MEAM520 Lab4 Report

Archit Hardikar

10 April 2022

1 METHOD

The steps we followed for Rapidly Exploring Random trees algorithm[1] was according to the pseudo-code provided below[2]

Algorithm 3: RRT

$V \leftarrow \{x_{\text{init}}\}$; $E \leftarrow \emptyset$;
for $i = 1, \ldots, n$ **do**
  $x_{\text{rand}} \leftarrow \texttt{SampleFree}_i$;
  $x_{\text{nearest}} \leftarrow \texttt{Nearest}(G = (V, E), x_{\text{rand}})$;
  $x_{\text{new}} \leftarrow \texttt{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
  if $\texttt{ObtacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
    $V \leftarrow V \cup \{x_{\text{new}}\}$; $E \leftarrow E \cup \{(x_{\text{nearest}}, x_{\text{new}})\}$;
return $G = (V, E)$;

Figure 1: RRT Pseudo-code

1. **Initialize:**
   We decided that using python classes (named as 'Node' class throughout this text) made the problem relatively easy. Class had q_0 through q_6 and parent_index as the variables assigned to that class. They could be accessed from wherever it was desired. Alternate approach would be to use struct of C++ for future applications in ROS2.
   Then, initialize the environment at the start point. The start and goal point are checked for potential collisions for obstacles and if it does collide, the program returns a blank path vector. Save the first node as the start point. The q values for the start point and q values for the goal are known. Also the list of obstacles is available in the obstacles declaration in maps which can be accessed through loadmap and .txt files provided.

2. **Define the workspace:**
   We decided to work exclusively in the configuration space by using a function named as sample(). By considering the joint limits through the lowerLim and upperLim variables given as inputs, using (np.round) and (random.uniform(lowerLim, upperLim)) we generated a list/array containing q values with q_n (where n = 0,1,...6) was within the joint limits specified for each joint. Then they were rounded to the nearest appropriate decimal.

3. **Building the actual RRT algorithm/Tree setup:**
   First step is to initialize the tree vector and the path vector. Both are arrays that inherit the nodes stored in the **Node**. Then we add the initial start q configuration to the tree. Then we call a distance calculating function getdistance() that takes the norm (numpy.linalg.norm) of the difference of 2 q values and returns a scalar. This is a preliminary, check of how far apart from each other; given that the configurations are given that the q values actually represent the angles of each joint.
   The next step is to call the function nearest() to check which nodes are closest. By looping through the tree we created, the function finds the configuration closest to the newly sampled point and returns its index in the tree.
   Next, a collision detection function checks whether the newly sampled node collides with the environment and prevents failures. Then the newly sampled point is inserted on the tree and the index of the nearest node is saved as its parent. This is easily done by accessing the class and storing the value using properties of classes: tree.append(Node[x].parent = nearest_node_index). This gets stored as the 8^{th} element in the array. This process keeps happening iteratively till a boolean variable flag gets changed which tells the loop to stop and backtrace a path.

4. **Collision check function:**
   This forms an important part of the RRT algorithm. At each point of time when this function is called, the function interpolates 'n' intermediate q values between the q_initial and q_final that are inputs to the function. The calculateFK.py is used here and the joint coordinates are obtained corresponding to all of these 2+n (2 inputs, n interpolated) values. Using the detectcollision function, we find potential collisions with the obstacle along the line joining each joint and its corresponding counterpart in the next configuration. This is a look ahead implemented to prevent potential future conflicts.
   Another check provided is for detecting self collisions. This helps prevent planning configurations which might cause self collisions. For this approach, we defined the equation of sphere with radius $r = 0.06$ centered at the joints[3].

   $$value = (x - x_1)^2 + (y - y_1)^2 + (z - z_1)^2 - r^2$$

   Then by substituting the value of (x_1, y_1, z_1) in the equation mentioned above, we used if-else to check if the point lies inside the sphere or outside. Using that we defined self collisions. Along with this, we also implemented the parametric equation[3] of sphere:

   $$(x, y, z) = (x_1 + \rho cos\theta sin\phi, y_1 + \rho sin\theta sin\phi, z_1 + \rho cos\phi)$$

