Technische Universität Berlin

Robotics Assignment 5

Anupama Rajkumar 415252

Yang Xu 373470

Ke Zhou 415253

1 Gaussian Sampling around start and goal position

Rapidly-exploring random tree (RRT) planners solve a particular planning task by incrementally growing a tree, starting from a specific location. Configuration space exploration is guided toward the largest Voronoi region associated with the existing samples. This drives exploration toward large unexplored regions. As the Voronoi regions of samples become approximately equal in size, the exploratory behavior gradually shifts from expansion of the tree to refinement [1].
Voronoi bias without goal-directed bias explores the entire configuration space. Various sampling strategies can be used. In our experiments, we compared uniform and gaussian sampling. With uniform sampling, the samples are placed uniformly all through the free space while in Gaussian sampling the samples are concentrated near the obstacle boundary.
The Rapidly exploring Random Tree (RRT) with uniform distribution sampling over the free space takes a lot of time to merge the trees from the start and goal configuration, due to the random growth of tree and the huge free space. When replacing the uniform distribution sampling with Gaussian distribution sampling around the start and goal nodes, the growth of both trees will head to the start or goal nodes respectively. before finally merging together.

1.1 Sampling in RRTConConBase vs Sampling in YourPlanner

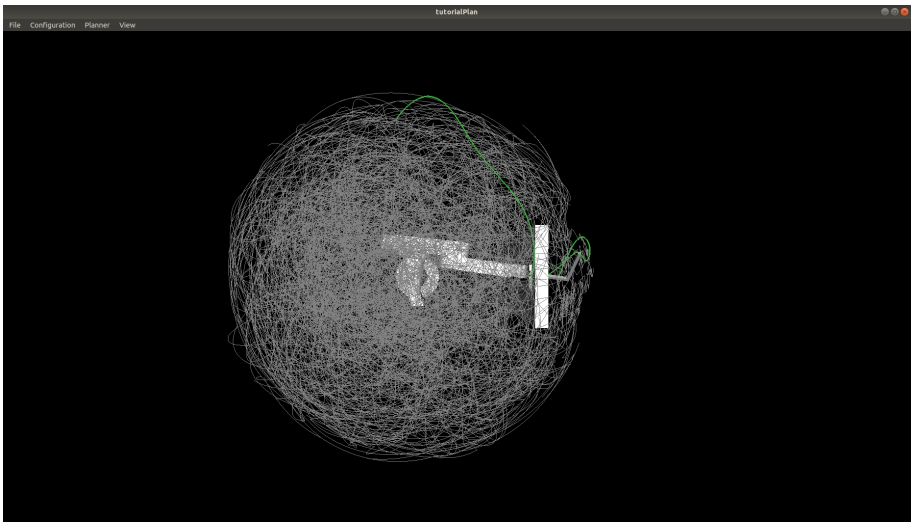

Figure 1: Uniform Sampling in RRTConConBase - not goal biased

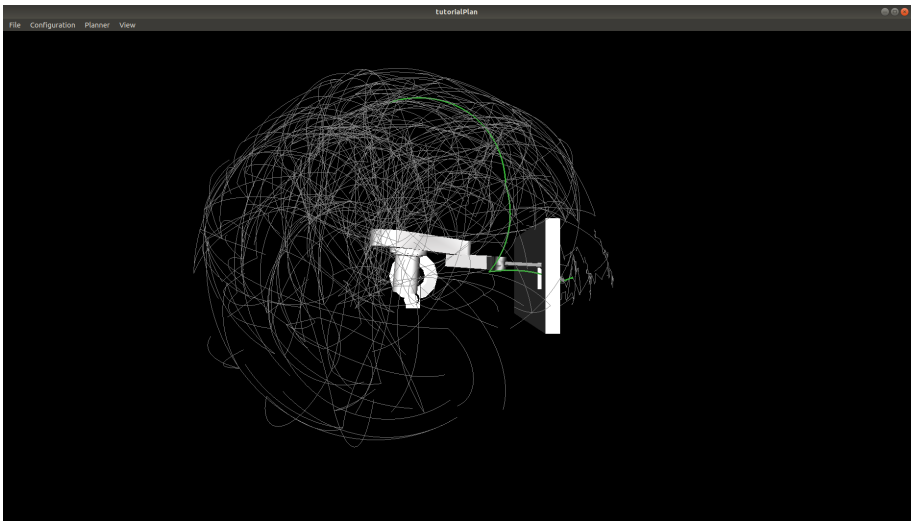

Figure 2: Gaussian Sampling in YourPlanner - goal biased with standard deviation = 1.0

1.2 Effect of Standard Deviation in Gaussian Sampling

Standard deviation of the gaussian play an important role in deciding where the samples will be placed. Bigger standard deviation would mean a broader gaussian and vice versa. Smaller value of standard deviation makes sampling points very close to each other as the width of the gaussian is reduced. It takes longer to search the path but when it's found, the path size is small and very smooth.

