



FORMATION CONTROL OF CRAZYFLIES

Bryce MACK

Chris NOE

Trevor RICE

May 2018

Advisors: Dr. Ahn & Dr. Wang

Abstract

This project studies the implementation of a distributed formation control algorithm using several nano quadcopters called Crazyflies. The control system contains two modules. The high-level control module is applied to generate the desired way-points for individual Crazyflies, and the low-level control module is used for stabilizing control of Crazyflies to follow trajectories formed by the way-points. The overall control algorithms were implemented with the aid of a python-based library developed by Bitcraze. Specifically, localization of Crazyflies was handled by an anchor-tag system known as the Loco Positioning System (LPS). A remote computer was used to run the high-level control laws and to transmit control signals to Crazyflies through a Crazyradio PA dongle. The radio can send multiple radio control signals to several Crazyflies at once, which allows us to update each Crazyflie's position individually. The on-board microcontroller of the Crazyflie handles the low-level PD and PID control algorithms for quick response and stabilization. Experiments including hovering control, trajectory following, and formation control were successfully conducted in an indoor environment.

Contents

1	Introduction	4
2	Problem Statement	4
3	Functional Requirements	6
3.1	High Level System Design	6
4	Modeling and Control of Crazyflie	7
4.1	Quadrotor Dynamics	7
4.2	Hovering Control Design	8
4.3	Formation Control Design	10
4.4	Simulation and Preliminary Results	10
5	Localization of Crazyflie using Xbox 360 Kinect	12
5.1	Functionality of Kinect Sensor	12
5.2	The Basic Idea Using Kinect for Localization	12
5.3	Kinect Position Output	13
5.4	How to Setup the Kinect and Crazyflie	13
5.4.1	Updating the Crazyflie Firmware	15
6	LOCO Positioning System	15
6.1	Setting Up the LOCO System[10]	16
6.2	Changing the LPS Mode	17
6.3	Two-Way Ranging Mode	17
6.4	TDoA Mode	17

6.5	Developing with the Bitcraze Python API Library	17
6.5.1	Installing the Python Library	18
6.5.2	Using the Python Library	18
6.5.3	Results of using the Python Library	19
7	Parts List and Work Plan	23
7.1	Parts List	23
7.2	Schedule	23
7.3	Work Division	24
8	Discussion	25
8.1	Future Work	25
9	Conclusions	25
	References	26
A		
	Example Python Code	27
A.1	Square Flight	27
A.2	4 Drone Pyramid Example	30
A.3	5 Drone Example	38

1 Introduction

Recent years have seen significant progress in the technological development of unmanned aerial vehicles (UAVs). Particularly, the usage of multiple UAVs has received a lot of attention in both industry and military. For instance, a group of small UAVs could be used in a search and rescue mission in a harmful region for humans.

In general, the individual UAV in a group has its own sensors and actuators. To fully coordinate the motion of multiple UAVs for certain common tasks, the fundamental questions become how to deal with sensing/communication among individual UAVs and how to design simple, yet efficient local control strategies for each UAV. There exist extensive distributed cooperative control algorithms for general linear dynamical systems. Some results may be applicable to the distributed control of multiple UAVs. However, few results are available in terms of experimental implementation of algorithms in real UAVs especially in the presence of various uncertainties including system dynamics uncertainty, and sensing and communication uncertainties. The core objective of this project is to design practically implementable distributed control algorithms for UAVs and to implement them using an agile nano quadcopter called Crazyflie.

2 Problem Statement

In this project, we will be using the Crazyflie 2.0 produced by Bitcraze (Fig. 1). The distributed control design will be based on the system model abstracted from crazyflie. The sensing/communication and control strategies will be implemented and tested using a number of crazyflies.

We chose to use this nano quadcopter over the others on the market because of its high quality components. This copter is also open source, so we will be able to program and control it using our own software. This copter is small for us to safely test indoors. It has 5-10 minutes of flight time with less than an hour charge time. Bitcraze also produces its own Crazyradio which can be used to control multiple crazyflies at once. This was a big part of our overall project and with the crazyflies we would not need to do much hardware development. All the hardware integration has already been taken care of by Bitcraze. We will be developing with the crazyflie Python API library. This library was created by Bitcraze and allows us to focus on the higher level control of the system. Figure 1 shows an image of a Crazyflie 2.0. Its specifications are given below.

- Weighs 27 g
- Size (LxWxH): 92x92x29mm (motor-to-motor and including motor mount feet)



Figure 1: Crazyflie 2.0

- 20 dBm radio amplifier tested to more than 1 km range LOS with Crazyradio PA
- STM32F405 main application MCU (Cortex-M4, 168MHz, 192kb SRAM, 1Mb flash)
- nRF51822 radio and power management MCU (Cortex-M0, 32Mhz, 16kb SRAM, 128kb flash)
- IMU: 3-axis gyro, accelerometer, and magnetometer
- Max recommended payload weight: 15 g

There are several main research problems to be addressed in this project:

1. Identify an approximate mathematical model to describe the dynamics of the crazyflie.
2. Design a software simulation platform (Matlab/Simulink or ROS-based) which can be used to simulate various control designs.
3. Design control algorithms for configuration stabilizing control and trajectory tracking control of crazyflies.
4. Develop an algorithm to estimate the yaw, roll, and pitch angles of crazyflies using its onboard IMU.
5. Develop an algorithm to estimate (x,y,z) coordinates of crazyflies using images captured by Kinect vision sensors.
6. Develop a distributed control strategy for formation motion of three crazyflies.
7. Implement and test the robustness of control algorithms.

3 Functional Requirements

The objective of this project was to design and implement distributed control algorithms for crazyflies. The overall functional requirement of the project was focused on completing two primary tasks. The primary tasks were making the crazyflie hover, and then implementing a distributed control algorithm for a swarm of crazyflies.

The crazyflies were controlled using a Crazyradio PA. This device is a USB radio dongle that can be plugged into any computer USB port. Our code base is written in python and utilizes the Bitcraze python library. Using the library's API we could communicate directly with the crazyflies and control them autonomously through a python script. Furthermore, with the Bitcraze library, we interfaced with the Loco Positioning System (LPS), which was able to pinpoint each crazyflies location within our workspace.

In addition to reading the LPS localization data, the laptop continuously sends position setpoints to the crazyflie. In the case of multiple crazyflies, they are all controlled with one Crazyradio PA.

3.1 High Level System Design

The High Level System Diagram is shown in Figure 2. The Loco Positioning System communicates with the crazyflies via 2.4 Ghz pings. This allows the base station to know each crazyflies' position in 3D space. The base station takes the localization information from the crazyflies' logs. The localization data is currently only used for creating plots so that we can see flight stability over time. Otherwise, the base station is able to send position updates to the crazyflies as setpoints via the Crazyradio.

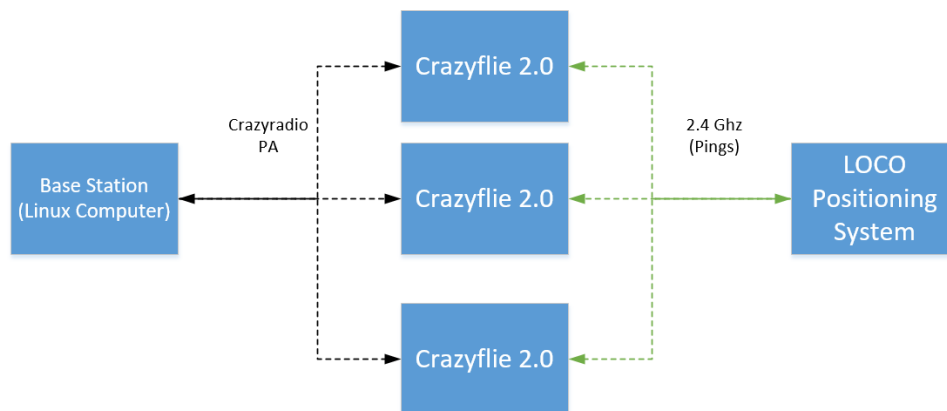


Figure 2: High Level Diagram of the System

4 Modeling and Control of Crazyflie

The quadrotor model included in the toolbox[3] is extremely detailed with how it is set up. There are dozens of parameters that are specific to the quadrotor that Corke and his research team based their model on.

We read multiple research documents on system identification for the crazyflie. However, there were still parameters that were not covered in the research. The researchers focused on identifying thrust coefficients and rotor speeds. So, instead of doing our own system identification for the crazyflie, we chose to leave the quadrotor model as is.

The model shown in Figure 3 is used to direct the quadrotor to a specific point. The model contains 4 PD controllers: velocity, height, yaw and attitude.

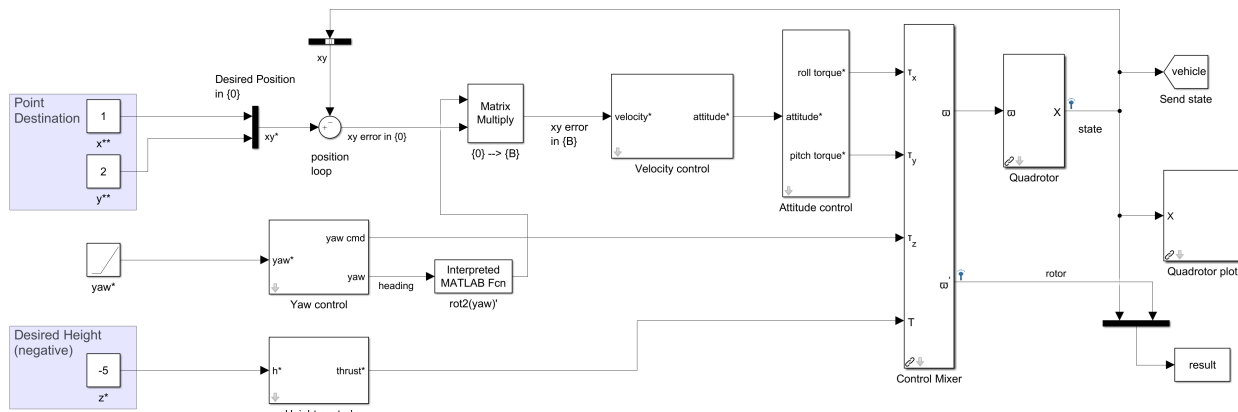


Figure 3: Simulink Quadrotor Model from Robotics, Vision and Control Toolbox

4.1 Quadrotor Dynamics

This quadrotor model requires a transformation from the Inertial Frame to the crazyflie Body Frame. The body frame that Corke used has the standard x-y axes but the positive z-axis is in the downward direction. This way, gravity corresponds to the positive z direction. While using this frame of reference for the crazyflie, we must remember that all values entered for z will be negative, i.e., anything above the ground has a negative z value.

