Drone Motion Planning
to
Follow and Record Snowboarder

Matthew McGraw

Introduction

The goal of this project is to create a motion planning system that allows a drone to follow a snowboarder (or skier) while always having the camera pointed at the individual. The snowboarder will record a path that can be applied to the motion planner, which will then output the motion plan for the drone. In this planner, we are assuming negligible crosswinds that may affect the movement of the drone, as well as the fact that the physical environment is known by the motion planner. This environment can be any mesh, either recorded from LIDAR data, or created virtually. For this project, the environments will be created virtually using Unreal Engine 5, which will allow us to test multiple various environments for the drone to traverse through. We will be using the SST planner from the Open Motion Planning Library (OMPL) to create the motion plan, and using the Flexible Collision Library (FCL) the check for collisions between the drone and the environment. Once the plan is created, we can apply our motion plan to the drone in Unreal Engine and view its performance.

After more more than 60+ hours struggling to use C++ and implement OMPL and FCL, for the sake of time and in order to complete a majority of the project, I decided it was best to switch from the attempt of using C++ to using Python, a language I'm far more familiar with. I had originally chosen to use C++ because I was told it would be simpler to finish the project using OMPL than starting from scratch. Since I was unable to implement OMPL in C++, would have to implement the SST planner from scratch, however, but since I had written each of my planners throughout the semester myself, I felt confident in my ability to complete the project in time using Python, even though the attempts at using C++ had used a large chunk of the time I had available.

The model for the drone that will be used in this project is the same model that was provided for the mini-project. If given a sufficient amount of time, I would then like to implement a more complex model of a drone.

Problem Statement

The central problem to be faced in this project is planning the motion of a drone following a snowboarder through backcountry slopes covered in trees. The overarching goal for this project is to develop an algorithm that will allow multiple drones to follow a snowboarder, while keeping the camera on the drone pointed at the snowboarder for most every state. In order to reach this goal, a few subproblems have been created as milestones to help get closer to the more ambitious goal.

The first subproblem is to develop a terrain that both the drone and snowboarder can traverse through, as well as create planners for both the drone and the snowboarder. Since this is purely simulation, the best way to create a path for the snowboarder is to create a motion plan for it as well. Once we have the motion planners for both the snowboarder and the drone, we can move onto our second subproblem.

The second subproblem for the project is to create a motion planner that remains within the region around the snowboarder AND has the camera pointed towards the snowboarder for a minimum of 60% of the states. The percentage of 60% is an arbitrary decision and goal, as increasing this value to higher percentages may mean the drone is unable to find a full path for the entire run. This subproblem allows us to make sure that the drone is getting reasonable footage of the snowboarding during his run.

The final and most ambitious subproblem is to add the ability for multiple drones to be flying within the region of the snowboarder and having their cameras pointed towards him also around 60% of the time. The challenge of this subproblem is forcing the drones to obtain footage for 60% of their states while not colliding with each other as well as the environment they are flying through, all while remaining within a reasonable distance of the snowboarder.

Methodology

Setting Up the Environment, Snowboarder, and Robot

Environment

The real first problem I had to overcome was creating a realistic environment for the drone and snowboarder to traverse through. The original plan was to search for and use LIDAR data of back-country ski slopes found online, but that proved to be more challenging than expected, as I could not find the proper data with both foliage and terrain that was required. Since the environments couldn't be found online, the next best option was to create a simulated environment using Unreal Engine (UE). Having to learn UE from scratch, I decided it would be best to start out our motion planner with a simple environment of a flat plane with dense foliage. In order to do this I created a flat landscape, and added Megascanes of various trees to simulate a realistic environment, then exported from UE into Blender to create STL and OBJ files of the environment, as shown below:

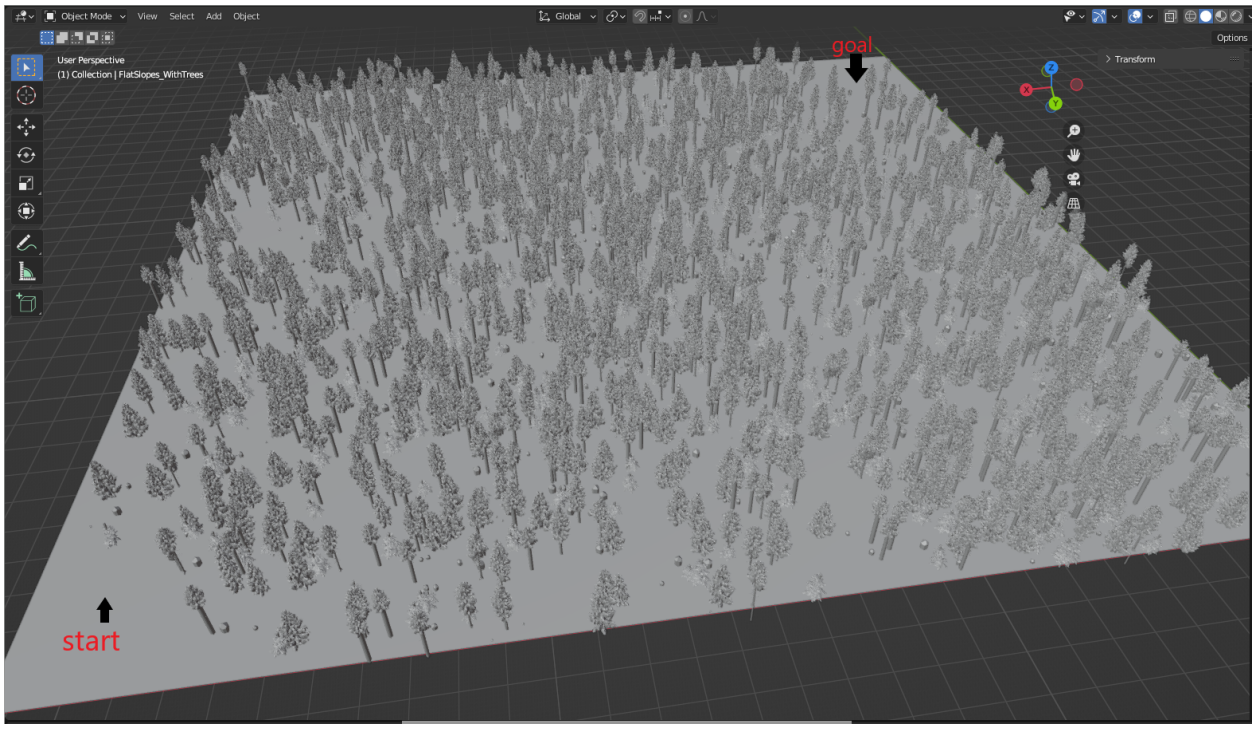

Figure 1: Mesh Environment from Unreal Engine in Blender, with Starting Point (red) and Goal Point (green)

