eYSIP2018

Autotuning Of Controller For Drones

Interns:
Mahadev Mishal
Karthik Nayak
Amit Kumar

Mentors:
Fayyaz Pocker
Vamshi Krishna
Simranjeet Singh

Duration of Internship: 21/05/2018 – 06/07/2018

2018, e-Yantra Publication

Contents

Autotuning Of Controller For Drones

1.1 Abstract

Proportional-integral-derivative (PID) controllers are widely used in industrial systems despite the significant developments of recent years in control theory and technology. They perform well for a wide class of processes. Also, they give robust performance for a wide range of operating conditions. Furthermore, they are easy to implement using analogue or digital hardware. In practical implementation of a controller and tuning its control parameter, there is a high possibility that due to human intervention the process is not tuned to obtain optimum control. Also this method of hit and trial is not an efficient way. Hence we propose two methods of auto-tuning the PID and estimating the values of PID parameter for Pluto drone. The first method is based on Ziegler-Nichols approach and second method is iteration based Auto-Tuning.

1.2 Completion status

- Implemented PID on AR-Drone model in Gazebo.
- Implemented position holding of Pluto drone using whycon marker by manual tuning
- Implemented position holding of Pluto drone using whycon marker by applying Ziegler-Nichols
- Implemented autotune on the AR-Drone model and tested it in Gazebo.
- Implemented autotune on Pluto drone using two different techniques.

1.3 Hardware parts

- **Pluto drone by Drona Aviation Pvt. Ltd.**

  - STM32F103C8 Micro-controller
    Datasheet — Reference Manual
  - MPU9250 Accelerometer and Gyroscope
    Datasheet
  - AK8963 Magnetometer
    Datasheet
  - MS5611 Barometer
    Datasheet
  - ESP8266 Wi-Fi Wireless Module
  - Coreless Motors and Propellers
  - 3.7V Li-Po Battery

- **Wide angle camera**

  - **ROS compatible**
  - **Resolution**: 640 x 480
  - 30 frames per second

- **Whycon marker**
  WhyCon is a vision-based localization system that can be used with low cost web cameras. It achieves millimeter precision with high performance. These markers consist of a dark Outer ring and a concentric white circle as shown in figure 1.3.1

figure 1.3.1: Whycon marker

You can find more information about Whycon marker here

1.4 Drone Motion

- **Introduction**
  A quadcopter, also called a quadrotor helicopter or quadrotor, is a multirotor helicopter that is lifted and propelled by four rotors. Quadcopters are classified as rotorcraft, as opposed to fixed-wing aircraft, because their lift is generated by a set of rotors (vertically oriented propellers).Quadcopters generally use two pairs of identical fixed pitched propellers; two clockwise (CW) and two counterclockwise (CCW). These use independent variation of the speed of each rotor to achieve control.In drones, each motor spins in the opposite direction of the adjacent motor as shown in figure 1.4.1. This allows it to achieve vertical lift.

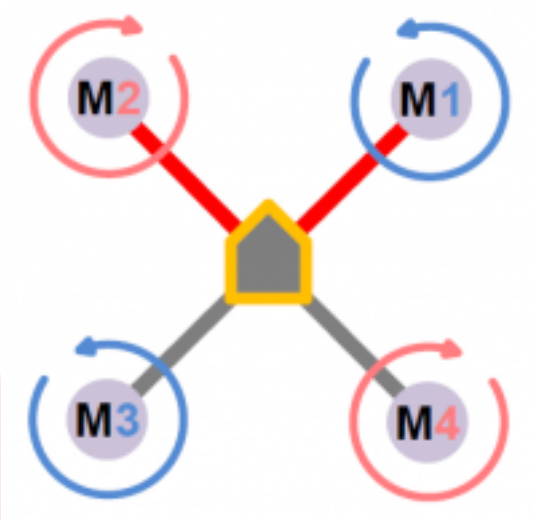


figure 1.4.1

- **Drone motion.**
  A drones thrust, roll, pitch and yaw is changed by manipulating the angular velocity of the motors. Following details how to move the drone by manipulating the angular velocity of the motors:

  I **Thrust/Throttle** - In order to change the drones height, we can decrease or increase the velocity of all 4 motors. Reducing the velocity of all the motors lowers the drone height, while increasing the velocity increases the height of the drone with respect to the ground.

  II **Pitch** - By changing the motor speed of the front and back motors, we can move the drone forward and backward. Increasing the speed of the forward motors moves the drone back while increasing the speed of the back motors moves the drone backward.

  III **Roll** - To move the drone left or right, we simply manipulate the left and right motors. By increasing the speed of the two left motors the drone bends to the right. By increasing the speed of the two right motors the drone bends to the left.

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

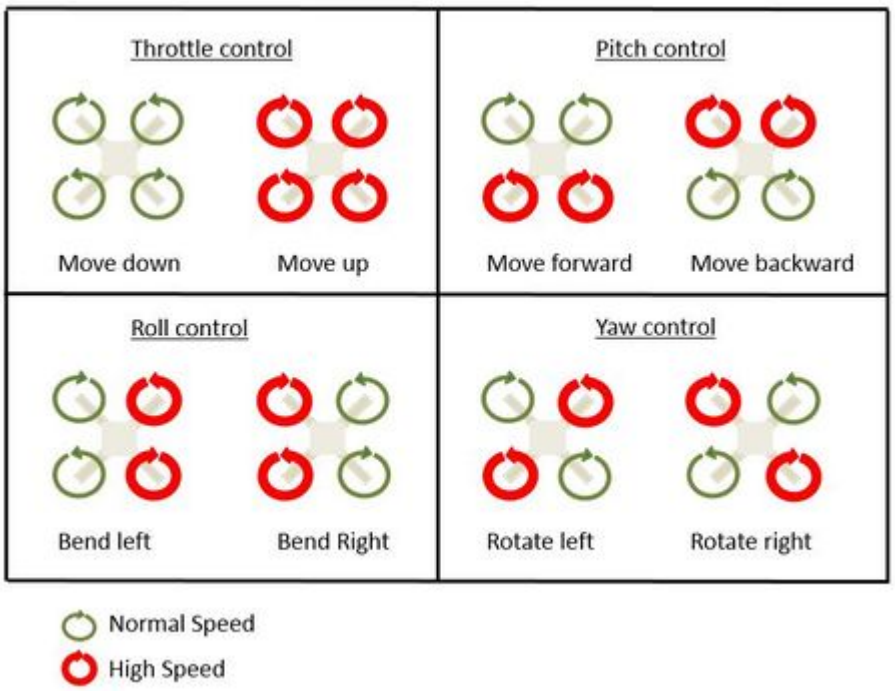


figure 1.4.2: Drone motion in 3D space

1.5 Software used

- **ROS *indigo***
  Robot Operating System (ROS) is a framework which provides tools and libraries to help software developers to create robot applications. The primary goal of ROS is to support code reuse in robotics research and development. Testing of robot code can be time-consuming and error-prone and sometimes physical robot might not be present. ROS provides a solution to this problem as it separates the hardware part and decision making (coding) part. Because of this separation, we can replace hardware part with a model in the simulator and test the behavior of decision-making part. It is open-source software. It also provides a simple way to record and play data.

- **Gazebo**
  Gazebo is a real world physics simulator. In this simulator, you can set up a world and simulate your robot moving around in this world. By installing "**Desktop-Full**" version of ROS *indigo* the simulator like **Gazebo, Rviz** also get installed.

- **Download link**
  - Ubuntu 14.04
    (64Bit)/ (32 Bit)
  - ROS *indigo*
    download link

1.6 Arena Setup

The figure 1.6.1 depicts the arena in which the Pluto drone will be made to fly.

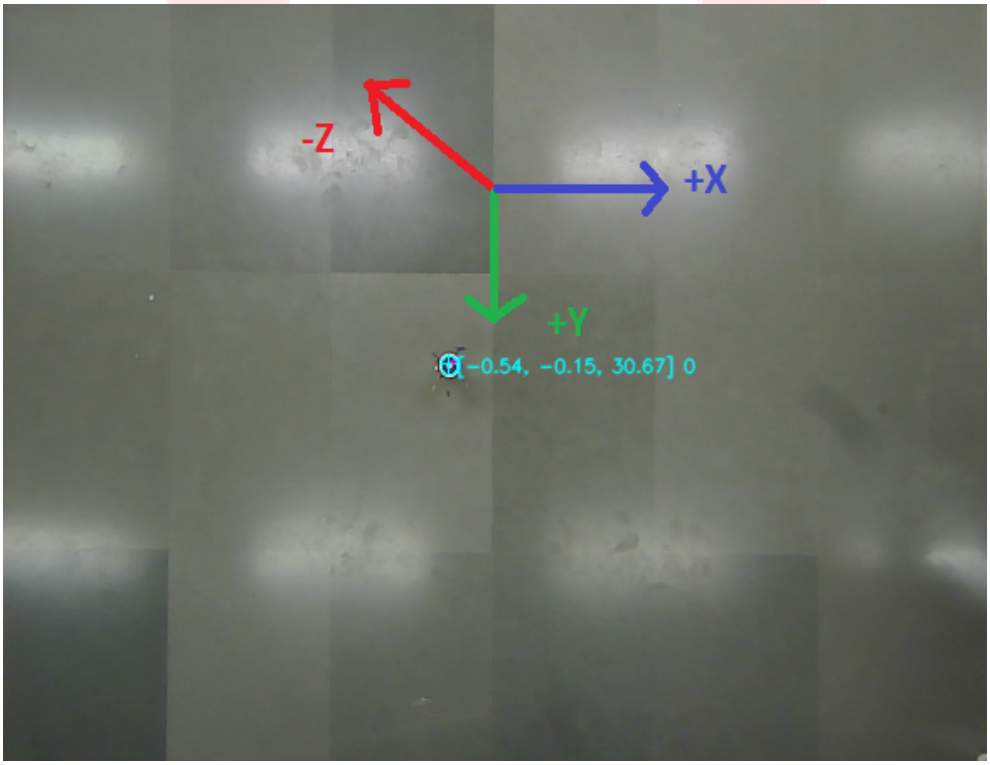


figure 1.6.1

The x, y and z axes are as shown. As the z axis is w.r.t the overhead camera the +z axis is downwards.
Pitch axis of the drone is along the x-axis , roll axis is along y-axis and throttle is along z-axis. To move in +x axis the pitch must be increased,

the same must be decreased to move in -x axis. To move in +y axis the roll must be increased, to move in -y axis the same must be decreased To move in +z axis the throttle must be decreased , to move it in -z axis it must be decreased.

