ENAE788M Project 2
Team Bouncing Rainbow Zebras

Erik Holum
Graduate Student
University of Maryland
Email: eholum@gmail.com

Edward Carney
Graduate Student
University of Maryland
Email: carneyedwardj@gmail.com

Derek Thompson
Graduate Student
University of Maryland
Email: derekbt@yahoo.com

***Abstract*—In this paper, we discuss the implementation of control software for following three pre-specified trajectories using the Bebop drone. We provide an overview of the control software written the Robot Operating System (ROS). Discuss the details of the PID position controller and gain tuning, using both a simulated quadrotor as well as the actual Bebop. Then finally present experimental results for following the trajectories using physical hardware.**

X	=	x,y,z pose
V	=	Current velocity
X_{des}	=	Desired position
$X_{current}$	=	Current position
V_{des}	=	Desired velocity
V_{err}	=	Error in velocity
$V_{min/max}$	=	Max and min velocities
T_{proj}	=	Projected time step
A_{vel}	=	Acceleration
e	=	Error vector
K_p	=	Proportional gain
K_d	=	Derivative gain
K_i	=	Integral gain

I. Introduction

In this project, we utilized existing open-source frameworks and tool sets to develop and implement a software architecture capable of issuing trajectories to a Bebop drone and commanding the drone to follow these trajectories autonomously. Initial development was done in a simulation environment to verify software logic and process flow, followed by full hardware testing. This work culminated in a live test event where the drone was required to follow three different trajectories autonomously; a video capture system was used to obtain 'truth' data for the drone's performance to compare to the desired trajectories.

II. Implementation Details

We implemented all code in Python using the ROS framework [1]. In order to test our software setup without the Bebop, we made extensive use of a Gazebo simulation environment to verify software design and implementation prior to full hardware testing. This was done via the open-source tum_simulator available from the main ROS website [2] configured to run on Ubuntu 16.04. Although this simulation is designed for the Ardone quadcopter and therefore would require different gains than the actual PID controller that would be implemented for our Bebop, the framework still allowed us to test the design and implementation of our controller, waypoint publisher, and supporting modules in ROS without the additional constraints of hardware testing.

The first step was to write a communications class for reading/writing commands from either the bebop_autonomy ROS package [3]. We wrote a $DroneComms$ class that allowed us common access to the takeoff, land, command, and odometry readings from both the Tum simulation and the Bebop.

Our primary position control node is the $main\_navigation$, which handles wrapping the $DroneComms$ object in a skeleton code that enables position control of the quadrotor. The benefit is the precise details of both the controller and the underlying drone comms are not necessary for the main loop in $main\_navigation$. Desired positions are read from the $/main\_navigation/des\_pose$ ROS topic, and the resulting command velocity are published to the active quadrotor.

In order to execute the specified trajectories, we wrote the $waypoint\_publisher$ node, which reads pre-canned trajectory files, then sends desired positions to the $main\_navigation$ node for execution. We used a simple Euclidean error metric to determine when the quadrotor had reached the desired pose, r_{cmd}, namely

$$|e| = |X - X_{des}| \leq 0.1 \tag{1}$$

Once this condition is reached, the $waypoint\_publisher$ updates r_{cmd} to the next point in the trajectory file.

Finally, especially when first starting testing our code with the Bebop, we often found it necessary to halt the drone without it crashing into a net or the ground. We implemented a $ground\_control$ node, which leverages the PyGame (pygame.org) Python module and an xbox control to provide manual control of the Bebop. By flipping a condition in the $main\_navigation$ node, manual control is instantly passed to the xbox controller to allow the driver to safely land the quadrotor.

Each of these nodes and the relevant topics between them is presented in figure 1.