   Where ρ is the constant radius and (x_1, y_1, z_1) represents the coordinates of the point around which the sphere is centered. Then by using convenient loops, we obtained the coordinates of 100 points on one sphere around a joint and the corresponding 100 points on the next joint. Lines joining these 100 points help using the same collision detection technique discussed earlier to compensate for the arm width, thickness. It helps approximate the arm as a capsule made up of 100 lines.

5. **Path tracing:**
   Once the collision check returns true and valid, the algorithm then proceeds to check the interpolation collision check again between newest node and the goal point. This tells us whether the path between the newest point and the goal is free of collisions and obstacles. If it is indeed free of all collisions, then the boolean path flag resets and the loop ends. Then the algorithm opens the tree and scours the tree by finding the parent index and the corresponding node associated with it. It iterates to find the node history till it reaches the start point q configuration. In this way a well defined path vector is generated as an output.

2 EVALUATION

1. **Testing the main tree algorithm** We started by keeping the structure of tree same, but tracking the end effector position. This can be considered to be a global tracker by considering the robot to be a point object in the 3D space trying to navigate across obstacles. We were able to test and demonstrate that they navigated across the obstacles and avoided them.

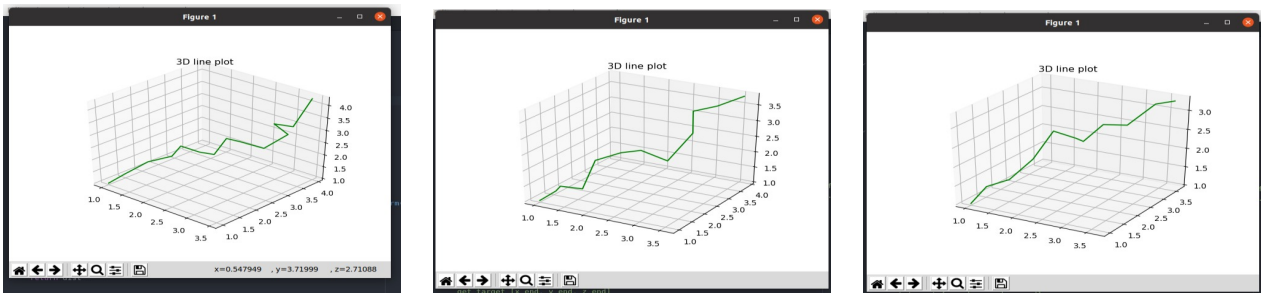

Figure 2: Same start, end points but different path.

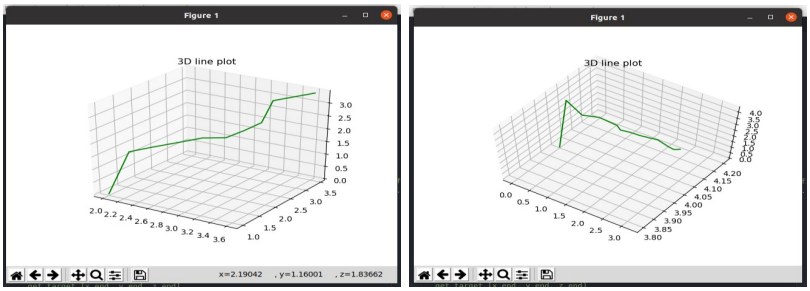

Figure 3: Same end point, but different start points

The plots above have XY plane as the plane seen from above and vertical axis is z axis.
For all plots, the start point is the bottom left and end point is the top right. (All axes measure distance)

