SWARM UAV
MOTION PLANNING PROJECT

Project Documentation

Team Members
Şeyda Subaşı
Burak Hocaoğlu
Tahsincan Köse

September 6, 2017

Contents

Figures

1 About The Project and Document

Swarm UAV project aims for a Robot Operating System (ROS) application that manages the autonomous motion planning of multiple Unmanned Aerial Vehicle(UAV)s in obstacle-rich environments. Throghout the development, project uses Hector Quadrotor Package for simulation.

The project group consists of three intern students and one assistant graduate student within KOVAN Research Laboratory. As interns, our work is mainly focused on ROS and Gazebo. All simulation environments are created and tested in Gazebo.

This document walks through the progress of the project task by task. While explanining tasks in further detail, supplementary materials, comprehensive pseudocodes and algorithm descriptions are used. In the meantime; challenges that were faced, the solutions that were embraced are covered under the relative tasks. At last, possible improvements which might be introduced to further enhance the project are offered.

At the beginning of corresponding subsections in **Background Information** section, suitable links to the detailed documentations are provided for readers who don't have familiarity with the libraries, packages and other technologies.

2 Background Information

This section covers the preliminary materials which are used in the project. It is highly recommended for readers who have not adequate information about them. Other readers may skip this section.

2.1 ROS-Kinetic

ROS-Kinetic is the latest release of ROS. It works on Ubuntu 16.04 (Xenial) distributions. Since this is the only release we developed the project on, this document is only applicable to ROS-Kinetic. Other releases might have compatibility issues with this project. Hence it is not recommended to migrate to another release, unless one possess an advanced and comprehensive ROS knowledge.

2.2 Catkin Workspace

Catkin workspace and catkin packages are the constituting units of this project and any other ROS application. For tutorials, please refer to this link.

2.3 ROS Commands

ROS Commands are required in order to operate on the project. One can do many useful things like inspection, visualization and recording while execution time. Below, the mostly used commands are listed:

- roslaunch
- rosrun
- rosnode
- rostopic

A full-fledged list of commands reside here. One can also refer to the cheat sheet, which contains a summarized version.

2.4 XACRO File

XML Macro, abbreviated as XACRO, is an XML macro language. It can create shorter and more readable XML files by using macros that expand to longer XML expressions. For further information, please refer to the link.

A practical example of this expansion process is illustrated on **Usage** section.

2.5 URDF File

Universal Robot Description Files (URDF) are used for describing the robot model to use in simulations, contains joints, links, sensors and physical features of the robots and written XMl. Since quadrotores come as they are, additional robot designing is not necessary. However, for custom obstacle object designation it might be useful. Further information can be obtained from ROS URDF

2.6 The Open Motion Planning Library (OMPL)

OMPL consists of many state-of-the-art sampling based motion planning algorithms. In our project, we have extensively used this library. It does not implement any collision checking or visualization, therefore it is easily integrable to the various projects. In the following sections, links are provided to specific pages on the website. As a start point, one can refer to this link. There are many materials on the website spanning from tutorials to demos. The documentation is, also, quite detailed for further inspection.

2.7 Eigen

Eigen is a C++ template library for linear algebra. It provides dynamic and static class types for various matrix types. Since we are dealing with multiple quadrotors, in order to process the relations between them, we had to come up with a solution that includes matrixes and vectors. Eigen also exploits Vectorization, which eases the computation complexity in huge scales.

The installation of Eigen Library is explained in **Usage** section. For users, who don't want to bother with step-by-step installation, we will provide a script file that automates this process.

3 Usage

Assuming ROS-Kinetic is built, this section will step through installation and running phases of the project.

1 Go to Hector Quadrotor Repository and get a clone of it under ~/`catkin_ws/src`
2 cd to ~/`catkin_ws` and run `catkin_make`
3 Go to our project repo and get a clone of it under ~/`catkin_ws/src`
4 Repeat **2**.

If no problem is encountered up until now, the project workspace is successfully built except Eigen Library. Below is the step-by-step installation of Eigen Library:

Current directory should be ~/`catkin_ws/src`
5 Open the terminal and type the below commands sequentially:

```
hg clone https://bitbucket.org/eigen/eigen/
cd eigen
mkdir build
cd build
cmake ..
```

```
make
sudo make install
```

After Eigen is succesfully installed, all necessary installation is done. Now, in order to run the simulation two more commands are necessary:

Open two terminals. In each of them go to `~/catkin_ws` *and run* `source devel/setup.bash`
This step is required to prepare the layer.

6 In the first terminal run `roslaunch hector_manipulator simulate.launch`

7 In the second terminal run `roslaunch hector_manipulator quad_manipulator.launch`

4 Project Layout

In this section, brief explanations of several layouts and items within them are given. One should be able to thoroughly understand all aspects of the project after reading this section.

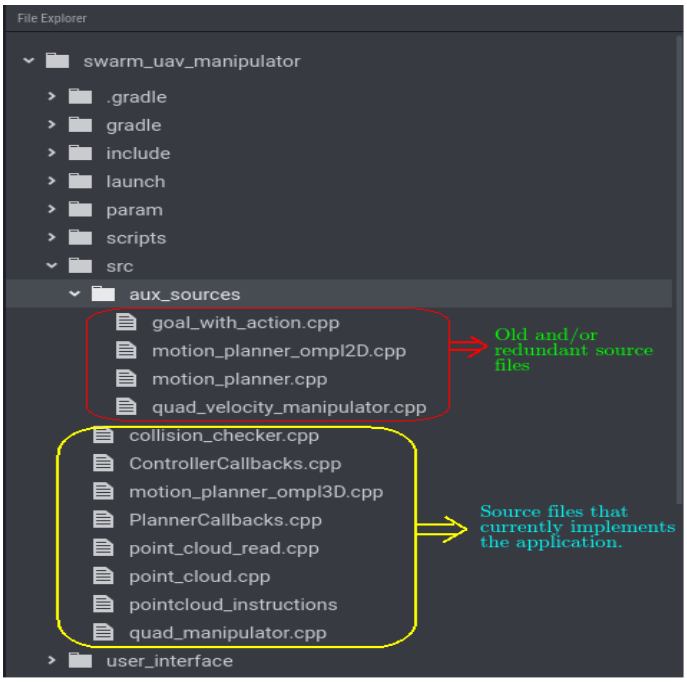

Figure 1: C++ Sources Layout

4.1 aux_sources

This directory includes source files used throughout the development. They are no longer used, but might be useful in the future improvements.

