Robot Arm Planning Using Rapidly-Exploring Random Trees

Thomas Jansen
260554029

Paper Used: “Rapidly-Exploring Random Trees: A New Tool for Path Planning”

Introduction

Path planning has always been one of the difficult pieces in the jigsaw puzzle that is the creation of intelligent, mobile machines. In very low-dimensional scenarios, techniques such as Voronoi diagrams or visibility graphs are excellent ways of quickly creating paths in simple worlds. Random techniques such as probabilistic roadmaps [1] are a good way to extend path planning to higher dimensions with relative ease. But even with probabilistic roadmaps, the equations involved in connecting nodes and checking for collisions in both configuration and Cartesian space can quickly become very difficult and very expensive.

RRT's (Rapidly-Exploring Random Trees) [2] are another random path-planning option that are much easier to handle in very high-dimensional problems than probabilistic roadmaps. As the name implies, RRT's are trees that expand quickly and explore the entirety of configuration space. Paths can be found by following the nodes along the branches of these trees, and those paths can bring their agent from one point in their configuration space to another with far less computation than in many other path-planning algorithms.

The basis of the RRT algorithm is that it picks a random point in free space, then attempts to move a little towards that point from the closest point to it on the tree, and then repeats. If ever there is an obstacle in the way, the new point will simply never be made in the direction through that obstacle, and so that option will be discarded. Eventually, the tree will wrap around the obstacle via extending to other random points in space that do not have that obstacle in the way. The contrived example shown in figure one exemplifies this phenomenon.

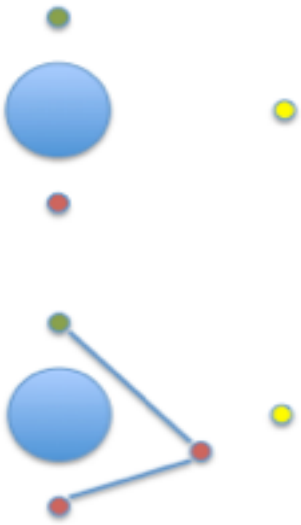

Fig. 1

Imagine the setup in figure one where the red dot at the bottom represents the root node, the green dot represents the goal node, the yellow dot represents a point in free space, and the blue circle represents an obstacle. If the tree only ever attempted to go directly to the goal, it would never be able to because of the presence of the obstacle. However, if it had randomly picked the yellow dot as a location to extend to first, it would be able to build the tree around the obstacle and get to the goal as is shown in the second image.

In this paper, I will apply an RRT through two different methods to a robot arm with four degrees of freedom. This robot

arm, pictured in figure two, has a main axis of rotation (labeled axis 1) at its base at the origin of the world frame along the positive y-axis (the up direction). This axis spins with the main sphere on which the rest of the arm is situated; all of the other degrees of freedom depend on its orientation. The next axis of rotation (labeled axis 2) is set perpendicular to the first axis, and rotates the first arm joint with respect to the origin as well. The exact direction of this axis of rotation is dependent on the main axis, but it will always be perpendicular to the y-axis. The next axis of rotation (labeled axis 3) is set at the second arm joint and is aligned with the second axis of rotation. Its position in space depends on the orientations of the first two axes. The final degree of freedom is a prismatic joint (labeled axis 4) that extends from the end of the arm's second cylinder and effectively represents the end effector for the arm. Its axis of translation is aligned with the arm's second cylinder and depends on the orientation of all the other joints.

