The University of California, Los Angeles

Robotics Design Capstone

EE 183DB

Off-center spinning mass controller for Quadcopters

Author:
Lin Li
Angel Jimenez
Wilson Chang
Amirali Omidfar

Professor:
Ankur Mehta

June 16, 2018

Abstract

We aim to design an off-center spinning mass underactuated controller to steer flying objects. A quadcopter with a rotating arm attached to it is used to demonstrate the principle of such controller. To approach this control problem, we first derive the Mathematical Model, followed by simulation and motor control, and finally execute the actual implementation. We believe the problems we faced and the solutions we created can provide wisdom and experimental knowledge to anyone interested in picking up where we left off. All source code and demo videos can be found in this Github repo.

Contents

1 Introduction

Modern Rocket uses 2 DOF revolute joint to turn the nozzle to directly control the direction of thrust. Challenges are it has to resist a very high temperature and the joint need a large amount of energy to keep the nozzle in a specific direction. Instead, a precisely controlled off-center mass in the front of the rocket can create a torque that steers the Rocket.

We aim to explore an alternative way to steer flying vehicles with under-actuated controller. Taking the motivation from modern rocket control, we are going to implement such controller in a quadcopter and explore the possibilities of such control. We hope to extend such controller to steer rockets in a more cost and energy efficient manner.

2 Mathematical Model

2.1 Symbols

Here is a list of all symbols used in this section:

$\boldsymbol{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$	linear position vectors
$\boldsymbol{q} = \begin{bmatrix} q_r \\ q_i \\ q_j \\ q_k \end{bmatrix}$	angular orientation in quaternion
$\boldsymbol{F_T}$	thrust force
$\boldsymbol{F_G}$	gravitational force
$\boldsymbol{F_{AB}}$	reaction force acted from A on B
$\boldsymbol{\tau_{AB}}$	reaction torque acted from A on B
$\boldsymbol{\tau_M}$	torque generated by the motor
$\boldsymbol{\tau_{RF}}$	torque generated by the reaction force
m_A	mass of A
I_A	moment of inertial of A

2.2 Appendix

The Quaternion-derived Rotation matrix is defined as follow,

$$ {}^{O}_{B}R = R(\boldsymbol{q_B}) = \begin{bmatrix} q_r^2 + q_i^2 - q_j^2 - q_k^2 & 2q_iq_j - 2q_rq_k & 2q_iq_k + 2q_rq_j \\ 2q_iq_j + 2q_rq_k & q_r^2 - q_i^2 + q_j^2 - q_k^2 & 2q_jq_k - 2q_rq_i \\ 2q_iq_k - 2q_rq_j & 2q_jq_k + 2q_rq_i & q_r^2 - q_i^2 - q_j^2 + q_k^2 \end{bmatrix} $$

2.3 Quadcopter Body Dynamics

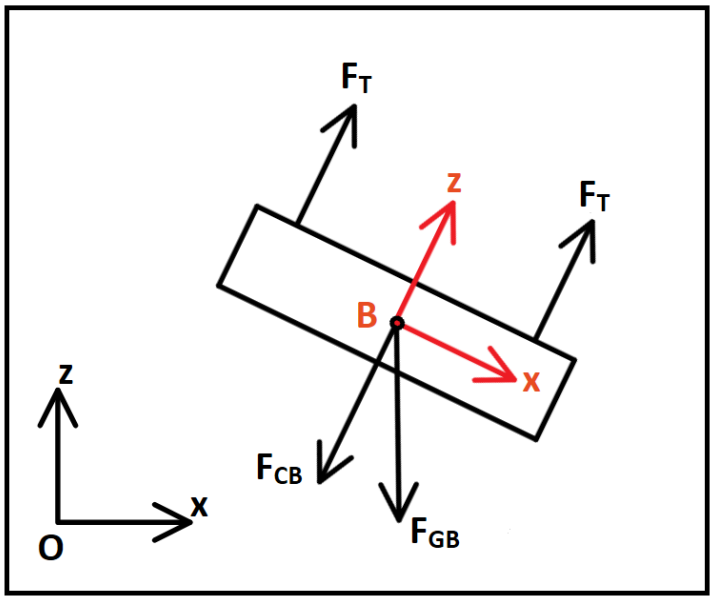

Figure 1: Free-Body diagram of Body

Forces and Torques:

$$
\begin{aligned}
{}^{B}\boldsymbol{F_T} &= \begin{bmatrix} 0 \\ 0 \\ F_{TB} \end{bmatrix} \\
{}^{O}\boldsymbol{F_{GB}} &= \begin{bmatrix} 0 \\ 0 \\ -m_b g \end{bmatrix} \\
{}^{O}\boldsymbol{F_{CB}} &= \begin{bmatrix} F_{CBx} \\ F_{CBy} \\ F_{CBz} \end{bmatrix} \\
{}^{B}\boldsymbol{\tau_{CB}} &= \begin{bmatrix} \tau_{CBx} \\ \tau_{CBy} \\ -\tau_M \end{bmatrix}
\end{aligned}
$$

Net Force and Torque

$$
{}^{O}\boldsymbol{F_{net,B}} = {}^{O}\boldsymbol{F_{GB}} + {}^{O}\boldsymbol{F_T} + {}^{O}\boldsymbol{F_{CB}} = m_B\, {}^{O}\boldsymbol{a_B} \tag{1}
$$

$$
{}^{O}\boldsymbol{\tau_{net,B}} = R(\boldsymbol{q_B})\, {}^{B}\boldsymbol{\tau_{CB}} = {}^{O}I_B\, {}^{O}\boldsymbol{\alpha_B} \tag{2}
$$

2.4 Controller Dynamics

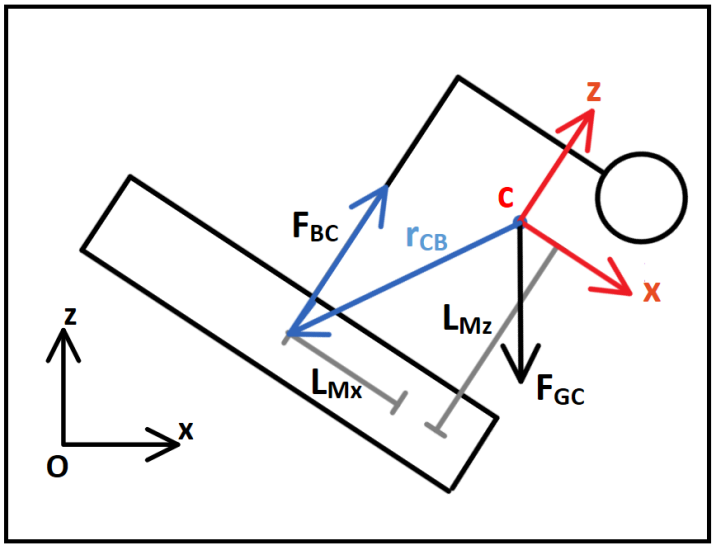

Figure 2: Free-Body diagram of Controller

Forces and Torques:

$$
\begin{aligned}
{}^{O}\boldsymbol{F_{BC}} &= \begin{bmatrix} F_{BCx} \\ F_{BCy} \\ F_{BCz} \end{bmatrix} \\
{}^{O}\boldsymbol{F_{GC}} &= \begin{bmatrix} 0 \\ 0 \\ -m_c g \end{bmatrix} \\
{}^{C}\boldsymbol{\tau_{BC}} &= \begin{bmatrix} \tau_{BCx} \\ \tau_{BCy} \\ \tau_{M} \end{bmatrix} \\
{}^{O}\boldsymbol{r_{CB}} &= R(\boldsymbol{q_C}) \begin{bmatrix} -L_{Mx} \\ 0 \\ -L_{Mz} \end{bmatrix} \\
{}^{O}\boldsymbol{\tau_{RF}} &= {}^{O}\boldsymbol{r_{CB}} \times {}^{O}\boldsymbol{F_{BC}}
\end{aligned}
$$

Net Force and Net Torque:

$$
{}^{O}\boldsymbol{F_{net,C}} = {}^{O}\boldsymbol{F_{BC}} + {}^{O}\boldsymbol{F_{GC}} = m_C \, {}^{O}\boldsymbol{a_C} \tag{3}
$$

$$
{}^{O}\boldsymbol{\tau_{net,C}} = R(\boldsymbol{q_C}) \, {}^{C}\boldsymbol{\tau_{BC}} + {}^{O}\boldsymbol{\tau_{RF}} = {}^{O}I_c \, {}^{O}\boldsymbol{\alpha_C} \tag{4}
$$

2.5 Constraints and Manipulation

In the derivation below, assume everything is in the inertial frame unless explicitly stated.

The two bodies are contrainted (attached together), there are some relationship between the states and the forces between the body and the controller,

Let $\boldsymbol{p_{sys}} = \boldsymbol{p_B}$ and $\boldsymbol{q_{sys}} = \boldsymbol{q_B}$,

$$
\begin{bmatrix} \boldsymbol{p_C} \\ \boldsymbol{q_C} \end{bmatrix} = \begin{bmatrix} \boldsymbol{p_B} + \boldsymbol{r_{BC}} \\ \boldsymbol{q_\theta q_B} \end{bmatrix} = \begin{bmatrix} \boldsymbol{p_{sys}} + \boldsymbol{r_{BC}} \\ \boldsymbol{q_\theta q_{sys}} \end{bmatrix} \tag{5}
$$

$$
\begin{bmatrix} \dot{\boldsymbol{p}}_{\boldsymbol{C}} \\ \dot{\boldsymbol{q}}_{\boldsymbol{C}} \end{bmatrix} = \begin{bmatrix} \dot{\boldsymbol{p}}_{\boldsymbol{sys}} + \dot{R}(\boldsymbol{q_{sys}}) \, {}^{B}\boldsymbol{r_{BC}} \\ \boldsymbol{q_\theta} \dot{\boldsymbol{q}}_{\boldsymbol{sys}} + \dot{\boldsymbol{q}}_{\boldsymbol{\theta}} \boldsymbol{q_{sys}} \end{bmatrix} \tag{6}
$$

$$
\begin{bmatrix} \ddot{\boldsymbol{p}}_{\boldsymbol{C}} \\ \ddot{\boldsymbol{q}}_{\boldsymbol{C}} \end{bmatrix} = \begin{bmatrix} \ddot{\boldsymbol{p}}_{\boldsymbol{sys}} + \ddot{R}(\boldsymbol{q_{sys}}) \, {}^{B}\boldsymbol{r_{BC}} \\ \boldsymbol{q_\theta} \ddot{\boldsymbol{q}}_{\boldsymbol{sys}} + 2[\dot{\boldsymbol{q}}_{\boldsymbol{\theta}} \dot{\boldsymbol{q}}_{\boldsymbol{sys}}] + \ddot{\boldsymbol{q}}_{\boldsymbol{\theta}} \boldsymbol{q_{sys}} \end{bmatrix} \tag{7}
$$

Newton's Third Law

$$^{O}\boldsymbol{F_{BC}} = -\,^{O}\boldsymbol{F_{CB}} \tag{8}$$

$$^{O}\boldsymbol{\tau_{BC}} = -\,^{O}\boldsymbol{\tau_{CB}} \tag{9}$$

To limit our degree of freedom in the system, we have set a constraint for our quaternions, namely unit quaternion:

$$q_r^2 + q_i^2 + q_j^2 + q_k^2 = 1 \tag{10}$$

$$q_r\dot{q}_r + q_i\dot{q}_i + q_j\dot{q}_j + q_k\dot{q}_k = 0 \tag{11}$$

$$q_r\ddot{q}_r + q_i\ddot{q}_i + q_j\ddot{q}_j + q_k\ddot{q}_k + \dot{q}_r{}^2 + \dot{q}_i{}^2 + \dot{q}_j{}^2 + \dot{q}_k{}^2 = 0 \tag{12}$$

Last but not least, in the derivation below we use $\boldsymbol{q_\theta}$ directly for ease of typsetting, however, q_θ is not our state variable but θ, their relationship is defined below,

