HW2

November 11, 2014

1 Homework 2: Motion Planning

Robot Intelligence - Planning CS 4649/7649, Fall 2014
Instructor: Sungmoon Joo
11/10/14 ## Team 3 * Siddharth Choudhary * Varun Murali * Yosef Razin * Ruffin White

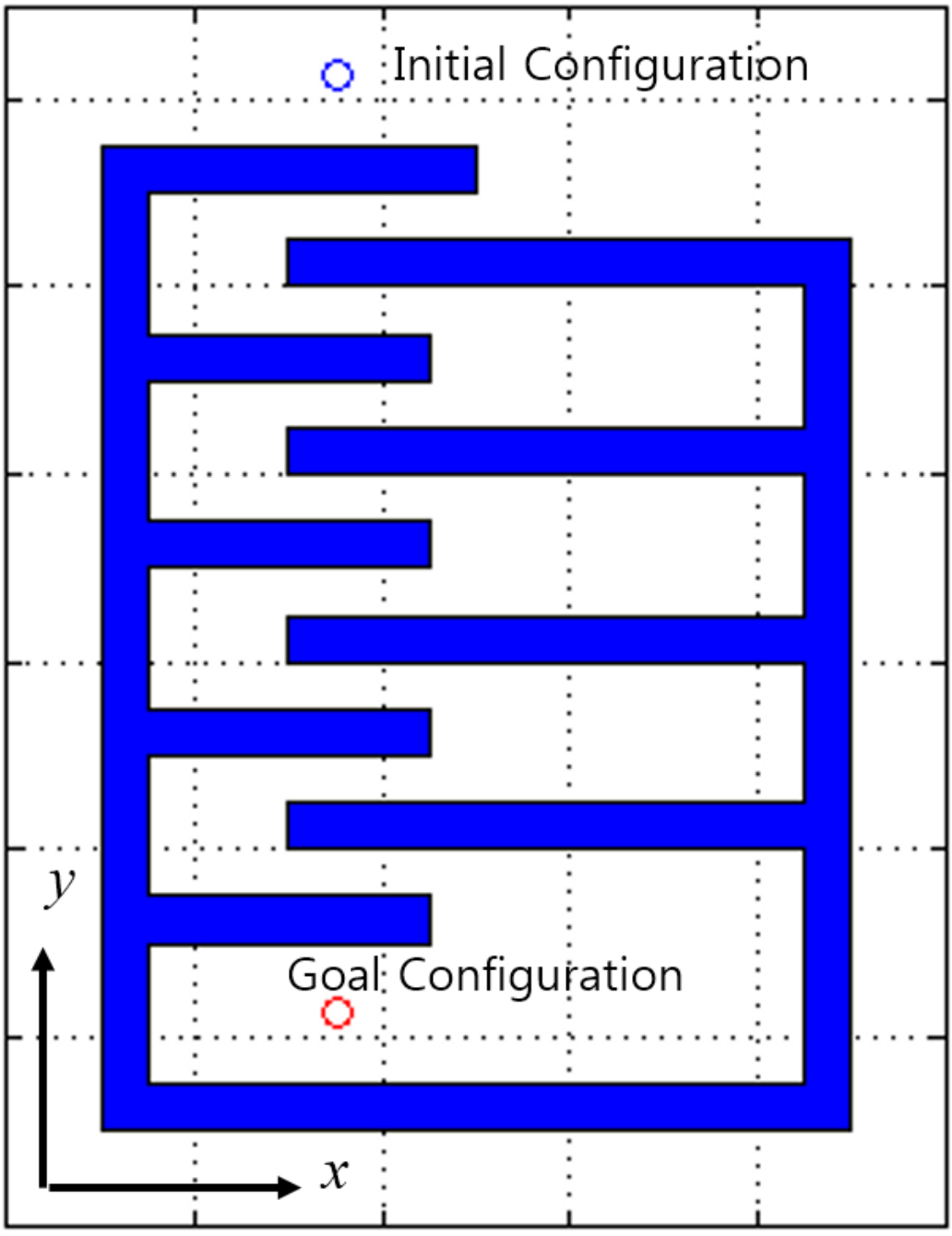
1.1 1) Navigation Planning - Bug Algorithms for Point Robot

1.1.1 a) Bug 1 algorithm:

Implement the Bug 1 algorithm for domain1 and 2 in Figure 1. Provide videos of the simulations(upload the movies to your repository and include the snapshots in your summary report). For each domain, compute the ratio

$$CR_{Bug1} = \frac{\text{actual traveled distance}}{\text{Euclidean distance between init. and goal}}$$

Out[3]:



Bug 1 algorithm is implemented in `Q1/bug1.m`. Domain information is encoded in `domain1.m` and `domain2.m`.

Domain 1:

```
Out[2]: <IPython.core.display.HTML at 0x7f16313b1ed0>
```

$$CR_{Bug1}^{Domain1} = \frac{\text{actual traveled distance}}{\text{Euclidean distance between init. and goal}} = \frac{1128}{401} = 2.812102$$

Domain 2:

```
Out[4]: <IPython.core.display.HTML at 0x7f16313b1290>
```

$$CR_{Bug1}^{Domain2} = \frac{\textbf{actual traveled distance}}{\textbf{Euclidean distance between init. and goal}} = \frac{666}{280} = 2.378571$$

1.1.2 b) Bug 2 algorithm:

Implement the Bug 2 algorithm for domain1 and 2 in Figure 1. Provide videos of the simulations(upload the movies to your repository and include the snapshots in your summary report). For each domain, compute the ratio

$$CR_{Bug1} = \frac{\text{actual traveled distance}}{\text{Euclidean distance between init. and goal}}$$

Bug 2 algorithm is implemented in `Q1/bug2.m`. Domain information is encoded in `domain1.m` and `domain2.m`.

Domain 1:

```
Out[9]: <IPython.core.display.HTML at 0x7f1630b71390>
```

$$CR_{Bug2}^{Domain1} = \frac{\text{actual traveled distance}}{\text{Euclidean distance between init. and goal}} = \frac{506}{401} = 1.261457$$

Domain 2:

```
Out[10]: <IPython.core.display.HTML at 0x7f16313a7150>
```

$$CR_{Bug2}^{Domain2} = \frac{\text{actual traveled distance}}{\text{Euclidean distance between init. and goal}} = \frac{1066}{280} = 3.807143$$

Analysis: As we can see from the competitive ratio (CR) the performance of Bug algorithms depends a lot on the domain used for it. Bug 1 performs similarly for both the domains. Bug 2 performs better than Bug 1 in domain 1 but it is worse in domain 2. This is because Bug 1 is exhaustive and predictable where as Bug 2 is greedy and not so predictable. In many cases Bug 2 out performs Bug 1 because of its greedy nature but in some special domains like domain 2, it can perform very badly.

1.2 2) Navigation Planning - Potential Field Navigation for Point Robot

1.2.1 a) Potential field navigator:

Implement a simple potential field navigator with attractive and repulsive fields for domain1 in Figure 1. Note that unlike the problem 1, your navigator has access to a global map and it will use the map to construct its potential field. Provide a video of the simulation(upload the movie to your repository and include the snapshots in your summary report). Compute the ratio

$$CR_{Bug1} = \frac{\text{actual traveled distance}}{\text{Euclidean distance between init. and goal}}$$

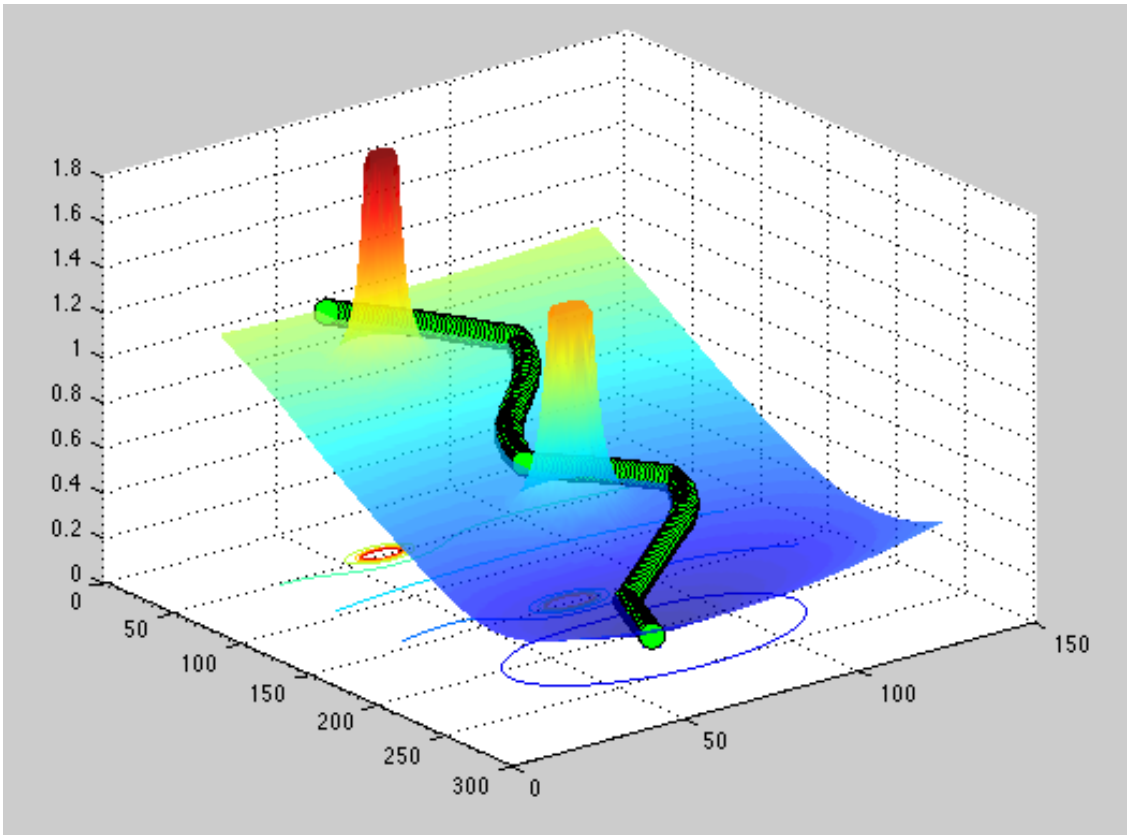
The potential field navigator is implemented by Potential_Nav.m. To run, run each section of the M-file Problem2.m, which calls Potential_Nav for domain1 for part (a), a modified domain1 with two new obstacles for part (b), and a field with 30 random obstacles titled "Extra". The movie for part (1) is saved under the title prob2a (both an .avi and .mov are supplied).

The Potential_Nav algorithm takes the following inputs: initial position (x,y), goal (x,y), obstacles [(x,y,radius)], video recording flag (0/1), and video name (if recording). The obstacle matrix can take any number of obstacles arranged in a nx3 format. Videos are recorded in the .avi format.

The Potential_Nav algorithm first creates a properly-sized field for effecient calculations, creates a potential well of attraction around the goal (U_a), and repulsive potential fields around each article (U_r), and adds them together (U). U_a is the Euclidean distance of each point in the field from the goal and U_r is the obstacle radius/distance^2 of each point in the field from the obstacle. The dynamic solver then checks if the solution is stuck in a local minimum or at goal, if not it checks the potential values around the current solution and finds the direction with least cost ($- \bigtriangledown U$). The solution then moves and the total distance travelled so far is calculated. Finally, if a movie is being recorded, it is exported and the distance ratio $CR_{P.F.}$ is calculated.

In part (a), the $CR_{P.F.} = 1.25$ which means that the solution found was fairly efficient. The solution (represented by the green ball on the potential field and by the blue x on the contour map) skirts to the left side of both obstacles before descending toward the goal, as can be seen in the accompanying figure. This is nearly the same solution as given by Bug 2 on Domain 1, which had a very similar $CR_{P.F.} = 1.26$.

Out[12]:

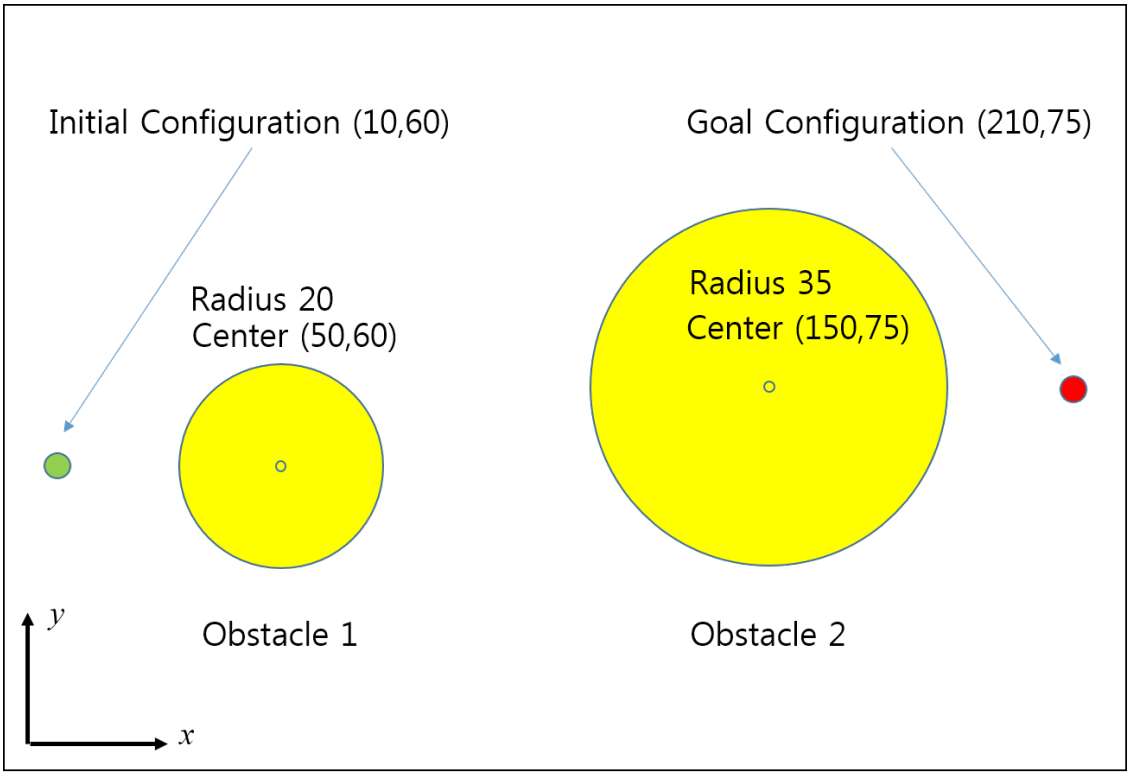


Out[13]: <IPython.core.display.HTML at 0x7f1630b71510>

1.2.2 b) Local minimum:

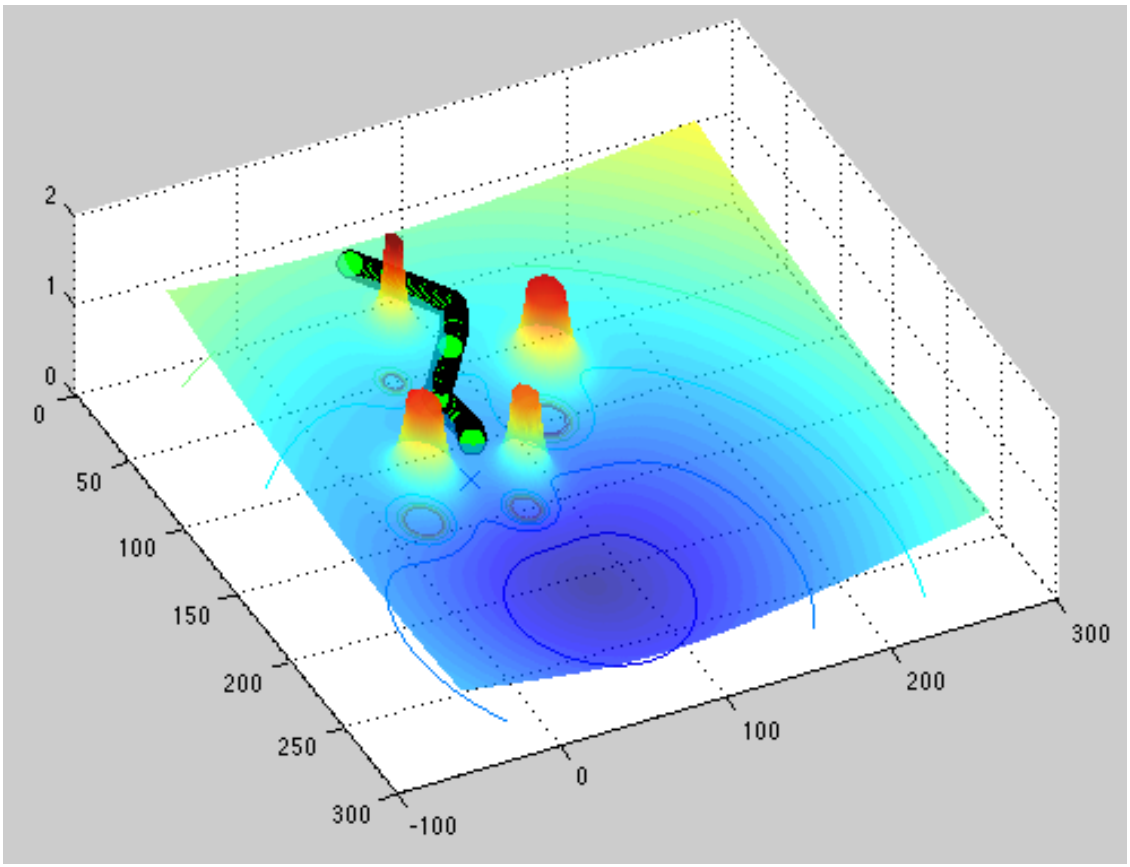
Construct an example domain by either adding obstacles to domain1 or changing the positions of obstacles in domain1 such that there is(are) local minimum(minima) and apply your potential field navigator to the modified domain. Show that your P.F. navigator fail to find a solution even when one exists. Provide a video of the simulation(upload the movie to your repository and include the snapshots in your summary report).