1.7 Software and Code

1.7.1 Types of controller -

- **Proportional Controller (P Controller)**
  P controller is mostly used in first order processes with single energy storage to stabilize the unstable process. The main usage of the P controller is to decrease the steady state error of the system. As the proportional gain factor K increases, the steady state error of the system decreases. However, despite the reduction, P control can never manage to eliminate the steady state error of the system. As we increase the proportional gain, it provides smaller amplitude and phase margin, faster dynamics satisfying wider frequency band and larger sensitivity to the noise. We can use this controller only when our system is tolerable to a constant steady state error. In addition, it can be easily concluded that applying P controller decreases the rise time and after a certain value of reduction on the steady state error, increasing K only leads to overshoot of the system response. P control also causes oscillation if sufficiently aggressive in the presence of lags and/or dead time. The more lags (higher order), the more problem it leads. Plus, it directly amplifies process noise.

- **Proportional Derivative Controller (PD Controller)**
  The aim of using P-D controller is to increase the stability of the system by improving control since it has an ability to predict the future error of the system response. In order to avoid effects of the sudden change in the value of the error signal, the derivative is taken from the output response of the system variable instead of the error signal. Therefore, D mode is designed to be proportional to the change of the output variable to prevent the sudden changes occurring in the control output resulting from sudden changes in the error signal. In addition D directly amplifies process noise therefore D-only control is not used.

- **Proportional Integral Derivative (PID Controller)**
  P-I-D controller has the optimum control dynamics including zero steady state error, fast response (short rise time), no oscillations and higher

stability. The necessity of using a derivative gain component in addition to the PI controller is to eliminate the overshoot and the oscillations occurring in the output response of the system.

1.7.2 P Controller

A P controller consists of only a linear gain Kp. The output of such controller can be simply given as

output = Kp * error

Block diagram

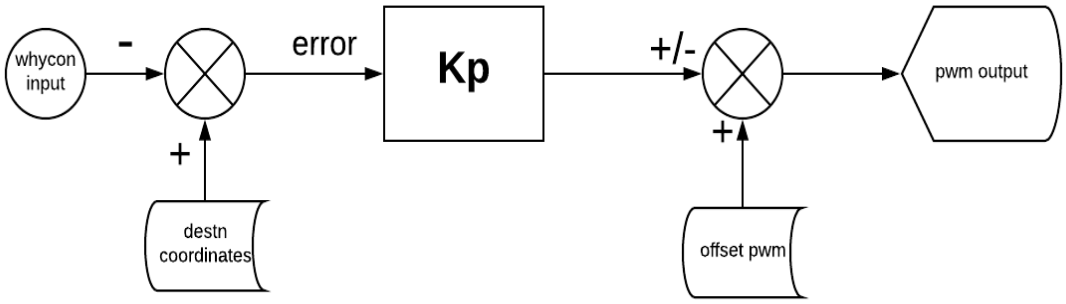


figure 1.7.2.1: Block diagram of a P-Controller

The whycon input consists of x, y and z coordinates which gives the current location of the drone.
Suppose the destination is say, (x1, y1, z1), then the difference of coordinates i.e (x1-x) ,(y1-y), (z1-z) will be fed as an input to the P-controller. The resultant product i.e Kp * error will be added or subtracted to the offset pwm as the need be to give the final output.

Click here to access the python program that implemented a P Controller on Pluto drone

Observations

It was observed that the drone never settled at its destination. Instead it oscillated about its destination. The higher the value of Kp the more the amplitude of the oscillation and the drone would be out of the flying zone.

Below is a plot(figure 1.7.2.2) between error in x, y, and z coordinates and time. Clearly it is visible that the drone was not able to stabilise itself at the destination.

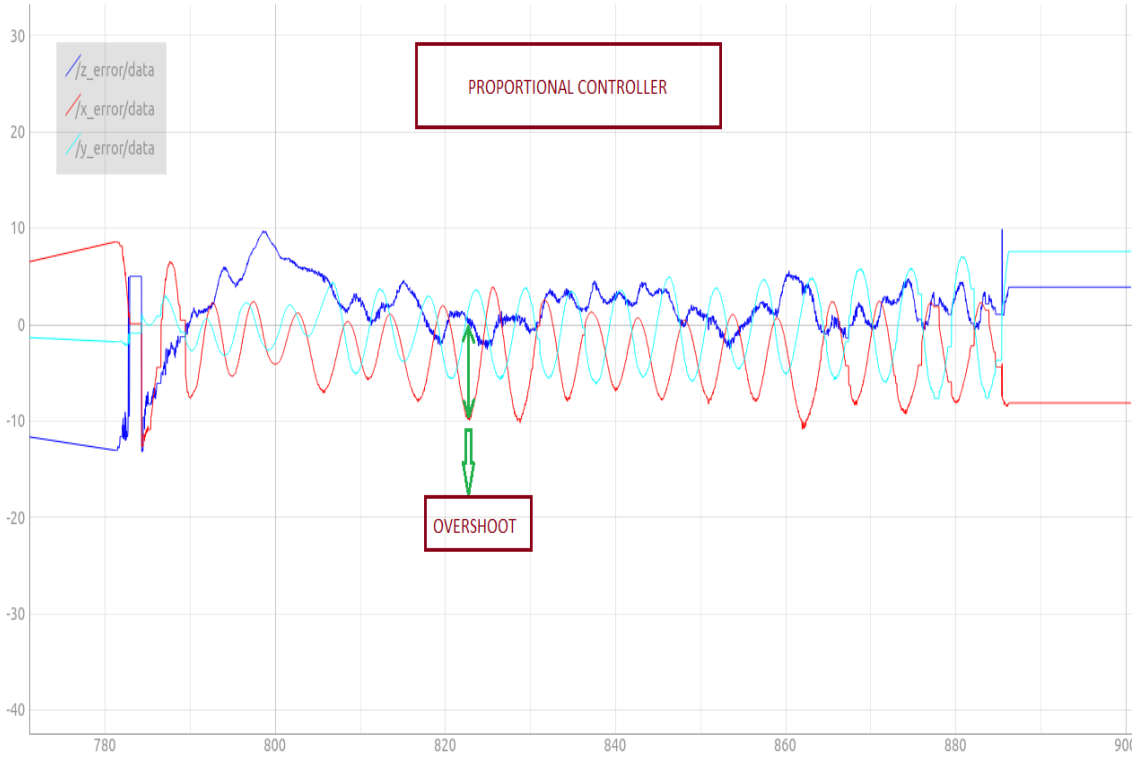


figure 1.7.2.2: Plot of error vs time

Hence, it was concluded that a P-controller by itself wasn't able to stabilise the drone at a required point.

1.7.3 PD Controller

In the previous section we saw how the P-Controller wasn't successful in stabilising the drone at a given point. It was observed that there were oscillations instead. These oscillations can be damped by using a differential gain along with the P-Controller. The system as a whole is said to be a PD Controller

Block diagram

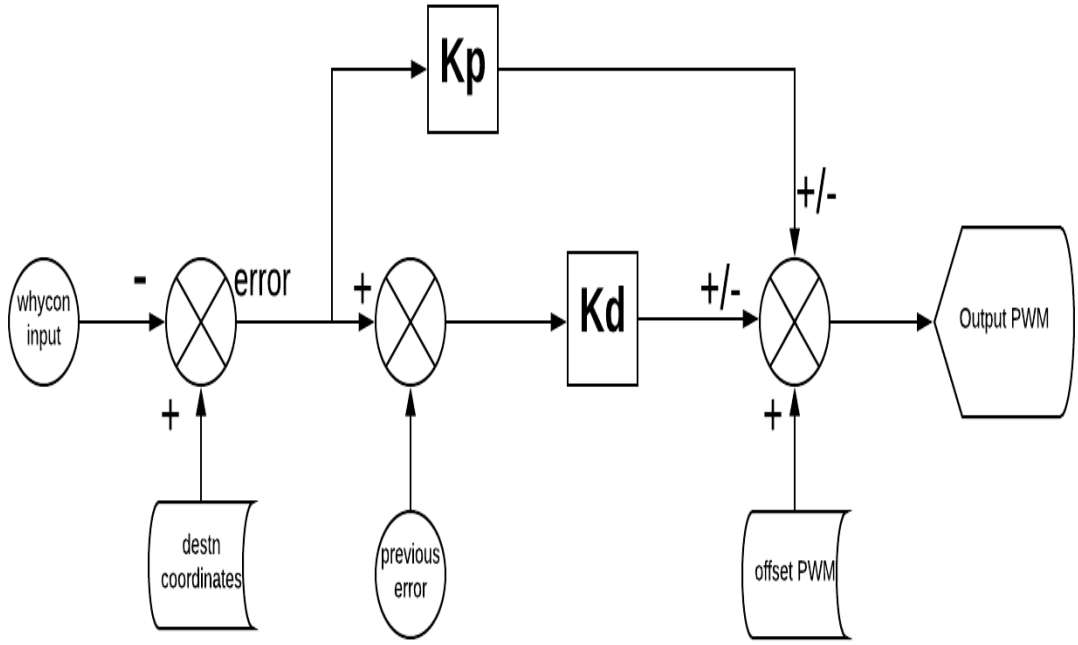


figure 1.7.3.1: Block diagram of a PD-controller

The differential gain Kd is multiplied with the difference of error and previous error. Previous error is a variable that holds the last error generated by the controller output.
The controller output in this case is given as

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

Note that Kd 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.

Click here to access the python program that implemented a PD Controller on Pluto drone

Observations

The results of this controller was no match to the P-controller.
The oscillations were damped with change in time. Here is a plot(figure

1.7.3.2) of error and time for a PD controller implemented on the Pluto drone.