- `goal_with_action.cpp:` This source file used in the early phase of development. The demos in the `Hector_quadrotor_gazebo` package use Action Library Stack to send and satisfy goals. This source file simply manipulates this stack.

- `motion_planner_ompl2D.cpp:` 2D version of our current implementation. For algorithm optimizations and different collision avoidance strategies, this can be more useful compared to 3D, because dimensional complexity reduces and so do Degree of Freedom(DOF)s.

- `motion_planner.cpp:` This source file includes our own RRT implementation aside with motion planning. It can be useful if a hybrid algorithm is needed to be created.

- `quad_velocity_manipulator.cpp:` This source file created to mimic the behavior of the controllers in `hector_quadrotor_controller` package. However, it has no effect when integrated to the application. Therefore, it is not being used as a part of the project.

4.2 collision_checker.cpp

Collision check logic is implemented in here. Implements the priority policy for UAVs. The node name is `collision_checker`.

Publishing:

`$UAV_NAME/update_goal:` This topic stimulates the corresponding UAV to update its goal or path.

`$UAV_NAME/check_collision:` The messages sent through this topic are received from `quad_controller` node.

Subscribing:

`$UAV_NAME/remaining_step:` Receives the remaining path length at each finished step. These values are stored in a matrix and used in priority calculation.

`$UAV_NAME/arrival:` This topic used only once per UAV. If a UAV finished its path, it is recorded as "arrived". This information used in the priority calculation.

4.3 ControllerCallbacks.cpp

Callbacks declared in the `Wrapper::QuadController` class are defined in here.

4.4 quad_manipulator.cpp

Driver source file for the quadrotor manipulators. PID unit is initialized and objects are created in here. The node name is `quad_controller`.

Publishing:

`$UAV_NAME/cmd_vel:` This topic sends the calculated velocity.

`$UAV_NAME/Done:` Simply sends a done message. If necessary logic would be added to the user interface, further goals can be given after this message is received.

Subscribing:

`$UAV_NAME/move_base_simple_/goal:` Receives the goal from this topic.

`$UAV_NAME/ground_truth_to_tf/pose:` Continuously receives the current position of the UAV. This is important, because the velocity computation strictly bounded to correct detection of robot's location.

`$UAV_NAME/check_collision:` If collision risk is detected, a warning message is received from this topic, and corresponding UAV will brake until it stops.

`$UAV_NAME/path_number:` A discrete integer that represents the number of waypoints in the path is received from this topic. This integer simply used to scale velocity, because there is an empirical issue regarding to faster execution of UAVs. That is, there is an inclination to low-speed flight when there are relatively more waypoints in the path and vice-versa. The path number is multiplied by a constant and resulting value is used as scale for calculated velocities.

4.5 PlannerCallbacks.cpp

Callbacks declared in `WrapperPlanner` class and some auxiliary functions are implemented in here.

4.6 motion_planner_ompl3D.cpp

Driver source file for the planners of UAVs. Planner objects are created in here. A threaded approach is embraced. The node name is `planner_with_ompl`.

Publishing:

$UAV_NAME/remaining_step: This topic sends the calculated path length.
$UAV_NAME/move_base_simple/goal: Sequentially sends the waypoints as goal to the quad_controller node.
$UAV_NAME/arrival: Sends the arrival information about the UAV.
$UAV_NAME/path_number: Sends the discrete number of waypoints in the path.

Subscribing:

$UAV_NAME/Done: Receives done message as the steps are finished.
$UAV_NAME/actual_uav_goal: Receives the ultimate goal from the user interface.
$UAV_NAME/update_goal: Listens for a stimuli about the goal or path update. If receives such stimuli, a new path computation is initialized.

4.7 point_cloud sources

Since, point_cloud is not integrated to the project yet, there isn't any topic/node related information.

4.8 quad_user_interface.java

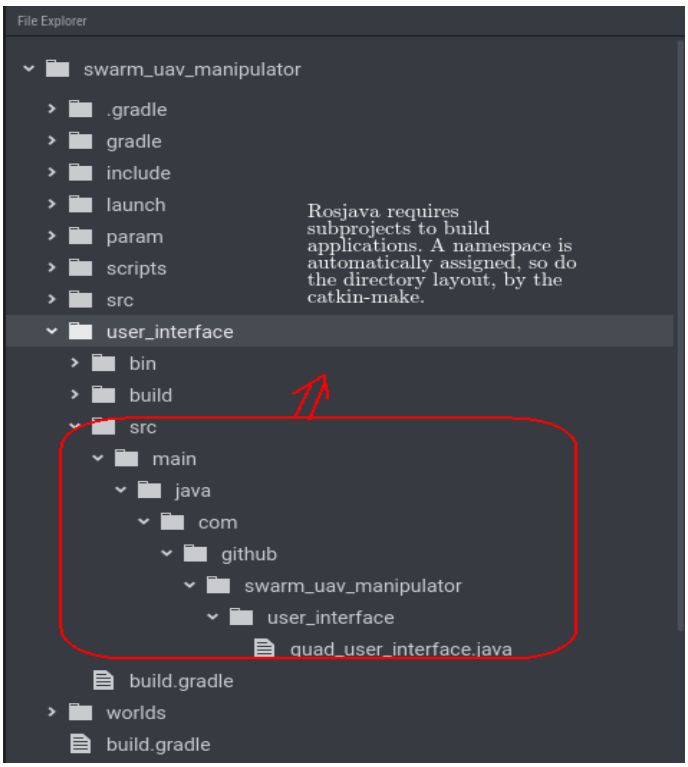

Figure 2: Java Sources Layout

Implemented in Java. JavaFX is used to build the GUI. The node name is `quad_ui`. The only publication is done to `$UAV_NAME/actual_uav_goal` topic. It has an asynchronous structure natively because of the GUI.