While researching quadrotors, [8] presented a simple, rigid-body model of a quadrotor, assuming low speeds. The equations defining the model are given by

$$\begin{aligned}
\ddot{x} &= (\cos \phi \sin \theta \cos \psi + \sin \phi \sin \psi) \frac{U_1}{m} \\
\ddot{\phi} &= \dot{\theta} \dot{\psi} \left(\frac{I_y - I_z}{I_x} \right) - \frac{J_R}{I_x} \dot{\theta} \Omega_R + \frac{L}{I_x} U_2 \\
\ddot{y} &= (\cos \phi \sin \theta \sin \psi + \sin \phi \cos \psi) \frac{U_1}{m} \\
\ddot{\theta} &= \dot{\phi} \dot{\psi} \left(\frac{I_z - I_x}{I_y} \right) - \frac{J_R}{I_y} \dot{\phi} \Omega_R + \frac{L}{I_y} U_3 \\
\ddot{z} &= -g + (\cos \phi \cos \theta) \frac{U_1}{m} \\
\ddot{\psi} &= \dot{\phi} \dot{\theta} \left(\frac{I_x - I_y}{I_z} \right) + \frac{1}{I_z} U_4
\end{aligned} \tag{1}$$

As they say in [8], the desired position and orientation (pose) of a quadrotor is hovering. Therefore, we can reduce Eqn. 1 using small angle approximations. Let $U_1 = mg + \Delta U_1$. Under these approximations the equations become

$$\begin{aligned}
\dot{x} &= g\theta \\
\ddot{\phi} &= \frac{L}{I_x} U_2 \\
\ddot{y} &= -g\phi \\
\ddot{\theta} &= \frac{L}{I_y} U_3 \\
\ddot{z} &= \frac{\Delta U_1}{m} \\
\ddot{\psi} &= \frac{1}{I_z} U_4
\end{aligned} \tag{2}$$

The four inputs U_1, U_2, U_3 and U_4 represent the inputs for roll, pitch, yaw and thrust. These are the equations that Corke's model is based on. The 4 PD controllers are used to continuously update the input commands during simulation.

4.2 Hovering Control Design

Our first task with the crazyflie is to have it hover at one point in 3D space. By following the experiments performed in [2], we will have the crazyflie go from the ground to a specific point a few feet away but above the ground. The crazyflie will hover stably in that position for up to 30 seconds. In addition to just hovering, we would be able to introduce disturbances to the bot and it would be able to overcome any disturbances to maintain stability.

In order for us to be able to test controlling a quadcopter, we needed a model that we could run to simulate inputs and disturbances. Instead of starting from scratch, we were able to use a model created by Peter

Corke. We installed his MATLAB toolbox that has been developed by Corke and multiple graduate research students over the past two decades. Corke has also written a textbook [3] to explain how to use various parts of his toolbox.

Originally, the height controller used PD control and had a feed-forward thrust term that was used to compensate for the thrust needed to keep the quadrotor airborne.

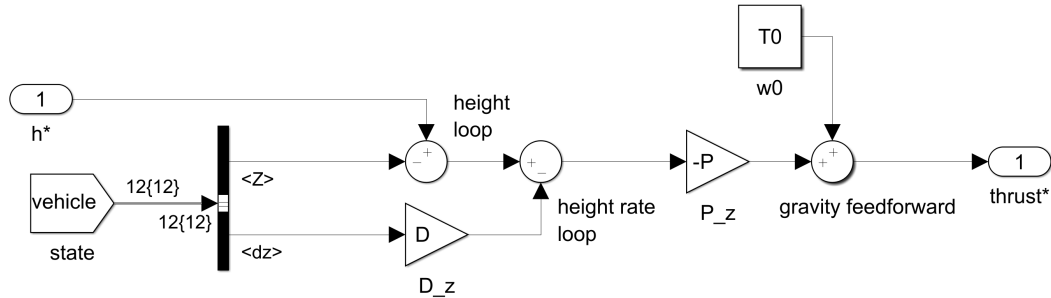


Figure 4: Original Height Controller

Corke used this because his quadrotor weighed 4 kg and so it required a large amount of thrust to stay in the air. By including a constant feed-forward term, Corke eliminated the need for integral control. This allowed his quadrotor model to stabilize much faster since an integral controller reduces settling time.

Our crazyflie only weighs 27 g and the thrust needed to keep it airborne is negligible. Dr. Wang advised us to introduce integral control into the height controller. The integral controller can be seen in Figure 5. This replaced the T_0 term shown in Figure 4.

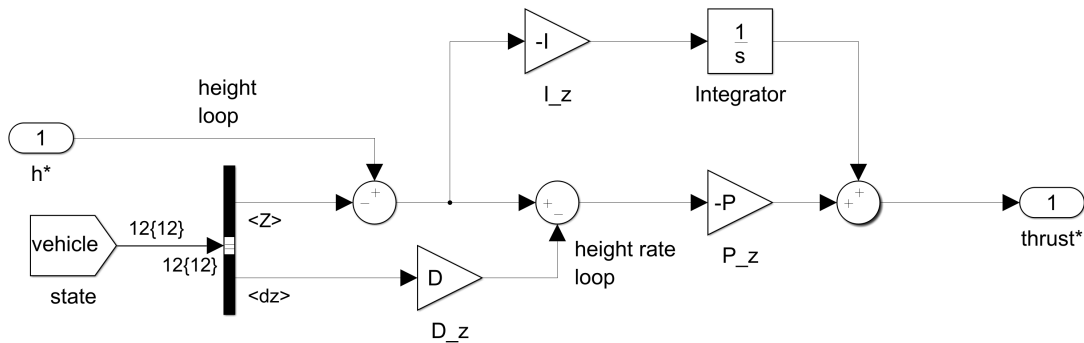


Figure 5: PID Height Controller

4.3 Formation Control Design

The baseline design for formation control will be based on the linearized quadrotor model, which can be represented as a double integrator given below

$$\ddot{x}_i = u_i, \quad i = 1, \dots, n$$

where x_i denotes the state variable and u_i is the control input. To illustrate our idea of design, in what follows, we simply assume $x \in \mathfrak{R}$ and $u \in \mathfrak{R}$. Further design Details will be presented in the final report.

The formation control is of the form

$$u_i = \sum_{j=1}^n a_{ij}(x_j - x_i) + \sum_{j=1}^n a_{ij}(\dot{x}_j - \dot{x}_i) \quad (3)$$

4.4 Simulation and Preliminary Results

After making the replacement in Figure 5, we needed to modify the control values to optimize overshoot and settling time. We modified the subsystem mask to include inputs for kd , kp and ki . These are the proportional, derivative and integral gains, respectively. Dr. Wang recommended that we use the control design of $kp * kd > ki$ with all control gains > 0 . For the initial test, we chose to use

$$\begin{aligned} kp &= 5 \\ kd &= 5 \\ ki &= 10 \end{aligned} \quad (4)$$

For these simulations, we are focusing on the response of Z, shown in Fig. 6. By running the MATLAB command `stepinfo`, we got the step response information. With these control values, the overshoot was 70.2% and a settling time $> 30s$. We definitely want to reduce these and have $< 20\%$ overshoot and settling time $< 10s$.

After a few more trials, we learned how the controller responds to the control gains. Reducing kd decreased the overshoot. Increasing kp decreased the settling time. With this in mind we chose to use the following control values