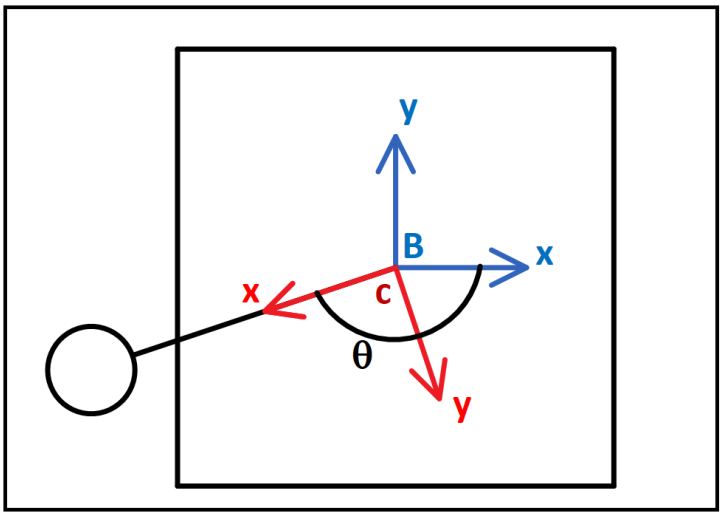

Figure 3: The yaw angle difference between Body and Controller

$$\boldsymbol{q_\theta} = \cos(\frac{\theta}{2}) + \sin(\frac{\theta}{2})R(\boldsymbol{q_{sys}})\,^{B}\hat{\boldsymbol{z}}_{\boldsymbol{B}}$$

$$\dot{\boldsymbol{q}}_{\boldsymbol{\theta}} = -\frac{1}{2}\sin(\frac{\theta}{2})\dot{\theta} + \frac{1}{2}\cos(\frac{\theta}{2})\dot{\theta}R(\boldsymbol{q_{sys}})\,^{B}\hat{\boldsymbol{z}}_{\boldsymbol{B}} + \sin(\frac{\theta}{2})R(\dot{\boldsymbol{q}}_{\boldsymbol{sys}})\,^{B}\hat{\boldsymbol{z}}_{\boldsymbol{B}}$$

where $^{B}\hat{\boldsymbol{z}}_{\boldsymbol{B}} = \begin{bmatrix}0\\0\\1\end{bmatrix}$

2.5.1 Combining the Force equations

From (1),

$$^{O}\boldsymbol{F_{CB}} = m_B\, ^{O}\boldsymbol{a_B} - ^{O}\boldsymbol{F_{GB}} - ^{O}\boldsymbol{F_T}$$

From (3),

$$^{O}\boldsymbol{F_{BC}} = m_C\, ^{O}\boldsymbol{a_C} - ^{O}\boldsymbol{F_{GC}}$$

Using (6),

$$m_B\, ^{O}\boldsymbol{a_B} + m_C\, ^{O}\boldsymbol{a_C} = ^{O}\boldsymbol{F_{GC}} + ^{O}\boldsymbol{F_{GB}} + ^{O}\boldsymbol{F_T}$$

Simplifying the above expression, we get

$$(m_b + m_c)\ddot{\boldsymbol{p}}_{\boldsymbol{sys}} + m_c\ddot{R}(\boldsymbol{q_{sys}})\, ^{B}\boldsymbol{r_{BC}} = \boldsymbol{F_{GC}} + \boldsymbol{F_{GB}} + \boldsymbol{F_T} \tag{13}$$

2.5.2 Combining the Torqe equations

From (2),

$$^{O}\boldsymbol{\tau_{CB}} = ^{O}I_B\, ^{O}\boldsymbol{\alpha_B}$$

From (4),

$$^{O}\boldsymbol{\tau_{BC}} = ^{O}I_c\, ^{O}\boldsymbol{\alpha_C} - ^{O}\boldsymbol{\tau_{RF}}$$

Using (7),

$$^{O}I_B\, ^{O}\boldsymbol{\alpha_B} + ^{O}I_c\, ^{O}\boldsymbol{\alpha_C} = ^{O}\boldsymbol{\tau_{RF}}$$

Assuming all the vectors are represented in the inertial O frame, using the quaternion representation for angular acceleration,

$$I_B 2\left[\ddot{\boldsymbol{q}}_{\boldsymbol{B}}\boldsymbol{q}^{*}_{\boldsymbol{B}} - (\dot{\boldsymbol{q}}_{\boldsymbol{B}}\boldsymbol{q}^{*}_{\boldsymbol{B}})^2\right] + I_c 2\left[\ddot{\boldsymbol{q}}_{\boldsymbol{C}}\boldsymbol{q}^{*}_{\boldsymbol{C}} - (\dot{\boldsymbol{q}}_{\boldsymbol{C}}\boldsymbol{q}^{*}_{\boldsymbol{C}})^2\right] = ^{O}\boldsymbol{\tau_{RF}} \tag{14}$$

Substituting (5)-(7) in the above expression and isolating second derivative on the left, we have

$$2I_B[\ddot{\boldsymbol{q}}_{\boldsymbol{sys}}\boldsymbol{q}^{*}_{\boldsymbol{sys}}] + 2I_C[\boldsymbol{q_\theta}\ddot{\boldsymbol{q}}_{\boldsymbol{sys}}(\boldsymbol{q_\theta}\boldsymbol{q_{sys}})^{*}] + 2I_C[\ddot{\boldsymbol{q}}_{\boldsymbol{\theta}}\boldsymbol{q_{sys}}](\boldsymbol{q_\theta}\boldsymbol{q_{sys}})^{*} - \boldsymbol{r_{CB}} \times \boldsymbol{F_{BC}} = \zeta \tag{15}$$

where

$$\zeta = 2I_B(\dot{\boldsymbol{q}}_{\boldsymbol{sys}}\boldsymbol{q}^{*}_{\boldsymbol{sys}})^2 + 2I_C[(\boldsymbol{q_\theta}\dot{\boldsymbol{q}}_{\boldsymbol{sys}} + \dot{\boldsymbol{q}}_{\boldsymbol{\theta}}\boldsymbol{q_{sys}})(\boldsymbol{q_\theta}\boldsymbol{q_{sys}})^{*}]^2 - 4I_C(\dot{\boldsymbol{q}}_{\boldsymbol{\theta}}\dot{\boldsymbol{q}}_{\boldsymbol{sys}})(\boldsymbol{q_\theta}\boldsymbol{q_{sys}})^{*}$$

Note that we put $\boldsymbol{\tau_{RF}}$ on the left hand side, this is because we can express $\boldsymbol{F_{BC}}$ in terms of $\ddot{\boldsymbol{p}}_{\boldsymbol{sys}}$ from (1), a second derivative of positional state

$$\boldsymbol{F_{BC}} = m_B\ddot{\boldsymbol{p}}_{\boldsymbol{sys}} - \boldsymbol{F_{GB}} - \boldsymbol{F_T}$$

2.6 System of equations

From equation (12), (14), and (15), we have the function that relates our state variables together,

$$f(\ddot{\boldsymbol{p}}, \ddot{\boldsymbol{q}}, \ddot{\theta}, \dot{\boldsymbol{p}}, \dot{\boldsymbol{q}}, \dot{\theta}, \boldsymbol{p}, \boldsymbol{q}, \theta) = 0 \tag{16}$$

Assuming we can solve for $\ddot{\boldsymbol{p}}, \ddot{\boldsymbol{q}}, \ddot{\theta}$ given $\dot{\boldsymbol{p}}, \dot{\boldsymbol{q}}, \dot{\theta}, \boldsymbol{p}, \boldsymbol{q}, \theta$, let the state of our system to be

$$\boldsymbol{s_{sys}} = \begin{bmatrix} \dot{\boldsymbol{p}} \\ \dot{\boldsymbol{q}} \\ \dot{\theta} \\ \boldsymbol{p} \\ \boldsymbol{q} \\ \theta \end{bmatrix} \quad \text{so that} \quad \dot{\boldsymbol{s}}_{\boldsymbol{sys}} = \begin{bmatrix} \ddot{\boldsymbol{p}} \\ \ddot{\boldsymbol{q}} \\ \ddot{\theta} \\ \dot{\boldsymbol{p}} \\ \dot{\boldsymbol{q}} \\ \dot{\theta} \end{bmatrix}$$

We have our state evolution equations as

$$s_{t+1} = s_t + \dot{s}_t \Delta t \tag{17}$$

2.7 Matlab Implementation

Implementing the systems of equations in (16), and solve for $\ddot{\boldsymbol{p}}, \ddot{\boldsymbol{q}}, \ddot{\theta}$ given $\dot{\boldsymbol{p}}, \dot{\boldsymbol{q}}, \dot{\theta}, \boldsymbol{p}, \boldsymbol{q}, \theta$ in Matlab doesn't yield a solution. There must be something wrong with the equations / the implementation.

3 Simulation

3.1 The goals of simulation

Before starting the real-world implementation, we decided to make a dynamic 3D simulation of the quadcopter with spinning mass to visualize the motion of it and get a better understanding of the specific behavior of such a quadcopter.

We set three main goals for the 3D simulation. First, we want to see a 3D graphic model for the quadcopter with as many details as possible. Second, using our knowledge of physics and math, we want to describe the motion

of the quadcopter to be as close to the real-world-motion as possible in the 3D simulation world. Third, we want to obtain as many useful parameters and insights as possible from the simulation so that we can apply them in the real implementation.

3.2 Building 3D structure

The first thing we did was to measure every possible dimension of the quadcopter and sketch the corresponding geometric figure of it. Then we tried to decompose the entire geometric figure with several typical geometric components such as cylinder, sphere etc.

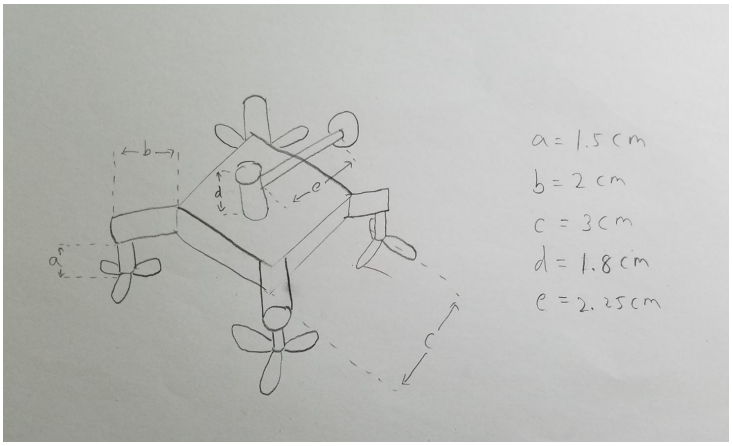

Figure 4: The rough sketch of the quadcopter

Geometric Components		
Name	Shape	Amount
Main Body	Box	1
Motor	Cylinder	1
Mass Stick	Cylinder	1
Mass	Sphere	1
Leg	Cylinder	4
Propeller Holder	Cylinder	4
Propeller	Propeller	4

The reason why we used the typical geometric components is that they can be easily defined in the 3D editor. In the 3D editor, we had to build the 3D

quadcopter structure piece by piece. And if we can define each piece with the simple geometric shape, then we can just try to combine them together properly to get the whole 3D structure.

As it's been mentioned, we used Simulink's 3D world editor to build the 3D geometric structure for the quadcopter. There're two important things I'd like to mention about the 3D world editor.

First, its parent-children relationship tree. Each geometric component of the structure plays a role as either a parent or a child or both in the relationship tree. For example, a leg of the quadcopter is the parent component of the propeller and at the same time it's the child component of the main body.

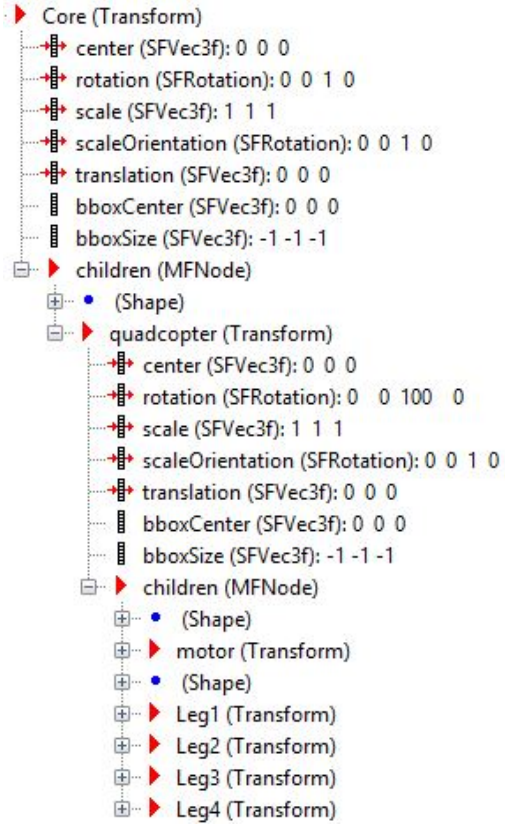

Figure 5: The parent-children relationship tree

When the parent component is changing its state, all its children components and its grandchildren components will also change their reference frame and origin with it. However, when a child component is changing its state, it will not effect on either the parent or other sibling children components. This is similar with the joints of an robotic arm, the (n)th joint will effect on the (n+1)th joint, but (n+1)th joint won't effect on its previous joint – (n)th joint.

Second, its state representation system. The 3D world editor is using the Axis-Angle representation(quaternion representation) for its rotational state representation.

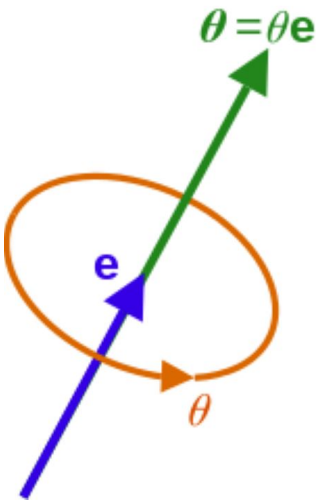

Figure 6: The Axis-Angle representation

Basically, it uses a 4-vector to represent the rotational state. The first 3 entries of the vector represent the x,y,z component of the rotating axis direction(vector (e) in the figure), and the fourth entry represents the angle(theta in the figure) that the corresponding object has been rotated due to the rotating axis.