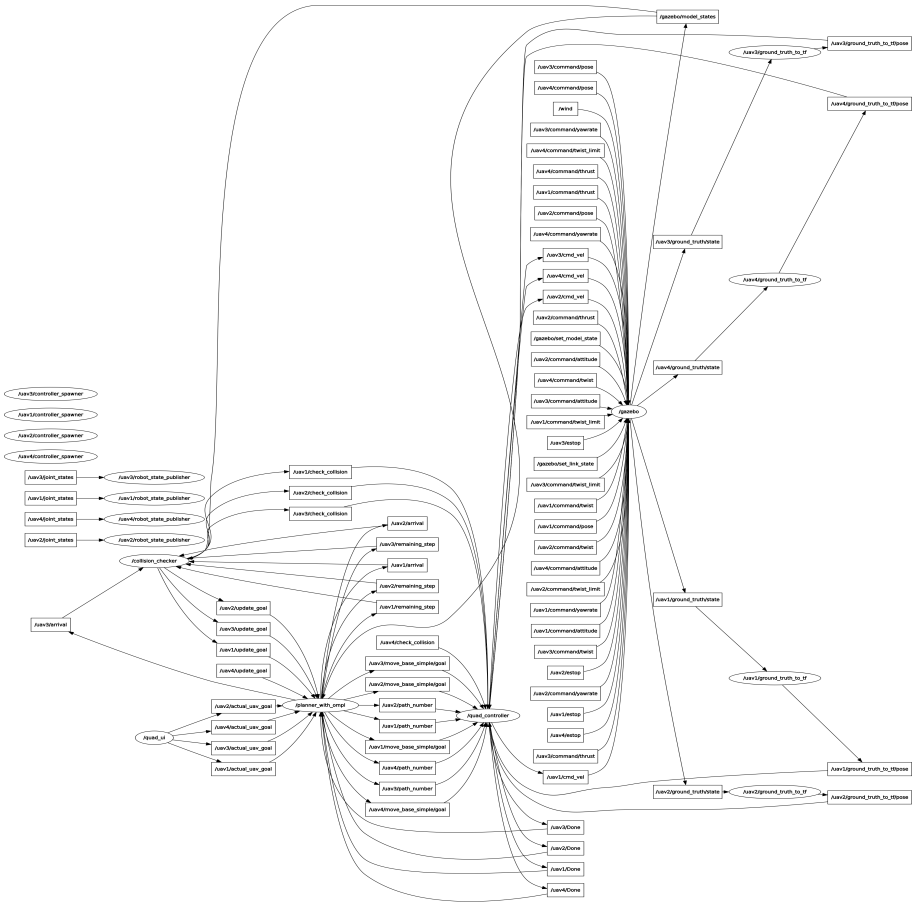

Figure 3: Topics and Nodes

In above figure, all topics, nodes and the relationships between them can be examined.

5 Development

This section comprises the tasks that we set as milestones throughout our development process. There are 5 major tasks. Each task has more difficulty than the previous one, hence reflects a *building block* process. Readers who want to replicate this project or improve their own development can follow this guide exactly or consider it as a template.

5.1 Task: Motion Planning of a Single Drone in Collision-Free 3D Environment

We have devised this as the first task for the project. It was succesfully accomplished in the end of first week.

In the development phase, our first approach was using Action Library Stack and it gave us the correct result. **However**; as we further inspected the logic behind, it is understood that Action Server directly communicates with Gazebo Plugins in Hector Quadrotor Package as Action Clients, and the rest of the process is automated by actionlib stack. Meaningly, the quadrotor can fly up to the point designed as goal through an optimal way, but we cannot control the velocities, wrenches, attitudes and other control units. This might not be a problem for the simulation purposes; but, in the sense of real-world project, we have to be able to control these units and critically manipulate them in the case of anomalies and unexpected events. Hence, this approach was abandoned.

Although the first approach was abandoned, it was not totally fruitless at all. We have reverse engineered the two plugins: `position_controller.cpp` and `velocity_controller.cpp`. Further inspection can be done from the hector_quadrotor_controller repository[1] if desired. During our inspection of these controllers, we have encountered a control algorithm: Proportional-Integral-Derivative (PID) Control. For this task and throughout the project, we have used this algorithm and it didn't create any major problems. In simple terms, it computes the difference between desired and current magnitudes and applies a smooth alternation. This feature makes it a perfect option for us, since our drone(s) should not make immediate bursts or brakes. End result can be viewed from here.

5.2 Task: Motion Planning of Multiple Drones in Collision-Free 3D Environment

We experimented this phase of the project in most of the 2nd week.

The task is fairly straightforward compared to what we have done for 2D version. As for source code, there was not much changes in the logic of the process. We wrote classes for controllers and gathered common callback rou-

[1] Keep in mind that there are 3 controllers in the /src directory of the repository: Position, Velocity and Attitude Controllers. Since we are not experts on control terms like wrench,thrust and attitude and there wasn't any problem during the flight, we only implemented our version of Position Controller, that is `quad_manipulator.cpp`. If, problems regarding to movement of quadrotores are encountered in further iterations of the project or current manipulation of drones are not sophisticated enough, one should consult to this repository.

tines and helper functions as there is one controller instance for each drone spawned. In order to enable correct copies of the classes and callback routines respond to messages, we added a data structure that maps from the names of the drones active in the simulation to their specific controller instances. That data structure is an `std::map` from C++ STL. Overall class name is `QuadController`.

While we were constructing classes, we encountered a problem caused by callback routines. If we add them into class definition, the callbacks should be defined as static routines. That causes another problem, a static method in a class cannot make use of non-static members. We solved this issue with defining a class wrapping all the details without changing the static methods and allows us to use non-static members of that wrapper class as well. That class is named as `Wrapper`. Then, overall controller instance has become `Wrapper::QuadController`.

The issues of this part of the task were about the launching configurations of the simulation and essential controller components most of the time. To solve this, we tracked down the launch files we used for single-drone motion planning task and found out that one launch file initiates the essentials of 2 quadrotors in the simulation at `hector_quadrotor_kinetic_devel/hector_quadrotor_gazebo/launch`.

Then, we created a file called `spawn_multiple_quadrotors.launch` so that when we simulate multi-drone motion planning, we will have the number of drones in the simulation as we specified in that launch file. One can increase the number of the drones by adding the repetitve pattern of group tags and launch them with **roslaunch**.

After adding multiple drones into simulation, the first thing we did was to give them the same ultimate goal position and see what happens. All drones obtained the position successfully; however, since there was no actual motion planner or collision avoidance module, the drones ended up hitting each other trying to reach their ultimate goal.

