# Micro-UAV Self-Characterization

Claire Beery, Greg Coleman, Sarah Walters

April 16, 2015

## 1 Goal/Objective Statements

The purpose of this project is to explore self-characterization of unmanned aerial vehicles (UAVs) by executing and characterizing simple maneuvers in order to enable a micro-UAV to perform more complicated maneuvers. Our UAV will operate in an xy-coordinate system which lies in a plane parallel to the ground. The UAV should be able to execute simple maneuvers in the xy plane (e.g. linear motions, measured turns, and circles) in order to characterize its flight, then it should be able to combine movements in order to perform more complex maneuvers within the plane. The complex maneuvers will include ellipses, letters, and user-sketched trajectories.

It is currently possible for UAVs to learn single maneuvers by executing them repeatedly and tuning performed trajectory in comparison to desired trajectory. However, single learned maneuvers provide extremely limited capability; a UAV would be far more flexible if it were able to characterize its motion a small number of simple maneuvers and then perform a wide variety of more complex maneuvers autonomously.

## 2 Definitions

Below is a list of terms we will be using to discuss the project.

- **real model** - stores parameters and position information to describe the real quadrotor
- **real controller** - modifies the real model based on the simulated model
- **view** - the GUI which displays the real quadrotor and its environment
  - **world** - how the environment appears in the GUI
  - **player** - how the quadrotor appears in the Gazebo GUI
- **simulated model** - stores parameters and position information to describe the simulated quadrotor; communicates with the real model
  - **learned parameters** - a set of initially unknown parameters which are tuned to minimize error using the training maneuvers; used to perform testing maneuvers
  - **real parameters** - a reference set of parameters which are pulled from the Gazebo file which defines the player. Used ONLY to convert from rotor speeds to angular/linear velocity - the simulated controller produces rotor speeds, but the real controller requires velocities as inputs
- **simulated controller** - the code which modifies the simulated model given a desired trajectory
- **maneuver** - a motion the player makes (in our case, in a plane parallel to the ground plane of the stage)
  - **training maneuver** - a simple motion used to tune the learned parameters

  - **testing maneuver** - a maneuver more complex than the training maneuvers, used to test the learned parameters (for instance, ellipses, letters, and user-sketched trajectories.)

- **trajectory** - the position and velocity of the player over time
  - **desired trajectory** - the state the simulated model is commanded to follow, via the simulated controller
  - **performed trajectory** - the state the real model actually follows

# 3 Materials

This project operates entirely within a simulation. A simulated environment allows us to develop a working model in isolation from the real-world variance that comes with actual hardware. We simulate the player and its environment using Gazebo (gazebosim.org), an open-source robotics simulator which includes a physics engine (which we use simulate the model's interactions with the stage) and a sensor library (which we use to simulate the controller's sensing capabilities). Gazebo has a built-in ROS interface, which will enable us to write the model in ROS and attach it to the simulated player via the simulated controller.

We are simulating the player using the Hector Quadrotor stack (hector_quadrotor), a ROS package which contains sub-packages related to modeling, control and simulation of quadrotor UAV systems. The stack includes the necessary launch files and dependency information for simulation of the quadrotor model in Gazebo, and it creates relevant ROS topics which allow us to subscribe to sensor information from the quadrotor and to publish commands.

We're running Gazebo and ROS on our student laptops (Dell Latitude E6430s).

# 4 Methods

## 4.1 System Architecture

Our model software operates in two stages (Figure 1): training and testing.

