SOCIETY OF ROBOTICS AND AUTOMATION

COPTER CONTROL

PROJECT REPORT

PROJECT CONTRIBUTORS

ALQAMA SHAIKH
ARYAN SHAH

ABOUT SRA VJTI

IDEATE
INNOVATE
INSPIRE.

A LITTLE BIT ABOUT OUR INSANELY DETERMINED STUDENT COMMUNITY

A big project involves overseeing many moving parts, often from different people. To have a successful rollout, project mentors rely on a well-crafted project plan to ensure objectives are met on time. SRA believes in the transfer of knowledge with consistency. Its community keeps on growing each year with more and more robotics contributing to its motive. This project was a part of its yearly event **Eklavya** which is a time-constrained event in which teams come together and work on a single project and learn about new domains.

ACKNOWLEDGEMENT

We are extremely grateful to our mentors Sagar Chotalia, Jash Shah, and Ayush Kaura for their support and guidance throughout the entire duration of the project. Their timely help, support, and mentoring have played a significant role in our Project reaching its completing phases.

We would also like to thank all the members of SRA VJTI for their timely support as well as for organizing Eklavya'22 and giving us a chance to work on this project.

ALQAMA SHAIKH

alqamascaptaina3@gmail.com

ARYAN SHAH

arsindia21@gmail.com

TABLE OF CONTENTS

PROJECT DESCRIPTION

Eklavya'22 Copter Control

Designing & Implementing an Optimal Control System of an Overactuated Hexa-copter with Co-axial Tilt-Rotors for Efficient Omnidirectional Flight in Simulation(**Gazebo**) using ROS & Python. Learning about the dynamics of a basic UAV and further implementing that knowledge in understanding the various dynamics of our system.

OUR GOALS AND OBJECTIVES

- The first step was to create an effective project plan.
- Start learning about Control systems and various control dynamics.
- Acquiring basic math skills required for the project.
- Learn about Git and Git-Hub and make use of the same throughout the project timeline.
- Importing required URDFs, SDFs, Launch, and world files.
- Installing and setting up ROS, gazebo, and simulating the model.
- Designing and Implementing a controller for a simplified hexacopter with no tilt units.
- Designing a controller for our use-case and enhancing our learnings.
- Finally, Implement the controller and doing the PID tuning

1. INTRODUCTION

Need for simulation

Testing experimental software on real-world hardware is risky as a variety of things can go wrong. It is very easy for the hardware to get damaged due to an errant programming bug; in extreme cases, people may also be at risk. In the domain of aerial vehicles, the risk of Control system failure during test flights can be very harmful. Also, hardware involves many unexpected variables which may increase the complexity of our project. So, testing models in simulation eliminates all of these concerns and gives us a stable risk-free environment whereby we can test software and iron out any bugs that may come up in it.

Introduction to Control systems

A control system is defined as a system of devices that manages, commands directs, or regulates the behaviour of other devices or systems to achieve the desired result. A control system achieves this through control loops, which are a process designed to maintain a process variable at the desired set point. A control system is a system to control other systems. In a quadcopter, we need to control the four primary motors of the plant. We vary their speeds to get desired results and control the altitude, roll, pitch, and yaw of the drone. This system is controlled via PIDs which translates to Proportional-Integral-Derivative control. PID will be explained in-depth further on.

Need for ROS/Gazebo

Robot Operating System (ROS) is an open-source robotics middleware suite. ROS was designed to be as distributive and modular as possible. ROS packages are fairly easy to make and distribute which makes them an ideal choice for testing robots. As a result, ROS fosters a vibrant community which makes it easy to find solutions for errors/bugs faced. The gazebo is a simulator that provides the facility to accurately and efficiently simulate robots in complex indoor/outdoor environments. It also provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.

Importance of Research Papers

Throughout this project, we have studied multiple Research papers. For every minute detail, we needed to go through the correct papers and understand how they affect our control algorithm. We didn't just review the Research Papers but we also took help from various other platforms like YouTube also. We learned about control systems, motor mixing algorithm, control allocation, tilt units, math, Copter, drone dynamics, and much more..
A research paper that we mostly considered was Voliro's hexacopter with tilt rotors and VoliroX hexacopter with co-axial rotors. These helped us a lot in shaping and bringing our project to life.

2. CONTROL ANALYSIS

Understanding the Control Systems

Everything around us working to make our lives easier is nothing but systems. How they interact with the environment and what should be their behaviour is decided by the control over the systems which we have. By combining real-life parameters, system variables, and control of the user we can make a control system.

Types of Systems

Non-Linear

These are real forms of systems that represent actual data of interaction with the surroundings. These are high fidelity, highly accurate form of systems and involves all non-linear factors, or one can say some real-life unpredictable factors in the system. For Example: Consider a drone, here if we take into account the bending stress, strain, strength of the materials, environment correction, etc. while designing the control system then this will be known as a non-linear system. Here the accuracy of the design is higher but the computational power required also increases drastically.

Linear

These are very ideal forms of systems that don't represent the real-world scenario. But these systems are often used alongside the non-linear system to make a program faster and don't involve extra computation which is usually required for the non-linear type of systems. For Example: Again consider a drone, but we are not taking into account all the factors that we took previously. So, now the variables are fewer so is the accuracy, but it can be improved by just verification by the non-linear model whenever required.

Open Loop Control

- The performance of the system isn't good enough to meet your requirements
- Commands not given according to the position of the robot
- Open-loop commanding is perfectly fine for systems that don't change much or where accuracy isn't as important
- No way of compensating for errors and making adjustments on its own

Closed Loop Control

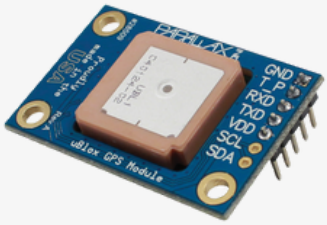
- Sensing the output of the process and feeding it back so the system can make adjustments accordingly
- Reference Value/Signal - which is the desired value or the ultimate goal
- We compare that to the measured value and what you're left with is the error or the Delta between where you are and where you want to be

WHAT IS A CONTROLLER?

A **Controller** is any device or algorithm that helps us gain control over a system that is into consideration. It is a mechanism that seeks to minimize the difference between the actual value of a system (i.e. the process variable) and the desired value of the system (i.e. the setpoint).

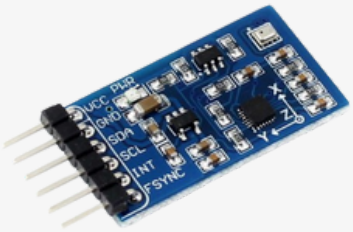
So, in our case, this controller is a python program which helps us achieve specific orientation and position with help of a special type of controller i.e. a PID controller.

