Dayananda Sagar College of Engineering

Department of Electronics and Communication Engineering

Shavige Malleshwara Hills, Kumaraswamy Layout, Bengaluru – 560 078.

(An Autonomous Institute affiliated to VTU, Approved by AICTE & ISO 9001:2008 Certified)

Accredited by National Assessment and Accreditation Council (NAAC) with 'A' grade

Assignment

Program: B.E.	**Branch:** ECE
Course: Programming in Python	**Semester :** 5th
Course Code: 18EC5DEBPP	**Date:** 01/01/2021

A Report on

"ALTITUDE CONTROLLER AND POSITION CONTROLLER OF A DRONE"

Submitted by

USN	**NAME**
1DS19EC410	**G CHANDAN**

Faculty In-charge

Prof. Shashi Raj K

Signature of Faculty In-charge

1. Introduction:

Drones or Unmanned Aerial Vehicles (UAVs) come in two variants – fixed wing and rotary drones. Rotary drones or multirotor drones consist of various configurations, some common ones are the, helicopter, four rotor quadcopter and six rotor hexcopter. Each rotor type has a specific usage and is useful for specific applications. The commonly used type of multirotor is the quadcopter (often shortened to quad). This is because it is a very mechanically simple system. In quads, each motor spins in the opposite direction of the adjacent motor as shown in Figure 1. This allows it to achieve vertical lift.

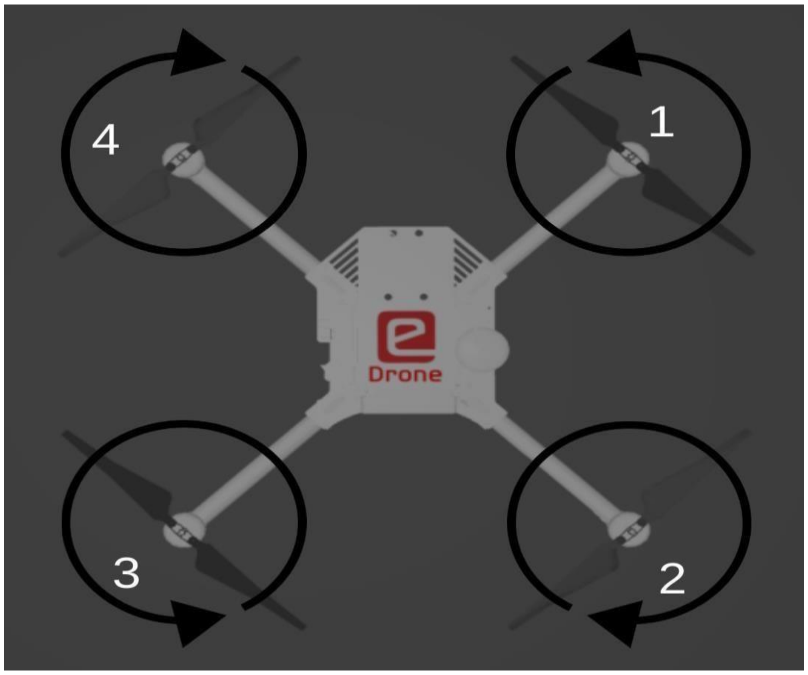


Figure 1: Rotation of propellers

In order to maintain its 'pose' in flight and stay stable, a quadcopter relies heavily on sensors that constantly monitor the quad's 'attitude'. These sensors provide feedback to the quad using which the flight controller makes corrections in the motor spin speed and thus adjusting and shifting the quad in flight so that it remains stable.

Sensors:

- **Inertial Measurement Unit (IMU)**: This sensor is a combination of and accelerometer, which measures linear acceleration (i.e. in X,Y & Z axis relative to the sensor) and a gyroscope which measures angular velocity in three axes, roll, pitch and yaw. Typically a magnetometer is also present which by measuring the earth's magnetic field, serves to give a reference direction.
- **Barometer**: Barometers are used to measure the air pressure and hence help flight controllers to predict the height of the quad from the ground. Barometers generally come handy when the quad is at large enough heights.
- **Ultrasonic Sensor**: Ultrasonic sensors help give precise height of the quad from the ground. Ultrasonic sensors are extremely useful when flying a quad within 50cm from the ground. Beyond that height it is recommended to rely on the barometer for estimating the quad height with respect to the ground.
- **Time of Flight Sensor**: It is a distance sensor which uses a laser to estimate the distance.
- **GPS Receiver**: Incorporating this on the UAV enables it to locate itself using the Global Positioning System and other satellite positioning systems.

Quadcopter Motion:

A quadcopter's thrust, roll, pitch and yaw is changed by manipulating the angular velocity of the motors. Following details how to move the quad by manipulating the angular velocity of the motors:

- **Thrust/Throttle -** In order to change the quad's height, we can decrease or increase the velocity of all 4 motors. Reducing the velocity of all the motors lowers the quad height, while increasing the velocity increases the height of the quad with respect to the ground.
- **Pitch -** By changing the motor speed of the front and back motors, we can move the quad forward and backward. Increasing the speed of the forward motors moves the quad back while increasing the speed of the back motors moves the quad forward.
- **Roll -** To move the quad left or right, we simply manipulate the left and right motors. By increasing the speed of the two left motors the quad bends to the right. By increasing the speed of the two right motors the quad bends to the left.