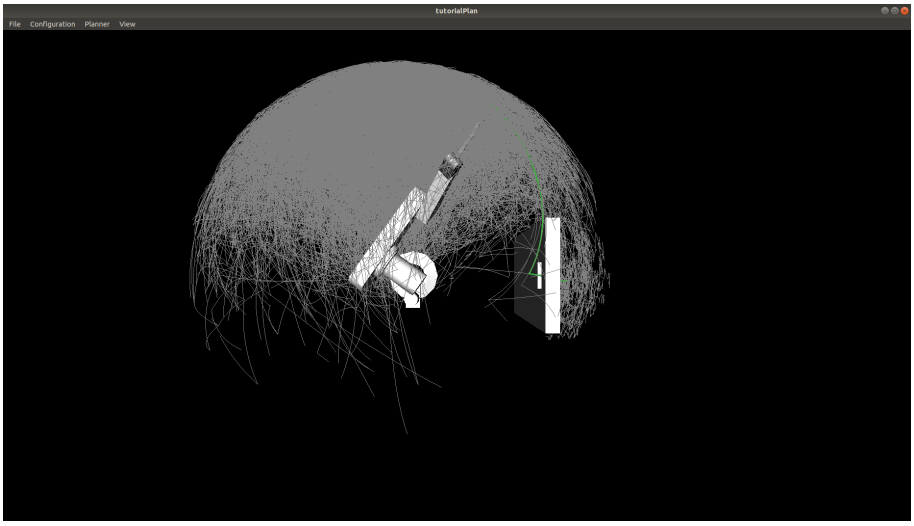

Figure 3: Gaussian Sampling in YourPlanner - around start and goal with standard deviation = 0.5, time needed : 100 s

On the contrary, on increasing the standard deviation, the time needed to find the path reduces

but the path is longer and not smooth.

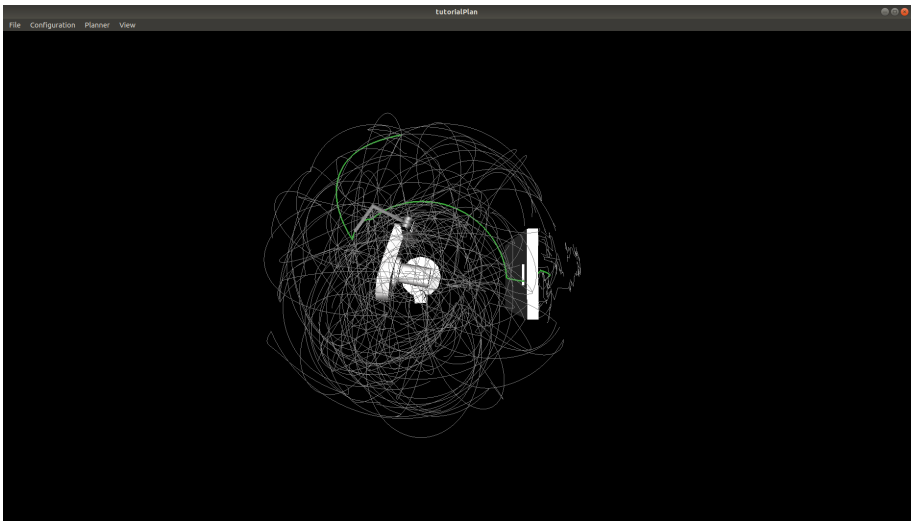

Figure 4: Gaussian Sampling in YourPlanner - goal biased with standard deviation = 0.5, time needed : 1.6 s

In our implementation, we found this value heuristically. We have set this value at 2.0 looking at the trade-off between accuracy and speed.

