Drone (Quadcopter) Design, Simulation and Control System in ROS-Gazebo

Project Report

Eklavya Mentorship Programme

At

SOCIETY OF ROBOTICS AND AUTOMATION, VEERMATA JIJABAI TECHNOLOGICAL INSTITUTE, MUMBAI

OCTOBER 2021

ACKNOWLEDGEMENT

We are extremely grateful to our mentors Saad Hashmi, Karthik Swaminathan and Dhruvi Doshi for their support and guidance throughout the duration of the project.
We would also like to thank all the members of SRA VJTI for their timely support as well as for organising Eklavya and giving us a chance to work on this project.

Jash Shah
jash28582@gmail.com

Toshan Luktuke
toshan1603@gmail.com

Table of Contents

Project Overview

The drone industry is one of the hottest emerging industries of the new decade. Drone transportation provides an automated approach to delivering and transporting goods which is far superior to the existing systems in terms of its speed and scalability. Its key features are that drones can be run with software without any intervention of humans and that drones fly above traffic, streets and conventional obstacles without any difficulty. Additionally, drones operate on electricity and are much more efficient than normal vehicles. To explore this potential for application, we have designed and simulated a drone in Gazebo as well as built its control system using ROS and Python.

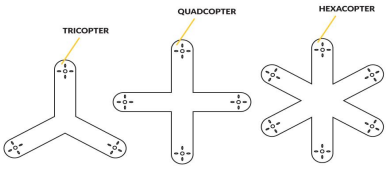
1. Introduction

1.1. What goes into designing a drone?

The design of a drone is very much influenced by the physical constraints of its environment. Depending upon the number of propellers drones can be classified as tricopters, quadcopters, hexacopters, etc. For our project we've chosen to work with the quadcopter(QC). The frame of a QC can be in the X(cross) or +(pull).

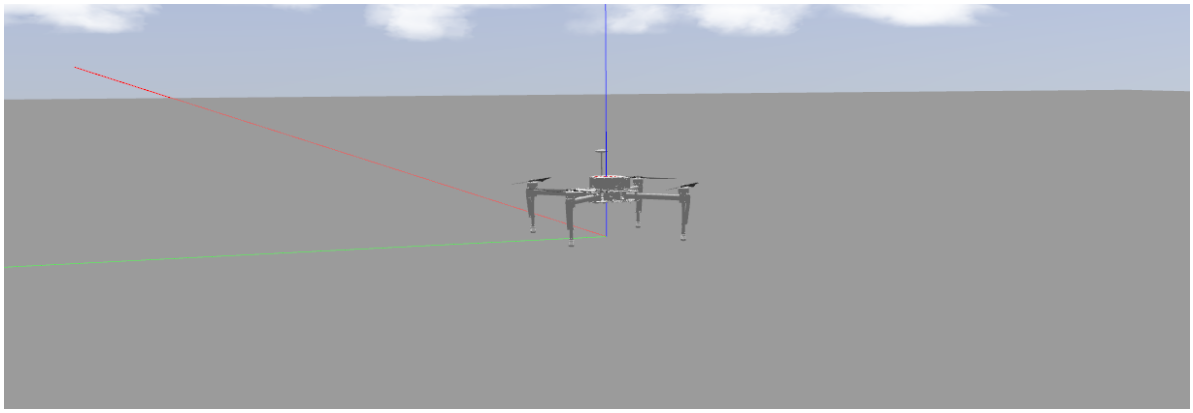
+ shaped quadcopter

X shaped quadcopter



1.2. Why the need for Simulation?

Testing experimental software on real world hardware is risky and dangerous as there are a variety of things that can go wrong. It is very easy for the hardware to get damaged due to an errant programming bug and in extreme cases, people may also be at risk. 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.



1.3. What are Control Systems?

A control system is defined as a system of devices that manages, commands, directs, or regulates the behavior of other devices or systems to achieve a desired result. A control system achieves this through control loops, which are a process designed to maintain a process variable at a desired set point.
Basically a control system is a system to control other systems.
In a quadcopter what we need to control are the four primary motors of the plant. We vary their speeds to get 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.

1.4. Use of ROS/Gazebo in simulating and testing robots.

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 it 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. Gazebo is a simulator which 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.

2. In-Depth Analysis

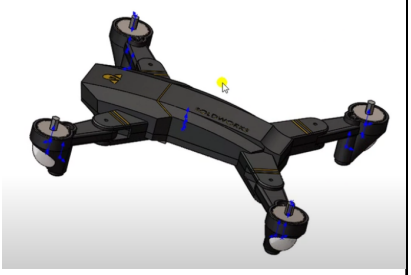
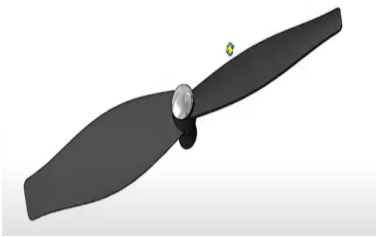
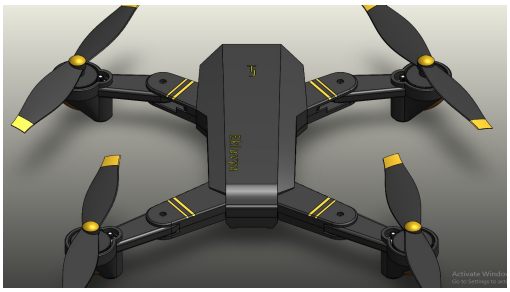
2.1. General

Working with ROS/Gazebo, the complete design, world set-up process and control of any robot involves:

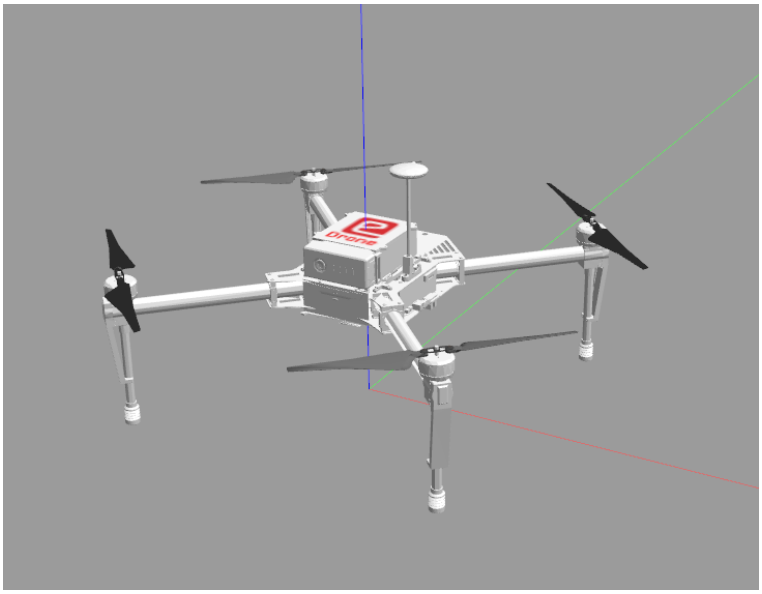
- Using a 3D design software to design the bot.
- Exporting the 3D model into a URDF (Unified Robotic Description Format)which allows it to be opened in a simulated world.
- Setting up the necessary physical constraints on the model using plugins.
- Designing a control system by using feedback loops with sensor data.
- Tuning the controllers for minimizing the errors.

2.2. Designing in Solidworks

SolidWorks(SW) is a model designing software convenient for robot designing.
A model in SW can involve a part or an assembly.An assembly being a multipart model made by combining different parts through joints. While there are many models already available for a Quadcopter assembly in SW, we initially decided to go with this one[1].
Our model involves two simple parts:

Base Link	Wing Link	Joints	Final Assembly
		Revolute Joints	

However, since there were some physics related problems we faced while simulating this model in gazebo, we decided to go with a pre-tested model taken from the E-Yantra 2020 repo[3].



2.3. Importing a URDF :

SW has tools available to export an assembly to a URDF model.The URDF (Universal Robot Description Format) model is a collection of files that describe a robot's physical description to ROS. These files are used by a program called ROS to tell the computer what the robot actually looks like in real life. The URDF file is a .xml file which describes the parameters of the bot to ROS. Gazebo can then be used to open this URDF file into a simulated world using a launch file.

2.3.1. ROS Terminologies:

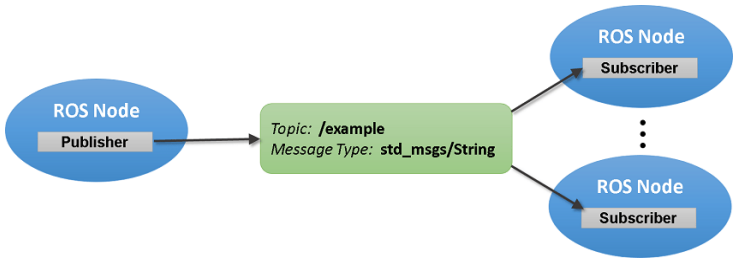
The best resource for learning ROS is through the docs [2]. Let's look at some basic concepts and how they've been implemented in our project.

1. Catkin Workspace:
   A catkin workspace is a folder where you modify, build, and install catkin packages.

2. ROS Package:
   Software in ROS is organized in packages. A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module.