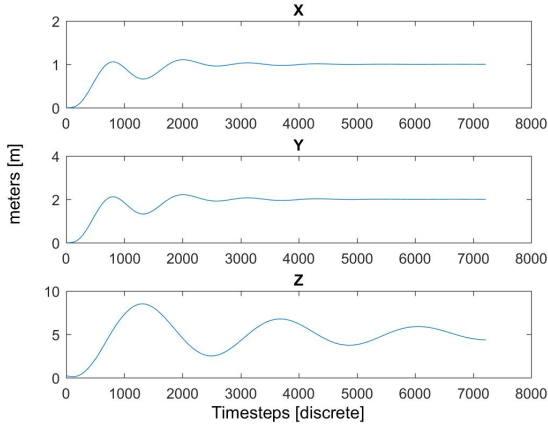


Figure 6: X,Y,Z Plots using control values in Eqn. 4

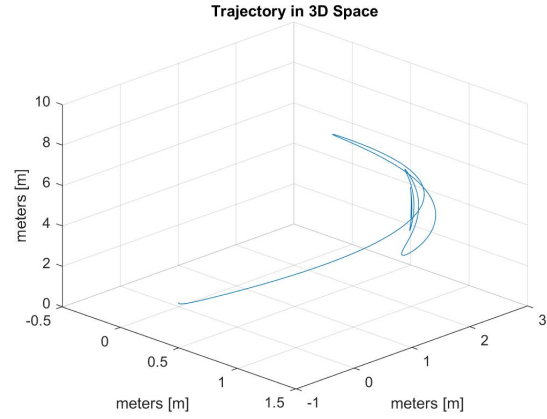


Figure 7: 3D Plot of Fig. 6

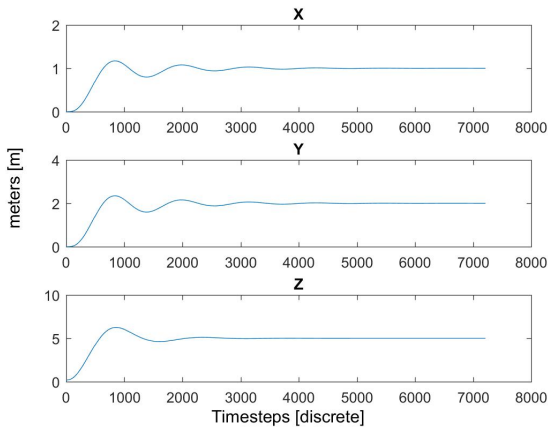


Figure 8: X,Y,Z Plots using control values in Eqn. 5

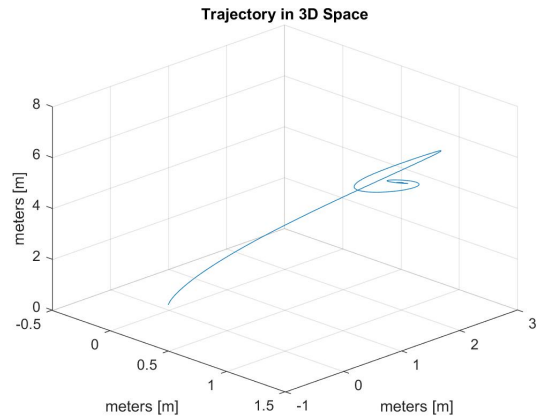


Figure 9: 3D Plot of Fig. 8

$$\begin{aligned}
 kp &= 10 \\
 kd &= 1 \\
 ki &= 8
 \end{aligned}
 \tag{5}$$

These changes to the control values had a huge effect on the height stabilization. The overshoot was reduced 25% and the settling time was reduced to 10.3s.

Following the modifications to the Height Controller, we will be experimenting with the other controllers to understand how to modify the response. From Figure 8, we would like the X-Y stabilization to be as quick as the Z stabilization.

We are executing these simulations to understand the controls of the model. Eventually, we will switch to developing the model in code for ROS. All of the control values will be different from what we use in Simulink.



Figure 10: Xbox 360 Kinect Module

However, we will know how to tune the control values to work for the crazyflie.

5 Localization of Crazyflie using Xbox 360 Kinect

5.1 Functionality of Kinect Sensor

The kinect sensor uses multiple camera sensors to measure position data. The first two camera sensors measure 3D distance data by tracking the light pattern disruptions as shown in [18], by using a known pattern of light. The third camera on the front uses color and depth from focus shown in [18], and although the exact method is undisclosed, it is based on the structured light principle. The kinect uses these sensors together to form a 3D position of an object. For our uses we will get the 3D position of our crazyflie and feed this information to our PD controller.

5.2 The Basic Idea Using Kinect for Localization

The algorithm suggested by the ROS book uses color thresholding to sense where the crazyflie is in the frame of the Kinect. The method reads a specified color from the Kinect image data and has a certain distance that the color has to be constant (within a certain tolerance of color). The Kinect can output this information as a 3 dimensional variable to the controller. The thresholds need to be set according to the specific Kinect model and the size and color of the marker that is set on the crazyflie itself. The method we chose implements a red tape marker and is effective in marking the crazyflie for control.

We have been provided with 3 Xbox 360 Kinect modules as shown in Figure 10. The ROS book references the Kinect V2.0 for its superior sensors and frame rate, which was a concern with our Kinect 360 modules. This was overcome during our implementation of the algorithms mentioned above. There are multiple Git Repositories that reference the Kinect V2.0, and we were able to find one that referenced the Kinect 360.

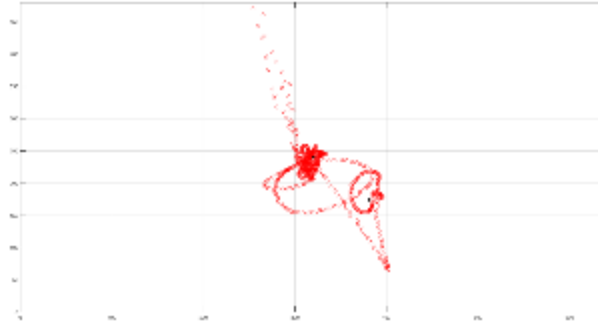


Figure 11: Kinect Flight Output

This repository was incredibly important and helped tremendously with the success of the project.

The Kinect modules were set up on stands keeping them about a meter off the ground, but was also placed on a table. This creates a more perfect square frame that we can use to control the crazyflies. We used one single Kinect module to get the localization for our crazyflie. but this does not allow for yaw control.

5.3 Kinect Position Output

The output of a flight that was controlled by the Kinect is shown in figure 11. The flight data is acquired by marking the X and Y position data every time the controller goes through a control loop. This is then recorded and output to a .csv file that MATLAB can read. This is fed to MATLAB and a plot is created. The graph can be expanded to 3D but it was decided that the 2D plot was more beneficial.

5.4 How to Setup the Kinect and Crazyflie

The first step to setting up the Kinect system that will control the crazyflie is to plug in the kinect power to a 12 volt wall outlet and to the host computer using a usb3.0 port. The Crazyradio PA system must be plugged into another usb3.0 port. The computer must be booted into Ubuntu, and must be in the correct directory. The directory for our project is: `"/Crazyflie/project/kinect-python/bin"` [15] and once in the directory, the user will have to send the specific command of `"/cfkinect"` which will initialize the crazyflie Kinect client. This will output text to the command window telling the user if the program is running successfully, including the channel the Crazyradio and crazyflie are running on, whether the Kinect can find the correct image, and any errors the program can detect. The crazyflie must be turned on, and placed on a flat surface to have the gyros calibrated. The crazyflie itself must have a large amount of red relative to the ambient image seen by the Kinect. The current marker is red tape placed around the center board.

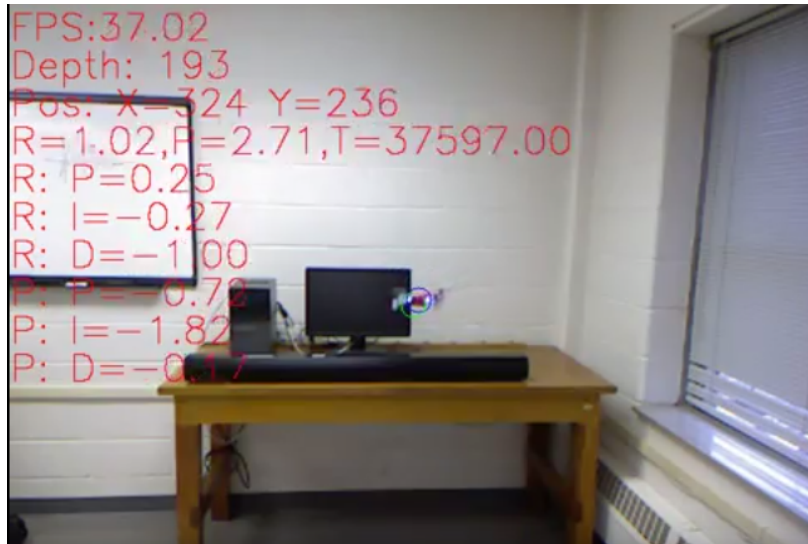


Figure 12: Kinect Flight

Once the command is typed and the crazyflie Kinect Client [15] is booted up, there will be a video stream showing what the Kinect can see. There will be two text readouts in red on the top left of the screen. During this time, there will be no commands sent to the crazyflie and the system will not show the desired set point or the localization of the crazyflie. When the user is ready to fly the crazyflie, they will hold the crazyflie in the center, while the two blue LEDs are facing towards the crazyflie camera. The video screen will draw a blue circle over the crazyflie because the program will find the red marker on the crazyflie. Just as this happens, there will be a green circle in the center of the screen and this is the desired set point for the crazyflie to approach. Once the Kinect has both the set point and the crazyflie found, the user will release the crazyflie (As close to the set point as possible) and the algorithm will start flying the crazyflie.

The desired position of the crazyflie can be changed by the user. The user can click anywhere on the video stream and the set point will be moved to the point that was clicked. The crazyflie will then fly over to the desired set point. The user must be careful to not set the set point too close to the edge of the screen. This could cause the crazyflie to fly outside the view of the Kinect. If this happens, the crazyflie throttle is set to zero and it lands. The Kinect must have a line of sight to the crazyflie or the control commands will not be sent. This is a security feature in case the video stream dies or the crazyflie flies too far away. This way there is less of a chance of injury to the user or observers, as well as to save the crazyflie in case it flies too high and keeps accelerating upwards.

5.4.1 Updating the Crazyflie Firmware

To access to the most recent features, the firmware on the crazyflie needs to be up-to-date. First, you need to download the firmware file from [5]. The quickest way to update the firmware is by using the crazyflie client. First install the client by downloading the install files at [4]. The client has a built-in bootloader to install the firmware. Go the menu "Connect/Bootloader". Power the crazyflie up in bootloader mode by pressing and holding the power button for 3 seconds until the blue light starts to blink. The drone is now in bootloader mode. Click "Initiate Bootloader Cold Boot" and the client should connect to the crazyflie. Select the firmware package (.zip) and click "Program". Once the firmware has been uploaded, click "Restart in Firmware Mode" [9].

The radio and address of the crazyflie can also be changed by connecting to the crazyflie and going into the menu "Connect/Configure 2.0". Changing the address allows the drones to be connected to individually when multiple drones are powered on. For our project, we used addresses formatted like "E7E7E7E7XX" where "XX" stands for the crazyflie number starting with "01". So the first crazyflie address was "E7E7E7E701".