For the translational state, it uses the normal Cartesian coordinate system representation. It has a 3-vector with each of the entry represents x,y,z component in the cartesian coordinate system.

In other words, each of the geometric piece has up to 7 entries(4 from the rotational state, 3 from the translational state) that can be updated in each cycle of the 3D frame generating.

We had to initialize these 7 states for each of the geometric component one by one. We started with the core of the quadcopter. The core can be understood as the geometric center of the main body of the quadcopter. Then we added the main body as the child of the core. The motor and the four legs were added as the children of the main body, the mass stick and the mass were added as the child and the grandchild of the motor.

And finally, the propeller holders and the propellers were added as the children and the grandchildren of the legs. In this way, we finished building the 3D geometric structure of our quadcopter as it can be seen in the figure.

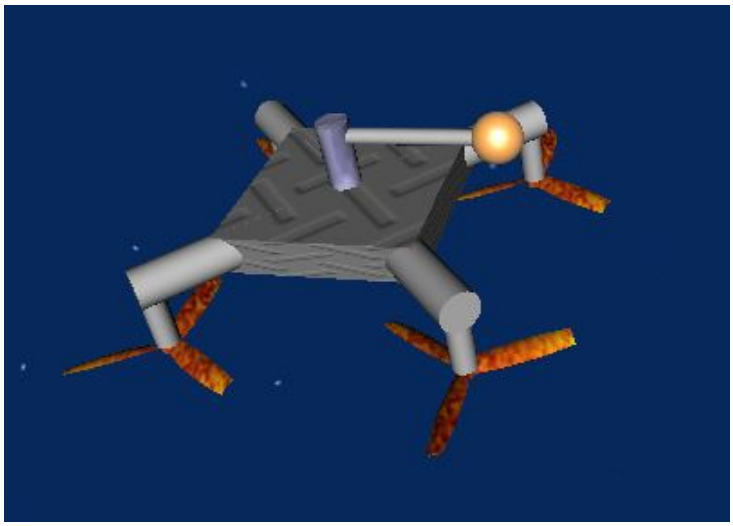

Figure 7: The completed quadcopter 3D model

3.3 Creating dynamic 3D simulation

After building the 3D model of the quadcopter, we started creating the dynamic 3D simulation using Simulink blocks. We embedded the 3D structure we built into the block VR sink.

When we run the Simulink program, in each unit time cycle, the VR sink block will generate one 3D frame according to the current state of the 3D structure in it.

So if we can update the state of the 3D structure, we can obtain a dynamic 3D simulation from it. The state of the 3D structure in the VR sink block can be updated through the namely "state updating windows".As it can be seen in the figure, there are some arrows pointing at "windows" called "Core.translation", "motor.rotation" etc. For example, we can use the window called "Core.translation" to update the translational state of the core of the quadcopter.

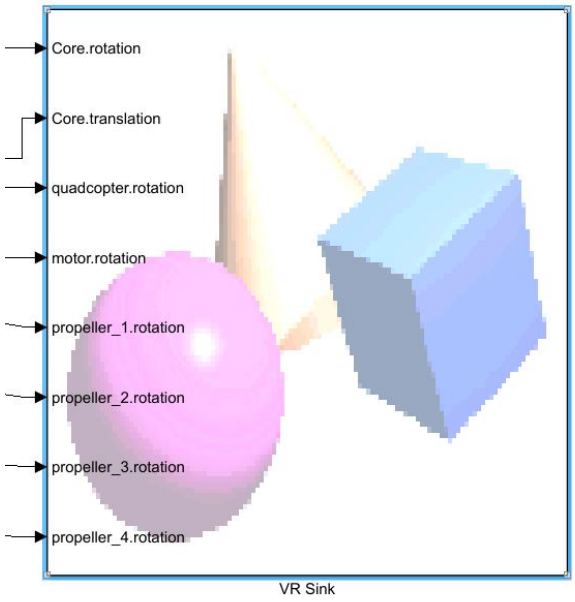

Figure 8: The VR sink block and the state updating windows

Before updating any state, we had to determine exactly how many and which states from which geometric components we'd like to update with the state updating windows.

We have 4 rotating propellers attached on the legs, 1 rotating arm(or motor) on the top of the quadcopter, and the main body of the quadcopter will also rotate, the core of the entire quadcopter will be rotating and translating.

Therefore, in order to obtain a realistic simulation, we needed to update 6 rotational states and 1 translational state. And each of the rotational state had to be updated using a 4-vector data and the translational state had to be updated using a 3-vector data. So we added 7 state updating windows on the VR sink block with the proper Simulink block settings.

Apparently, the most important states among the 7 states we chose are the translational and rotational states of the core. Because they are the states that can be used to describe the overall movement of the quadcopter.

Other states, such as the rotational state of the propellers are just for creating more realistic animation, so they just needed to be set up with a reasonable constant as its rotating speed.

In this simulation, we were assuming that the relative positions of the propellers are perfectly symmetric and the thrust force from each of them is exactly the same so that there's a constant thrust force that is applied at right on the center of the core.

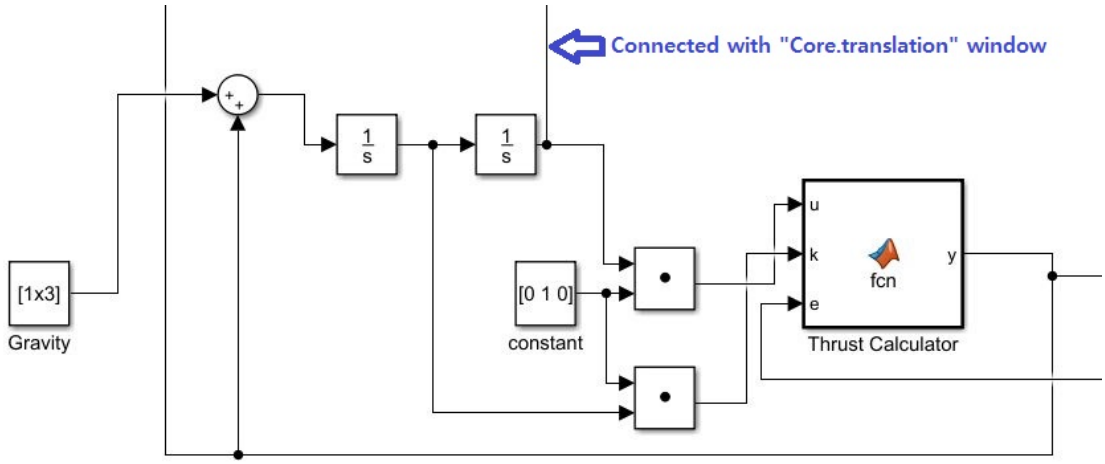

Figure 9: The translational acceleration calculating system

The direction and the amplitude of that thrust force will be calculated in the thrust calculator block in each unit time cycle and the result will be used to

update the translational state of the core.

Other than that, the main role of the translational acceleration calculating system is to add the translational acceleration which come from the thrust force and gravitational force and use that to update the translational state of the core. The translational state has only 3 DOF and they can be fully described by this translational acceleration calculating system.

For the rotational state, however, unfortunately we don't have any complete math model to fully describe all situations. So we had to make assumptions that can limit the complexity of the motion.

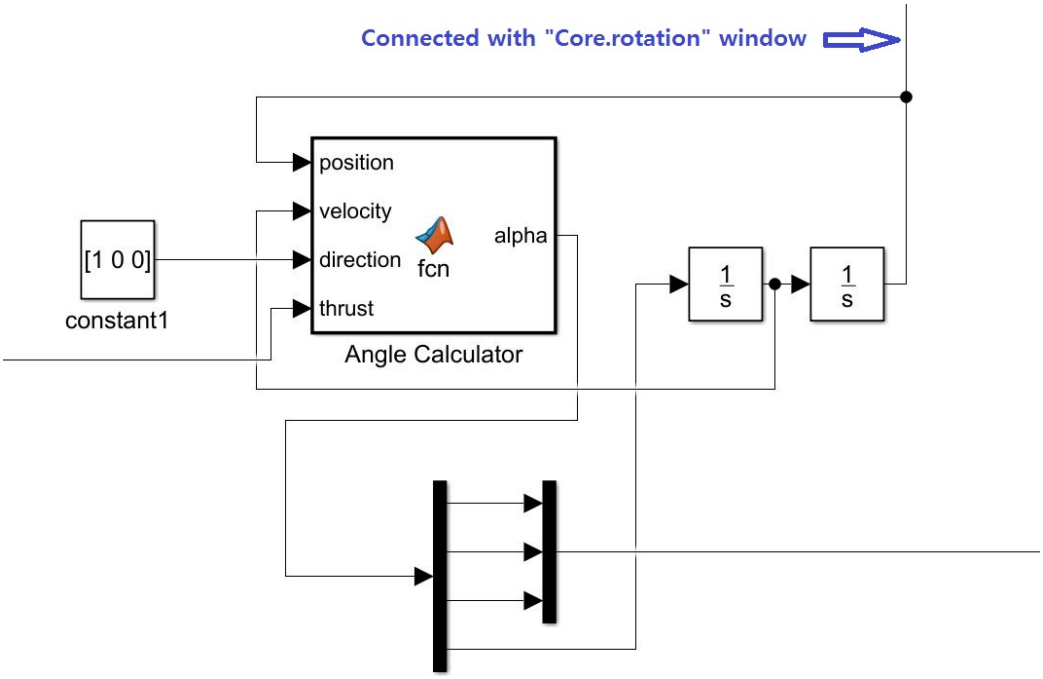

Figure 10: The rotational acceleration calculating system

We assumed that we already have the perfect control method for the spinning mass that is we can let the mass stay at one position as long as we want and we can move it to another position instantly.

For example, suppose the full range of the spinning mass is from 0 degree to 360 degree, we let it stay at 0 degree for a certain period of time and then let

it move to 180 degree position instantly and stay there for a certain period of time and then move back to 0 degree position instantly and so on.

In this way, the overall center of mass will always stay in one plane, shifting back and forth so that we can fully describe it by updating the 4-vector of the rotational state of the core.

Under such assumptions, the rotational acceleration calculating system will use the information of the current state to calculate the rotational state for the next cycle and update it through the state updating window for the rotational state of the core.

In this way, with the help of other blocks and the proper connections, the translational and rotational acceleration calculating systems can work together to generate the updated state information for each geometric component of the 3D quadcopter model in each time cycle so that we can see a continuous, realistic dynamic simulation when we run the program. For the simulation demo videos and more details, please check our Github repo.

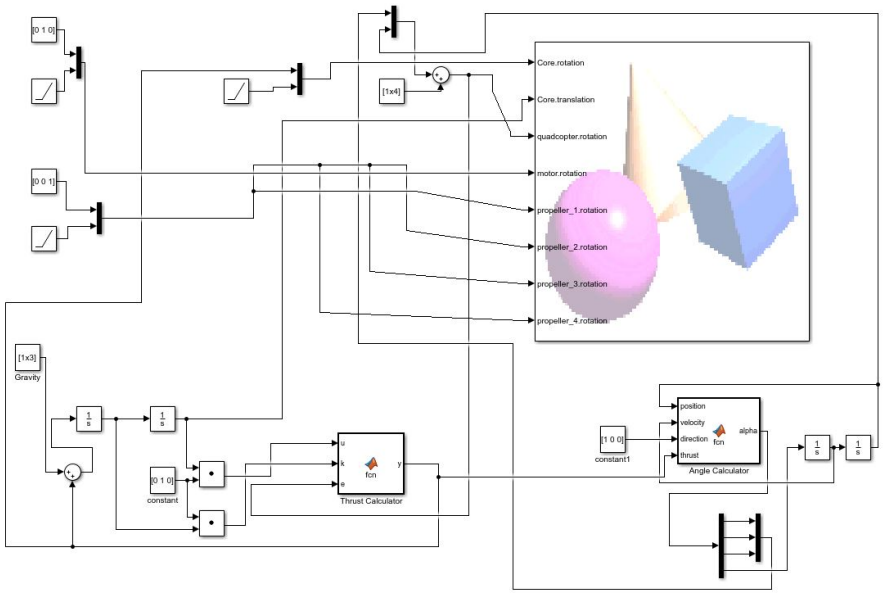

Figure 11: The completed simulink block diagram

3.4 Obtaining parameters and insights

As it can be seen in the simulation demo video, we achieved most of our goals for the simulation part pretty well. We built a decent 3D structure for the quad-copter. Though our knowledge of physics and math is limited to derive the complete solution for the motion, we made assumptions under which we can build a simulation model that is as close to the reality as possible.

And finally, we also expected to get some useful parameters and insights that we can apply in the real implementation. The parameters that are mentioned in this section are obtained experimentally by trying different values one by one and observing the behavior of the quad-copter in the simulation.

The first parameter we were interested in was the spinning rate of the arm. When we set it up to be a very small number, in other words if the mass stays at one position for too long time, the quad-copter gets flipped over very soon after running the simulation.