1.3 Gaussian Sampling Code Snippet

We used a user defined GaussainRandom function. We used the implementation from Util.cpp used in previous assigment. We did not use the GaussianRandom from Robotics Library. The reason for this was that we found that there were many function call overheads in the implementation of Robotics Library which slowed down the implementation.

```
216 //-------------------------------------------------
217
218 //c&p from assignment 4 Util.cpp
219 double
220 YourPlanner::gaussianRandom( double mean, double std ) {
221         const double norm = 1.0 / (RAND_MAX + 1.0);
222         double u = 1.0 - rand()*norm;
223         double v = rand()*norm;
224         double z = sqrt(-2.0*log(u))* cos(2.0*M_PI*v);
225         return mean + std*z;
226 }
227
228
```

Figure 5: Gaussian Random Implementation

For goal biasing, we modified solve() function in which we added the call to gaussianRandom() concentrated along the start and goal. Since for Uniform random sampling, lots of useless nodes in the free space are taken into concern, which is not optimal. Therefore, we only sample around the start point and goal point to avoid large amount of useless nodes.

```
while ((::std::chrono::steady_clock::now() - this->time) < this->duration)
{
    //First grow tree a and then try to connect b.
    //then swap roles: first grow tree b and connect to a.
    for (::std::size_t j = 0; j < 2; ++j)
    {
        ::rl::math::Vector dof(this->model->getDof());
        if(j==0){
            dof = *this->start;                 //goal based gaussian sampling
            //dof = this->sampler->generate();
            for(int i = 0; i < chosen.size();i++){
                chosen[i] = YourPlanner::gaussianRandom(dof[i], 2.0);
            }
        }else{
            dof = *this->goal;
            //dof = this->sampler->generate();
            for(int i = 0; i < chosen.size();i++){
                chosen[i] = YourPlanner::gaussianRandom(dof[i], 2.0);
            }
        }

        //Find the nearest neighbour in the tree
        Neighbor aNearest = this->nearest(*a, chosen);

        //Do a CONNECT step from the nearest neighbour to the sample
        Vertex aExtended = this->extend(*a, aNearest, chosen);

        //If a new node was inserted tree a
        if (NULL != aExtended)
```

Figure 6: Gaussian Random Implementation

2 Modifying Step Size

2.1 Adjusting the step size by changing delta

Step size is one of the most important parameter in RRT. It configures the step width for the connection attempt. We tried different step size from $0.5f$ to $1.5f$, but these changes don't have an impressive impact on the benchmark.
Setting the value of step size is a trade-off question, if the Step Size is set too large, in principal, it will accelerate the whole process. However, it will lead to problems like the robotic arm might collide with the obstacle. Also, it may be hard to find the right path since lots of free spaces might be ignored, which means the accuracy will be decreased. On the other hand, a smaller step size increases the possibility to find the right path to the goal. But a bunch of nodes will be added to the tree, which will also decrease the efficiency.

2.2 Changing Interpolation Step Size in Extend

As explained in [2], RRT consists of two techniques, connect and extend in order to reach the goal. This is in-tune with the explore and exploit principle which RRT uses. The extend operation is such that it selects the nearest vertex which is determined by the nearest() function. When the nearest neighbor is found and if it is not the final goal, a new vertex and edge is added till the neighbor node. This is how node by node the tree reaches the goal.
In extend function in RrtConConBase, distance from the nearest neighbor is determined. This is used to calculate the step size of extend. Interpolate() function is used to move to the neighboring node by taking step width as specified by the step size. Irrespective of whether the space is free or not, the step size remains constant. This makes extend function very slow.
As per the extension we added, we checked for collision between the current node and it's nearest neighbor, if no collision is detected, the node interpolates to the nearest neighbor by taking longer step size. If collision is detected, the step size is reduced to the value of delta. This reduces the time needed by extend function.

2.3 Code Snippet of Modified Extend

```
RrtConConBase::Vertex
YourPlanner::extend(Tree& tree, const Neighbor& nearest, const ::rl::math::Vector& chosen)
{
  //your modifications here
    ::rl::math::Real distance = nearest.second;
    ::rl::math::Real step = (::std::min)(distance, this->delta);

    ::rl::plan::VectorPtr next = ::std::make_shared< ::rl::math::Vector >( x: this->model->getDof());

    //this->model->interpolate(*tree[nearest.first].q, chosen, step / distance, *next);

    if(!this->model->isColliding())
    {
        this->model->interpolate(*tree[nearest.first].q, chosen, distance,  &: *next);
    }
    else
    {
        this->model->interpolate(*tree[nearest.first].q, chosen, step,  &: *next);
    }

    this->model->setPosition(*next);
    this->model->updateFrames();

    if (!this->model->isColliding())
    {
        Vertex extended = this->addVertex( &: tree, next);
        this->addEdge(nearest.first, extended,  &: tree);
        return extended;
    }

    tree[nearest.first].fails +=1;
    return NULL;
}
```