2. **Performance statistics:**
In order to do the tests, map 1 and map 2 were used. This was primarily due to how map 1 is a planar obstacle close to the starting point while map 2 gives a scenario where the path is more distinct. In both cases, A-star should be viable, especially for map 3. For both these maps, the rrt planner was able to find a route each time. However, the path that was traced each time was not the same. Same is demonstarted through the tests shown above.
This is not the same with A-star, where the path that was traced appears to be similar each time. We found out that running the same test repeatedly 4 times in a row yielded the same path of A* with execution times as 56.77, 58.93, 63.38, 65.31 sec for the Map 2. This shows that the repeatability is quite good as same path is returned.
For RRT, all of the paths would get executed within seconds. More complex obstacles would lead to longer run times, and after adding contingencies for -
1) Self collision
2) Ground collision
3) Obstacle collision using spheres and cylinder approach
We introduced a lot of for loops in detect collision function. So the execution times ranged between 30 sec to 2 min depending on how challenging the RRT planning problem was.

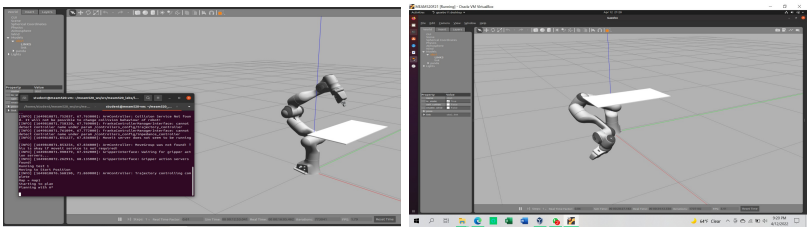

Figure 4: Planning A and RRT with Map1*

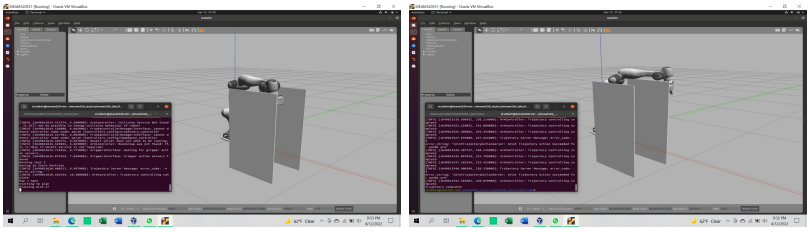

Figure 5: Planning A and RRT with Map2*

We also observe that A* takes 1200-1400 sec to run on Map2. For more complex maps and test cases we designed, where we varied the start , end configuration we found that planning took longer, or would final. In such cases, RRT would return good values. We used 1 test for each map with a starting position close to the obstacles. The average run time for rrt and astar is as provided:

RRT:
Test 1: 6.8s average
Test 2: 16.4s average
A*:
Test 1: 50sec
Test 2: 40 min average

Environment issues:
A* cant converge well with complex paths. It takes a long time to run. In these cases, RRT performs well.

3. **How often does your path planner succeed?**
   It appears as though the A-star struggles to find a path if the starting point is too close to the boundary of the obstacle and there is no apparent path forward. It is however good at finding a path should a distinct path be available, and easily avoids the obstacle boundaries.
   Although the rrt seems much less efficient in a scenario where an "obvious path" is present, it is capable of finding a path in tougher scenarios due to how the nodes are random sampled. That being said, the repeatability is an issue. Gradescope actually helped us quite a lot for this evaluation. When we reached a coding efficiency where the algorithm would return a path 100% of times, with the same code, we we were able to find out that RRT would succeed in giving a valid path every single time. There was a 5-15% chance of it returning path=[ ] for difficult and more complex obstacle scenarios that we fed.
   After extensive testing, we were able to conclude that if we added large number of obstacles, then the probability of the algorithm returning a valid path would reduce. In this case, increasing the number of nodes, or increasing the iterations helped increase the accuracy quite a bit.

3 ANALYSIS

1. **Ideal conditions for planners**
   The rrt planner is good for conditions where an obvious path is not clearly identifiable. The randomness of the approach helps here. However, the lack of an algorithm means that it's less ideal for a situation where an obvious mathematical path exists; such as a straight line. It consumes more resources and time to identify this path while an algorithm-based path like A-star can efficiently find it.