- **Yaw -** By changing the speed of the alternate motors, the quad yaws to the left or right respectively.

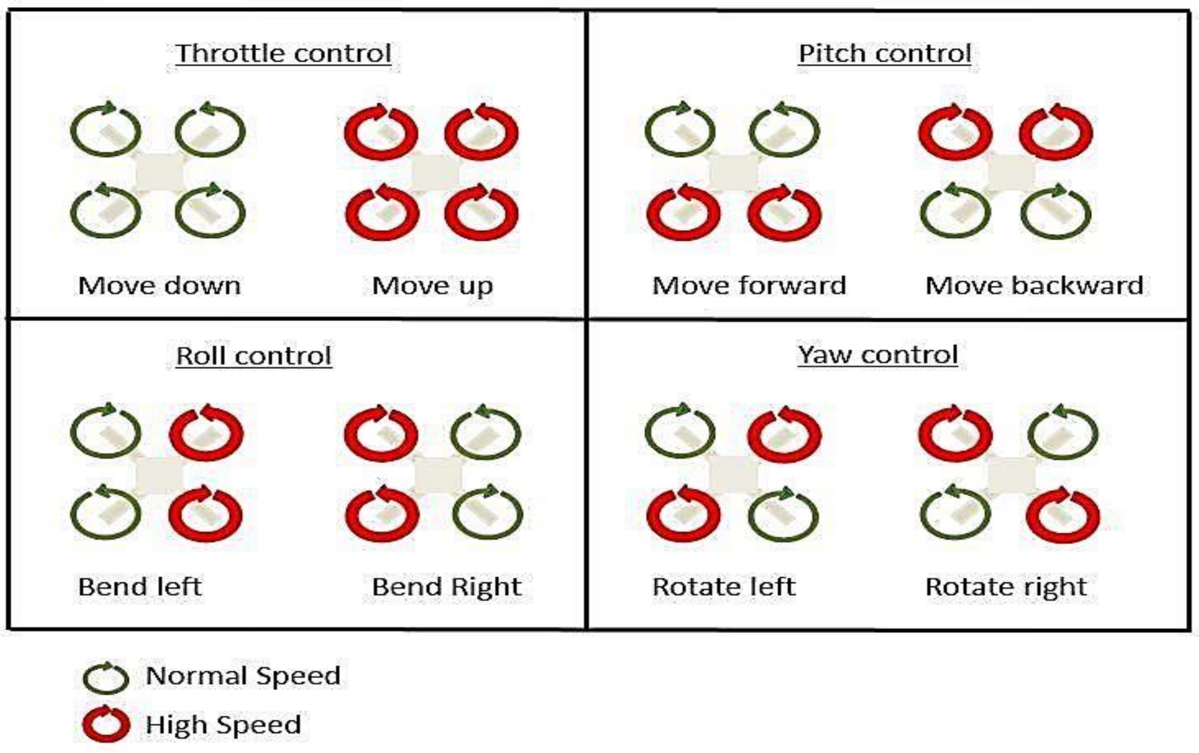


Figure 2: Motion in 3D space

A quadcopter's motion in 3D space is explained visually in Figure 2.

Flight Controller:

A flight controller is responsible for issuing commands to the motors as per the required motion. The flight controller relies on sensor data to generate accurate motion commands so that the UAV maintains its pose as it moves from Point A to Point B. Commonly used flight controllers in quadcopters are PixHawk, KK and MultiWii.

PID:

The PID or Proportional Integral Derivative algorithm is a widely used control loop feedback mechanism in control theory. At the crux of the algorithm an error value is calculated $e(t)$ based on the difference between a set point or a desired point and the measured current point. A correction value is calculated based on

Proportional, Derivative and Integral terms and is then applied to the system to reduce the error value. PID can be applied to any system oscillating like a pendulum and desires to maintain a center reference.

Common examples in robotics that use the PID algorithm are: line following robot, object tracking using image processing and maintaining UAV pose during flight.

Here in we keep track of the error over time i.e. sum up the errors over a specified sampling time.

Iterm = (Iterm + error) * Ki

Further explanation regarding implementation of this Iterm is given in the code. Note that Ki is calculated keeping the sampling time in consideration. This output will be further added or subtracted to the offset pwm as the need be to give the final output.

output = Kp*error + Iterm + Kd*(error - previous error)

