A Survey of Motion Planning for Robotic Arms Milestone Report

Tony Chen (chensiyu), Holly Liang (xuejiao), Zhihan Jiang (zhihanj)

1. Introduction

The goal of this project is to make robotic arms move to a predefined location without colliding with any obstacles in its trajectory. This has broad industry applications, such as a vehicle engine-oil-changing robot arm where the arm must avoid colliding with the car itself. We are going to experiment with different approaches to develop a robust algorithm for such motion planning.

2. Methodology

Our topology of the project design is shown in figure 1. The milestone will discuss in detail how we research on each part of the design.

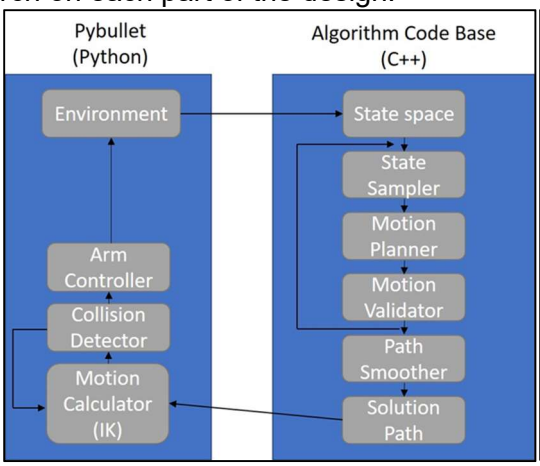

Figure 1. Topology of Motion Planning

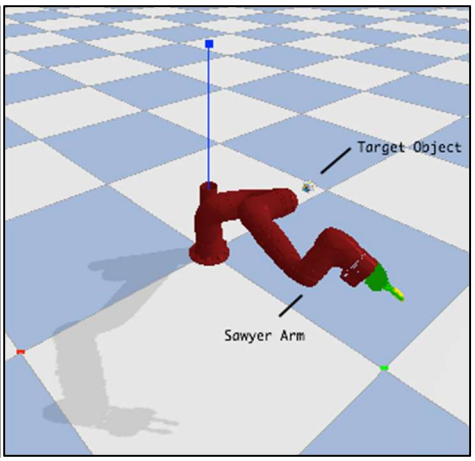

Figure 2. Pybullet Robotic Arm Environment

2.1 Pybullet

Pybullet[1] is an easy-to-use Python module for physics simulation for robotics, games, visual effects and machine learning, in which we can load articulated bodies from URDF. Below is how we setup the test environment:

Robot arm: Our urdf file contains a 7-joint robotic arm. We took out the base of the robot since we assume that the robot is not moving. Figure 2 shows the model after loading into Pybullet.

Environment: one small cube to indicate the location of the target cube, with the ground as an obstacle.

Pybullet has many inherent functions for the robot control. We have a few wrapper functions for our implementation:

setup(gravity, timeStep, urdfFile): set up the environment and robot with provided urdf files.

resetPos(val): reset to an initial pose.

readQ(): read every joint position of the robot

getTargetJointPosition(xyz_pos): Given a cartesian space location, find joint positions for the robot to reach this location by solving the inverse kinematics.

getTargetPosition(): Track the position of the target in cartesian space.

checkCollision(): check whether the robot is colliding with another object.

moveTo(joint_position): move the robot to a defined posture.

2.2 Motion/path planning

For the robot arm, path planning in the high dimensional space can be extremely time consuming with the existence of obstacles. Thus, finding an efficient planning algorithm is the key. For the milestone, we focused on researching 3 planning algorithms from 2 general categories: A*, which is a node-based algorithm, PRM[2] and RRT[3], which are sampling based algorithms.

2.2.1 A*:

We first experiment with a simple A* algorithm. To make it extendable to the n-dimensional space, we defined an "object" class to represent the agent (the robotic arm), the target, and any obstacle in the discretized space. The object specifies its vertices and edges using coordinates, and hence the location is implicitly determined. The class also provides two other important methods: distance(self, obj), that measures its distance to another object, and collided(self, obj) to evaluate if it has collided with another object. Such objects contain all the information we need in the state to find the shortest path forward. The model is thus defined as below:

- startState(): an object called *agent* in the starting position
- isEnd(s): True if s reaches the target
- actions(s): adding or deducting a unit step from one of the dimensions
- succ(s, a): the resulting (moved) object of s taking an action a, given it does not collide with an *obstacle* or go out-of-bound
- h(s): euclidean distance from s to the target
- cost(s, a): 1 + h(succ(s,a)) - h(s)