6 LOCO Positioning System

The LOCO Positioning System operates like a small indoor GPS system. Anchors around the room act as position reference markers in the 3D space. Nodes are tracked through the 3D space, these are what are on the top of the crazyflies. The Base Station does not communicate directly with the LOCO system. All wireless communication with the anchors is done through the crazyflie 2.0.

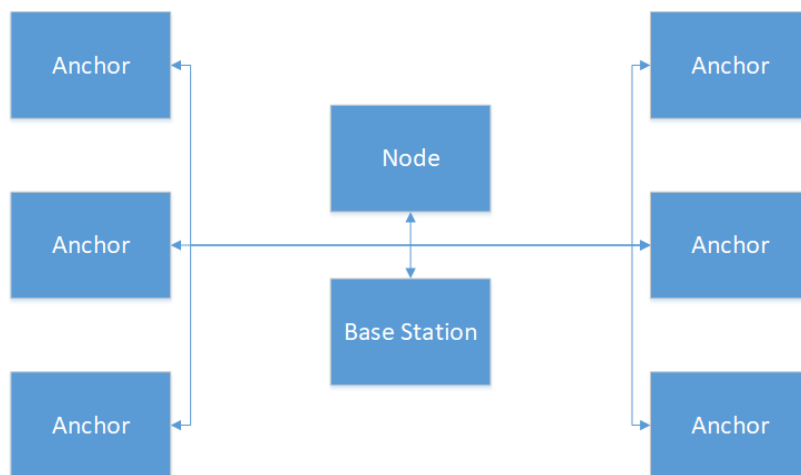


Figure 13: LOCO System Diagram

This system may be used in place of the Kinect Module. The LOCO system is accurate to 10cm which is

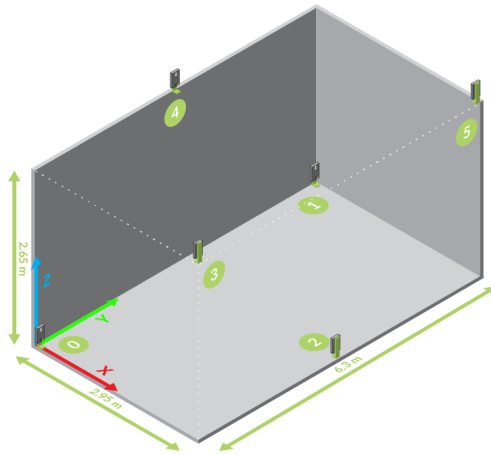


Figure 14: LOCO Setup Diagram

accurate enough for our project as we do not plan on doing extremely tight maneuvers.

The localization data is determined by using the time delay of 2.4 Ghz pings. There are two modes of operation for the system: Two way ranging and TDoA. The position data can be retrieved from the crazyflies using the logging functionality.

6.1 Setting Up the LOCO System[10]

The Loco Positioning system can be set up in virtually any open indoor space. The anchors can be placed on 3D-printed brackets[12] to put space between the anchors and the floor, ceiling, and walls. This will help prevent interference from the signals reflecting off the solid surfaces.

In order to use the latest features of the system, you must have the most up to date firmware for both the LPS and the crazyflies. The latest firmware can be found at [14] and [5]. To install the latest firmware on the crazyflies see the previous section on "Updating the Crazyfie Firmware". To update the firmware on each of the nodes, download the LPS anchor configuration tool from [13]. Press and hold the DFU button on the anchor before plugging it into the computer, this will start the anchor in DFU mode. Windows will not recognize the anchor the first time it is plugged in. Follow the instructions [16] to set up the drivers properly.

The LPS anchors can then be placed around the flight space. It is recommended that the anchors be placed at least 2 meters apart for best coverage. The anchors should also be placed both high and low in the space, preferably in triangular formations. This allows for the most accurate estimations. An example for how to place the anchors in a space can be seen in Figure 14.

6.2 Changing the LPS Mode

The Loco Positioning System can be used in one of two modes: Two-Way Ranging (TWR) and TDoA mode. This mode can be set using the crazyflie client. If you cannot see the LPS tab, go to "View/Tabs/LoCo Positioning system". The following instructions are assuming the anchors are set to TWR mode. To switch the mode, make sure you are connected to a crazyflie with the LOCO Deck installed. In the LPS tab, select the TWR button in Crazyflie Status. Next, click "Switch mode to TDoA". Wait a few seconds for all of the anchors to be highlighted in red instead of green. In Crazyflie Status, click the auto button and see that the TDoA mode turns green. At this point all of the anchors should then turn green again showing that the crazyflie is seeing the anchors in TDoA mode. See [10] for more in depth instructions.

6.3 Two-Way Ranging Mode

In Two-Way ranging mode, both the anchors and the nodes transmit packets. Based on the time-delay between transmission and reception, a location in 3D space is calculated. Because both the crazyflie and the anchors are transmitting in this mode, this mode is more accurate than TDoA mode. However, this mode is not able to be used with multiple crazyflies.

6.4 TDoA Mode

Time-Distance-of-Arrival (TDoA) mode is a new mode that Bitcraze released in January, 2018. In this mode, only the anchors transmit packets. The crazyflies then use the packets from all the anchors to estimate its current position. This mode is less accurate than the TWR mode, but it allows flight of multiple crazyflies[17].

6.5 Developing with the Bitcraze Python API Library

Bitcraze developed a python library to control the crazyflie[7]. Bitcraze developed this library to be used by the Crazyflie Client. This library takes care of the low level control of the crazyflies and the Loco Positioning System. This allowed us to be able to send the crazyflie position setpoints to the crazyflie. Setpoints were sent every 0.2 seconds, even if there was no change in the setpoint. This was to avoid having the watchdog timer activate and shutting down the crazyflie.

We have been able to export positioning data from the crazyflies to a csv file. This allowed us to create graphs showing the flight paths of the drones. The positioning data is retrieved by using the logging functionality of the crazyflies. We used Python 3.5 to run our code.

6.5.1 Installing the Python Library

The simplest way to develop with this library is to run the Bitcraze Virtual Machine[1]. The instructions at [11] recommend using the Oracle VirtualBox environment for the machine. However, the machine will also run on VMware if desired. The virtual machine will have the python library already installed and ready to run any code to fly the drones. It was found that this is the best way to run the python code on a virtual machine. There were issues when running stock Ubuntu on a virtual machine with the library installed. These issues were likely caused by latency issues with USB ports on the virtual machine. The Bitcraze VM managed to avoid these latency issues.

The library can also be installed on a machine with Ubuntu 14.04 natively installed. When installing the library on any machine other than the Bitcraze VM, it is recommended that you do so in a virtual environment to avoid causing conflicts with other python installations. We tested this on a machine with Ubuntu Trusty installed as a dual-boot. Follow the instructions on the python library github[7] to set up a virtual environment and install the library in that environment. The program virtualenv was used on our machine.

Sidenote: When simply updating the python library on your machine, you need to update the code inside the "cflib" directory in your virtual environment library location.

6.5.2 Using the Python Library

The python scripts run a sequence function that loops through and continuously sends setpoint updates every 0.2 seconds to keep the crazyflies active. This function is called by the `swarm.parallel()` function that handles the multi-threading to fly multiple crazyflies. The sequence function runs everything from the takeoff sequence, to the formation flight, and ends with the landing sequence. The formation control is run by calling another function that updates the desired setpoints over time. See Figure 15 for a simple block diagram of the code.

If you are using a virtual python environment, activate the environment. Navigate to the directory that the python script is in. Run the code by typing the following:

```
python3 example.py
```

The crazyflies are identified in the code by their URI. This allows the program to differentiate between the crazyflies. In the future, the URI will be used to read position data from the log on each crazyflie in order to set up a cooperative control.

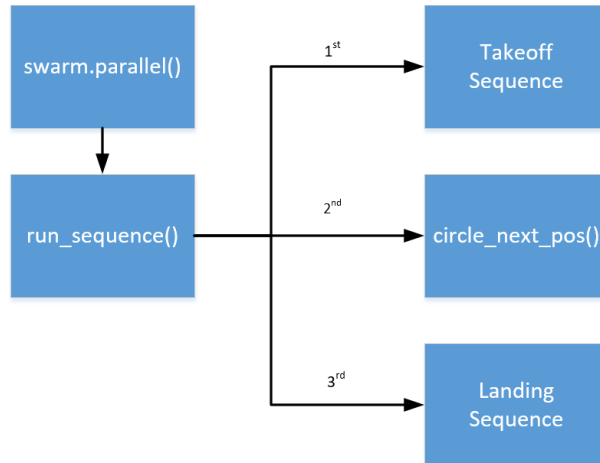


Figure 15: Block Diagram of the Pyramid Flight Code

6.5.3 Results of using the Python Library

Starting off, we were able to get one crazyflie to fly a square and circular pattern. More than 35 flights of the square flight path were run to test the accuracy of the LPS. The plot can be seen in Figure 16. From these flights, we were able to see that the crazyflie remained within 20% of the desired height of 1 m (0.8049 m-1.19 m) during the flight. The maximum overshoot at takeoff was 26.9% (1.269 m). A plot of a circular pattern flight using one crazyflie can be seen in Figure 17.