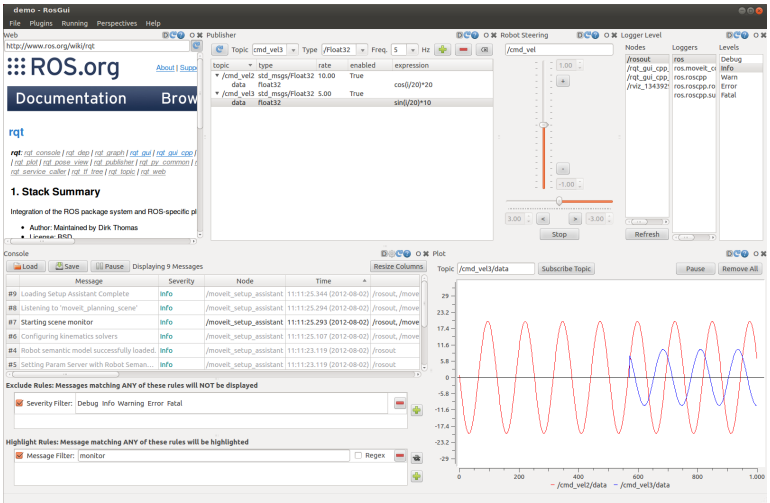
3. ROS Node, Topics and Messages:

Nodes, topics and messages are the basic components of the communication system in ROS. Nodes are points which can either publish data or subscribe to a topic to read its data. Topics are the pathways through which the data is transferred. A single topic may subscribe or publish to multiple nodes. Messages are the actual data which is sent. There are many in-built message types(like string,int,float,etc.) but custom types(like prop_spped used in our pkg)can also be used.



4. RQT:

RQT is a software framework of ROS that implements the various GUI tools in the form of plugins. One can run all the existing GUI tools as dockable windows within rqt! The tools can still run in a traditional standalone method, but rqt makes it easier to manage all the various windows on the screen at one moment



2.3.2. Converting to SDF:

Simulation Description Format (SDF) was created for use in Gazebo to solve the shortcomings of URDF. A URDF file can be easily converted to SDF using the following ros cmd:

```
:~/catkin_ws$ gz sdf -p /drone_urdf.urdf > /drone_sdf.sdf
```

2.4. Designing a World

The term **world** in gazebo is used to describe a collection of robots, objects and global parameters including the sky, ambient light, and physics properties.

A typical world will have all the necessary global properties of the simulation environment including the lighting, physics, wind, gravity and the robots.
For our purposes we are using a custom world available with Gazebo called empty_sky.world.
This world along-with the robot is launched using a .launch file

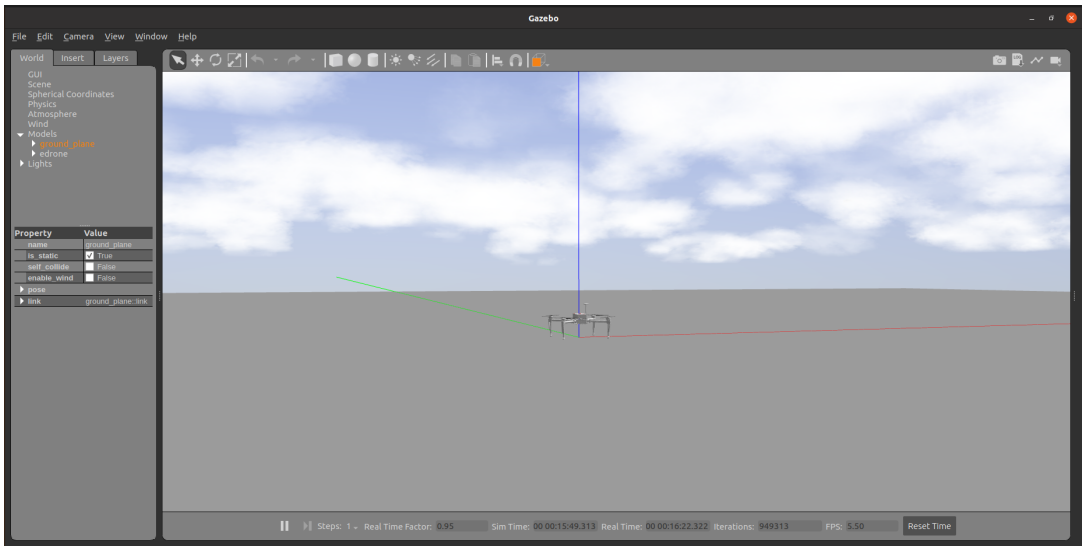
```
<include
    file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name"  value="$(find vitarana_drone)/worlds/drone$(arg static).world"/>
</include>
```

This is the launch file we used, it includes an <include> tag with the basic parameters being to find the empty_world file and use it with the attribute world_name=”$(find vitarana_drone)/worlds/drone$(arg static).world”

The world launch file is as follows:

```
<?xml version="1.0"?>
<sdf version="1.6">
  <world name="default">
    <scene>
      <shadows>false</shadows>
      <grid>false</grid>
      <sky>
        <clouds>
          <speed>4</speed>
        </clouds>
      </sky>
    </scene>
    <include>
      <uri>model://sun</uri>
    </include>
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
      <uri>model://edrone</uri>
      <pose>0 0 0 0 0 0</pose>
    </include>
  </world>
</sdf>
```

It is a very simple description of the world and the model to be included written in XML and SDF.



This is our world with the model spawned at origin (0,0,0)

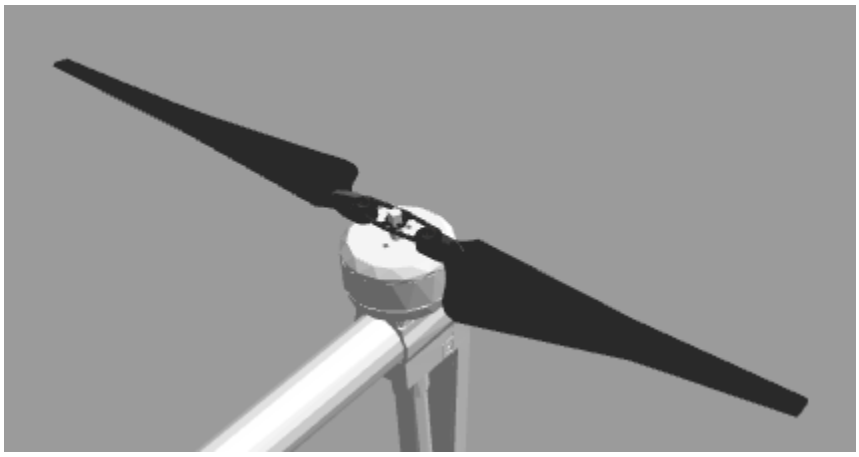
2.5. Propellers and Lift

A propeller is a device with a rotating hub and radiating blades that are set at a pitch to form a helical spiral that, when rotated, exerts linear thrust upon a working fluid, such as water or air.

The fundamental parts of propellers are as follows: -

- **Chord Line**: The chord line of a propeller is an imaginary line drawn through the centre of the blade from its leading edge (at the hub) to its trailing edge (tip).
- **Pitch**: The blades of a propeller are not straight, they are on an angle similar to that of a screw. The pitch is effectively a measure of how far the propeller would move forwards in one revolution. The pitch is used to control the speed of the air leaving the back of the propeller.The pitch is calculated using the formula: pitch = 2.36 diameter height/width.
- **Blade Angle**: This is the angle between the chord line and the plane of rotation and is measured (in degrees) at a specific point along the length of the blade. Pitch is largely determined by blade angle, the two terms are often used interchangeably. An increase or decrease in one is usually associated with an increase or decrease in the other.
- **Angle Of Attack**: This is defined as the angle at which the air strikes the propeller blade. In simple terms the angle of attack can be described as the difference between where a wing is pointing and where it is going. Increasing the angle of attack results in an increase in both lift and induced drag, up to the point of a stall. The twist of a propeller blade is used to maintain a more constant angle of attack along the length of the blade to counteract the differences in blade speed at the hub and the tip of the propeller.

In the case of a quadcopter its propellers exert a linear thrust downward which is proportional to their speeds of rotation.
Depending on the make, weight and angle of pitch propellers have different aerodynamic qualities which make them suitable for different kinds of working fluids.



This was the final propeller used by us in the drone.
It is a propeller with a large diameter, suitable for long flights and has 2 blades which rotate to generate lift.

2.6. Sensors Onboard the Drone

Our drone has onboard:

- A Camera:
  - We haven't used the camera so far but it can be used for a variety of pick & place activities as well as object detection,measurement and precision landing.
- An IMU (Inertial Measurement Unit):
  - 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.
- A GPS (Global Positioning System):
  - The GPS provides us with the altitude of the drone as well as the velocities in the x, y and z directions.

All these sensors are physically attached to the drone and publish their data to rostopics which can be subscribed to in code.

2.7. Plugins in ROS and Gazebo

In ROS plugins are used to provide greater functionality than is normally available, they can be compared to libraries in programming languages but instead for specific robots or applications.
The detailed creation of plugins is beyond the scope of this report, but we're going to discuss the key features of the plugins we have used in the implementation section.