In a simple 2-D plane using a 1x1 point-object, we plot the results as shown below:

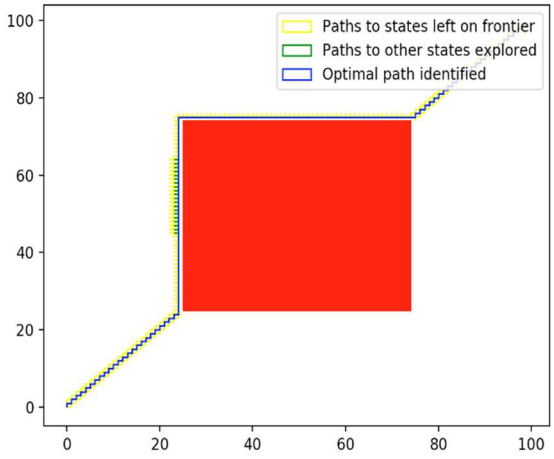

Figure 3. A* algorithm visualization

The solution path is not a smooth line for two reasons: 1) in general, the agent will first attempt to greedily explore states that are closer to the target; 2) we are only allowing movements orthogonal to the axes in order to limit the number of successors from a state. The tradeoff here is creating fewer successors versus creating a smoother path. One way to solve this problem is to use a proper path smoothing algorithm. We will discuss such an algorithm in the next section.

Finally, the same algorithm also applies to irregular objects in higher-dimensional spaces. In Pybullet, we will study how rotations of the object come into play.

2.2.2 RRT, PRM, State Sampling:

Sampling based algorithms such as PRM and RRT rely on discretizing and sampling the state space. The general steps of such algorithms are:
1) Generate random vertices by certain constraints,
2) try to connect edges between vertices that don't violate the constraints (collision avoidance),
3) try to find a path from the start to the goal. If path not found, repeat 1) to 3).
The non-trivial difference between these 2 algorithms is that, PRM samples vertices randomly across the graph, with the start and the goal as initial vertices, while RRT samples vertices from the start point, and biases the tree to grow towards the goal.
Since it's hard to tell which algorithm is better by definition, we decide to implement these 2 algorithms in both 2D and 3D spaces, and compare them based on various metrics such as: number of states created (which directly affects the dynamic run space size), runtime, and smoothness (through path visualization).
We implement our algorithms based on OMPL[4], a motion planning library (C++) developed by Rice University. We generate our path visualization in MATLAB using the output from the C++ code.

2.2.2.1 2D Path Planning

We first implement a simple path search in 2D spaces, using both PRM and RRT. The problem is set up to find a path in the x-y plane, from [-0.9 -0.9] to [0.9 0.9], with each dimension bounded by [-1, 1]. A square obstacle is placed in the middle of the bounded plane with side length 1.
Since the raw paths generated by PRM and RRT are connected edges between randomly sampled vertices, we need to smooth the paths. The smoothing algorithms we implement will 1) reduce redundant vertices, 2) collapse close-by vertices, 3) find shortcut paths if possible, 4) use spline interpolation to smooth the curve. We will compare the raw paths with the smoothed path in our experiment.
Figure 4 and 5 visualize the paths and edges generated by PRM and RRT. We can clearly see that, RRT samples much fewer vertices before reaching the destination (however, this might lead to a greedy algorithm in a more complicated setup) than PRM. But RRT relies heavily on the path smoothing algorithm to get a nature curve. The runtime of the algorithms in 2D is negligible (< 10^{-3} sec), and the final smoothed paths are not significantly different. We will examine more metrics in the next section.