Then we assigned different goal points to UAVs to illustrate their proper and collision-free execution throughout the simulation.

5.3 Task: Motion Planning of a Single Drone in Collision-Rich 2D Environment

At the end of 2nd week, we are given the task of motion planning with single robot in an environment with obstacles.

The first phase of development was to construct the essential parts of the planning process. Firstly, Rapidly-exploring Random Trees (RRT) algorithm

is considered for planning algorithm. This algorithm samples points in the space and checks if those points fall into collision areas. If so eliminates them, otherwise checks them with the nearest point to them accepted up to that time instance. At last, it checks if motion between those points are valid or not in an incremental manner. If it goes through an obstacle, then its not connected to the tree. Resolution can be adjusted. Variations of algorithm may try other eligible points as well. Secondly, a random number generator is added to source code. Random number generator is the sampler for RRT algorithm. It was given the bounds of the space where the initial robot position and the ultimate goal position exist. Thirdly, to build a map, a tree data structure should be constructed. If RRT accepts the point and then the motion, the line (edge for the tree) is added to the map. Then, there is a search algorithm to use when tree construction is completed. The simplest approach is to go over the parents of the nodes starting from the goal node, which is added to the tree the last, and find the initial position of robot to determine the planned path.

This approach is hard-coded at the first phase of this task without any help from third-party motion planning libraries. The source code is in the file **motion_planner.cpp**. However, hard-coded approach failed to construct fully valid paths as some of the line pieces were going through obstacles in the environment and reflected into tests as the quadrotors colliding with them. Additionally, the planned path was not optimal. Therefore, a new approach was planned. A third-party library is decided to be used in this task. OMPL has essential motion planning components for multi-dimensional spaces tested.

After the OMPL decision, we changed the planning configuration to use, firstly in 2D space, RRT* algorithm as planner. The difference between original RRT and RRT* is that latter uses a probability to sample points/states near goal point or the goal itself. The first tests showed that RRT* of OMPL was much more efficient in terms of the path validity and optimality. In that sense, OMPL library has its planners tested. We changed our approach by integrating OMPL into our source code, **motion_planner_ompl3D.cpp** and **motion_planner_ompl2D.cpp** for 3D and 2D path planning respectively. As to bulding the essentials, we defined the bounds of the 2D space by $[-10, 10]$ and chose manually constructing the problem as `ompl::base::ProblemDefinition` class requires. For the start and goal positions, it is required to set the orientation for `ompl::base::ValidStateSampler` to generate valid states/positions in space. With the help of open tutorials of OMPL, writing source code for motion planning with OMPL was easy. Finally, this phase of the project has ended in the 4th week.

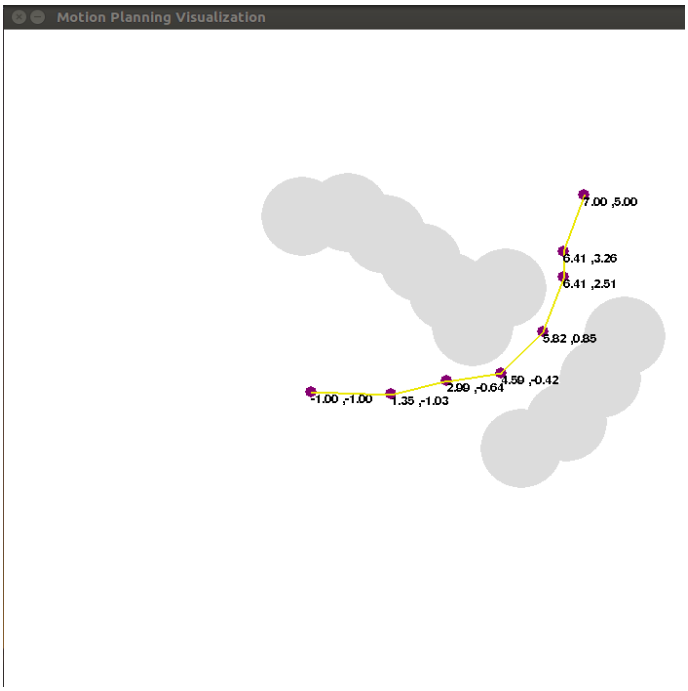

Figure 4: Static Motion Planning with Single Drone

Path planning can be viewed in Figure 4. It is fairly straightforward. The execution of simulation can be watched from here

5.4 Task: Motion Planning of Multiple Drones in Collision-Rich 2D Environment

This task is the natural follow-up of the *Single Drone in Collision-Rich 2D Environment*. It lasted approximately 3 weeks and by far the most difficult task that we had to cope with. In the end, it is mostly solved. The minor problems regarding to all sorts of possible approaches are discussed later on in this subsection. At first, we had to manage the planning of multiple drones, we need to convert to an object-oriented approach and thus devised a **Planner** class. After this first adjustment, we had to solve the dynamic obstacle detection. The problem difficulty is increased through this problem where the most work and decision process that we got under. Naturally, UAVs must be considered dynamic obstacles to each other and during execution time they should not collide with each other.

5.4.1 Approach: Paths as Obstacles

In this section, **Paths as Obstacles** approach is described in detail. There were 2 implemented versions of it; but, current source files don't include the second one. Despite it does not exist concretely, a thorough description of it is provided later on this subsection, hence it would be easy to implement that version, too.

First version is a totally static implementation. It observes other UAVs' paths as obstacles. Therefore, even the paths do not intersect with each other. By applying this as a function, one can wipe off any collision risk, but the trade-off is non-optimal paths and some of the UAVs may not be able to plan their path at all. In the cases where collision avoidance is much more important than always-optimal path planning and UAV number is small, this can be a suitable approach to adopt.
As the theoretical aspect is explained, one should also consider the practical aspects to familiarise with intrinsic features. Although the planners belonging to UAVs work in parallel on different threads, their planning and movement is synchronized in order to sustain a consistent application execution. That is, none of the UAVs move before all the UAVs plan their own paths. This is known as Rendezvous Problem. Synchronization is done by a Semaphore. Since Standart C++ Library does not implement Semaphore, we have implemented a minimalistic version of it using Mutex and Condition Variable.