2.8. Roll, Pitch, Yaw and Thrust

- Roll:

  The angle made by the drone with the X-axis is called Roll

- Pitch:

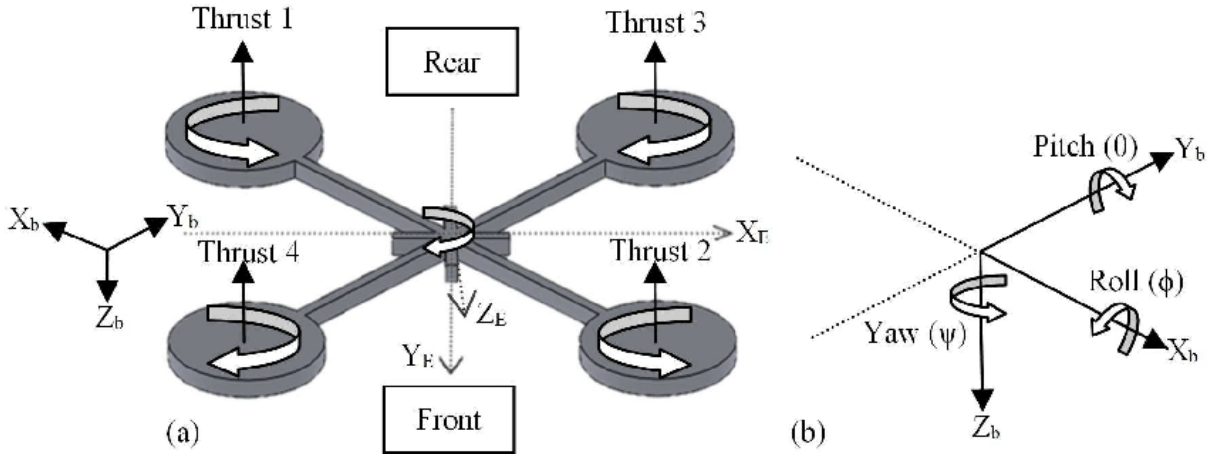
  The angle made by the drone with the Y-axis is called Pitch.

- Yaw:

  The angle made by the drone with the Z-axis is called Yaw.

- Thrust:

  The upward force generated by the rotation of the propellers is called Thrust. Thrust is the main component which leads to propulsion.

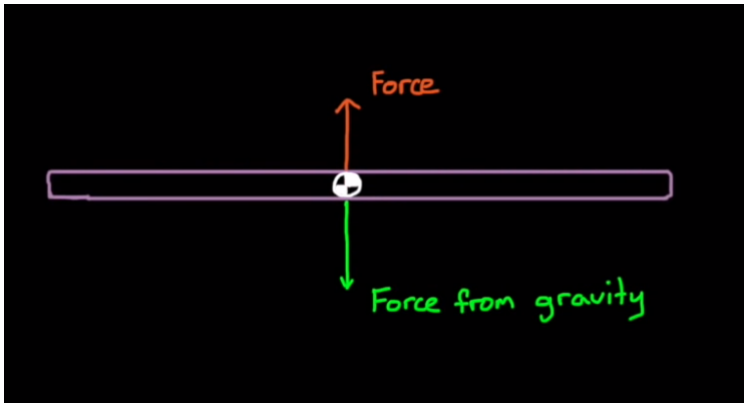


After obtaining the sensor info, the current roll.pitch,yaw(rpy) and altitude of the drone can be calculated. From these the errors in rpy and altitude can be found.

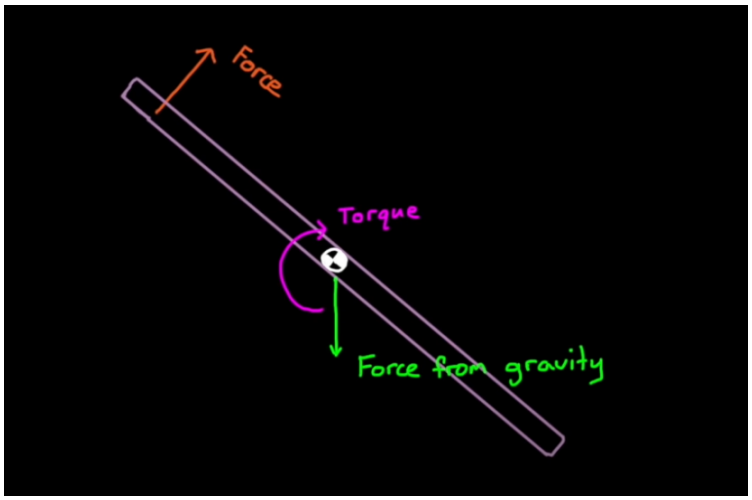
2.9. Motor Mixing Algorithm

The Motor mixing algorithm is the algorithm we use to convert our correction in roll,pitch,yaw and altitude to actual motor speeds which can be sent to the QC propellers in Gazebo using the previously discussed prop_speed rosmsg.

A motor produces thrust by spinning a propeller which pushes air down causing a reaction force that is up. If the motor is placed in a position that the force is applied through the center of gravity of an object then that object will move in pure translation with no rotation at all and if the force of the thrust is exactly equal to and opposite the force of gravity then the object will hover in place.

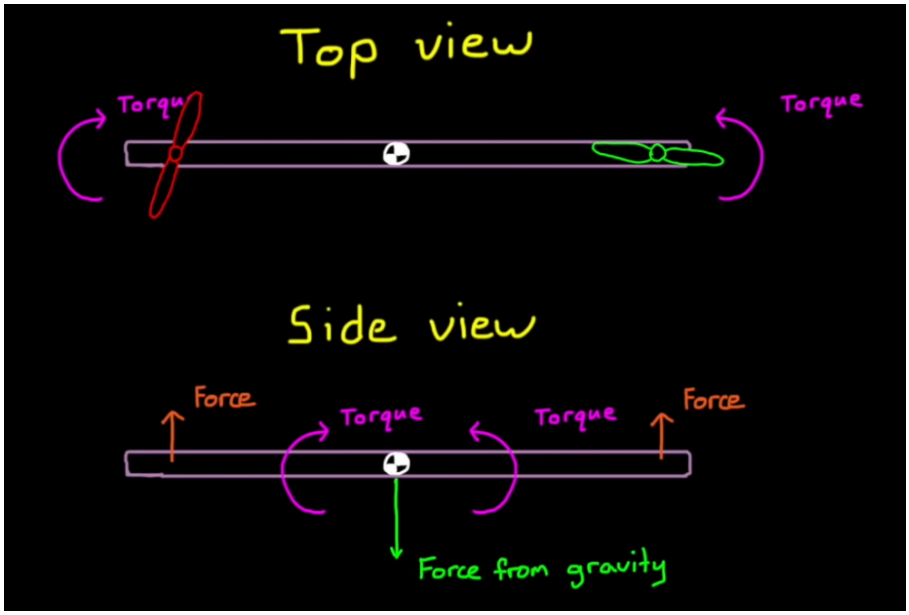


A force at a distance from the center of gravity produces both a translational motion as well as a torque or a rotating moment about the center of gravity and if our motor is attached to the bar as it rotates the torque will stay constant since the distance between the force and the center of gravity stays the same but the force is now no longer always in the opposite direction of gravity and therefore our bar will begin to move off the side and fall out of the sky.

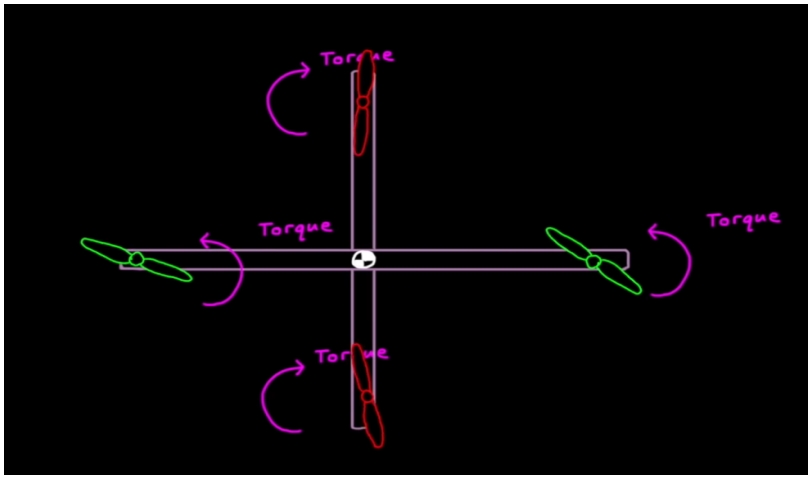


Now if there's a counter force on the opposite side of the center of gravity and each force is half of the force of gravity then the object will again stay stationary because the