Out[2]:



This problem only differs from part (a) in that a new obstacle with a radius of 100 was added at (100,120), a new obstacle was added with a radius of 80 was added at (140,20), and the radius of the obstacle at (150,75) was increased to 50. As the second figure and the movie (prob2b.avi) show, the solution gets stuck in the local minimum and the appropriate error is thrown. The same global solution exists as in part (a) but the solution cannot reach it.

Out[14]:



Out[15]: <IPython.core.display.HTML at 0x7f1630b71310>

1.3 3) Manipulation Planning - Differential Kinematics

Let $x = f(q)$ be the forward-kinematics function that takes the joint angles q and outputs the pose of the end-effector in the world frame. We know that the end-effector velocity can be written in terms of joint velocities as follows:

$$\dot{x} = \dot{J}q(1)$$

where J is a Jacobian which has the partial derivatives of $f(q)$ with respect to q. For a 3 degrees of freedom, planar robot arm, the pose of the robot x would denote the (x, y) location of the end-effector and q, the orientation (in radians) of the end-effector with respect to the $+x$ axis. In this case, the Jacobian would be a 3×3 matrix with 3 columns for each of the 3 joints. For problem 3 - problem 5, we will be working with a 3 DoF planar robot arm with link lengths $l_1 = 2$, $l_2 = 2$ and $l_3 = 1$ (Figure 2).

1.3.1 a)

Move the end-effector from the initial pose of $p_i = (2.6, 1.3, 1.0)$ to the goal pose $p_g = (-1.4, 1.6, -2.0)$ using the inverse Jacobian. The idea is that the end-effector takes small steps towards the goal with a small target velocity $\dot{x_j}$ at each iteration j. Provide the video of the arm as it moves from the initial to goal pose (upload the movie to your repository and include the snapshots in your summary report).

The inverse jacobian is computed and a potential field is used to minimize the norm of the end effector's start pose and the end pose. The directional derivative for the cost function is computed and then multiplied by the jacobian inverse to give the change in the joint angles.

To run the code set question_no to 1 (line 12) in Code/Q3/rip_hw2.m

See Code/Q3/videos/hw2q3a.mp4,

1.3.2 b)

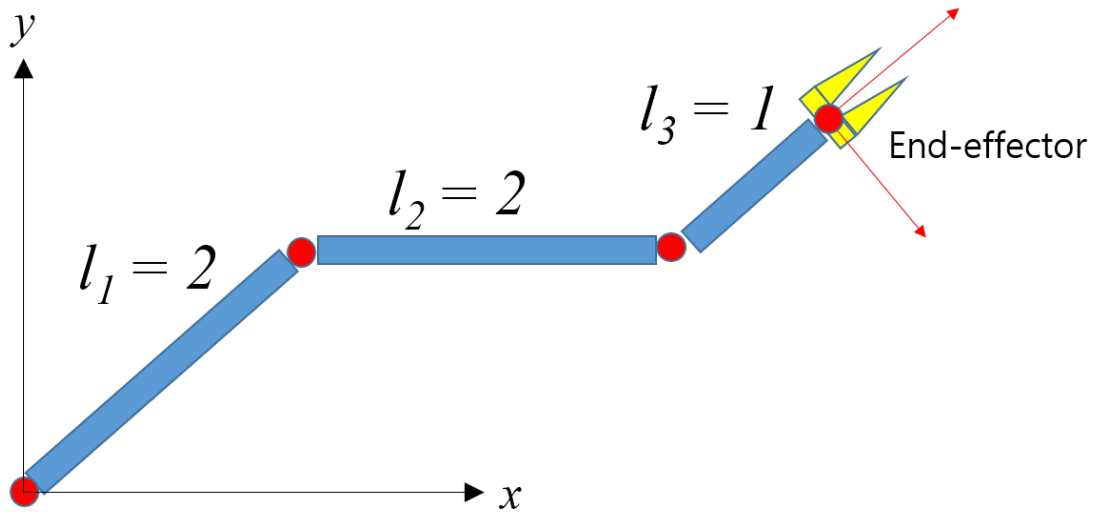
> You have implemented the Jacobian control approach and probably saw that joint limits and collisions are not taken into consideration. How would you incorporate these two important aspects of planning into this approach? To verify your idea, construct an example with a circular obstacle (radius 1), and implement your idea for collision avoidance(assume there are no joint limits.) Provide a video of the simulation(upload the movie to your repository and include the snapshots in your summary report.)

The same potential field is augmented with the repulsive force in the cost function. The directional derivative is computed and fed through the inverse jacobian to get the joint angles. The code can be run with/ without collision avoidance to see the difference.

To run the code set question_no to 2 (line 12) and avoid_obstacle (line 13) to 0(to not avoid)/1(to avoid) in Code/Q3/rip_hw2.m

See video Code/Q3/videos/hw2q3b.mp4

Out[4]:



1.4 4) Manipulation Planning - RRTs

> In this problem, you will implement 4 types of RRTs - (a) baseline, (b) goal-directed, (c) connect, (d) bidirectional and (e)analyse the result. For each implementation, provide (i) a video of the tree as it grows until it finishes ex- panding and (ii) a video of the arm as it moves from start to goal configuration(upload movies to your repository and include the snapshots in your summary report). Use the same arm as in problem 3.

1.4.1 a) Baseline:

> Create a tree of 300 nodes for both without and with obstacle cases (see scene1.txt for obstacle info). In this case, no video for arm motion is needed.

We started by importing some of our code used to define our 3 DOF robot arm, the scene geometry, our RRT algorithm, as well as plotting tools for rendering resulting tree structures and trajectories. Note that for Scene 1 is corrected to be similar to Scene 2, otherwise the start joint state specified, $q_s = (0.14, 1.45, 1.0)$, would place a portion of the arm inside the obstacle.

For our first RRT algorithem, we'll begin with a basic baseline approch, i.e. without any goal bias, connect methods or multiple exploring trees. The RRT is first seeded at the starting configuration, q_{start}, in joint space and expands toward a random sample, q_{rand}, drawn uniformly from the joint space, and extended from the nearest node on the tree, q_{near} to valid q_{new} by a distance of a given δ in the direction of q_{rand}.

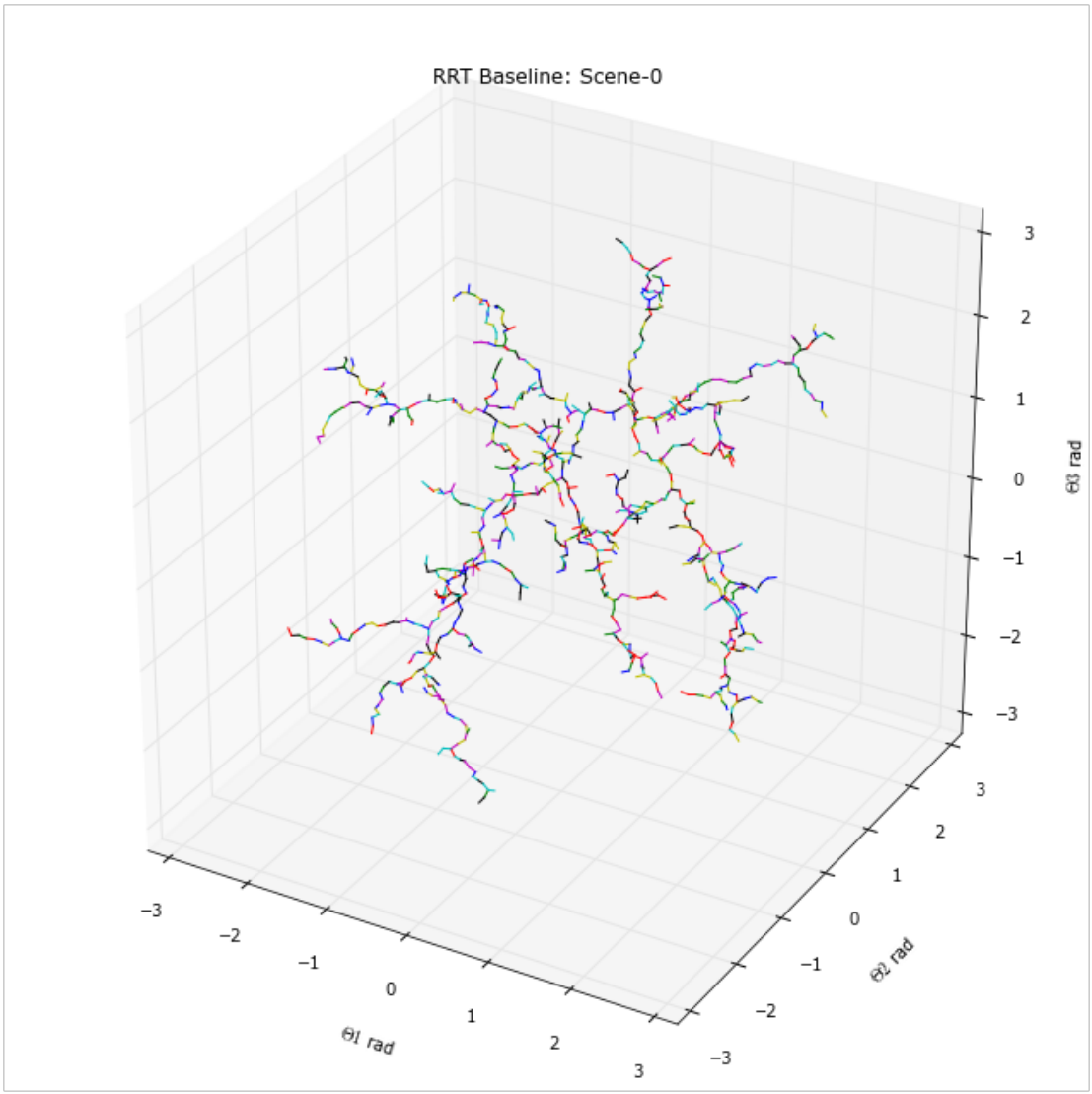
For our purposes here, we'll define our q_{new} points to be valid using a simple collision detection method. This is done by subsampling points along the joints and lengths of the arm, 5 points per length in this case, and looping through each subsample with each obstacle in the workspace. This may result in small overlaps in the arm model with corners of obstacles. This could be improved by using line and polygon intersection formulas or KD trees defining collision boundries, but in the intrest of simplicty and speed, as python is a scripting language, we'll opt for a basic subsampling collision check.

Also, as it is highly improbable that we happen to chose a q_{new} that lands exactly on our goal state, we'll broaden our goal to a region within a given tolerance in our jointspace, for here we'll be using $\pm 10^\circ$ in our sum joint configuation with the goal state to be our goal region. We will remove this relaxation when we gain a goal bias behavior.

```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('tolerance: ', 0.17453292519943295)
('delta: ', 0.08726646259971647)
('iterations: ', 1000)
('success: ', False)
```

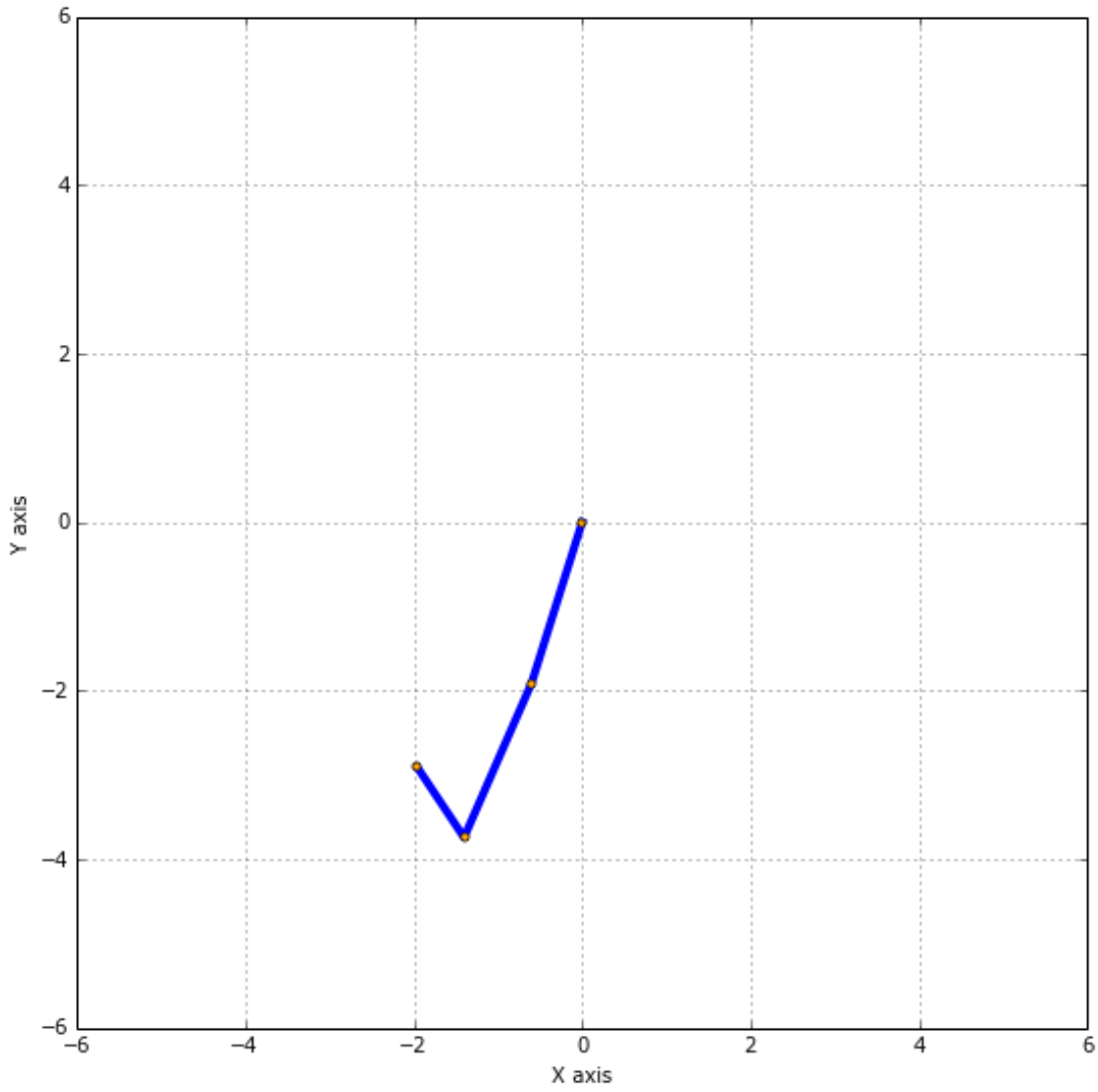
So we see that after 1000 iterations we didn't get close enough to the goal region. However, let's take a look at the resulting RRT. In the figures below we can see the joint space for all three DOF in the 3D axes plot, with the the range of each axes spanning over the region $\{-\pi, \pi\}$. We can see here that scene devoid of obstacles, our RRT grows uniformaly into the void of the joint space.

```
Out[8]: <matplotlib.animation.FuncAnimation at 0x7f09c3aa1b10>
```