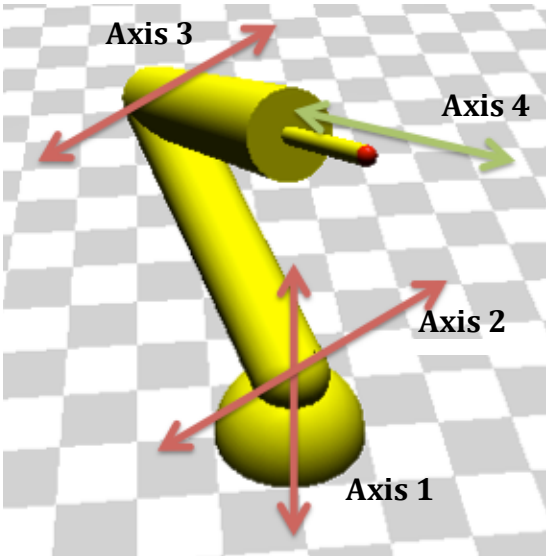

Fig. 2

For the rest of the paper, the absolute angles of rotation around the first three axes will be referred to respectively as θ_1, θ_2, and θ_3. The distance along the final translational axis will be referred to as α (if α is zero, the end effector is entirely inside the arm's second cylinder). The different parts of the robot arm will be referred to in the following manner: The main sphere (the sphere that sits on the plane), the first cylinder (that comes out of the main sphere), the second cylinder (that is attached to the first cylinder at the third axis), and the end-effector (that slides in and out of the second cylinder).

It is also important to note that using ROS, Gazebo, and Moveit was the original intention for this project. However, after many trials and tribulations, and after finally having all the appropriate packages installed into the Trottier machines, all the computers crashed (possibly due to those packages that I installed) and all of the newly installed packages were wiped from the memory. It was then that I decided that it would be a safer bet to work using my own tools from home.

Discussion of Background and Related Work

As described by LaValle [2], RRT's are a very simple idea that can be extrapolated to many diverse problems in the realm of path-planning and robotics. The basic algorithm for all RRT's comes down to a few simple steps: find a random state, find the nearest node on the tree to that random state, move towards the random state from that node, create a new node at the new location, connect it to the previous node, repeat. The details as to how one would go about moving towards a state, or creating new nodes varies from problem to problem and from robot to robot, but the same general outline can be followed for any path-planning situation. And as LaValle's [2] paper describes, the path that the algorithm finds usually falls between a factor of 1.3 to 2.0 times the length of the optimal path.

Given enough time, the algorithm also guarantees to eventually find a path if one exists. Because the algorithm explores new space randomly, every reachable point in free space will eventually be reached. In addition to this, the new nodes will eventually become uniformly distributed in free space. This is due to the fact that at a certain point in the algorithm, all remaining points in free space will be close enough to an existing node in the tree that it can just be added to the tree directly. And assuming that the randomly chosen points in free space are chosen according to a uniform distribution, the new nodes will be added to the tree in a uniform distribution as well [2].

In other related work, LaValle has continued to explore the possibilities of using various altered RRT techniques for path-planning. In one of his more recent works, he has created an RRT algorithm that is resolution complete, meaning that the algorithm guarantees to find a path in a finite number of iterations, which is an idea that had not previously been tried. They were able to build the algorithm by combining a few techniques, namely accessibility graphs, discretization of state space, and Lipschitz conditions [3].

Method

Let's begin by introduction the User Interface for the robot arm (figures three and four). The first slider, titled "Min Distance to Goal", is how close the end effector needs to get to the goal location in order for the tree to stop exploring and consider the final node to be a success. There are no units associated with the numbers, but to understand the relative size of all the values, the robot arm at full extension reaches 1.5 units above the origin. The slider title "Max Loops" describes the maximum amount of loops that the algorithm will take before it terminates without a solution. In some scenarios the tree may take incredibly

long to find a solution or a solution may not exist so it is important to pick a cap to the number of loops that will take a reasonable amount of time to terminate. The "Random Target Location" button sets the goal to a random location in free space, or if a specific location is preferred, the next three sliders can move the goal along any of the X, Y, or Z axes.