Sensors Involved



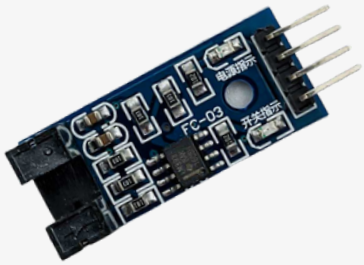
GPS sensor

- The GPS provides us with the altitude of the drone as well as the velocities in the x, y and z directions.



IMU sensor

- The IMU is used to measure the roll, pitch and yaw of the drone. It is an integral part of the control system.
- The IMU actually provides the data in terms of quaternion which has to then be converted to Euler angles using the **tf.transformation** module



Odometry sensor

- Odometry sensors provide the speed of the drone or hexacopter. In our case, it also provides orientation of the body.

Note: These sensors are here only for illustrative purposes. As we are implementing the model on simulation, these sensors are attached physically to the model and publish their data to their respective rostopic.

Access Sensor Readings

Sensor - ODOMETRY

rostopic echo /omav/odometry_sensor1/odometry

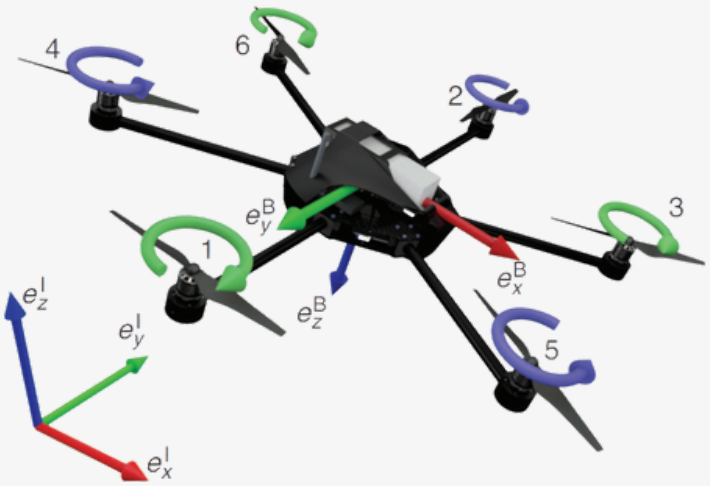
```
header:
  seq: 58922
  stamp:
    secs: 589
    nsecs: 620000000
  frame_id: "world"
child_frame_id: "omav/odometry_sensor1"
pose:
  pose:
    position:
      x: 1.316846911743286e-13
      y: 5.5470986310829374e-14
      z: 0.17499999998013013
    orientation:
      x: -1.5860646793656728e-13
      y: 3.762432126743893e-13
      z: -2.370962518150757e-19
      w: 1.0
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
twist:
  twist:
    linear:
      x: 1.3407892616043454e-11
      y: 2.4698354329028265e-11
      z: 1.9868390821482666e-11
    angular:
      x: -1.4113352006185848e-10
      y: 7.661654765069939e-11
      z: 2.433003997450725e-18
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
---
```

Accessing data in Python Script

```python
def get_Position(msg):
    global x,y,altitude

    # print("\n Odo frame:",msg.header.frame_id)
    x = round(msg.pose.pose.position.x,2)
    y = round(msg.pose.pose.position.y,2)
    altitude = round(msg.pose.pose.position.z,2)

    return(x, y, altitude)
```

Dynamics of a hexa-copter

It is nothing but the logic used to make the body go in the way we want it to. So, there are basically main 6 conditions that we need to satisfy.

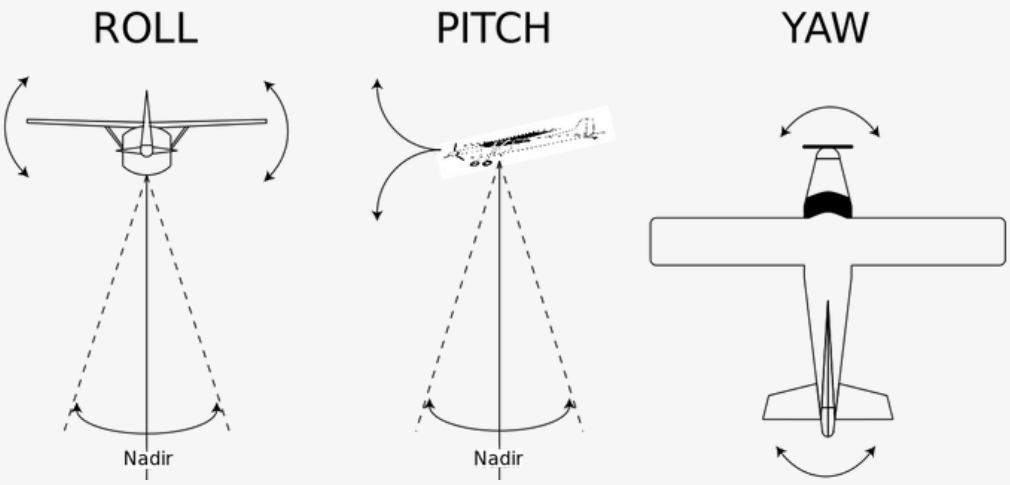


Before, jumping directly onto the dynamics, let's just start simply by defining the frame of reference. So, there are majorly 2 frames. One is in which we think in and understand dynamics i.e., inertial or the world frame. And the other frame is the body frame in which the body understands the given commands.

So, coming back to the 6 major conditions.

- Force desired - 3 conditions
- Moment desired - 3 conditions

All these 3 conditions are the required forces and moments in the 3 directions which are x, y and z.