Figure 7: Extend Implementation

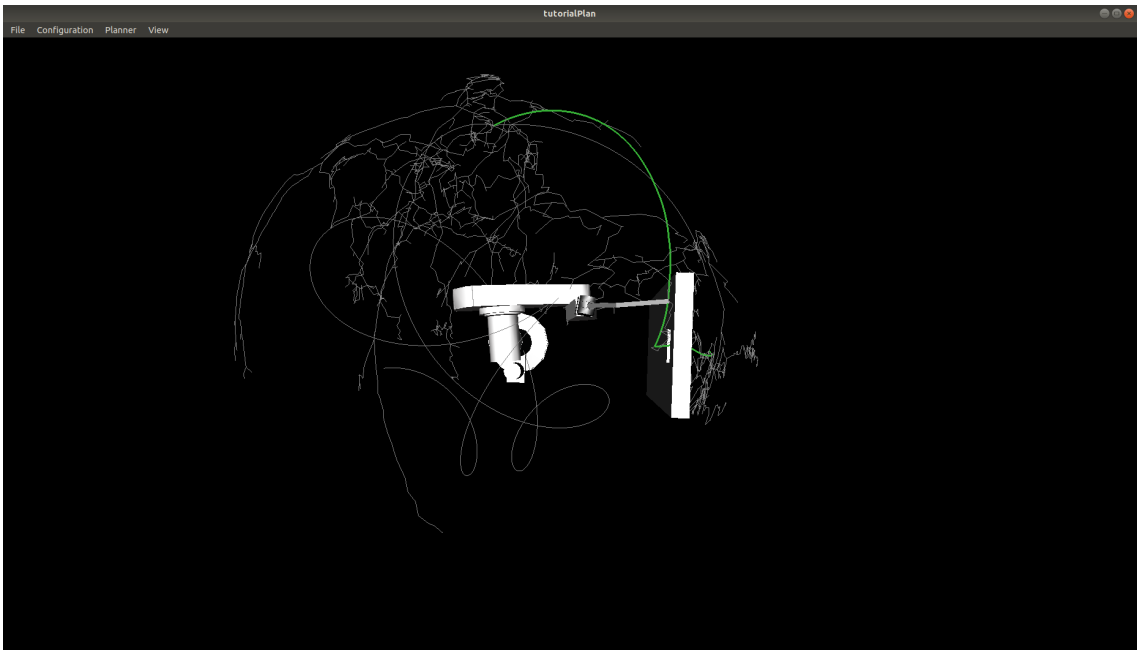

Figure 8: Extend Implementation, time to find path : 523 ms

3 Nearest Neighbor Selection

3.1 Nearest Neighbor Selection with the guide of tree growth direction

During the growth of the two trees, each pair of node and edge addition to the trees needs to select a nearest node from the respective tree. And, this process consumes a lot of computation time. As the Voronoi space gets more and more crowded, the process gets slower. Such that, given the random chosen nodes, it can quickly give a set of nodes which is much more close to the chosen one. Thus, the selection process does not have to iterate all nodes in the tree. But, it seems so hard to design such a property of the tree.

Moreover, this heuristic guide provided by gaussian sampling using goal biasing gives us a valuable clue to quickly choose the nearest node from the respective tree. Since the growth of both trees will head to the start or goal nodes respectively, this means that the nodes added later will be more closer to the respective tree. This observation results in a strategy to improve the nearest node selection process. That's to narrow down the search space to the nodes added recently.