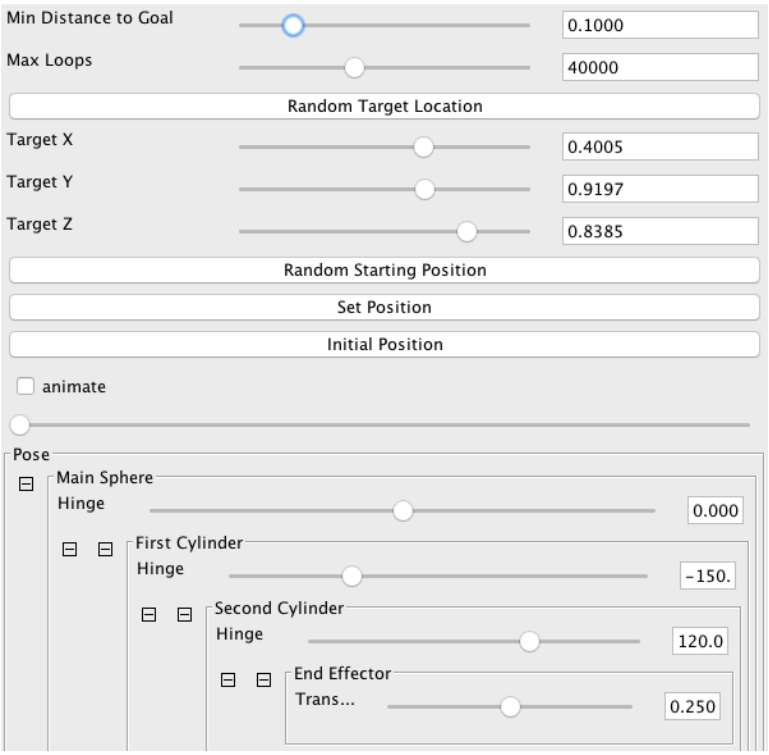

Fig. 3

The "Random Starting Position" button sets all of the degrees of freedom of the robot to random values that do not intersect with any existing obstacles. The "Set Position" and "Initial Position" buttons allow the user to set the robots position to whatever position it is currently in, and then to reset the robot to that position if desired. If a tree exists and a path from the base node to the goal node exists, the animate button will animate the movement of the robot arm along that path, interpolating all the joint values between nodes. The slider beneath the animate button allows the user to arbitrarily move the robot along its path as they please. The next four sliders allow the user to change the robot's degrees of freedom to the desired states. The first three are measured in degrees seeing as those joints are rotating hinges, while the last slider is measured in units for the distance α that the prismatic joint is reaching out.

The rest of the user interface can be found in figure 4. The "Add Obstacle" button allows the user to add a sphere in a random location as an obstacle in the world. The "Randomize Obstacles" button rearranges all existing obstacles randomly. The "Next Sphere Size" slider decides the radius of the next sphere obstacle that will be added. The epsilon slider describes the minimum distance that the robot has to be from obstacles, when set to zero the robot can touch the surface of obstacles. The "Clear Obstacles" button, as it suggests, clears all existing obstacles. The next four sliders give the maximum change in each degree of freedom that can occur between two neighboring nodes. The values are set

upon initialization of the application to one degree of change for each rotating joint, and 0.025 units for the prismatic joint. To create the tree, two different methods were implemented as the buttons suggest, and the second method relies on the "Threshold for Prismatic" slider which will be explained in the section about method two. The last button visualizes the tree by placing a sphere at the location of where the tip of the end effector would be given the states at that node. Cylinders are used to connect all of the nodes to their respective children.

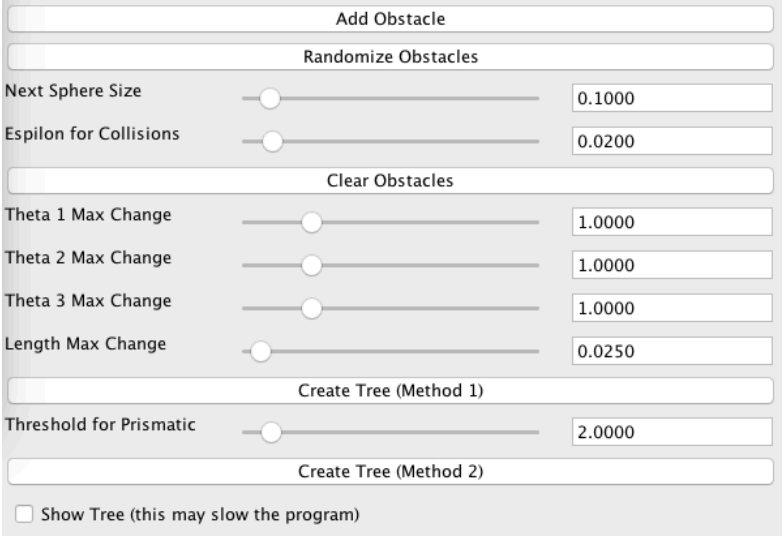

Fig. 4

To find the location of the end-effector in Cartesian space, forward kinematics had to be used given the known properties of the robot as well as the current state of each of its degrees of freedom. Spherical coordinates are related to Cartesian coordinates via the equations in figure five (where θ is the angle in the x-y plane, and ϕ is the angle out of that plane).