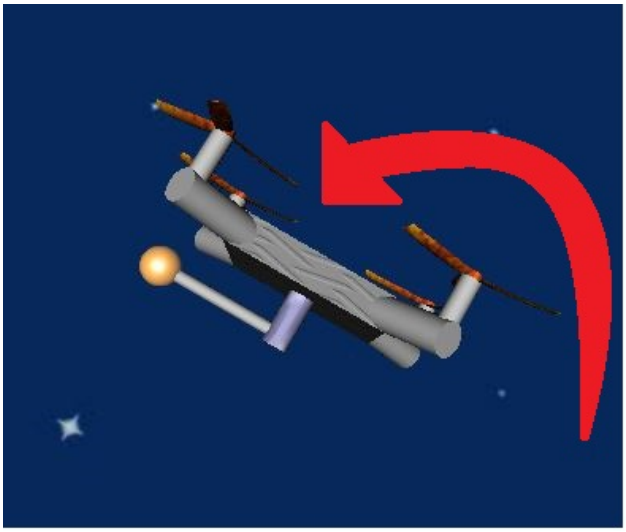

Figure 12: The quad-copter gets flipped over

And when we set it up to be larger and larger number, we noticed that it takes longer and longer until it gets flipped over. And when the rate is going

over about 10 cycles/second, the quad-copter seems to be very stable.

Due to the assumptions we made, the spinning rate value cannot be very reliable in this case, but it tells us the essential insight that until we can get a fast enough spinning rate for the arm, we will never get the quad-copter stable.

The next parameter we were interested in was the pitching angle. The pitching angle here is the angle it pitches by when it's moving towards a certain direction.

We realized that if we set this angle too large, the quad-copter will get flipped easily, and if we set this angle too small, the quad-copter will not move towards a certain direction but wiggle at the same position.

After trying many different angles, we found that the pitching angle that has the best performance is about 22 degree measuring from the vertical axis that is pointing up.

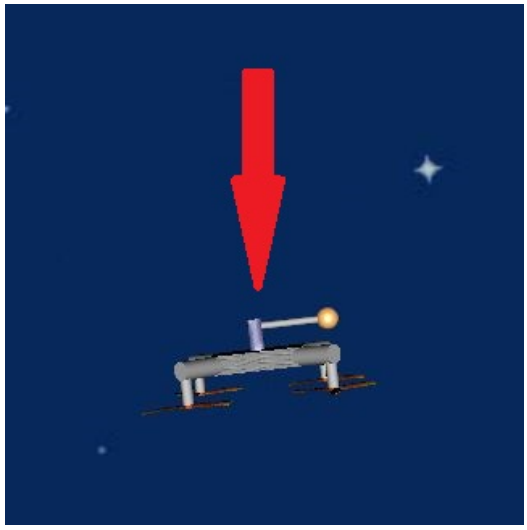

Figure 13: The quad-copter falls down if the parameter is not appropriate

This value we obtained from the simulating experiment is supposed to be more reliable than the spinning rate we mentioned earlier. Because even though we assumed that the mass can move from one place to another in-

stantly, that won't effect on the accuracy of the optimal value of the pitching angle since the pitching direction is in the same plane with the positions that the mass is staying.

It tells us that no matter where we let the mass stay, we should never let the pitching angle go over 22 degree on that specific direction, otherwise it will get flipped over.

Lastly, we expected to know how much thrust force would be enough to maintain a stable movement. We noticed that if the thrust force is too small, it will just keep falling, and if the thrust force is too large, it will either keep going up or get flipped over quickly.

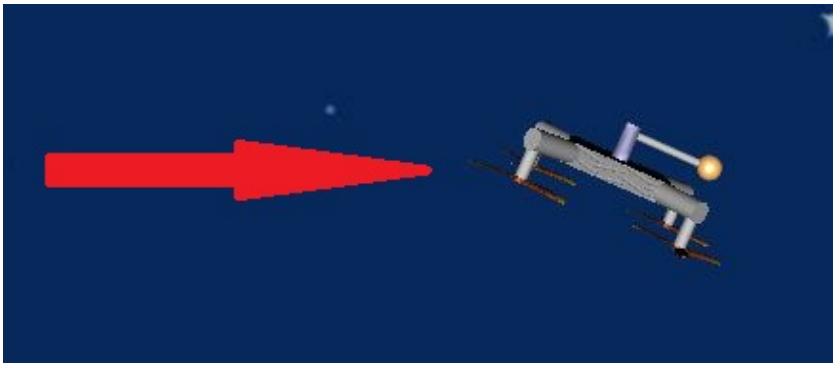

Figure 14: The quad-copter with appropriate parameters

The ideal thrust force we found after trying many possible values was about 1.2G. 1G here represents the force that equals to the gravitational force that comes from the entire quad-copter. So for example, if the weight of the quad-copter is 10g, we need to provide a thrust force that equals to the gravitational force of a 12g object.

Similar with the pitching angle, this parameter for the thrust force also can be considered as a pretty reliable parameter, because again, the thrust force is in the same plane with the positions that the spinning mass is staying. Basically, any parameters that come from the same plane with the spinning mass is supposed to be reliable since their accuracy gets little influence from the assumption we made.

This was confirmed later in our real implementation, we designed the spinning arm using the weight limitation that was calculated based on the 1.2G thrust force parameter we obtained from the simulation. And when we tested it using different load, the quad-copter was behaving almost as the same as what we saw in the simulation.

4 Off center spinning module design

In order to clarify the steps taken in designing the spinning mass module, we start with reviewing some main specifications of the quad-copter used for the purpose in the project.

4.1 Quad-copter Specifications

Crazyflie 2.0 is a 27 gram nano quad-copter used for this project. As for designing our spinning mass module, the main constraint was the maximum payload that Crazyflie could handle. According to Crazyflie 2.0 hardware specifications maximum recommended payload is 15 grams. Therefore the spinning mass module initially assumed to have maximum weight of 15 grams.

Figure 15: Crazyflie 2.0

4.2 Spinning mass considerations

Our design of spinning mass was consisted of selecting below components:

1. DC motor
2. Rotating arm and Spinning mass
3. Rotary encoder

4.2.1 DC motor

As mentioned, mainly constrained by maximum payload for spinning mass module, we needed small, light motor with control feedback. Next important constraint was voltage and current criteria of the motor. Ideally we aimed to power the module from Crazyflie power supply which was 3.7V (250 mAh) battery. Adding external battery was not ideal for our weight limit, so before implementation we tested our candidate motor along with the quad-copter four motors and measured voltage and current drawn by selected motor. There we observed the selected motor would draw around 0.4 A spinning at sutiable range for our goal Consequently we selected the same type of DC motor as those utilized in the quad-copter.

Figure 16: 7mm Core less DC motor

Using the Selected 7x16 mm Core-less DC motor,we estimated the complete system (quad-copter and spinning mass module) to run for 4 minutes. We agreed this amount of time would be enough for demonstrations purposes at this point. Below is the specifications of the DC motor used.

Mechanical Specifications	
Diameter	7.0 mm
Length	16.0 mm
Shaft length	3.5 mm
Shaft diameter	0.8 mm
Weight	2.7g
Wire length	32.0 mm
Electrical Specifications	
Kv	14000 rpm/V
Rated voltage	4.2v
Rated current	1000mA

4.2.2 Rotating arm and Spinning mass

Now we shall explain the design process of the rotating arm. Below is the initial design for rotating arm.

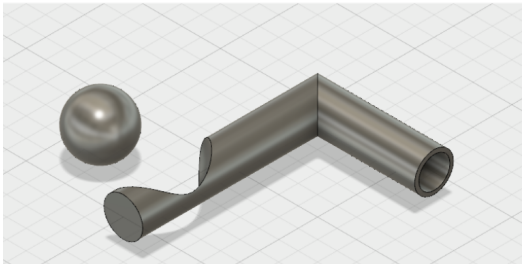

Figure 17: Rotating arm and spinning mass design

The arm length and height, considering the specifications of the quad-copter, were decided to be 5 cm and 3 cm respectively.The diameter of the cylindrical was 10 mm with 2mm thickness on sides to provide slightly larger gap than 7mm(motor diameter) with small hole where the motor can get attached to the rotating arm.
Going back to our weight constraint. Our weight limit for rotating arm along with spinning mass was almost around 12 gram. (Explained in the table below)

Weight constraints	
Maximum payload	15 g
Motor weight	2.7 g
Rotating arm and spinning mass	?

PLA filament with density of 1.25 g/cm^3 was used for printing the rotating arm. The printed arm (shown in figure 7) weighted about 7 grams, which limited our spinning mass weight to under 5 grams.$(12 - 5 = 7grams)$

Figure 18: Rotating arm

For spinning mass we used few different types of bolts (all lighter than 5 grams). Also we thought for final design an ideal case would be to use small precision ball bearings with weight range between 3-5 grams.(shown i figure 8)

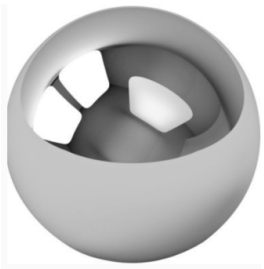

Figure 19: ball bearing for spinning mass

4.2.3 Rotary encoder

In order to have better feedback of the system dynamics and to have more control over the angular velocity of rotating arm, we decided to add off-axis encoder.

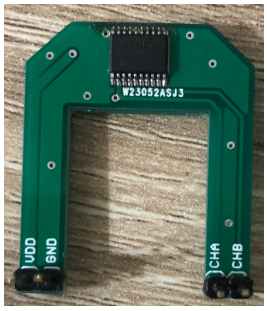

Figure 20: Encoder

The rotary off axis encoder required us to redesign our rotating arm. We added an extra hole to place magnet on rotating arm and Styrofoam to hold our motor and the encoder on top of our quad-copter.

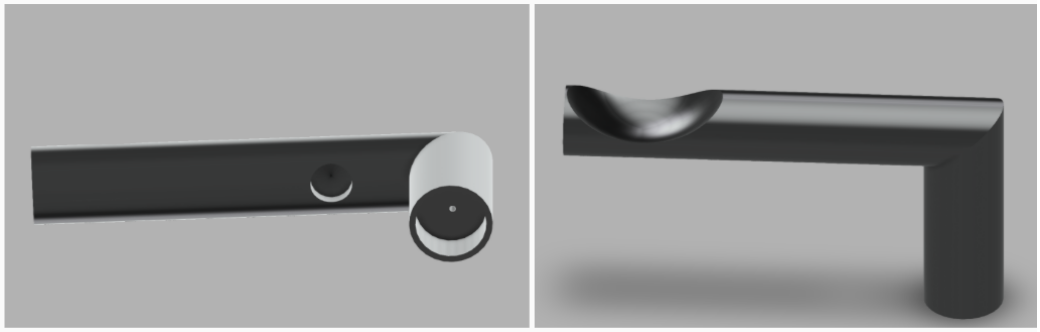

Figure 21: Modified rotating arm

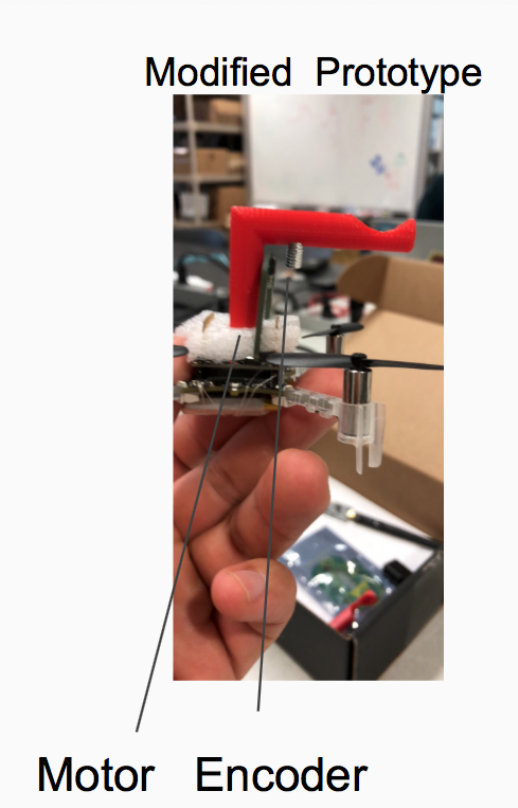

Figure 22: Spinning mass module with encoder on quad-copter

5 Motor controller circuit

Now with rotating arm components defined we move on to motor driver for spinning module controller.