3.2 Nearest Neighbor Selection Code Snippet

```
64 RrtConConBase::Neighbor
65 YourPlanner::nearest(const Tree &tree, const ::rl::math::Vector &chosen)
66 {
67   //create an empty pair <Vertex, distance> to return
68   Neighbor p(Vertex(), (::std::numeric_limits<::rl::math::Real>::max)());
69
70   // get the number of vertices in the tree
71   int numVer(::boost::num_vertices(tree));
72
73   // only consider the recently added 200 nodes when search for the nearest node
74   int rangeN(200);
75
76   int j(0);
77   //Iterate through all vertices to find the nearest neighbour
78   for (VertexIteratorPair i = ::boost::vertices(tree); i.first != i.second; ++i.first)
79   {
80     //if (tree[*i.first].fails > 10)      // set a threshold of fails... not very useful acutally
81       //continue;
82       if(j > numVer - rangeN){
83           ::rl::math::Real d = this->model->transformedDistance(chosen, *tree[*i.first].q) ;
84           if (d < p.second)
85           {
86               p.first = *i.first;
87               p.second = d;
88
89           }
90       }
91       j++;
92   }
93
94   // Compute the square root of distance
95   p.second = this->model->inverseOfTransformedDistance(p.second);
96
97   return p;
98 }
99
```

Figure 9: nearest search with tree growth guide Implementation

4 Comparison: ConCon and ExtCon

We implemented and tested our extensions for following two configurations:
1. ConCon: Both trees connect()
2. ExtCon: Tree from start - extend() and tree from goal - connect()

With ConCon, with our existing extensions, we observed that the path is longer and the time taken to find the path is higher. Connect consists of multiple extend steps as mentioned in [2] where the tree takes small steps to reach goal. This accounts for increased time in finding the path.

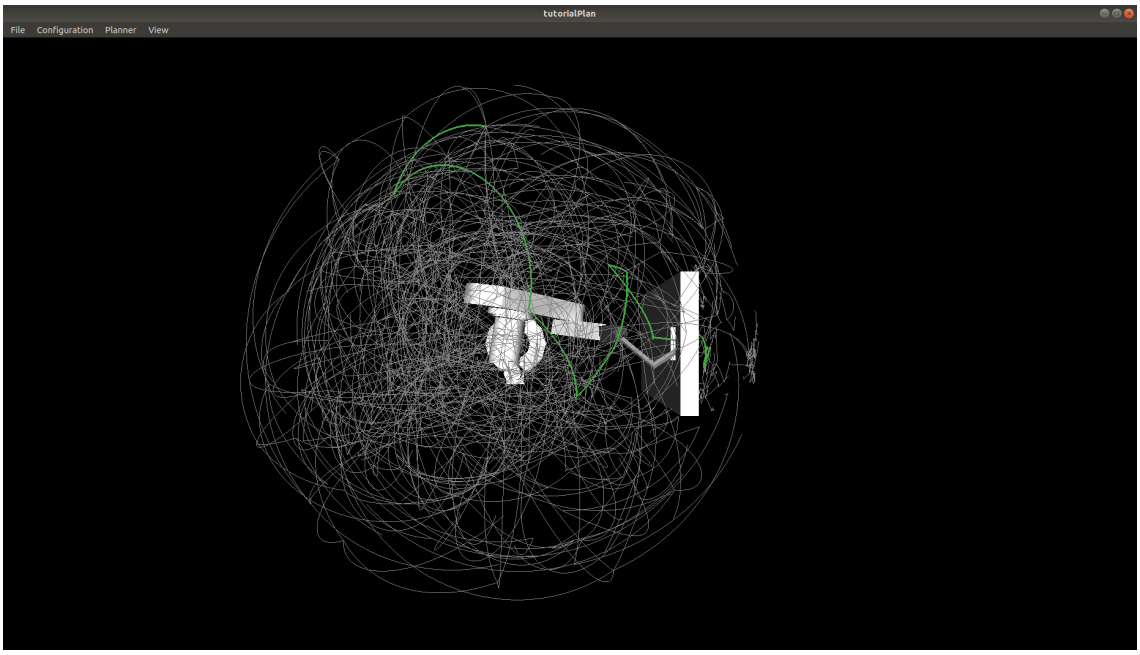

Figure 10: ConCon Implementation, time to find path : 5 s

With ExtCon, with our existing extensions, we observe that the free space is explored quickly because of gaussian sampling and increased distance in absence of collision and hence we get shorter path and quicker computation.