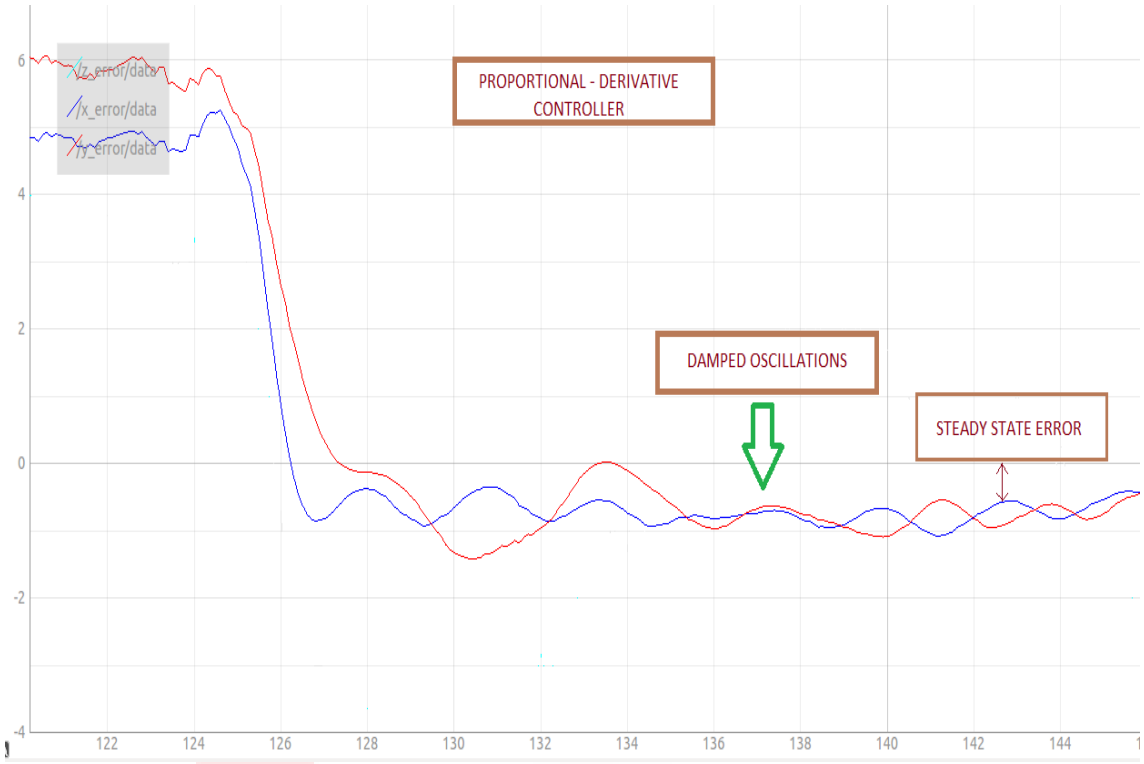


figure 1.7.3.2: Plot of error vs time

But, there was a hitch!
On having a closer look it was observed that though the drone could hover with respectable stability, it did not do so over the correct point, i.e the drone did not reach its destination instead it would hover at a point near to the destination.
This slight error is known as the steady state error.
Hence, it was concluded that a PD controller also by itself wasn't able to stabilise the drone at the correct destination.

1.7.4 PID Controller

In the previous section we saw how a PD controller was not quite enough.
In order to minimise the steady state error we introduce another gain called Ki ,the integral gain.
Such a system is said to be a PID Controller

Block diagram

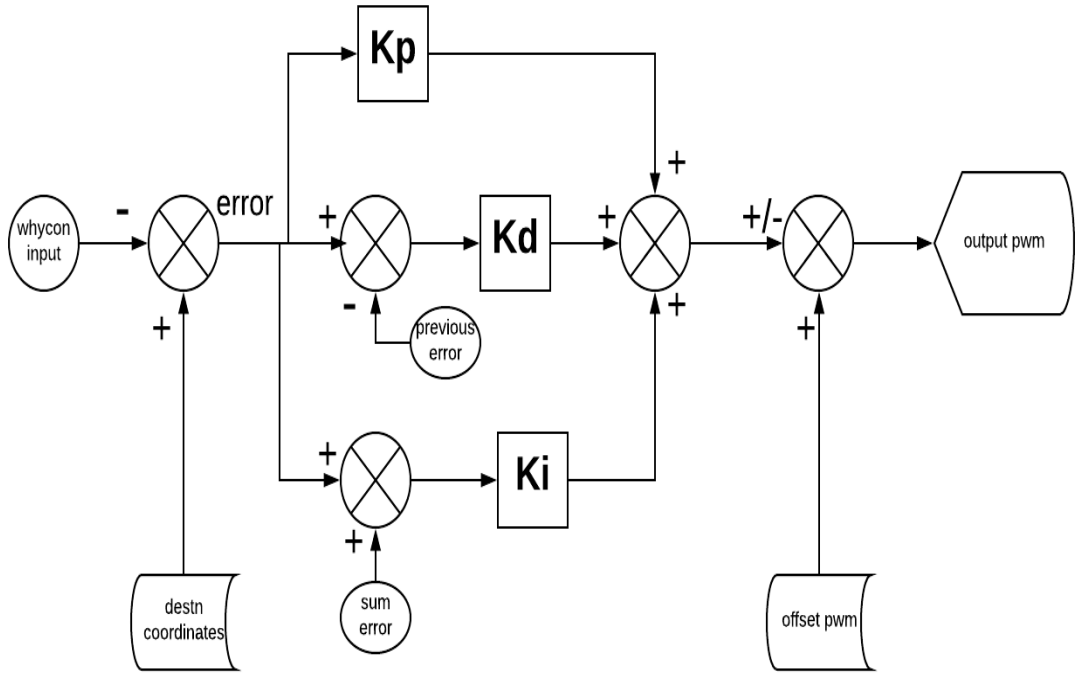


figure 1.7.4.1: Block diagram of a PID-Controller

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

$$\textbf{Iterm} = \textbf{(Iterm + error) * Ki}$$

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.

$$\textbf{output} = \textbf{Kp*error + Iterm + Kd*(error - previous error)}$$

Click here to access the python program that implemented a PID Controller on Pluto drone

Observations

The PID controller was successful in hovering the drone above the destination point with minimal error .
It overcome the steady state error which was noticeable in previous controllers. Here is a plot of error and time for a PID controller implemented on the Pluto drone.