An additional requirement to sample within the desired area was to create a bounding volume surrounding the entire mesh. Creating this bounding volume is quite simple and only requires a few steps in Blender. First, insert the ground plane of your desired environment. After insertion, duplicate your ground plane and translate it to your desired upper-boundary of Z. Once you have

the lower and upper bounds for Z, the last step is to insert planes and place them along the edges of the lower and upper Z-bounds. Once complete the mesh is exported from Blender and used in the motion planner. The bounding mesh is used to quickly sample points within the desired volume of your environment. While it may seem unnecessary, this step is crucial and will allow us to properly sample points within more complex volumes when we move on to non-planar environments.

Snowboarder

Since the snowboarder isn't the focus of the goal, I decided it was best to implement a simple non-dynamic motion planner for them. The original plan was to integrate Unreal Engine with the motion planner, so that a user could take control of the snowboarder and provide it with a random path for the motion planner. Having switched from C++ to Python, however, it would be much more difficult to implement this original plan. Instead of having a user making a random path for the snowboarder, the snowboarder will be modeled as a cylinder approximating the size of a person, and will have its own planner that won't use dynamics, but will then provide the drone with the time-step position information so it can follow it properly. The dimensions of the cylinder are 40 cm in diameter, and 170cm in height, which are the general shape and size of the average human being.

Robot

The robot that will be simulated in this motion planner is a personal sized drone (quadrotor). The model for this drone was pulled from GrabCad, and was designed by GrabCad member Logesh Kumar P. The model for the done is shown below:

Figure 2: Rendered Model of Quadrotor Drone

While I will be using the model of the drone to display its various states throughout the environment, I believe that it is best to generalize the shape of the drone as a sphere, using the maximum dimension of the model as the diameter of the sphere. Since we are dealing with an environment that can have various extrusions and obstacles, the drone model itself may end up getting close to an obstacle with a valid trajectory. Once the drone is in that state, however,

finding trajectories out of that state could prove to be difficult. For this reason, I believe it is best to simplify the drone's shape as a sphere to represent all possible orientations in a state. Simplifying the drone as a sphere also allows us to create primitive spheres at sampled positions, without having to struggle and worry about applying transformations to the drone model. The size of the sphere is shown below in comparison to the drone model:

Figure 3: Generalized Model of Drone to Account for all Orientations

Drone Dynamics, Equations of Motion, and Boundary Conditions

Now that we have our geometric boundaries, our first environment, and our generalized model of our drone, we can set up the dynamics and apply boundary conditions for the inputs. The dynamics for the model of our drone will be the same as the one shown in the mini-project, which can be scene below:

$$
\begin{aligned}
\dot{x} &= v\ \cos(\psi)\ \cos(\theta), \\
\dot{y} &= v\ \sin(\psi)\ \cos(\theta), \\
\dot{z} &= v \sin(\theta), \\
\dot{\psi} &= \omega \\
\dot{\theta} &= \alpha \\
\dot{v} &= a,
\end{aligned}
$$

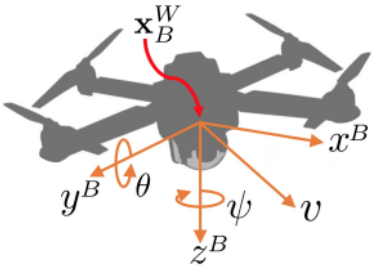

Figure 4: Simple Equations of Motion for Drone Model

Our state-space X, along with the matrix for the equations of motion, is given by:

$$X = \begin{bmatrix} x \\ y \\ z \\ \psi \\ \theta \\ v \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}$$

$$\dot{X} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\psi} \\ \dot{\theta} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v\cos(\psi)\cos(\theta) \\ v\sin(\psi)\cos(\theta) \\ v\sin(\theta) \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} x_6\cos(x_4)\cos(x_5) \\ x_6\sin(x_4)\cos(x_5) \\ x_6\sin(x_5) \\ \dot{x}_4 \\ \dot{x}_5 \\ \dot{x}_6 \end{bmatrix}$$

Where x,y,z are the 3D coordinates of the drone, ψ is the yaw of the drone, θ is the pitch of the drone, and v is the velocity of the drone. Taking the derivative of X provides us with the equations of motion (EOM) shown above

Given the equations of motion above, we can use the `odeint` function from the SciPy library to solve our trajectories, given an initial state x_0, our inputs u_1, u_2, u_3, and the propagation time t. Similar to the mini project from earlier in the semester, to ensure the stability of the drone as it moves throughout the the slopes, we will limit its velocity and pitch by applying upper and lower boundaries with $\theta \in \left[-\frac{\pi}{3}, \frac{\pi}{3}\right]$ and $v \in [-1, 1]$.

For the inputs u_1, u_2, u_3, where u_1 is the yaw angle rate, u_2 is the pitch angle rate, and u_3 is the linear acceleration rate, I will assign the upper and lower bounds given by:

$$u_1, u_2 \in [-\frac{\pi}{6}, \frac{\pi}{6}], \quad u_3 \in [-\frac{1}{2}, \frac{1}{2}].$$

The values u_1, u_2, u_3, are very much subject to decrease (will not increase as that could make the system unstable) if needed to properly navigate through the dense forestry.

Implementation of SST

In order to properly implement the SST motion planning algorithm, the first thing I did was find research articles that discussed its benefits and implementations. The most helpful article I found was titled *Asymptotically Optimal Sampling-based Kinodynamic Planning,* written by Li, Littlefield, and Bekris. In this research paper they discuss multiple types of kinodynamic sampling based planners such as RRT, RRT with the Best Nearest, RRT with Drain, and Sparse Stable RRT otherwise known as SST, along with SST star. As they mentioned that it is hard to represent C-Space obstacles for higher dimensional sampling based planners, I also found it was best to check the robot at vertices for collisions in the state-space rather than use a C-Space. (Li et al. 2010).