Second version adopts a dynamized implementation. It is developed, in order to solve some deadlock problems arise from first version. First version's problem is that it plans paths only once. No UAV could have the chance to replan its path.

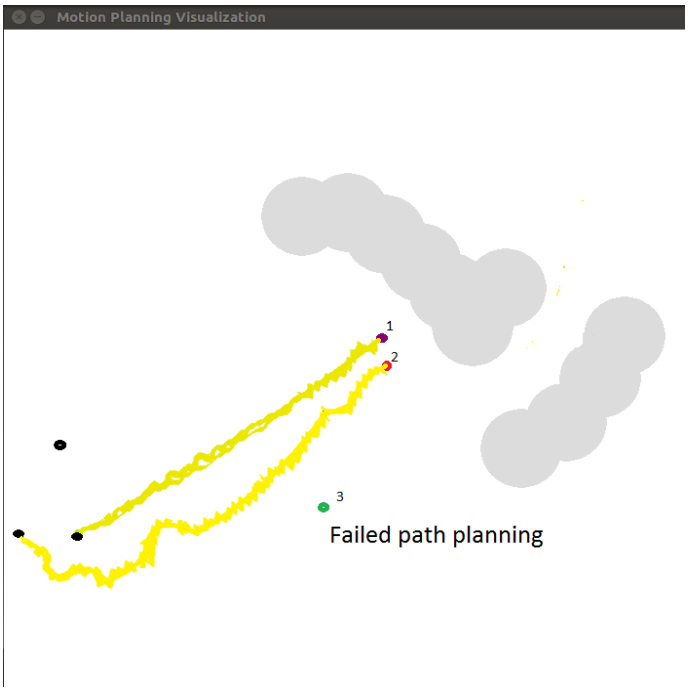

Figure 5: Static Motion Planning with Paths as Obstacles

As it can be seen from the Figure 5, the 3rd UAV cannot plan its path in the given time, since it will fail totally. Second version removes the path segments of reached waypoints and when a UAV does this, it informs the waiting UAV (since it couldn't plan its path, it must wait until obstacle field reduces). Then, waiting UAV tries to plan a path, if it cannot find a path again, same procedure recurs until UAV plans a valid path. Wait operation of UAV can be done with a Condition Variable. The trade-off of this approach is somewhat intuitional. Optimal paths are not guaranteed to be produced, and as the UAV number increases, total execution time may increase exponentially. After reading the description above, one should be able to easily implement the procedure.

5.4.2 Approach: Dynamic Origin Check with Static Planning

This approach is the basis of our current implementation strategy. It does not prevent the planners from producing optimal paths. Therefore, it is much more suitable when path-optimality possesses great importance. Planners plan the paths only once considering only static obstacles, thus collision avoidance responsibility is assigned totally to `collision_checker.cpp`.

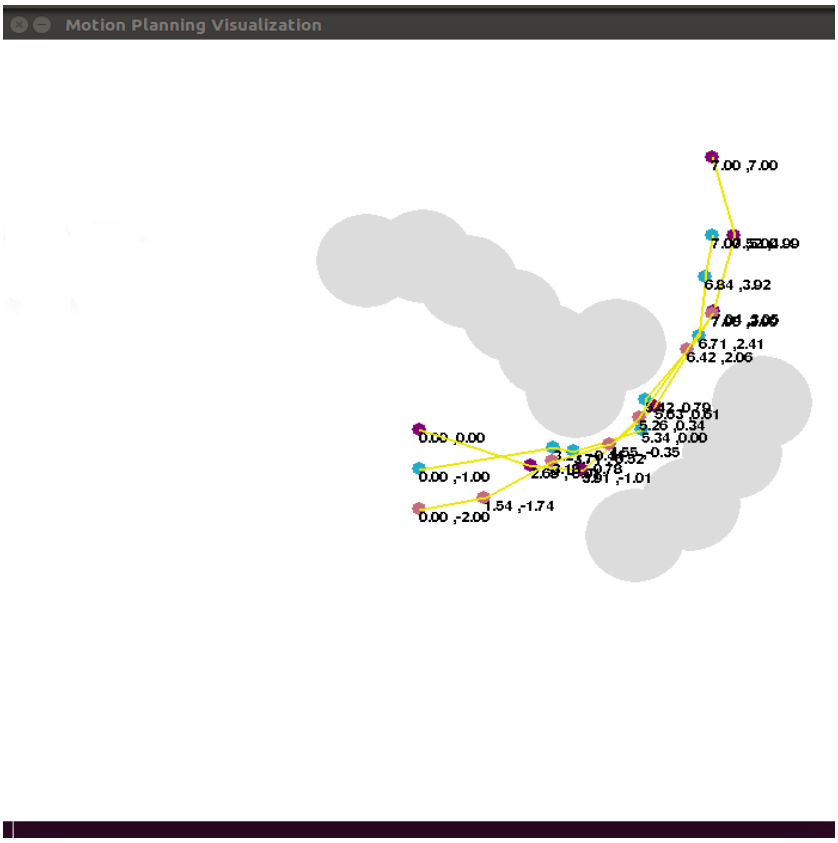

Figure 6: Static Motion Planning with Dynamic Origin Check - 3 UAV

Mechanism of Collision Checker is simple. It continously calculates the distances between each UAV. If there are UAVs in the collision radius[2], it sends a **STOP** message for a lower priority UAV. This priority is calculated based on remaining path length to reach ultimate goal.
This is an over-simplified approach for priority calculation; therefore, for certain cases, it would produce bad results. For example, head-to-head collision risks cannot be solved by this approach. There are many algorithms proposed to solve such conditions. One promising algorithm is Artificial Potential Fields (APF). It would provide great usefulness and solve many major problems related to current implementation, if properly introduced to the project. For further information on how to enhance the project, please refer to **Future Improvements** section.

Another problem of this approach is, it cannot understand whether other

[2]Can be given different values for different safety measures, but it should be at least 1.4, since the sum of horizontal radiuses of 2 drones equals to 1.4

UAVs reached their goal or not. When a UAV reaches its goal, its remaining path length equals to 0. This results in a priority problem, because it will always have the highest priority and other UAVs may "caught" before reaching their goal and stay still.