$$r = \sqrt{x^2 + y^2 + z^2} \quad \theta = tan^{-1}\left(\frac{x}{z}\right) \quad \phi = cos^{-1}\left(\frac{y}{r}\right)$$

Fig. 5

By simply plugging the known values into all of these equations, we can find the Cartesian coordinates at the end of the arm's first cylinder (since that point essentially represents the point on a sphere with a radius of that first cylinder's height). The point at the end of the second cylinder is at the surface of yet another sphere, however this time its origin is centered at the end of the first cylinder. The ϕ angle this time is an addition of both θ_2 and θ_3 since their axes are aligned, and the angle at θ_2 affects θ_3 directly. Then, by adding the locations of the point at the end of the first cylinder with respect to the origin and the point at the end of the second cylinder with respect to the point at the end of the first cylinder, the point at the end of the second cylinder can be found with respect to the origin.

The point at the end of the end effector can be found similarly by adding α to the length of the arm's second cylinder when calculating the equation for the second sphere.

The nodes that compose the tree are made up of several different object variables. Each node, when created, has a set of its own immutable values for each degree of freedom of the robot. It also contains the position in Cartesian space of the tip of the end effector, a list of children nodes, as well as its parent node labeled dad. Within the node class is a traversal algorithm (figure six) that takes as input a location in Cartesian space, and returns the node in the tree rooted at the current node with the closest end-effector-location to the given location. It works by first calculating the distance between the given (target) location, and the location where the current node is situated. Then for each of its children, it recursively traverses their sub-trees always returning the node with smallest distance to the target location. For each of its children, the algorithm replaces the current node with the best candidate based on the distance calculations from the target to the nodes.

```
Traverse (target)
        distance = target - location
        return_node = this

        for (n in children)
                temp_node = n.traverse(target)
                temp_distance = target – temp_node.location

                if (temp_distance < distance )
                        return_node = temp_node
                        distance = temp_distance

        return return_node
```

Fig. 6

When creating the tree, the base node is chosen based on the robot's current state. Then, in the trivial case, the base node is returned if it is already within the minimum distance to the goal. Otherwise, the algorithm starts generating the tree by picking a random point in space to extend the tree towards. Approximately one in ten times the goal itself is chosen as the random point, just to add a bias towards moving in the direction of the goal. A point along each of the axes is also chosen as the random point once in every fifty iterations. This is to make sure that the tree will force the robot to extend towards its outer boundaries. So in the end, one in five random positions are not, in fact, random. The algorithm then traverses the base node to find the node closest to the random point. What happens next depends on which tree generating method is used.

In the first method, instead of calculating the inverse kinematics to find how the degrees of freedom in the closest node should change to move towards the given point, each

of the θ_1, θ_2, θ_3, and α values are randomly tweaked a little bit according to the given max change values in the UI. The distance to the target point with the new values are checked against the old values, and if the new values do not bring the end-effector closer to the randomly chosen point, the changed values are discarded and tried again. This process repeats until either a closer value is found, or twenty iterations are reached. This method of randomly changing the configuration space variables may not be optimal, but it is efficient and it guarantees the new node to move closer to the random point than any other node. This can easily be proved because the traversed node is the current closest node to the random target location. Since the random movement creates a new node that is closer to that location than the current closest node, it must be the new closest node to that location out of all the nodes that currently exist.