Now lets take a look at the resulting trajectory. Our RRT did not have a suffecient number of allowable iterations to reach our desired goal region, but let's check how close it got. We can do this by using the path to the node on the current tree that grew closest to the goal instead. You may also have noticed that the arm attempts the long way arround to the goal state, i.e. in a clockways manner. This is due to the absence of wrapping in our joint space, this lack of wrapping joint angels however more closly aligns with the physical limatations to which most robot arms are constrianed.

```
Out[9]: <matplotlib.animation.FuncAnimation at 0x7f09c241e5d0>
```



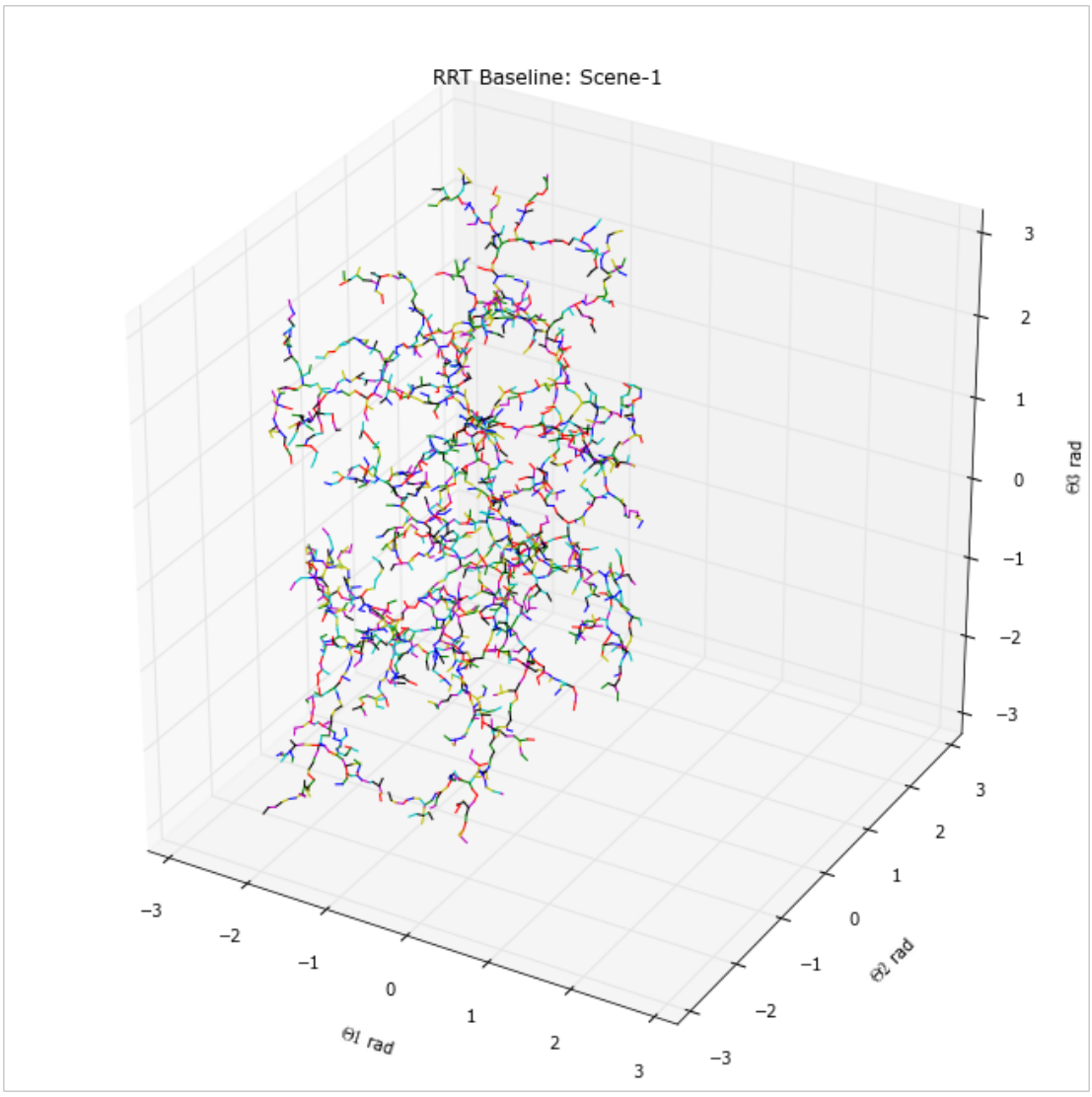
It looks like after 1000 iterations, the basline did get us a close. We can see the jerk in the arms movements resulting from the discrete deltas we grew our tree with. We could smooth our motions by utilizing a smaller growth delta, or using a spline curve. But this will add to the search time of our RRT. And with a just a simple basline method, this naive way of sampling is a lengthy procedure already. Let's try our basline with some other scene types, but with $\times 5$ as many allowed iterations as before.

```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('tolerance: ', 0.17453292519943295)
('delta: ', 0.08726646259971647)
('iterations: ', 5000)
('success: ', False)
```

Even with 5000 iterations we were not able to reach our goal. In the figures below we'll see how the gowth of the tree has changed. Ideally the tree grows larger, as the number of iteration was incresed, but there are now regions within the joint space that are no longer explored, such as the $\theta 0 > 0 rad$. This is due

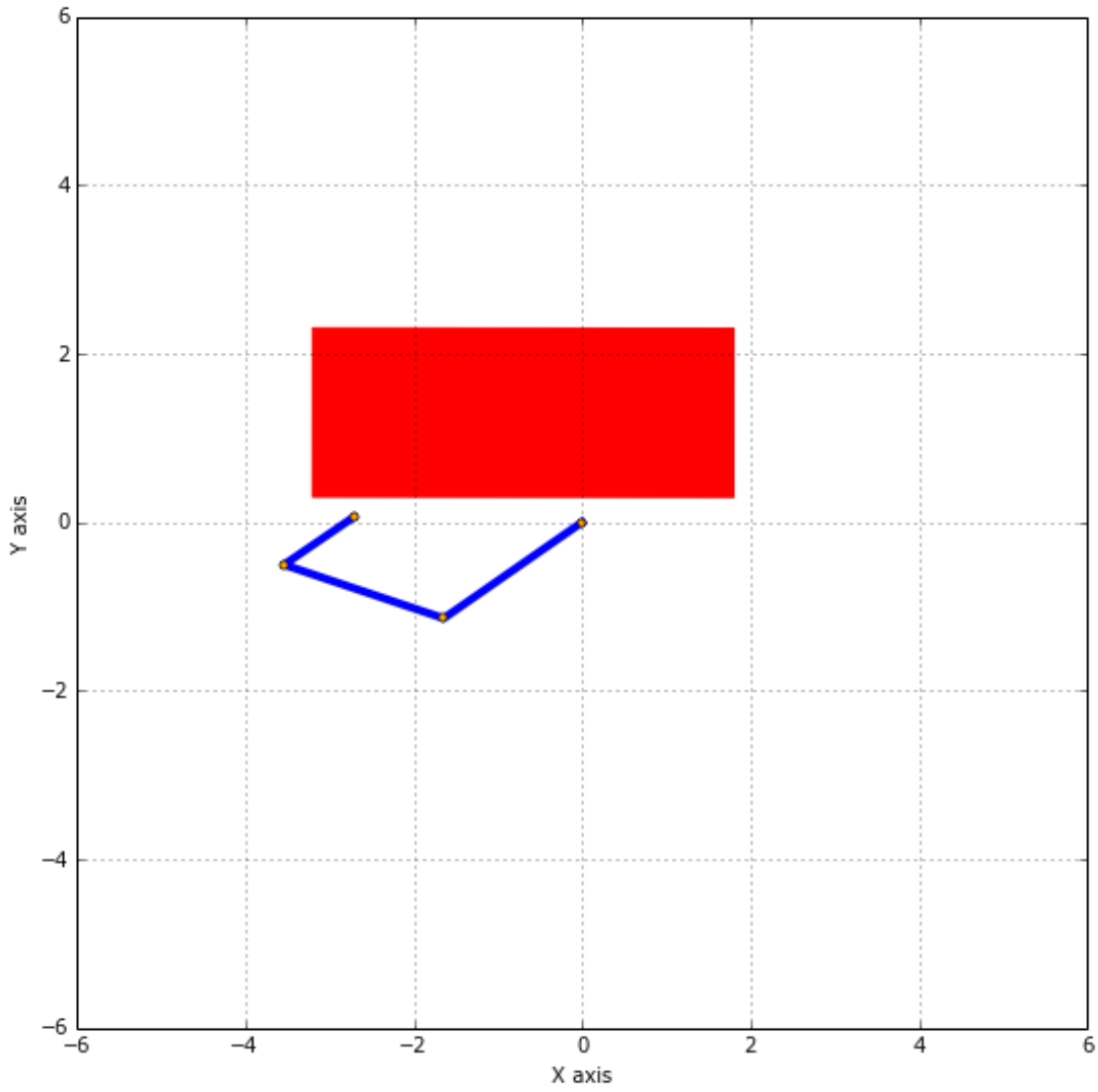
to how the obstacle in scene 1 now projects itself into the joint space, as the first joint, or the sholder of the arm can no longer move into the upper two quadrents of the workspace due to collisions with the rectangle.

Out[12]: <matplotlib.animation.FuncAnimation at 0x7f09c222d710>



Let's take a look at the resulting trajectory. We see there is a bit of an oscillation as we narrow down the ends of tree to the goal region. This is because by the time the tree encountered the goal region, it had already gained a dense number of branches around the same voronoi cell.

Out[13]: <matplotlib.animation.FuncAnimation at 0x7f09bedaee50>

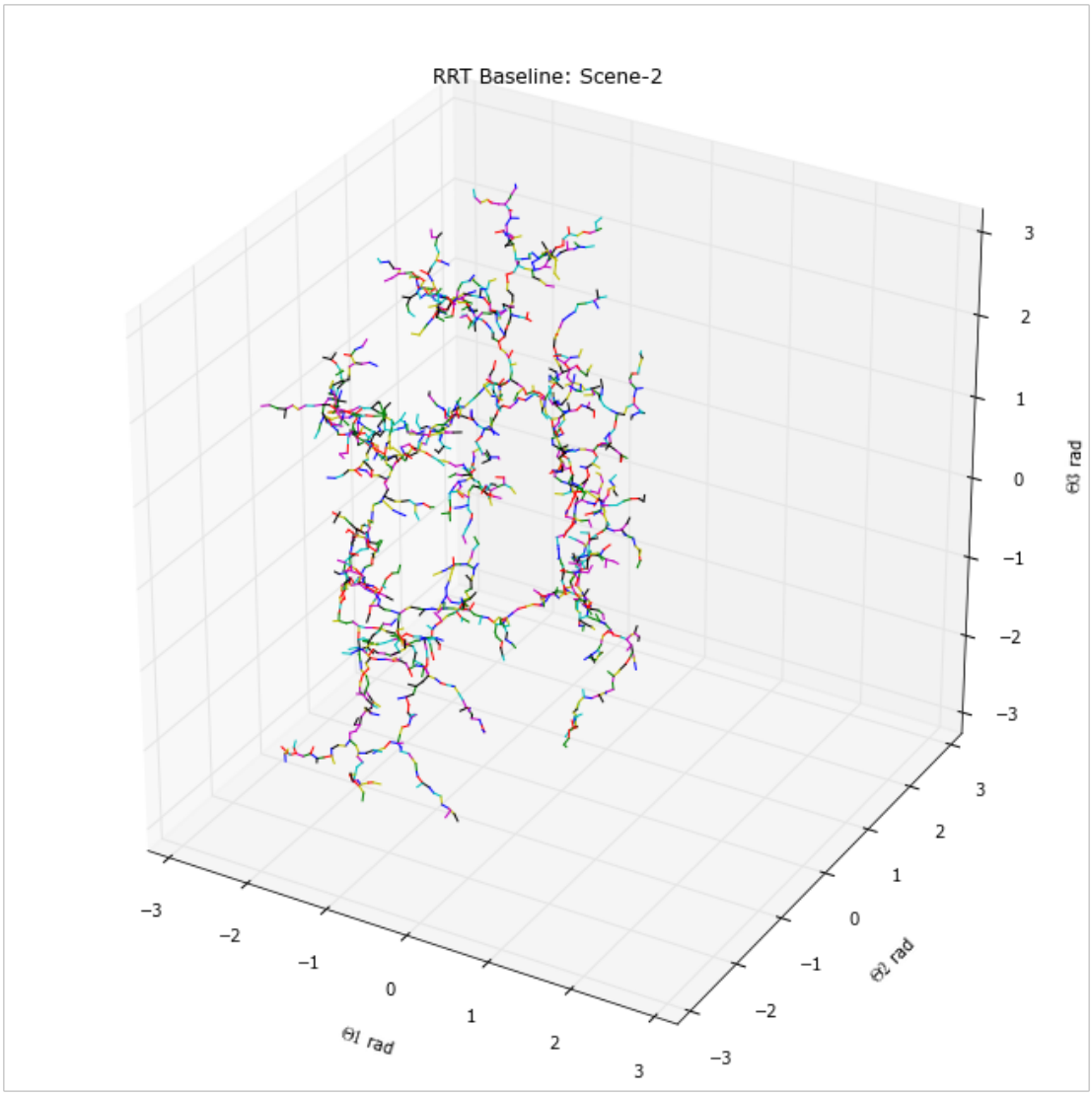


Now let's take a look at scene 2 using the baseline again.

```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('tolerance: ', 0.17453292519943295)
('delta: ', 0.08726646259971647)
('iterations: ', 5000)
('success: ', False)
```