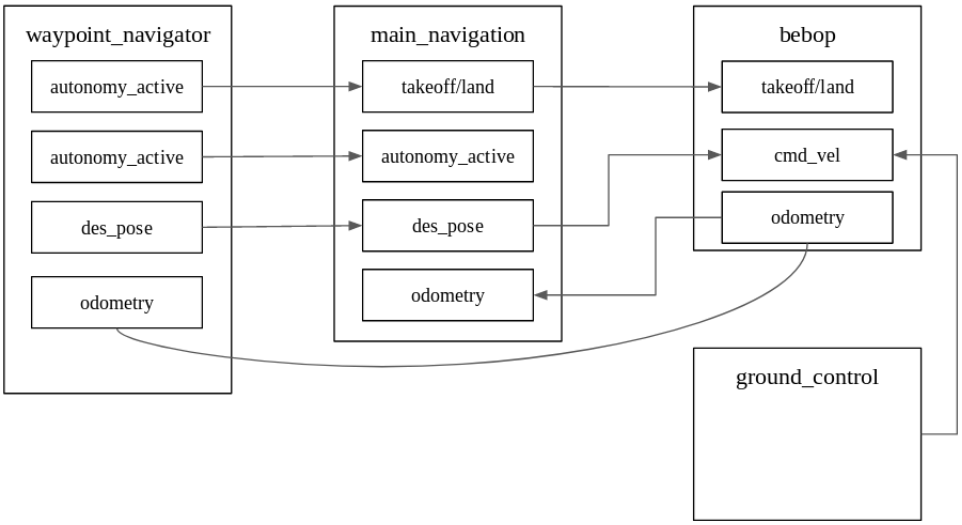

Fig. 1. Nodes and relevant ROS topics between them for controlling the Bebop. RViz and other tools not included.

III. Controller Design

A. PID Controller Design

In our initial implementation we used PID controllers for all three axis of the positional controller. After defining a point in space the decoupled X, Y, and Z controllers would calculate a output which would be translated to the vehicle frame and executed. Using the dynamically re configurable ROS parameters we were able to tune the drone to a state where it could complete the course but it would take a while. The control for the yaw and altitude were already very damped systems so a PID controller, or specifically a PD controller, worked very well with these controllers. The lateral movement however, was not very damped so we had to increase the derivative terms very high and turn the proportional very low to get a trajectory that did not consistently overshoot the desired point.

B. Gains Tuning

The Dynamic Reconfigure ROS package (wiki.ros.org/dynamic_reconfigure) provides a mechanism for modifying a ROS node's internal parameters during run time. The appeal in integrating this with our PID controllers was immediately obvious, since we were able to change K_p, K_d, and K_i while the Bebop was airborne.

Moreover, by using rqt as specified in the tutorials, we were able to use a GUI for both changing the gains, and to issue new desired waypoints in the world frame during flight. The combination of these two features allowed us to set gain values, issue positional commands, observe the drone response, and update the gain values based on the resultant behavior. We continued to iterate through this tuning process to converge on feasible gain values.

Visualization of the tuning process is provided in figure 3.

C. Velocity Controller

As we were not able to accurately control the position of the drone through a positional PID controller we decided we needed to implement a velocity controller on the lateral control. To implement a velocity controller we used a sort of receding horizons approach. The controller defines a desired velocity V_{des}, curve given by the equation below.

$$V_{des} = \min(\max(A_{vel}(X_{des} - X_{current}), -V_{max}), V_{max}) \tag{2}$$

The slope of the velocity with regards to position is defined by the velocity slope term A_{vel} and a absolute maximum velocity of V_{max}. At each call back the controller calculates the estimated position a time step T_{proj} in to the future, X_{proj}, and the desired velocity at this point $(V_{des})_{proj}$. With the desired velocity the time step T_{proj} into the future, a velocity error can be calculated and a subsequent commanded acceleration required over the time step to achieve the desired velocity.

$$V_{err} = (V_{des})_{proj} - V_{current} \tag{3}$$

$$A_{cmd} = \frac{V_{err}}{T_{proj}} \tag{4}$$

Knowing that the drone operates in an altitude hold mode the commands to the drone can then be calculated from the resulting lateral accelerations from a given roll/pitch input.

$$command = (\arctan(A_{cmd}/9.81) * 180/\pi)/40 \tag{5}$$

The lateral controllers work in the global X and Y directions and the commands are rotated to the vehicle frame before being sent to the drone.