In this research paper they clearly outline the process of using and implementing the SST algorithm as shown an algorithm 5, pool from the research paper shown below:

Algorithm 5: STABLE_SPARSE_RRT($\mathbb{X}$, $\mathbb{U}$, x_0, T_{prop}, N, δ_{BN}, δ_s)

$\mathbb{V}_{active} \leftarrow \{x_0\}, \mathbb{V}_{inactive} \leftarrow \emptyset$;
$G = \{V \leftarrow (\mathbb{V}_{active} \cup \mathbb{V}_{inactive}), \mathbb{E} \leftarrow \emptyset\}$;
$s_0 \leftarrow x_0$, $s_0.rep = x_0$, $S \leftarrow \{s_0\}$;
for N *iterations* **do**
  $x_{selected} \leftarrow$Best_First_Selection_SST($\mathbb{X}$, $\mathbb{V}_{active}$, δ_{BN});
  $x_{new} \leftarrow$ MonteCarlo-Prop($x_{selected}$, $\mathbb{U}$, T_{prop});
  if CollisionFree($\overline{x_{selected} \rightarrow x_{new}}$) **then**
    if Is_Node_Locally_the_Best_SST(x_{new}, S, δ_s) **then**
      $\mathbb{V}_{active} \leftarrow \mathbb{V}_{active} \cup \{x_{new}\}$;
      $\mathbb{E} \leftarrow \mathbb{E} \cup \{\overline{x_{selected} \rightarrow x_{new}}\}$;
      Prune_Dominated_Nodes_SST(x_{new}, $\mathbb{V}_{active}$, $\mathbb{V}_{inactive}$, $\mathbb{E}$);
return G;

Figure 5: SST Algorithm from *Asymptotically Optimal Sampling-based Kinodynamic Planning* (Li et al. 2010)

The SST algorithm uses 7 parameters for sampling: the state-space X, the inputs U, the start-state x_0, the maximum propagation time T_{prop}, the maximum number of iterations N, the limiting radius δ_{BN} for states to connect to neighboring states, and the witness region radius δ_s. SST begins by creating two sets of vertices V_{active} and $V_{inactive}$, where x_0 is inserted into V_{active} and $V_{inactive}$ is initialized to have 0 vertices. We then create the (vertice,edge) set G, which is initialized with V as the union of V_{active} and $V_{inactive}$, and initializing E as an array of size 0. The SST algorithm then moves onto creating its first witness region s_0, which is centered around the state x_0, and added to the set of witness regions S. Once the sets of vertices, edges, and witness regions are initialized, the planner moves onto collecting and connecting sample states (Li et al. 2010).

The SST algorithm begins with the Best_First_Selection_SST function outlined below in Algorithm 6:

Algorithm 6: Best_First_Selection_SST($\mathbb{X}$, $\mathbb{V}$, δ_{BN})

$x_{rand} \leftarrow$ Sample_State($\mathbb{X}$);
$X_{near} \leftarrow$Near($\mathbb{V}, x_{rand}, \delta_{BN}$);
If $X_{near} = \emptyset$ **return** Nearest($\mathbb{V}, x_{rand}$);
Else return $\arg\min_{x \in X_{near}} cost(x)$;

Figure 6: Best_First_Selection algorithm from *Asymptotically Optimal Sampling-based Kinodynamic Planning* (Li et al. 2010)

The purpose of the Best_First_Selection algorithm is to choose a vertice in V_{active} to choose to apply a trajectory to, in order to continuously grow the tree. Bset_First_Selection begins by

taking a random sample of the State-Space X and assigning it to x_{rand}. We then find the nearest neighbors within the radius δ_{BN} of state x_{rand}, and return it as an array sorted by cost called X_{near}. If X_{near} is empty, then rather than attempting to use the state with the lowest cost, it will instead use the closest state to x_{rand}. The vertice that Best_First_Selection returns is the $x_{selected}$ point to be used in the MonteCarlo-Prop algorithm (Figure 7.), which generates the random inputs for the drone and the random amount of time in the range $[0,T_{prop}]$ that the drone performs those inputs. The reason for using the MonteCarlo-Propagation algorithm is because when using kinodynamic planners, even if we sample a random point in 3D space, we would then have to solve the two-point boundary value problem in order to connect the starting state to the ending state. In order to avoid the two-point BVP, instead of sampling a random point, we chose a state to extend from, and apply random inputs to create a new trajectory and state. (Li et al. 2010). The formulated Python function for Best_First_Selection is shown in Appendix A.a.

Algorithm 3: `MonteCarlo-Prop`$(x_{prop}, \mathbb{U}, T_{prop})$

$t \leftarrow$ Sample$(0, T_{prop})$; $\Upsilon \leftarrow$ Sample$(\mathbb{U}, t)$;

return $x_{new} \leftarrow \int_0^t f(x(t), \Upsilon(t))\, dt + x_{prop}$;

Figure 7: MonteCarlo-Propagation Algorithm (Li et al. 2010)

Once the state on the tree $x_{selected}$ is found, it is then used in the MonteCarlo-Prop algorithm shown above. This algorithm first samples a propagation time from the range of $[0,T_{prop}]$, where T_{prop} is a parameter that is chosen by the individual using the planner. Given a range of boundaries for our inputs U, the algorithm also samples within those ranges (such as $u_1 \in \left[-\frac{\pi}{6}, \frac{\pi}{6}\right]$). The random input and propagation time values are then integrated with the state-space, with $x_{selected}$ as our initial conditions to provide the trajectory and final state x_{new}. The Python function for the MonteCarlo-Prop algorithm can be found in Appendix A.b and A.c respectively.

After we obtain the new trajectory for the drone as well as x_{new}, we need to apply state validity checking and collision checking to both the trajectory and x_{new}. Before I perform collision checking, I perform the state validity checking, because even though we are sampling inside the boundary region of the environment, there's still the chance that the propagation time and inputs supplied to $x_{selected}$ created a trajectory and state that lead the drone outside of the boundary region. IF the drone is still inside the boundary region I then check against our dynamical constraints, both the bounds for the pitch "THETA" and velocity "V". If all of the dynamical constraints are maintained during the drones new trajectory and new state then we move on to the collision checking. For the collision checking of our trajectory and x_{new}, I iterate through the set of trajectories, and create a sphere at each XYZ position. Once the sphere is created, it is added to the collision manager that already contains the environment, where the collision manager then checks for an internal collision between the mesh environment and the

mesh sphere that represents the drone. The Python code for checking the trajectory and x_{new} against collisions and state-constraints are given in Appendix A.d and A.e respectively.