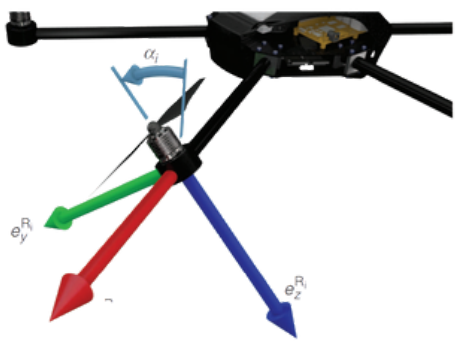
In a normal drone or a hexacopter (without tilt units). The major locomotion is performed using the simple inertial concept. So, in these types of systems, the RPY (Roll, Pitch, and Yaw) are used to navigate through the given coordinates. And to keep the Torque and forces in equilibrium motor mixing is often used. Motor mixing is nothing but a simplified approach to avoid matrix calculations involved in satisfying the 6 main conditions of **rigid body dynamics.**

Our basic conditions

Rigid Body Model

For the modeling of the body dynamics, the Newton–Euler formalism was used. Its general form is the following:

$$\begin{bmatrix} m\mathcal{I}_3 & \mathbf{0} \\ \mathbf{0} & \boldsymbol{J} \end{bmatrix} \begin{bmatrix} {}_B\dot{\boldsymbol{v}} \\ {}_B\dot{\boldsymbol{\omega}} \end{bmatrix} + \begin{bmatrix} {}_B\boldsymbol{\omega} \times (m \cdot {}_B\boldsymbol{v}) \\ {}_B\boldsymbol{\omega} \times (I \cdot {}_B\boldsymbol{\omega}) \end{bmatrix} = \begin{bmatrix} {}_B\boldsymbol{F} \\ {}_B\boldsymbol{M} \end{bmatrix}, \quad (1)$$



Our use case involves the tilt rotors which help us to gain more control over the overall dynamics of the copter and also make the system more complex.

So, to gain the thrust force the force from the individual rotor must be resolved into its components according to the tilt angles.

What we have V/S What we need

- We have all the forces and moments in the inertial frame.
- We also have the current and desired orientations and positions.
- We want the angles for the tilt rotors and angular velocities of the rotors

Force Desired

The position control is based on a PID controller with a feed-forward term that generates a desired force vector in the vehicle body frame $\mathcal{F}_B$. The position error is defined as the difference between the desired and estimated position ${}_I\boldsymbol{p}_{\text{err}} = {}_I\boldsymbol{p}_{\text{des}} - {}_I\hat{\boldsymbol{p}}$. The position control law is given by

$${}_B\boldsymbol{F}_{\text{des}} = \boldsymbol{R}_{IB}^T \left(k_{p,pI}\boldsymbol{p}_{\text{err}} + k_{p,pI}\dot{\boldsymbol{p}}_{\text{err}} + k_{i,p}\int \boldsymbol{p}_{\text{err}}\,dt + m\boldsymbol{g} + {}_I\ddot{\boldsymbol{p}}_{\text{des}} \right),$$

Note:

- **A rotation matrix is needed to transform the required force into the body frame.**
- **All calculations are performed w.r.t the body frame.**

Moment Desired

$$\boldsymbol{q}_{\text{err}} = \boldsymbol{q}_{\text{des},IB} \otimes \hat{\boldsymbol{q}}_{IB}^{*} = \begin{pmatrix} q_{w,\text{err}} \\ \boldsymbol{q}_{v,\text{err}} \end{pmatrix}. \tag{13}$$

The desired body rate $\boldsymbol{\omega}_{\text{des}}$ is generated from the vector part of the quaternion error $\boldsymbol{q}_{v,\text{err}}$ as follows:

$$\boldsymbol{\omega}_{\text{des}} = k_q \,\text{sign}\,(q_{w,\text{err}})\,\boldsymbol{q}_{v,\text{err}}, \tag{14}$$

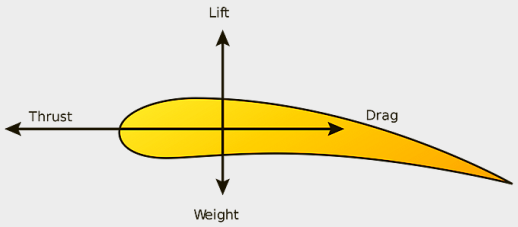
where k_q is a tuning parameter. Note that the sign of the real part of the quaternion error is used to avoid the unwinding phenomenon.

The desired moments $\boldsymbol{M}_{\text{des}}$ are computed as

$$\boldsymbol{M}_{\text{des}} = k_r\,(\boldsymbol{\omega}_{\text{des}} - \hat{\boldsymbol{\omega}}) - \boldsymbol{r}_{\text{off}} \times {}_B\boldsymbol{F}_{\text{des}} + \hat{\boldsymbol{\omega}} \times \boldsymbol{J}\hat{\boldsymbol{\omega}}, \tag{15}$$

where k_r is the rate controller gain, $\hat{\boldsymbol{\omega}}$ is the estimated angular velocity, $\boldsymbol{r}_{\text{off}}$ is the center of mass offset, and $\boldsymbol{J}$ is the vehicle's inertia matrix.

Now, we have our requirements ready, so we just need to solve the equations by applying a simple concept of allocation and that would be it!



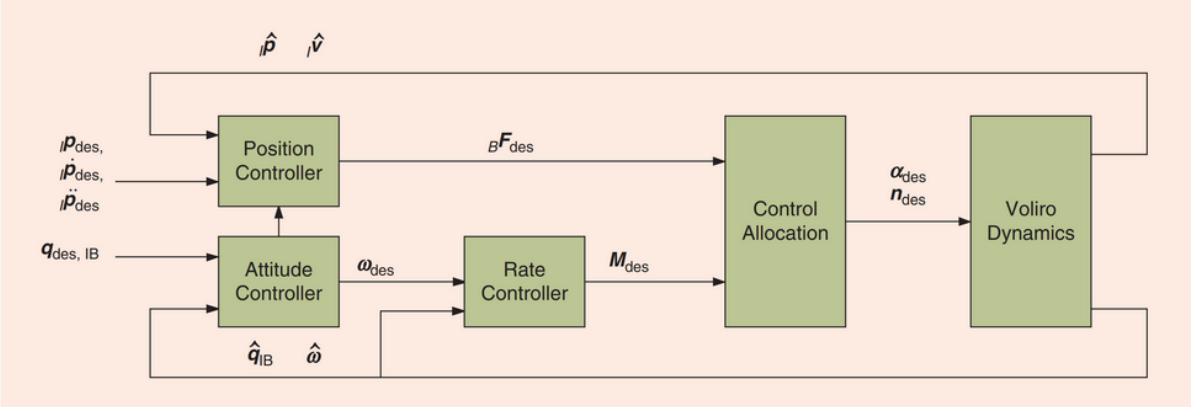
$$F = \mu n^2,$$
$$\tau = \kappa n^2,$$

The four main force forces acting on an aerofoil are weight, lift, drag, and thrust. Aerofoil is the structure or the shape of the cross-section of the rotor blades used in the hexacopter, which provides the necessary torque and forces to manoeuvre the body.