5.1 PCB Design

Initially we decided to design separate PCB that can be mounted on the quad-copter with its individual circuitry. For that purpose we created a list of components we required and made the schematics for our motor controller circuit. (Schematics file shown in figure below is attached with project files)

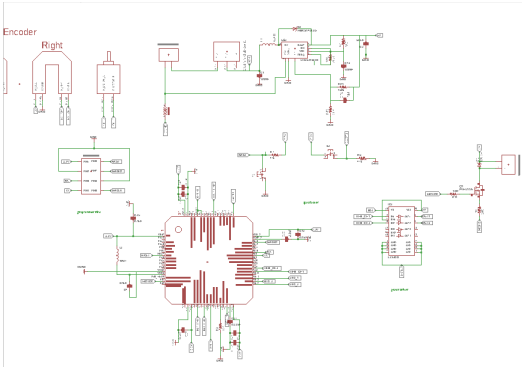

Figure 23: Controller circuit Schematics

Our initial schematics included all below surface mount components:

1. MCU
2. Motor Driver
3. Voltage Regulator
4. DC motor and rotary encoder

5.1.1 MCU

The micro-controller selected is STM32F405, a 32 bit MCU by STM. Possibly a controller with lower power and less computation power would have sufficed since there was no heavy computation in the encoder part of our circuit. However, the reason to select STM32F405 was mainly due to our prior experience with this chip and the fact that it was easy to program with programmable cable we had.

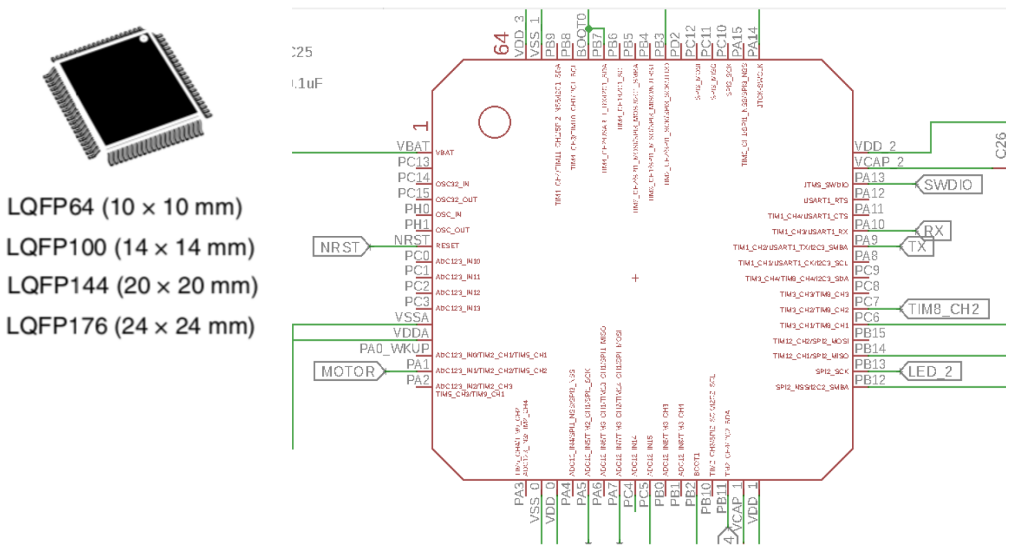

Figure 24: MCU of the motor controller circuit

5.1.2 Motor driver

We designed two motor driver circuits for our spinning mass module.

1. Unidirectional motor driver: Here in this motor driver the SMD N-channel Mosfet "irlml2502trpbf" acts as on/off switch which is ideally controlled by MCU through user's input. This driver given PWM signals was able to run the motor at different speeds only in one direction.

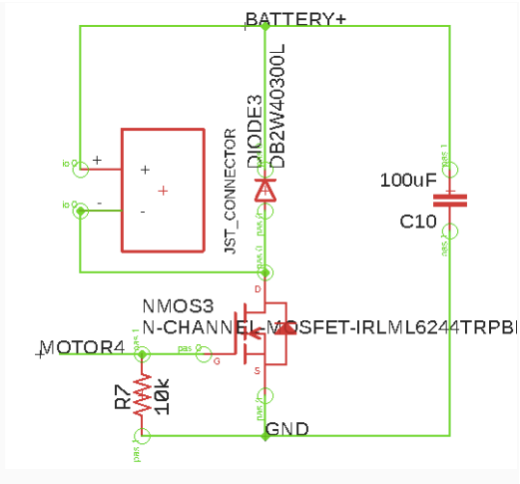

Figure 25: schematics of the unidirectional motor driver

2. Bidirectional driver: Just like the case with encoder, in order to have more robust control over our spinning mass module, we decided to have bidirectional motor driver where moving in opposite directions could help to stabilize the very under-actuated system of our quad-copter. In this setup two separate PWM signals from MUC are input to H-bridge that determine the direction of rotation for spinning mass and its angular velocity. For this purpose, SMD version of L293DD was implemented as H-bridge.

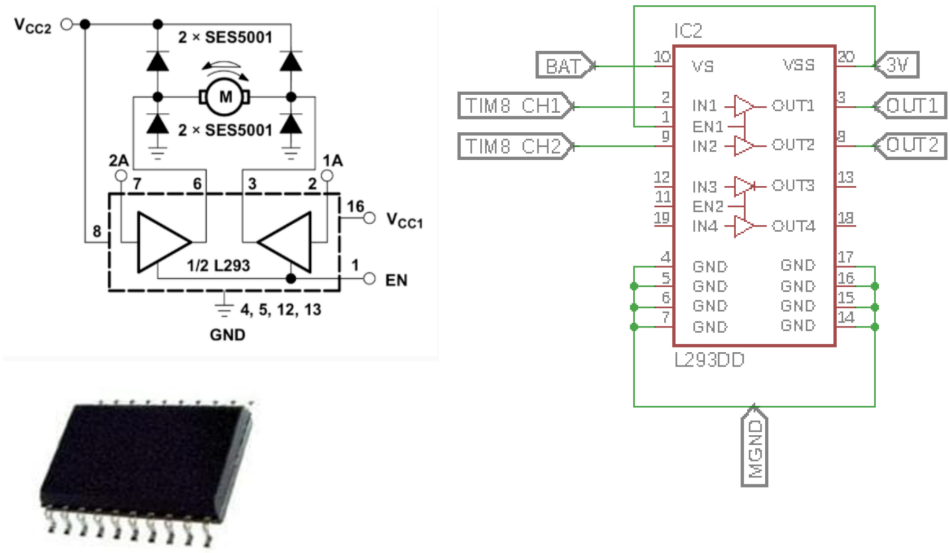

Figure 26: Bidirectional motor driver L293DDD (H-bridge)

5.1.3 Voltage regulator

As shown in bidirectional motor driver schematics, L293DD requires two different voltage level to work properly. Therefore voltage regulator was needed to create the second voltage level required (slightly less than source voltage). The regulator we picked was SMD version of LM2621 which can give out adjustable output in range of 1.2-14 volts.

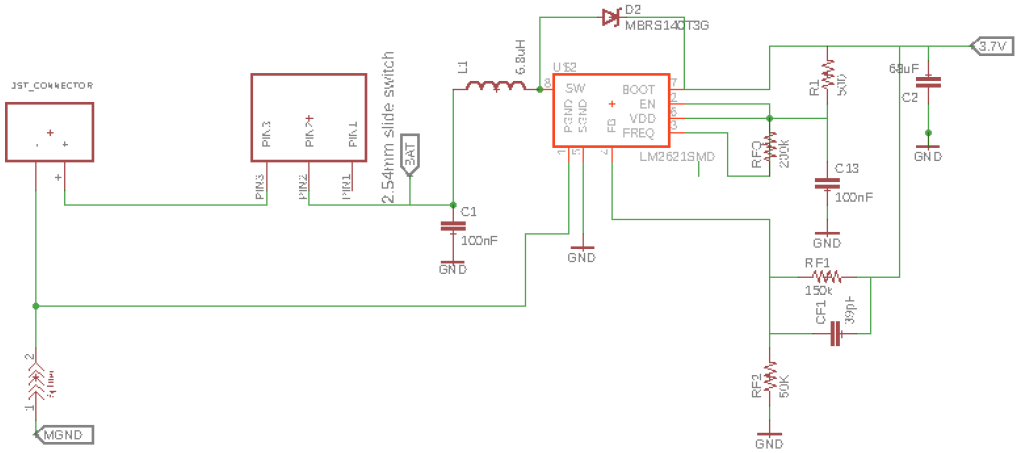

Figure 27: Voltage regulator in motor controller schematics

5.1.4 DC Motor and Encoder

These two components were fully explained in previous section. (Refer to 4.2.1 and 4.2.3)

The remaining components were mostly passive elements such as resistor and capacitors and there were few Schottky diode used for circuit protection purposes.

5.2 Crazyflie quad-copter control board

There were few implementation issues with our separate motor controller circuit.

1. Communication issue: Since the input from user was transmitted to our quad-copter through RF channel, once the connection was setup there was no way to have another connection to our motor driver simultaneously. One way around this issue was to set up another serial communication between motor driver and the MCU of quad-copter. But the delay of the channel could have still been problematic.

2. Too many smd components: The amount of our controller components along with the hard job of debugging SMD soldering and time consuming process of ordering PCB, given the time window we had, made

us to explore more possibilities of implementing more basic controller driver for demonstration purposes.

Studying the schematics of controller board of our quad-copter we realized a lot of components in our design are already considered in in Crazyflie board. Components such as the MCU (STM32 Bit MCU), voltage regulator and ... . We also figured out several GIO pins and analog pins and PWM are left unused on the MCU of our quad-copter.

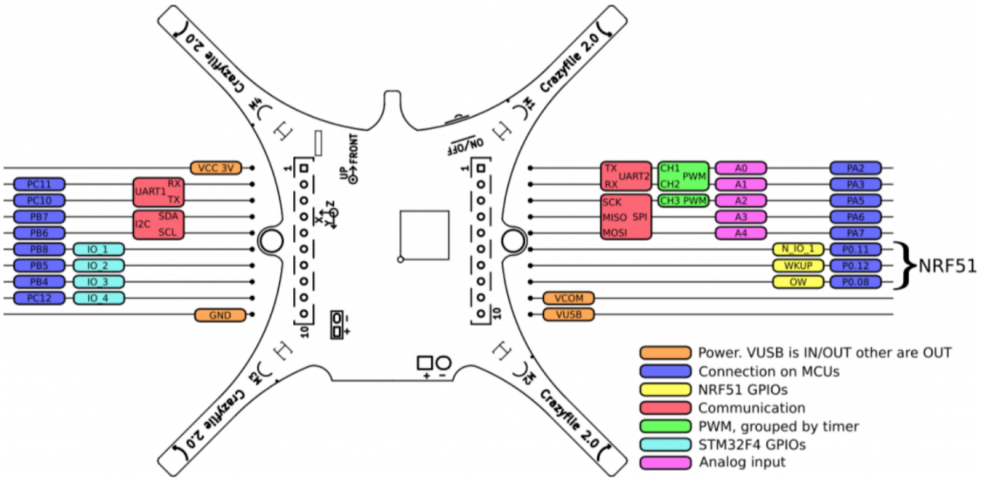

Figure 28: Voltage regulator in motor controller schematics

Therefore in order to avoid the redundancy, fully capture the capabilities of our drone and simplify our controller circuit we did not implement the designed PCB. Instead we only soldered the needed components on pref board and mounted on top our drone. Again for simplicity we picked the unidirectional motor driver since it had less components and we only had small space of 9 cm x 9 cm grid on our drone. Now that we did not have PCB it was easier to have less components to deal with.

Some challenges here were how to stabilize and firmly hold the spinning module on top of drone as well as how to power the the motor controller circuit. As mentioned before, We had already tested powering all 5 motors from our single power source (3.7 V battery), so knowing the fact that this structure would work we added extra parallel connector from battery to motor controller circuit to power the motor of spinning mass and our drone at the same time. Also in order to hold the spinning mass module in place we

used Styrofoam which was able to hold the module on top of the quad-copter firmly.

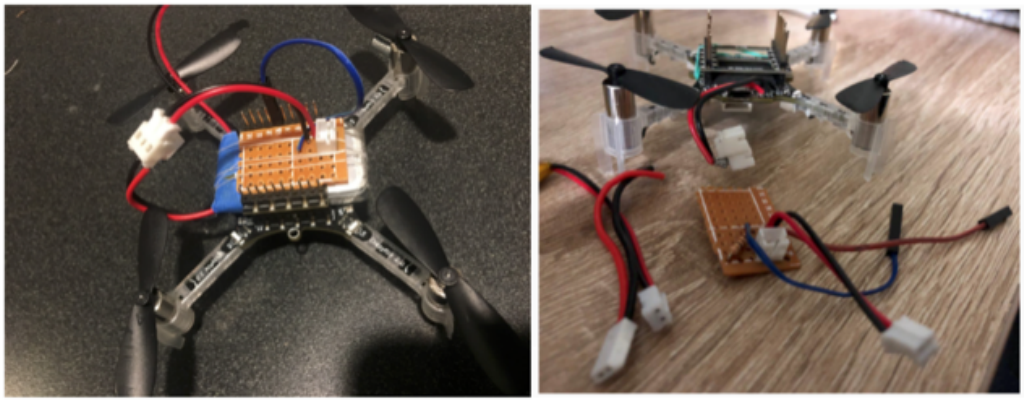

Figure 29: Parallel jst-connector added to power DC motor of the spinning mass module

6 System Dynamics

The ultimate goal of this project was mainly tied to this part. Where the final outcome was supposed to be a dynamically controllable quad-copter considering the spinning mass module and given all the previously mentioned assumptions.
Throughout the course of the project this key section turned to be a very complicated control theory problem that even if the right mathematical model approach was given, it would still require a well built hardware and functional accurate software in order to work out. In the time frame of 10 weeks we made progress in each of these three main sections, however none became fully completed. As for mathematical derivation and hardware, our approach is already explained. In this part we briefly go over some challenges we faced in software.
First of all we were not successful in hacking the quad-copter firmware to map user's instant input to our spinning mass module. So next stop we tried to statically upload code to the quad-copter and compare the outcome to expected behavior. As mentioned before one assumption in motor controller circuit was the Crazyflie's MCU is easily programmable so that we we could upload some specific spinning patterns using PWM signals to our system.