After both the collision checker and the state validity checker confirm that both x_{new} and its trajectory are valid, the SST algorithm then moves onto checking if x_{new} is in any previously made witness regions. This is checked via the algorithm Is_Node_Locally_the_Best_SST, shown below:

Algorithm 7: Is_Node_Locally_the_Best_SST(x_{new}, S, δ_s)

$s_{new} \leftarrow$ Nearest(S,x_{new});
if $||x_{new} - s_{new}|| > \delta_s$ **then**
  $S \leftarrow S \cup \{x_{new}\}$;
  $s_{new} \leftarrow x_{new}$;
  $s_{new}.rep \leftarrow NULL$;
$x_{peer} \leftarrow s_{new}.rep$;
if $x_{peer} == NULL$ *or* cost(x_{new}) $<$ cost(x_{peer}) **then**
  return true;
return false;

Figure 8: Is_Node_Locally_the_Best_SST (Li et al. 2010)

If x_{new} is in a previously created witness region, the above algorithm will compare the cost of x_{new} versus the cost of x_{peer}, the representative of the witness region. If the cost of x_{new} is less then x_{peer}, or if x_{new} is not in previously made witness region at all, then the algorithm returns true, and x_{new} will be made the representative of the witness region. If the algorithm returns false, x_{new} and its found trajectory are dropped from the propagation. The python implementation of Is_Node_Locally_the_Best_SST can be found in Appendix A.f.

The final step in the SST algorithm, and the one that causes it to be so optimal, is the pruning algorithm shown below:

Algorithm 8: Prune_Dominated_Nodes_SST(x_{new}, $\mathbb{V}_{active}$, $\mathbb{V}_{inactive}$, $\mathbb{E}$)

$s_{new} \leftarrow$ Nearest(S,x_{new});
$x_{peer} \leftarrow s_{new}.rep$;
if $x_{peer}! = NULL$ **then**
  $\mathbb{V}_{active} \leftarrow \mathbb{V}_{active} \setminus \{x_{peer}\}$;
  $\mathbb{V}_{inactive} \leftarrow \mathbb{V}_{inactive} \cup \{x_{peer}\}$;
$s_{new}.rep \leftarrow x_{new}$;
while $x_{peer}! = NULL$ **and** IsLeaf (x_{peer}) **and** $x_{peer} \in \mathbb{V}_{inactive}$ **do**
  $x_{parent} \leftarrow$Parent(x_{peer});
  $\mathbb{E} \leftarrow \mathbb{E} \setminus \{\overline{x_{parent} \rightarrow x_{peer}}\}$;
  $\mathbb{V}_{inactive} \leftarrow \mathbb{V}_{inactive} \setminus \{x_{peer}\}$;
  $x_{peer} \leftarrow x_{parent}$;

Figure 9: Prune_Dominated_Nodes_SST (Li et al. 2010)

The algorithm above is one of the most crucial steps of the SST algorithm, because it's what keeps the tree *sparse*. If the algorithm Is_Node_Locally_the_Best_SST returns true, that means x_{new} is the new representative of the witness region, but the old representative x_{peer} is still in the active state set. The Prune_Dominated_Nodes_SST algorithm fixes this issue by checking if x_{peer} is still state in its witness region with the lowest cost. If x_{peer} does not have a lower cost than x_{new}, x_{peer} is then removed from V_{active} and added to $V_{inactive}$, and x_{new} is made the new representative of the witness region and added to the set of states V_{active}. Afterwards, the algorithm then loops through and checks to see if x_{peer} is a leaf node, and if it is, it is removed from the set of states $V_{inactive}$ and its trajectory from its parent x_{parent} to x_{peer} is removed from the set of trajectories E. The state x_{parent} is then turned into x_{peer} and the loop begins again to check if that state is a leaf node as well. The python implementation of Prune_Dominated_Nodes_SST is shown in Appendix A.g. After the algorithm has completed, the state that combines the inactive and active sets $V = V_{active} \cup V_{inactive}$ is updated, which is then used to update G, the set of states V and trajectories E.

Putting each of these algorithms together gives us the final SST algorithm! The python implementation for the SST algorithm can be found in Appendix A.h.

Results

Unfortunately I was not able to reach my most ambitious goal and I also wasn't able to reach my second sub problem. After I finished developing the SST algorithm in Python I implemented it with the small mesh forest that is covered in a dense amount of trees that the drone has to navigate through. Before I implemented the dynamics for the drone in the system, I used a non-dynamic model that was able to complete and reach the goal state relatively fast. For my PC this took roughly thirty second to two minutes. However when I implemented the dynamics for the drone and started calculating trajectories, the time it took to complete the planning algorithm increased dramatically, from two minutes to at least 10 minutes for the same parameters. Compared to the non-dynamical model that just used sampled states, the dynamic model was unable to create a reasonably sized path to the goal, it was much more difficult than I would have expected.

The image below shows the output of the SST algorithm using the dynamics of the drone. The value for $\delta_s = 0.8$, the value for $\delta_{BN} = 1$, the value for $T_{prop} = 3$, and the value for the number of iterations N=2000. As can be seen in the image the drone is barely able to leave the starting area. I was extremely surprised that it was barely able to leave the starting area, and knew that I would have to start playing around with the parameters.