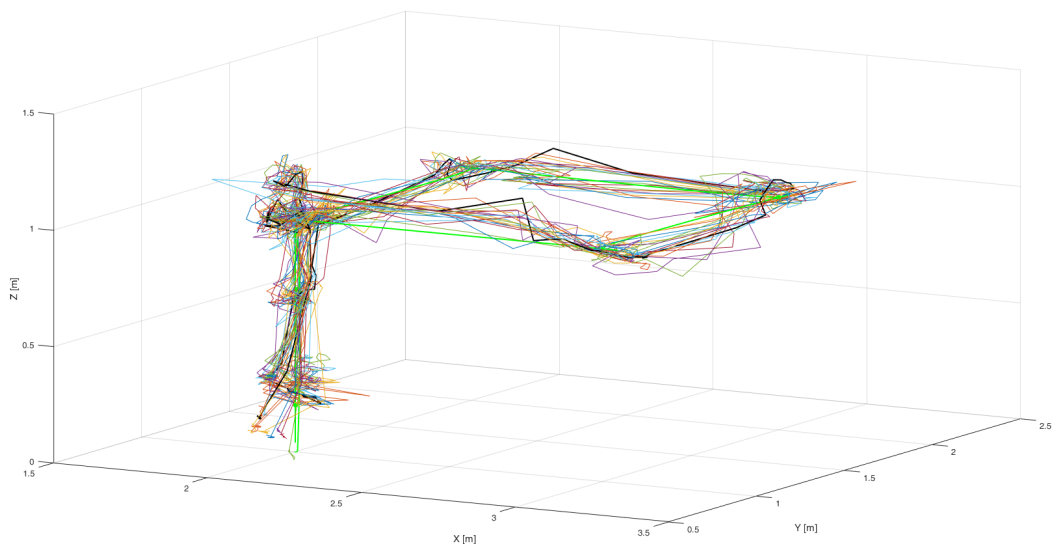


Figure 16: MATLAB Plot of Multiple Square Flights (1 Drone)

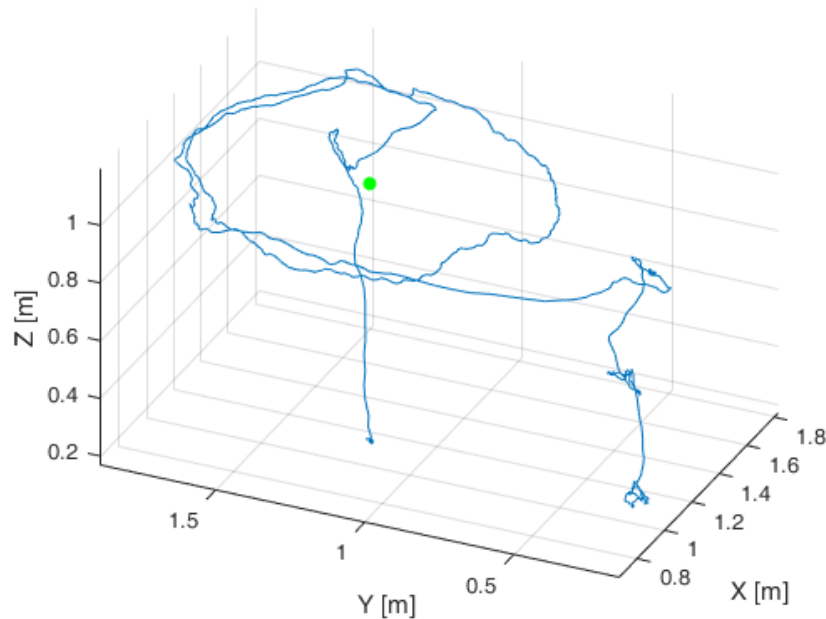


Figure 17: MATLAB Plot of Single Drone Circular Flight

Next we expanded the programs to fly up to 5 crazyflies at the same time. Figure 18 shows 2 crazyflies flying at the same time. This was accomplished by using the multi-threading function of the crazyflie python library. In order to fly multiple crazyflies with the LPS, the mode needed to be changed from Two-Way Ranging mode to TDoA mode.

As more crazyflies were added to the program, more complex patterns were able to be attained. Figure 19 shows a pyramid formation where one crazyflie hovers at a high point and 3 crazyflies fly in a circular formation at a lower formation. The hovering drone was at a height of 1.75 m, while the circling drones were at a height of 1.0 m. The hovering drone was in the center of the 0.7 m radius circle.

Figure 21 shows a formation where 5 crazyflies are flying at the same time. The drones were flying in concentric circles at 2 heights. Two drones were flying at a height of 1.75 m with a radius of 0.7 m. Three drones were flying at a height of 1.0 m with a radius of 0.7 m. It was discovered that the crazyflies are not great at handling turbulence. If one drone is flying directly above another, the rotor wash of the higher drone would cause the lower drone to crash. A MATLAB plot of one of the flights is in Figure 20.

Many test flights can be seen in videos on our website at [6].



Figure 18: 2 Crazyflies Flying at Once

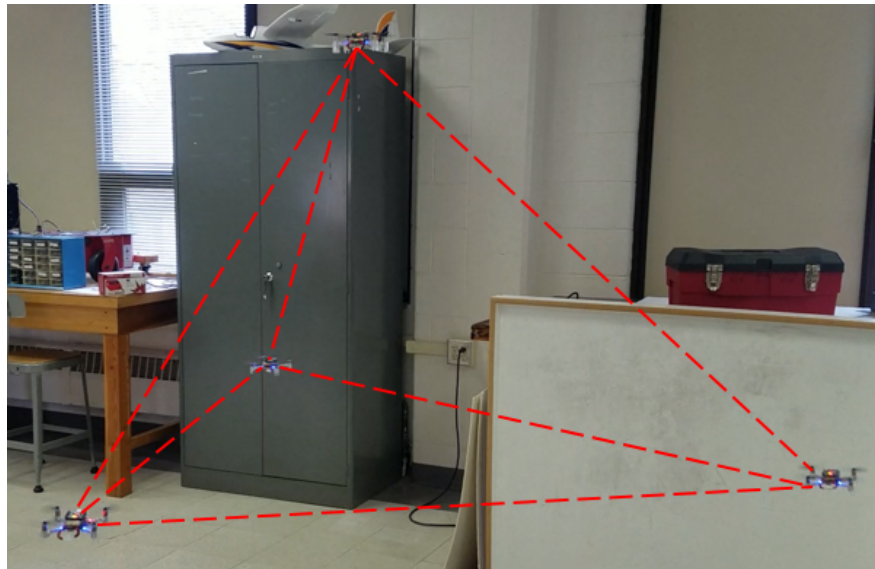


Figure 19: 4 Crazyflies in Pyramid Formation

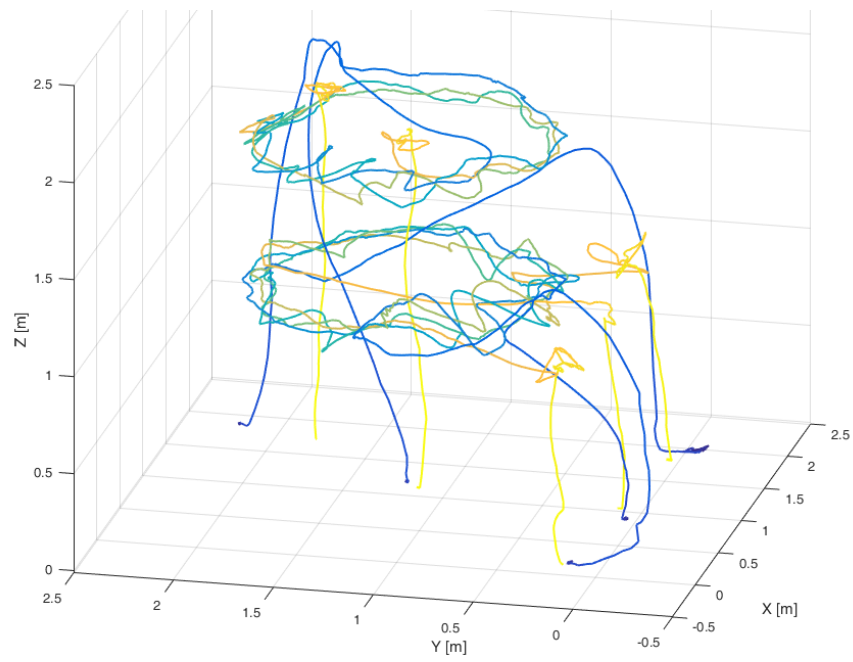


Figure 20: MATLAB Plot of the 5 Crazyflie Formation

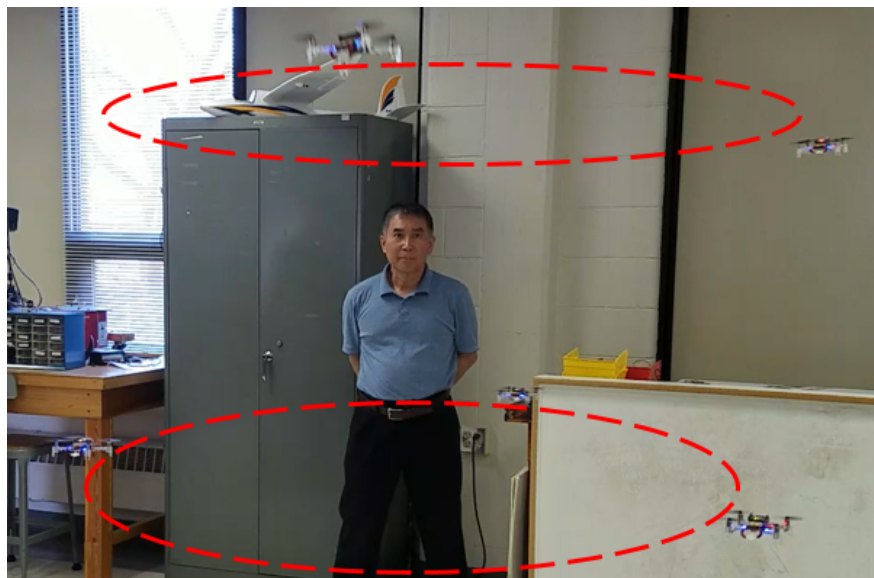


Figure 21: 5 Crazyflies in a Double Circular Formation

7 Parts List and Work Plan

7.1 Parts List

1. Bitcraze Components
 - 6 x Crazyflie 2.0
 - 3 x Crazyradio PA
 - 1 x Z-ranger Deck
2. Xbox Components
 - 3 x Xbox 360 Kinect
 - 3 x Xbox 360 Kinect Stand
 - 3 x Xbox 360 Kinect Power Supply
3. LOCO Positioning System
 - 6 x LOCO Anchors
 - 6 x Anchor Power Supply
 - 6 x 3D Printed Anchor Brackets
 - 5 x Loco Crazyflie Deck
4. Laptop running Ubuntu 14.04 Trusty

7.2 Schedule

Once we were assigned to our project we planned to start by exploring ROS and how to build a library to interface with the crazyflies and the Kinect 360 module. We spent much of the first semester researching ROS used in other projects done for other university research. There were multiple projects done using the crazyflie that we were able to read through to understand how this particular quadcopter operates. During November, we had a working model that we could test and observe to get a feel for how quadcopters react to various thrusts and trajectories. Overall, our research throughout September, October, and November contributed greatly to what we needed to complete development during the spring semester.