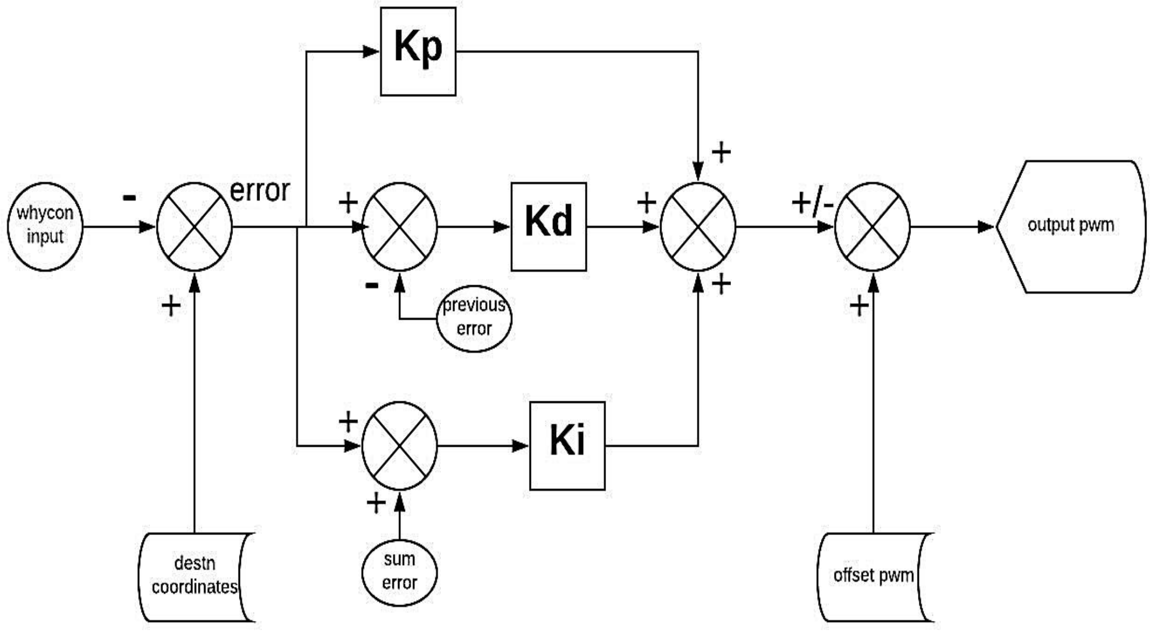


Figure 3: Block diagram of PID for eDRONE

2. Algorithm:

- First setting environment for simulation in “Gazebo”.
- Importing custom edrone libraries.
- Initialize all the parameters to zero(default).
- Declaring publishers and subscribers necessary for communication.
- Obtaining different parameters values like GPU, IMU, ultrasonic etc.,
- Conversion Quaternion to Euler angles which is obtained for IMU for easy computation.
- Setting a stabilization point for drone.
- Compute error, change in error and sum of error from PID algorithm at the set sample rate.
- The above calculated error values is used to generate PWM which in turn control propeller speed to get drone into motion.
- By using plot-jogglers and PID tuners we adjust the stabilization point for effective response.

3. Requirements:

- Ubuntu 18.04.
- ROS Melodic Morenia.
- Gazebo.
- Proportional-integral-derivative (PID) controllers.
- Sensors & actuators present on *eDrone* .

4. Program:

Altitude controller:

```
#!/usr/bin/env python

# Importing the required libraries
from vitarana_drone.msg import *
from pid_tune.msg import PidTune
from sensor_msgs.msg import Imu
from std_msgs.msg import Float32
import rospy
import time
import tf


class Edrone():
    """docstring for Edrone"""
    def__init_(self):
        rospy.init_node('attitude_controller')  # initializing ros node with name drone_control
        self.drone_orientation_quaternion = [0.0, 0.0, 0.0, 0.0]
        self.drone_orientation_euler = [0.0, 0.0, 0.0]
        self.setpoint_cmd = [0.0, 0.0, 0.0, 0.0]
        self.setpoint_euler = [0.000000000, 0.000000000, 0.000000000]

        self.pwm_cmd = prop_speed()
        self.pwm_cmd.prop1 = 0.0
        self.pwm_cmd.prop2 = 0.0
        self.pwm_cmd.prop3 = 0.0
        self.pwm_cmd.prop4 = 0.0
        self.Kp = [0, 0.0, 0]
        self.Ki = [0, 0.0, 0]
        self.Kd = [0, 0.0, 0]
        self.correct_roll = 0.0
        self.correct_pitch = 0.0
        self.correct_alt = 0.0
        self.throttle = 0.0
        self.error_sum = [0.0, 0.0, 0.0]
        self.last_alt_error = 0.0
        self.alt_error_sum = 0.0
        self.last_yaw_error = 0.0
        self.yaw_error_sum = 0.0
        self.last_pitch_error = 0.0
        self.pitch_error_sum = 0.0
        self.last_roll_error = 0.0
        self.roll_error_sum = 0.0
        self.max_values = 1023
        self.min_values = 0
        self.last_time = 0.0
        error = 0.0

        # This is the sample time in which you need to run pid. Choose any time which you seem
fit. Remember the stimulation step time is 50 ms
        self.sample_time = 0.04  # in seconds

        # Publishing /edrone/pwm, /roll_error, /pitch_error, /yaw_error
        self.pwm_pub = rospy.Publisher('/edrone/pwm', prop_speed, queue_size=1)
        self.roll_pub = rospy.Publisher('/roll_error', Float32, queue_size=1)
        self.pitch_pub = rospy.Publisher('/pitch_error', Float32, queue_size=1)
        self.yaw_pub = rospy.Publisher('/yaw_error', Float32, queue_size=1)
        self.alt_pub = rospy.Publisher('/alt_error', Float32, queue_size=1)

        # Subscribing to /drone_command, imu/data, /pid_tuning_roll, /pid_tuning_pitch,
/pid_tuning_yaw
        rospy.Subscriber('/drone_command', edrone_cmd, self.drone_command_callback)
        rospy.Subscriber('/edrone/imu/data', Imu, self.imu_callback)
        rospy.Subscriber('/pid_tuning_roll', PidTune, self.roll_set_pid)
        rospy.Subscriber('/pid_tuning_pitch', PidTune, self.pitch_set_pid)
        rospy.Subscriber('/pid_tuning_yaw', PidTune, self.yaw_set_pid)
        rospy.Subscriber('/pid_tuning_altitude', PidTune, self.alt_set_pid)

    def imu_callback(self, msg):
        self.drone_orientation_quaternion[0] = msg.orientation.x
        self.drone_orientation_quaternion[1] = msg.orientation.y
        self.drone_orientation_quaternion[2] = msg.orientation.z
        self.drone_orientation_quaternion[3] = msg.orientation.w

     def drone_command_callback(self, msg):
        self.setpoint_cmd[0] = msg.rcRoll
        self.setpoint_cmd[1] = msg.rcPitch
        self.setpoint_cmd[2] = msg.rcYaw
        self.setpoint_cmd[3] = msg.rcThrottle
```