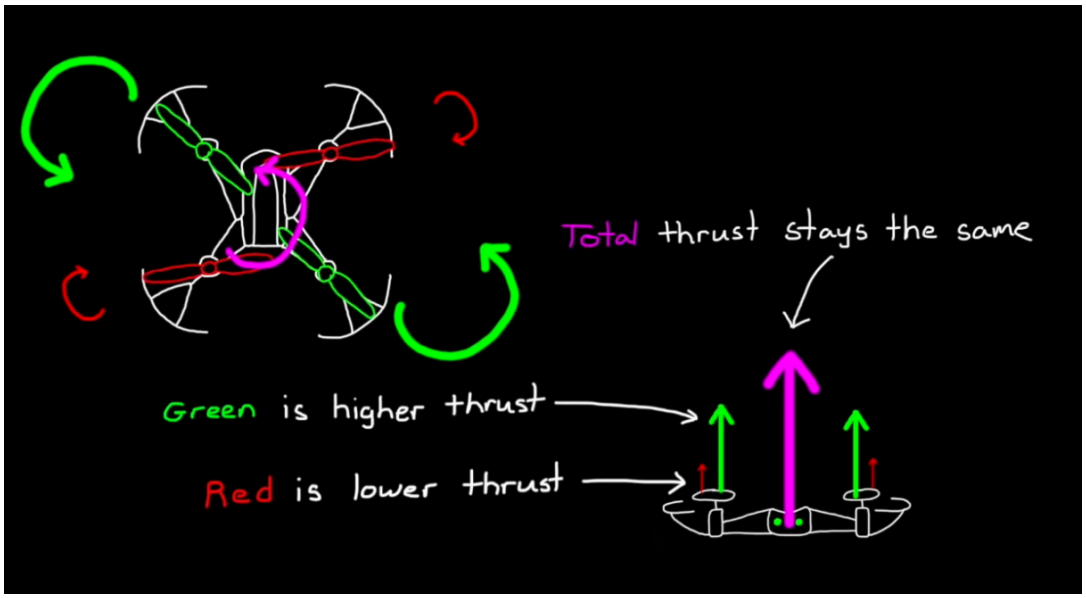
torques and forces will cancel each other out but our actuators don't generate purely force when it generates thrust since it accomplishes thrust by rotating and torquing a propeller which has mass our actuators are also generating a reaction torque that is in the opposite direction and if both of our motors are spinning in the same direction then the torque is doubled and our bar would start to rotate. Now to counter this torque we could spend the two motors in opposite directions and that would work just fine for two dimensions but a bar with only two motors would not be able to generate torques in the third dimension that is we wouldn't be able to roll this bar.



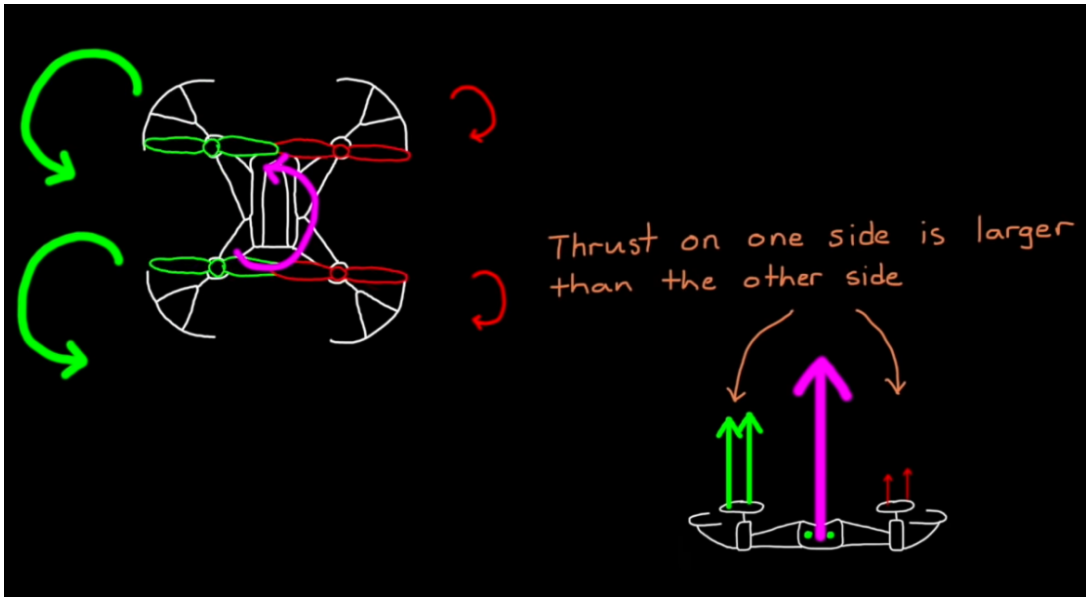
So we add a second bar with two more motors to create our quadcopter. With this configuration we can hover by accelerating each motor until they each produce a force one fourth that of gravity and as long as we have two counter-rotating motors the torques from spinning the propellers will balance out and the drone will not spin. But the configuration should be with opposing motors spinning in the same direction because of how yaw interacts with roll and pitch.



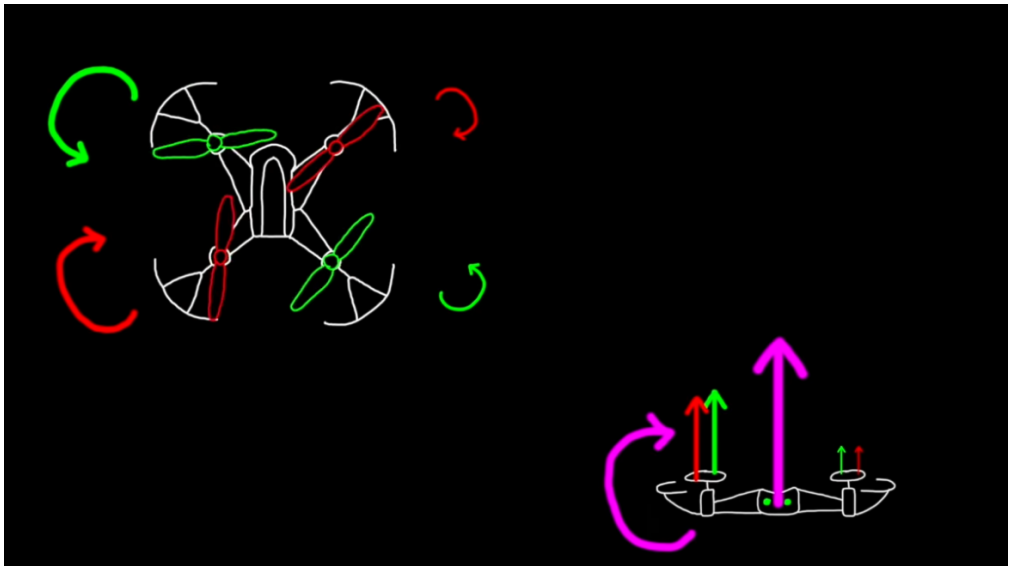
Now, a yaw torque can be created by slowing two motors down that are running in the same direction and speed the other two up resulting in the same total force throughout the maneuver so that we're still hovering and counteracting gravity but the summation of the motor torques is nonzero and the vehicle will spin.So in this way we can yaw without affecting thrust.



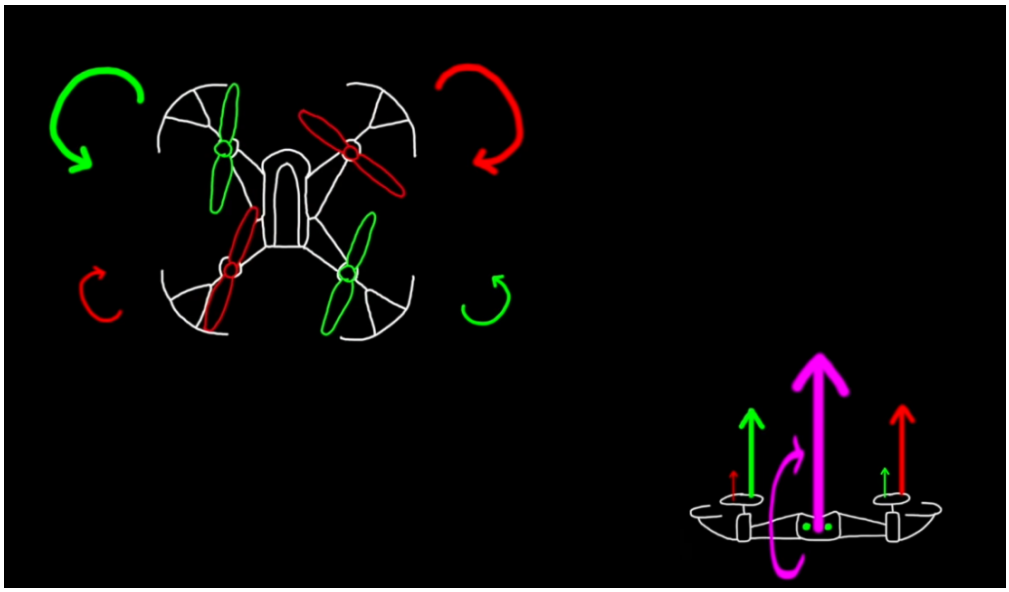
If the rotating motor pairs are on the same side then slowing one pair-down and increasing the other pair will cause an imbalance of forces about the center of gravity and the vehicle will either pitch or roll depending on which side the motor pairs are on



Similarly we can look at roll and pitch-
To roll we decrease one of the left/right pairs and increase the other causing a rolling torque.