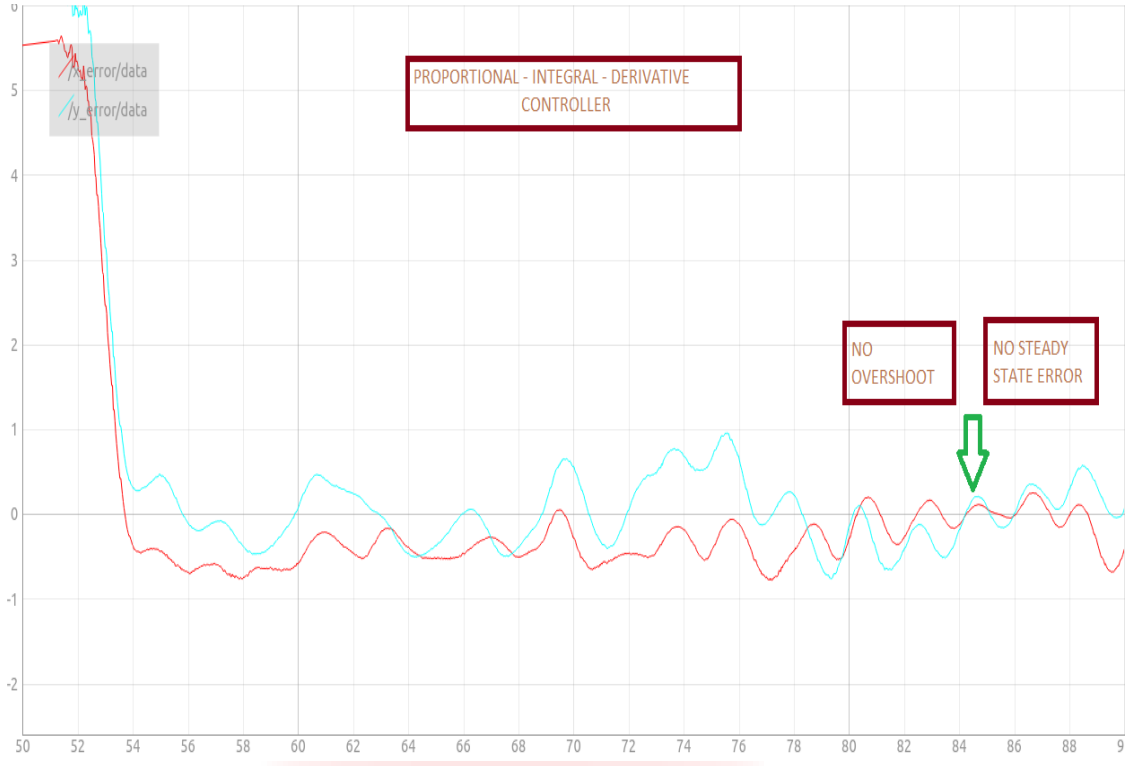


figure 1.7.4.2: Plot of error vs time

To sum it up

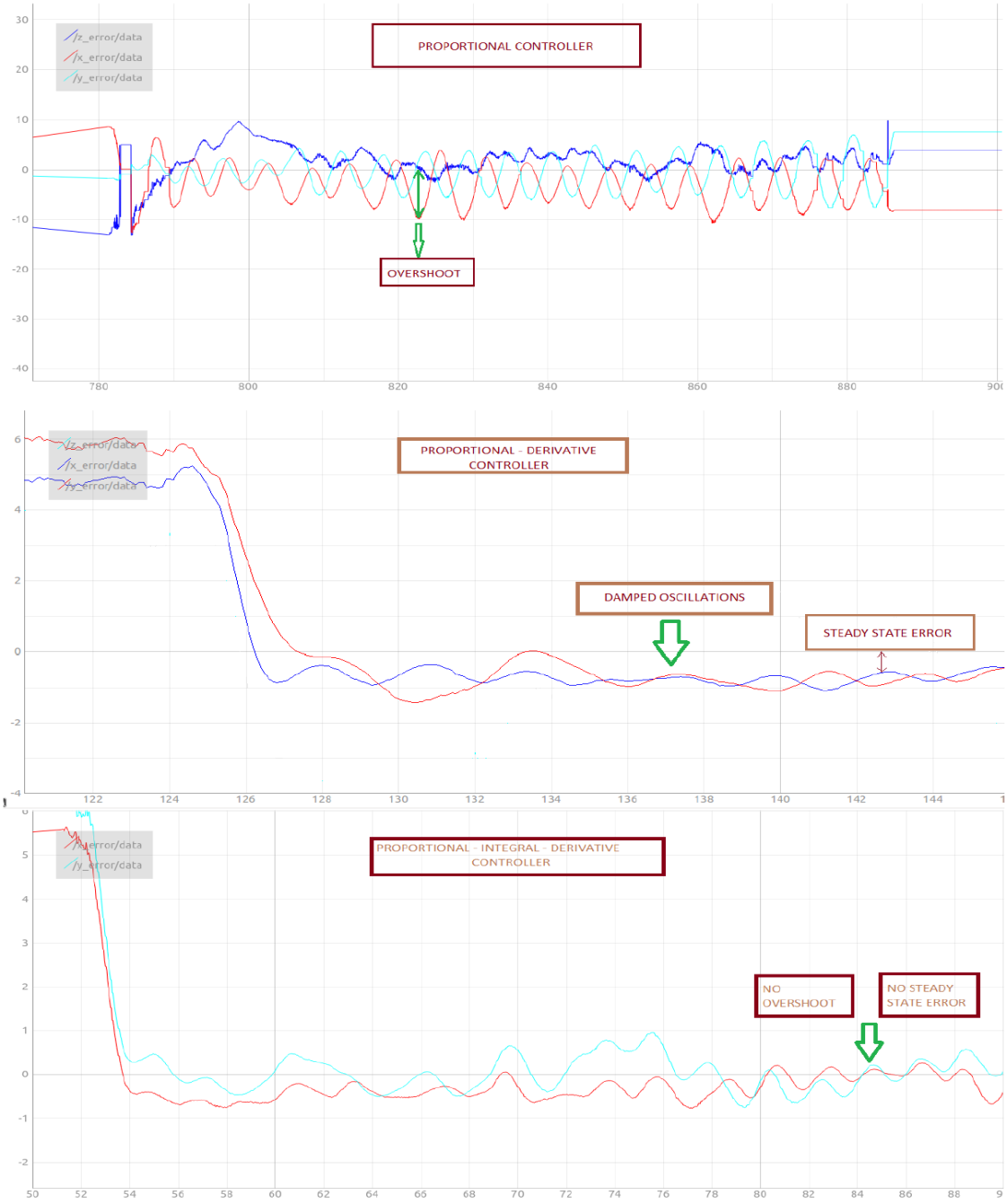


figure 1.7.1 Plots of error vs time for a P, PD and PID controller

1.7.5 Manual Tuning - A tedious job!

The constants Kp, Ki and Kd need to be tuned to get the best response, however this job of tuning them manually is tedious.
Moreover if trial and error method is used to tune it , it may take longer time and may also compromise on efficiency.

The Solution

In order to reduce the human effort and time spent in tuning these parameters manually it is worth finding solutions that could somehow auto-tune these parameters.
In this project we propose two such methods.

1.7.6 Auto-tuning of Controllers

In this section we present two methods of auto-tuning.

1. **Auto-tuning based on Ziegler-Nichols approach**
2. **Iteration based auto-tuning**

1. Auto-tuning based on Ziegler-Nichols approach

In this method of auto-tuning we try to analyse the nature of what the controller is driving, then reverse-engineer to calculate tuning parameters from the output.
We do this by changing the PID Output and then observe how the Input responds.

Algorithm

1. Force the PID output to maximum.
2. Wait for the input(hereon the drone's current location will be referenced as input), to cross above the setpoint(destination).
3. Once the input is above the setpoint, force the PID output to minimum.
4. Wait for the input to cross below the setpoint and then again force the PID output to maximum again.

5. This constitutes an oscillation about the setpoint. Keep track of the time period of oscillations
6. Determine Ultimate Gain ,Ku as

$$\mathbf{Ku = (4*d)/(pi*A)}$$

   where,
   d –> amplitude of PID output
   a –> amplitude of oscillation about the setpoint
7. The period of oscillation is known as Ultimate period **Tu**
8. Use Ziegler-Nichols method to determine the values of Kp, Ki, Kd.

More about Ziegler-Nichols approach in next section.!

Ziegler - Nichols method of tuning PID constants

The ZieglerNichols tuning method is one the methods to tune a PID controller. It was developed by John G. Ziegler and Nathaniel B. Nichols. Initially the integral gain, Ki and derivative gain, Kd, are set to zero. The proportional gain,Kp is then increased (from zero) until it reaches the ultimate gain Ku, at which the output of the control loop has stable and consistent oscillations.
This Ultimate gain Ku and Ultimate period Tu are eventually used to determine the Kp, Ki and Kd values.
The following table establishes the relation between the above mentioned parameters.

Controller type	**Kp**	**Ti**	**Td**
P	0.5 * Ku	–	–
PI	0.5 * Ku	Tu/1.25	–
PD	0.8 * Ku	–	Tu/8
Classic PID	0.6 * Ku	Tu/2	Tu/8
Pessen Integral rule	0.7 * Ku	Tu/2.5	3Tu/20
Some overshoot	0.33 * Ku	Tu/2	Tu/3
No overshoot	0.2 * Ku	Tu/2	Tu/3

Depending upon the need an appropriate controller type is selected and their corresponding equations can be used to compute the required parameters.
From the values of Ti, Td, and Kp it is possible to deduce the values of Ki and Kd as,

$$\mathbf{Ki = Kp/Ti}$$
$$\mathbf{Kd = Kp * Td}$$

Once Kp, Ki, Kd are determined they can be just entered into the previously designed PID architecture.

Simulation on AR-Drone in Gazebo

The auto-tuning logic was applied to determine the Kp, Ki and Kd values of the PID controller that controlled the motion of the ARDrone.

Click here to access the python program that simulated an auto-tuned PID Controller on AR drone in Gazebo

Observations

The auto-tuned values of Kp, Kd and Ki were properly tuned and the PID could bring about stable way point navigation of the drone.

Here are a couple of plots that were obtained during the simulation

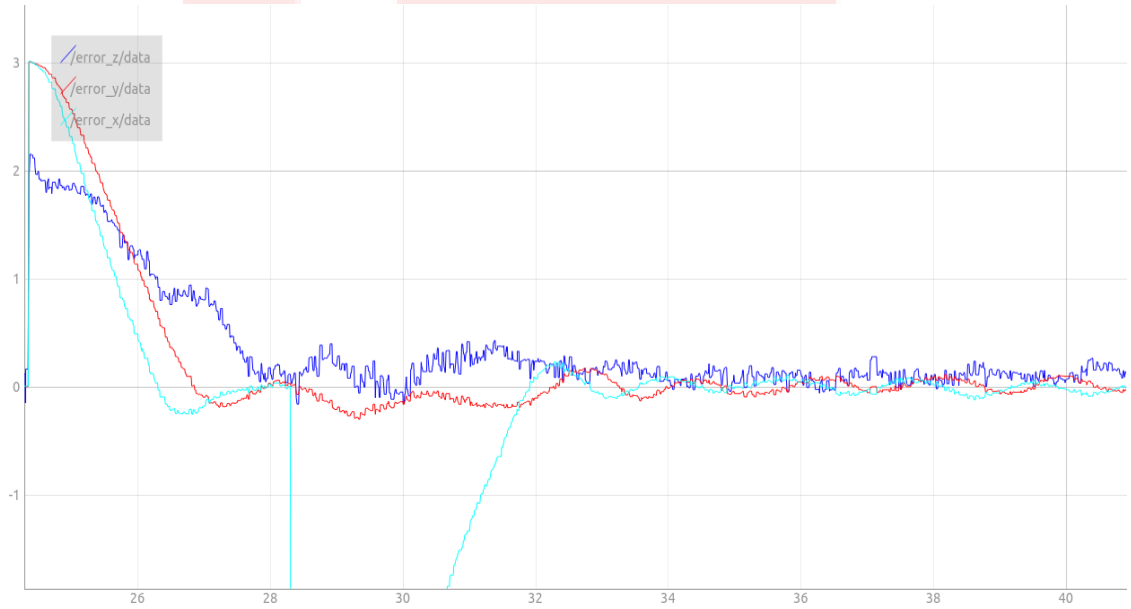


figure 1.7.6.1.1: Plot of error vs time

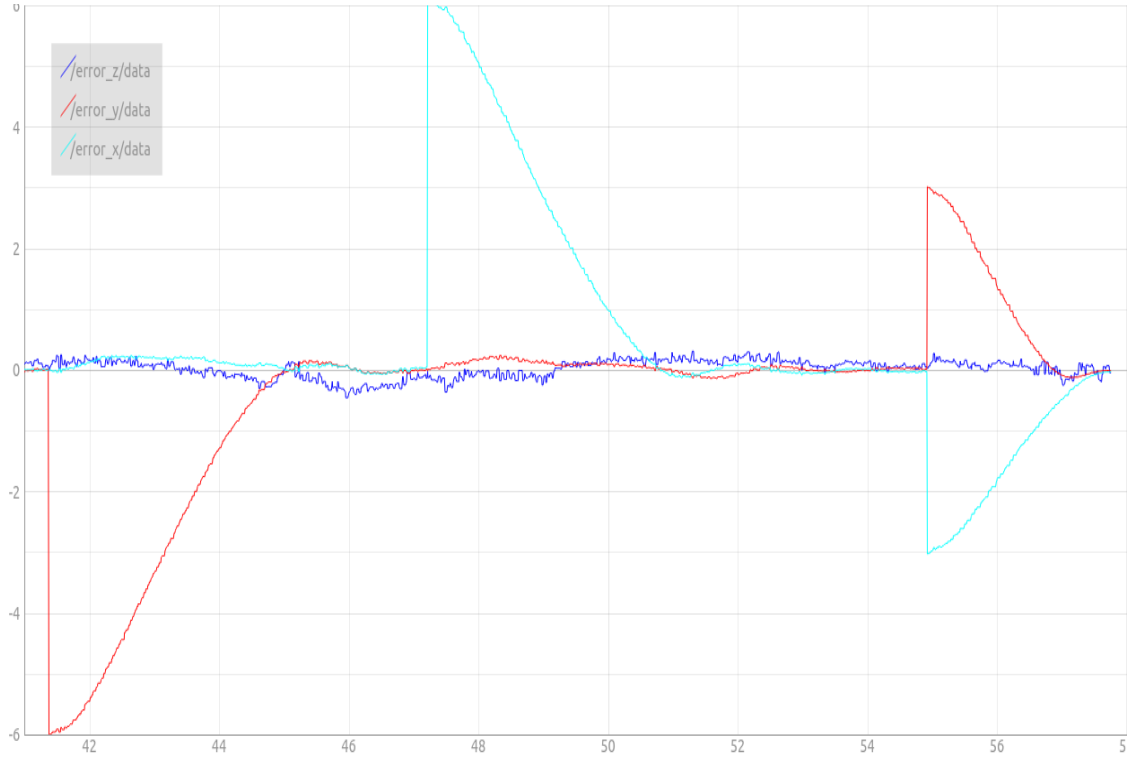


figure 1.7.6.1.2: Plot of error vs time

After successful simulation of auto-tuning logic on AR-Drone in Gazebo, the same logic was applied for a Pluto drone.
The next section deals with the implementation of auto-tuning logic on Pluto Drone.