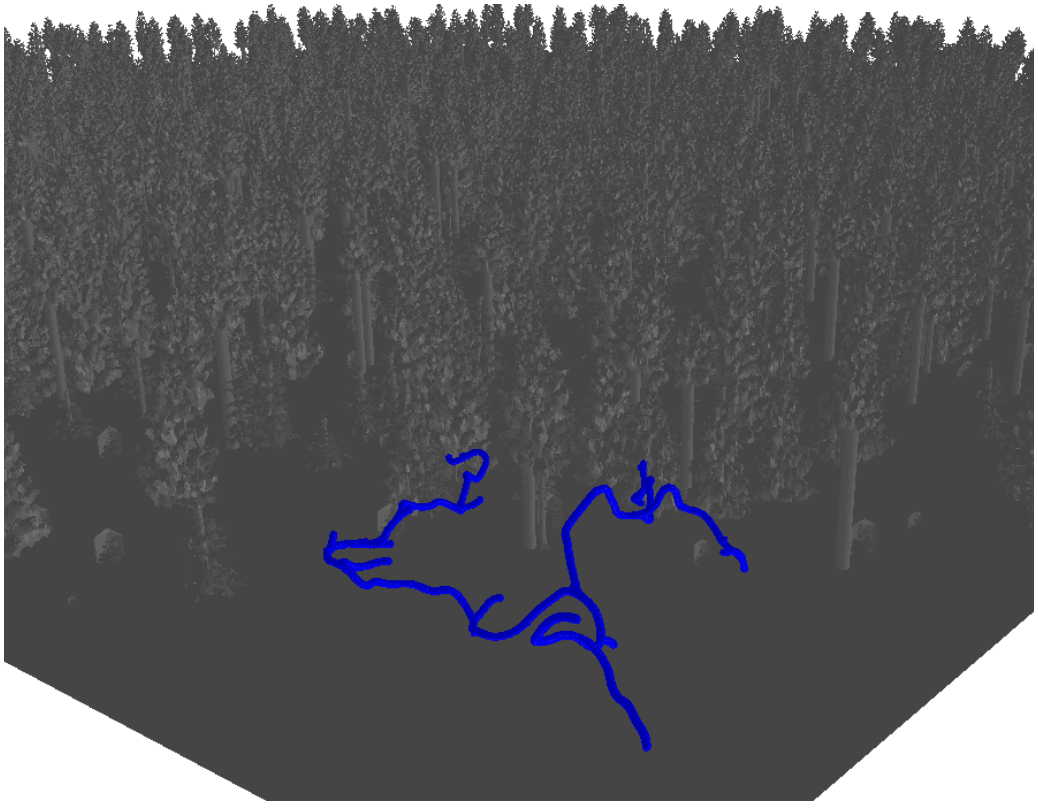

Figure 10: SST paths with δ_s =0.8, δ_{BN} =1, T_{prop} =3, N=2000

The first thing I did was nearly double all of the parameters, just to see if it would have any impact on the motion planner's output. As can be seen below, the drone was finally able to leave the starting area, but did not make it far, and especially nowhere near the goal.

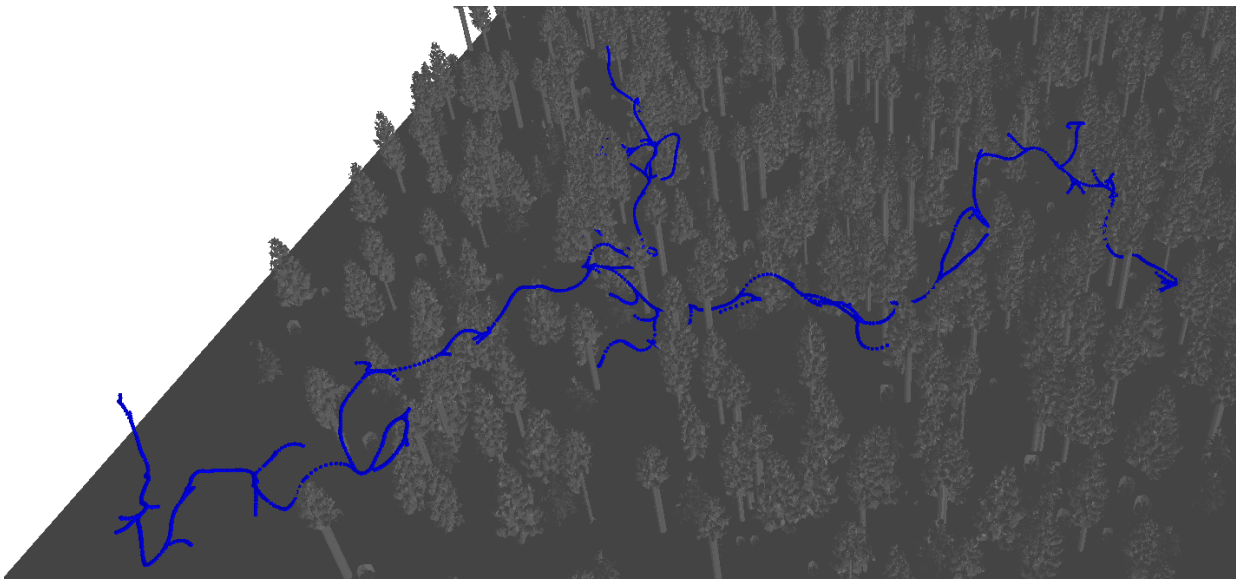

Figure 11: SST paths with δ_s =1.5, δ_{BN} =2, T_{prop} =8, N=5000

Of course, there is one solution to this issue, and it would be to raise the z-boundary so the drone could fly over the trees. I chose not to do that because it wasn't what I had intended when starting this project. I wanted to make a motion planner that will allow the drone to seamlessly weave between trees while recording a snowboarder, so allowing it to fly over the trees would have defeated the purpose.

Instead of increasing the z-boundary, I wanted to determine the parameters necessary for the drone to make it all the way from the starting point to the goal. I found out early on that making δ_s and δ_{BN} similar to each other (while keeping $\delta_s < \delta_{BN}$) made it extremely difficult for new states to be created. I also found that increasing T_{prop} too much will also negatively impact the performance of the planner, unless I were to alter the boundaries for the input values. For my next attempt I decided to decrease the yaw/pitch angle rate boundaries from $\pm \frac{\pi}{6}$ to $\pm \frac{\pi}{12}$, and decrease the acceleration rate from ± 0.5 to ± 0.35. Doing so provided me with a more successful tree shown in Figures 12.