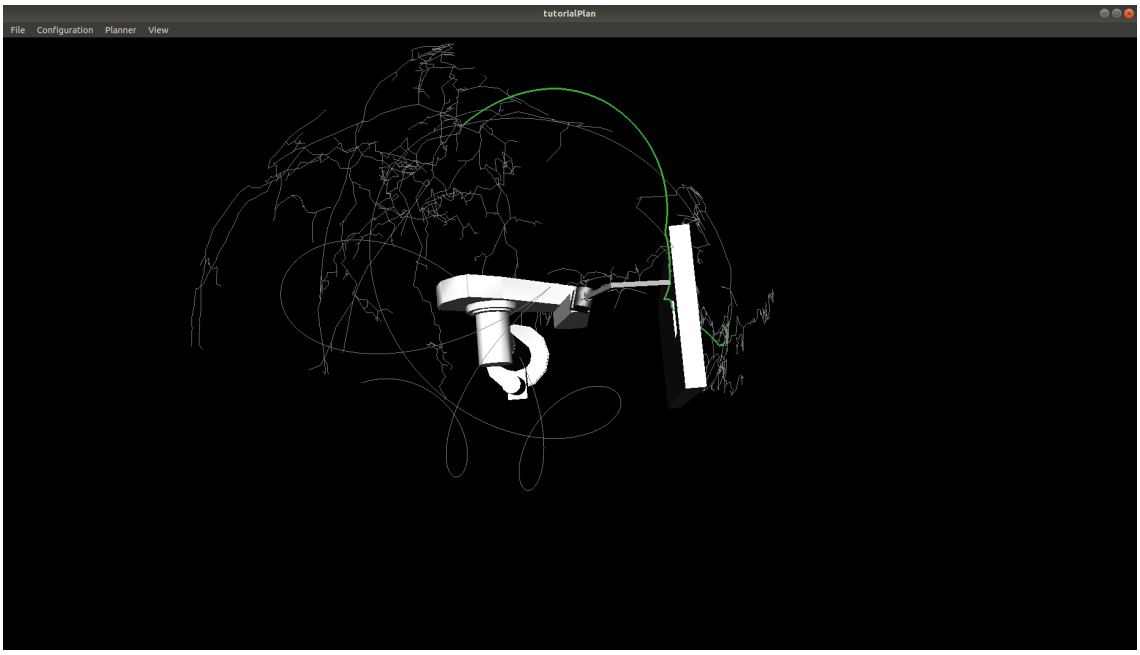

Figure 11: ExtCon Implementation, time to find path : 650 ms

4.1 Conclusion

Looking at the improvement in performance with ExtCon as compared to ConCon we implemented RRT planner with ExtCon.

5 Non Performing Extensions

While testing and trying out various extensions we found some implementations that did not really improve performance or even if they improved performance were highly application specific. Following sections discuss few of such non performing extensions.

5.1 Reduce the Nodes Near Start Point

We implemented a way to reduce the useless nodes. Most of the nodes around the start position are useless because there are no obstacles around the start position.
In the RrtConConBase, the program swaps the start and goal every time to build the tree from start and goal and try to connect, which leads to huge amount of useless nodes in free space and also increases the running time of nearest search. However, if we assume that we know that the obstacles are only around the goal position (like in our case), we can sample more nodes around the goal position instead of in the start position. Therefore, we implemented that for every 49 sampling around the goal position, we will sample only once around the start position. This leads to a bunch of useless nodes in free space around the start to be reduced, and the running time of nearest search is reduced in a large way.

We managed to find the path in the order of 100 ms-200 ms following this implementation. But, we consider this to be a non-performing extension because this solution is highly tailored to the application and is likely to fail when position of obstacle changes.

```
 unsigned int i = 1;
while ((::std::chrono::steady_clock::now() - this->time) < this->duration)
{
 //First grow tree a and then try to connect b.
 //then swap roles: first grow tree b and connect to a.
 for (::std::size_t j = 0; j < 2; ++j)
 {
   //Sample a random configuration
  // this->choose(chosen);
 double p = rand()/(double)RAND_MAX;
 ::rl::math::Vector dof(this->model->getDof());
        dof = *this->goal;  //goal bias gaussian sampling

    for(int i = 0; i < chosen.size();i++){
        chosen[i] = YourPlanner::gaussianRandom(dof[i], 1.0f);
     }
 Vertex aConnected;
 // every 50 times sample one time around the start, since the obstacle is around the goal
 if(i%50==0){
        swap(a, b);
       //Find the nearest neighbour in the tree
        Neighbor aNearest = this->nearest(*a, chosen);

          //Do a CONNECT step from the nearest neighbour to the sample
           aConnected = this->extend(*a, aNearest, chosen);
 }
 else if(i%50==1){
        swap(a, b);
       //Find the nearest neighbour in the tree
        Neighbor aNearest = this->nearest(*a, chosen);

          //Do a CONNECT step from the nearest neighbour to the sample
            aConnected = this->connect(*a, aNearest, chosen);
 }
 else{

   //Find the nearest neighbour in the tree
  Neighbor aNearest = this->nearest(*a, chosen);

   //Do a CONNECT step from the nearest neighbour to the sample
   aConnected = this->connect(*a, aNearest, chosen);
 }
 i++;
```