Implementation on the Pluto drone

The concept of auto-tuning based on Ziegler-Nichols was tested on the Pluto drone.
The drone was forced to oscillate on the z, x and y axis in order to determine the Ultimate gain, Ku and Ultimate period, Tu.

Click here to access the python program that implemented an auto-tuning of PID Controller of Pluto Drone
The auto-tuning was performed on the drone for various types of PID as given in the previous table.

Observations

Auto-tuning performed by taking constant expressions from the previous table for classic PID, Pessen Integral rule, Some overshoot PID and No over-

shoot PID yielded good results.
Below are the plots of error vs time for each of the above mentioned cases.

Classic PID

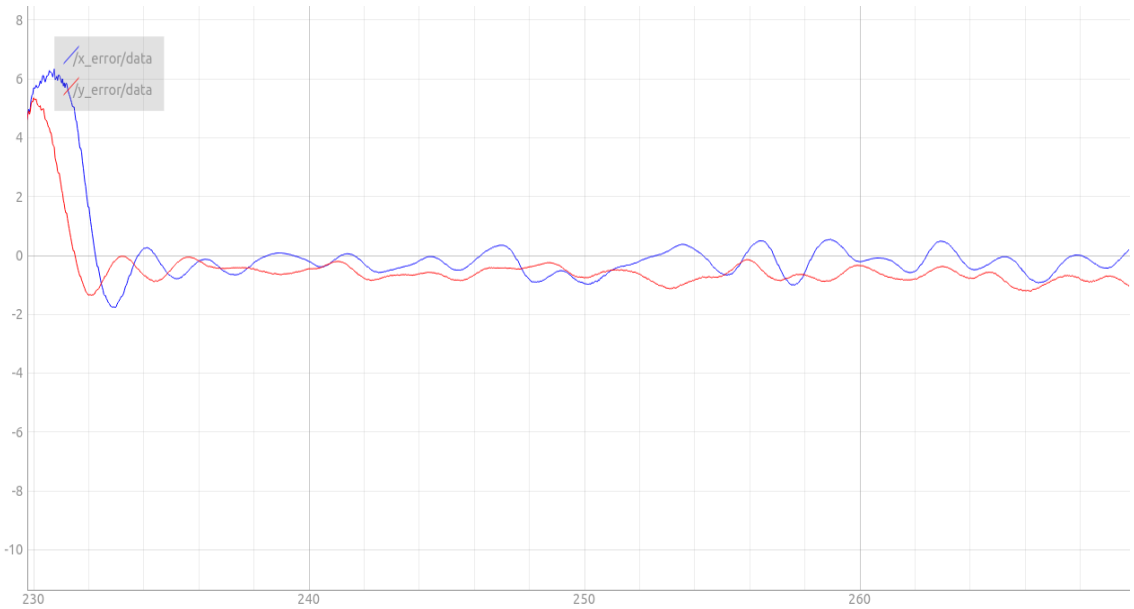


figure 1.7.6.1.3: Response of a classic PID controller

Pessen Integral rule

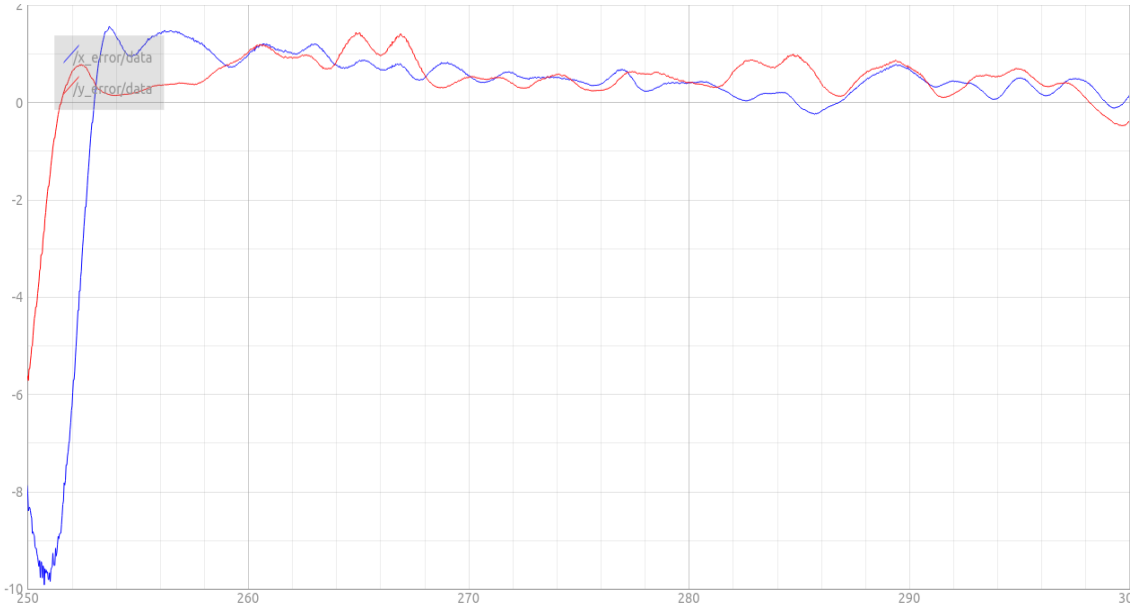


figure 1.7.6.1.4: Response of a Pessen Integral Rule PID controller

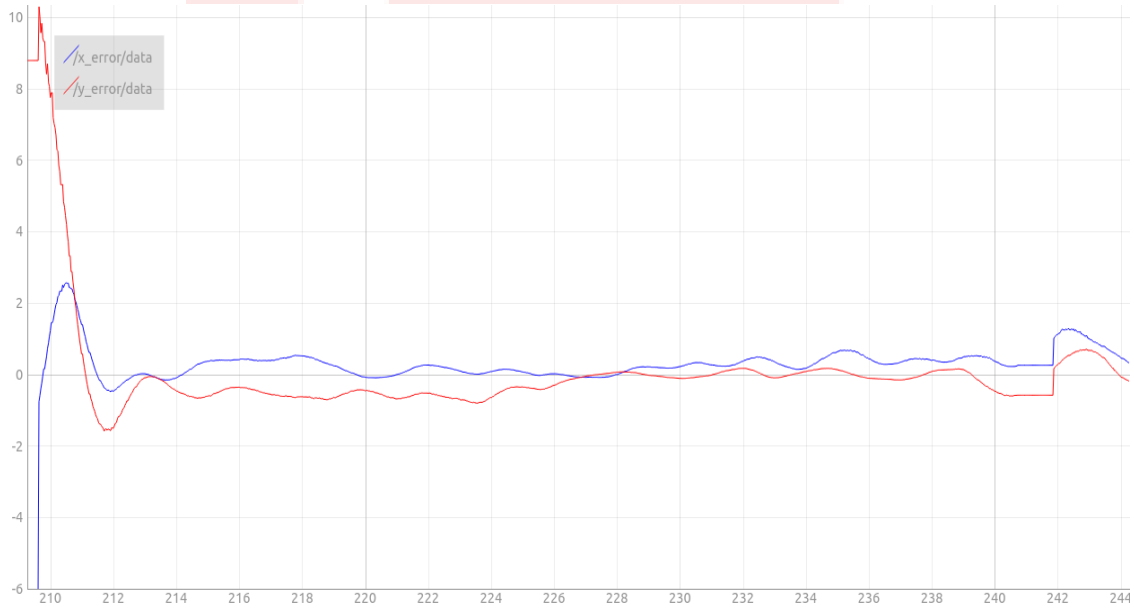


figure 1.7.6.1.5: Response of a Pessen Integral Rule PID controller

Less Overshoot PID

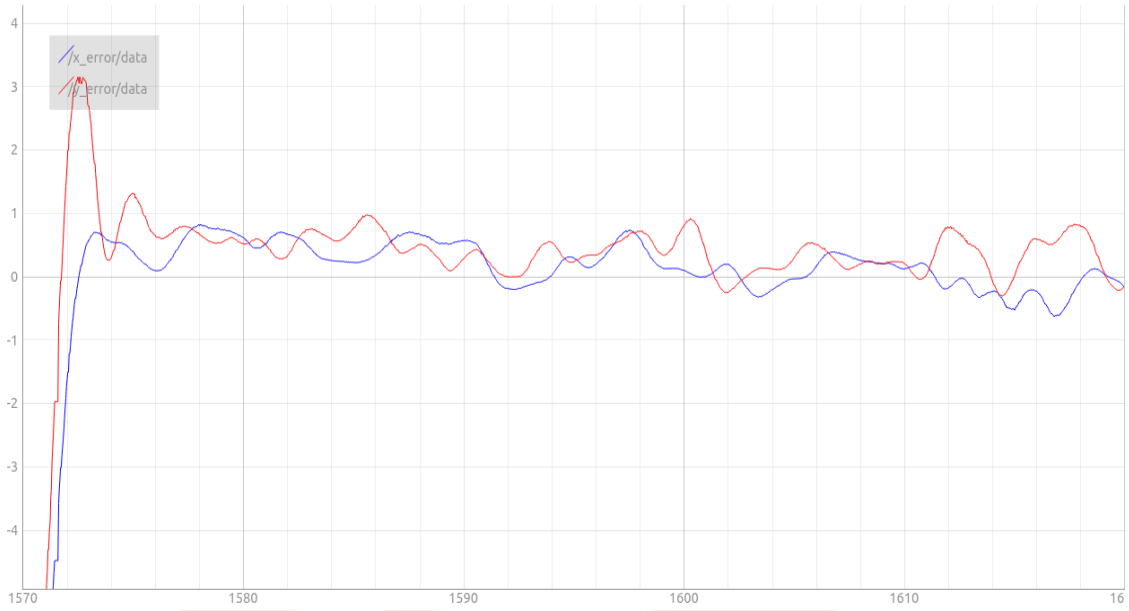


figure 1.7.6.1.6: Response of a less-overshoot PID controller

In this, it is expected that there will some overshoot above the setpoint.