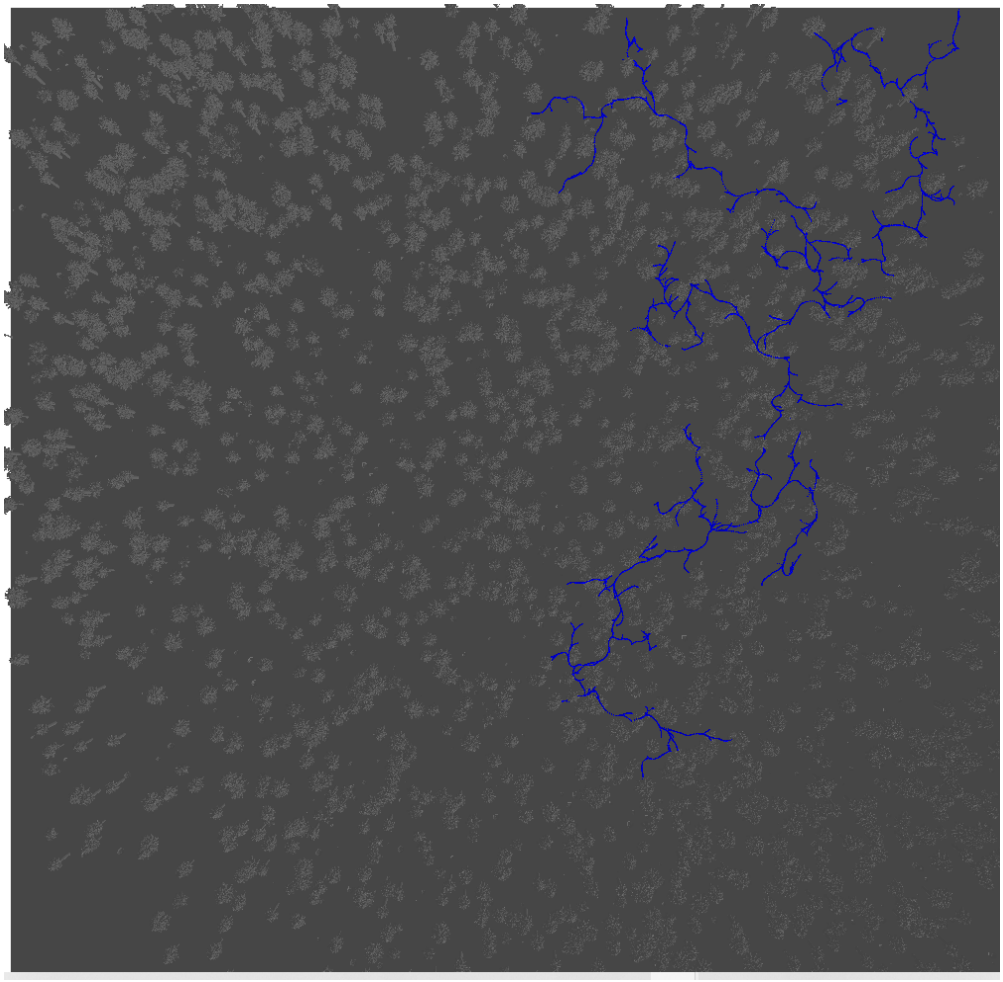

Figure 12: SST paths with δ_s =1.5, δ_{BN} =2, T_{prop} =8, N=5000, $\mathbf{u_1}, \mathbf{u_2} \in \pm \frac{\pi}{6}$, $\mathbf{u_3} \in \pm\ 0.35$

Surprisingly these new values provided a computation time that was less than the prior examples, closer to around two minutes. The last parameter I could really tune was N, the number of iterations, so I decided to let N=50,000 as well as increase the goal-bias to 25% from the 5% it was set to originally, and let it run in hopes that it would provide me with a route to the goal. The output mesh of this tree was so large (~3GB) that I wasn't even able to render it in python. I had to export each of the trajectory spheres as an STL, and import them all into a blender with the environment. The output of the planner can be seen below in Figure 13:

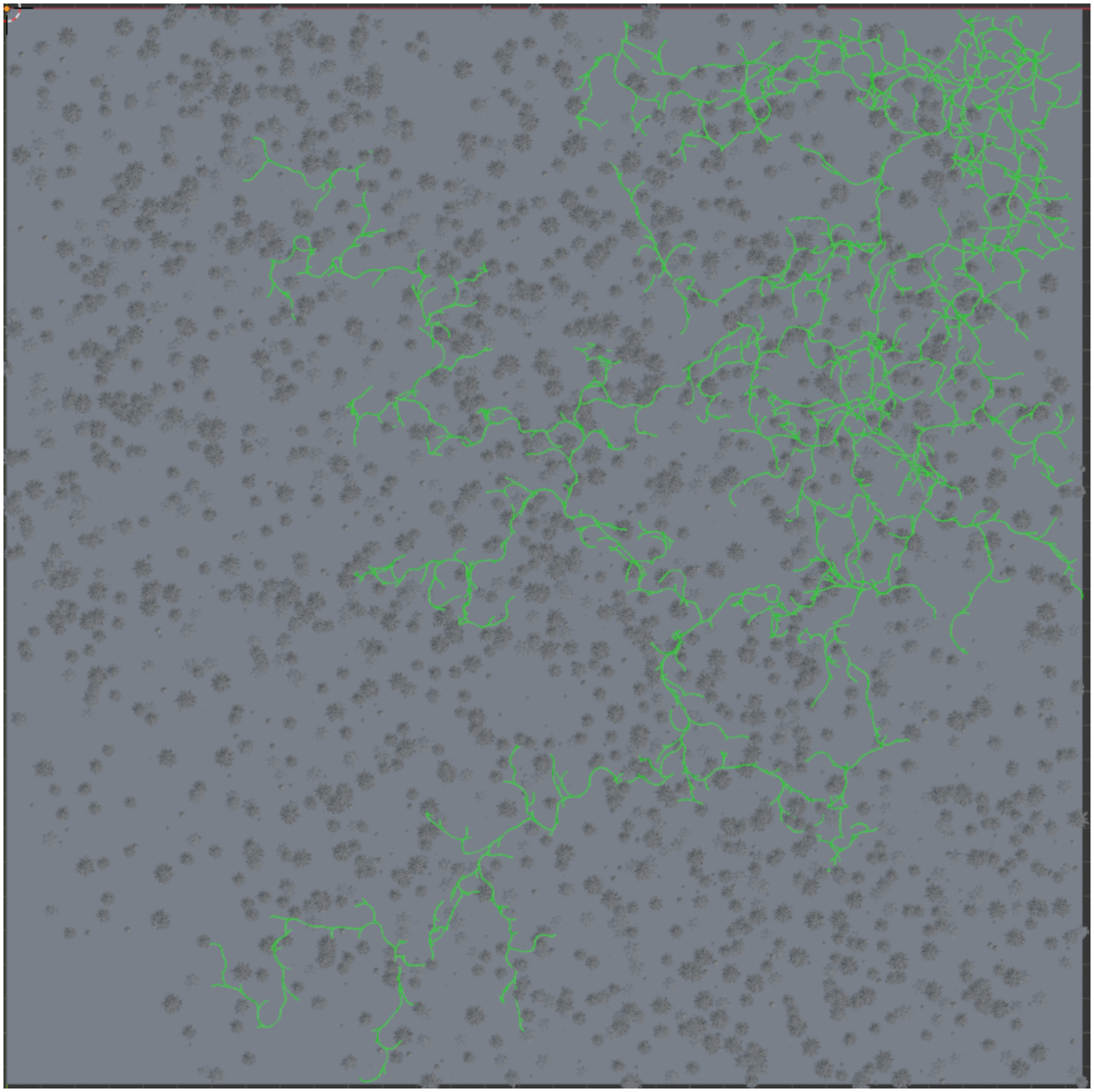

Figure 13: **SST paths with δ_s =1.5, δ_{BN} =2, T_{prop} =8, N=50,000, $u_1, u_2 \in \pm \frac{\pi}{6}$, $u_3 \in \pm\ 0.35$, $prob_{goal}$ =25%**

