Project 5: The Final Race

Team 6: Noob Quaternions

Flying Nimbus

Prateek Arora
Masters of Engineering in Robotics
University of Maryland, College Park
Email: pratique@terpmail.umd.edu

Abhinav Modi
Masters of Engineering in Robotics
University of Maryland, College Park
Email: abhi1625@umd.edu

I. Overview

The aim of the project is to complete the final track (figure 2) consisting of sub tasks that we completed in the previous projects. We named this project **Flying Nimbus** inspired by the flying nimbus in the Dragonball Z anime.

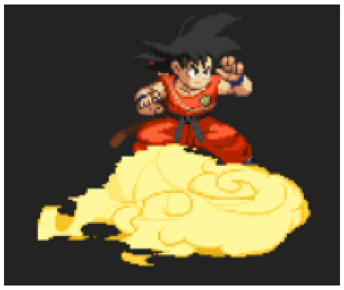


Figure 1: Flying Nimbus

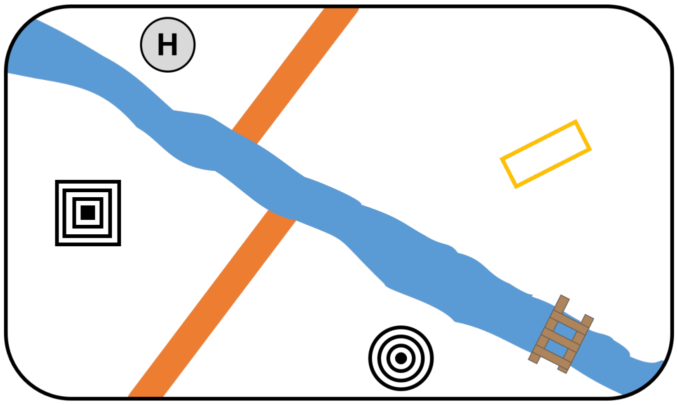


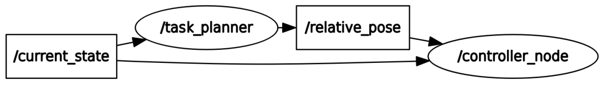
Figure 2: System Architecture

II. System Architecture

For this project we redesigned our software structure into 3 modules:

1) State Estimation
2) Controller
3) Task Planner

All these modules use ROS framework and thus run in parallel as 3 different nodes.

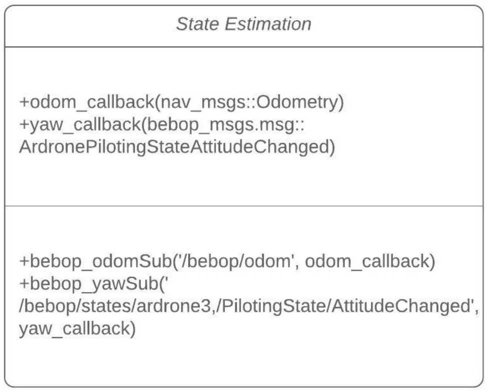


$$\begin{tikzcd}
 & \text{/task\_planner} \arrow[r] & \text{/relative\_pose} \arrow[dr] & \\
\text{/current\_state} \arrow[ur] \arrow[rrr] & & & \text{/controller\_node}
\end{tikzcd}$$

Figure 3: System Architecture

A. State Estimation Module

This node subscribes to bebop's odometry messages and computes position by integrating velocity messages. The yaw is computed using a different message - */bebop/states/ardrone3/PilotingState/AttitudeChanged* which we found out was more stable as compared to orientation messages form the bebop's odoemtery. The State Estimation module design is shown here:



State Estimation

```
+odom_callback(nav_msgs::Odometry)
+yaw_callback(bebop_msgs.msg::
ArdronePilotingStateAttitudeChanged)
```

```
+bebop_odomSub('/bebop/odom', odom_callback)
+bebop_yawSub('
/bebop/states/ardrone3,/PilotingState/AttitudeChanged',
yaw_callback)
```