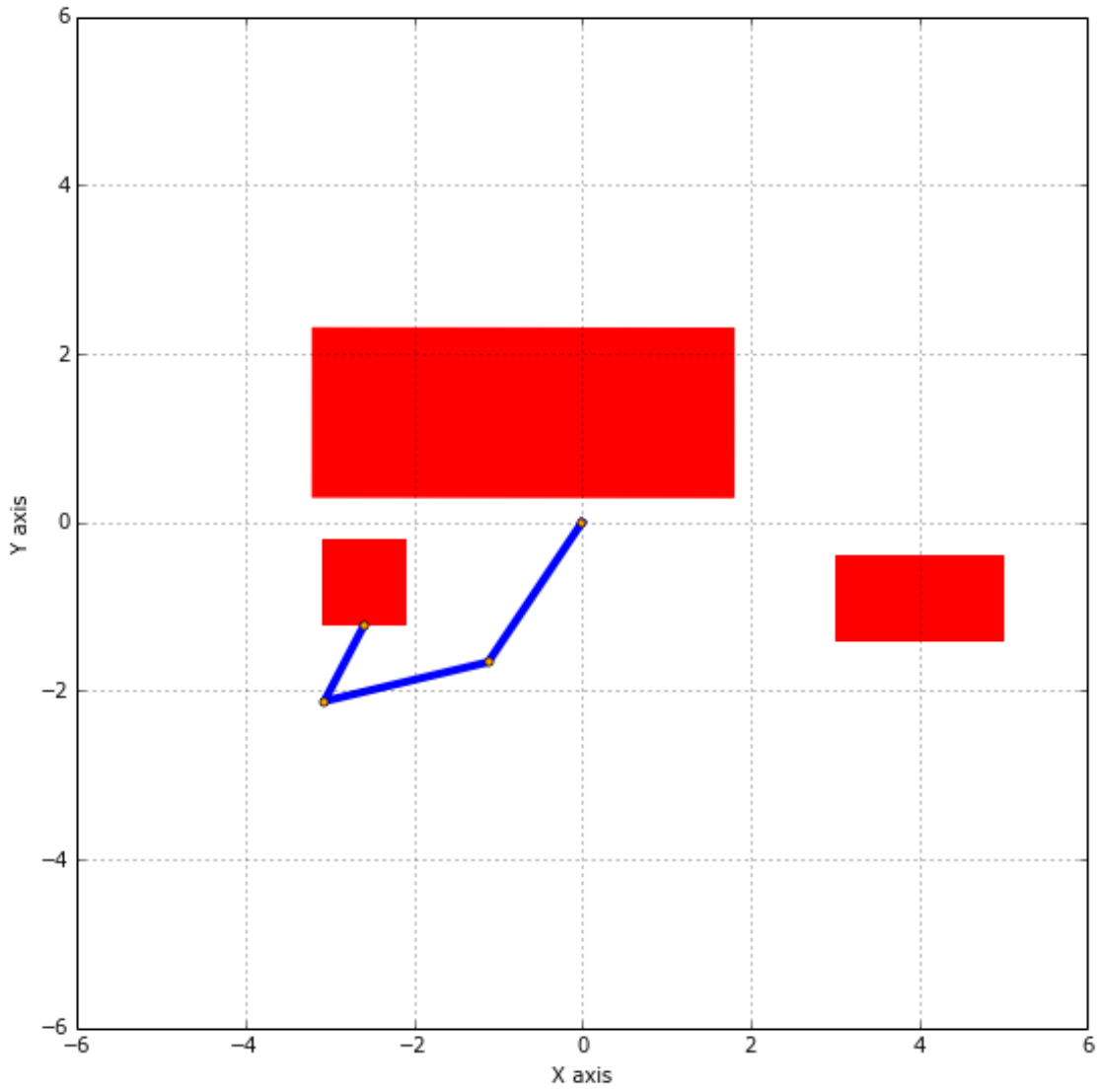
We see that due to the additional obstcales, our tree has been hollowed out some more and has been confined to a tighter volume in the joint space.

```
Out[16]: <matplotlib.animation.FuncAnimation at 0x7f09bed54710>
```



In the resulting trajectory we again see the violent oscillations as our naive sampling jumps around our goal sporadically, and in fact it never converges with the limited number of iterations.

Out[17]: <matplotlib.animation.FuncAnimation at 0x7f09bcb28b50>



1.4.2 b) Goal-directed:

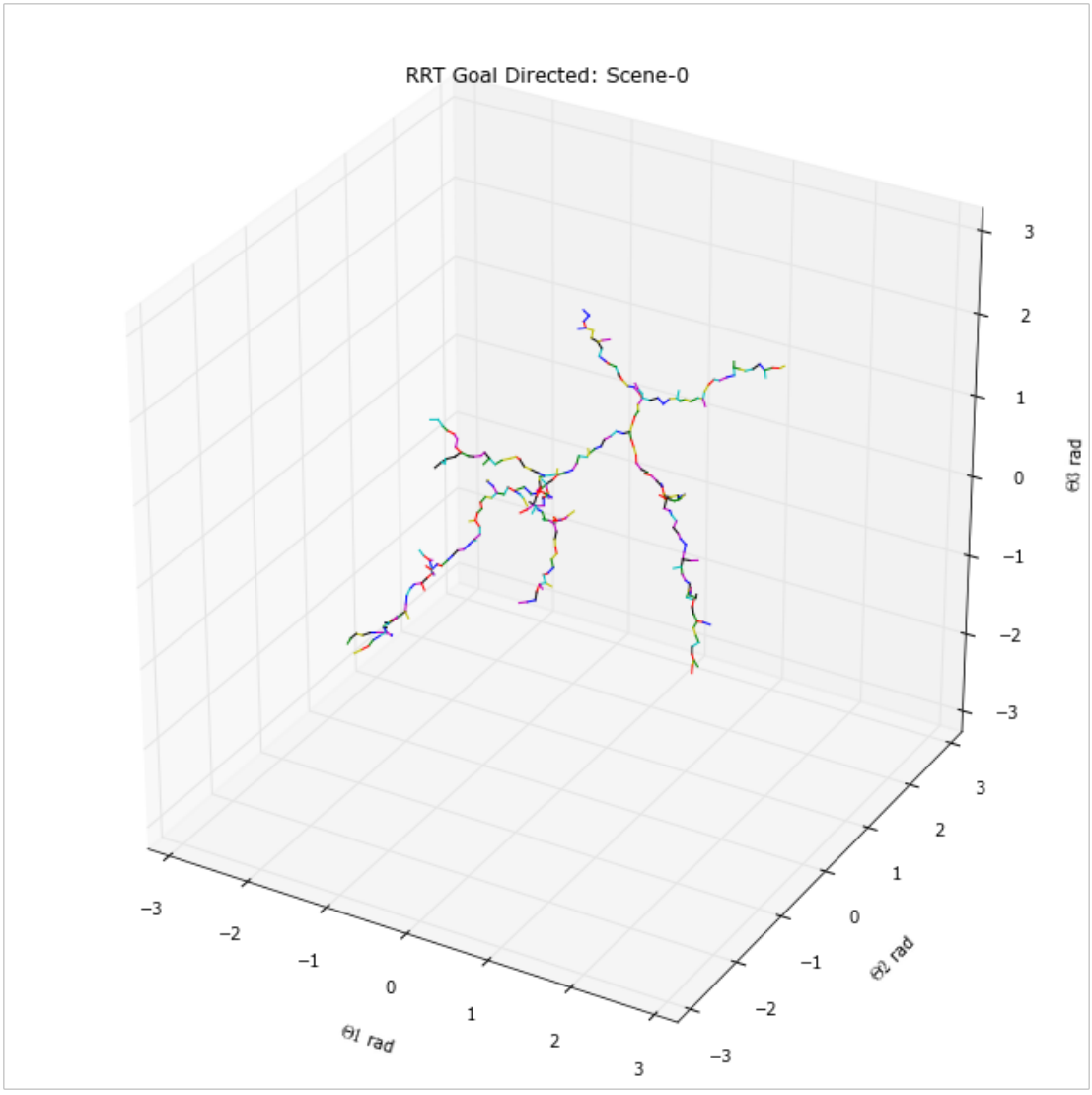
> The idea is to move towards the goal configuration every now and then, and keep expanding randomly other times. The start configuration is $q_s = (0.14, 1.45, 1.0)$ and goal is $q_g = (-2.5, -0.5, -2.1)$. Try for both without and with obstacle cases (see scene2.txt for obstacle info.)

Now let's introduce our goal-directed approach. For this we use nearly the same baseline approach as before, but we'll adjust things in two ways. First we'll add in a 5% chance that a generated target sample is in fact the goal state. With the previous change now implemented, we can remove the need for a tolerance, and can change our goal state to be instaed when one of our extended nodes does reach the exact goal state.

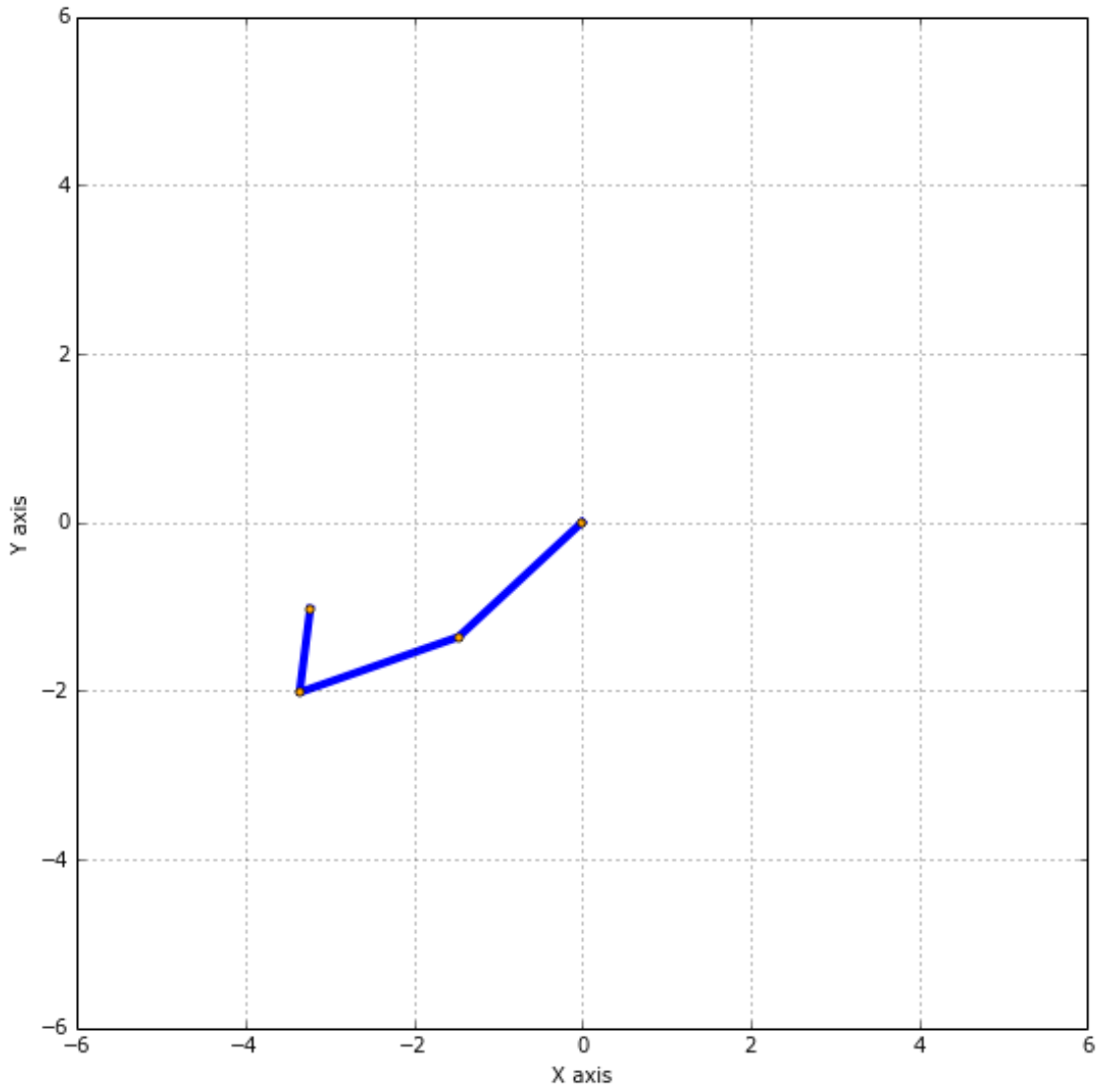
```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('delta: ', 0.08726646259971647)
```

```
('iterations: ', 5000)
('success: ', True)
```

Out[21]: <matplotlib.animation.FuncAnimation at 0x7f09bc507c50>



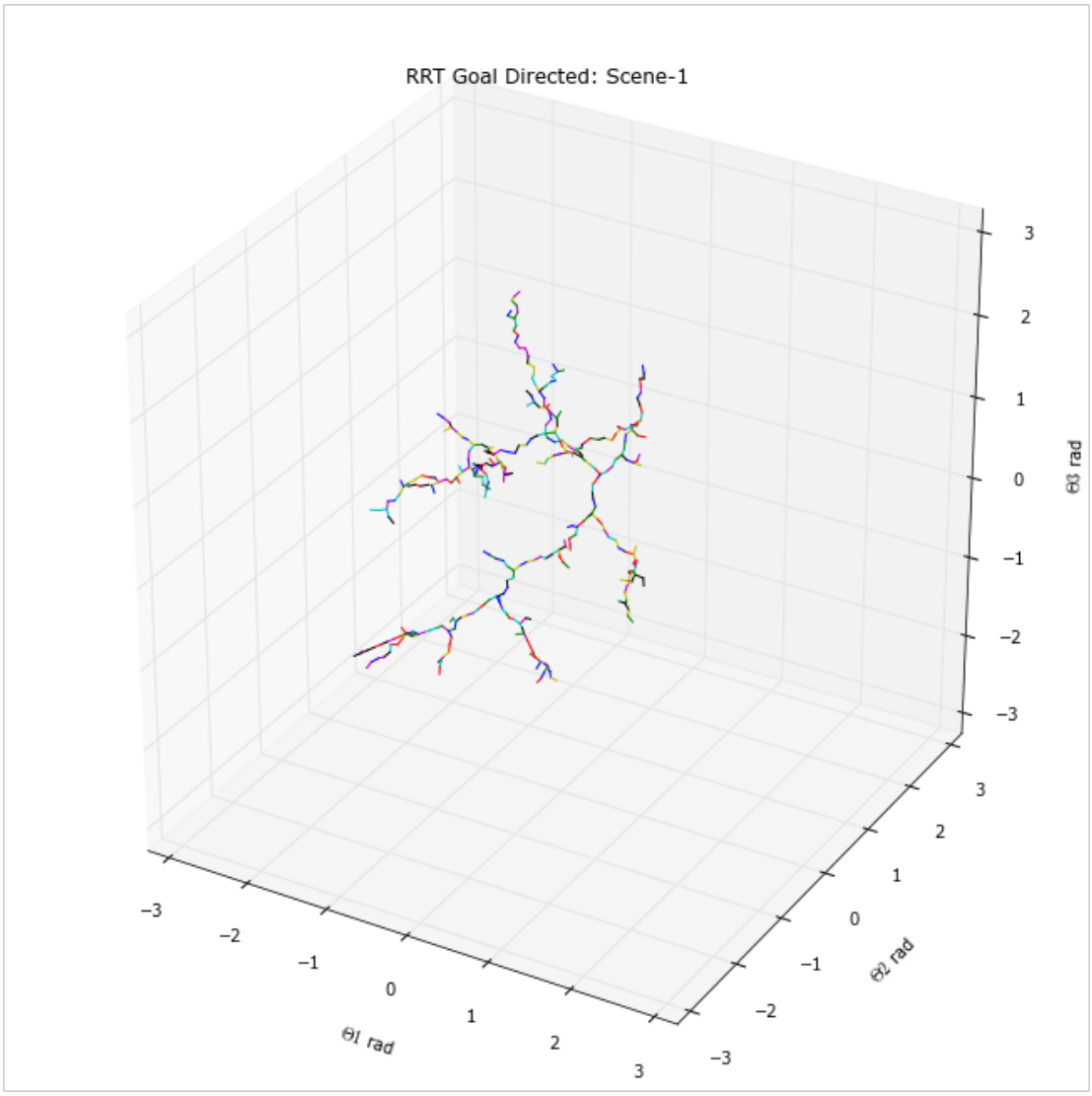
Out[22]: <matplotlib.animation.FuncAnimation at 0x7f09bc33a050>




```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('delta: ', 0.08726646259971647)
('iterations: ', 5000)
('success: ', True)
```