If the new node is found to move towards the target location, it is then processed for collisions. A collidable class has a list of methods that allow for simple object intersection tests. To simplify the collision processing, shapes were restricted to planes, spheres, and cylinders, with various collision methods between all of these shapes. A simple, example sphere-sphere collision detection algorithm can be seen in figure seven. Here, l represents the center of the given sphere in Cartesian coordinates, and r represents the radius of the given sphere. Location and radius, meanwhile, represent those values for the collidable in this instance of the class. Epsilon is how far away from the object the given sphere needs to be for it not to count as a collision. As can be seen, the only verification for there not to be a collision between two spheres is that their centers are sufficiently far away from each other. If the addition of their two radiuses plus the additional epsilon is smaller than their physical distance, the two objects are not intersecting and therefore they are not colliding. The math is a bit more intensive for collisions involving cylinders, but the underlying principles of intersecting geometry are the same.

```
SphereSphere(l,  r, epsilon) {

        Distance = l - location

        if (Distance < r + radius + epsilon)
                return true
        return false
}
```

Fig. 7

If the new node is found to be closer to the random target location, and it doesn't collide with any of the existing collidable objects, it is added to the tree as a child of the current closest node to the target. The algorithm then checks the proximity of this new node

to the goal, and if it is within the minimum distance set by the "Min Distance to Goal" variable, the tree terminates and returns that node.

The second method uses many of the same ideas, but differs in the how the prismatic joint randomly changes. After the algorithm has chosen a random position in space, the three rotating degrees of freedom are randomly adjusted just as in the first method, but the manner in which the prismatic joint gets adjusted is dependent on certain conditions. Before the α is changed, the angle between the second cylinder's orientation in the current node and orientation that it should have to be facing the random target position in space is checked against the threshold angle value for the prismatic joint. The angle is checked using the equation in figure eight, which describes the angle between two vectors. Then, if the angle is within the threshold and there is no obstacle in the path of the prismatic joint, the prismatic joint is allowed to move purely randomly. If these conditions aren't met, the prismatic joint is allowed only to retract into the second cylinder.

$$\cos(\theta) = \frac{\vec{x} \cdot \vec{y}}{\|\vec{x}\| \, \|\vec{y}\|}$$

Fig. 8

These conditions were chosen for the prismatic joint after observing the behavior of the robot arm in many test case scenarios. Because movement of the prismatic joint allows the robot to easily alter the end-effector's location in Cartesian space, the tree created by the first method tended to have many jerky movements from the prismatic joint. And since movement in and out of the arm's second cylinder generally didn't help the tree explore new areas in the rotating angles of the arm, it was reasoned that restricting the prismatic joints movement until it was aligned with the goal would help the arm focus more on its orientation towards the goal rather than on the physical location of the end-effector.

Finally, after the tree is generated using either method, it returns the final node. Then, following the final node's parents up to the root of the tree creates a path from the base node to the final node. This path describes the motion of the robot to the goal, with all the values between the nodes simply interpolated from their values in the previous node to the next one.

Experimental Results

To test the two different tree generating methods, six different test scenarios were created using random starting positions, target locations, and obstacle locations. Three test scenarios had no obstacles, and were made incrementally more difficult from easy to medium to hard. The starting position for the robot arm in the medium difficulty scenario