Figure 12: Code Snippet of modified solve()

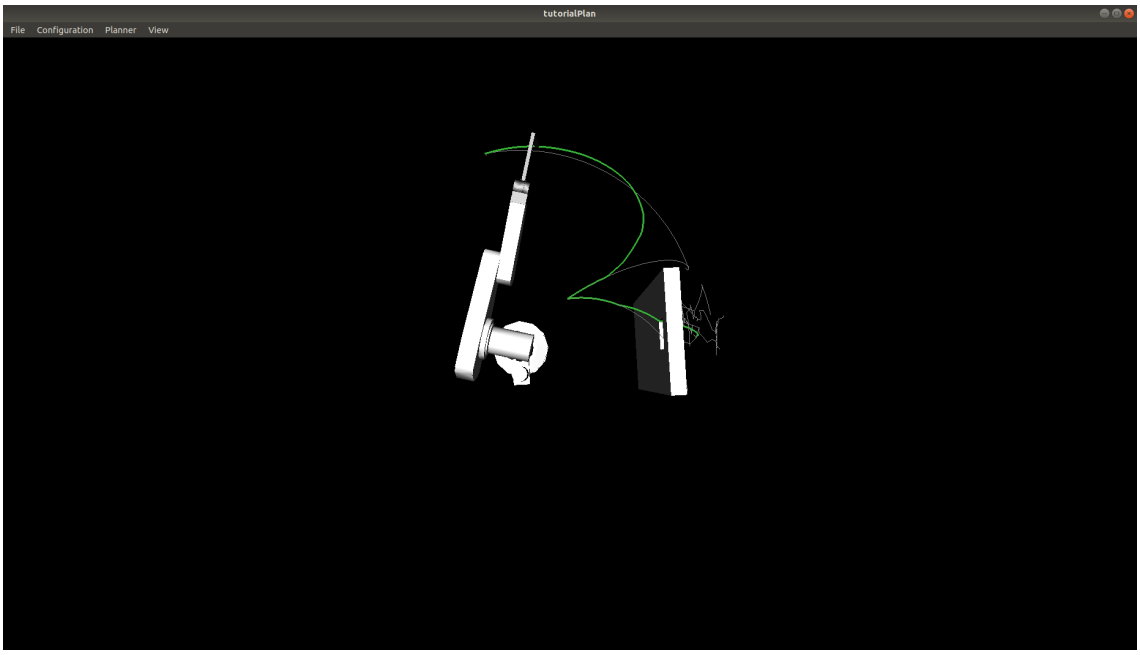

Figure 13: Performance after the change in solve(), time taken to find path : 160 ms

5.2 Removing the Exhausted Nodes

In the research paper from J Cortes et al.[3], mentioned 2 strategies to deal with the exhaust nodes, we tried their first strategy. For each node, the number of consecutive times that its expansion fails is counted. When the counter reaches a given limit number l, the node is considered to be exhausted and it is no longer selected. However, the limit l is difficult to determine, depends on how large the free space where the nodes was trapped, the fail time can be quite different. Therefore, we implemented it in a more global way, e.g. we count the failures for each node globally instead of consecutively, if the sum of failures is larger than a certain threshold, these nodes would be very likely to be an exhaust node and shouldn't be extended.
But such an implementation doesn't improve the benchmark, we tried different threshold from 10 - 50, but we didn't notice a great progress. The possible reason might be that we didn't find an appropriate threshold for the model. Moreover, our modification of the global counter may also delete some useful nodes (even if they fail so many times) so the program need to find a new path again.

```
RrtConConBase::Neighbor
YourPlanner::nearest(const Tree &tree, const ::rl::math::Vector &chosen)
{
    //create an empty pair <Vertex, distance> to return
    Neighbor p(Vertex(), (::std::numeric_limits<::rl::math::Real>::max)());
    //Iterate through all vertices to find the nearest neighbour
    for (VertexIteratorPair i = ::boost::vertices(tree); i.first != i.second; ++i.first)
    {
        if (tree[*i.first].fails > 100)         // set a threshold of fails... not very useful acutally
            continue;
            ::rl::math::Real d = this->model->transformedDistance(chosen, *tree[*i.first].q) ;
            if (d < p.second)
            {
                p.first = *i.first;
                p.second = d;

            }
    }

    // Compute the square root of distance
    p.second = this->model->inverseOfTransformedDistance(p.second);

    return p;
}
```