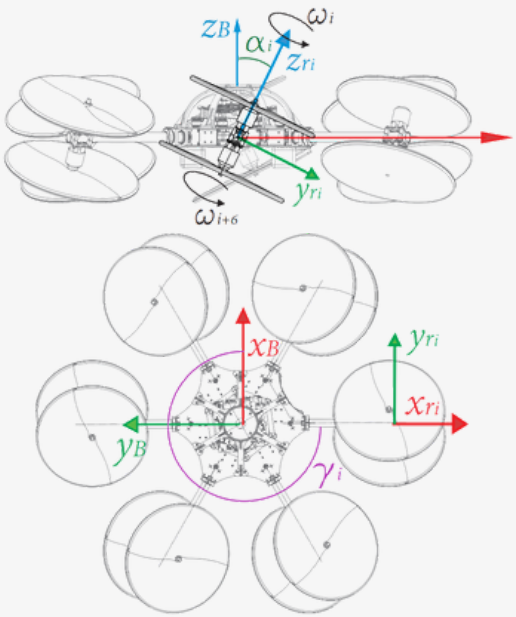
How we used these forces?

There is a relation between thrust and the speed of rotors. This relation is not linear instead we are assuming it to be a quadratic relation i.e. for x unit rise in speed there is kx^2 rise in the thrust force induced. This in turn gives us lift force kx^2. And due to the fact that the rotor has some drag, a drag force is also induced.

There are two controllers here, Force desired and Moment desired. Position controller gives us the force desired and the attitude controller will give us the moment desired.



Complexity of the system



Here, the frames of the body are defined.

The next part is to think of a way so we can reach a setpoint by minimizing the error. This error in orientation and position will help us find the moment and forces desired as described earlier.

So, now we need to introduce logic in order to give the tilt angles and rotation speed to the individual rotors.

.

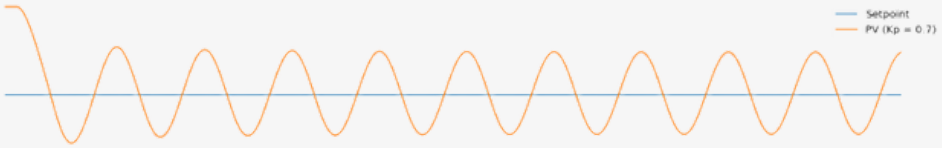
So, the complexity in our system arises due to the fact that there are 3 unknowns for each arm unit, in total 3x6 = 18 unknowns, but we only have 12 equations. So, now we need to couple the tilt angles and angular velocities such that the variables we need to find our 12 and thus we can solve using simple matrix math.

$$\boldsymbol{A} = c_f \begin{bmatrix} \sin(\gamma_i) & 0 & \dots \\ -\cos(\gamma_i) & 0 & \dots \\ 0 & 1 & \dots \\ -s_j c_d \sin(\gamma_i) & l_x \sin(\gamma_i) & \dots \\ s_j c_d \cos(\gamma_i) & -l_x \cos(\gamma_i) & \dots \\ -l_x & -s_j c_d & \dots \end{bmatrix} \quad \boldsymbol{u} = \begin{bmatrix} \sin(\alpha_i)\Omega_{i,} \\ \cos(\alpha_i)\Omega_i \\ \vdots \\ \sin(\alpha_i)\Omega_{i+6} \\ \cos(\alpha_i)\Omega_{i+6} \\ \vdots \end{bmatrix} \begin{matrix} \forall i \in \{1...6\} \\ \forall j \in \{1...12\} \end{matrix}$$

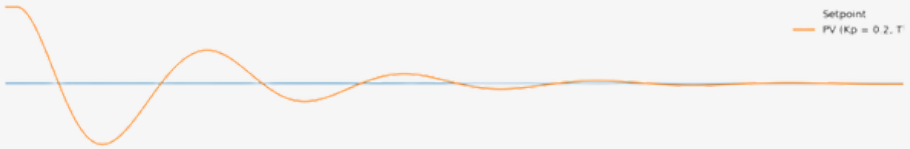
PID Tuning

PID is basically a type of controller which takes into account the errors between the current position and setpoint position and then provides us the control of the system using the Proportional, derivative, and Integral terms

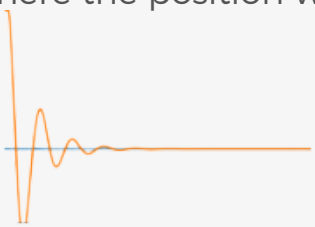
- **Proportional term:** which multiplies Kp directly with the error value.



- **Integral term:** which multiplies Ki with the integral of the error with respect to time. The integrator state "remembers" all that has gone on before, which is what allows the controller to cancel out any long-term errors in the output.



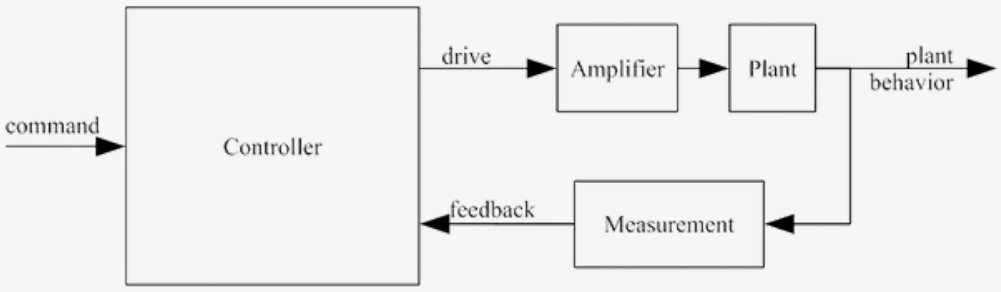
- **Derivative term:** which multiplies Kd with the derivative of the error. This gives you a rough estimate of the velocity (delta position/sample time), which predicts where the position will be in a while