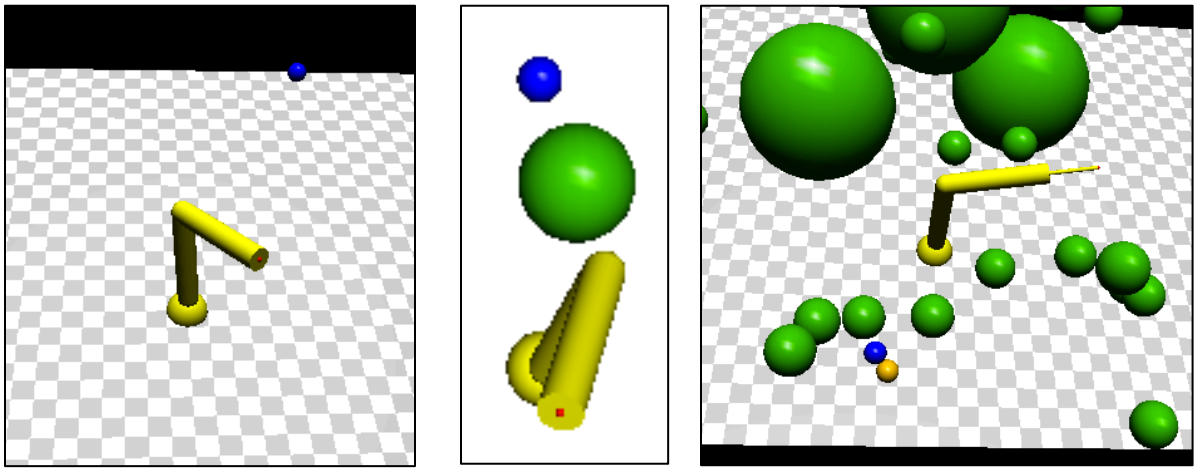

Fig. 9

with no obstacles is displayed in the left of figure nine. The blue sphere in the distance represents the goal location. Three test scenarios were then also created with obstacles. These tests were also incrementally made more difficult, with the setups for the easy and hard tests displayed in the middle and right of figure nine respectively. The green spheres represent the obstacles that the arm has to avoid when attempting to reach the goal (disregard the orange sphere, it just represents the position of the final node after a tree was created in one of the test cases). Trees were generated for each of the test scenarios until either ten successful trees were made or the algorithm failed ten times. All of the variables for the tree generation were left to their default values as can be seen in the UI images in figures three and four.

The averaged results for the test scenarios without obstacles are displayed in figure ten, while the averaged results for the test scenarios with obstacles are displayed in figure eleven. The green, yellow, and red texts represent easy, medium, and hard test scenarios respectively. In each scenario, the average number of nodes along the path, total nodes generated, and total loops that the algorithm took were recorded for both methods.

	Path	Nodes	Loops	Fails (Successes)
Method 1 (Test 1)	146.6	1603.0	2059.3	0 (10)
Method 2 (Test 1)	195.0	1654.3	2075.4	0 (10)
Method 1 (Test 2)	487.4	10358.4	14627.0	3 (10)
Method 2 (Test 2)	485.2	10523.5	13508.0	1 (10)
Method 1 (Test 3)	593.8	11500.7	17714.6	4 (10)
Method 2 (Test 3)	745.7	10886.1	15307.5	5 (10)

Fig. 10

	Path	Nodes	Loops	Fails (Successes)
Method 1 (Test 4)	232.4	3083.7	4502.9	0 (10)
Method 2 (Test 4)	190.2	1398.7	1795.7	0 (10)
Method 1 (Test 5)	385.8	6760.7	9000.0	0 (10)
Method 2 (Test 5)	649.4	9258.1	13196.0	0 (10)
Method 1 (Test 6)	529.4	11389.0	20456.0	10 (5)
Method 2 (Test 6)	703.3	15739.7	25516.7	10 (3)

Fig. 11

By looking at the results of the easy and medium difficulty obstacle-free scenarios, it can be seen that the two methods performed approximately the same. What's interesting is that the hard obstacle-free scenario generally forced the second method to find a longer path to the goal, while the total number of nodes and loops remained lower for the second method. This could be attributed to the fact that retracting the prismatic joint forces the algorithm to move much smaller distances in between nodes, which would create a longer path. On the other hand, moving smaller distances would restrict the tree from expanding too quickly, which is what might keep the number of loops and nodes lower for method two than for method one.

This phenomenon can be seen by looking at the trees generated by the two methods. The medium-difficulty obstacle-free scenario is shown in figure twelve. The top image is a tree created by method two, while the bottom one is a tree created by method one. As can be seen, the tree for method two is much tighter and less spread out than in method one, but this does not necessarily mean that the tree from method one is exploring more space. It is definitely exploring more Cartesian space by allowing the end effector to slide in and out as it pleases, but as far as the configuration space is concerned, it is difficult to say which tree has more variation in the values of all of the robot's degrees of freedom.