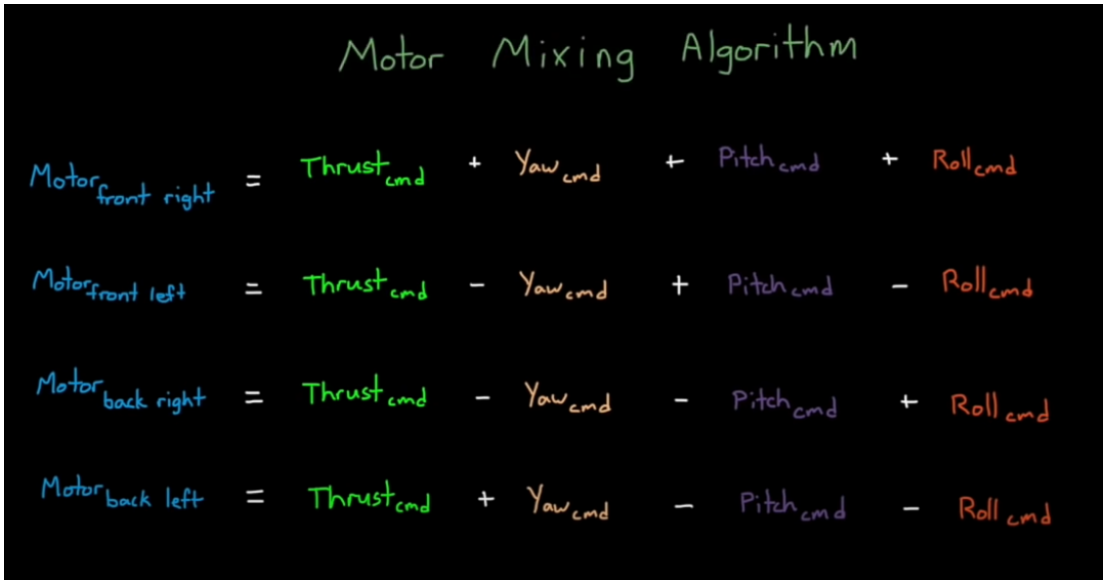
And to pitch we decrease one of the front/back pairs and increase the other causing a pitching torque.



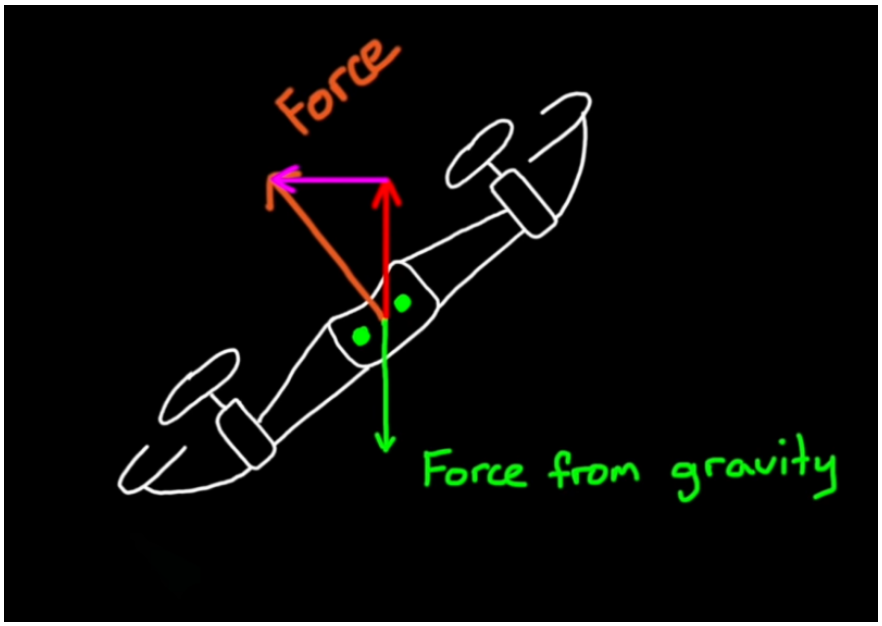
Both of these motions would have no effect on yaw since we're moving counter rotating motors in the same direction and their yaw torque would continue to cancel each other out.

Now to change thrust we need to increase or decrease all four motors simultaneously.

In this way roll pitch yaw and thrust are the four directions that we have direct control over and the commands to the motors would be a mix of the amount of thrust roll pitch and yaw required.



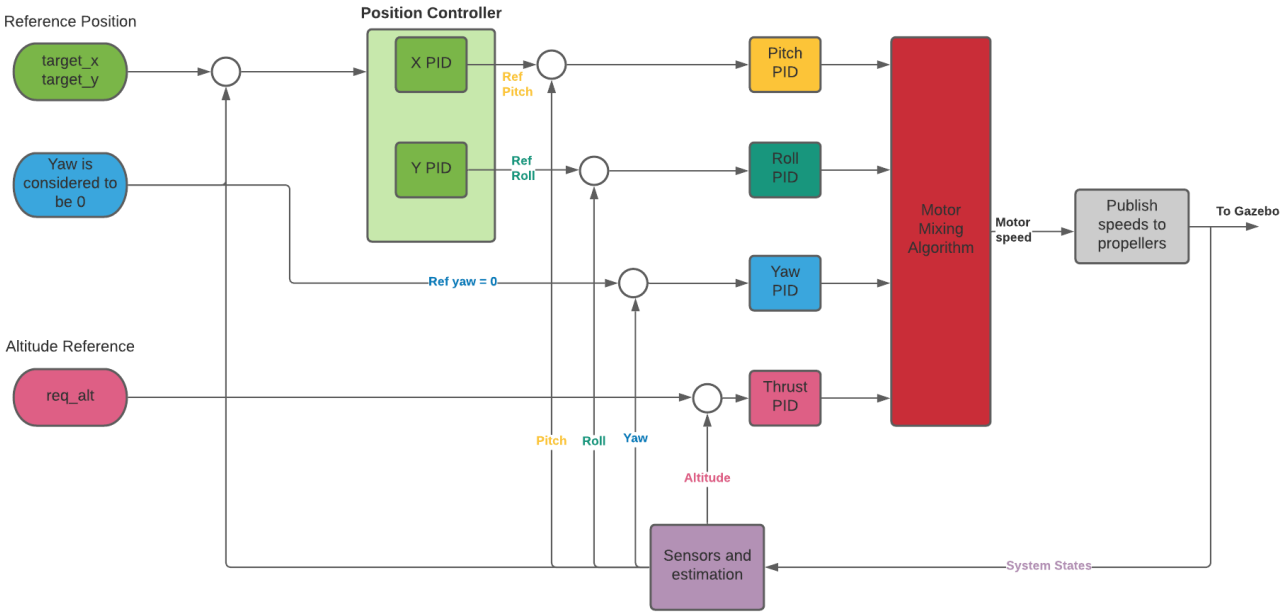
Moving forwards backwards left and right are unactuated motions and the way we get around that is by first rotating into an attitude where the thrust vector is partially in the gravity direction and partially in the direction of travel to accelerate the drone in that way. Now if we wanted to maintain altitude while we do this maneuver then we would increase the thrust so that the vertical component is still cancelling out the downward pull of gravity.



2.10. Control System

A control system manages, commands, directs, or regulates the behavior of other devices or systems using control loops. The control system for our QC is as follows

PID DIAGRAM



The above control loop can be essentially broken down into three controllers.

1. **Roll-Pitch-Yaw Controller** which takes in the target rpy(each being 0 in our case)
2. **Altitude Controller** which takes in the target altitude
3. **Position Controller** which does the two fold job of taking in the target x and y position in the world frame.
   The purpose of all three controllers is to reduce the error between their target values and the corresponding current values of the QC. This is done using a PID controller.

   How the loop works:

   - The loop starts by first taking in the target values of yaw,altitude, X and Y position.
   - Then the position controller kicks in, which depending on the error in X and Y sets the target pitch and roll respectively so as to minimize that error. So now we have the target roll,pitch,yaw and altitude.
   - The rpy and altitude controllers are now activated which again try to minimize the error in each.
   - All these terms are then fed into the motor mixing algorithm which adds them appropriately to the thrust to give final motor speeds.
   - The speeds are then sent to the propellers using ROS.
   - The drone responds accordingly, and then the updated sensor values are fed into the loop again and the cycle repeats.

2.11. PID Tuning

PID stands for Proportional, Integral And Derivative. It is a control loop mechanism which uses the feedback to give a correction term as an output which when provided to a plant helps it achieve a set target.

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

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

As seen above the PID algorithm mainly involves three terms which are respectively multiplied by their gains:-

1. Proportional Term which multiplies Kp directly with the error value.
2. Integration 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.
3. 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.

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.