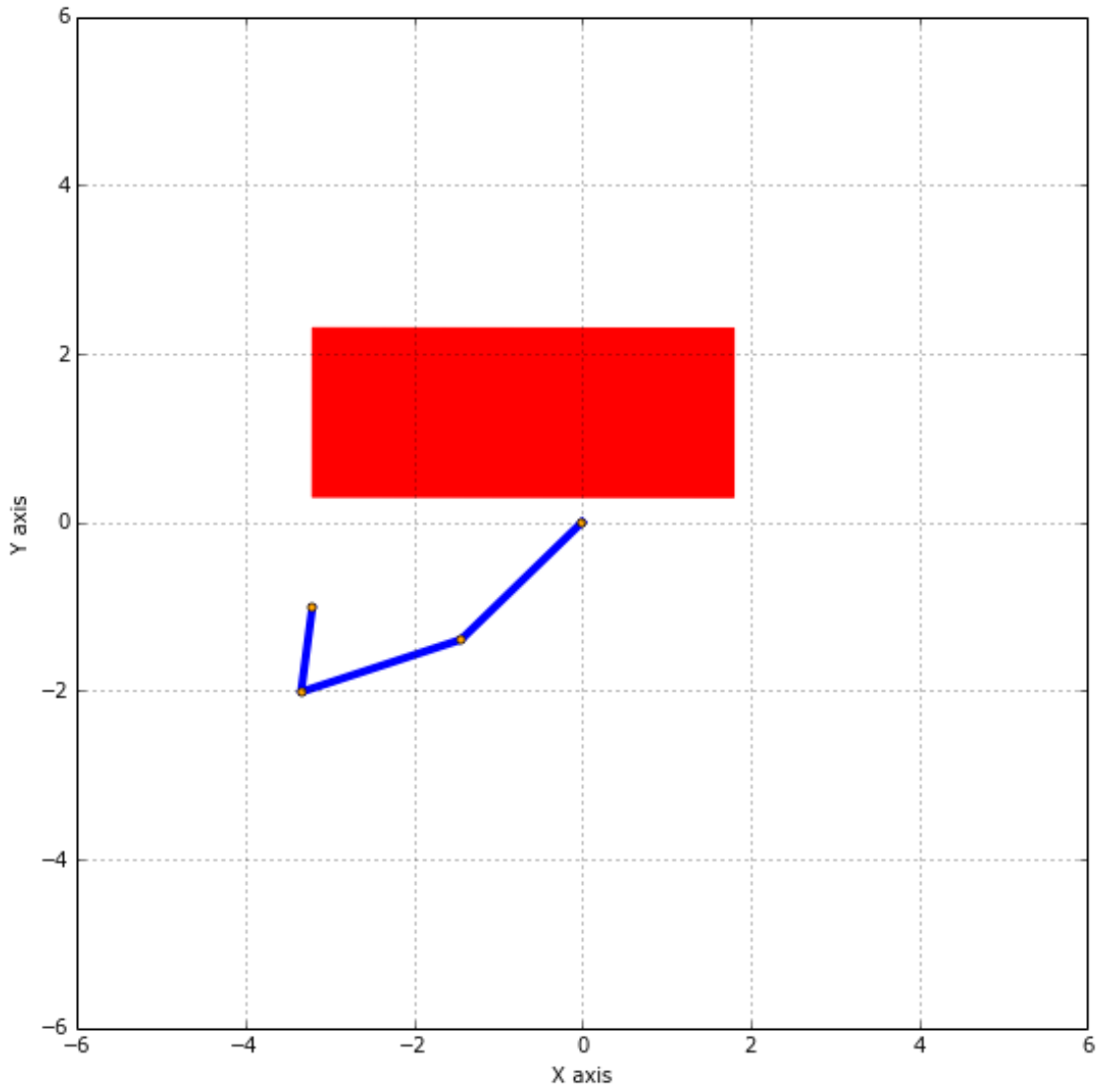
For scene 1, we can see that this has greatly reduced the number of nodes needed to expand, which is nice from a performance perspective for scenes that are not entirely detremental for greedy approaches.

```
Out[25]: <matplotlib.animation.FuncAnimation at 0x7f09bbfd0490>
```



We can see now that the trajectory is now much shorter and rather reasonable in behavor while exactly reaching our target goal state.

Out[26]: <matplotlib.animation.FuncAnimation at 0x7f09bbadf7d0>

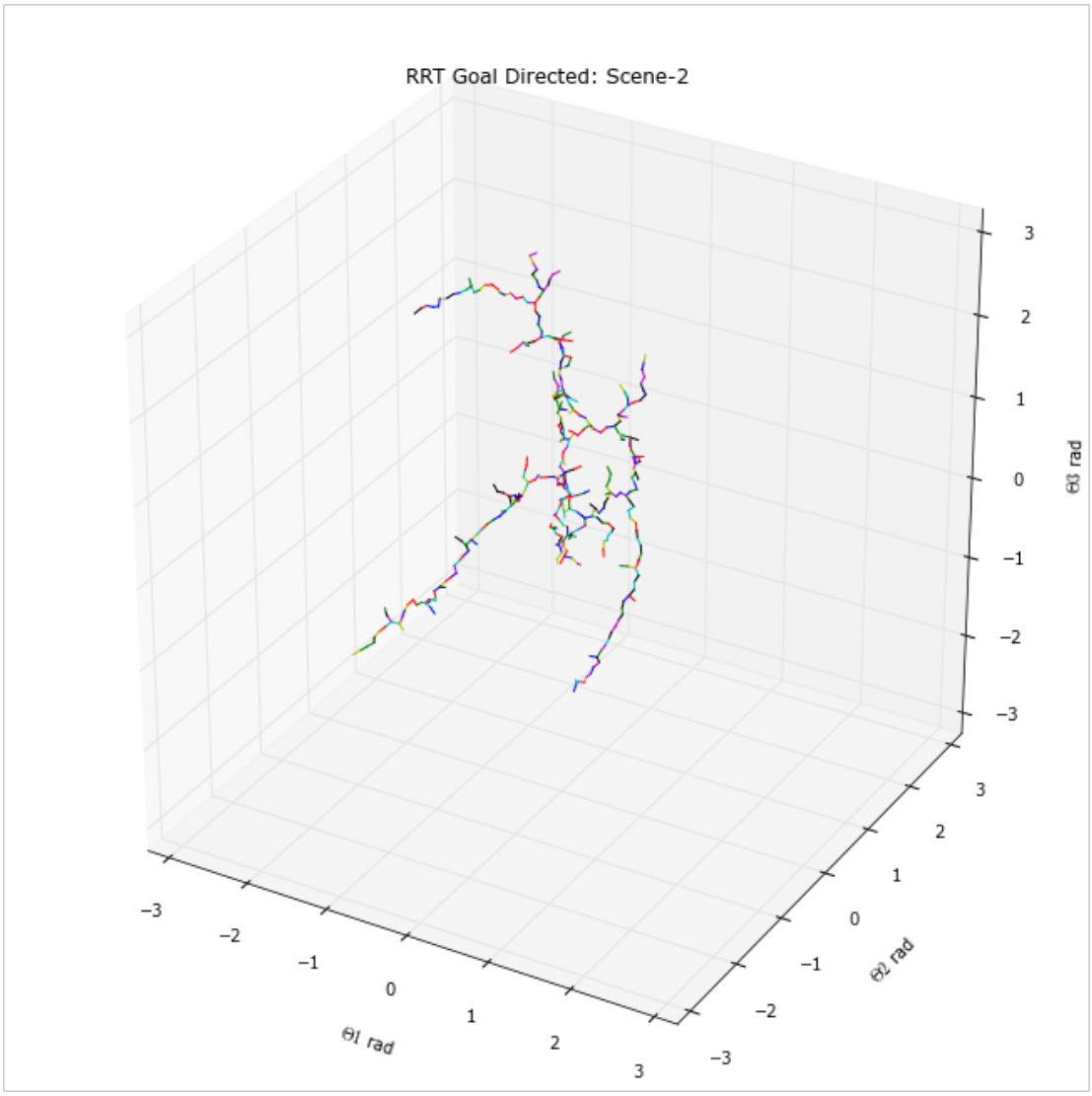


Let's move on the the more difficult challenge of scene 2 using this goal-directed bias.

```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('delta: ', 0.08726646259971647)
('iterations: ', 5000)
('success: ', True)
```

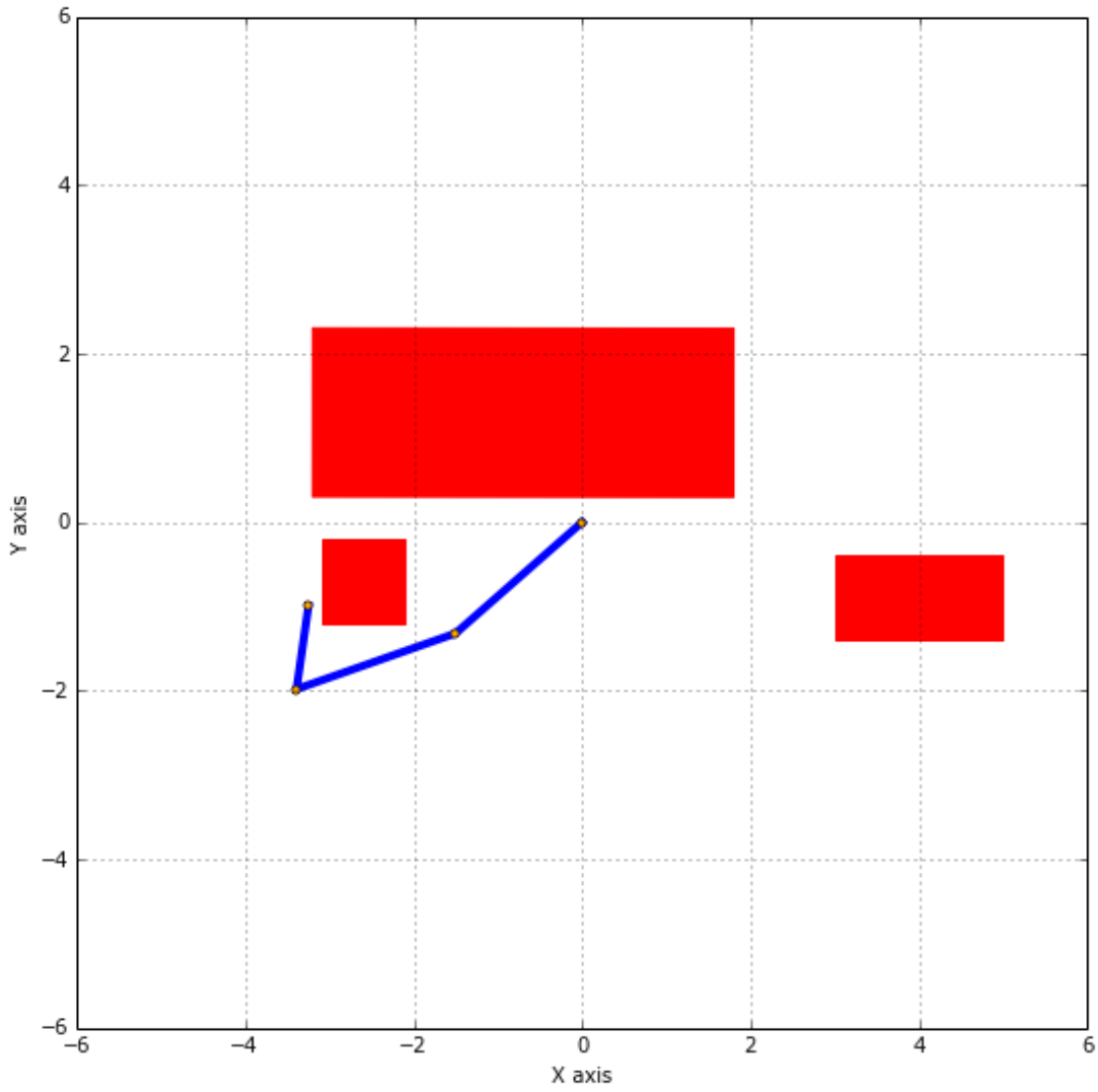
The density of the tree has now narrowed to such an extent that we can now clearly watch the eventual path steadily grow towards the goal in an arching manner around the surrounding collision volumes in the joint space.

```
Out[29]: <matplotlib.animation.FuncAnimation at 0x7f09bb74b890>
```



In the final trajectory we can see how the arm make steady progress to the goal with minimal backtracking thanks to the small amount of bias towards the goal.

Out[30]: <matplotlib.animation.FuncAnimation at 0x7f09bb0ef4d0>



1.4.3 c) Connect:

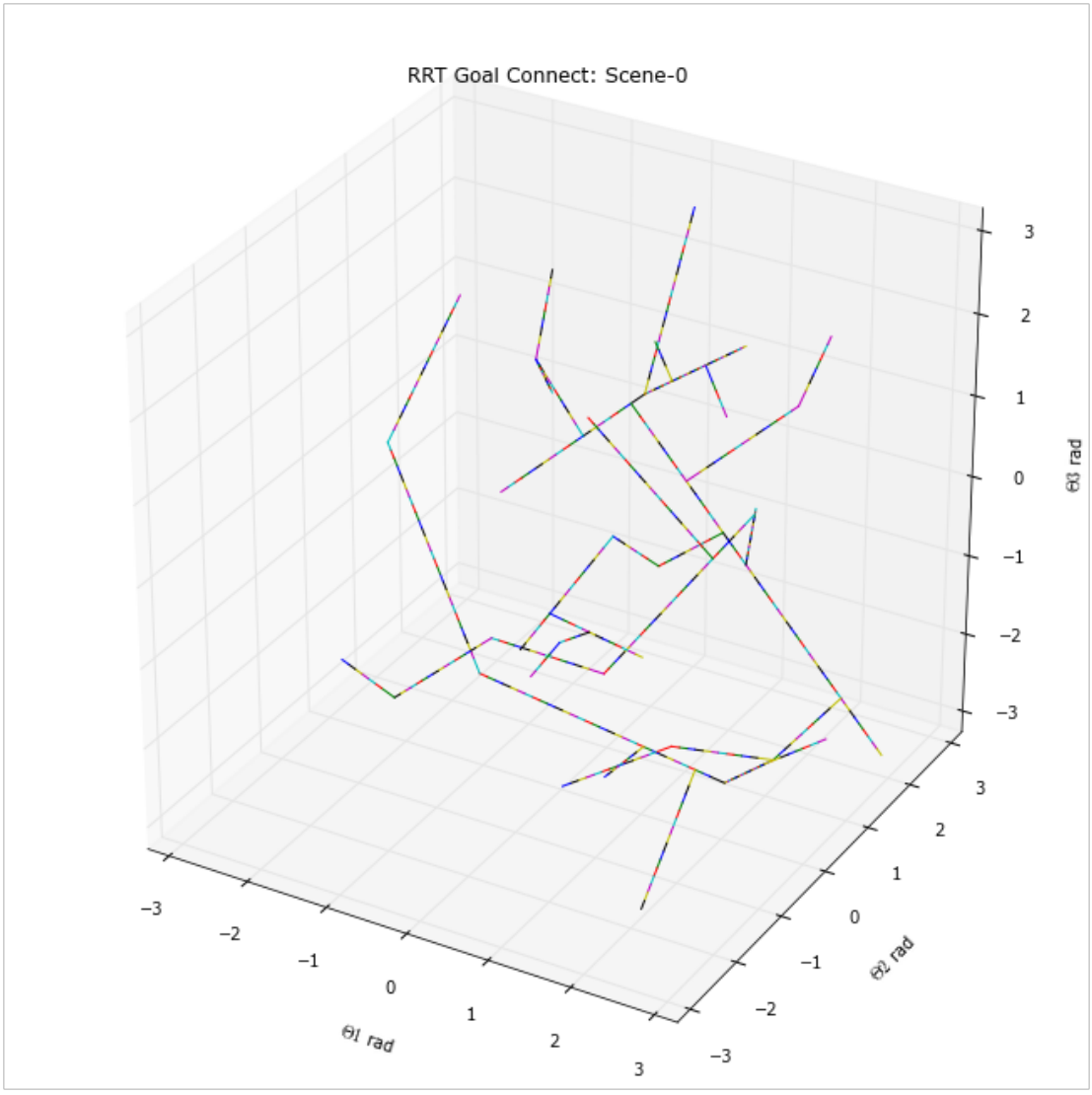
> The idea is to move towards the goal and take as many steps as possible. Use the same start/goal configurations(as in (b)) and try for both without and with obstacle cases(see scene2.txt for obstacle info.)