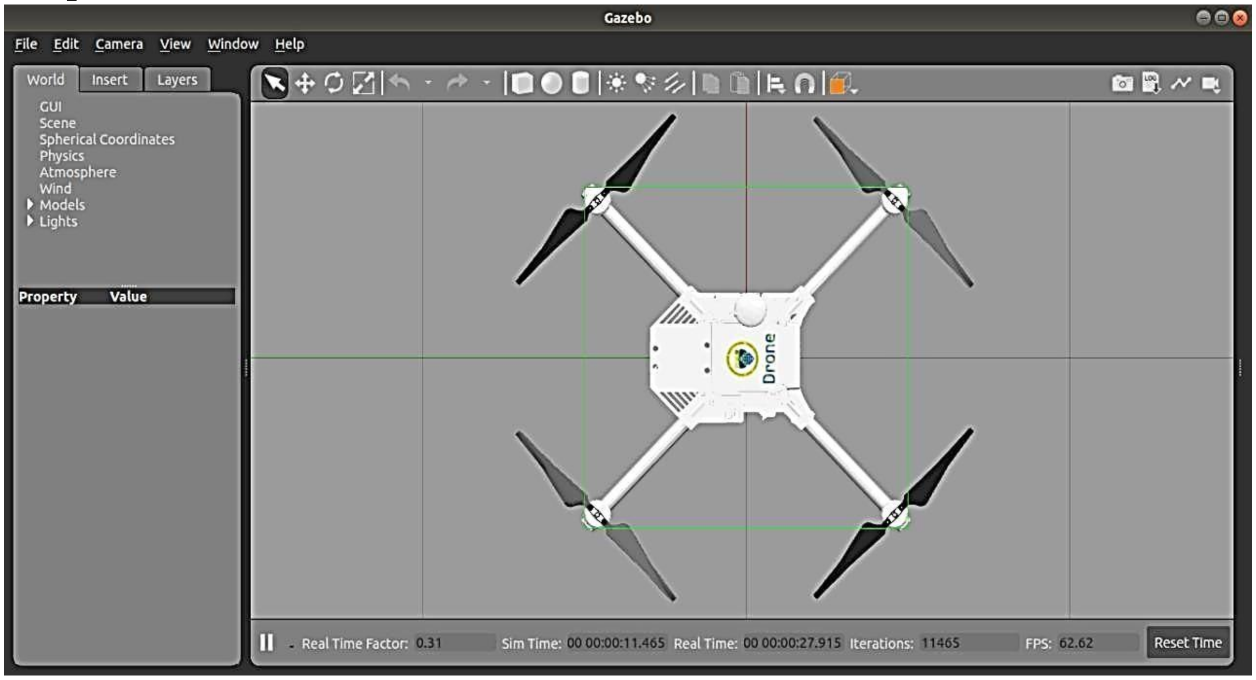
```
# Callback function for /pid_tuning_tool
    def roll_set_pid(self, roll):
        self.Kp[0] = roll.Kp * 0.06  # This is just for an example. You can change the ratio/fraction value accordingly
        self.Ki[0] = roll.Ki * 0.008
        self.Kd[0] = roll.Kd * 0.3

    def pitch_set_pid(self, pitch):
        self.Kp[1] = pitch.Kp * 0.06
        self.Ki[1] = pitch.Ki * 0.008
        self.Kd[1] = pitch.Kd * 0.3

    def yaw_set_pid(self, yaw):
        self.Kp[2] = yaw.Kp * 0.06
        self.Ki[2] = yaw.Ki * 0.008
        self.Kd[2] = yaw.Kd * 0.3

    def alt_set_pid(self, alt):
        self.Kp[3] = alt.Kp * 0.06
        self.Ki[3] = alt.Ki * 0.008
        self.Kd[3] = alt.Kd * 0.3

    def pid_roll(self):
        time_now = time.time()
        time_change = self.sample_time

        error = self.setpoint_euler[1] - self.drone_orientation_euler[0]
        self.roll_error_sum += (error*time_change)
        dErr = (error-self.last_roll_error)/time_change

        self.correct_roll = (self.Kp[0] * error) + (self.Ki[0] * self.roll_error_sum) + (self.Kd[0] *
dErr)
        self.last_roll_error = error
        self.roll_pub.publish(error)

    def pid_pitch(self):
        time_now = time.time()
        time_change = self.sample_time

        error = self.setpoint_euler[0] - self.drone_orientation_euler[1]
        self.pitch_error_sum += (error*time_change)
        dErr = (error-self.last_pitch_error)/time_change
        self.correct_pitch = (self.Kp[1] * error) + (self.Ki[1] * self.pitch_error_sum) + (self.Kd[1] *
dErr)
        self.last_pitch_error = error
        self.pitch_pub.publish(error)

    def pid_yaw(self):
        time_now = time.time()
        time_change = self.sample_time

        error = self.setpoint_euler[2] - self.drone_orientation_euler[2]
        self.yaw_error_sum += (error*time_change)
        dErr = (error-self.last_yaw_error)/time_change

        self.correct_yaw = (self.Kp[2] * error) + (self.Ki[2] * self.yaw_error_sum) + (self.Kd[2] *
dErr)
        self.last_yaw_error = error
        self.yaw_pub.publish(error)

    def pid(self):

        while not rospy.is_shutdown():

            # Converting quaternion to euler angles
            (self.drone_orientation_euler[0], self.drone_orientation_euler[1],
self.drone_orientation_euler[2]) =
tf.transformations.euler_from_quaternion([self.drone_orientation_quaternion[0],
self.drone_orientation_quaternion[1], self.drone_orientation_quaternion[2],
self.drone_orientation_quaternion[3]])

            # Convertng the range from 1000 to 2000 in the range of -10 degree to 10 degree for roll axis
            self.setpoint_euler[0] = self.setpoint_cmd[0] * 0.02 - 30
            self.setpoint_euler[1] = self.setpoint_cmd[1] * 0.02 - 30
            self.setpoint_euler[2] = self.setpoint_cmd[2] * 0.02 - 30

            self.seconds = time.time()
            current_time = self.seconds - self.last_time
            if(current_time >= self.sample_time):
                self.pid_roll()
                self.pid_pitch()
                self.pid_yaw()
                self.last_time = self.seconds

            if (self.setpoint_cmd[3] == 0):
```

```
            throttle = 0
        else:
            throttle = ((((self.setpoint_cmd[3] - 1000) * 1023) / 1000) + 0)

        self.pwm_cmd.prop1 = throttle + self.correct_yaw + self.correct_pitch + self.correct_roll
            self.pwm_cmd.prop2 = throttle - self.correct_yaw - self.correct_pitch +
self.correct_roll
            self.pwm_cmd.prop3 = throttle + self.correct_yaw - self.correct_pitch -
self.correct_roll
            self.pwm_cmd.prop4 = throttle - self.correct_yaw + self.correct_pitch -
self.correct_roll

        if self.pwm_cmd.prop1 > self.max_values:
            self.pwm_cmd.prop1 = self.max_values
        elif self.pwm_cmd.prop1 < self.min_values:
            self.pwm_cmd.prop1 = self.min_values

        if self.pwm_cmd.prop2 > self.max_values:
            self.pwm_cmd.prop2 = self.max_values
        elif self.pwm_cmd.prop2 < self.min_values:
            self.pwm_cmd.prop2 = self.min_values

        if self.pwm_cmd.prop3 > self.max_values:
            self.pwm_cmd.prop3 = self.max_values
        elif self.pwm_cmd.prop3 < self.min_values:
            self.pwm_cmd.prop3 = self.min_values

        if self.pwm_cmd.prop4 > self.max_values:
            self.pwm_cmd.prop4 = self.max_values
        elif self.pwm_cmd.prop4 < self.min_values:
            self.pwm_cmd.prop4 = self.min_values

        self.pwm_pub.publish(self.pwm_cmd)

if__name__== '_main_':

    e_drone = Edrone()
    r = rospy.Rate(e_drone.sample_time) # specify rate in Hz based upon your desired PID
sampling time, i.e. if desired sample time is 33ms specify rate as 30Hz
    while not rospy.is_shutdown():
        e_drone.pid()
        r.sleep()
```