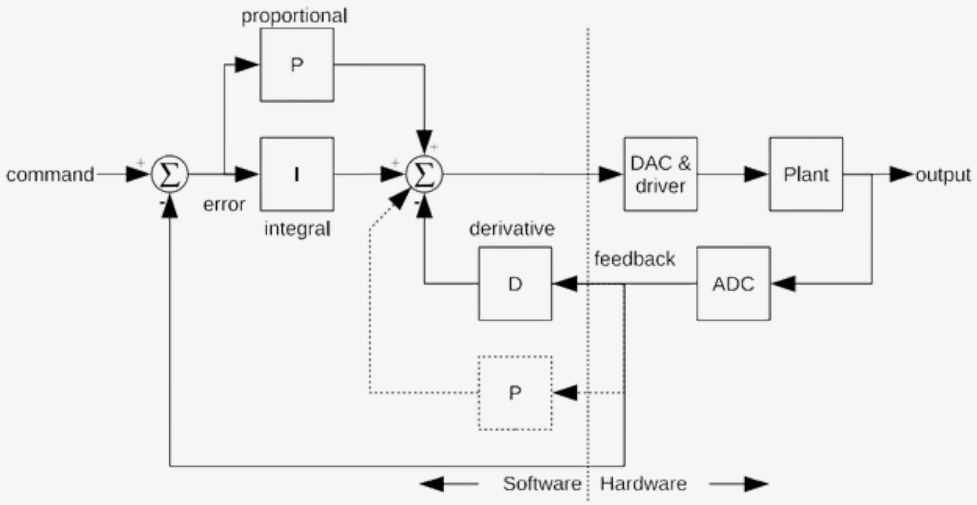


$$u(t) = K_p e(t) + K_i \int e(t)dt + K_p \frac{de}{dt}$$

$u(t)$ = PID control variable
K_p = proportional gain
$e(t)$ = error value
K_i = integral gain
de = change in error value
dt = change in time

- By tuning the gains appropriately we can reduce the time taken for the plant to become stable thereby minimizing the amount of time spent outside the set target. An ideal PID gives the below curve of error vs time although achieving ideal gain values is easier said than done.





IMPLEMENTATION

Cloning the Repository

assets	Added all image files to assets folder
rotors_comm	Deleted .rst files
rotors_description	corrected urdf
rotors_gazebo	Changes omav.xacro
rotors_gazebo_plugins	Deleted .rst files
scripts	Added Buildable src files (rotors_simulator), scripts(which contain c…
Installations.md	Updated Installation.md
README.md	Added all image files to assets folder
dependencies.rosinstall	Added Buildable src files (rotors_simulator), scripts(which contain c…
rotors_demos.rosinstall	Added Buildable src files (rotors_simulator), scripts(which contain c…
rotors_hil.rosinstall	Added Buildable src files (rotors_simulator), scripts(which contain c…
rotors_minimal.rosinstall	Added Buildable src files (rotors_simulator), scripts(which contain c…

Position Controller

$_I\boldsymbol{p}_{\text{err}} = {}_I\boldsymbol{p}_{\text{des}} - {}_I\hat{\boldsymbol{p}}$. The position control law is given by

$${}_B\boldsymbol{F}_{\text{des}} = \boldsymbol{R}_{IB}^{T}\left(k_{p,pI}\boldsymbol{p}_{\text{err}} + k_{p,pI}\dot{\boldsymbol{p}}_{\text{err}} + k_{i,p}\int \boldsymbol{p}_{\text{err}}\,dt + m\boldsymbol{g} + {}_I\ddot{\boldsymbol{p}}_{\text{des}}\right),$$

Body to Inertial Frame Rotation Matrix

```
# Body to Inertial Frame Rotation Matrix
rotation_matrix = np.array([[math.cos(theta)*math.cos(gamma),math.sin(gamma)*math.cos(theta),-math.sin(theta)],
                            [math.sin(phi)*math.sin(theta)*math.cos(gamma)-math.cos(phi)*math.sin(gamma),
                            math.sin(phi)*math.sin(theta)*math.sin(gamma)+math.cos(phi)*math.cos(gamma),
                            math.sin(phi)*math.cos(theta)],
                            [math.cos(phi)*math.sin(theta)*math.cos(gamma)+math.sin(phi)*math.sin(gamma),
                            math.cos(phi)*math.sin(theta)*math.sin(gamma)-math.sin(phi)*math.cos(gamma),
                            math.cos(phi)*math.cos(theta)]])
```

Tuning Parameters

```
# Tuning Parameters Subscriber Nodes
rospy.Subscriber("pid_x", Float64MultiArray, setPID_x)
rospy.Subscriber("pid_y", Float64MultiArray, setPID_y)
rospy.Subscriber("pid_z", Float64MultiArray, setPID_z)
rospy.Subscriber("Tuning_Parameter", Float64, set_tuning_parameter)
rospy.Subscriber("Rate_Controller_Gain", Float64, set_rate_controller_gain)
```

- PID x : Proportional, Integral & Derivative Terms in X-Axis - **3*1 MATRIX**
- PID y : Proportional, Integral & Derivative Terms in Y-Axis - **3*1 MATRIX**
- PID z : Proportional, Integral & Derivative Terms in Z-Axis - **3*1 MATRIX**
- Tuning Parameter : **kq** : Parameter which tunes Quaternion Orientation Error - **SCALAR**
- Rate Controller Gain : **kr** : Parameter which tunes Angular Velocity Error - **SCALAR**

```
# Force Desired Calculation
F_desired = mass_total*( grav_matrix +  p_position_err + d_position_err + i_position_err )

F_desired = np.round_((np.matmul(rotation_matrix,F_desired)).real,decimals=2)
```

Angular Velocity Desired Calculation

$$\boldsymbol{q}_{\text{err}} = \boldsymbol{q}_{\text{des},IB} \otimes \hat{\boldsymbol{q}}^{*}_{IB} = \begin{pmatrix} q_{w,\text{err}} \\ \boldsymbol{q}_{v,\text{err}} \end{pmatrix}. \tag{13}$$

The desired body rate $\boldsymbol{\omega}_{\text{des}}$ is generated from the vector part of the quaternion error $\boldsymbol{q}_{v,\text{err}}$ as follows:

$$\boldsymbol{\omega}_{\text{des}} = k_q \,\text{sign}\,(q_{w,\text{err}})\,\boldsymbol{q}_{v,\text{err}}, \tag{14}$$

where k_q is a tuning parameter. Note that the sign of the real part of the quaternion error is used to avoid the unwinding phenomenon.

```
# Quaternion Desired and Quaternion Current
p0 = q_w_desired
p1 = q_x_desired
p2 = q_y_desired
p3 = q_z_desired

q0 = q_w_current
q1 = -q_x_current
q2 = -q_y_current
q3 = -q_z_current

# Quaternion Error Equations (Knocker Product):
q_w_error = (p0*q0 - p1*q1 - p2*q2 - p3*q3)
q_x_error = (p0*q1 + p1*q0 + p2*q3 - p3*q2)
q_y_error = (p0*q2 - p1*q3 + p2*q0 + p3*q1)
q_z_error = (p0*q3 + p1*q2 - p2*q1 + p3*q0)


# We Require only the sign of q_w_error for further Calculation
if(q_w_error < 0):
    sign_q_w_error = -1
elif(q_w_error > 0):
    sign_q_w_error = 1
# Initializing the 3*1 Matrix of the vector part of Quaternion Error for Further Calculations
q_v_error = np.round_(np.array([[q_x_error],
                                [q_y_error],
                                [q_z_error]]),decimals=2)

# Desired Angular Velocity :
w_desired = (kq* sign_q_w_error * q_v_error)
```