Now let's add a new capability for our evolving RRT, that is the connecting. This approach will build on our progress with goal-biased sampling, but will add a connecting method for even more rapid expansion. This is done here by sampling a $/q_{rand}$ and finding a q_{near} just like before, only now extending towards $/q_{rand}$ as many times as we can before colliding with an obstacle. In practice, we can just devide the distance to q_{rand} from q_{near} and looping over the delta points in-between untill we collide or reach the sample.

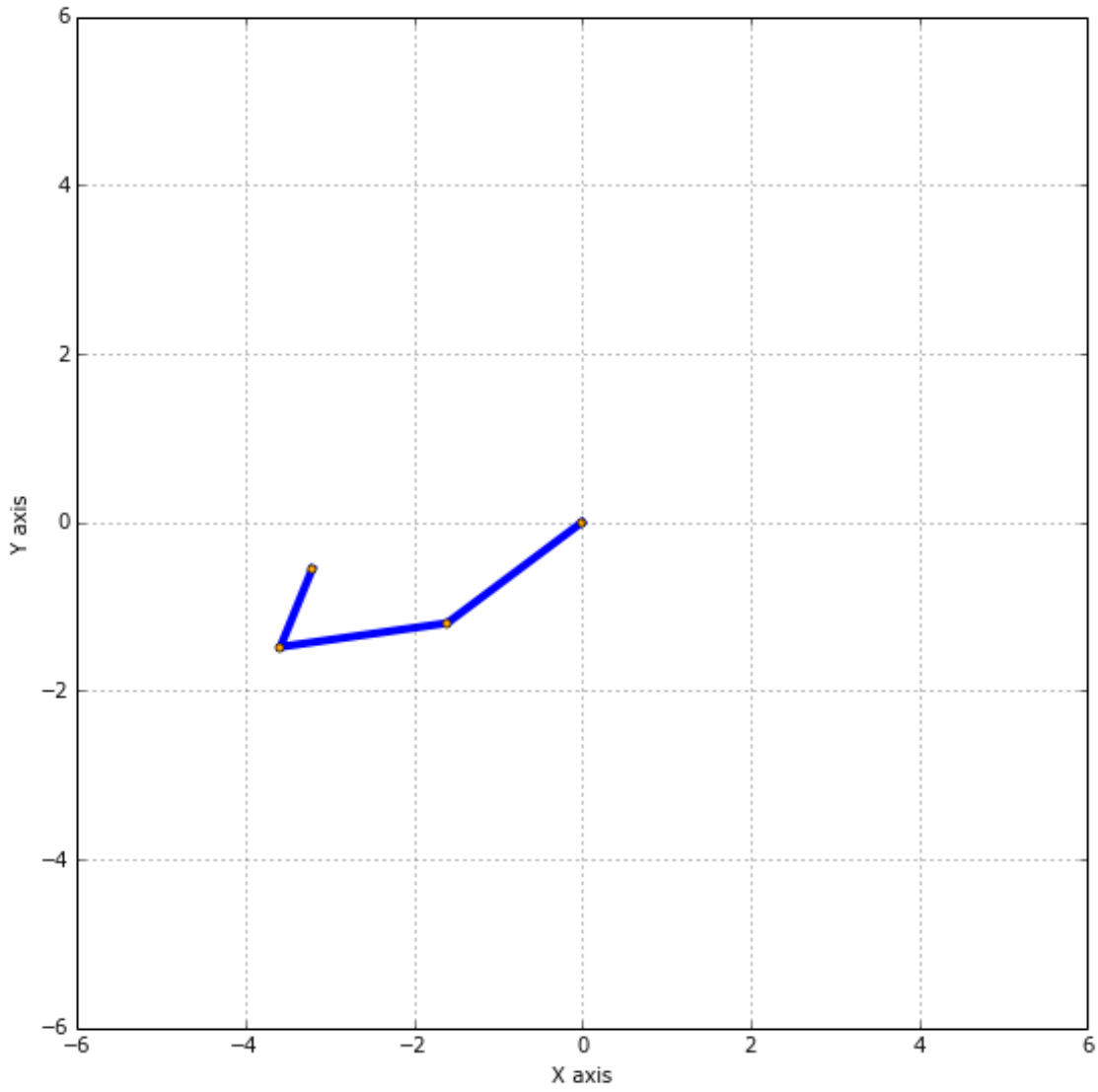
```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('delta: ', 0.08726646259971647)
```

```
('iterations: ', 5000)
('success: ', True)
```

Out[34]: <matplotlib.animation.FuncAnimation at 0x7f09bad62d90>



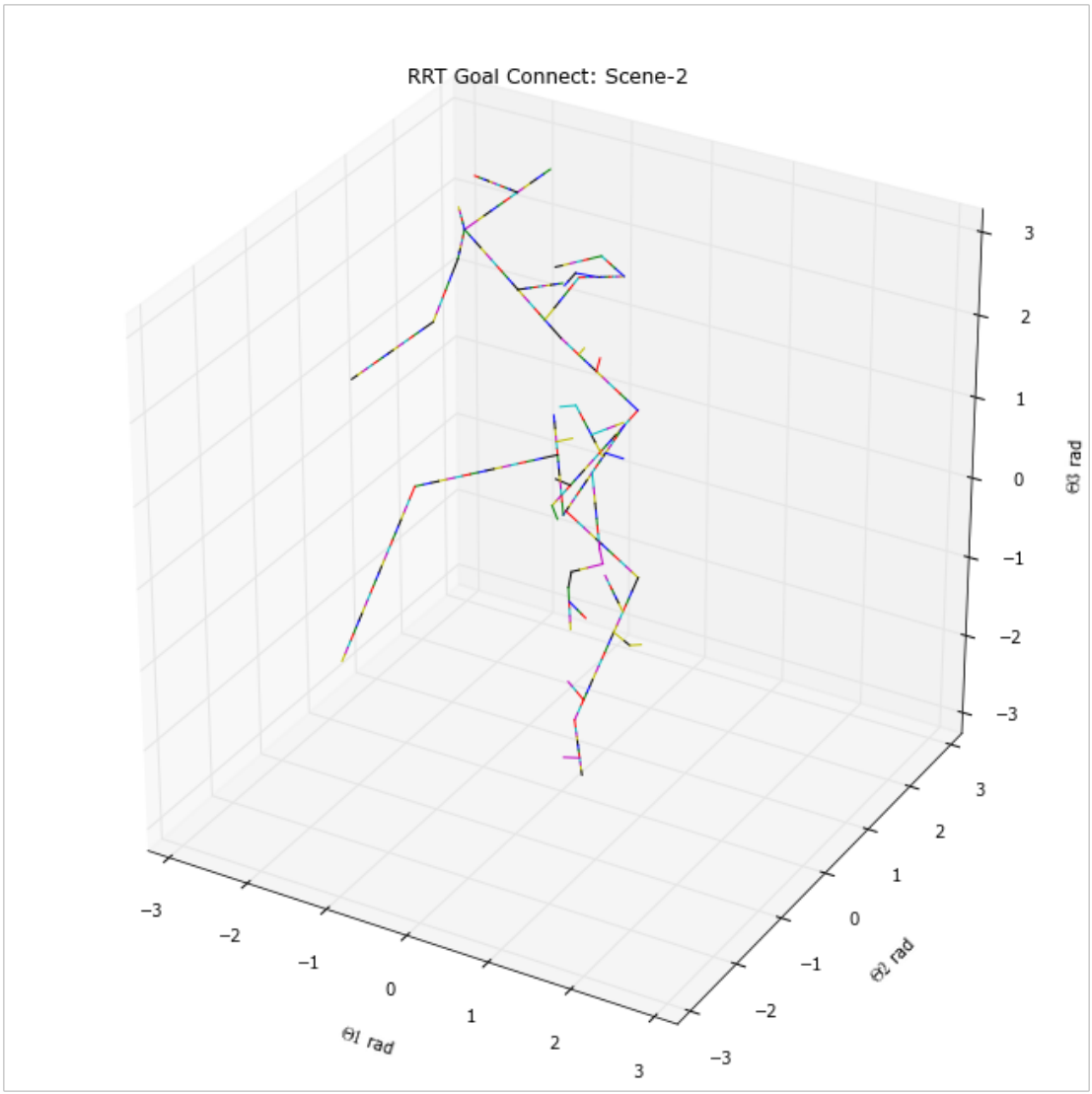
Out[35]: <matplotlib.animation.FuncAnimation at 0x7f09ba539590>




```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('delta: ', 0.08726646259971647)
('iterations: ', 5000)
('success: ', True)
```

That was quick! Let's take a look at the resulting tree stucture. We can see the straght line in the state space resulting from the new connecting behavior.

```
Out[38]: <matplotlib.animation.FuncAnimation at 0x7f09b9fa6590>
```

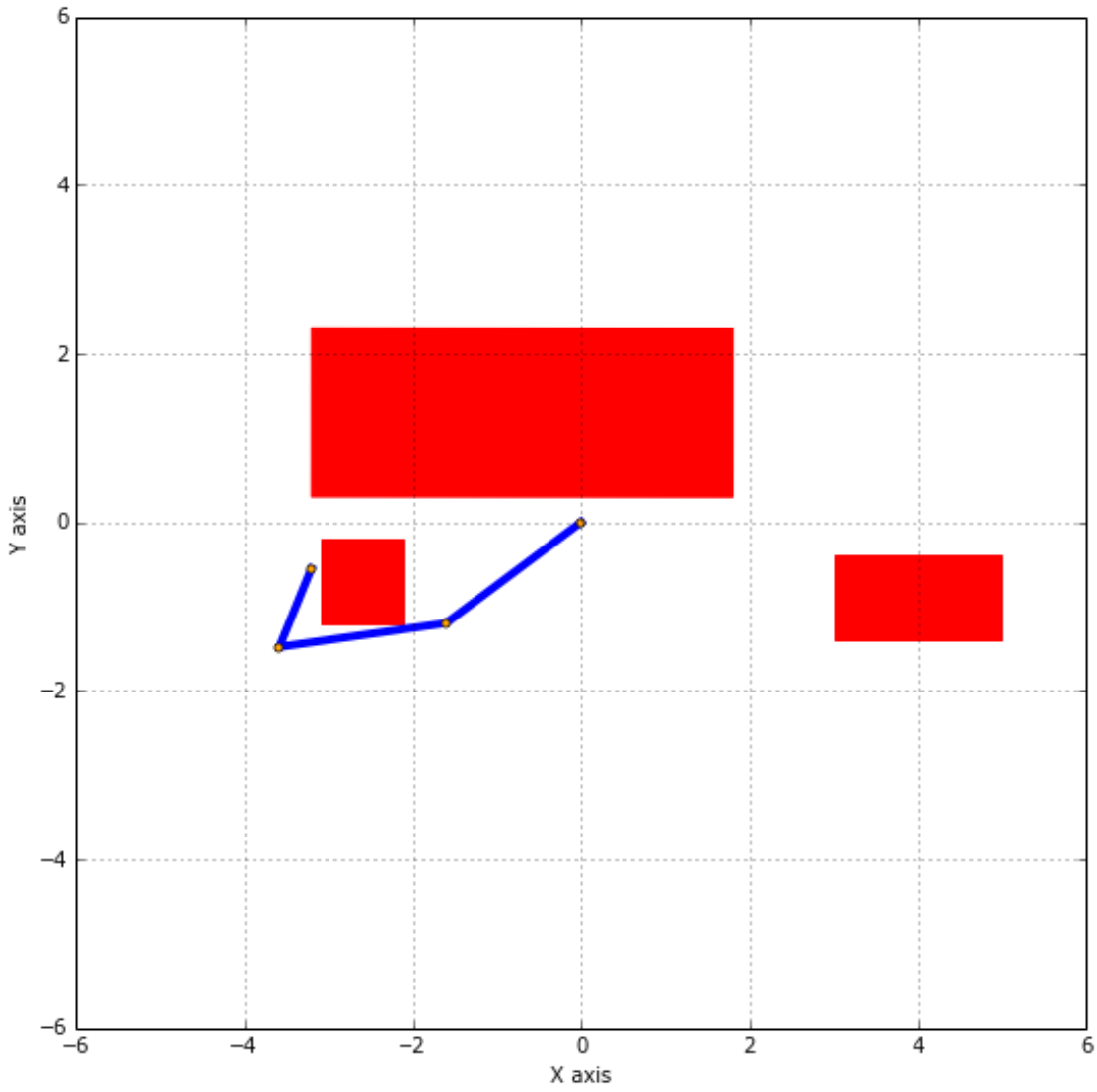


The arm's trajectory and motion behavior of the arm has now changed again.

First we notice how the straight lines in the joint space generated by the connecting method has resulted in continuous- like joint velocities, smoothing the arms motion aside from the moments when the arm changes directions in the joint space.

The second thing we see is how the arm now tends to bounce around obstacles, as the connect-like startegy now only changes tarjectory when it reaches the selected q_{rand} sample, or more often encounters a collision in the next δ direction.

Out[39]: <matplotlib.animation.FuncAnimation at 0x7f09b9d2ed10>



1.4.4 d) Bidirectional:

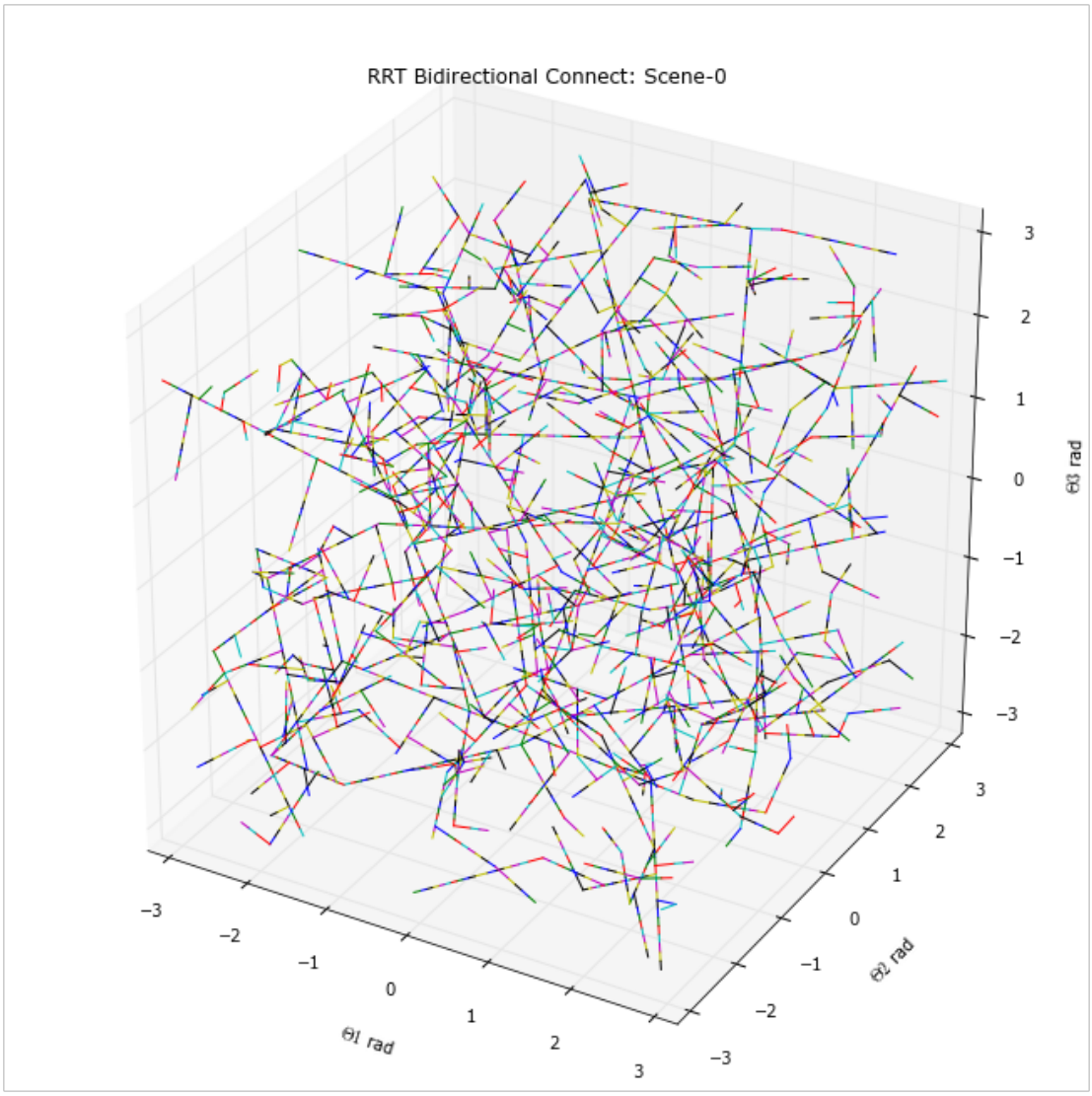
> Instead of growing one tree from the start to goal, we can grow two trees: one from start and the other from goal. Every now and then, select a random configuration and grow both trees with connect- style towards it. If they meet, a path is found. Otherwise, randomly expand. Use the same start/goal configurations(as in (b)) and try for both without and with obstacle cases(see scene2.txt for obstacle info.)

