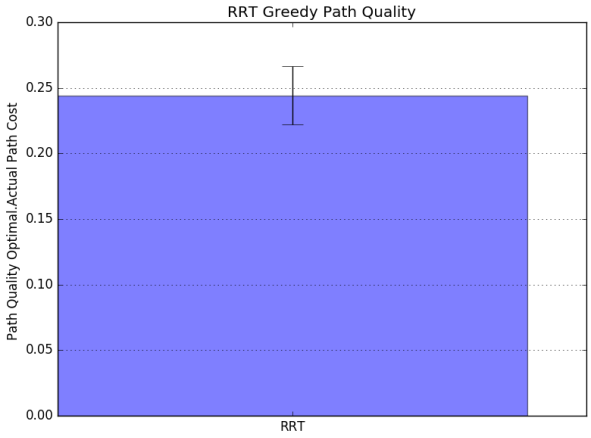


Figure 9: RRT Path Quality (greedy)

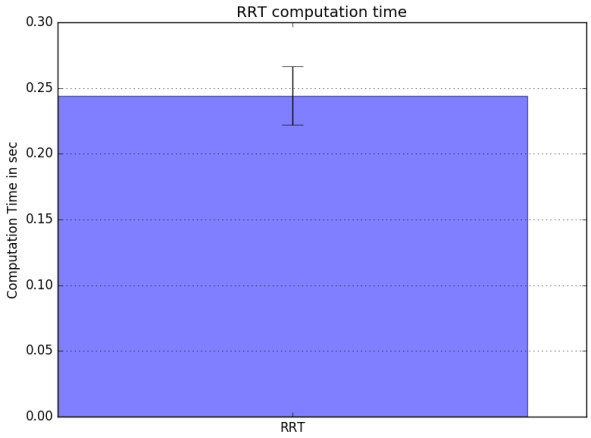


Figure 10: RRT Computation Time

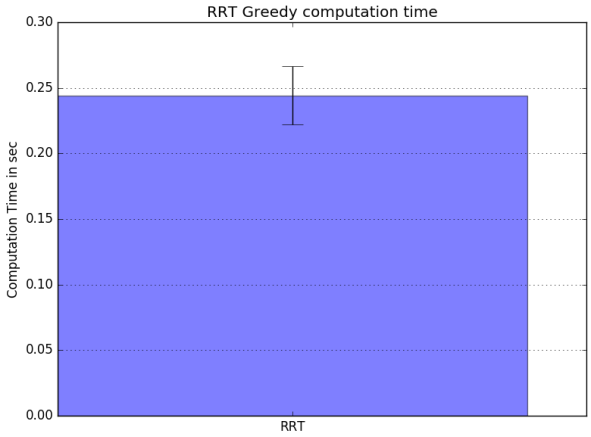


Figure 11: RRT Computation Time (greedy)

References

[1] S. Karaman and E. Frazzoli. Sampling-based algorithms for optimal motion planning. 2011.

[2] J. J. Kuffner. Effective sampling and distance metrics for 3d rigid body path planning. *Proc. of the IEEE Intern in Conference on Robotics and Automation (ICRA)*, 2004.