No Overshoot PID

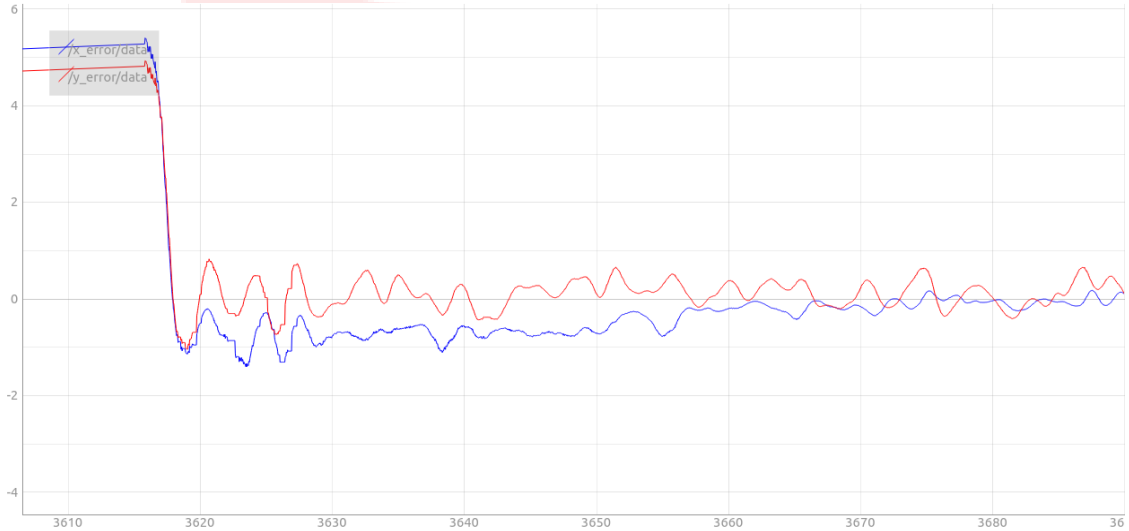


figure 1.7.6.1.7: Response of no-overshoot PID controller

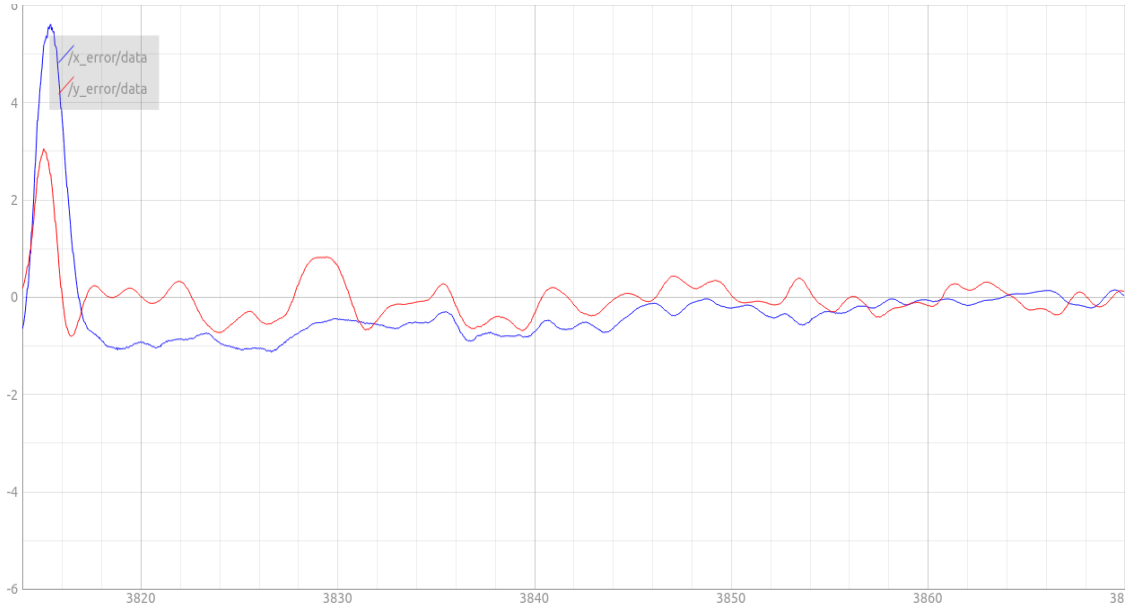


figure 1.7.6.1.8: Response of a no-overshoot PID controller

In this, it is expected that there will not be any overshoot above the set point.

Depending upon the need and suitability any of the above types can be selected for use.

Advantages

- The PID parameters could be found automatically and it is found to be consistent
- It was observed that the constants do not change over consistent period of time hence, we can only auto-tune it once and just use the obtained constants over and over again.

Disadvantage

- The drone must be monitored while auto-tuning so that it doesn't go out of camera frame. (watch the video in demo section for better understanding)

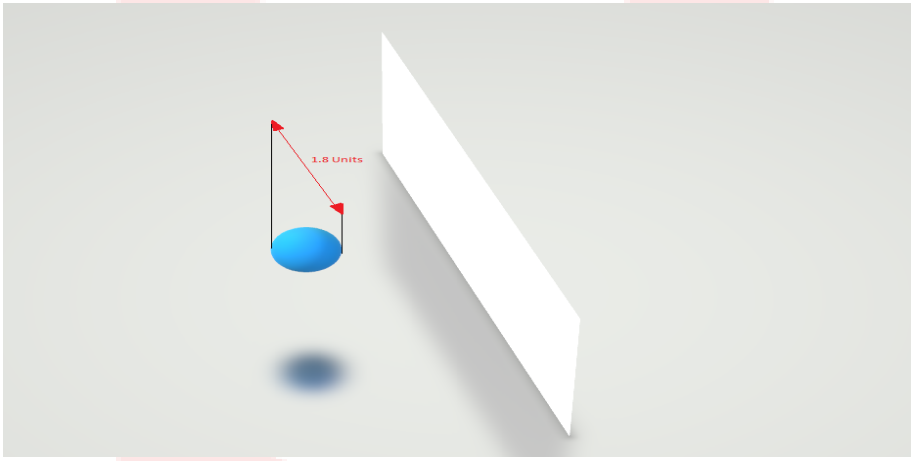
2. Iteration Based Auto - Tuning

This method calculates optimum values of PID parameters for the controller by tuning during the flight. The user enters a range for the possible values

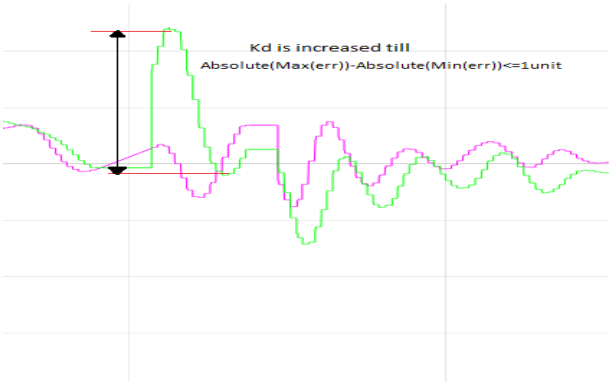
of the parameters and the code continuously changes them based on the principles of control theory.

Algorithm

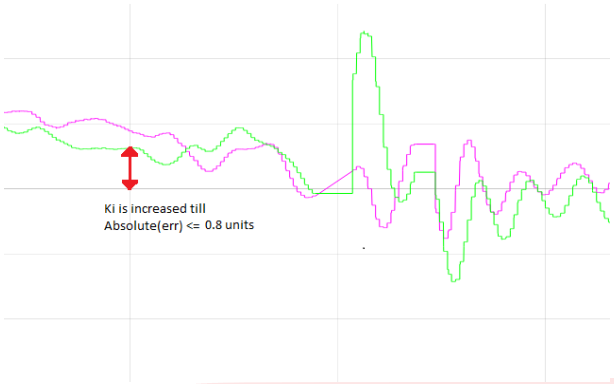
1. Set Kp, Ki and Kd of pitch and roll to minimum.
2. Set Kp to maximum and Ki, Kd of altitude to minimum.
3. Increase Kp wrt rate of change of error for roll and pitch and decrease for altitude.
4. After the drone crosses a sphere of radius 1.8 units ten times, Kp is set constant.



5. Kd is increased again wrt rate of change of error till twice amplitude is less than 1 unit (for fifty cycles)



6. Then Ki and Kd are changed till the drone maintains itself in a position of 0.8 units circle (for ten seconds)



Click here to access the python program for Iteration Based Auto - Tuning

Understanding the code

These small snippets from the code will help in understanding it better

1. We subscribe to the co ordinates given by camera overhead

```
rospy.Subscriber("/whycon/poses", PoseArray, callback1) #Subscribed to whycon co-ordinates
```

2. The global variables for the x, y and z co ordinate store the float values given by the camera.

```
def callback1(msg):
  global x_co
  global y_co
  global z_co
  x_co=msg.poses[0].position.x
  y_co=msg.poses[0].position.y
  z_co=msg.poses[0].position.z
```

3. Initialisation function asks for the input range for the PID parameters

```
def Initialisation():
  global Kp_min
  global Kp_max
  global Kd_min
  global Kd_max
  global Ki_min
  global Ki_max
  global kp_auto_x
  global kp_auto_y
  global kp_auto_z
  global kd_auto_x
  global kd_auto_y
  global kd_auto_z
  global ki_auto_x
  global ki_auto_y
  global ki_auto_z

  print 'Enter the minimum Kp value for pitch, roll and z axis respectively'
  Kp_min[0]=input()
  Kp_min[1]=input()
  Kp_min[2]=input()
  print Kp_min
```

4. A node is initialised to publish commands to drone.

```
    rospy.init_node('drone_server')
    command_pub = rospy.Publisher('/drone_command', PlutoMsg, queue_size=1)
```

5. This part publishes the necessary values for the drone to get armed.

```
if(j==0):
  timeout1 = time.time() + 3 #For 3 seconds the drone is armed
  while True:
     if (time.time()>=timeout1):
       j=1
       timeout1=0
       break
     cmd.rcRoll=1500
     cmd.rcYaw=1500
     cmd.rcPitch =1500
     cmd.rcThrottle =1000
     cmd.rcAUX4 =1500
     command_pub.publish(cmd)
```

6. After 3 seconds, the code enters into the auto tuning state.

```
elif(j==1): #After three seconds this gets initiated
  print 'Auto tuning initiated'
  start_time_tuning=time.time()
  while True:
      motion()
      if(out[2]>max_vel_z):
        out[2]=max_vel_z
      elif(out[2]<min_vel_z):
        out[2]=min_vel_z
      if(out[1]>max_vel):
        out[1]=max_vel
      elif(out[1]<min_vel):
        out[1]=min_vel
      if(out[0]>max_vel):
        out[0]=max_vel
      elif(out[0]<min_vel):
        out[0]=min_vel
      cmd.rcThrottle =1500 - out[2]
      cmd.rcRoll=1500 - out[1]
      cmd.rcPitch =1500 - out[0]
      cmd.rcYaw=1500
      command_pub.publish(cmd)
      err_pub_x.publish(err[0])
      err_pub_y.publish(err[1])
      err_pub_z.publish(err[2])
```