In late November, Dr. Wang decided to order the Loco Positioning System for our project. After researching the capabilities of the LPS, we realized that with it we could overcome our localization problem. The LPS arrived in mid-December. After winter break, we setup the workspace with the Loco anchors. Throughout

January and February we explored the Bitcraze python library to understand how to utilize the API and control a crazyflie autonomously. Meanwhile, in early February while exploring the Bitcraze python API, we successfully performed autonomous control using the Kinect module. We would continue to explore any possibilities to integrate the two systems together. Near the end of February, we requested more Loco decks from Bitcraze. This would allow us to expand our autonomous control to handle multiple crazyflies. At the end of March we had completed everything that we set out to accomplish. We took many videos for demonstration purposes and posted them on our website. For the rest of the semester we worked on completing our deliverables and preparing for the presentations.

7.3 Work Division

At the beginning of the project, Bryce Mack worked on developing the Simulink model and running simulations. He worked on tuning the controllers so that the simulations had smooth flights. Along with running simulations, he created simulation videos for comparison to the actual flights of the drones. Bryce also created the project website. The website details the project and contains all deliverables and video demonstrations. Over the course of the project he kept the group on track for deliverables and assisted Chris and Trevor with whatever they needed to get their applications running.

Chris Noe initially researched using ROS to control the crazyflies using the Kinect 360 module. Once the Loco Positioning System was purchased, he moved into researching how to implement crazyflie control using the LPS. He researched the Bitcraze Python Library API, and how to use it to control the crazyflies. He started off by controlling one crazyflie and ran multiple flights of a square pattern to get some initial data on the accuracy of the LPS. He then moved on to expand the system to incorporate multiple crazyflies. He created a number of different python scripts that fly various formations.

Trevor Rice also studied ROS during initial development. He researched how to control the crazyflie through a Kinect 360 camera module. He found and documented various libraries and packages needed to setup the Kinect system. He explored how to use the Kinect 360 cameras to detect position. Trevor set up the space in the lab to be used for physical experimentation. The space needed to be clean so that nothing would interfere with the Kinect's cameras and sensors. A clean area will prevent the possibility of a false detection of the crazyflie. Trevor tested the Kinect system to compare the accuracy of the two applications.

8 Discussion

8.1 Future Work

Future researchers on this project will be able to pick up development using the python library API. The first step would be to create a way to read localization data in real time from each individual crazyflie. This could be accomplished using the URI of the drones to update location data on the base station as it is received. This real time localization updating would allow for the expansive development of higher level cooperative control algorithms.

The future researchers should be able to focus primarily on developing and tuning higher level control algorithms that update position setpoints sent from the base station to the crazyflies. They should not need to focus on the lower level control of the crazyflie as it is already done by the Bitcraze developers in the python library.

9 Conclusions

In this project, we developed a control system designed to synchronously control multiple crazyflies in formation. We were able to successfully get five crazyflies flying in a circular pattern simultaneously.

In addition to formation control, we also researched how to control a crazyflie using a Kinect 360 module. Through our research, we discovered that the Kinect can only track one crazyflie at a time. Even though we will not be using the Kinect in future development, we feel our research was successful at exploring the possibilities as well as the limitations of the Kinect.

References

- [1] *Bitcraze Virtual Machine*. URL: <https://github.com/bitcraze/bitcraze-vm/releases/>.
- [2] Pedro Castillo, Rogelio Lozano, and Alejandro Dzul. “Stabilization of a Mini Rotorcraft with Four Rotors”. In: *IEEE Control Systems* (2005).
- [3] Peter Corke. *Robotics, Vision and Control: Fundamental Algorithms in MATLAB®*. Springer, 2011. ISBN: 9783642201431.
- [4] *Crazyflie Client Installation Files*. URL: <https://github.com/bitcraze/crazyflie-clients-python/releases>.
- [5] *Crazyflie Firmware*. URL: <https://github.com/bitcraze/crazyflie-firmware/releases>.
- [6] *Crazyflie Project Website: Videos*. URL: <http://ee.bradley.edu/projects/proj2018/crazy/videos/videos.html>.
- [7] *Crazyflie Python Library*. URL: <https://github.com/bitcraze/crazyflie-lib-python>.
- [8] Zachary T. Dydek, Anuradha M. Annaswamy, and Eugene Lavretsky. “Adaptive Control of Quadrotor UAVs: A Design Trade Study With Flight Evaluations”. In: *IEEE Transactions on Control Systems Technology* 21.4 (2013).
- [9] *Getting Started with the Crazyflie Client*. URL: <https://wiki.bitcraze.io/doc:crazyflie:client:pycfclient:index#bootloader>.
- [10] *Getting Started with the LOCO Positioning System*. URL: <https://www.bitcraze.io/getting-started-with-the-loco-positioning-system/#intro22>.
- [11] *Installing the Virtual Machine*. URL: <https://www.bitcraze.io/getting-started-with-the-crazyflie-2-0/#inst-virtualmachine>.
- [12] *LPS 3D Bracket Design File*. URL: <https://github.com/bitcraze/bitcraze-mechanics/blob/master/LPS-anchor-stand/anchor-stand.stl>.
- [13] *LPS Anchor Configuration Tool*. URL: <https://github.com/bitcraze/lps-tools/releases>.
- [14] *LPS Firmware*. URL: <https://github.com/bitcraze/lps-node-firmware/releases>.
- [15] *Piloting the Crazyflie with the Kinect*. URL: <https://wiki.bitcraze.io/misc:hacks:kinect>.
- [16] *Setup Bitcraze Windows Drivers*. URL: <https://wiki.bitcraze.io/misc:usbwindows>.
- [17] *Using LPS TDoA mode*. URL: <https://wiki.bitcraze.io/doc:lps:tdoa>.
- [18] Winjun Zeng. “Microsoft Kinect Sensor and Its Effect”. In: *IEEE Computer Society* ().

A

Example Python Code

A.1 Square Flight

```
"""
Square flight example that connects to the crazyflie and sends it through a
    ↪ series of setpoints in the sequence.
Currently setup in a square pattern.

This program is designed to use the Loco Positioning system running in TWR
    ↪ mode.
"""
import time
import csv
import datetime

import cflib.crtplib
from cflib.crazyflie import Crazyflie
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.syncCrazyflie import SyncCrazyflie
from cflib.crazyflie.syncLogger import SyncLogger

# URI to the Crazyflie to connect to
uri = 'radio://0/80/2M/E7E7E7E701'

# Change the sequence according to your setup
#           x     y     z   YAW
sequence = [
    (1.0,0.25, 1.0, 0.0),
    (2.0,0.25, 1.0, 0.0),
    (2.0,1.25, 1.0, 0.0),
    (1.0,1.25, 1.0, 0.0),
    (1.0,0.25, 1.0, 0.0),
    (1.0,0.25, 0.7, 0.0),
    (1.0,0.25, 0.3, 0.0),
```

```
]
```

```
def wait_for_position_estimator(scf):  
    print('Waiting_for_estimator_to_find_position...')  
  
    log_config = LogConfig(name='Kalman_Variance', period_in_ms=500)  
    log_config.add_variable('kalman.varPX', 'float')  
    log_config.add_variable('kalman.varPY', 'float')  
    log_config.add_variable('kalman.varPZ', 'float')  
  
    var_y_history = [1000] * 10  
    var_x_history = [1000] * 10  
    var_z_history = [1000] * 10  
  
    threshold = 0.001  
  
    with SyncLogger(scf, log_config) as logger:  
        for log_entry in logger:  
            data = log_entry[1]  
  
            var_x_history.append(data['kalman.varPX'])  
            var_x_history.pop(0)  
            var_y_history.append(data['kalman.varPY'])  
            var_y_history.pop(0)  
            var_z_history.append(data['kalman.varPZ'])  
            var_z_history.pop(0)  
  
            min_x = min(var_x_history)  
            max_x = max(var_x_history)  
            min_y = min(var_y_history)  
            max_y = max(var_y_history)  
            min_z = min(var_z_history)  
            max_z = max(var_z_history)  
  
            # print("{} {} {}".format(min_x, min_y, min_z))
```

```

#         format(max_x - min_x, max_y - min_y, max_z - min_z))

    if (max_x - min_x) < threshold and (
        max_y - min_y) < threshold and (
            max_z - min_z) < threshold:
        break

def reset_estimator(scf):
    cf = scf.cf
    cf.param.set_value('kalman.resetEstimation', '1')
    time.sleep(0.1)
    cf.param.set_value('kalman.resetEstimation', '0')

    wait_for_position_estimator(cf)

def position_callback(timestamp, data, logconf):
    x = data['kalman.stateX']
    y = data['kalman.stateY']
    z = data['kalman.stateZ']
    print('pos:-( {},- {},- {} )'.format(x, y, z))
    with open('/home/bitcraze/Documents/'+datetime.datetime.now().strftime('%Y
        ↪ -%m-%d-%H')+ '_flight_data.csv', 'a') as csvfile:
        writer = csv.writer(csvfile, delimiter=',')
        writer.writerow([x, y, z])
    csvfile.close()

#
def start_position_printing(scf):
    log_conf = LogConfig(name='Position', period_in_ms=500)
    log_conf.add_variable('kalman.stateX', 'float')
    log_conf.add_variable('kalman.stateY', 'float')
    log_conf.add_variable('kalman.stateZ', 'float')

    scf.cf.log.add_config(log_conf)
    log_conf.data_received_cb.add_callback(position_callback)