Orientation Control

The desired moments $\boldsymbol{M}_{\text{des}}$ are computed as

$$\boldsymbol{M}_{\text{des}} = k_r\,(\boldsymbol{\omega}_{\text{des}} - \hat{\boldsymbol{\omega}}) - \boldsymbol{r}_{\text{off}} \times {}_B\boldsymbol{F}_{\text{des}} + \hat{\boldsymbol{\omega}} \times \boldsymbol{J}\hat{\boldsymbol{\omega}}, \qquad (15)$$

where k_r is the rate controller gain, $\hat{\boldsymbol{\omega}}$ is the estimated angular velocity, $\boldsymbol{r}_{\text{off}}$ is the center of mass offset, and $\boldsymbol{J}$ is the vehicle's inertia matrix.

```
# To Find Error in Angular Velocity which is a 3*1 Matrix
# To Calculate : werror = (wdes - ω^)
#                werror = (w_current - w_current)
curr_w_error = w_desired - w_current

# To Calculate 1st Term in Moment_Desired Equation : q_intermediate_1 = kr.(wdes - ω^)
#                = kr.werror
q_intermediate_1 = (kr * curr_w_error)

# To Calculate intermediate for 3rd term in Moment_Desired Equation : q_intermediate_3_1 = J.ω^
# Dot Product
q_intermediate_3_1 = np.round_(np.matmul(I, w_current), decimals=2)

# To Calculate 3rd Term in Moment_Desired Equation : q_intermediate_3_2 = ω^ x (J.ω^)
#                                                  = ω^ x q_intermediate_3_1
# Cross Product
q_intermediate_3_2 = np.round_(np.cross(w_current, q_intermediate_3_1,axis=0), decimals=2)

M_desired = q_intermediate_1 + q_intermediate_3_2
```

Control Allocation

Components of Decomposed Force

$$F_{v,i} = \mu n_i^2 \cos \alpha_i$$
$$F_{l,i} = \mu n_i^2 \sin \alpha_i .$$

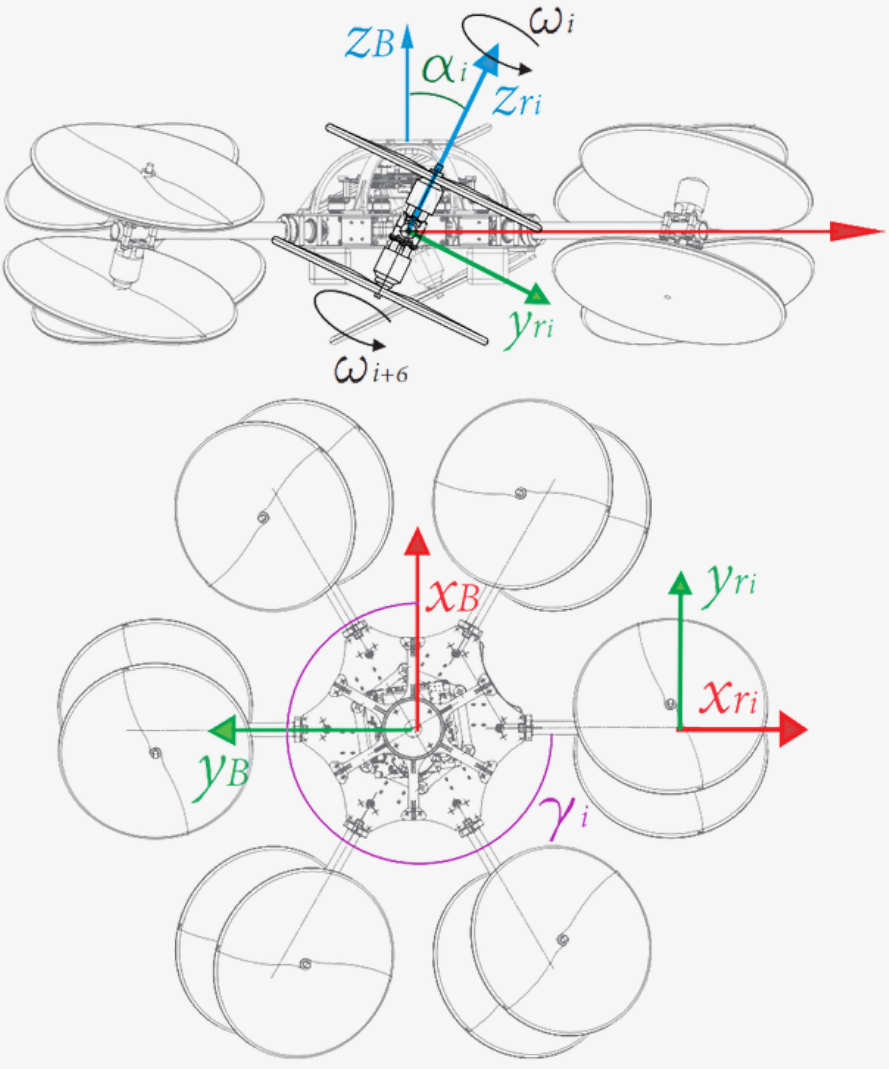
Decomposed Force Calculations

$$\boldsymbol{F}_{dec} = \begin{bmatrix} \sin(\alpha_i)\Omega_{i,} \\ \cos(\alpha_i)\Omega_i \\ \vdots \\ \sin(\alpha_i)\Omega_{i+6} \\ \cos(\alpha_i)\Omega_{i+6} \\ \vdots \end{bmatrix} \begin{matrix} \forall i \in \{1...6\} \\ \forall j \in \{1...12\} \end{matrix}$$

```
# Combined Desired Matrix which is a 6*1 Matrix
# It contains Force Desired and Moment Desired Terms
desired = np.array([[F_des[0][0]],
                    [F_des[1][0]],
                    [F_des[2][0]],
                    [M_des[0][0]],
                    [M_des[1][0]],
                    [M_des[2][0]]])

# Decomposed Force Matrix which is multiplied with
# Pseudo Inverse of the Static Allocation Matrix
F_dec =  np.matmul( A_pseudo_inv, desired)
```