7. The motion function helps in setting the initial values given by the user and calculating the error between the set point and the current position.

```
if(flag_initialize==0): # Initial values are set with the help of this flag.
  Kp=[kp_auto_x,kp_auto_y,kp_auto_z]
  Ki=[ki_auto_x,ki_auto_y,ki_auto_z]
  Kd=[kd_auto_x,kd_auto_y,kd_auto_z]
  err[a]=list_co[count][0]-x_co
  err[a+1]=list_co[count][1]-y_co
  err[a+2]=list_co[count][2]-z_co
  flag_initialize=1
  PID(err)
else:
  err[a]=list_co[count][0]-x_co #The zeroth, third, sixth..fifteenth values of the array store pitch error
  err[a+1]=list_co[count][1]-y_co #Similarly first, fourth, seventh...sixteenth values of the array store roll error
  err[a+2]=list_co[count][2]-z_co #Similarly second, fifth, eigth...seventeenth values of the array store altitude error

  err_x[w]=list_co[count][0]-x_co #Array of 54 elements to store pitch error. This is used to find max and min values
  err_y[w]=list_co[count][1]-y_co #Array of 54 elements to store roll error. This is used to find max and min values
  err_z[w]=list_co[count][2]-z_co #Array of 54 elements to store altitude error. This is used to find max and min values
  PID(err)
```

8. The motion function calls and passes the error to PID function for calculating the PID output. A sample time of 23000 micro seconds is used. This is the time it takes for the communication to take place between drone and the code.

```
now_time=time.time()
delta_time=now_time-prev_time
if((delta_time)>0.023): #Sample time
 for f in xrange(a,a+3,1): #As mentioned above, this is for arrangement of the array
   sumerr[f]=sumerr[f]+Ki[f%3]*(err[f])*delta_time
   sumerr1[f]=(err[f]-err_prior[f])/delta_time
   out[f%3] = Kp[f%3] * err[f] + sumerr[f] + Kd[f%3] * sumerr1[f]
   err_prior[f]=err[f]
   prev_time=now_time
```

9. Since the co ordinates update at a frequency of 30 Hz, sampling is applied here as well. Kp tuning gets initiated.

```
now_whycon=time.time()
delta_whycon=now_whycon-prev_whycon
if(delta_whycon>0.030): #Rate of subscribing whycon co-ordinates
  if(flag_Kp_start==0): #For Finding value of Kp
    print 'Kp tuning initiated'
    if(math.fabs(err[a])<1.8 and math.fabs(err[a+1])>1.8 and math.fabs(err[a+2])>2):
      r=0
      err_z_auto=err[a+2]
      Kp[1]=Kp[1]+0.01 #rate of change of error
      Kp[2]=Kp[2]-(err_z_auto-err_prior_z_auto)/(delta_whycon*100)
      err_prior_z_auto=err_z_auto
    elif(math.fabs(err[a])<1.8 and math.fabs(err[a+1])>1.8 and math.fabs(err[a+2])<2):
      r=0
      Kp[1]=Kp[1]+0.1 #rate of change of error
```

10. If the drone enters into an imaginary sphere of approximately 1.8 units radius, Kp tuning gets completed.

```
    elif(math.fabs(err[a])<=1.8 and math.fabs(err[a+1])<=1.8 and math.fabs(err[a+2])<=2):
      r=r+1
      if(r==10): # To check the condition 10 times
        flag_Kp_start=1
        r=0
```

11. After Kp, Kd tuning gets initiated. The code continuously monitors the amplitude of the error graph and increases Kd wrt rate of change of error. If twice the amplitude is less than 1 unit for x and y axis and less than 1.6 units for the z axis (for 10 cycles) then next part of the code starts.

```
elif(flag_Kp_start==1): #Kd tuning initiated
print 'Kp tuned'
print 'Kd tuning initiated'
if((math.fabs(max(err_x))-math.fabs(min(err_x)))<=1 and (math.fabs(max(err_y))-math.fabs(min(err_y)))<=1 and (math.fabs(max(err_z))-math.fabs(min(err_z)))<=1.6):
#Twice the amplitude is less than 1 unit.
  r=r+1
  if(r==10): #Condition checked 10 times
    r=0
    flag_Kp_start=2
#When pitch and roll are damped but altitude is not
elif((math.fabs(max(err_x))-math.fabs(min(err_x)))<=1 and (math.fabs(max(err_y))-math.fabs(min(err_y)))<=1 and (math.fabs(max(err_z))-math.fabs(min(err_z)))>1.6):
  err_z_auto=err[a+2]
  Kd[2]=Kd[2]+(math.fabs((err_z_auto)-(err_prior_z_auto))/(delta_whycon*150))
  err_prior_z_auto=err_z_auto
#When pitch and altitude are damped but roll is not
```

12. An important addition to this part is to take care of the condition where over damped case in any of the axis might arrive. Hence if twice the amplitude is greater than 2 units in the x and y axis, Kd starts to reduce wrt rate of change of error.

    Video link describing the problem and solution is mentioned in the Challenges Section.

```
elif((math.fabs(max(err_x))-math.fabs(min(err_x)))<1 and (math.fabs(max(err_y))-math.fabs(min(err_y)))>1 and (math.fabs(max(err_y))-math.fabs(min(err_y)))<=2 and (math.fabs(max(err_z))-math.fabs(min(err_z)))<1.6):
  r=0
  err_y_auto=err[a+1]
  Kd[1]=Kd[1]+(math.fabs(err_y_auto-err_prior_y_auto)/(delta_whycon*150))
  err_prior_y_auto=err_y_auto
#When pitch and altitude are damped and roll is overdamped
elif((math.fabs(max(err_x))-math.fabs(min(err_x)))<1 and (math.fabs(max(err_y))-math.fabs(min(err_y)))>2 and (math.fabs(max(err_z))-math.fabs(min(err_z)))<1):
  r=0
  err_y_auto=err[a+1]
  Kd[1]=Kd[1]-(math.fabs(err_y_auto-err_prior_y_auto)/(delta_whycon*150))
  err_prior_y_auto=err_y_auto
#When pitch is damped but roll and altitude are not
```

13. From here, Ki and Kd tuning goes on simultaneously. With increase in Ki to reduce the steady state error, oscillations generally occur. Hence Kd is increased to damp the oscillations. Again as discussed above the conditions take care of the problem that might arise because of the over damped condition.

```
elif(flag_Kp_start==2): #Ki tuning initiated
    print 'Kp tuned'
    print 'Kd tuning in progress'
    print 'Ki tuning initiated'
    now_ki=time.time()
    if((now_ki-prev_ki)>=1): #Ki value is updated after every 2 seconds
    #When pitch, roll and altitude have offset
      if(math.fabs(err[a])>0.8 and math.fabs(err[a+1])>0.8 and math.fabs(err[a+2])>1):
        r=0
        Ki[0]=Ki[0]+0.25
        Ki[1]=Ki[1]+0.25
        Ki[2]=Ki[2]+0.25
    #When pitch and roll have offset
      elif(math.fabs(err[a])>0.8 and math.fabs(err[a+1])>0.8 and math.fabs(err[a+2])<1):
        r=0
        Ki[0]=Ki[0]+0.25
        Ki[1]=Ki[1]+0.25
```

14. After the error is less than 0.8 units in the x and y axis and less than 1 unit in the z axis for 10 seconds, the auto tuning gets completed. From here on, Kp, Ki and Kd are constants. Way point navigation gets initiated.

```
if(flag_Kp_start==3 and j==1): #If drone is tuned increase count for waypoint after 10 sec of reaching the required point.
  if(math.fabs(err[a]<=0.8) and math.fabs(err[a+1]<=0.8) and math.fabs(err[a+2])<=1):
    if(time.time()>timeout):
      print 'Moving to point ',count
      count=count+1
      if(count==4):
        count=0
        j=2
      timeout=time.time()+10
```

15. These conditions are set for all the PID parameters so that they stay in the range provided by the user.

```
if(Kp[0]<=Kp_min[0]): #Pitch Kp minimum cap
    Kp[0]=Kp_min[0]
elif(Kp[0]>=Kp_max[0]): #Pitch Kp maximum cap
    Kp[0]=Kp_max[0]
```

16. After way point navigation, landing gets initiated. Throttle is kept constant so that the drone slowly comes down. At the same time, PID controller keeps on working for x and y axis so that the drone lands at the same position specified by the user.

```
elif(j==2):
  print 'Landing initiated'
  timeout1 = time.time() + 8
  while True:
      motion()
      if(z_co>=27 and (time.time()>=timeout1)):
        timeout1=0
        j=3
        i=1
        break
      if(out[1]>max_vel):
        out[1]=max_vel
      elif(out[1]<min_vel):
        out[1]=min_vel
      if(out[0]>max_vel):
        out[0]=max_vel
      elif(out[0]<min_vel):
        out[0]=min_vel
      cmd.rcThrottle =1475
      cmd.rcRoll=1500 + out[1]
      cmd.rcPitch =1500 + out[0]
      cmd.rcYaw=1500
      command_pub.publish(cmd)
      err_pub_x.publish(err[0])
      err_pub_y.publish(err[1])
      err_pub_z.publish(err[2])
```