```

```

log_conf.start()

def run_sequence(scf, sequence):
    cf = scf.cf

    cf.param.set_value('flightmode.posSet', '1')
    time.sleep(5.0) #Wait to initialize the loco system
    for position in sequence:
        print('Setting_position_{}'.format(position))
        for i in range(50):
            cf.commander.send_setpoint(position[1], position[0],
                                       position[3],
                                       int(position[2] * 1000))

            time.sleep(0.1)

    cf.commander.send_setpoint(0, 0, 0, 0)
    # Make sure that the last packet leaves before the link is closed
    # since the message queue is not flushed before closing
    time.sleep(0.1)

if __name__ == '__main__':
    cflib.crtp.init_drivers(enable_debug_driver=False)

    with SyncCrazyflie(uri, cf=Crazyflie(rw_cache='./cache')) as scf:
        reset_estimator(scf)
        start_position_printing(scf)
        run_sequence(scf, sequence)

```

A.2 4 Drone Pyramid Example

"""

Program that connects to 4 crazyflies and flies them in a "pyramid" formation starting locations as well as circle positions and heights are setup in the

↪ params values

This program is designed to work with the Loco Positioning System running in

↔ TDoA mode

"""

```
import time
```

```
import csv
```

```
import datetime
```

```
import numpy as np
```

```
import cflib.crtip
```

```
from cflib.crazyflie.log import LogConfig
```

```
from cflib.crazyflie.swarm import CachedCfFactory
```

```
from cflib.crazyflie.swarm import Swarm
```

```
from cflib.crazyflie.syncLogger import SyncLogger
```

```
# Change uris and sequences according to your setup
```

```
URI1 = 'radio://0/80/2M/E7E7E7E701'
```

```
URI2 = 'radio://0/80/2M/E7E7E7E702'
```

```
URI3 = 'radio://0/80/2M/E7E7E7E704'
```

```
URI4 = 'radio://0/80/2M/E7E7E7E705'
```

```
#URI5 = 'radio://0/70/2M/E7E7E7E705'
```

```
#URI6 = 'radio://0/70/2M/E7E7E7E706'
```

```
#URI7 = 'radio://0/70/2M/E7E7E7E707'
```

```
#URI8 = 'radio://0/70/2M/E7E7E7E708'
```

```
#URI9 = 'radio://0/70/2M/E7E7E7E709'
```

```
#URI10 = 'radio://0/70/2M/E7E7E7E70A'
```

```
z0 = 0.4
```

```
z = 1.0
```

```
x0 = 1.25
```

```
x1 = 1.0
```

```
#x2 = -0.7
```

```

y0 = 1.25
y1 = 0.25
#y2 = 0.4
#y3 = 1.0

#           r: radius, z: height, x: starting_x, y: starting_y, offset: degree
#           ↪ offset for circle

params1 = {'r': 0.0, 'z': 1.75, 'x_s': 1.25, 'y_s': 1.25, 'x_l': 1.25, 'y_l':
           ↪ 1.25, 'offset': 0}
params2 = {'r': 0.7, 'z': 1.0, 'x_s': 1.75, 'y_s': 0.25, 'x_l': 0.25, 'y_l':
           ↪ 0.25, 'offset': 0}
params3 = {'r': 0.7, 'z': 1.0, 'x_s': 1.0, 'y_s': 0.25, 'x_l': 1.0, 'y_l':
           ↪ 0.25, 'offset': 180}
params4 = {'r': 0.7, 'z': 1.0, 'x_s': 0.25, 'y_s': 0.25, 'x_l': 1.75, 'y_l':
           ↪ 0.25, 'offset': 90}

params = {
    URI1: [params1],
    URI2: [params2],
    URI3: [params3],
    URI4: [params4],
}

# List of URIs, comment the ones you do not want to fly
uris = {
    URI1,
    URI2,
    URI3,
    URI4,
#    URI5,
#    URI6,
#    URI7,
#    URI8,
#    URI9,
#    URI10

```



```

}

def wait_for_position_estimator(scf):
    print('Waiting_for_estimator_to_find_position...')

    log_config = LogConfig(name='Kalman_Variance', period_in_ms=500)
    log_config.add_variable('kalman.varPX', 'float')
    log_config.add_variable('kalman.varPY', 'float')
    log_config.add_variable('kalman.varPZ', 'float')

    var_y_history = [1000] * 10
    var_x_history = [1000] * 10
    var_z_history = [1000] * 10

    threshold = 0.001

    with SyncLogger(scf, log_config) as logger:
        for log_entry in logger:
            data = log_entry[1]

            var_x_history.append(data['kalman.varPX'])
            var_x_history.pop(0)
            var_y_history.append(data['kalman.varPY'])
            var_y_history.pop(0)
            var_z_history.append(data['kalman.varPZ'])
            var_z_history.pop(0)

            min_x = min(var_x_history)
            max_x = max(var_x_history)
            min_y = min(var_y_history)
            max_y = max(var_y_history)
            min_z = min(var_z_history)
            max_z = max(var_z_history)

#         print("{} {} {}".format(min_x, min_y, min_z))

```

```

#             format(max_x - min_x, max_y - min_y, max_z - min_z))

        if (max_x - min_x) < threshold and (
            max_y - min_y) < threshold and (
                max_z - min_z) < threshold:
            break

def wait_for_param_download(scf):
    while not scf.cf.param.is_updated:
        time.sleep(1.0)
    print('Parameters downloaded for ', scf.cf.link_uri)

def reset_estimator(scf):
    cf = scf.cf
    cf.param.set_value('kalman.resetEstimation', '1')
    time.sleep(0.1)
    cf.param.set_value('kalman.resetEstimation', '0')

    wait_for_position_estimator(cf)

def take_off(cf, position):
    take_off_time = 0.8
    sleep_time = 0.1
    steps = int(take_off_time / sleep_time)
    vz = position[2] / take_off_time

    print(vz)

    for i in range(steps):
        cf.commander.send_velocity_world_setpoint(0, 0, vz, 0)
#         cf.commander.send_setpoint(position[1], position[0], 0,
#                                     int(vz * 1000))
#
        time.sleep(sleep_time)

```

```

def land(cf, position):
    landing_time = 1.5
    sleep_time = 0.1
    steps = int(landing_time / sleep_time)
    vz = -position[2] / landing_time

    print(vz)

    for i in range(steps):
        cf.commander.send_velocity_world_setpoint(0, 0, vz, 0)
        time.sleep(sleep_time)

    cf.commander.send_setpoint(0, 0, 0, 0)
    # Make sure that the last packet leaves before the link is closed
    # since the message queue is not flushed before closing
    time.sleep(0.1)

def circle_next_pos(t, r, offset):
    # t: time value, r: radius of circle, offset: position in circle(Degrees)
    x_c = 1.25          #Center x value
    y_c = 1.25          #Center y value
    x_pos = x_c + (r * np.cos(t/10 + offset))
    y_pos = y_c + (r * np.sin(t/10 + offset))
    # yaw = 90 + (5.45 * t)
    desired_pos = (x_pos, y_pos, 1.0, 0.0)
    # print(yaw)
    # print(t)
    return desired_pos

def run_sequence(scf, params):
    #params: dict that holds parameter values
    try:
        cf = scf.cf
        cf.param.set_value('flightmode.posSet', '1')

```

```

r = params[ 'r' ]
z = params[ 'z' ]
x_s = params[ 'x_s' ]
y_s = params[ 'y_s' ]
x_l = params[ 'x_l' ]
y_l = params[ 'y_l' ]
offset = params[ 'offset' ]

# Takeoff Sequence
take_off(cf, (x_s, y_s, 0.5, 0)) #(x,y,z,yaw)

# Circle Sequence
for t in range(100):
    #Run the circle position updates
    position = circle_next_pos(t,r,offset)
    cf.commander.send_setpoint(position[1], position[0],
                                position[3],
                                int(z * 1000))

    time.sleep(0.25)

# Move to landing positions
for t in range(30):
    cf.commander.send_setpoint(y_l, x_l,
                                0,
                                int(z * 1000))

    time.sleep(0.2)
    land(cf, (x_l, y_l, z, 0))
except Exception as e:
    print(e)

def position_callback(timestamp, data, logconf):
    x = data[ 'kalman.stateX' ]
    y = data[ 'kalman.stateY' ]
    z = data[ 'kalman.stateZ' ]
    print( 'pos:_{},_{},_{}' .format(x, y, z) )

```

```

# Write position data to .csv; Must be updated to match your specific
  ↪ filesystem
with open('/home/bitcraze/Documents/'+datetime.datetime.now().strftime('%Y
  ↪ -%m-%d-%H-')+ '_flight_data.csv', 'a') as csvfile:
    writer = csv.writer(csvfile, delimiter=',')
    writer.writerow([x, y, z, timestamp])
csvfile.close()

def start_position_printing(scf):
    log_conf = LogConfig(name='Position', period_in_ms=50)
    log_conf.add_variable('kalman.stateX', 'float')
    log_conf.add_variable('kalman.stateY', 'float')
    log_conf.add_variable('kalman.stateZ', 'float')

    scf.cf.log.add_config(log_conf)
    log_conf.data_received_cb.add_callback(position_callback)
    log_conf.start()

if __name__ == '__main__':
    # logging.basicConfig(level=logging.DEBUG)
    cflib.crtp.init_drivers(enable_debug_driver=False)

    factory = CachedCfFactory(rw_cache='./cache')
    with Swarm(uris, factory=factory) as swarm:
        # If the copters are started in their correct positions this is
        # probably not needed. The Kalman filter will have time to converge
        # any way since it takes a while to start them all up and connect. We
        # keep the code here to illustrate how to do it.
        swarm.parallel(reset_estimator)
        time.sleep(2.0)

        # The current values of all parameters are downloaded as a part of the
        # connections sequence. Since we have 10 copters this is clogging up
        # communication and we have to wait for it to finish before we start