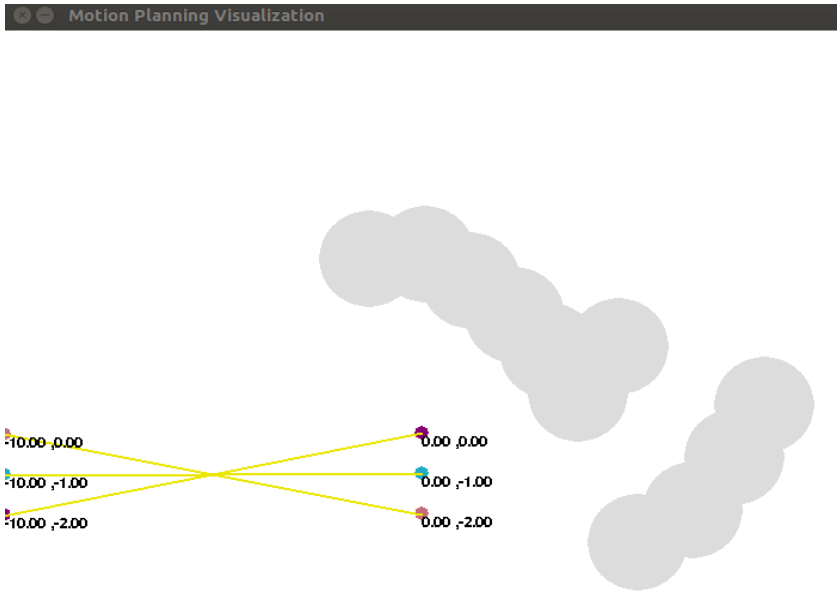

Figure 7: Dynamic Motion Planning with Dynamic Origin Check on Open Field - 3 UAV

As illustrated in the figure above, when there are no obstacles between the goal and initial points, the paths are fairly straightforward. The simulation execution of this scenario can be seen from here and here.

5.4.3 Approach: Dynamic Origin Check with Dynamic Planning

Our current implementation adopts exactly this approach. As it is mentioned, we had to understand whether UAVs stopped or not. Understanding it was

not adequate to solve the entire problem, because UAVs could have stopped in other UAVs path. This obligation enforced us bring back the Dynamic Planning. Replanning is restricted to the cases such that when a UAV finishes its movement -that is, it becomes a static obstacle-, other UAVs replan their paths considering finished UAVs as static obstacles, too.

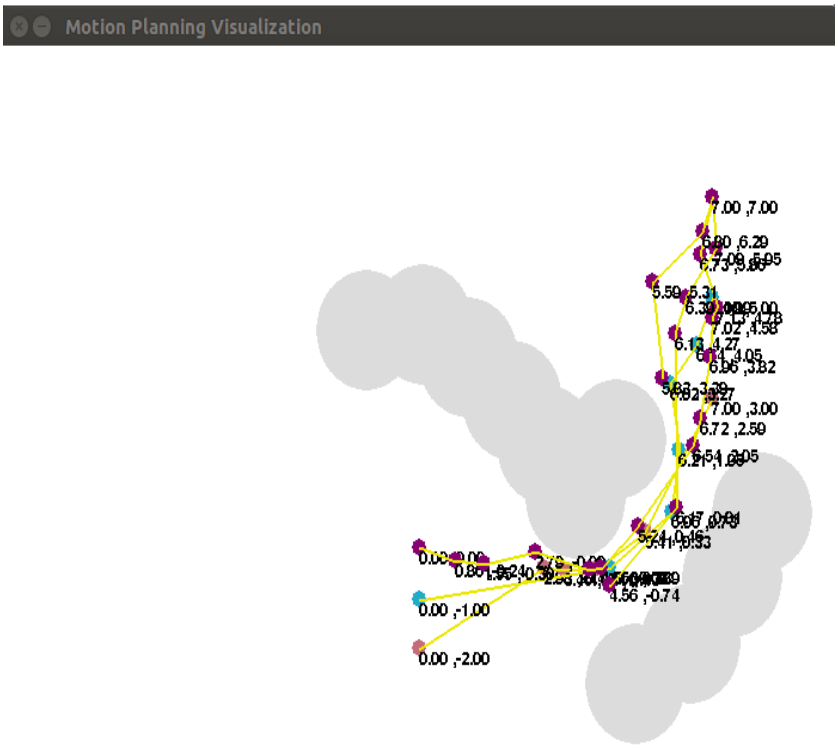

Figure 8: Static Motion Planning with Dynamic Origin Check - 3 UAV

As illustrated in Figure 8, after first UAV finishes its movement, other two UAVs replan their paths and avoids collision. At first, their path were at intersection with each other. Through that, optimality is covered at high percents and collision is avoided. However; when a UAV plans a new path, the starting point of replanned path may stay behind of the current position of UAV. This decreases the optimality of the plan and execution time increases. Possible ways of optimizing this implementation is covered in **Future Improvements** section.

See the videos from here, here and here. It can be observed that as planning time increases, "stay-behind" effect decreases. This is an empirical informa-

tion, so one can devise more experimentations to understand the exact relation.

5.5 Task: Motion Planning of Multiple Drones in Collision-Rich 3D Environment

We are given this task in the 4th week and accomplished it in 5th week.

This task was straightforward as the only thing needed was to change the instructions from 2D to 3D. The `std::pair<T, T>` used to represent the centers of any static or non-static model in simulation are changed into `std::tuple<T, T, T>` with 3 floats representing (x, y, z) coordinates. To make it more readable, we changed the long name as `Coord3D`. Additionally, we set the state space configuration to `ompl::base::SE3StateSpace` and set the bounds for x and y as $[-10, 10]$ and for z as $[0, 4]$. In that sense, the planner is allowed to sample points at most 4 meters up from the **ground_plane**. With these changes, we had to change our collision checking mechanism with a simple addition of z axis in the equation.

The first time we tested our 3D planner, RRT* failed to find a valid path and gave an error indicating that the start position is invalid. Then we asked this issue to the actual developers of OMPL library via *BitBucket*. As stated in the link, it was an invalid orientation issue. Since we did not define a valid orientation, state sampler generated invalid samples from the beginning. `ompl::base::SE3StateSpace` has 2 distinct components:

1) `ompl::base::RealVectorStateSpace`: Holds the (x, y, z) coordinates of the state.

2) `ompl::base::SO3StateSpace`: Holds the $(roll, pitch, yaw, w)$ orientation values of the sampled state.