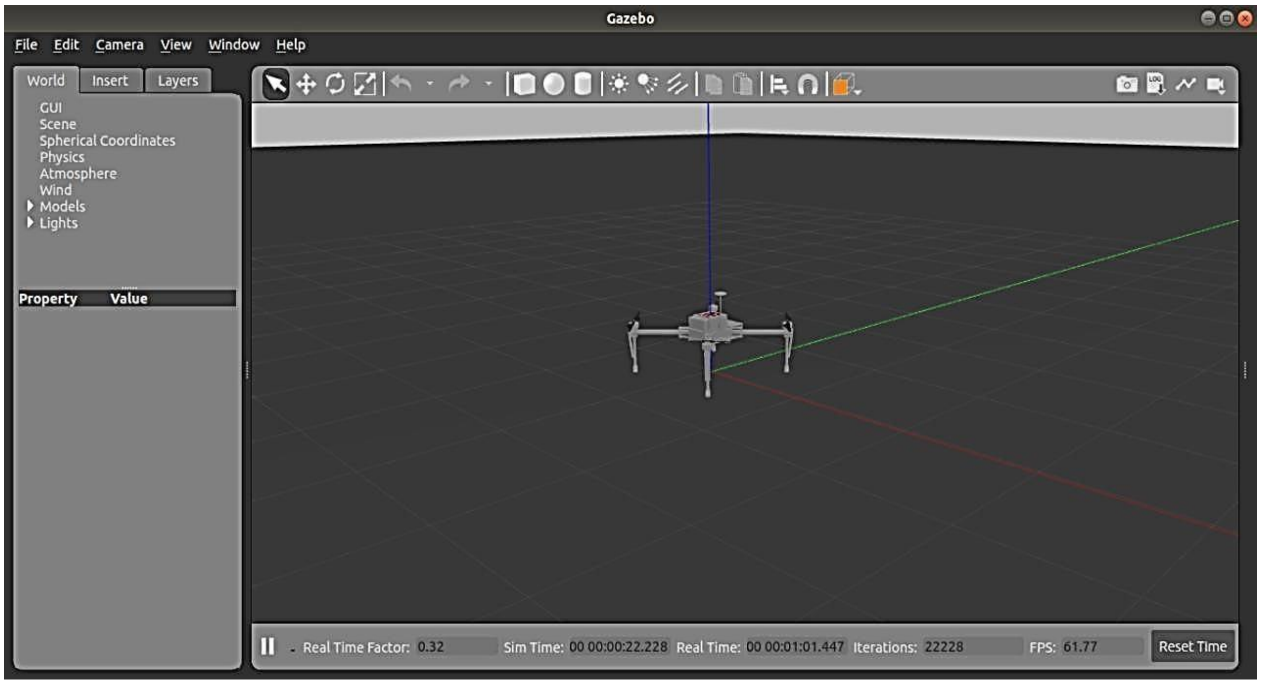
Position controller:

```
#!/usr/bin/env python

# Importing the required libraries
from vitarana_drone.msg import *
from pid_tune.msg import PidTune
from sensor_msgs.msg import NavSatFix
from std_msgs.msg import Float32
import rospy
import time
import tf

class Edrone():
    """docstring for Edrone"""
    def__init_(self):
        rospy.init_node('position_controller')  # initializing ros node with name drone_control
        self.drone_home = [0.0, 0.0, 0.0]
        self.drone_orientation_euler = [0.0, 0.0, 0.0]
        self.setpoint_cmd = [0.0, 0.0, 0.0]
        self.setpoint_destination = [19.0, 72.0, 3.0] #target location

        self.cmd = edrone_cmd()
        self.cmd=[1500,1500,1500,1500]
        self.Kp = [0, 0.0, 699]
        self.Ki = [0, 0.0, 0]
        self.Kd = [0, 0.0, 961]
        self.correct_roll = 0.0
        self.correct_pitch = 0.0
        self.correct_alt = 0.0
        self.alt_error_sum = 0.0
        self.last_yaw_error = 0.0
        self.yaw_error_sum = 0.0
        self.last_pitch_error = 0.0
        self.pitch_error_sum = 0.0
        self.last_roll_error = 0.0
        self.roll_error_sum = 0.0
        self.last_time = 0.0

        self.sample_time = 0.04  # in seconds

        # Publishers
        self.cmd_pub = rospy.Publisher('/drone_command', edrone_cmd, queue_size=1)
        self.roll_pub = rospy.Publisher('/roll_error', Float32, queue_size=1)
        self.pitch_pub = rospy.Publisher('/pitch_error', Float32, queue_size=1)
        self.yaw_pub = rospy.Publisher('/yaw_error', Float32, queue_size=1)
        self.alt_pub = rospy.Publisher('/alt_error', Float32, queue_size=1)

        # Subscribing to /drone_command, imu/data, /pid_tuning_roll, /pid_tuning_pitch,
/pid_tuning_yaw
        rospy.Subscriber('/edrone/gps', NavSatFix, self.gps_callback)
        rospy.Subscriber('/pid_tuning_roll', PidTune, self.roll_set_pid)
        rospy.Subscriber('/pid_tuning_pitch', PidTune, self.pitch_set_pid)
        rospy.Subscriber('/pid_tuning_yaw', PidTune, self.yaw_set_pid)
        rospy.Subscriber('/pid_tuning_altitude', PidTune, self.alt_set_pid)

    def gps_callback(self, gps):
        self.drone_home[0] = gps.latitude
        self.drone_home[1] = gps.longitude
        self.drone_home[2] = gps.altitude

    def roll_set_pid(self, roll):
        self.Kp[0] = pitch.Kp * 0.06
        self.Ki[0] = pitch.Ki * 0.008
        self.Kd[0] = pitch.Kd * 0.3
    def pitch_set_pid(self, pitch):
        self.Kp[1] = pitch.Kp * 0.06
        self.Ki[1] = pitch.Ki * 0.008
        self.Kd[1] = pitch.Kd * 0.3
    def alt_set_pid(self, alt):
        self.Kp[2] = 699 * 0.06
        self.Ki[2] = 0 * 0.008
        self.Kd[2] = 961 * 0.3

    def pid_roll(self):
        time_now = time.time()
        time_change = self.sample_time

        error = self.setpoint_destination[0] - self.drone_home[0]
        self.roll_error_sum += (error*time_change)
        dErr = (error-self.last_roll_error)/time_change

        self.correct_roll = (self.Kp[0] * error) + (self.Ki[0] * self.roll_error_sum) + (self.Kd[0] *
dErr)
        self.last_roll_error = error
        self.roll_pub.publish(error)
    def pid_pitch(self):
        time_now = time.time()
        time_change = self.sample_time

        error = self.setpoint_destination[1] - self.drone_home[1]
        self.pitch_error_sum += (error*time_change)
        dErr = (error-self.last_pitch_error)/time_change

        self.correct_pitch = (self.Kp[1] * error) + (self.Ki[1] * self.pitch_error_sum) + (self.Kd[1] *
dErr)
        self.last_pitch_error = error
        self.pitch_pub.publish(error)

    def pid_alt(self):
        time_now = time.time()
        time_change = self.sample_time

        error = self.setpoint_destination[2] - self.drone_home[2]
        self.alt_error_sum += (error*time_change)
        dErr = (error-self.last_alt_error)/time_change

        self.correct_alt = (self.Kp[2] * error) + (self.Ki[2] * self.alt_error_sum) + (self.Kd[2] *
dErr)
        self.last_alt_error = error
        self.alt_pub.publish(error)

    def arm(self):
        self.cmd.aux4 = 1500
        self.cmd.rcThrottle = 1100
        self.cmd_pub.publish(self.cmd)
        rospy.sleep(.1)

    def disarm(self):
        self.cmd.aux4 = 1100
        self.cmd_pub.publish(self.cmd)
        rospy.sleep(.1)

    def pid(self):
        print "DISARMED"
        self.disarm()
        rospy.sleep(.2)

        print "ARMED"
        self.arm()
        rospy.sleep(.1)

        while not rospy.is_shutdown():

            self.seconds = time.time()
            current_time = self.seconds - self.last_time
            if(current_time >= self.sample_time):
                self.pid_roll()
                self.pid_pitch()
                self.pid_alt()
                self.last_time = self.seconds

            self.cmd.rcRoll = 1500 + self.correct_roll
            self.cmd.rcThrottle = 1500 + self.correct_alt
            self.cmd.rcPitch = 1500 + self.correct_pitch

            if self.cmd.rcRoll > 1800:
                self.cmd.rcRoll = 1800
            elif self.cmd.rcRoll < 1000:
                self.cmd.rcRoll = 1000

            if self.cmd.rcPitch > 1800:
                self.cmd.rcPitch = 1800
            elif self.cmd.rcPitch < 1000:
                self.cmd.rcPitch = 1000

            if self.cmd.rcThrottle > 1800:
                self.cmd.rcThrottle = 1800
            elif self.cmd.rcThrottle < 1000:
                self.cmd.rcThrottle = 1000

            self.cmd.rcYaw = 1500
            self.cmd_pub.publish(self.cmd)

if__name__== '_main_':
    e_drone = Edrone()
    r = rospy.Rate(e_drone.sample_time)  # specify rate in Hz based upon your desired PID
sampling time, i.e. if desired sample time is 33ms specify rate as 30Hz
    while not rospy.is_shutdown():
        e_drone.pid()
        r.sleep()
```