IV. RVIZ Display

In order to aid development of the controller the vehicle, target, and commands were displayed. The target position was

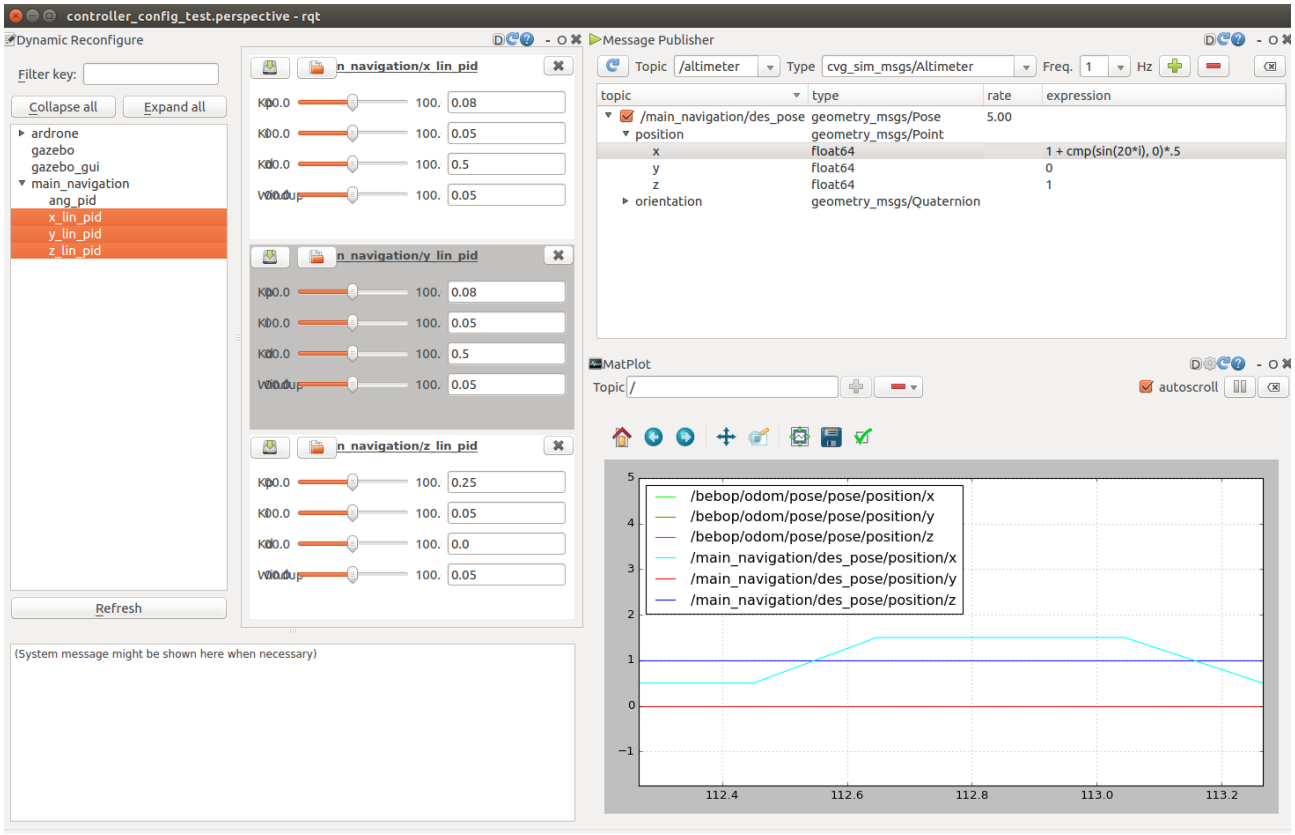

Fig. 2. Tuning PID gains using the rqt dynamic reconfigure package.

displayed as a red ellipsoidal sphere as it was moved around the environment. Another very helpful feature was to plot the navigation commands in the vehicle frame. The commands are projected from the vehicle displaying the controller output with regards to the vehicle.

V. Results

Results for the three tested trajectories are presented below. Vicon data captured during testing are compared to the commanded trajectories sent to the Bebop. It is apparent from these plots that while the general trajectory is maintained for each scenario the odometry has suffered from some degree of bias in each case. It should also be noted that the Vicon system could not capture the entirety of trajectories 2 and 3 due to the height the Bebop was required to fly to.

A. Trajectory 1 (Helix)

Results for trajectory 1 are presented in Figures 5 - 7.

B. Trajectory 2 (Diamond)

Results for trajectory 2 are presented in Figures 9 - 11.

C. Trajectory 3 (Staircase)

Results for trajectory 3 are presented in Figures 13 - 15.