However it turned out to be much more complicated, than expected, to program the MCU. The code was written in very low level C, which made it hard to understand and very time consuming to upload.
The final flight testing included various unsuccessful attempts where PWM signal was set to high all time and the goal was only to get as close as possible to model a stable flight pattern without making any turns.

7 Motor Control and Models

7.1 Introduction

7.1.1 Background

Due to the constraints of the project design, we are opting to use what motors we have available. These are the same motors that came with the CrazyFlie. Simple specs can be found here. *Note: no actual spec sheet exists for this component, very little data exists for the component* Furthermore, in order to create the proper dynamic model for the Spinning Mass Actuator, it was necessary to first develop the DC Motor model and control implementations. Once this was complete, then I could begin finalizing a realistic model. The idea was to work from the ground up adding more and more complexities.

7.1.2 Theory

The most important aspect of the spinning mass drone project is the motor control. Comparisons between the types of motors we can use (Stepper, servo, DC). Here I present the calculations of a DC motor, the first system modeling is on the motor speed control. The theory is we can determine the speed and the location of the mass using a sinusoidal phase. The second system explores the system modeling with an emphasis on the motor position. A stepper motor with an encoder would allow us both models, but we cannot use one.So we'll begin with a DC motor and no encoder, then improve the design with the implementation of an encoder.

7.1.3 Motor Comparisons

1. Permanent Magnet DC Motor
   Great starting torque, good speed regulation. However, torque is lim-

ited.

2. Series DC Motor
   Field is wound with few turns of a large wire carrying full current. Create large starting torque. No speed regulation and prone to damage when run with load. Bad for variable speed drive applications. The speed may vary widely between no load and full load applications. These cannot be used in situations where constant speed is required under varying loads. This problem shouldn't affect us. Also, if run without a load, the speed can increase enough to damage the motor.

3. Shunt DC Motors
   Great speed regulation due to shunt field can be excited separate from current windings (due to parallel connections).

4. Compound DC Motors
   Separately excited shunt field. Good starting torque but prone to control problems during variable speed drive applications. This motor has better torque and speed regulations than a shunt.

5. Stepper Motor
   Come in a wide range of angular resolution. Steppers don't require encoders, can accurately move between their poles. Open loop (without encoder), close loop (with encoder). A single rotation requires more current exchanges through the windings. Torque degradation at higher speeds is observed. Traditional Stepper motors operate in an open loop constant current mode. Due to this design, energy is conserved due to the lack of an encoder. However, due to this operation, a significant amount of heat is created. With the addition of an encoder, can be controlled with a closed loop model. Excellent for speeds less than 2,000 RPM and for low to medium acceleration rates and hight holding torque.

6. Servo Motor
   Requires an encoder to keep track of their position. Using the encoder, servos can measure the difference between the motor encoder and the commanded position (closed loop). Supplies the motor with the current required to move or hold the hold. Excellent for situations requiring speeds greater than 2,000 RPM and for high torque at high speeds.

7.2 Physical Representations

7.2.1 Schematics

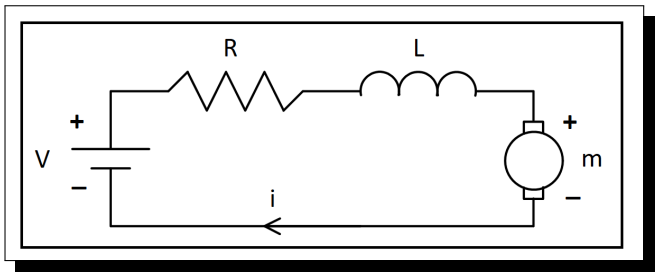

Figure 30: DC motor schematics

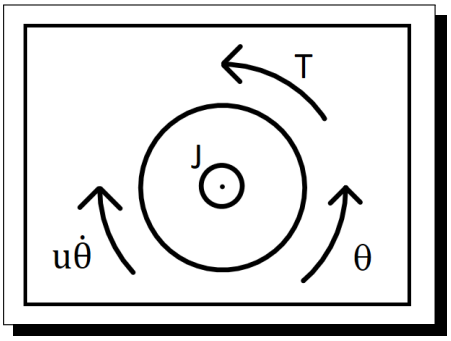

Figure 31: DC motor rotor forces

7.2.2 Assumptions

The assumptions made are relatively the same for both models. We assume that the input of the system will be the voltage source (V). In the speed control model, we assume the output is the rotational speed of the rotor shaft ($\dot{\theta}$), depicted in Figure 31. All components are rigid. The friction torque is proportional to the rotor angular velocity. We assume the magnetic field permeated is constant. Similarly for the Position model, the assumption of the about now includes the rotational displacement as one of the outputs.

7.2.3 Parameters and Components

Symbol	Name	Data	Units
V	Voltage Source	3.7v lipo battery	V
R	Resistor	10	Ω
L	Inductor	1	H
m	DC Motor	7mm brushed	-
i	Electric Current	-	A
T	Motor Torque	-	$[N \cdot m]$
J	Rotor Moment of Inertia	0.1	$[kg \cdot m^2]$
θ	Angular Displacement	-	rad
$\dot{\theta}$	Angular Velocity	-	$[\frac{rad}{sec}]$
u	Friction Coefficient	0.1	$[N \cdot m \cdot s]$
K_i	Motor Constant Coefficient	0.1	$[\frac{N \cdot m}{A}]$
E_b	Motor back EMF	-	$[\frac{N \cdot m}{A}]$

7.3 Speed Control Derivations

7.3.1 Speed Equations

The torque generated by a DC motor is proportional to the current and the magnetic field strength. For simplicity we assume the magnetic field is constant. This further simplifies our model by making the torque proportional only to the current. The torque equation is

$$T = K_i \psi i \tag{18}$$

where ψ, is the flux, and the K_i is a constant factor. Because of our constant field assumption, this equation becomes

$$T = K_i i \tag{19}$$

The back emf (Electromotive Force) is governed by the following equation

$$m = E_b = K_i \dot{\theta} \tag{20}$$

*** Note: The Constant Coefficients aren't necessarily the same, but due to our assumptions and their SI units, we can consider them equal. ***

Looking at Figure 30, we can apply Kirchoff's Voltage Law to derive the following system relationship

$$L\frac{di}{dt} + Ri + K_i\dot{\theta} = V \tag{21}$$

Applying Newton's Second Law on Figure 31 we derive the following equation

$$J\ddot{\theta} + u\dot{\theta} = K_i i \tag{22}$$

Applying the Laplace Transform to equations (4) and (5) we obtain

$$LI(s)s + RI(s) + K_i\Theta(s)s = V(s) \tag{23}$$

$$J\Theta(s)s^2 + u\Theta(s)s = K_i I(s) \tag{24}$$

We want to consider the Voltage as our only input thus we solve using I(s) and arrive at the Open Loop Transfer Function.

$$\frac{s\Theta(s)}{V(s)} = \frac{K_i}{JLs^2 + JRs + uLs + uR + K_i^2} \tag{25}$$

which can be simplified into this form

$$H(s) = \frac{\dot{\Theta(s)}}{V(s)} = \frac{K_i}{(Js+u)(Ls+R) + K_i^2} \qquad [\frac{rad}{V \cdot sec}] \tag{26}$$

Now, we can derive our state space equations. From the transfer function we can form our state space representation of the system

$$\frac{d}{dt}\begin{bmatrix}\dot{\theta} \\ i\end{bmatrix} = \begin{bmatrix}-\frac{u}{J} & \frac{K_i}{J} \\ -\frac{K_i}{L} & -\frac{R}{L}\end{bmatrix}\begin{bmatrix}\dot{\theta} \\ i\end{bmatrix} + \begin{bmatrix}0 \\ L^{-1}\end{bmatrix} V \tag{27}$$

and

$$z = \begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix}\dot{\theta} \\ i\end{bmatrix} \tag{28}$$

7.3.2 Speed MATLAB

*** Note: MATLAB 2017b can sometimes crash or take a long time to compile when using the syms function, especially with a lot of variables. *** We use MATLAB for our purposes of simulation due to the comprehensive tools and applets provided by the software for the purpose of simulation. Python is a great alternative, however, there is greater documentation available for MATLAB.

```
1   % Speed TF S
2   R = 10;
3   L = 1;
4   J = 0.1;
5   K = 0.1;
6   u = 0.1;
7   s = tf('s');
8   H_motor = K / ((J*s + u ) * (L*s + R) + K^2);
9   display(H_motor);
10
11  linearSystemAnalyzer('step', H_motor, 0:0.1:10);
```

Figure 32: This is the code snippet for the transfer function in matlab

The output for the code in Figure 32 should look like this

```
>> matlab_DC_speed

H_motor =

           0.1
  ----------------------
  0.1 s^2 + 1.1 s + 1.01

Continuous-time transfer function.
```

A box should appear after a loading bar sequence finishes. The results appear as follows

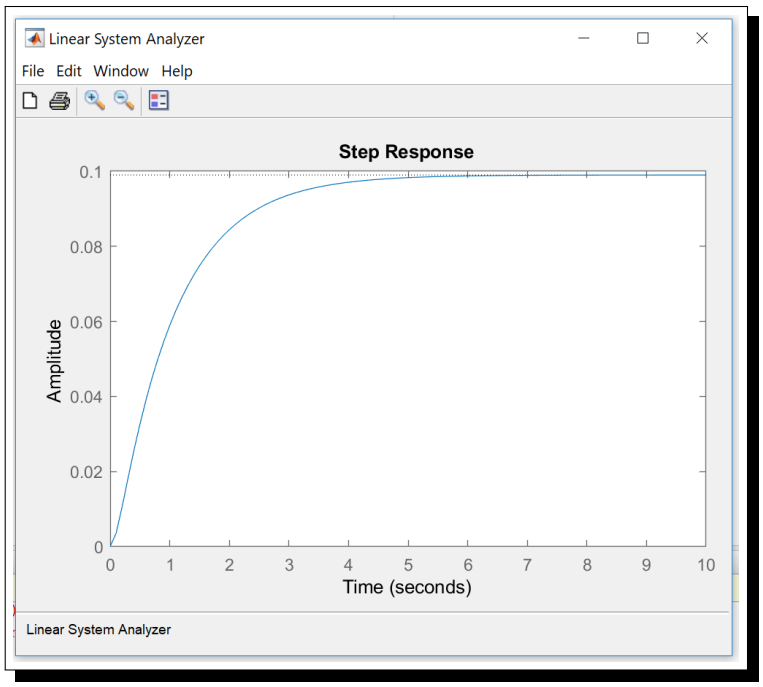

Figure 33: This is the result of the Linear System Analyzer function

7.3.3 Speed Simulink

We will model the system analyzing the forces and torques on the DC motor rotor. To begin you have to model the integrals of our current and displacement. We can derive these models from the equations we formulated earlier. The models for equations (6) and (7) are

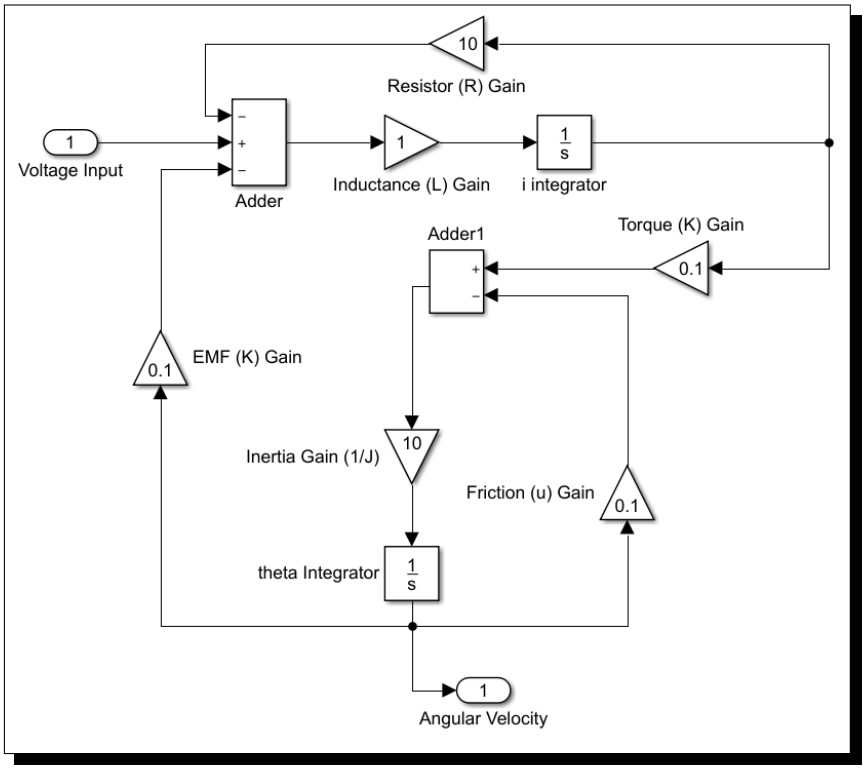

Figure 34: Simulink representation of the DC Motor model

To run any form of simulation, you have to modify the input and output signals of the model. We do this by right clicking the line between the input and the model then selecting "Linear Analysis Points" and selecting "Open Loop Input" for the input and "Open Loop Output" for the output