As seen in figure 13 above, even this planner was unable to reach the goal state. This version of the planner with these parameters took just over an hour to run on my PC, much slower than the two minutes it took the previous version. This was my final attempt of this project creating a tree for the drone from the start state to the goal state. That being said, I plan to dive deeper into this and see if I can figure out a way to allow the tree to be grown faster and quicker while still implementing the dynamics of the drone.

Discussion

Seeing the results that I just showed above I am starting to believe that it might have bit off more than I could chew for this project with the time available. I believe that planning for a dynamical system such as a drone in such a densely obstructed environment is a difficult problem in itself. One thing that I could end up doing is creating a mesh with a less dense population of foliage. Once I create a mesh that the drone is able to traverse through and find trajectories for in a shorter amount of time, I could then begin to scale up the density of the forest and see where the breaking point is. I could also implement a more heavily control-based method so that the use of random inputs and random propagation times wouldn't be necessary, even though it might lead me to having to solve two-point BVPs. I'm also curious if a more complex dynamical drone model would have a positive effect on the growth of the tree, because perhaps the model that I was implementing wasn't accurate enough to properly represent the dynamics of a drone. Based on the image of my final attempt, I can clearly see that foliage obstacles aren't making the planner incomplete and that with more iterations the drone would have been able to reach the goal state. If I had more time I believe that I could develop a more complex algorithm to deal with this densely populated forest. Once the algorithm was complete I could then implement the more complex problems of staying pointed towards the snowboarder and possibly even introduce multiple drones into the scenario.

Conclusion

While I wasn't able to complete the scenario of the Drone following a snowboarder, I do believe that I've learned a lot from this project oh, and I plan to explore the issues that I faced even after the course is over. I may have struggled implementing C++ and OMPL, but I do plan on learning them, because I'm curious if the Sparse-Stable RRT planner that's provided has the same issue. I myself am happy that I was able to implement the SST motion planner from scratch, and I'm extremely happy I was able to learn how to calculate trajectories for the drone. To continue the work that I've already started, I am going to begin to build out a Python Library for motion planners. The original reason I had chosen to attempt to use OMPL was because I was told it would be simpler than starting from scratch in Python, but I overestimated my ability to learn C++ in such a short amount of time. Having faced this issue I believe it would be useful to develop a more user-friendly version of OMPL. Before I do that however, my intention is to finish this project and develop a motion planning algorithm that can handle the backcountry slopes environment with ease.

References

[1] Li, Y., Littlefield, Z., & Bekris, K.E. (2014). Asymptotically optimal sampling-based kinodynamic planning. The International Journal of Robotics Research, 35, 528 - 564.

[2] “SciPy Tutorial (2022): For Physicists, Engineers, and Mathematicians.” *YouTube*, YouTube, 1 June 2021, https://www.youtube.com/watch?v=jmX4FOUEfgU. Accessed 12 Dec. 2022.

Appendix

A. Code

a. **Best_First_Selection()**:

```
def Best_First_Selection_SST(X,V,del_BN,sampler):
    useGoal = random.uniform(0,1)
    # occasionally choose state closes to goal in order to attempt to reach goal
    if(useGoal > 0.95):
        x_rand = goal_state
    else:
        x_rand = sampler.StateSample(sampler.boundingMesh,pitch_bounds=pitch_bounds,velocity_bounds=velocity_bounds)
    X_near_within_delBN = [[V[i].cost,V[i]] for i in range(0,len(V)) if Get3Ddist(V[i].state[0:3],x_rand[0:3]) < del_BN]
    X_near_distances = [[V[i],Get3Ddist(V[i].state[0:3],x_rand[0:3])] for i in range(0,len(V))]
    X_near_distances = sorted(X_near_distances, key = lambda x: x[1])

    closest = X_near_distances[0][0]

    if(len(X_near_within_delBN) == 0):
        #no neighbors within delta_BestNeighbor, use closest neighbor and extend branch by delta_BN
        x_selected = closest
        mode = "Branch Extension"
        return x_selected, mode
    else:
        X_near_within_delBN.sort(key=lambda x: x[0])
        x_selected = X_near_within_delBN[0][1]
        mode = "Best Neighbor"
        return x_selected, mode
```

b. **MonteCarlo-Prop():**

```
def MonteCarlo_Prop(x_selected,U,T_prop):
    #x_selected is chosen neighbor, U is input bounds dict, T is max propagation time
    numInputs = len(U)
    random_inputs = np.empty([1,numInputs])
    i = 0
    for input_bounds in U:
        # inputBounds = input.value #get bounds of input from dictionary
        # inputSample = random.uniform(inputBounds[0],inputBounds[1])
        random_inputs[0][i] = random.uniform(input_bounds[0],input_bounds[1])
        i+= 1
    t = random.uniform(0,T_prop)
    return random_inputs, t
```

c. CalculateTrajectory():

```
def CalculateTrajectory(state,inputs,t):
    pi = math.pi
    cos = math.cos
    sin = math.sin
    x_0 = (state[0], state[1], state[2], state[3], state[4], state[5])
    #x_0 is the state at the x_selected
    omega = inputs[0]
    alpha = inputs[1]
    acc = inputs[2]
    #define state-space model for odeint to solve
    def dXdt(X,t):
        x,y,z,yaw,theta,v = X
        return [v*cos(yaw)*cos(theta),
                v*sin(yaw)*cos(theta),
                v*sin(theta),
                omega,
                alpha,
                Acc]

    t = np.linspace(0,t,10)
    sol = odeint(dXdt,x_0,t)
    x,y,z,yaw,theta,v = sol.T
    return x,y,z,yaw,theta,v
```