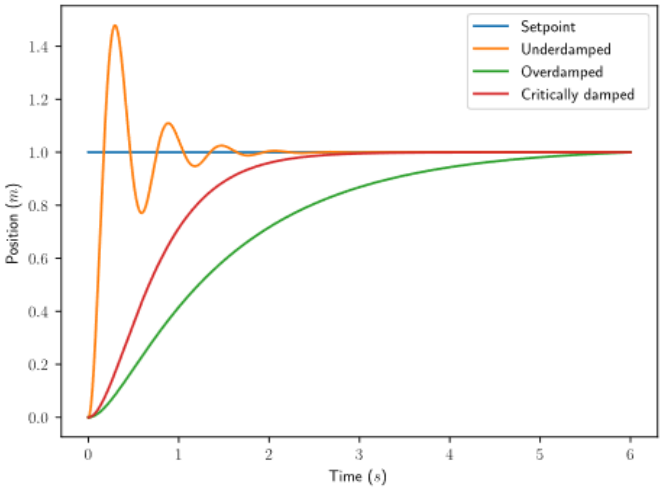
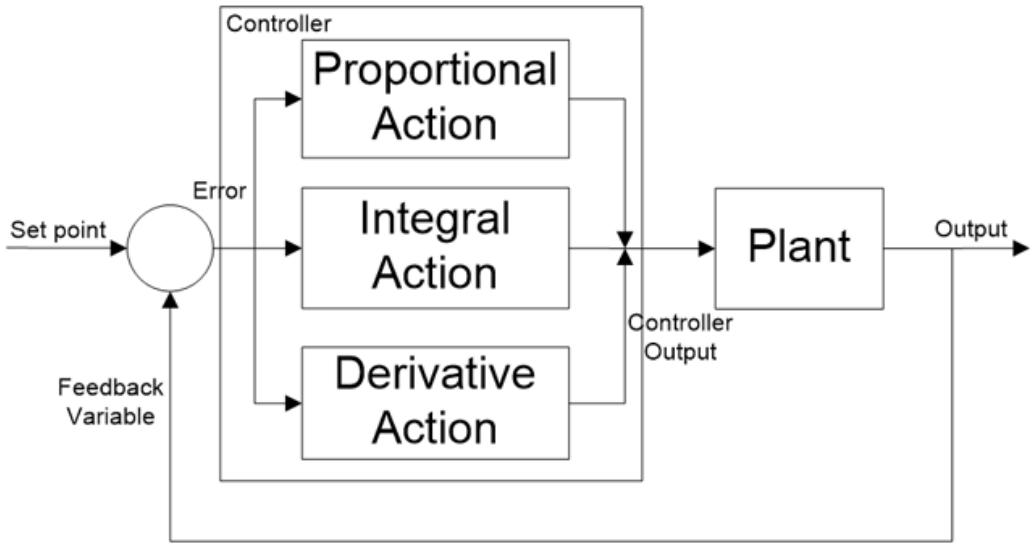


Figure 2.8: PID controller response types



3. Implementation

3.1. Eklavya---Drone package overview

Let's look at the Eklavya---Drone package :

```
📦Eklavya---Drone
 ├ 📂assets
 ├ 📂cfg
 ├ 📂include
 |  └ 📂vitarana_drone
 |  |  ├ 📜gazebo_edrone_propulsion.h
 |  |  ├ 📜gazebo_ros_gps.h
 ├ 📂launch
 |  └ 📜drone.launch
 ├ 📂models
 |  └ 📂edrone
 |  |  ├ 📂materials
 |  |  ├ 📂meshes
 |  |  ├ 📜model.config
 |  |  └ 📜model.sdf
 ├ 📂msg
 ├ 📂scripts
 |  ├ 📜control.py
 |  ├ 📜pid.py
 ├ 📂src
 |  ├ 📜gazebo_edrone_propulsion.cpp
 |  └ 📜gazebo_ros_gps.cpp
 ├ 📂worlds
 |  ├ 📜drone.world
 ├ 📜CMakeLists.txt
 ├ 📜README.md
 └ 📜package.xml
```

1. **cfg :** Contains config files for Sensor Models which work with the dynamic reconfigure package available in ROS.
2. **include :** Contains the necessary header files for propulsion and GPS plugins.
3. **launch :** Contains the drone.launch file which is used to load the model into Gazebo.
4. **models :** Contains the meshes and textures for the vitarana-drone model. These files are auto-generated when the URDF is exported from SW.

5. **msg :** Contains custom messages which are used to control drone functions like the prop_speed msg.
6. **scripts :** Contains the main python scripts used to interact with the simulation of the QC.
7. **src :** Contains the two custom plugins - propulsion and GPS.
8. **world :** Contains the world file which loads in a simple open sky world with a ground. This file is called automatically when the drone.launch file gets launched.
9. **CMakeLists.txt :** Used to link all these files according to their dependencies when the package is built.

3.2. SDF and Launch files

SDFormat (Simulation Description Format), sometimes abbreviated as SDF, is an XML format that describes objects and environments for robot simulators, visualization, and control. Originally developed as part of the Gazebo robot simulator, SDFormat was designed with scientific robot applications in mind.[5]

Within our SDF file we describe all the parts of the robot.
There are a few main structural points of this file, The robot has one base link which is connected to its children prop links through joints.

1. Base Link

```
<!-- ***************************** Links ********************************** -->

  <link name="base_frame">
    <visual name="visual">
      <geometry>
        <mesh><uri>model://edrone/meshes/edrone.obj</uri></mesh>
      </geometry>
    </visual>
    <collision name="base_collision">
      <pose>0 0.028 0.37 0 0 0</pose><geometry><box><size>0.466 0.48 0.2</size></box></geometry>
    </collision>
    <collision name="stand_1">
      <pose>0.46 0.46 0.2 0 0 0</pose> <geometry><box><size>0.09 0.09 0.4</size></box> </geometry>
    </collision>
    <collision name="stand_2">
      <pose>0.46 -0.46 0.2 0 0 0</pose> <geometry><box><size>0.09 0.09 0.4</size></box> </geometry>
    </collision>
    <collision name="stand_3">
      <pose>-0.46 -0.46 0.2 0 0 0</pose> <geometry><box><size>0.09 0.09 0.4</size></box> </geometry>
    </collision>
    <collision name="stand_4">
      <pose>-0.46 0.46 0.2 0 0 0</pose> <geometry><box><size>0.09 0.09 0.4</size></box> </geometry>
    </collision>
    <inertial>
      <mass>1</mass>
      <inertia>
        <ixx>0.08666</ixx><ixy>0</ixy><ixz>0</ixz><iyy>0.08666</iyy><iyz>0</iyz><izz>0.16666</izz>
      </inertia>
      <pose>0 0 0.35 0 0 0</pose>
```

This base link is called “base_frame”. It has 4 stands whose collision characteristics have been described in the above snippet. Further, the mass has been described along with the inertia in the <mass> and <inertia> tags.

2. Prop Links

   These links describe the propellers of the drone

```
<link name="prop1">
    <pose>0.46 0.46 0.41  0 0 -0.785398</pose>
    <inertial>
      <mass>0.1</mass>
      <inertia><ixx>0.00001</ixx><ixy>0</ixy><ixz>0</ixz><iyy>0.001</iyy><iyz>0</iyz><izz>0.001</izz></inertia>
      <pose>0 0 0.015 0 0 0</pose>
    </inertial>
    <visual name="visual">
      <geometry>
        <mesh><uri>model://edrone/meshes/prop_ccw.dae</uri></mesh>
      </geometry>
    </visual>
    <collision name="collision">
      <pose>0 0 0.015 0 0 0</pose>
      <geometry>
        <box><size>0.696 0.1 0.03</size></box>
      </geometry>
    </collision>
  </link>
```

   Each has its <mass> and <inertia> described, it also has its <pose> specified along with its <collision> characteristics and the look of the propeller is also specified in the <visual> tag, it takes its mesh from the model file in a .dae format

3. Prop Joints

   These joints are responsible for joining the base_frame and propellers together. They also describe the relative movement that is allowed between these two elements.

```
<!-- ************************** Joints ******************************** -->

  <joint name="prop1_joint" type="revolute">
    <parent>base_frame</parent>
    <child>prop1</child>
    <pose>0 0 0  0 0 0</pose>
    <axis>
      <xyz>0 0 1</xyz>
    </axis>
  </joint>
```

All the propellers are linked to the base_frame using revolute joints, these allow the propellers to rotate around a specified axis. In this case it is the Z-axis around which the propellers rotate.

4. Plugins

Plugins are also described in the robot's SDF file. This is how plugins are loaded along with the robot

```
<plugin name="gazebo_edrone_propulsion" filename="libgazebo_edrone_propulsion.so">
    <namespace_model>/edrone</namespace_model>
    <activate_pid_control>yes</activate_pid_control>
    <prop_kp> 1 </prop_kp>
    <prop_ki> 0 </prop_ki>
    <prop_kd> 0.0 </prop_kd>
    <robotNamespace>edrone</robotNamespace>
    <bodyName_1>prop1</bodyName_1>
    <bodyName_2>prop2</bodyName_2>
    <bodyName_3>prop3</bodyName_3>
    <bodyName_4>prop4</bodyName_4>
    <topicName>pwm</topicName>
  </plugin>
```

This is the gazebo_edrone_propulsion plugin. It describes the prop_kp, ki and kd values which are then used in the code of the plugin. Additionally it specifies a topic name from which it receives commands.
Additionally we have plugins for the GPS, IMU and the Camera as well. Additionally all of these elements have joints and collision attributes too as they are in the form of physical devices.