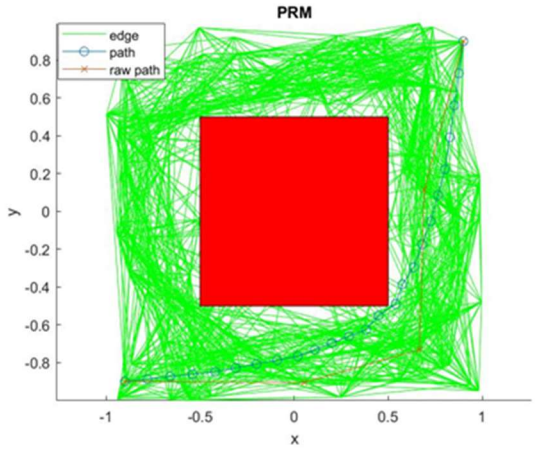

Figure 4. 2D Path Planning Using PRM

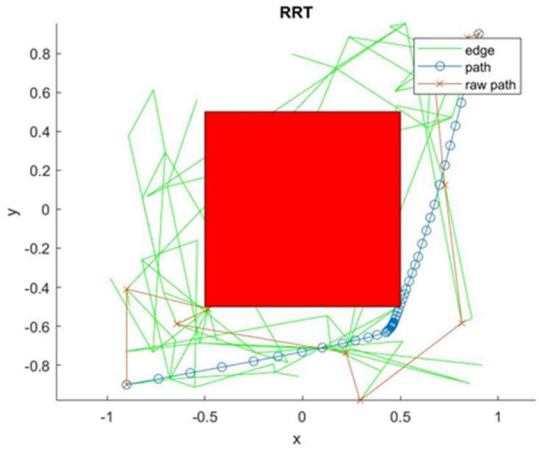

Figure 5. 2D Path Planning Using RRT

2.2.2.2 3D Path Planning:

We now move to the 3D path planning. The problem is to find a path in x-y-z plane, from [0 0 0] to [0 0 1], with each dimension bounded by [-1 1]. The obstacle is defined as when $0.25 < z < 0.5$, $|x| > 0.8$ and $|y| > 0.8$. Geometrically it means that the solution path must pass through one of the four square cubes defined by the constraints to reach the destination.

A point worth mentioning is the sampling method. The naïve method is just to sample uniformly distributed points in the graph. However, points sampled in the obstacle area are useless because edges will never be connected to them (plus, it will cost time to validate these states). In the experiment, we implement a valid state sampler, which only sample points in the valid region in the graph, and compare with the baseline.

In figure 6 and 7, all four of our experiments succeed in finding a solution path, while not breaking the constraints. We run the 4 algorithms for a reasonable amount of times and calculate the average runtime and number of states created in table 1. We see that, although RRT generates more states than PRM, it has a much lower runtime. Moreover, the valid state sampler (specific sampler) reduces the runtime and average number of states by a significant amount. However, the sampler has little help to PRM. This makes sense because PRM samples randomly across the graph, so the possibility of vertices in obstacle area is low, and an invalid point will not harm the runtime anyway. Whereas RRT generates lots of invalid states when sampling near the obstacle region. Using specific sampler solves this problem by guiding points carefully through the obstacles.

Overall, we conclude that PRM is a better algorithm for the lower dimensional space, and RRT works better in the higher dimensional space. The valid state sampler and the path smooth algorithm also help with RRT to create a better solution. Nonetheless, there are variants of PRM and RRT, like PRM*[2], RRT*[5], VF-RRT[6]. We will research more in the next phase of the project.

	PRM		RRT	
	Uniform	**Specific**	**Uniform**	**Specific**
Runtime	**5.39**	**5.74**	**1.89**	**1.12**
States	**95.25**	**103.5**	**399**	**194.25**

Table 1. Runtime and number of states created by PRM and RRT, using different sampler

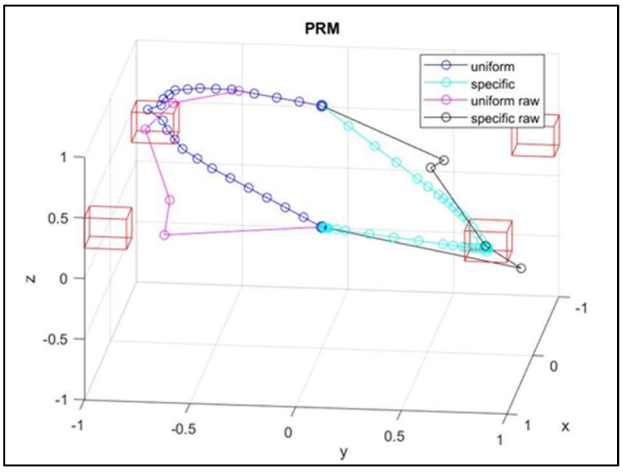

Figure 6. 3D Path Planning using PRM

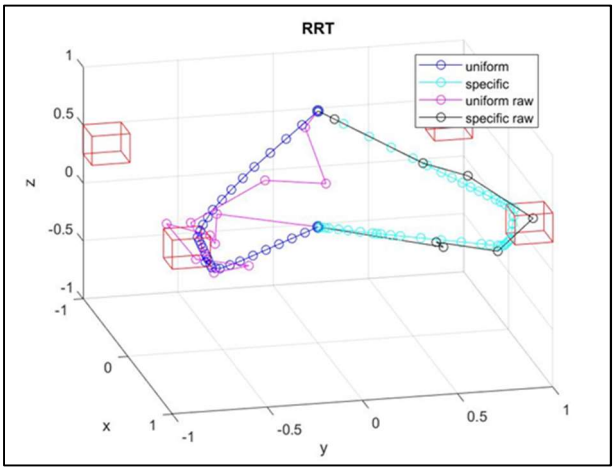

Figure 7. 3D Path Planning using RRT

2.2.3 RL:

We also try to implement a Q-learning algorithm defined as below:

State Definition: We discretize each joint angle from a continuous [0, π] into steps of 0.1. Each joint can take on π/0.1 ~= 31 different values. Since Sawyer arm has 7 revolute joints, there are a total of $\left(\frac{\pi}{0.1}\right)^7$ ~= 3^{10} different states.

Action Definition: each joint can either move forward, backward, or stay the same. Our action vector has length 7, with each entry as the value of the corresponding joint movement (-0.1: move back, 0.1: move forward, 0: stay). We assume that only one joint is going to move at a time (to control the size of the action space), thus our action space looks like:
action_space = [[0.1,0,0,0,0,0,0],...,[0,0,0,0,0,0,0.1],[-0.1,0,0,0,0,0,0],...[0,0,0,0,0,0,-0.1]]]

Reward: At first, we are only setting rewards for reaching the target position (+100) and colliding with environment (-100). If no collision occurred, we provide a small punishment (a.k.a a living cost) based on the distance from the robot end-effector to the destination. More specifically, the reward for any intermediate state equals:

$$-\left(position_{end-effector} - position_{target}\right)^2$$

Q-Learning steps: we start with Q matrix (size of #states * #actions) initialized to all -1000000, and robot arm start in the same starting position. In each iteration, we move robot arm by giving it a state of 7 numbers, with each number representing the angle of a joint in radians, and use the softmax to determine which action to take (87% chance take the action that maximizes Q_next, 1% chance other states). Then update Q using this equation, where alpha (learning rate) and gamma (discount factor) are both set to 0.9 to enable some extent of consideration for previous states:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left(r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right)$$

If a target/obstacle is hit, a terminal state is reached. When this occurs, we then start again from an initial state. We will terate until Q values converge. However, we haven't reach convergence yet due to parameter setup and runtime issues for the time being.

3. Next step

1. Apply PRM, RRT and A* algorithms in Pybullet to start guiding the robotic arm. For RRT we will try to generate multiple solution paths, and discard the ones that will potentially make the robot arm collide with the obstacles.
2. Tune parameters for a better convergence of Q-learning. We will also try SARSA and SARSA-Lambda to improve the converging speed.
3. Quantitatively analyze different approaches, and propose the best solution.
4. Add more complicated obstacles to fully evaluate these approaches.

4. Conclusion

When collision is not considered, and under a known and static environment, PRM and RRT will provide stable and smooth solutions. Meanwhile, Q-learning would be more flexible if there exist more uncertainties (such as noise or randomness in the location of the target). Moreover, the RL method does not require huge storage of the state space. However, Q-learning subjects to a slow convergence rate due to the large exploration space. A more comprehensive research on how to achieve better convergence and faster runtime needs to be conducted in the next phase.

Reference:

[1] Erwin Coumans, Yunfei Bai, Pybullet quickstart guide, 2017, https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdAvK3dsihxVUA/edit

[2] L.E. Kavraki, P.Švestka, J.-C. Latombe, and M.H. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, IEEE Trans. on Robotics and Automation, vol. 12, pp. 566–580, Aug. 1996.

[3] J. Kuffner and S.M. LaValle, RRT-connect: An efficient approach to single-query path planning, in Proc. 2000 IEEE Intl. Conf. on Robotics and Automation, pp. 995–1001, Apr. 2000.

[4] Ioan A. Șucan, Mark Moll, Lydia E. Kavraki, The Open Motion Planning Library, IEEE Robotics & Automation Magazine, 19(4):72–82, December 2012. http://ompl.kavrakilab.org

[5] S. Karaman and E. Frazzoli, Sampling-based Algorithms for Optimal Motion Planning, International Journal of Robotics Research, Vol 30, No 7, 2011. http://arxiv.org/abs/1105.1186

[6] I. Ko, B. Kim, and F. C. Park, Randomized path planning on vector fields, Intl. J. of Robotics Research, 33(13), pp. 1664–1682, 2014. DOI: 10.1177/0278364914545812