   While an A* would merely follow the shortest path available, RRT introduces the risk of having random intermediate configurations in between which returns a jagged path. This however helps us find paths round obstacles in th emost difficult configuraitons like MAP3 or MAP4 very easily.

2. **Issues faced with RRT**
   There were a number of issues we faced when making the RRT code. To begin with, even though the program was mathematically accurate for a point mass system, the existence of dimensions beyond the first resulted in several collisions. We tried to first face this issue by increasing the boundary of the obstacle to be more than it really was. However, such a method is truly inefficient as a trade-off needs to be reached between ambitious paths, and conservative/safe planning strategies. Too much deviation on either side leads to failure.
   To model our rrt more alike to the real world, we decided to introduce the dimensions into the system. To do this, we used spheres defined at each joint to help us detect self-collisions. Then, we converted each link into a capsule with a radius equal to the cross section of the largest link. This gave us a performance which was much closer to the real world.
   However, we still faced issues of the nodes being too random at times, resulting in a path too roundabout. This did not align with what we had in mind, which was to find a path as close to linearity as possible. However, this is not very surprising as the sampled configurations do indeed depend a lot on RNG.
   Also, if the chosen path is too close to the table, many times, the simulation would abort because it would venture too close. At the same time, increasing the "safety bubble" or a buffer would lead to the solver failing to return a path many times.

3. **Recommended Changes**
   We have dealt with a lot of contingencies, but indeed, a lot could be improved further -
   1) Implement RRT*- Introduce a cost function which decreases th epath length, and rewires the node connections to find a much more optimal and less random path.
   2) Another method is to reduce randomness in sampling points by actually sampling in a specific direction i.e. sampling bias so more points are sampled around what we'd expect to be the shortest route; often a straight line connecting the start and end.
   3) Introduce a much more efficient collision model. Improvements include, better approximation of the links, more full-proof contingencies. Also, the code could be cleaned up further by changing the testing parameters for the capsules to joint-specific and link-specific so it's easier to localize a collision. The code in its current state does not use the capsules completely for detecting self collision but the segregation of the values to joint-specific and link-specific can help in identifying self-collisions using the capsule alone, rather than relying on two separate models to detect each collision, thereby reducing computation time.
   4) Another improvement is to reduce the number of iterative loops. This will help add more nodes to the tree, have faster execution times and better convergence onto a solution.

4. **Differences between RRT and A***
   This topic was visited earlier and is iterated here. The most significant difference[4] is that A* algorithm is a well-known method in motion planning problems which can find the optimum path between two point in a finite time. In contrast the RRT family algorithm, by random sampling from the environment, converges to a collision-free path.
   1) The benefit is that due to the random initialization RRT is a really fast solver. A* on the other hand creates a grid and searches incrementally until the goal is reached.
   The pitfalls of RRT here, however, include its lack of optimization, incapability to repeat the same result multiple times and inefficient resource consumption.
   2) A* takes a really long time to converge especially for difficult obstacles and configurations. Execution times range from 20 min to 1 hour on an Intel i7-11th gen H with VM 8GB allocated RAM on 4 cores. As against, RRT finds it within seconds. But the path generated by A* is optimum and it remains the same even after iterating many times.A* algorithm's heuristic always directs it towards the direction of the goal. RRT on the other hand, although quick to give results, gives random configurations.

4 REFERENCES

1. Robot Modelling and control, 4^{th} edition - Mark W. Spong, Seth Hutchinson, M. Vidyasagar
2. Sampling-based Algorithms for Optimal Motion Planning, Sertac Karaman and Emilio Frazzoli, 5 May 2011
3. Wikipedia: https://en.wikipedia.org/wiki/Sphere
4. Comparison Between A* and RRT Algorithms for 3D UAV Path Planning , Christian Zammit and Erik-Jan van Kampen