VI. Videos

Videos of the resultant test flights are available at:

1) https://www.youtube.com/watch?v=2UQTmHEkgWY
2) https://youtu.be/Cybi_7t54Sw
3) https://www.youtube.com/watch?v=LX-WfwG-pRo

Note: Apologies for trajectory 2, the phone camera crashed as soon as the waypoint navigation started. The provided video is from testing.

VII. Important Lessons Learned

A. Simple PID Position is Insufficient

Our first attempt at writing a stable controller was to use basic PID controllers for each of the x, y, z components of r_{cmd}. Namely, given an x error measurement, $e_x = r_x - r_{cmd,x}$, we have,

$$u_x = K_{p,x}e_x + K_{i,x}(e\Delta t) + K_{d,x}\frac{\Delta e}{\Delta t}.$$

This type of controller worked extremely well for vertical movement. But for any kind of translation in the $x-y$-plane, there was basically no gains that we could find that would allow for any kind of reasonable settling time. In most cases, the result was unstable, and the drone would oscillate continuously around r_{cmd}.

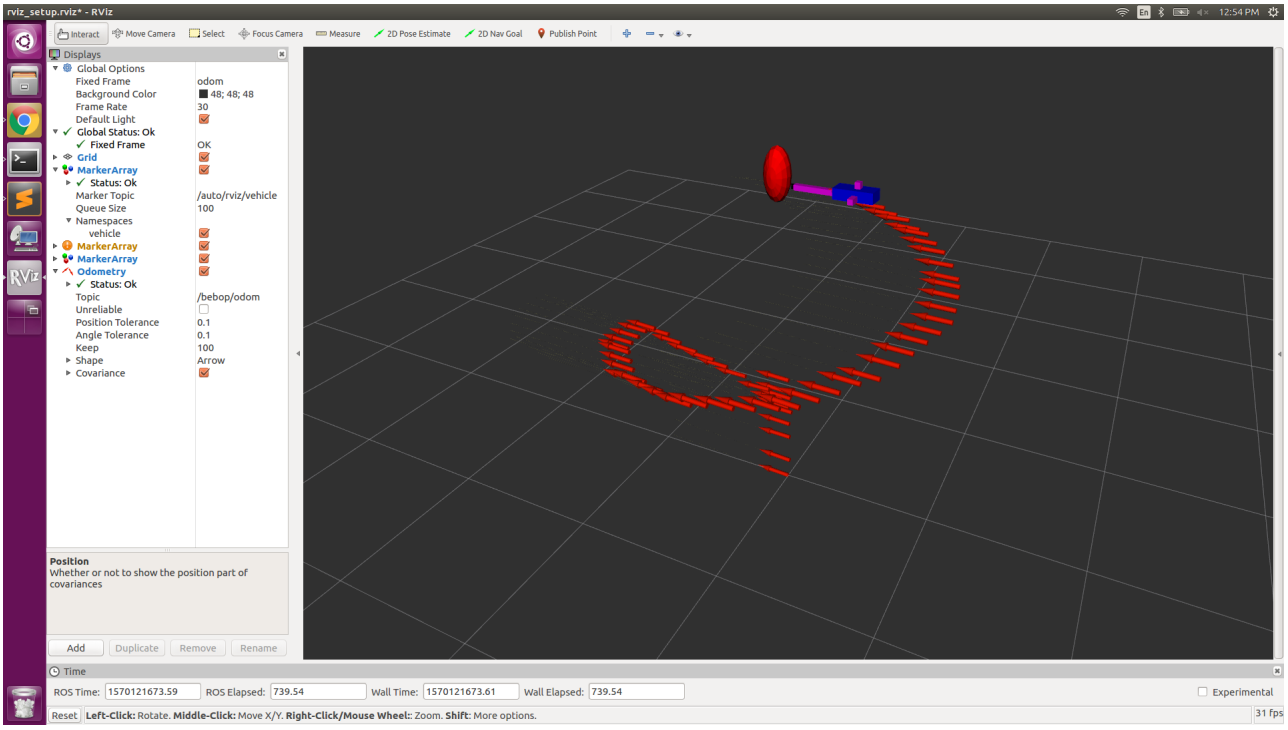

Fig. 3. Rviz display output

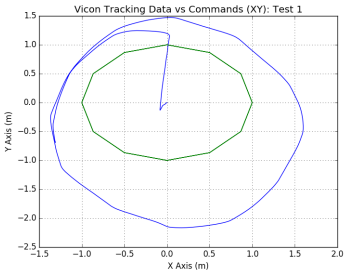

Fig. 4. Trajectory 1 (XY): Vicon vs Commanded Trajectory

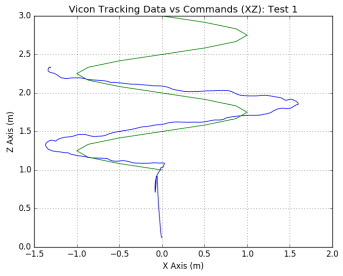

Fig. 6. Trajectory 1 (XZ): Vicon vs Commanded Trajectory

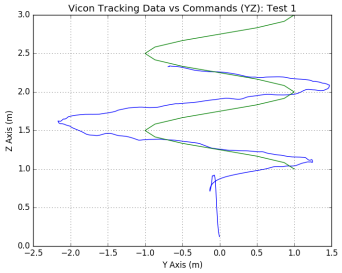

Fig. 5. Trajectory 1 (YZ): Vicon vs Commanded Trajectory

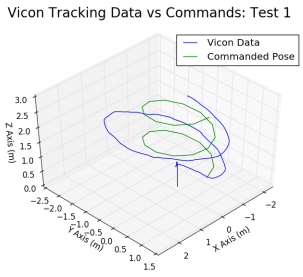

Fig. 7. Trajectory 1 (3D): Vicon vs Commanded Trajectory

B. Simulation is not Reality

This project taught us a great deal about the utility of an available simulation environment. The ability to design against this simulation (which closely mimicked the node structure that existed for the open-source bebop_autonomy ROS package) allowed us much more freedom to experiment with design choices and effectively gave us more development time by eliminating the overhead required to setup and run the test frameworks on hardware. Indeed, once we began hardware testing there were few changes that needed to be made to the