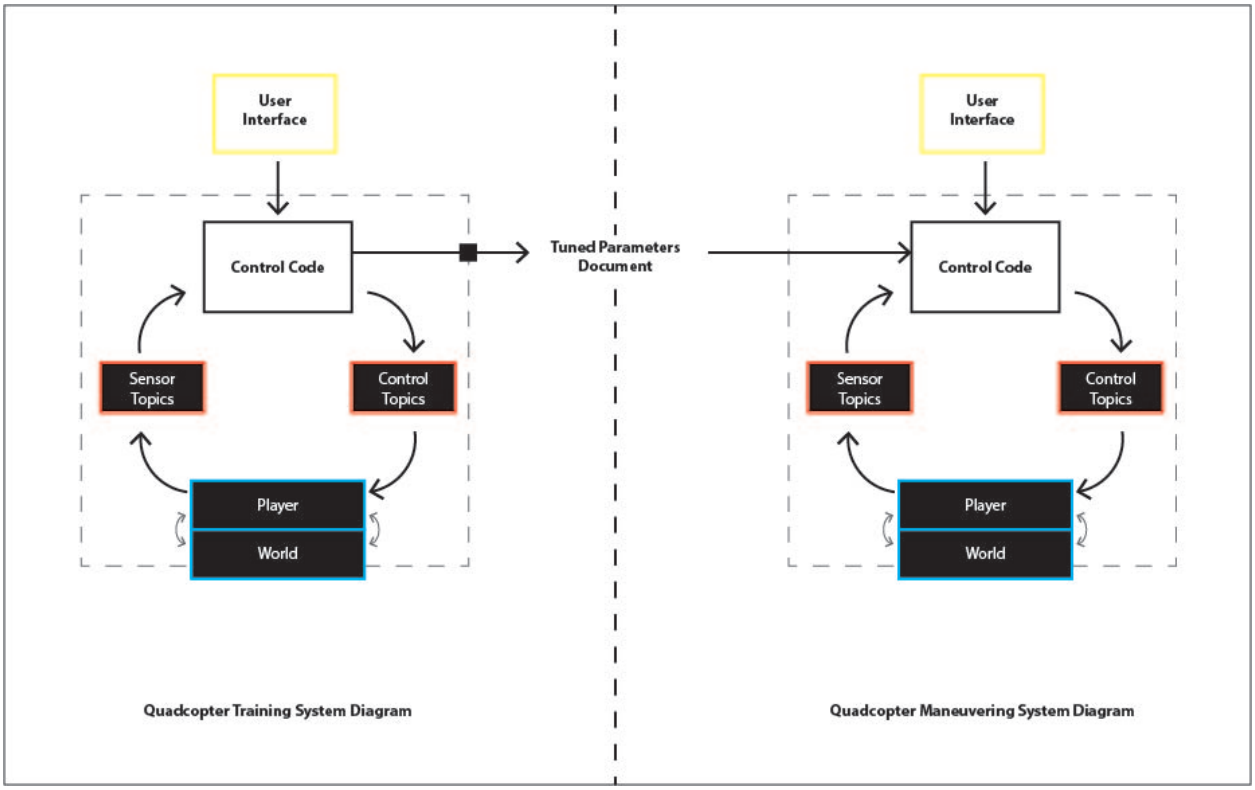


Figure 1: System Flow

The first stage, training, tunes the learned parameters that describe the simulated model. The user gives the simulated controller a simple start command via a user interface. Upon receiving the start command, the quadrotor moves through a series of training maneuvers. Physical quadrotors are controlled via four parameters - the angular speed of each of the four rotors. By contrast, the Hector stack allows for control of the player via linear velocity and the derivative of yaw (four values total - linear velocity in three directions and angular velocity in the z-direction). As a result, the simulated controller computes rotor speeds as if it were controlling a physical quadrotor; it then uses the real parameters to simulate the motion of the player, thereby converting the rotor speeds into publishable velocities. Once the system has minimized the error response of the learned parameters sufficiently, the parameters are saved and the second stage begins.

The second stage, testing, controls the quadrotor through a series of testing maneuvers. The system takes trajectory input from the user via a pygame GUI. The simulated controller uses the learned parameters to move the quadrotor through the desired trajectory, and the system collects information about the real trajectory for post-maneuver comparison.

### 4.2 Trajectory

We take trajectory input from the user via a click-and-drag pygame GUI. We define trajectories as set of keyframes evenly-spaced in time, each keyframe describing pose and location. Upon receiving user input as discrete points, we use a Gaussian smoothing algorithm (from NumPy) to get rid of sharp discontinuities and curves. We then compute the arc length of the curve to drop a prespecified number of evenly spaced keyframes, then use a prespecified constant linear velocity for the quadrotor to associate each keyframe with a time. To associate an orientation with each keyframe, we use the previous, current, and following points to form an angle (with the current point at the vertex). We then use the perpendicular of the angular bisector to set the orientation of the quadrotor. The direction of the quadrotor always is set toward the next point in time.

## 5 Result/Goal Verification

To verify the the success of the self-characterization, we will compare the desired trajectory to the performed trajectory. Our system will collect absolute location, orientation, and velocity data from the simulated model and compare to the desired trajectory. We plan to compare by summing absolute errors the from desired trajectory over all of the points for which the controller collects information, then dividing the position absolute error by the longest dimension of the trajectory and the orientation absolute error by 90 degrees to standardize. We are placing a success threshold of 20% on each error statistic.