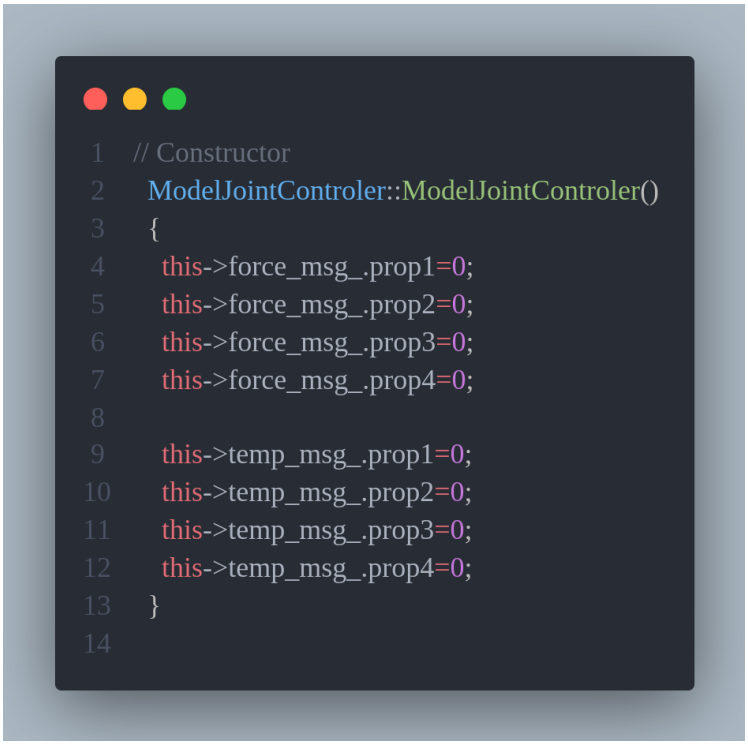
3.3. Plugins used

3.3.1. gazebo_ros_gps.cpp

We use this plugin written by Johannes Meyer, TU Darmstadt to get the values of latitude, longitude and altitude from the GPS we have onboard our drone.
The explanation of this plugin is beyond the scope of the current project. A link is included in references for those interested in the mathematics. [3].

3.3.2. gazebo_edrone_propulsion.cpp

This plugin first initialises a ModelJointController to give force values to the four propellers named prop1, prop2, prop3 and prop4



```
// Constructor
ModelJointControler::ModelJointControler()
{
  this->force_msg_.prop1=0;
  this->force_msg_.prop2=0;
  this->force_msg_.prop3=0;
  this->force_msg_.prop4=0;

  this->temp_msg_.prop1=0;
  this->temp_msg_.prop2=0;
  this->temp_msg_.prop3=0;
  this->temp_msg_.prop4=0;
}
```

It creates a subscriber called /edrone/pwm which receives messages of the type prop_speed. From these messages it takes in 4, 64 bit float values which give speeds to each propeller:

```
// Create subscriber for prop speed
  ros::SubscribeOptions so = ros::SubscribeOptions::create<vitarana_drone::prop_speed>(
    this->topic_name_,1,
    boost::bind( &ModelJointControler::OnRosMsg_prop_speed,this,_1),
    ros::VoidPtr(), &this->queue_);
  this->sub_ = this->rosnode_->subscribe(so);

  // Spin up the queue helper thread.
  this->callback_queue_thread_ = boost::thread( boost::bind( &ModelJointControler::QueueThread,this ) );

  ROS_INFO("Loaded Plugin with parent...%s", this->model->GetName().c_str());
```

It also limits the speeds given to 1024 to avoid errors resulting from excessive speed and doesn't allow the speed to go below 0 either

```
void ModelJointControler::OnRosMsg_prop_speed(const vitarana_drone::prop_speedConstPtr &_msg)
  {

    this->temp_msg_.prop1=_msg->prop1;
    this->temp_msg_.prop2=_msg->prop2;
    this->temp_msg_.prop3=_msg->prop3;
    this->temp_msg_.prop4=_msg->prop4;

// get in range from 0 to 1024
    if (this->temp_msg_.prop1>=1024.0)
      this->temp_msg_.prop1=1024;
    if (this->temp_msg_.prop2>=1024.0)
      this->temp_msg_.prop2=1024;
    if (this->temp_msg_.prop3>=1024.0)
      this->temp_msg_.prop3=1024;
    if (this->temp_msg_.prop4>=1024.0)
      this->temp_msg_.prop4=1024;

    if (this->temp_msg_.prop1<0)
      this->temp_msg_.prop1=0;
    if (this->temp_msg_.prop2<0)
      this->temp_msg_.prop2=0;
    if (this->temp_msg_.prop3<0)
      this->temp_msg_.prop3=0;
    if (this->temp_msg_.prop4<0)
      this->temp_msg_.prop4=0;
```

The above function is called when the user sends a message of type->prop_speed

3.4. Control System Code

The control system code is effectively divided into two python scripts and follows the logic mentioned in 2.10. The function of the program is to ask the user for a target altitude and the drone should hover at that altitude. The target X and target Y have been kept 0 currently.

1. control.py:

   Gets the respective sensor information from the GPS(altitude and velocity), IMU(roll, pitch, yaw)

```
def control():
    # define global values for required parameters to avoid resetting to 0
    global altitude, thrust, speed

    # initialize node
    rospy.init_node("altitude", anonymous = False)

    # Creating subscribers to get all relevant sensor data
    gps_sub = message_filters.Subscriber("/edrone/gps", NavSatFix)
    vel_sub = message_filters.Subscriber("/edrone/gps_velocity", Vector3Stamped)
    imu_sub = message_filters.Subscriber("/edrone/imu/data", Imu)
    ts = message_filters.TimeSynchronizer([gps_sub, vel_sub, imu_sub], 2)

    #one of these publishers is slower than the others
    #which is why the messages are loading relatively slowly
    ts.registerCallback(alt_control)
    rospy.spin()
```

Gets the respective PIDs of all controllers which can be set by subscribing to their respective topics using ROS terminal or RQT.
(Default values have also been set in case no user values are given)

```
# Subsribe to all required topics to get PID for all controllers
rospy.Subscriber("alt_pid", Float64MultiArray, setPID_alt)
rospy.Subscriber("roll_pid", Float64MultiArray, setPID_roll)
rospy.Subscriber("pitch_pid", Float64MultiArray, setPID_pitch)
rospy.Subscriber("yaw_pid", Float64MultiArray, setPID_yaw)
rospy.Subscriber("x_pid", Float64MultiArray, setPID_x)
rospy.Subscriber("y_pid", Float64MultiArray, setPID_y)
rospy.Subscriber("vel_x_pid", Float64MultiArray, setPID_vel_x)
rospy.Subscriber("vel_y_pid", Float64MultiArray, setPID_vel_y)

```

2. pid.py

   First the position controller is called which, depending upon the error in X and Y position sets the pitch and roll to compensate for the errors after going through the PID loop.
   Note: The output correction terms are limited to prevent the drone from swinging too much. For high setpoint_roll and setpoint_pitch values it becomes difficult for the drone to recover to a stable state after inclining at these angles.

   ### Why the need for velocity controllers?

   Once the drone gets into a certain region around the target point the velocity controller kicks in. The code prioritizes the velocity controller over the position controller in this region. The reason for this is to prevent the drone from overshooting its target.

```
if(dTime >=sample_time):
    pMem_x = kp_x*err_x
    pMem_y = kp_y*err_y
    pMem_vel_x = kp_vel_x*err_x
    pMem_vel_y = kp_vel_y*err_y


    iMem_x += err_x*dTime
    iMem_y += err_y*dTime
    iMem_vel_x += err_vel_x*dTime
    iMem_vel_y += err_vel_y*dTime

    if(iMem_vel_x>250): iMem_vel_x = 250
    if(iMem_vel_x<-250): iMem_vel_x=-250
    if(iMem_vel_y>10): iMem_vel_y = 10
    if(iMem_vel_y<-10): iMem_vel_y=-10

    if(iMem_x>10): iMem_x = 10
    if(iMem_x<-10): iMem_x=-10
    if(iMem_y>10): iMem_y = 10
    if(iMem_y<-10): iMem_y=-10

    dMem_x = dErr_x/dTime
    dMem_y = dErr_y/dTime
    dMem_vel_x = dErr_vel_x/dTime
    dMem_vel_y = dErr_vel_y/dTime

    if(dMem_vel_x>100): dMem_vel_x = 100
    if(dMem_vel_x<-100): dMem_vel_x=-100
    if(dMem_vel_y>100): dMem_vel_y= 100
    if(dMem_vel_y<-100): dMem_vel_y=-100

prevErr_x = err_x
prevErr_y = err_y
```

```
output_x = pMem_x + ki_x*iMem_x + kd_x*dMem_x
  output_y = pMem_y + ki_y*iMem_y + kd_y*dMem_y
  output_vel_x = pMem_vel_x + ki_vel_x*iMem_vel_x + kd_vel_x*dMem_vel_x
  output_vel_y = pMem_vel_y + ki_vel_y*iMem_vel_y + kd_vel_y*dMem_vel_y
```

```
  # Limiting output terms to prevent from loiosing control
  if(output_x>5):
     output_x = 5
     print('MAXIMUM HIT')
  if(output_x<-5):
     output_x = -5
     print('MINIMUM HIT')

  if(output_y>5):
     output_y = 5
     print('MAXIMUM HIT')
  if(output_y<-5):
     output_y = -5
     print('MINIMUM HIT')

# Defining region inside which the drone should prioratize making velocity 0 raher than making errors in X and Y
  if(abs(err_x) > 2 and abs(vel_x) < 1.5):
     setpoint_pitch = -(output_x)
  else:
     setpoint_pitch = -(output_vel_x)

  if(abs(err_y) > 2 or abs(vel_y) < 1.5):
     setpoint_roll = output_y
  else:
     setpoint_roll = output_vel_y
```