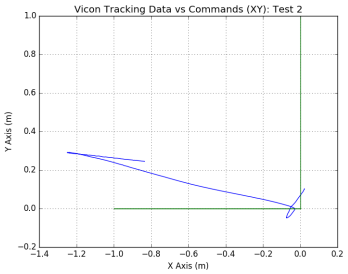

Fig. 8. Trajectory 2 (XY): Vicon vs Commanded Trajectory

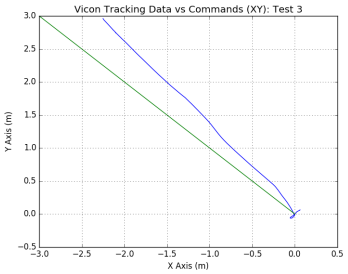

Fig. 12. Trajectory 3 (XY): Vicon vs Commanded Trajectory

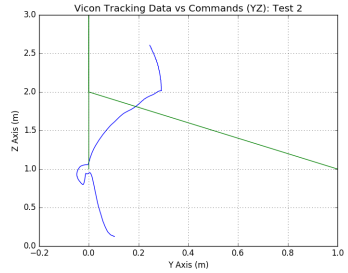

Fig. 9. Trajectory 2 (YZ): Vicon vs Commanded Trajectory

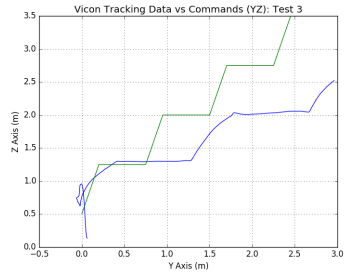

Fig. 13. Trajectory 3 (YZ): Vicon vs Commanded Trajectory

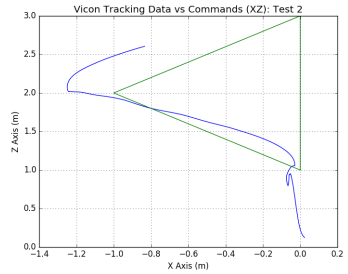

Fig. 10. Trajectory 2 (XZ): Vicon vs Commanded Trajectory

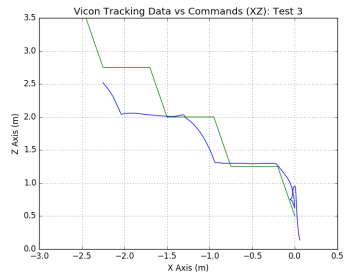

Fig. 14. Trajectory 3 (XZ): Vicon vs Commanded Trajectory

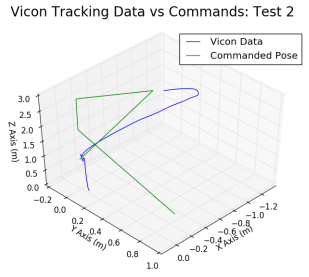

Fig. 11. Trajectory 2 (3D): Vicon vs Commanded Trajectory

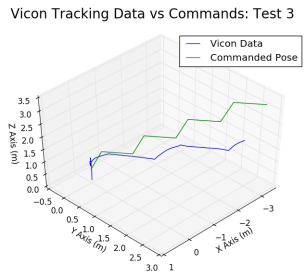

Fig. 15. Trajectory 3 (3D): Vicon vs Commanded Trajectory

core software architecture and the solution functioned mostly as designed, increasing the amount of time available to us for running through the test trajectories and tuning the gain values.

However, one thing that came back to haunt us was that the simulated Ar Drone published its odometry at a much higher rate than the Bebop (approximate 25 Hz vs 5 Hz). As such, our initial PID position controller performed perfectly in simulation, but when applying it to the Bebop the slower odometry updates prevented the basic PID from the previous subsection from converging.

C. ROS Configuration is Cumbersome

We ran into innumerable issues in getting our ROS catkin workspace functional. For example, in order to use Dynamic Reconfigure we had to add the cfg, node prefixes, and correct Catkin commands all in exactly the right locations and spec-

ifications in order to get it to run. Even when we finally did get it operational, we found that we were unable to set default values on a per-PID basis, so then had to go back and figure out how to load default values into the ROS param server using a .yaml file. This was just one example of many, but the important takeaway here is that we should allot ample time for debugging ROS configuration issues, since for this project we probably spent more time on that than actually implemented control code.

D. The Floor Matters