Alright, let's add in one last extension to our RRT, bidirectional growth! We'll start with two trees, one at the goal and one at the start state. We then toggle between the tree we are extending from the start state and the tree being extended from the foal. This alone, however, would still just be a very greedy algorithm, so we limit the probability to sampling the non-currently expanding tree instead of the whole joint space to 5%, like with did with our goal bias before.

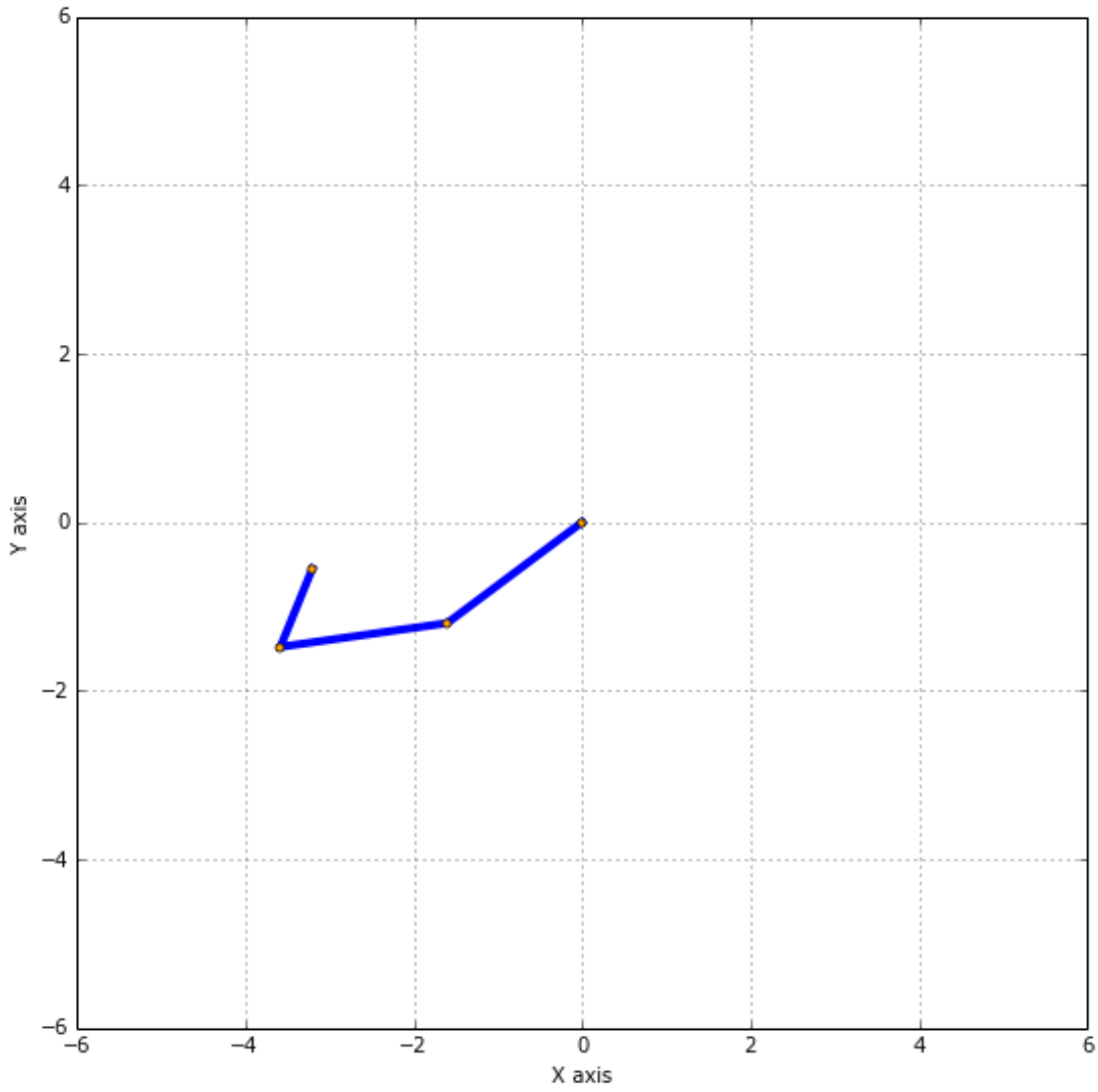
```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
```

```
('delta: ', 0.08726646259971647)
('iterations: ', 5000)
('success: ', True)
```

Out[44]: <matplotlib.animation.FuncAnimation at 0x7f09b9a11fd0>



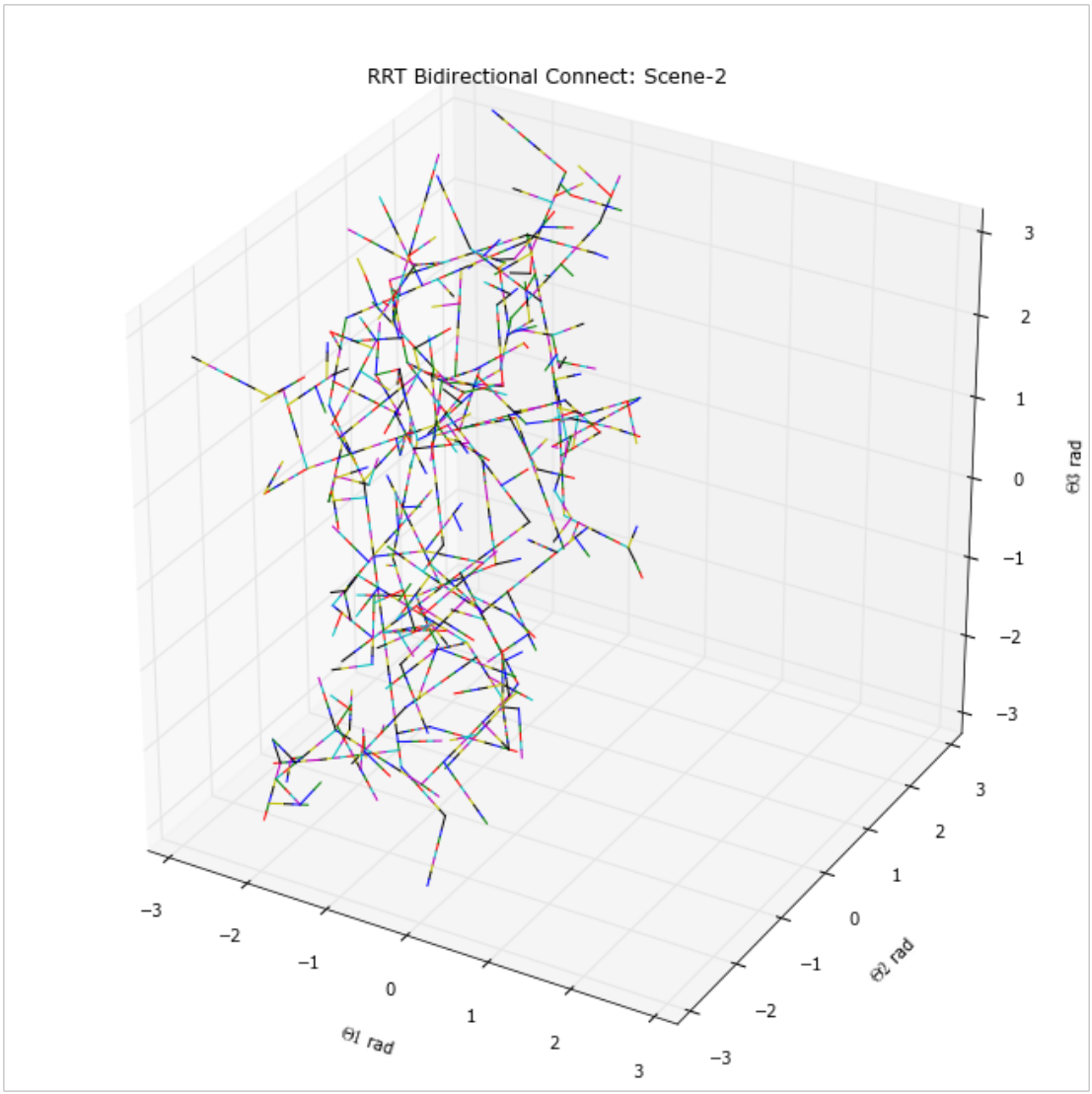
Out[45]: <matplotlib.animation.FuncAnimation at 0x7f09b3d0ea50>




```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('delta: ', 0.08726646259971647)
('iterations: ', 5000)
('success: ', True)
```

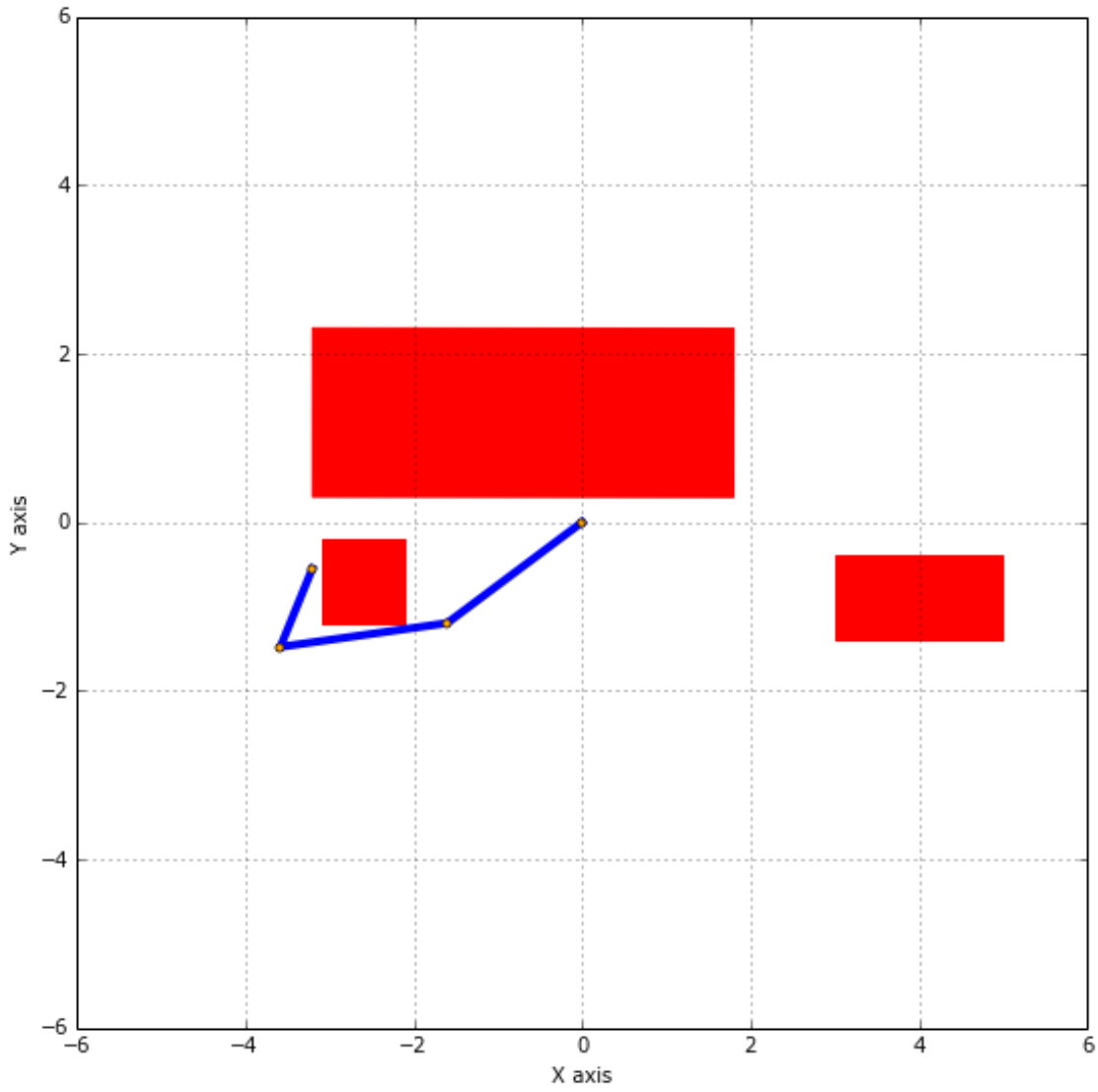
Hmm... with the Bidirectional RRT using connect, it seem to be a bit of a hit or miss scenario. I've witnesed the search time vary drastically: for this same start and goal it may only take about 600 iterations, while other times it might be much slower, as this hairy-looking intersection of two growing trees shows.

```
Out[49]: <matplotlib.animation.FuncAnimation at 0x7f09b32f8cd0>
```



The trajectory we acquire is also a bit of a hit or miss. In terms of performance in completeness, this approach finds a solution well under the maximum limit of iterations, but sometimes misses in optimality. In a few instances it appears that by the time the two trees intersect, there already exists a good deal of exploration overlap. This can result in a bit of backtracing in trajectory as paths through the joint space jostle around the voronoi region overlaped by the two trees before moving towards the root of the other tree. By tuning the biases, we could make the bidirectional search a bit more greedy and lessen the chance of a stagnant overlapping.