d. IsTrajectoryInCollision():

```
def IsTrajectoryInCollision(collision_manager_with_env,botRadius, xyzPoints):
    isCollision = False
    for [x,y,z] in xyzPoints:
        sphere = trimesh.creation.icosphere(radius=botRadius)
        sphere = sphere.apply_translation([x,y,z])

        collision_manager_with_env.add_object("Drone",sphere)
        isCollision = collision_manager_with_env.in_collision_internal()
        collision_manager_with_env.remove_object("Drone")
        if(isCollision == True):
            break
    return isCollision
```

e. StateValidityChecker():

```
def StateValidityChecker(boundingMesh,pitch_bounds,velocity_bounds,trajectoryStates):
    xyzPoints = []
    allPointsInBounds = True
    x = trajectoryStates[0]
    y = trajectoryStates[1]
    z = trajectoryStates[2]
    yaw = trajectoryStates[3]
    theta = trajectoryStates[4]
    v = trajectoryStates[5]
    for i in range(0,len(x)):
        if(not boundingMesh.contains([[x[i],y[i],z[i]]])):
            allPointsInBounds = False
            break
        if(theta[i] < pitch_bounds[0] or theta[i] > pitch_bounds[1]):
            allPointsInBounds = False
            break
        if(v[i] < velocity_bounds[0] or v[i] > velocity_bounds[1]):
            allPointsInBounds = False
            break

        xyzPoints.append([x[i],y[i],z[i]])
    if(allPointsInBounds == False):
        return False
    else:
        return True
```

f. Is_Node_Locally_the_Best_SST():

```
def Is_Node_Locally_the_Best_SST(xnew, S = PlanningAlgorithms.SST.WitnessRegions.__init__, delta_s=1):
    rgnIdx = 0
    regionID = None
    inOldRegion = False
    for witness_region in S.Regions: #check if new or old cost is better and make that one region rep
        regionID = witness_region.regionID
        if(witness_region.regionMesh.contains(([xnew.state[0:3]]))):
            if(witness_region.cost > xnew.cost):
                regionID = witness_region.regionID
                inOldRegion = True
                return True, regionID, inOldRegion
            else:
                return False, regionID, inOldRegion
        rgnIdx += 1


    #if it reaches this point, it was not in an old region, new witness region will be created
    regionID += 1
    S.AddNewRegion(xnew,delta_s,xnew.cost,regionID)
    rgnIdx += 1
    inOldRegion = False
    return True, regionID, inOldRegion
```

g. Prune_Dominated_Nodes_SST():

```
def Prune_Dominated_Nodes_SST(xnew, V_active, V_inactive, E, rgnIdx, S,isOldRegion):
    s_new = None
    idx = 0
    for i in range(0,len(S.Regions)):
        if(S.Regions[i].regionID == rgnID):
            s_new = S.Regions[i]
            break
        else:
            idx+=1

    x_peer = s_new.representative
    if(isOldRegion and x_peer is not None): #move x_peer from active to inactive
        V_active.remove(x_peer)
        V_inactive.append(x_peer)
    S.Regions[idx].representative = xnew
    while(x_peer is not None and len(x_peer.Children)==0 and x_peer in V_inactive):
        print(x_peer)
        x_parent = x_peer.Parent
        print(x_parent)
        if([x_parent,x_peer] in E):
            E.remove([x_parent,x_peer])
        V_inactive.remove(x_peer)
        x_peer = x_parent
```

h. STABLE_SPARSE_RRT():

```
E = [] #trajectory list
V_active = [] #active state list
initialState = Vertice(x_0,0)
V_active.append(initialState)
V_inactive = [] #inactive state list
V = V_active+V_inactive #active and inactive states
G = [V,E]
S = PlanningAlgorithms.SST.WitnessRegions(initialState,delta_s,0)
print(V[0].state[0:3])

ss_sampler = StateSampler(boundingMesh)
x_selected,  mode = Best_First_Selection_SST(STATE_SPACE,V_active,delta_bn,ss_sampler)

scene = trimesh.Scene()
scene.add_geometry(envMesh)
for iteration in range(0,N):
    x_selected, mode = Best_First_Selection_SST(STATE_SPACE,V_active,delta_bn,ss_sampler)

    random_inputs, t = MonteCarlo_Prop(x_selected,U,T_prop)
    random_inputs = random_inputs[0]

    x,y,z,phi,theta,v = CalculateTrajectory(x_selected.state,random_inputs,t)
    x,y,z,phi,theta,v = CalculateTrajectory(x_selected.state,random_inputs,t)
    trajectoryValues = [x,y,z,phi,theta,v]

    validState,xyzPoints = StateValidityChecker(boundingMesh,pitch_bounds,velocity_bounds,trajectoryValues)
    if(validState == False):
        continue

    new_V_state = [x[-1],y[-1],z[-1],phi[-1],theta[-1],v[-1]]
```

```
    new_V_cost = CalculateCost(xyzPoints) + x_selected.cost
    x_new = Vertice(new_V_state,new_V_cost)
    x_new.AddParent(x_selected)
    parent_idx = V.index(x_selected)

    droneTrajCollision = IsTrajectoryInCollision(ENV_COLIS_MNGR,droneMaxDim/2,xyzPoints)

    #IF NO COLLISION OCCURS, WE CAN MOVE ONTO CHECKING IF VERTICE IS BEST IN WITNESS REGION
    if(droneTrajCollision == False):
        isNodeBest, rgnID, inOldRegion = Is_Node_Locally_the_Best_SST(x_new,S=S,delta_s=delta_s)

        if(isNodeBest):

            V_active[int(parent_idx)].Children.append(x_new)

            V_active.append(x_new)

            for i in range(0,len(xyzPoints)):
                sphere = trimesh.creation.icosphere(radius=droneMaxDim/2,color=(0,0,255))
                sphere.apply_translation(xyzPoints[i])
                scene.add_geometry(sphere)

            E.append(trajectory)
            Prune_Dominated_Nodes_SST(x_new,V_active, V_inactive, E, rgnID, S,inOldRegion)
            V = V_active+V_inactive
            G = (V,E)

            if(In_Goal_Region(x_new.state)):
                print("GOAL REACHED")
                break
```