```

```

# flying.
print('Waiting_for_parameters_to_be_downloaded...')
swarm.parallel(wait_for_param_download)
swarm.parallel(start_position_printing)    #activate the position
    ↪ printing to screen and to csv
time.sleep(3.0)
swarm.parallel(run_sequence, args_dict=params) #run the sequence and
    ↪ input parameters

```

A.3 5 Drone Example

```

"""
Program that connects to 5 crazyflies and flies them in concentric circles at
    ↪ 2 different heights
starting locations as well as circle positions and heights are setup in the
    ↪ params values

This program is designed to work with the Loco Positioning System running in
    ↪ TDoA mode
"""

import time
import csv
import datetime
import numpy as np

import cflib.crtplib
from cflib.crazyflie.log import LogConfig
from cflib.crazyflie.swarm import CachedCfFactory
from cflib.crazyflie.swarm import Swarm
from cflib.crazyflie.syncLogger import SyncLogger

# Change uris and sequences according to your setup
URI1 = 'radio://0/80/2M/E7E7E7E701'
URI2 = 'radio://0/80/2M/E7E7E7E702'

```

```

URI3 = 'radio://0/80/2M/E7E7E7E703'
URI4 = 'radio://0/80/2M/E7E7E7E704'
URI5 = 'radio://0/80/2M/E7E7E7E705'
#URI6 = 'radio://0/70/2M/E7E7E7E706'
#URI7 = 'radio://0/70/2M/E7E7E7E707'
#URI8 = 'radio://0/70/2M/E7E7E7E708'
#URI9 = 'radio://0/70/2M/E7E7E7E709'
#URI10 = 'radio://0/70/2M/E7E7E7E70A'

z0 = 0.4
z = 1.0

x0 = 1.25
x1 = 1.0
#x2 = -0.7

y0 = 1.25
y1 = 0.25
#y2 = 0.4
#y3 = 1.0

#           r: radius, z: height, x: starting_x, y: starting_y, offset: degree
#           ↪ offset for circle, dir: direction around circle

params1 = {'r': 0.7, 'z': 1.75, 'x_s': 1.25, 'y_s': 1.25, 'x_l': 1.25, 'y_l':
           ↪ 1.25, 'offset': 45, 'dir': 1}
params2 = {'r': 0.7, 'z': 1.0, 'x_s': 1.75, 'y_s': 0.25, 'x_l': 0.25, 'y_l':
           ↪ 0.25, 'offset': 0, 'dir': 1}
params3 = {'r': 0.7, 'z': 1.0, 'x_s': 1.0, 'y_s': 0.25, 'x_l': 1.0, 'y_l':
           ↪ 0.25, 'offset': 180, 'dir': 1}
params4 = {'r': 0.7, 'z': 1.0, 'x_s': 0.25, 'y_s': 0.25, 'x_l': 1.75, 'y_l':
           ↪ 0.25, 'offset': 90, 'dir': 1}
params5 = {'r': 0.7, 'z': 1.75, 'x_s': 0.25, 'y_s': 1.75, 'x_l': 1.75, 'y_l':
           ↪ 1.75, 'offset': 225, 'dir': 1}

```

```

# assign the parameters to the URIs
params = {
    URI1: [params1],
    URI2: [params2],
    URI3: [params3],
    URI4: [params4],
    URI5: [params5],
}

# List of URIs, comment the one you do not want to fly
uris = {
    URI1,
    URI2,
    URI3,
    URI4,
    URI5,
#    URI6,
#    URI7,
#    URI8,
#    URI9,
#    URI10
}

def wait_for_position_estimator(scf):
    print('Waiting for estimator to find position...')

    log_config = LogConfig(name='Kalman_Variance', period_in_ms=500)
    log_config.add_variable('kalman.varPX', 'float')
    log_config.add_variable('kalman.varPY', 'float')
    log_config.add_variable('kalman.varPZ', 'float')

    var_y_history = [1000] * 10
    var_x_history = [1000] * 10
    var_z_history = [1000] * 10

```



```

threshold = 0.001

with SyncLogger(scf, log_config) as logger:
    for log_entry in logger:
        data = log_entry[1]

        var_x_history.append(data['kalman.varPX'])
        var_x_history.pop(0)
        var_y_history.append(data['kalman.varPY'])
        var_y_history.pop(0)
        var_z_history.append(data['kalman.varPZ'])
        var_z_history.pop(0)

        min_x = min(var_x_history)
        max_x = max(var_x_history)
        min_y = min(var_y_history)
        max_y = max(var_y_history)
        min_z = min(var_z_history)
        max_z = max(var_z_history)

#         print("{} {} {}".
#               format(max_x - min_x, max_y - min_y, max_z - min_z))

        if (max_x - min_x) < threshold and (
            max_y - min_y) < threshold and (
            max_z - min_z) < threshold:
            break

def wait_for_param_download(scf):
    while not scf.cf.param.is_updated:
        time.sleep(1.0)
    print('Parameters downloaded for', scf.cf.link_uri)

def reset_estimator(scf):

```

```

cf = scf.cf
cf.param.set_value('kalman.resetEstimation', '1')
time.sleep(0.1)
cf.param.set_value('kalman.resetEstimation', '0')

wait_for_position_estimator(cf)

def take_off(cf, position):
    take_off_time = 0.8
    sleep_time = 0.1
    steps = int(take_off_time / sleep_time)
    vz = position[2] / take_off_time

    print(vz)

    for i in range(steps):
        cf.commander.send_velocity_world_setpoint(0, 0, vz, 0)
#         cf.commander.send_setpoint(position[1], position[0], 0,
#                                     int(vz * 1000))
        time.sleep(sleep_time)

def land(cf, position):
    landing_time = 1.5
    sleep_time = 0.1
    steps = int(landing_time / sleep_time)
    vz = -position[2] / landing_time

    print(vz)

    for i in range(steps):
        cf.commander.send_velocity_world_setpoint(0, 0, vz, 0)
        time.sleep(sleep_time)

cf.commander.send_setpoint(0, 0, 0, 0)

```

```

# Make sure that the last packet leaves before the link is closed
# since the message queue is not flushed before closing
time.sleep(0.1)

def circle_next_pos(t,r,offset,direction):
# t: time value, r: radius of circle, offset: position in circle(Degrees),
  ↪ direction: direction to go around circle
    x_c = 1.25
    y_c = 1.25
    t = t * direction
    x_pos = x_c + (r * np.cos(t/10 + offset))
    y_pos = y_c + (r * np.sin(t/10 + offset))
#   yaw = 90 + (5.45 * t)
    desired_pos = (x_pos, y_pos, 1.0, 0.0)
#   print(yaw)
#   print(t)
    return desired_pos

def run_sequence(scf, params):
#params: dict that holds parameter values
    try:
        cf = scf.cf
        cf.param.set_value('flightmode.posSet', '1')
        r = params['r']
        z = params['z']
        x_s = params['x-s']
        y_s = params['y-s']
        x_l = params['x-l']
        y_l = params['y-l']
        offset = params['offset']
        direction = params['dir']

# Takeoff Sequence
        take_off(cf, (x_s, y_s, 0.5, 0)) #(x,y,z,yaw)

```

```

# Circle Sequence
    for t in range(100):          #Run the circle position updates
        position = circle_next_pos(t,r,offset,direction)
        cf.commander.send_setpoint(position[1], position[0],
                                    position[3],
                                    int(z * 1000))

        time.sleep(0.25)

# Move to landing positions
    for t in range(30):
        cf.commander.send_setpoint(y_l, x_l,
                                    0,
                                    int(z * 1000))

        time.sleep(0.2)
        land(cf, (x_l, y_l, z, 0))
except Exception as e:
    print(e)

def position_callback(timestamp, data, logconf):
    x = data['kalman.stateX']
    y = data['kalman.stateY']
    z = data['kalman.stateZ']
    print('pos:-({},{},-{})'.format(x, y, z))

# Write position data to .csv; Must be updated to match your specific
↪ filesystem
with open('/home/bitcraze/Documents/'+datetime.datetime.now().strftime('%Y
↪ -%m-%d-%H_')+ '_flight_data.csv', 'a') as csvfile:
    writer = csv.writer(csvfile, delimiter=',')
    writer.writerow([x, y, z, timestamp])
csvfile.close()

def start_position_printing(scf):
    log_conf = LogConfig(name='Position', period_in_ms=50)
    log_conf.add_variable('kalman.stateX', 'float')

```

```

log_conf.add_variable('kalman.stateY', 'float')
log_conf.add_variable('kalman.stateZ', 'float')

scf.cf.log.add_config(log_conf)
log_conf.data_received_cb.add_callback(position_callback)
log_conf.start()

if __name__ == '__main__':
    # logging.basicConfig(level=logging.DEBUG)
    cflib.crtp.init_drivers(enable_debug_driver=False)

    factory = CachedCfFactory(rw_cache='./cache')
    with Swarm(uris, factory=factory) as swarm:
        # If the copters are started in their correct positions this is
        # probably not needed. The Kalman filter will have time to converge
        # any way since it takes a while to start them all up and connect. We
        # keep the code here to illustrate how to do it.
        swarm.parallel(reset_estimator)
        time.sleep(2.0)

        # The current values of all parameters are downloaded as a part of the
        # connections sequence. Since we have 10 copters this is clogging up
        # communication and we have to wait for it to finish before we start
        # flying.
        print('Waiting for parameters to be downloaded...')
        swarm.parallel(wait_for_param_download)
        swarm.parallel(start_position_printing) #activate the position
            ↪ printing to screen and to csv
        time.sleep(3.0)
        swarm.parallel(run_sequence, args_dict=params) #run the sequence and
            ↪ input parameters

```