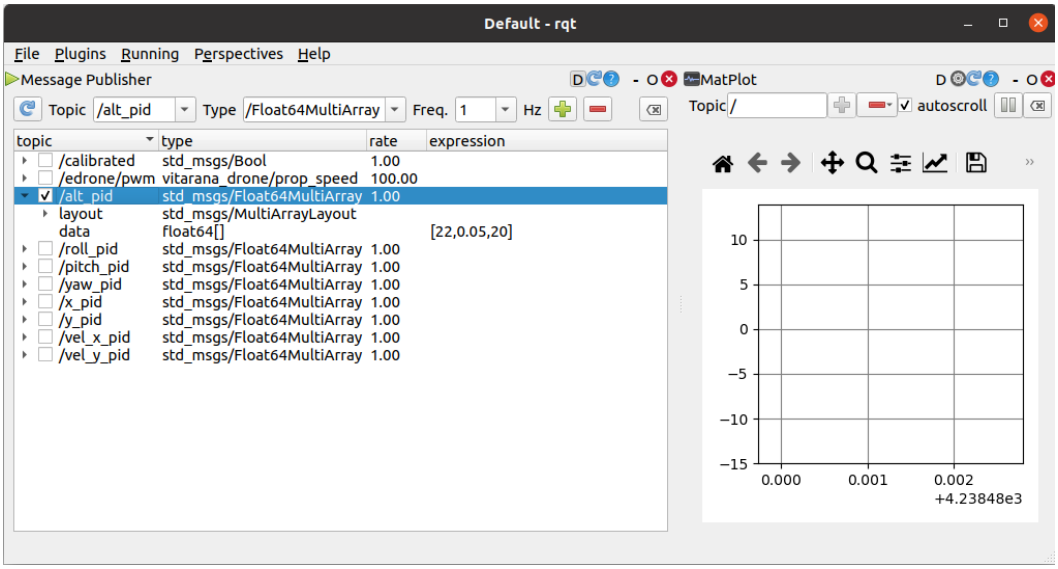
After getting the setpoints for pitch and roll a similar PID algorithm is used for the roll, pitch, yaw and altitude. A hover speed variable is also defined- this is basically the speed the propellers should have in order to generate enough upward thrust to exactly counter the downward gravity force(the correction in altitude is added to this term to get the final thrust).

Finally the motor mixing algorithm[2.9] is used to combine all these terms. And then the final speed message ios published to the propellers using ROS.

```
# Setting the output prop_speed msg
    speed = prop_speed()
    speed.prop1 = (thrust - output_yaw + output_pitch - output_roll)
    speed.prop2 = (thrust + output_yaw + output_pitch + output_roll)
    speed.prop3 = (thrust - output_yaw - output_pitch + output_roll)
    speed.prop4 = (thrust + output_yaw - output_pitch - output_roll)
```

3.5. Tuning and Testing using RQT

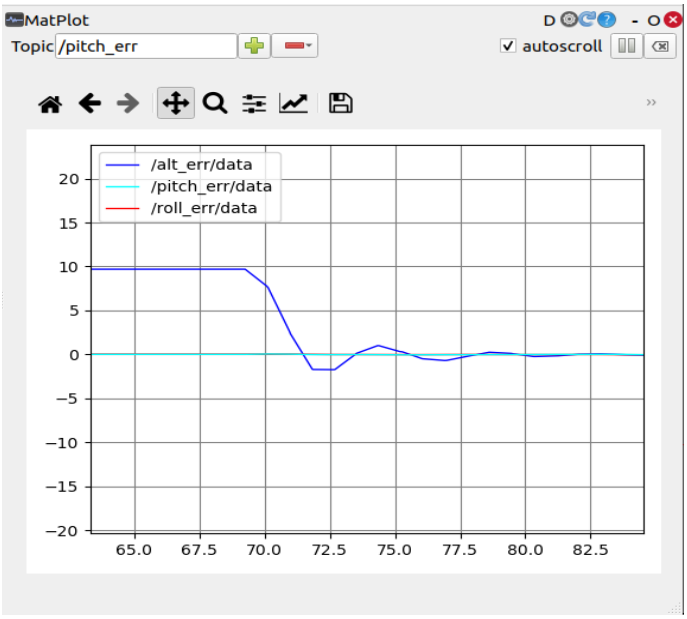
To stabilise the drone we need to configure the PID values perfectly. In order to ease our work, we use RQT to do the same. The advantage of using RQT is that we do not need to stop the program to manually change the values every time. We simply change the values in RQT and the changes are reflected in the program as well.
In our Python files, we have described Subscribers for each of the PID values. We create Publishers to these Subscribers in RQT which we then use to send values for the PIDs.



Example of values of kp, ki and kd being sent through RQT

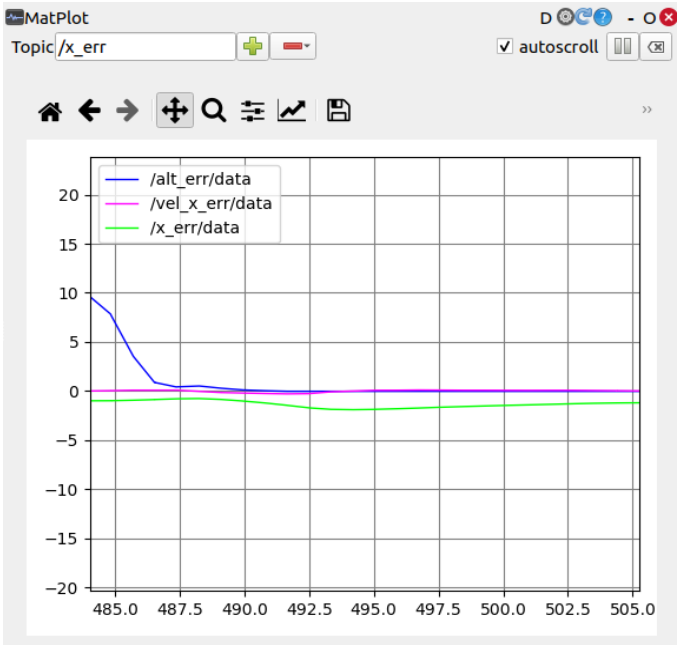
We can edit the values in the data field according to our convenience.

Further RQT is a very powerful tool with MatPlotLib built into it. We can create Publishers in our program and graph the values sent in RQT.



Error in altitude, roll and pitch is being graphed as the drone takes off

Further we can graph errors in velocity and position as well: -



Errors of velocity and position in X with time

4. Application

4.1. Precision Landing

The basic idea of precision landing is to have a distinguishable platform for the drone to land on if detected by it. Using the onboard camera on the drone and various Computer Vision libraries like OpenCV, Keras, MatPlotLib, etc available in Python precision landing can be achieved.

4.2. Pick & Place

Pick and place involves adding a Gripper arm to the drone which allows it to pick and place feasible objects. While the gripper functionality was already available in the Vitarana-Drone we decided to ignore it for now.

Drones used for agricultural work use both precision landing(or object detection to be precise) and pick & place functionality to provide seeds or pick crops to deliver to a Silo.

4.3. Mapping and Surveillance

Using the numerous onboard sensors, mapping works by acquiring hundreds of aerial images and then 'stitching' them together digitally with specialized mapping software to make a larger more accurate composite image.

Many naturally restricted regions of the world like oceans, mountains, thick forests, etc. have been mapped and studied using drones. Military drones also make use of drones for surveillance purposes.

5. Conclusion and Future Work

5.1. Conclusion

In conclusion, drone design is a balancing act of PID variables. However once this balancing act is completed the drone will fly in a stable manner forevermore. We faced a lot of challenges with our initial model, however we couldn't solve them all to the required degree as we were still new to ROS and Gazebo. Once we implemented the Vitarana Drone though, we managed to get work done in a fast and efficient manner.
Implementing the control and PID files was interesting and fruitful work.
Over these past 4 weeks we've learnt a lot about drones as well as control systems in general. This knowledge of theory as well as the first-hand experience in designing as well as implementing will prove invaluable in the future.

5.2. Flying to Specified Coordinates

While the drone remains stable above origin or co-ordinates close to the origin. It cannot remain stable for long periods of time above coordinates which are far away from its initial position.
These issues need to be ironed out by fine-tuning the PID variables.

5.3. Precision Landing

We have not implemented a definite landing system as of now. One way to do it would be to command the drone to reduce its altitude to a small amount and subsequently switching off the motors.
For precision landing, we need the drone to work together with the camera in order to identify the correct spot as well as maneuver itself to the point in order to then land.

References

[0] http://gazebosim.org/tutorials?tut=build_world
[1] https://youtube.com/playlist?list=PLAY546jcY_9BSwmzHlRRLo3KrN8FxrFb8
[2] http://wiki.ros.org/ROS/Tutorials
[3] http://wiki.ros.org/ublox
[4] https://github.com/smitkesaria/vitarana_drone
[5] http://sdformat.org/