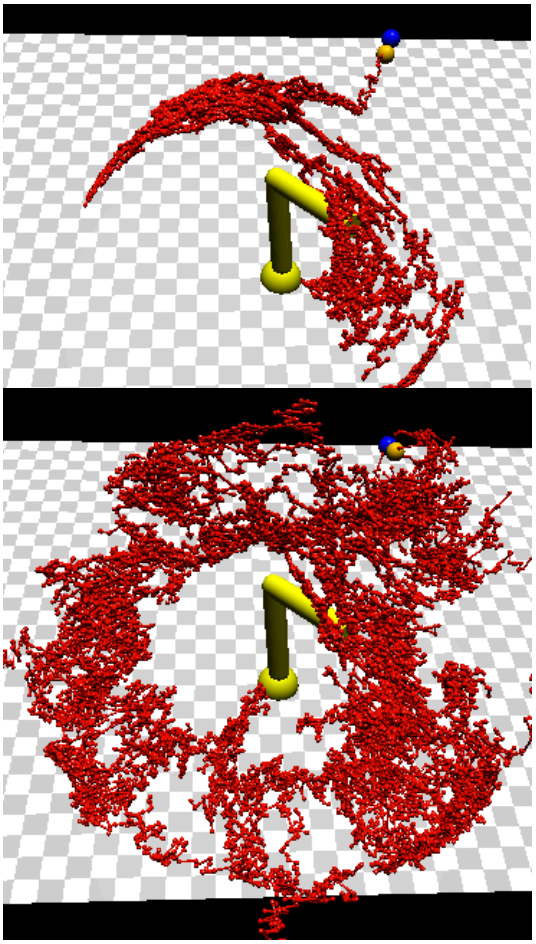

Fig. 12

In the easy obstacle scenario, method two performed far better than method one. This could be because, in this particular scenario, there was no advantage to prematurely extending the prismatic joint, which allowed the method two tree to quickly circumnavigate the obstacle and get to the goal.

Method one seemed to perform better for the rest of the obstacle scenarios. This could have been due to the fact that if method two never had a clear shot at the goal, the prismatic joint would never have been allowed to move. This would have forced the arm to try and get to the goal without using the prismatic at all, and since the first two arm cylinders were so much thicker than the end-effector, it made moving around obstacles particularly difficult, especially when the obstacles were densely packed.

Conclusion

In conclusion, it seems that the two different methods worked better under different circumstances. Method one was able to more finely move around densely distributed obstacles, while in simpler scenarios method two was able to build smoother, more direct trees that more completely explore the configuration space than the Cartesian space. The

latent procedure by which the trees were built was based mainly on finding random configurations that happen to move the robot towards its goal in Cartesian space. This allowed the algorithm to ignore what was happening in the underlying configuration space, which simplified algorithm and made it more efficient when creating nodes. However, using inverse kinematics may reduce the need to use so random movement by finding exactly how the degrees of freedom should change to move in particular directions. Although this technique would have been more costly, it would also have guaranteed the nodes to move in their intended directions on the first try.

Secondly, method two could be implemented in a way that differs it even further from method one. Currently, method two still tries to move the end-effector towards a target location by measuring distances in Cartesian space. The orientation of the robot is only taken into account when deciding the movements of the prismatic joint. But the check for whether or not a node is qualified to be added to the tree, or how the nodes are traversed in the traversal algorithm, could be altered to focus more on aligning the robot with the goal rather than trying to reach it directly. This would allow the robot to aim at the goal first, and then reach out to it with the prismatic joint at the very end.

In conclusion, it seems that the method of tree generation relies heavily on the setup of the world, and there may not truly be a best method. However, by creating several methods and by checking known information about the world, there could be a way of choosing an algorithm that works best for certain scenarios.

References

[1] Kavraki, L.e., P. Svestka, J.-C. Latombe, and M.h. Overmars. "Probabilistic Roadmaps for Path Planning in High-dimensional Configuration Spaces." *IEEE Trans. Robot. Automat. IEEE Transactions on Robotics and Automation* 12.4 (1996): 566-80. Web.

[2] LaValle, Steven M. "Rapidly-Exploring Random Trees A New Tool for Path Planning." (1998).

[3] Cheng, Peng, and S.m. Lavalle. "Resolution Complete Rapidly-exploring Random Trees." *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)* (n.d.): n. pag. Web.