When we set the start and goal orientation to identity with `SO3StateSpace::StateType::setIdentity()` method, we solved this issue.

The next issue was, again stated in the same link above, even if the orientations were valid, RRT* still failed to find a valid path. The reason was quite simple though. When configuring the planning essentials, we set the bounds of the state space for the sampler. Since we were working on 3D space, we set lower and upper bounds for x, y, z coordinates with `ompl::base::RealVectorBounds::setLow` and `ompl::base::RealVectorBounds::setHigh` respectively. Each coordinate has an index $x : 0$, $y : 1$ and $z : 2$. The functions are overloaded for bounding specific axes. Problem was that the indexes were forgotten to be put into these functions, when each time these functions are called all bounds are changed instead of individually. We solved this issue by adding the indexes to the functions.

5.6 Task: Detection of Non-uniform Objects and Sampling the Environment

As you can see in our previous tasks, uniform unit spheres is used as obstacles. However, in real world environment there will be many objects that are not uniform such as mountains or trees. We need to somehow represent these obstacles and this task aimed to do that.

At first, it is thought that objects could be represented by unit spheres since it would be easier to check whether there is a collision or not with spheres, but we cannot know the size or the shape of the objects and this will make representation process problematic. We searched on Internet for solutions like: "Can we use world or URDF file in order to determine the size or shape of the obstacle? " However, there was no such implementation related to that.

After that, by looking at some research studies and projects it was decided that using Point Cloud Library (PCL) would be better. This library is using the camera of the robot and converting the seen environment into points based one. To be able to use point cloud we first need an UAV that has Kinect or Asus camera. In the `hector_quadrotor-kinetic-devel` package there are URDF files for quadrotors with cameras and we tried to spawn an UAV using those urdf files by changing our launch file. However, despite all of our effort the package did not accept any of them. It was seen that only UAV with Asus Xtion can be spawn into an empty world, so we used it to initiate the process.

Since we had our Asus Xtion camera, it was time to use point cloud library. We wrote the C++ nodes that are named as `point_cloud` and `point_cloud_read`. Below is the step-by-step execution:

- Launch a world, spawn an UAV and put an object in front of it using following commands:
  - `roslaunch gazebo_ros empty_world.launch`
  - `roslaunch hector_quadrotor_gazebo spawn_quadrotor_with_asus.launch`
- Run the command to open rviz:
  - `rosrun rviz rviz`
- In another terminal run the command:
  - `rosrun [your_package_name] point_cloud`
- On the display part of the rviz window choose the frame id as the id that you arrange in these codes it is base_link.
- On the display part push the "Add" button and choose the "PointCloud2".

- In the PointCloud2 part choose the topic that is related with your point cloud data in this case it is /output.
- Then you should see the point cloud in base frame. You can choose the frame that you are interested in.

This was the basic representation. We also checked whether using Octomap would be better or not and we tried to generate an octomap but there were problems. After a while we realised that octomap also uses point cloud data, so we skiped it. Here is some pictures that illustrates sampling:

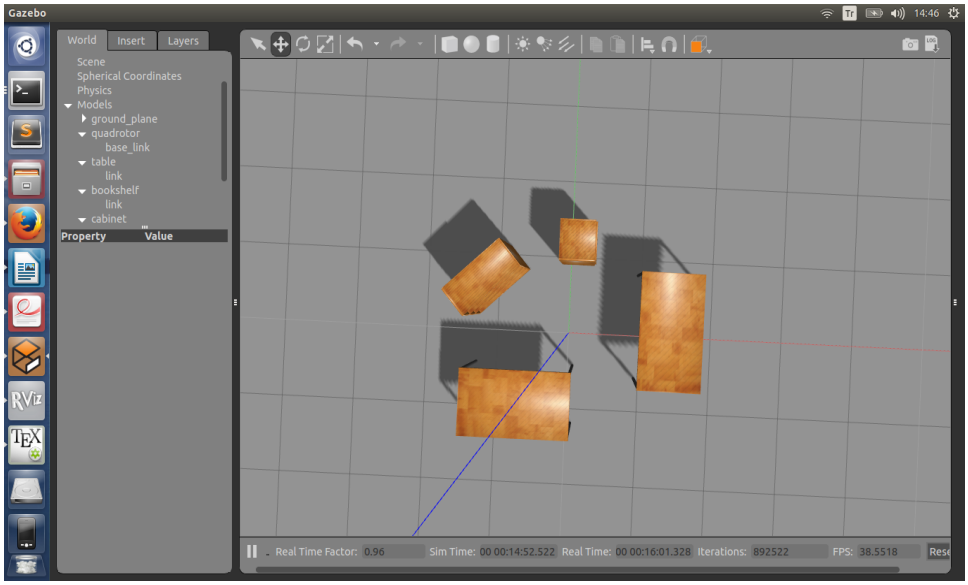

Figure 9: Original Environment

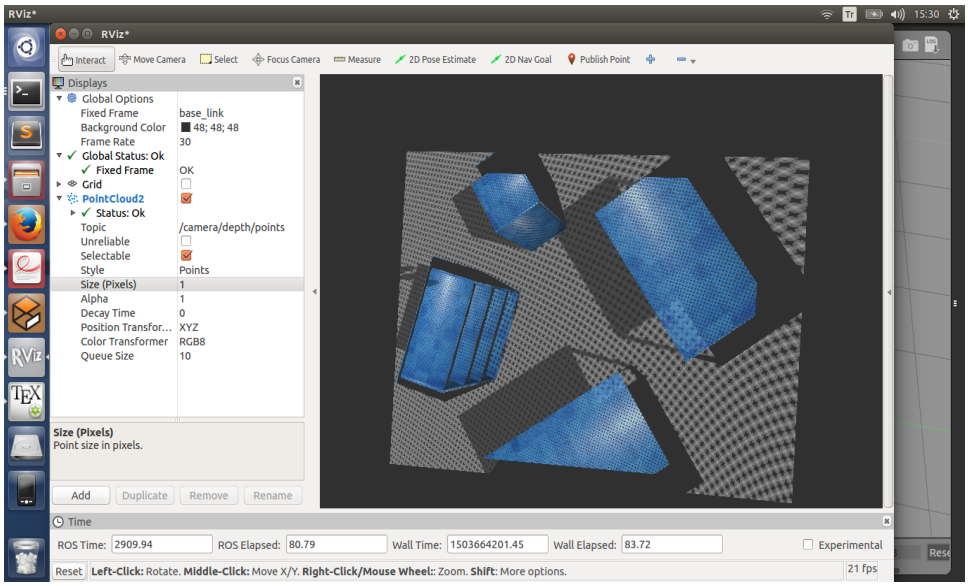

Figure 10: Environment represented with points

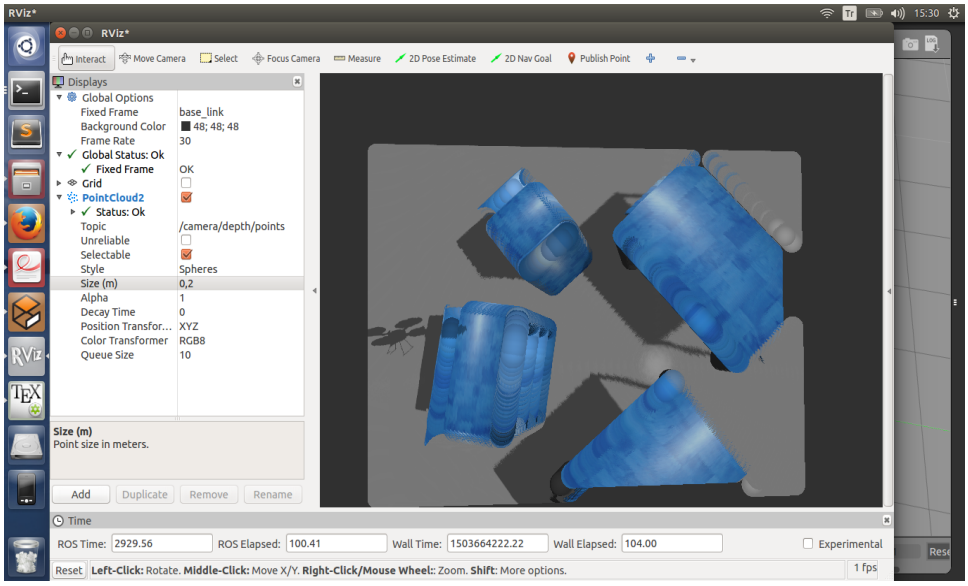

Figure 11: Environment represented with spheres

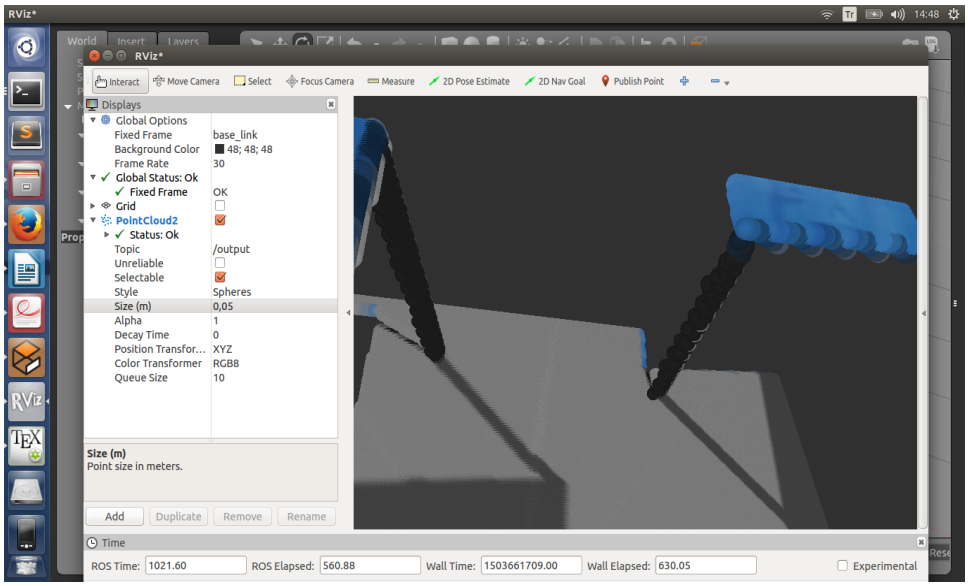

Figure 12: A part of the sphere representation

6 Known Bugs

Unfortunately, the **Detection of Non-uniform Objects and Sampling the Environment** was not completed due to some problems:

- Neither an UAV with Kinect nor with Asus Xtion could not be spawned into our package.
- The `point_cloud_read` node, which is written to generate PCD file, etablish an error. The solution could not be found after searching on Internet.

Motion Planning has one issue to be considered:

- UAVs can plan their optimal paths. After some of them reaches their ultimate goal, replanning of others occur. After these replanning phases, the new paths usually force UAVs to fly back their previous step waypoints. These might sometimes result in minor collisions among UAVs.

- The warning at **line 22** in `spawn_quadrotor.launch` of `hector_quadrotor_gazebo` package should be taken into consideration. Simply change the parameter **default** from "world" to "/world".

7 Future improvements

- Replanning of the paths are problematic as it was described before. In this context, problematic means that non-optimal path production and minor collision risks regarding to over-simplified priority policy. A new method that pre-controls the current paths of UAVs regarding to recently stopped UAVs positions can be implemented. This would dramatically decrease the execution time as UAV number increases.
- Other issue regarding to replanning is the braking issue. If we wait for UAV to literally stop after STOP command is received, we would waste a considerable amount of time. If we don't wait, the initial location that was fed into planning and current UAV position differs a lot. Hence, inconsistent flight condition may occur(e.g going back to the most initial point.). This can be solved by devising a unnecessary waypoint cancellation routine.
- To sample the environment you should make your UAV discover the environment since you need camera and you should be careful about the range of the camera.
- Since there was a problem of spawning UAV with Kinect, you can try to solve this problem. If possible, we advice you to choose Kinect camera as it gives you information about depth.
- A node that runs Pygame is already implemented to visualize 2D path planning. For 3D visualization purposes, you may use rosoct package.

8 Extra Resources

A Lecture that Discusses RRT Algorithms
A Study on Artifical Potential Fields
Artificial Potential Fields - First Paper
Multi-UAV Path Planning in Obstacle Rich Environments Using RRT