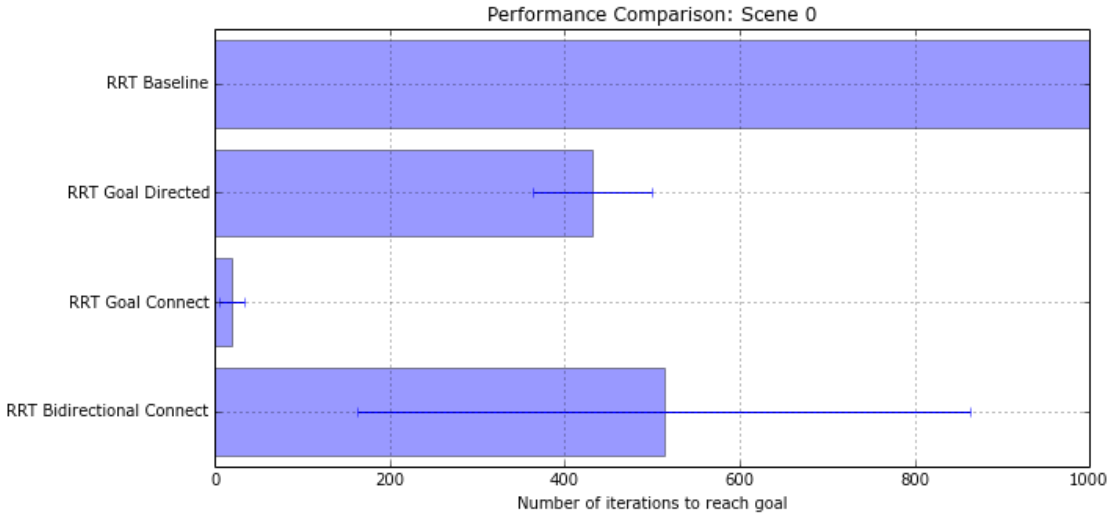
```
Out[50]: <matplotlib.animation.FuncAnimation at 0x7f09b13cc110>
```



1.4.5 e) Analysis:

Compare the results of the 3 approaches - goal-directed, connect and bidirectional RRTs- in terms of number of nodes and timings. Note that the RRT algorithms have several parameters such as the number of iterations, step size and etc. You might need to calibrate these values appropriately from problem to problem. Explain which parameters you think played the most important role in the outcomes and why.

```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('tolerance: ', 0.17453292519943295)
('delta: ', 0.08726646259971647)
('iterations: ', 1000)
```

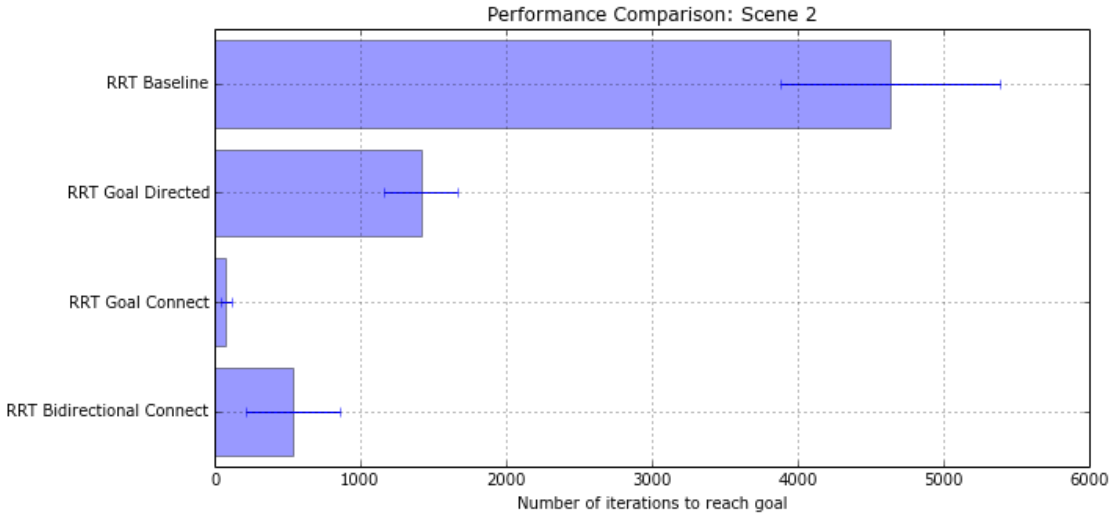


In the figure above we see the mean number of iterations to reach the goal for each RRT approach as shown previously over 10 trials. The whiskers are the standard deviations over the number of trials. For Scene 0, that was devoid of all obstacles, the trees were permitted to fully explore the entire joint space for the goal condition. For Scene 0, we limited our baseline to 5000 iterations, where an iteration considered to be an attempted to extension towards a sample point, whether successful or not.

What can be shown is that the Goal Connect method was by far the most efficient in terms of number of iterations, with Goal Directed coming in second, Bidirectional Connect third, and Baseline in last. Here the Baseline appears to max out consistently over 1000 iterations, most likely never satisfying our goal criteria. The rest of the methods that happen to use a goal bias for sampling appear to do much better. The Goal Connect offers a excellent combination for greed search and rapid growth to the goal, while Goal Directed lacks this connect capability and must more slowly expand into the joint space.

For Bidirectional Connect, we see the addition of a second tree has done less to improve iteration count. This is most likely due to the fact that the probability of one tree extending and connecting to a second random node on the second tree given a 5% is less probable that growing towards a static goal with the same about of bias. This distribution of performance also seem to be more wide spread.

```
('start: ', array([ 0.14,  1.45,  1.  ]))
('goal: ', array([-2.5, -0.5, -2.1]))
('tolerance: ', 0.17453292519943295)
('delta: ', 0.08726646259971647)
('iterations: ', 5000)
```



For the same results given the Scene 2, we see that now Bidirectional Connect has improved in comparison to Goal Directed, but still worse than Goal Connect. The obstacles in Scene 2 that volumetrically restrict the joint space help confine the growth of the Bidirectional Connect. With this increase in exploration density, this improves the probability that the two trees converge sooner when they fall in reach of each other.

We should note that some of the main parameters, such as the δ used for extension or the probability of the extent to goal bias could be adjusted and tuned for each approach or scene. The hazards encountered when radically changing these parameters however include: when the given delta is to small, the time required to densely explore the joint space increases inversely, while making the delta to large can result in impossible trajectories as the arm could discreetly teleport around surrounding obstacles. When tweaking the goal bais, we have to be careful, as increasing the bias to far can lead to suboptimal solutions or traps from local minimums, while keeping it so small more or less will get you the same nave and lengthy sampling as with the Baseline approach.

1.5 5) Manipulation Planning - RRT with Task Constraints

> The problem with the classical approach of configuration space RRTs is that it is not possible (or difficult, if possible) to impose workspace constraints on the trajectories in configuration space. For instance, if we want a robot arm to open a drawer, this requires the end-effector to move only horizontally, keeping the same height and vertical location. This constraint can be easily enforced as we build the tree by making sure that each node we add in the tree satisfies the constraint.
>
> How do we do it? The approach is similar to that of a normal RRT sampling. Select a random target, find closest neighbor and expand towards it to find a candidate node q_c. However, there is no guarantee that this candidate node fulfills the workspace criteria. Now, we can compute an error between the pose of the candidate node p_c and the workspace criteria, and move the node q_c to a goal node q_g with Jacobian control such that the error is diminished. See the papers [1] and [2] for more details.
>
> Move the same 3-DoF arm(as in problem 3) such that it would move from the start configuration $q_s = (1.5707, -1.2308, 0)$ to the goal configuration $q_g = (1.5707, 1.2308, 0)$ with the constraint end-effector y location fixed at $y = 3$ line. Provide the videos of the trees growing and the arm moving(upload the movies to your repository and include snapshots in your summary report.)

Lets start by importing our code for Constrained RRT with a Goal Directed bias. The task constraint will ensure our trajectory remains in bound with the specified limits, keeping the pose of the endefector along the line $y = 3$ in the workspace. The goal bias will help ensure we latch onto our exact goal.

To do this we first pass our new q_c sample to a new subroutine that calculates the error the workspace point p_c with respect to the task constraint, i.e. the difference in the endefector y displacement and the $y = 3$ axis. This error is then scaled and multiplied by and vector of ones to produce $\dot{s}$ that is then multiplied with the inverse jacobian to gain $\dot{q} = J^{-1} \cdot \dot{s}$. Then we re-assign our q_c such that $q_c = q_c - \dot{q}$ in order to move it back towards the constrained manifold. This is done iteratively until the error is below a 0.01 threshold.

After this is done, we'll still need to do some filtering, as thanks to singularities with in the jacobian, we would otherwise accrue a large number of outliers well outside our original joint space. To do this we simply check that the new corrected q_c is no further that around 10 times the chosen δ with respect the the original q_near we were attempting to extend from.

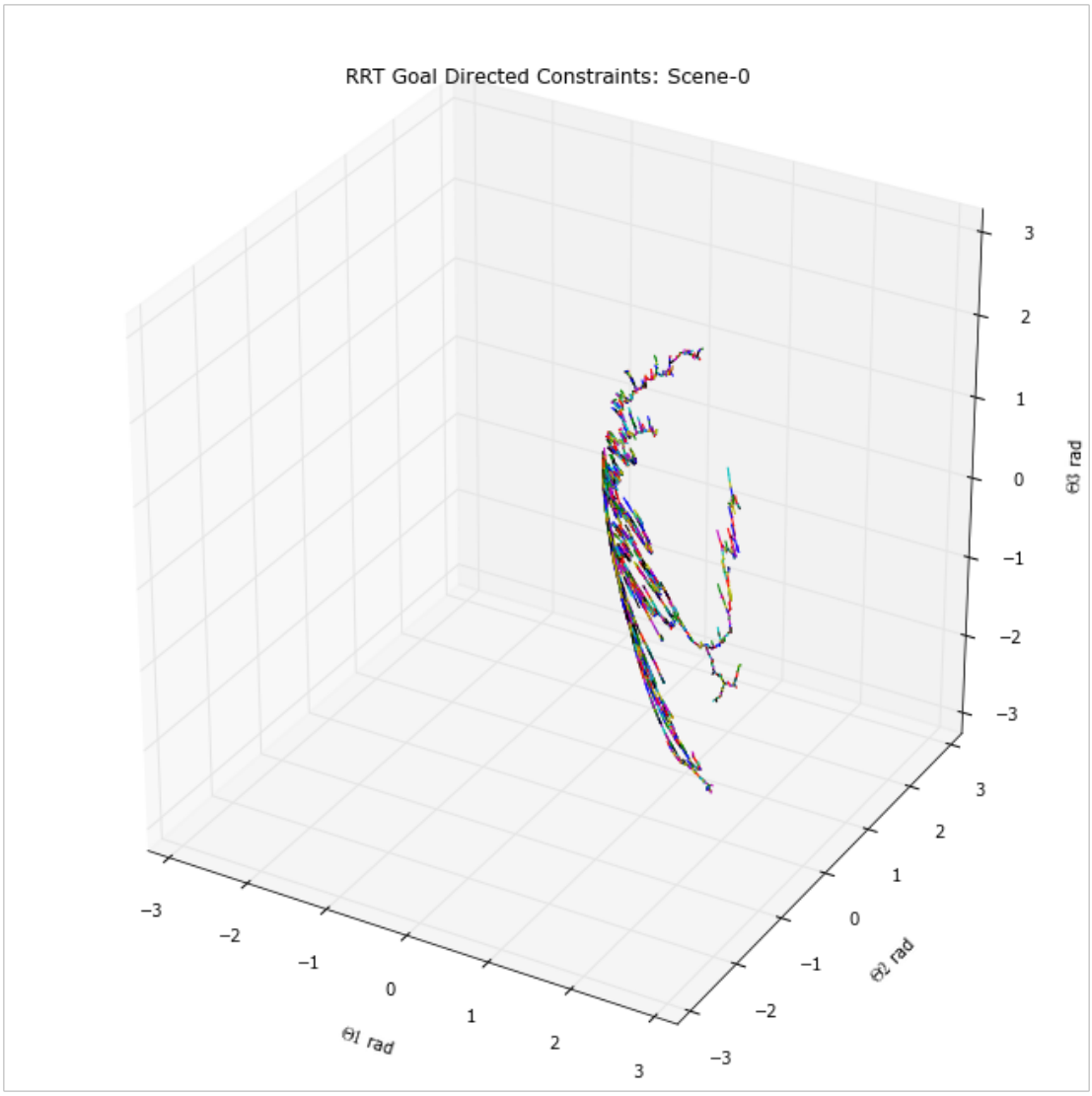
```
('start: ', array([ 1.5707, -1.2308,  0.    ]))
('goal: ', array([ 1.5707,  1.2308,  0.    ]))
('delta: ', 0.017453292519943295)
```

Out[13]:
```
True
```

```
('n_iters', 3346)
```

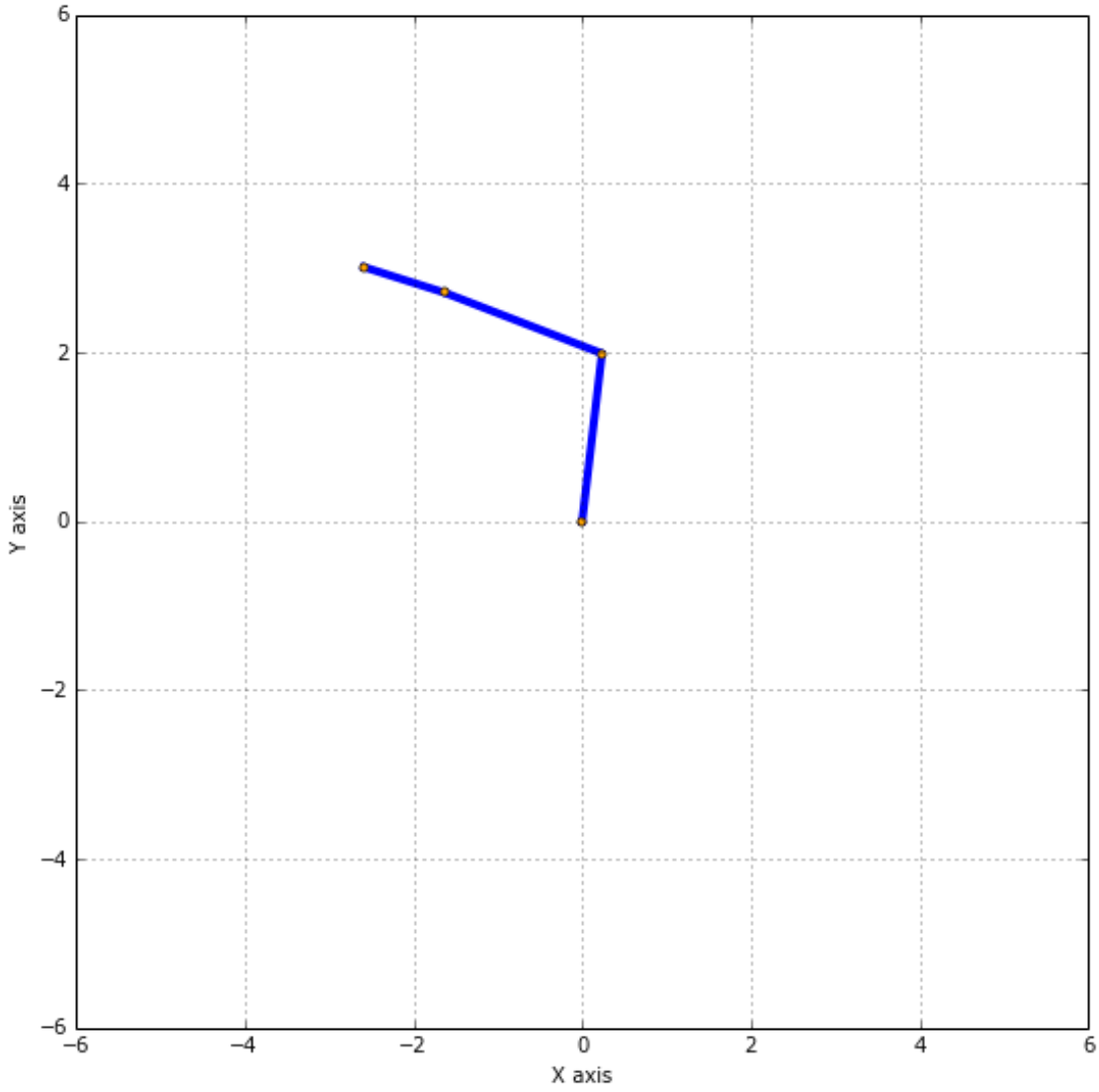
Below we see the projected workspace constraint manifest as a curved manifold with in the 3 DOF joint space. For this case, in order to produce finer trajectories, we've reduced our delta from 10° to just 1°. This will increase the number of required iterations, but should mitigate the jerking motions of the arm as it passed along the $y = 3$ task constraint.

Out[16]:
```
<matplotlib.animation.FuncAnimation at 0x7fc931554f50>
```



Below is the final result of the arm trajectory as it moves to the mirrored refection of its original pose.

```
Out[15]: <matplotlib.animation.FuncAnimation at 0x7fc9308b0350>
```



1.6 References

[1] Stilman, Mike. "Task constrained motion planning in robot joint space." Intelligent Robots and Systems, 2007. IROS 2007. IEEE/RSJ International Conference on. IEEE, 2007.
[2] Kunz, Tobias, and Mike Stilman. "Manipulation planning with soft task constraints." Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on. IEEE, 2012.

1.7 Participation

For this problemset we've divded the work appropriately:
Problem 1: Siddharth
Problem 2: Yosef
Problem 3: Varun
Problem 4: Ruffin
Problem 5: Ruffin & Varun