Static Allocation Matrix

$$\boldsymbol{A} = c_f \begin{bmatrix} \sin(\gamma_i) & 0 & \dots \\ -\cos(\gamma_i) & 0 & \dots \\ 0 & 1 & \dots \\ -s_j c_d \sin(\gamma_i) & l_x \sin(\gamma_i) & \dots \\ s_j c_d \cos(\gamma_i) & -l_x \cos(\gamma_i) & \dots \\ -l_x & -s_j c_d & \dots \end{bmatrix} \quad \begin{matrix} \forall i \in \{1 \dots 6\} \\ \forall j \in \{1 \dots 12\} \end{matrix}$$



Static Allocation Matrix

```
# Static Allocation Matrix
A = np.array([  [   0   ,   -1   ,   0   ,   1   ,   0   ,  0.5  ,   0   , -0.5  ,   0   , -0.5  ,
                [   0   ,   0    ,   0   ,   0   ,   0   ,   -t1  ,   0   ,  t1   ,   0   ,  -t1   ,
                [   1   ,   0    ,   1   ,   0   ,   1   ,   0   ,   1   ,   0   ,   1   ,   0   ,
                [(len*(-1)), ((-1)*s_cw*kap*(-1)), (len*(1)), ((-1)*s_acw*kap*(1)), (len*(0.5)), (
                [((-1)*len*(0)), (s_cw*kap*(0)), ((-1)*len*(0)), (s_acw*kap*(0)), ((-1)*len*(t1))
                [((-1)*s_cw*kap), -len, ((-1)*s_acw*kap), -len, ((-1)*s_cw*kap), -len, ((-1)*s_acw

#Transpose of Static Allocation Matrix
A_trans = np.transpose(A)

# <--------------------------------Moore-Penrose Pseudo Inverse------------------------------>
X = np.array(np.matmul(A,A_trans))
# Since, m <= n for Static Allocation Matrix therefore, A(pseudo_inverse) =  A^T * ( A * A^T )^-1
# RIGHT INVERSE

# Now, ( A * A^T )^-1
X_inv = np.linalg.inv(X)

# Now, A(pseudo_inverse) = A^T * ( A * A^T )^-1
A_pseudo_inv = np.matmul(A_trans,X_inv)
```

Moore-Penrose Pseudo Inverse

Moore-Penrose Pseudo Inverse is used when a Matrix to be inverted is not a Square Matrix

LEFT INVERSE

- If m>n

$$A_{\text{left}}^{-1} = (A^T A)^{-1} A^T$$

RIGHT INVERSE

- If n>m

$$A_{\text{right}}^{-1} = A^T (A A^T)^{-1}$$

where, m = no. of rows
n = no. of columns

Calculating Rotor RPM

Rotor RPM from Decomposed Force Matrix

$$\boldsymbol{u} = \begin{bmatrix} \sin(\alpha_i)\Omega_{i,} \\ \cos(\alpha_i)\Omega_i \\ \vdots \\ \sin(\alpha_i)\Omega_{i+6} \\ \cos(\alpha_i)\Omega_{i+6} \\ \vdots \end{bmatrix} \begin{matrix} \\ \\ \\ \forall i \in \{1...6\} \\ \forall j \in \{1...12\} \\ \\ \end{matrix}$$

$$\Omega_i = \Omega_i \sin^2(\alpha_i) + \Omega_i \cos^2(\alpha_i) = \sin(\alpha_i)u_{2i} + \cos(\alpha_i)u_{2i+1}$$
$$\Omega_{i+6} = \Omega_{i+6} \sin^2(\alpha_i) + \Omega_i \cos^2(\alpha_i) = \sin(\alpha_i)u_{2(i+6)} + \cos(\alpha_i)u_{2(i+6)+1}$$

```
# Calculating Rotor RPM from Decomposed Force Matrix
for i in range(0, 12):
    ang_vel[i]= round(abs((1/sqrt(Mu))*(sqrt(sqrt(pow(relation_matrix[2*i],2) + pow(relation_matrix[2*i+1],2))).real)), 2)
```

Calculating Tilt Rotor Angles

Tilt Rotor Angles from Decomposed Force Matrix

$$\alpha_i = \text{atan2}\left(F_{l,i}, F_{v,i}\right)$$

```
# Calculating Tilt-Rotor Angles from Decomposed Force Matrix
for i in range(0, 6):
    x1 = relation_matrix[2*i+1]
    x2 = relation_matrix[2*i]
    tilt_ang[i] = round(atan2(x1,x2),2) # atan2(sin/cos)
```

Speed Publishing

We need to give Rotor RPM & Tilt Rotor angles using
/omav/command/motor_speed
in the form of an **18*1 Matrix**

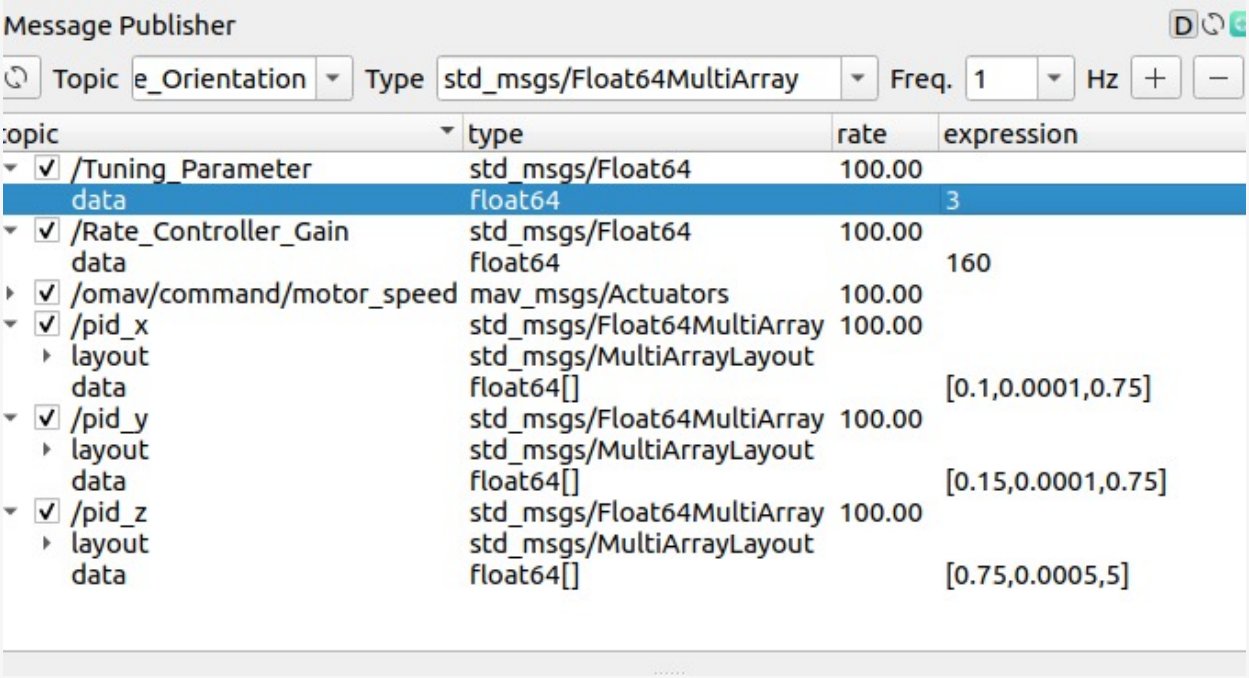
```
# Creating Publisher to give Rotor RPM & Tilt-Rotor Angles
speed_pub = rospy.Publisher("/omav/command/motor_speed", Actuators ,queue_size=100)
```