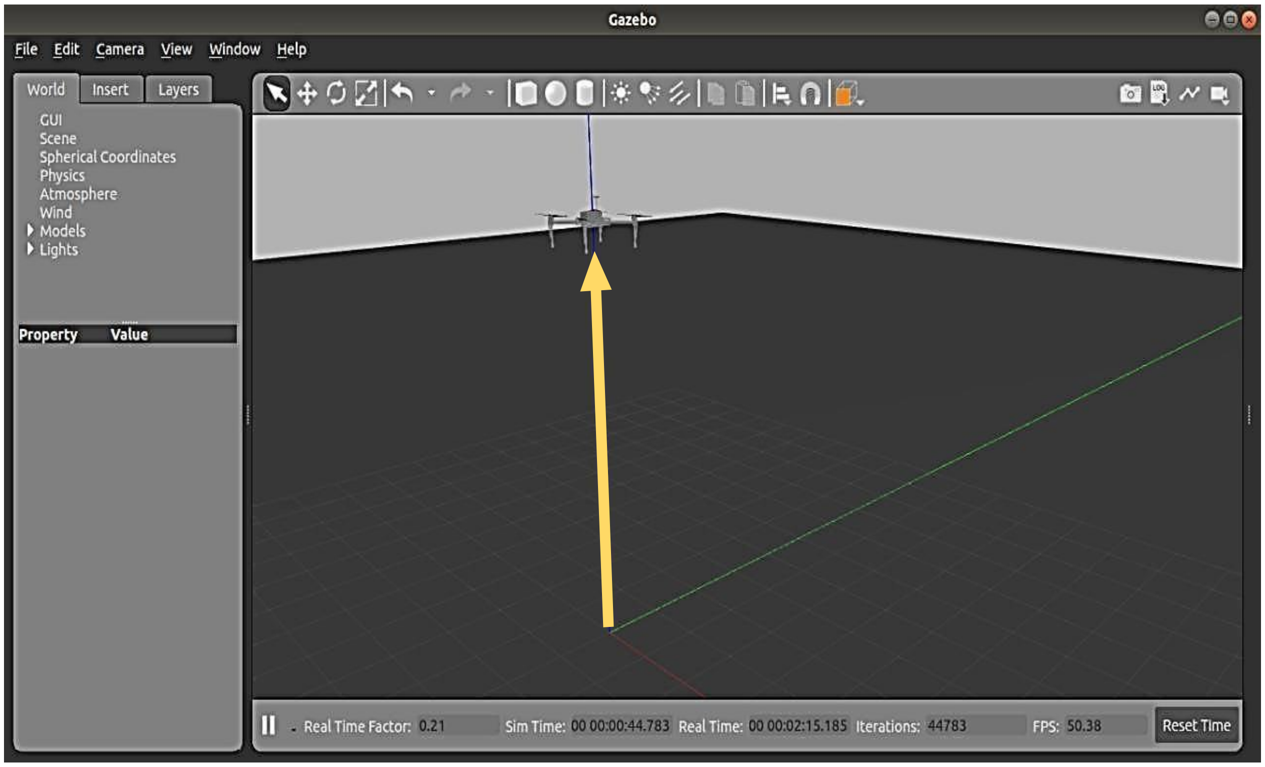
5. Output:



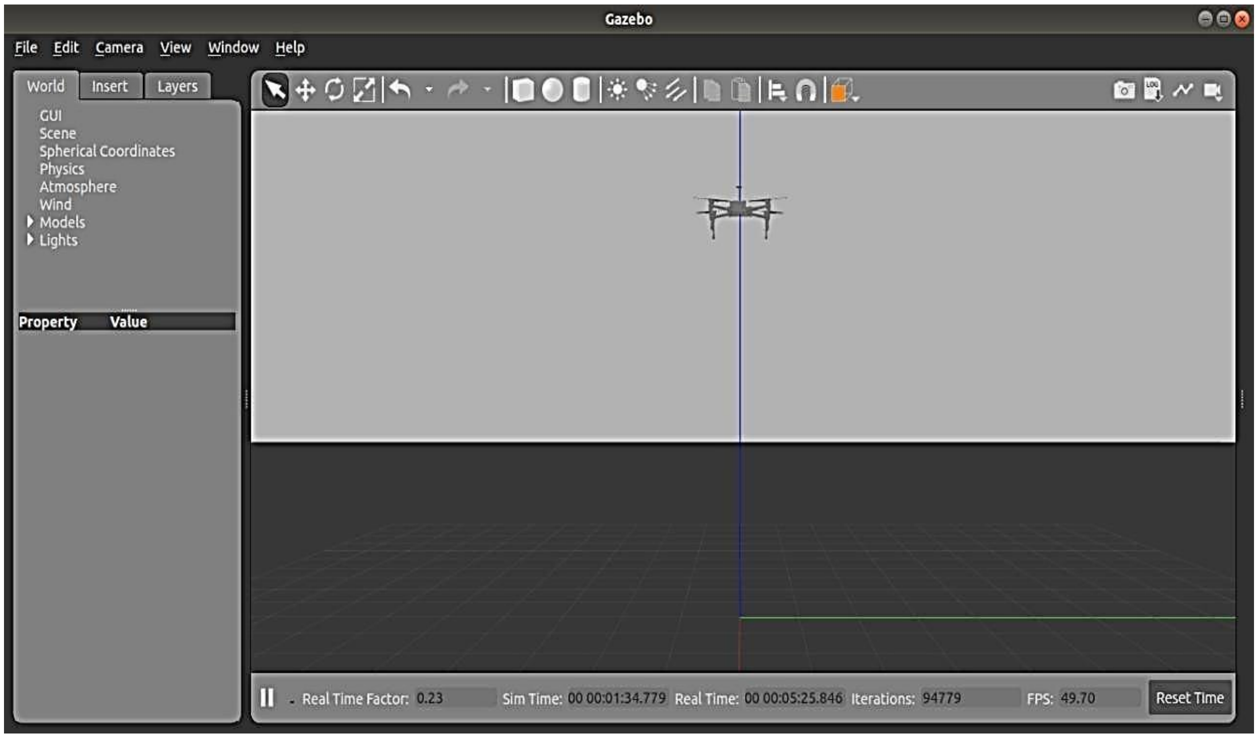
Real Time Factor: 0.31 Sim Time: 00 00:00:11.465 Real Time: 00 00:00:27.915 Iterations: 11465 FPS: 62.62