Figure 14: Code Snippet of exhausted node in neighbor()

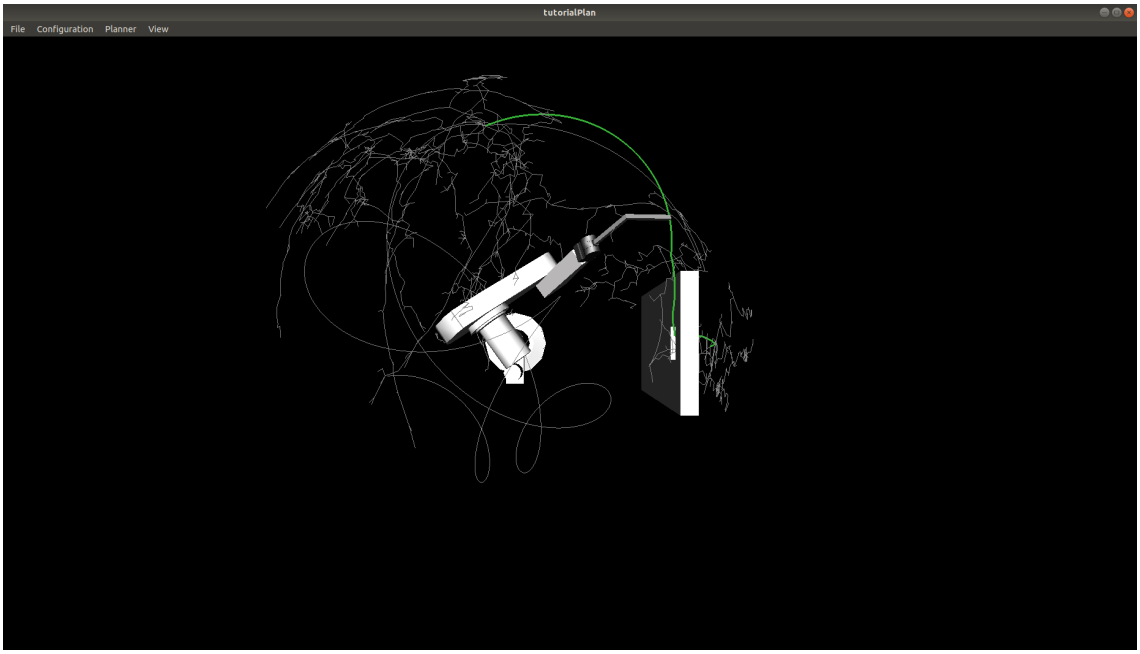

Figure 15: Performance after exhausted node check in neighbor(), time taken to find path : 3 s

6 Performance Evaulation of Final Algorithm:

	RRTConCon	RRTConCon(reversed)	YourPlanner	YourPlanner(reversed)
avgT	39.86 s	59.699 s	311.25 ms	398.14 ms
stdT	29.271	32.55	24.24	28.53
avgNodes	8871	12211	1018	1102
avgQueries	456731	606822	3566	5094

6.1 Conclusion

We added following extensions and observed the performance metrics of these with respect to the RrtConConBase implementation:
a. Gaussian Sampling around start and goal position instead of uniform sampling
b. Changing step size for extend
c. Modifying the way nearest neighbors are decided

As can be seen from the result with the given extensions and using ExtCon instead of ConCon, significant performance improvement can be achieved. However, there is always a trade-off between accuracy and performance. While increasing step size and increasing the standard deviation of the sampler reduces the time needed for finding the path, at the same time it also poses a compromise on the accuracy.

7 References:

1 Markus Rickert, Arne Sieverling, and Oliver Brock, "Balancing Exploration and Exploitation in Sampling-Based Motion Planning"
2 James J. Kuffner, Jr, Steven M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning"
3 J Cortes et al. Molecular disassembly with RRT-like algorithms

8 Table with participation tasks:

Student Name	Ext1	Ext2	Ext3	Ext1-Docu	Ext2-Docu	Ext3-Docu
Anupama Rajkumar	x	x		x	x	x
Yang Xu	x		x	x	x	x
Ke Zhou			x	x	x	x