The majority of our testing was done in an area with a solid color flooring with taped markers. In general, we found very high correlation between the down-facing camera's odometry readings and the canned trajectories. However, the actual carpeting in the testing facility proved to make those readings significantly more noisy. Particularly near the edges of the mat where the patterns dropped away. It is evident in the Helix trajectory that the odometer was drifting significantly towards the edges of the test space.

E. Manual Control

We were able to implement manual control over the drone through the pygame libraries and an Xbox controller. This allowed us to take control over the drone at any moment and save it potentially crashing. There were many times during testing in which manually flying the drone back to safety prevented a crash. Some situations where sending a land command rather then flying the drone to safety, would have resulted in a crash. From the controller we could manually move the drone, takeoff, land, and switch between manual and autonomous states.

VIII. Conclusion

Ultimately we were able to get the Bebop to follow the specified trajectories with reasonably high fidelity. In that sense, we consider ourselves to be successful. Reflecting on our controllers, code, and waypoint navigator however, we see many opportunities for improvement.

Most notably is simply how sluggish the Bebop is at moving from target to target. Given time constraints we were not able to implement any kind of B-spline or trajectory smoother for moving through points. So currently we basically hit 0 velocity each time we hit a waypoint. We would like to implement a different kind of controller/trajectory parameterization mechanism that allows for smooth movements through trajectory points.

Looking forward to the next task however, we note just how small the opening in the gates are. In order to ensure a clean pass through the gate, we will have to ensure that whatever controller we use can follow a trajectory with basically 0 overshoot. Ensuring that our system is critically damped will be necessary for successfully accomplishing the next task, so perhaps it will be better to focus our efforts there.

Acknowledgment

The authors would like to thank the professors for this course, Nitin J. Sanket and Chahat Deep Singh, as well as Dr. Inderjit Chopra.

References

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[2] H. Huang and u. Sturm, "Tum simulator." [Online]. Available: http://wiki.ros.org/tum_simulator

[3] "bebop_autonomy - ros driver for parrot bebop drone." [Online]. Available: https://bebop-autonomy.readthedocs.io/en/latest/index.html