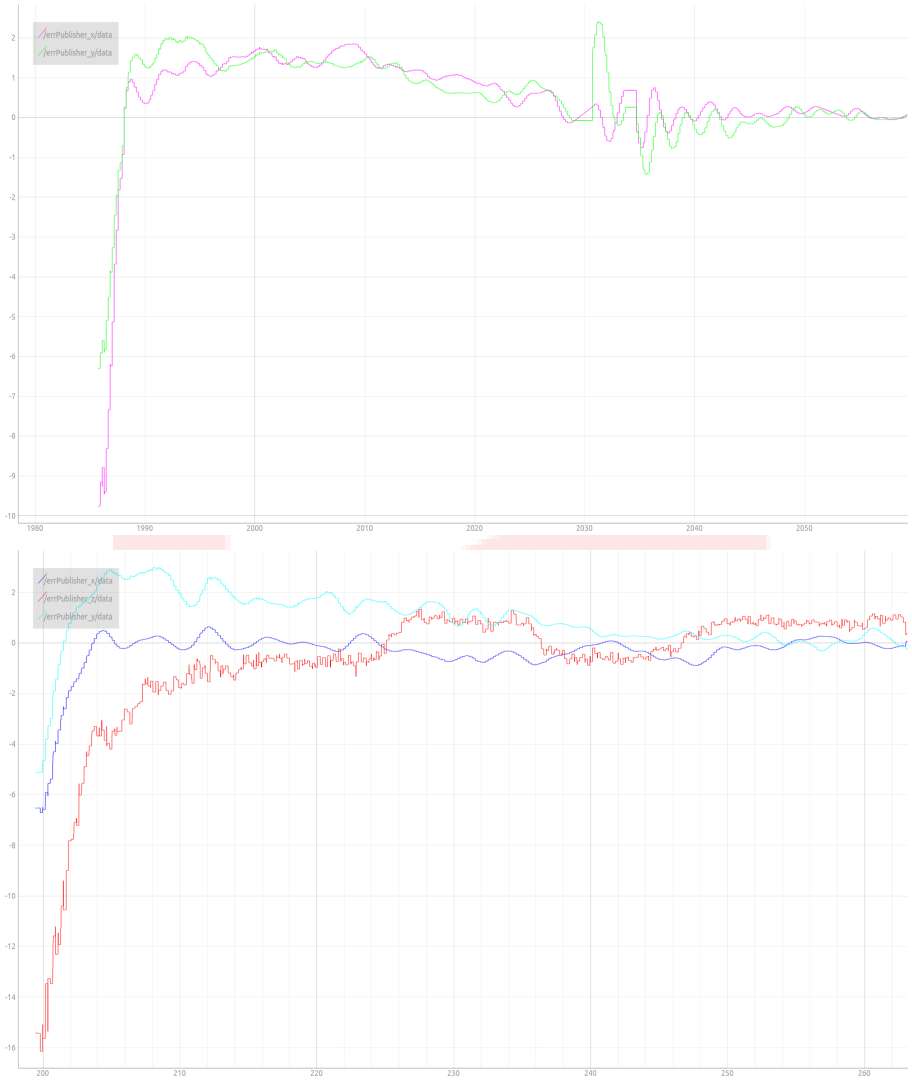
17. After landing, the drone gets disarmed.

```
elif(i==1):
  timeout1 = time.time() + 1
  while True:
    if (time.time()>=timeout1):
      i=0
      timeout1=0
      print 'The drone has landed'
      rospy.sleep(.1)
      break
    cmd.rcThrottle =1300
    cmd.rcAUX4 = 1200
    command_pub.publish(cmd)
```

18. The error is continuously published for the graph.

```
err_pub_x = rospy.Publisher('/errPublisher_x', Float64, queue_size=1) #Publishing error for graph
err_pub_y = rospy.Publisher('/errPublisher_y', Float64, queue_size=1)
err_pub_z = rospy.Publisher('/errPublisher_z', Float64, queue_size=1)
```

Below are couple of graphs obtained by auto tuning



Advantages:

1. No human monitoring or intervention required **even in a restricted frame.**
2. Auto tuning takes place on the go.

Disadvantages:

1. On an average it takes 75 seconds for auto tuning to complete.

1.8 Manual Tuning versus Auto - Tuning

One of the goals of the project was to ponder upon the advantages of auto-tuning over manual tuning.
The results that were obtained from auto-tuning clearly suggests that it is a more efficient way to tune the PID parameters.
Time taken to tune the PID parameters via auto-tuning is hardly about one minute whereas manual tuning may take an hour or sometimes even one day.
Manual tuning is human dependent i.e tuning depends on the person, his ability to interpret the tuning process and so on, where as auto-tuning doesn't require the human to do these mathematical stuffs.
As far as efficiency is concerned in terms of stability and response time, it depends on extent of tuning.
For example, a perfect manual tuned PID controller will have same characteristics as that of auto-tuning but a mediocre tuned PID controller will be no-match to the auto-tuning.
In short auto-tuning gives consistent and optimum response every time !

1.9 Demo

- **Auto-tuning based on Ziegler-Nichols approach**
  The drone is held at (0,0,20) and the code is run.
  The drone will first make oscillations in z axis , then in x and lastly in y axis. This is the auto-tuning phase.
  Once Kp, Ki and Kd values are obtained the way point navigation starts. It traverses to (-4,-4,15) , stays there for a few second then travels to (4,-4,20).After this it goes and lands on the platform at (0,0,30)

  Watch the demo here!

- **Iteration Based Auto - Tuning**
  The drone takes off from the ground and changes the PID parameters based on the method mentioned above. After stabalisiing at (0,0,15) it goes to (-6,-4,23) and then to (-6,4,19). After completing these way points, it stabalises itself in the x and y direction and then lands on the platform.

  Watch the demo here!

1.10 Future Work

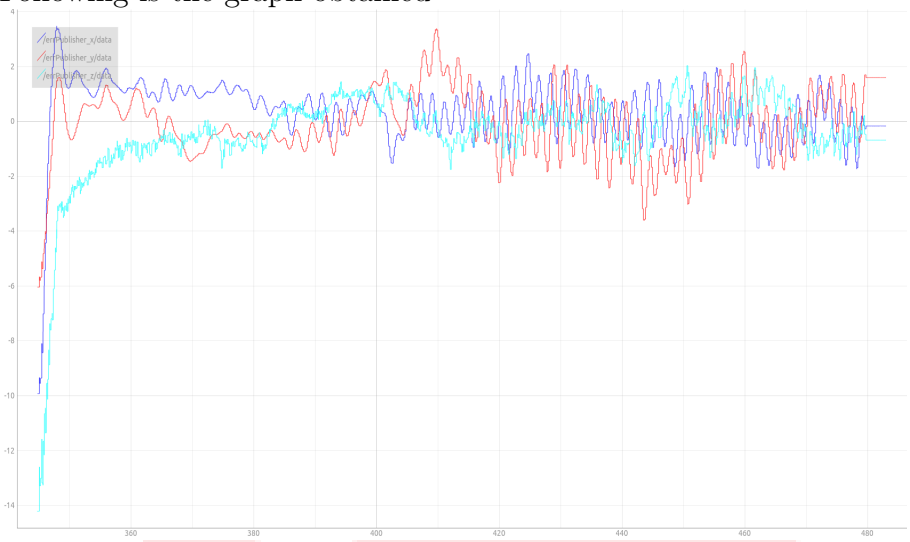
- Currently the PID takes the input only from the whycon, to make the navigation and position holding of the drone more robust the acceleration and gyroscope inputs could be taken and be fed as other inputs to the PID.

- Also in the current scenario, it is observed that the whycon input of z-coordinate isn't consistent at a given point, i.e consecutive camera frames give variation in z-coordinate even if it is as the same point. This makes the PID controller confused due to which it constantly tries to stabilise the drone. Hence there is scope for improvement in this domain.

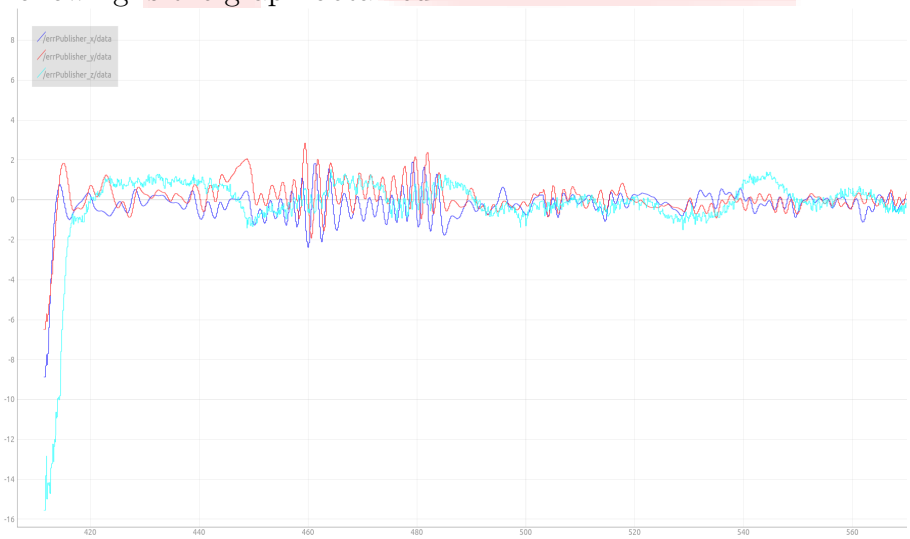
1.11 Bug report and Challenges

1.11.1 Challenges faced in Iteration Based Auto Tuning Method

Video Describing over damped PID Controller Click Here
Following is the graph obtained



Video Describing Solution for over damped PID Controller Click Here
Following is the graph obtained



Any issues in code and hardware.
Any failure or challenges faced during project

1.11.2 Hardware Breakdowns

Throughout the duration of the project there were a lot of hardware breakdowns.
The major failure in the hardware lied in faulty magnetometer.
It was observed that the magnetometer gave improper readings because of which the drone behaved inconsistently.

To check if the drone bears a proper magnetometer follow the code given in this section.
Click here to access the ROS package Here are a couple of snaps that depict the behaviour of a faulty(figure 1.11.2.1) and a working magnetometer(figure 1.11.2.2)

Faulty

figure 1.11.2.1: Response of a faulty magnetometer

Working

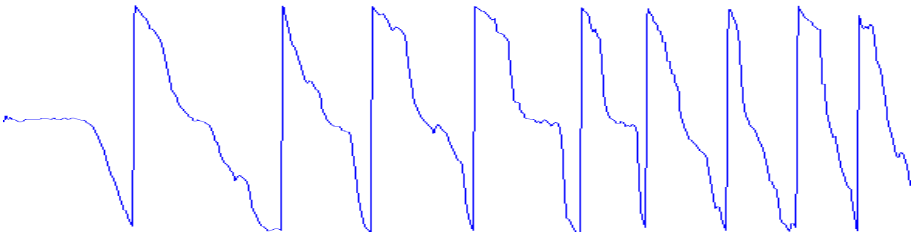


figure 1.11.2.2: Response of a working magnetometer

1.12 References

1. ROS tutorials
2. Improving the Beginners PID Introduction
3. PID Control - A brief introduction
4. Hardware Demo of a Digital PID Controller
5. Controlling Self Driving Cars
6. Arduino PID Autotune Library