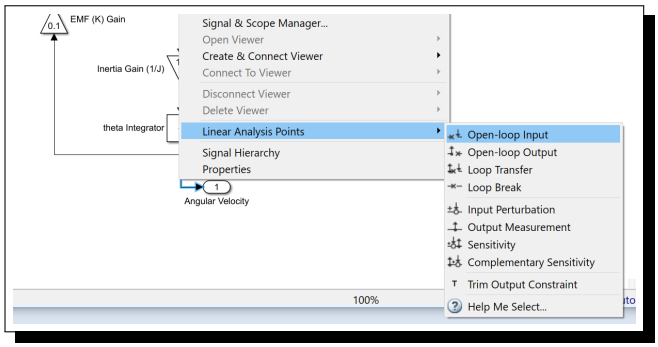

Figure 35: necessary adjustments to the signals

We have to ensure our Simulink (Figure 34) model matches up with our mathematical model we derived earlier (equations (6) and (7)). We have already performed the first part of this process and that was plotting the Linear System Analysis in response to a step function. Next, in the Simulink project, we click on Analysis on the top and proceed to Linear Analysis. This prompts the Linear Analysis Tool, in the top right, we select the "Step" plot. We should see this

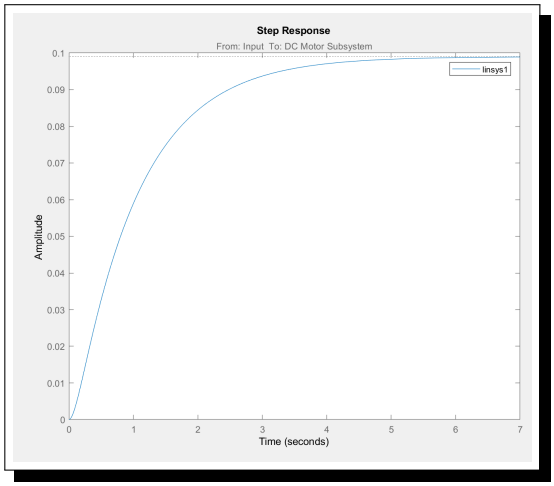

Figure 36: Linear Analysis Tool, system response to a step input

Comparing Figure 36 with Figure 33 we can see that the mathematical model step response matches the Simulink model step response. This confirms our models are accurate.

7.3.4 PID Design

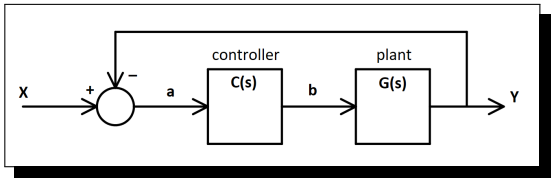

Figure 37: Control System Model

The structure of the Control system in Figure 37 is based on our derived mathematical equations. A PID is a three term controller, it is continuously

calculating an error value (in our case a) as the difference between our desired value and the measured value.The three terms are a proportional term, an integral term and a derivative term. Together we get the equation

$$c(t) = K_p a(t) + K_i \int_0^t a(\tau)d\tau + K_d \frac{da(t)}{dt} \tag{29}$$

The Transfer Function form is

$$C(s) = K_p + \frac{K_i}{s} + K_d s \tag{30}$$

Where K_p, K_i, and K_d are constant coefficients of the controller. *** NOTE: K_i is not to be confused with the K_i from our mathematical derivations. *** the values for your PID controller coefficients are largely dependent on the values you set for your parameters. Variations exist, and tuning is necessary for an optimal controller.

```
14 -   Kp = 250;
15 -   Ki = 200;
16 -   Kd = 20;
17 -   C = pid(Kp,Ki,Kd);
18 -   sys_dc_s = feedback(C*H_motor,1);
19 -   step(sys_dc_s, 0:0.01:4)
20 -   grid
21 -   title("DC Motor PID")
```

Figure 38: This is the code snippet for the PID controller in matlab

After tuning my values, I found $K_p = 250$, $K_i = 200$, and $K_d = 20$ where fairly good. An analysis if they're reasonable will have to be performed later. The Figure below demonstrates what the graph looks like after before and after tuning.

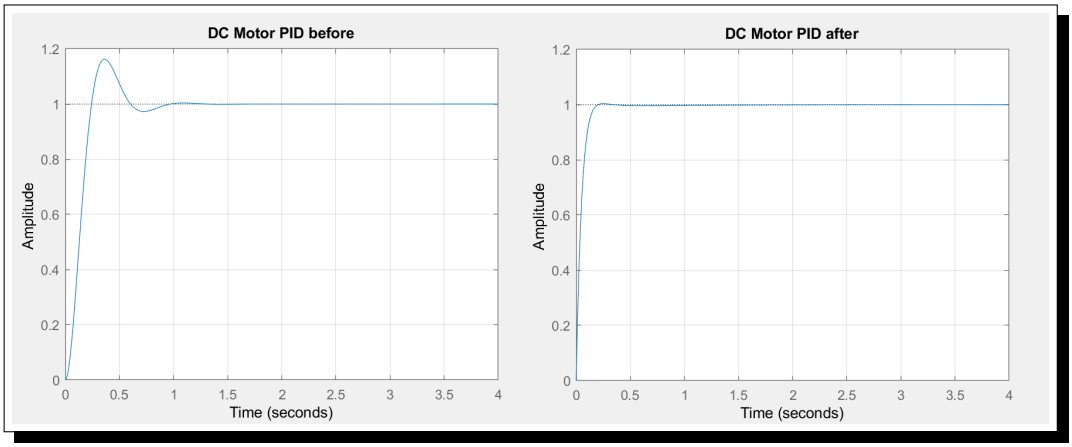

Figure 39: This is the code snippet for the PID controller in matlab

7.4 Position Control Derivations

7.4.1 Position Equations

The position equations are almost identical to the speed equations we derived before. The prominent change in this model is our assumption of the output variable, it is now our angular displacement. This directly affects out state space estimation equation and our transfer function.

Applying the Laplace Transform to the equations we obtain

$$LI(s)s + RI(s) + K_i\Theta(s)s = V(s) \tag{31}$$

$$J\Theta(s)s^2 + u\Theta(s)s = K_iI(s) \tag{32}$$

We want to consider the Voltage as our only input thus we solve using I(s) and arrive at the Open Loop Transfer Function.

$$\frac{\Theta(s)}{V(s)} = \frac{K_i}{s(JLs^2 + JRs + uLs + uR + K_i^2)} \tag{33}$$

which can be simplified into this form

$$H(s) = \frac{\Theta(s)}{V(s)} = \frac{K_i}{s((Js+u)(Ls+R) + K_i^2)} \qquad [\frac{rad}{V}] \tag{34}$$

Now, we can derive our state space equations. From the transfer function we can form our state space representation of the system

$$\frac{d}{dt}\begin{bmatrix}\theta\\ \dot{\theta}\\ i\end{bmatrix} = \begin{bmatrix}0 & 1 & 0\\ 0 & -\frac{u}{J} & \frac{K_i}{J}\\ 0 & -\frac{K_i}{L} & -\frac{R}{L}\end{bmatrix}\begin{bmatrix}\theta\\ \dot{\theta}\\ i\end{bmatrix} + \begin{bmatrix}0\\ 0\\ L^{-1}\end{bmatrix} V \tag{35}$$

and

$$z = \begin{bmatrix}1 & 0 & 0\end{bmatrix}\begin{bmatrix}\theta\\ \dot{\theta}\\ i\end{bmatrix} \tag{36}$$

7.4.2 Position MATLAB

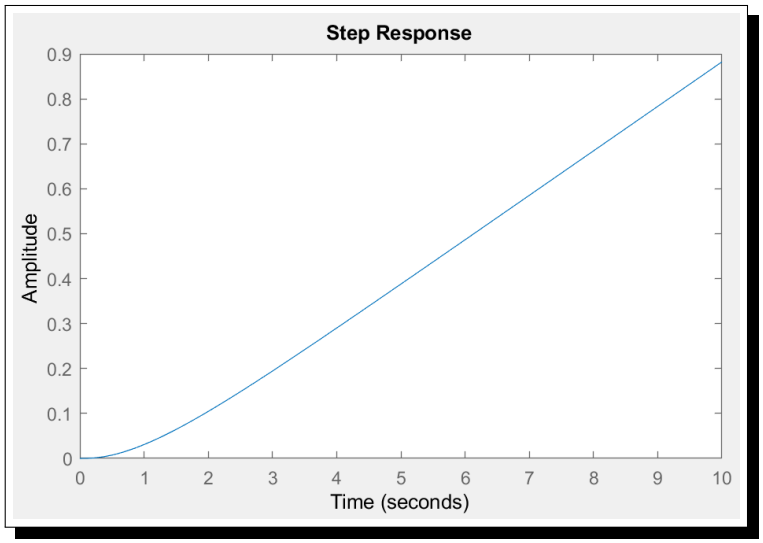

Figure 40: Transfer Function response to the Linear System Analysis function

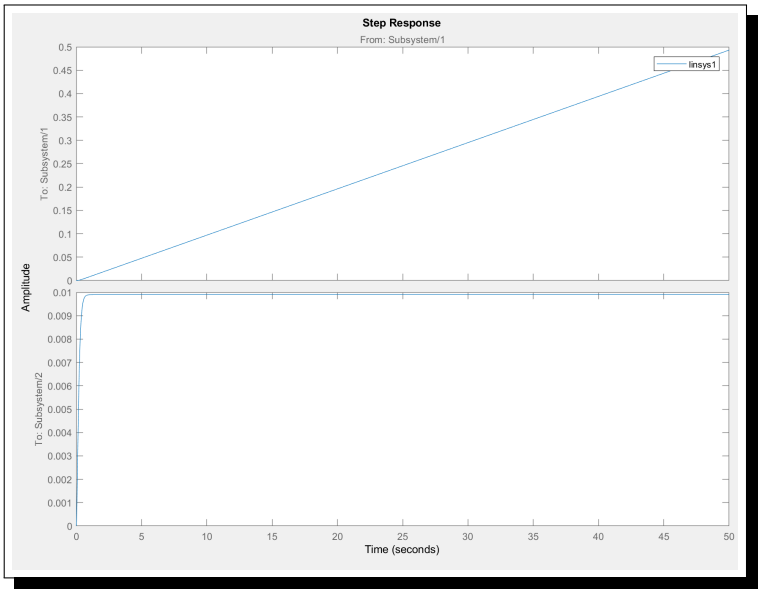

Figure 41: Simulink Model Analysis using the Linear Analysis Tool

The graph in Figure 40 matches the upper graph in Figure 41 thus the simulink model matches the mathematical simulation model.

7.4.3 Position Simulink

This (Figure 42) is largely similar to the position Simulink model, the difference being the output is now an angular displacement. Meaning, our original output goes through an integrator block.

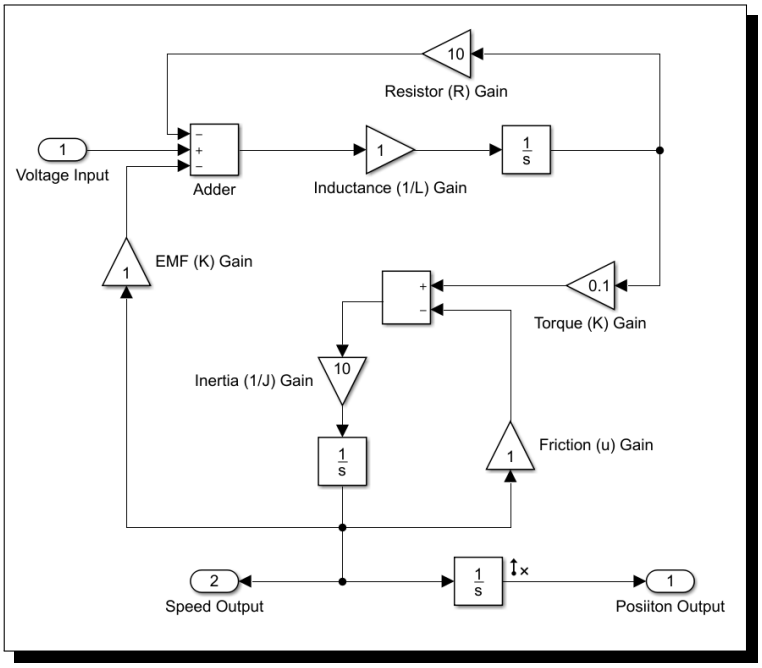

Figure 42: Simulink Model Analysis using the Linear Analysis Tool

7.5 PID Design

This process is similar to our previous PID implementation.

$$c(t) = K_p a(t) + K_i \int_0^t a(\tau)d\tau + K_d \frac{da(t)}{dt} \tag{37}$$

The Transfer Function form is

$$C(s) = K_p + \frac{K_i}{s} + K_d s \tag{38}$$

Where K_p, K_i, and K_d are constant coefficients of the controller. *** NOTE: K_i is not to be confused with the K_i from our mathematical derivations. *** the values for your PID controller coefficients are largely dependent on the values you set for your parameters. Variations exist, and tuning is necessary for an optimal controller.