Figure 4: State Estimation Module

B. Controller Module

This node subscribes to current state from the state estimation node, target messages from Task Planner nodes and computes control inputs which are essentially velocity commands for the Bebop. The control design is different for x-y, z and yaw. We developed an LQR controller for position and trajectory following in X-Y which was explained in detail in one of the previous projects. Whereas the height and yaw controller were simple proportional controllers. The module design is shown below:

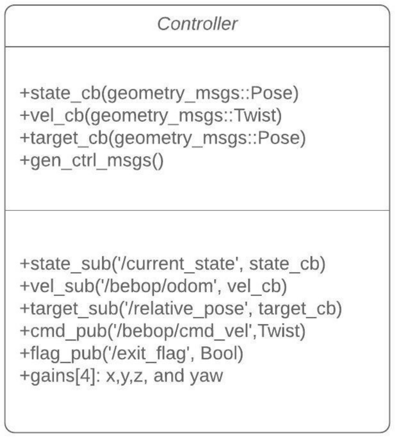

Controller
+state_cb(geometry_msgs::Pose)
+vel_cb(geometry_msgs::Twist)
+target_cb(geometry_msgs::Pose)
+gen_ctrl_msgs()
+state_sub('/current_state', state_cb)
+vel_sub('/bebop/odom', vel_cb)
+target_sub('/relative_pose', target_cb)
+cmd_pub('/bebop/cmd_vel',Twist)
+flag_pub('/exit_flag', Bool)
+gains[4]: x,y,z, and yaw

Figure 5: Controller Module

C. Task Planner Module

This was the main module in which all the different missions were executed. Each mission publishes target states which are relative target poses in the quadrotor's body frame. These messages are then used by the controller to generate the required velocity commands for the bebop. The module design for this module is hown below:

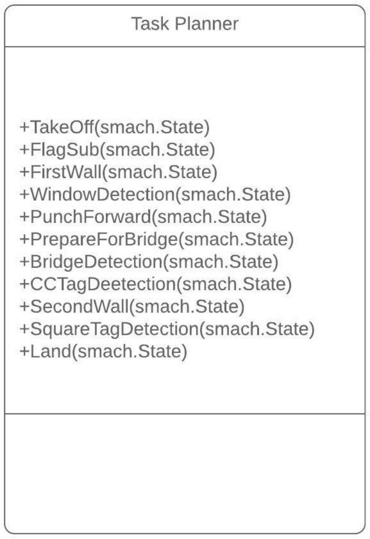

Task Planner
+TakeOff(smach.State)
+FlagSub(smach.State)
+FirstWall(smach.State)
+WindowDetection(smach.State)
+PunchForward(smach.State)
+PrepareForBridge(smach.State)
+BridgeDetection(smach.State)
+CCTagDeetection(smach.State)
+SecondWall(smach.State)
+SquareTagDetection(smach.State)
+Land(smach.State)

Figure 6: Task Planner Module

For designing this module we used SAMCH- which is short for State Machine. It is a python library for hierarchical state machines. This made adding new missions easy as each mission requires only an *init* method and an *execute* method. It inherently uses service calls to run each mission as and when the previous mission returns a predefined string upon exiting. The graph visualization for the state machine wasn't generated since the smach viewer is only compatible with ubuntu 14.04. We used this software architecture because we wanted to make our state estimation and controller independent of our missions, so that these two are always running in parallel enabling simple hover in case no new mission is running.

III. Issues

We encountered a multitude of issues ranging from occasional driver issues to fully frying the upboard. The bebop, on the other hand, was very reliable. The most time consuming task was the development of the software design.

IV. Future Scope and Improvements

Kalman filter: Our vision node runs at 5 hz while controller node runs on 10 hz. Due to slow feedback from vision node, the drone drifts during flight. A kalman filter can be implemented to get better estimates of state.
Drift compensation: After each yaw there is drift in bebop's position. Due to this the drone drifts away in a way the the window gets out of the frame.

Better controller: For our current state machine, we felt like we needed different gains specific for each task. A better controller which doesn't need gains scheduling could have been implemented.

video link: YouTube
Github page: Github

V. Acknowledgement

: We would like to thank our instructors Nitin J Sanket and Chahatdeep Singh for teaching the course and we acknowledge the amount of effort they have put to create the course. We would also like to thank them for their help as and when required.