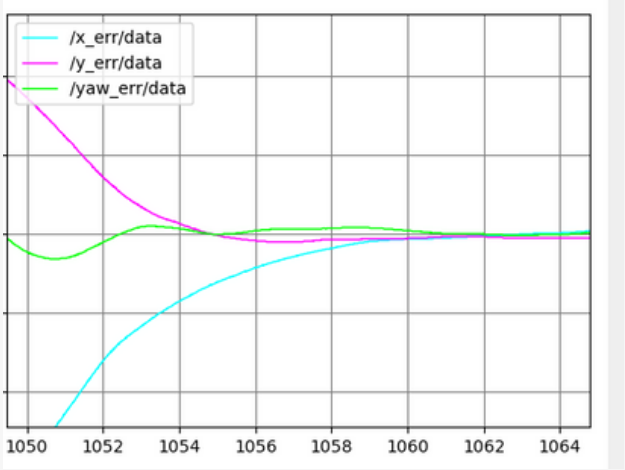
```
def speed_assign(tilt_ang, ang_vel_rot, speed):
    for t in range(0, 12):
        # 12 - Rotor RPM in order
        speed.angular_velocities[t] = ang_vel_rot[t]
    for t in range(0, 6):
        # 6 - Tilt Rotor Angles in order
        speed.angular_velocities[12+t] = tilt_ang[t]

    return(speed)
```

Tuning

- Tuning is one of the main parts of a Control System Project
- Tuning is done for efficient, accurate and fast flight

Tuning is done using Message Publisher feature in ROS





WE SHALL CONDUCT EVALUATION ACTIVITIES AND USE SUCCESS OR PERFORMANCE INDICATORS TO MEASURE THE PROGRESS OF THE PROJECT PLAN.

FUTURE WORK

- **Designing a more-stable orientation or attitude controller**: Control System for stable flight with desired orientation in X, Y and Z Axis simultaneously.
- **Stability in presence of the wind**
- **Different Control Algorithms:** PID is not an ideal controller for aerial vehicles since they require very fine, accurate tuning, which consumes a lot of time. So, we need a controller which reduces the number of variables and that can be tuned easily. Also, finding a more-efficient Control Algorithm for our system, like LQR Controller.
- **Implementing obstacle avoidance:** Currently, we are controlling the drone to go to a specific Coordinate with a specific Orientation in free space. We would like to elaborate on the functioning of our drone and have more realistic functionality.
- **Making a high-speed control for position and orientation for industrial applications**: One of the main limitations of our implementation is that it works for very low speeds.
- **Implementing the same model on C++:** As C++ is a very fast language than python so our calculations would be even more fast and accurate using C++

PROJECT TIMELINE

THIS SECTION INCLUDES THE PROJECT PROCESSES, IMPLEMENTATION, AND EXECUTION.

TASK	START DATE	END DATE
PHASE 01 LEARINING ABOUT DYNAMICS OF DRONES AND CONTROL SYSTEMS	15-08-2022	18-08-2022
PHASE 02 IN-DEPTH ANALYSIS OF RESEARCH PAPER AND IMPLEMENTATION OF CONTROLLER IN DUMMY MODEL (FIREFLY)	18-08-2022	25-08-2022
PHASE 03 ACTUAL IMPLEMENTATION ON OMAV	27-08-2022	TILL END

CONCLUSION

COLLABORATION ALLOWS US TO KNOW MORE THAN WE ARE CAPABLE OF KNOWING BY OURSELVES

Concluding, controlling a drone is a task of fine-tuning parameters. However, once this tuning act is completed the drone will fly in a stable manner holding position & orientation. Challenges came and went along the way, but it was a great learning experience to dive into the Dynamics of a UAV and a complex over-actuated system. Learning Control System, ROS, Gazebo, Python & Modern Robotics had their own challenges, but we enjoyed the entire journey with an awesome amount of learning and hands-on experience in testing a complex drone in Simulation. The knowledge as well as the first-hand experience we gained during the entire duration of the project will prove invaluable in the future. We also explored a variety of domains, and this technical exposure was crucial and helped us climb the ladder from Fresh clueless, aimless FYs to a bit knowledgeable, aimed SYs. The entire experience will also make it easier to pick a domain in the future. The peers and community formed will help in development at a personal and community level in a much faster and more detailed manner.

EVERYTHING SEEMS HARD BEFORE YOU STEP IN TO DO IT

REFERENCES

- Kamel, Mina, Sebastian Verling, Omar Elkhatib, Christian Sprecher, Paula Wulkop, Zachary Taylor, Roland Siegwart, and Igor Gilitschenski. "The Voliro Omniorientational Hexacopter : An Agile and Maneuverable Tiltable-Rotor Aerial Vehicle" IEEE Robotics & Automation Magazine 25, no. 4 (2018): 34-44.
- Karen Bodie, Zachary Taylor, Mina Kamel, and Roland Siegwart. "Towards Efficient Full Pose Omnidirectionality with Overactuated MAVs" In International Symposium on Experimental Robotics, pp. 85-95. Springer, Cham, 2018.
- Allenspach, Mike, Karen Bodie, Maximilian Brunner, Luca Rinsoz, Zachary Taylor, Mina Kamel, Roland Siegwart, and Juan Nieto. "Design and optimal control of a tiltrotor micro aerial vehicle for efficient omnidirectional flight" The International Journal of Robotics Research 39, no. 10-11 (2020): 1305-1325.