7.6 Dynamic Model

This section of the research is largely incomplete. The intention was to work from the ground up, beginning this portion after having completed the Motor modeling. Due to time constraints we could not finish this.

7.6.1 Physical Representation

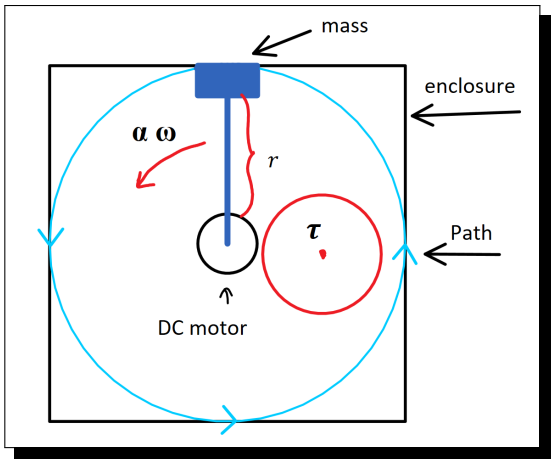

Figure 43: unfinished Top View of OCSM Dynamic Model

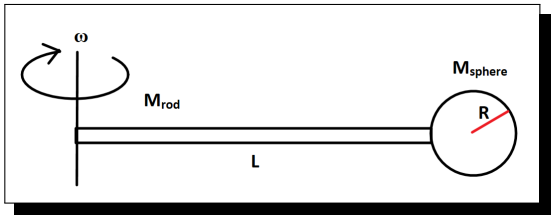

Figure 44: unfinished Side View of OCSM Dynamic Model

From Figure 44 we can derive the following equations of inertia. I feel that it is crucial in the representation of the model that this be as correct as possible. Thus my assumptions do not exclude the mass of the rod. Also, I felt it was necessary to use the Parallel Axis Theorem.

$$I = \frac{1}{3}M_{rod}L^2 + \frac{2}{5}M_{sphere}R^2 + M_{sphere}(L+R)^2 \tag{39}$$

7.7 3D Simulation

The 3D simulation demonstrates the control of both the position and speed. The model can be seen in

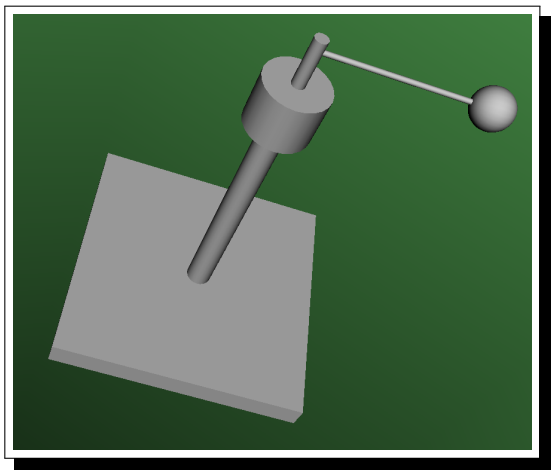

Figure 45: 3D model of the motor and OCSM module

8 Results

Using Newton's fundamental law of physics, we are able to derive systems of equations that characterize the dynamics of our system. However, due to the complexity introduced by rotational acceleration, even in quaternion, no close-solution in the form of $\dot{s} = f(s)$ is analytically derived. Moreover, solving the equations in the form $f(\dot{s}, s)$ numerically in Matlab yields no solutions.

As the exact Mathematical model we derived yields no fruitful results, we turned to building a simulation environment using Matlab's toolbox, in the hope that we can get an idea of our system dynamics in the simulation.

We have built a decent 3D model for the quandcopter and then we have generated several dynamic 3D simulation. Due to the limitation of our current math model, we had to make assumptions that the thrust force is always applied perfectly at the center of the core of the quadcopter and the quadcopter can only pitch inside a single plane. Theoretically, that could be possible if we have a perfectly assembled quadcopter and perfect controlling motor. Under these assumptions that we have ideal hardware and software, the simulation does a pretty good job.

We have obtained three major insights and parameters from the simulation. First,the spinning rate should be closer to about 10 rounds per second, otherwise it will easily get flipped over. Due to the assumptions we made, the accuracy of this value would be low. Second, the pitching angle for the movement should be around 22 degree for our quadcopter otherwise it won't move efficiently. This value is much more reliable because our assumption doesn't directly decrease this parameter's accuracy since the pitching direction is inside the same plane with the movement direction vector. Finally, the thrust force should be about 1.2 times bigger than the gravitational force of our quadcopter. This parameter is also very reliable since the thrust force is also inside the same plane with the movement direction vector and it's confirmed to be pretty accurate when we did the real world tests.

Results of the motor analysis are conclusive. As shown by the data collected from the linear analysis function in matlab and the Simlink linear anaylsis tool, the models are a perfect match. This is good because this is our base case model. Given the total drone derived model. If we were to simplify heavily, we could ultimately result in this DC motor model. We still need empirical data on the various types of motors available. We could not procure all the motors or tests we needed to perform due to the time and budget constraints. Lastly, the incomplete mathematical model meant we would have to do real time experiments with the motors in order to understand the behavior. With the support of our theoretical results, we started designing our spinning mass module, followed by implementing motor driver circuit and hacking the crazyflie. The final outcome was supposed to be a dynamically controllable quad-copter however we modified the final prototype as we were not able to obtain a functional math model for full implementation of we started with. Also our hardware and software issues limited the

scope of our project for the short given period of time.
Other issue to be addressed here, was our planning task. By getting to this part of the project, we realized due to some bad assumptions we made the path of our progress got deviated few times during past ten weeks. This could be meanly due to having wrong priorities. For example at the very beginning we started by setting up the joystick controller for our drone without considering whether we could map the user's input to our mass module in the time frame given, which turned out to be very hard and we could not complete it. Or for instance we stared configuring our rotary encoder before testing our spinning mass module on the quad-copter. Again with the wrong priority order, we spent time on secondary goals while our primary targets were way further to accomplished. We also learned about the design process of projects in this scope. The fact that your given problem is not always nice and smooth linear system with straightforward solution. We thought we should wait to finish the theoretical modeling first then move to building the module. While having a mathematical model is definitely necessary for such problems, on site testing in last two weeks showed us sometimes it is better to take reverse engineering approach. Visualizing the natural behavior of the system can bring us some useful insights that are not quite obvious or even reachable through theoretical equation deriving. Had we had another chance to start this project we would have started working on physical modeling from first few weeks as we realized theoretical modeling and actual implementation would work better if applied in parallel way.
Finally working on this project brought us some of very fundamental concepts, that we had learned before but this time facing them in an interesting unique way. Newton's third law, for instance, was not something new. However the understanding of how to apply it for a flying vehicle and its important to derive the mathematical equations of such system was like learning something new from scratch. Similarly, we had heard multiple times about pull down resistor in different classes but it was in this project that due to logic error in our initial motor circuit we learned again about pull down resistor. This time however we added it to our circuit and fully saw the results of adding such resistor in the behavior of our circuit. This aspect of our project can be considered as our interesting challenges where most of our learnings from this project happened. The part that was motivating and drove us forward.

9 Further Work

A few simple special cases can be used to test the validity of the Mathematical model. Specifically, setting L_{mx}, L_{mz}, and m_C to be 0 should reduce our system to a mere quad-copter with constant thrust. We could verify our equations if our system do behaves like such. Other special cases can also be tested such as setting gravity or thrust to zero and verify with our intuition.

To improve our model on simulation, we need to keep developing our math model. The current simulation model reaches its limitation because our current math model is not able to add efficient details onto the simulation anymore. If we can successfully derive the complete solution of the motion that would be the best scenario but there could also be other approaches.

One of the possible approaches could be adding more rotating axis. Two axis would look better than one axis, three axis would be even better than two axis. If we add many rotating axis and add the same amount of rotational acceleration calculator blocks, we may calculate the quaternion angle changes for each of the rotating axis and combine them together to generate the next frame of 3D graphic.

It's not a perfect method but it's definitely doable and if we add enough axis, after a certain point, it will be little difference between our result and the result from the completed solution. It's like doing integration using Riemann Sum method. If we can't derive the integration formula for a certain function, we can just try to find the Riemann Sum for the area under the function curve. And the more pieces we divide the function curve into, the more accurate result we can get from this method. Similarly, we can get more accurate result by adding more rotating axis in our case.

One of the challenges we faced when working on this project was determining which motor we could successfully use to control the Off-Center Spinning-Mass actuator. An ideal scenario would allow us to use an encoder with either a Servo motor or a Stepper motor. We would need to take empirical data to justify which of the two is necessary. We would suggest working on a medium to larger sized drone because the properties of the motor such as size, weight, and current draw can heavily affect the drone's performance. We were limited to the CrazyFlie provided by Professor Mehta and Nate the Teaching Assistant. The CrazyFlie is an ideal candidate for the project due to the open source nature of the drone. However, the biggest limitation was the CrazyFlie's size and load capacity. This meant we were restricted to using the 7mm brushed DC motor that came with the CrazyFlie and an

encoder of our selection.

Further work must be done on the OCSM (Off Center Spinning Mass) module dynamics. we were not able to complete this task, by the time I returned to this task, we had spent a lot of time reworking, retooling, and refocusing different aspects of the project. In order to continue this work, it is suggested that the dynamic model be based on the derived DC motor system, the state space equations are derived, it is now only a matter of making the model three dimensional. From there, we can simulate the characteristics of the OCSM without the complexities of the drone. A simple camera tripod with a ball joint head attachment and the OCSM connected to a computer is sufficient to test this. Once this system works, you may begin including the drone model to the system (or whichever vertical take off landing vehicle you choose). It is crucial that empirical and simulated data be measured through every step in the process. The DC motor model I derived has dummy values, in order to use real values, you must take lab data from the motor you're using. Lastly, the Stepper or Servo model may need to be derived again. This is something I wasn't able to accomplish in time.

Considering all above cases, these are mainly the modifications can be taken into account for each subsection of this enormous project. While it is true that each minor part can be extended, modified and obviously improved it is important to note whether there is not a requirement for building the physical prototype, it is better to begin with the theory behind this project. Even if our math equations led us to a solution there was not any solid way to prove our model in our current case. Whereas considering this project in simulation environment only, would allow us to ignore as many physical effects as needed in order to simplify the system and check for correctness of our model. Once a robust functional model for simpler cases obtained we shall continue with adding more complexity to our system.

10 Conclusion

Researchers can look into our work if they have similar ideas as our own. We took several different approaches when facing this control theory problem. The problems we faced and the solutions we created can provide wisdom and experimental knowledge to anyone interested in picking up where we left off. We recommend that future researchers plan better, and execute these plans more effectively. Simply brainstorming several ideas and approaches and

thoroughly working them out on paper before actually implementing them may be beneficial because some unforeseen scenarios may reveal themselves. This is crucial because it can save time and money. We also recommend future researchers read the appropriate literature before jumping into any task. We found it helpful that we read several published works on similar problems. Lastly, under actuated robotics which take advantage of the system dynamics is a very difficult task. A good approach is to begin with a simplified model, make it work in a way that returns defined and expected behavior. Then increase the complexity.

11 Reference

- Ecircuitcenter.com. (2018). DC Motor Model. [online] Available at: http://www.ecircuitcenter.com/Circuits/dc_motor_model/DCmotor_model.htm [Accessed 16 Jun. 2018].
- https://forum.bitcraze.io
- https://www.bitcraze.io/getting-started-with-the-crazyflie-2-0/
- https://www.amci.com/industrial-automation-resources/plc-automation-tutorials stepper-vs-servo/
- https://www.mdtmag.com/blog/2015/09/dc-motor
- https://en.wikipedia.org/wiki/Motor_constants
- https://www.mathworks.com/products/simscape.html
- https://www.micromo.com/technical-library/dc-motor-tutorials/motor-calculations
- https://en.wikipedia.org/wiki/Armature_Controlled_DC_Motor
- http://tutorial.math.lamar.edu/pdf/Laplace_Table.pdf
- https://en.wikipedia.org/wiki/State-space_representation
- http://ctms.engin.umich.edu/CTMS/index.php?aux=Home
- http://www.ecircuitcenter.com/Circuits/dc_motor_model/DCmotor_model.htm