Real Time Factor: 0.32 Sim Time: 00 00:00:22.228 Real Time: 00 00:01:01.447 Iterations: 22228 FPS: 61.77



Real Time Factor: 0.21 Sim Time: 00 00:00:44.783 Real Time: 00 00:02:15.185 Iterations: 44783 FPS: 50.38



Real Time Factor: 0.23 Sim Time: 00 00:01:34.779 Real Time: 00 00:05:25.846 Iterations: 94779 FPS: 49.70

6. Advantages:

- ❖ Safe Environment
- ❖ Cost saving Technology
- ❖ Quality of aerial imaging
- ❖ Precision and security
- ❖ Easy controllable or deployable
- ❖ Can reach Hazardous condition

7. Application:

- ❖ Aerial photography
- ❖ Shipping and delivery
- ❖ Geographic mapping
- ❖ Disaster management
- ❖ Precision agriculture
- ❖ Search and rescue
- ❖ Weather forecast

8. Reference:

- ❖ https://portal.e-yantra.org/storage/FjbIfxILQH_vd/res/learn/linux/learn-linux.html
- ❖ http://wiki.ros.org/ROS/Tutorials
- ❖ https://portal.eyantra.org/storage/FjbIfxILQH_vd/res/learn/control_systems/understanding_pid.html
- ❖ https://portal.e-yantra.org/storage/FjbIfxILQH_vd/res/learn/drone/eyrc-VD-Exploring_eDrone_ROSGazebo_model.html