25-05-2022

EEH4DSE 01 Dynamiske systemer.

Drone's orientation control: pitch movement function simulation and control.

Studerende:

Ekaterina Wyss Storm au286995

Jesper Egon Petersen au684131

Course Instructors:

Anders Lehmann

Kristian Peter Lomholdt

Content

Introduction.

An unnamed aerial vehicle (UAV), known as a drone, is ai aircraft without a human pilot on board. Essentially, a drone can be remotely controlled or fly autonomously using software-controlled flight plans in its embedded systems, that work together with onboard sensors and a global positioning system (gps) (1.1). Drones' movement is associate with 6 degrees of freedom, they can move in translational directions (left/right, up/down, forward/ backward) and 3 rotational directions (roll, pitch, and yaw). Each motion can be commanded independently of each other. In this project we will limit our focus to the control and simulation one the functions among rotational directions- pitch movement (rotation along horizontal axis). We will build a prototype for a single wing and test the algorithm of pitch control in Matlab &Simulink.

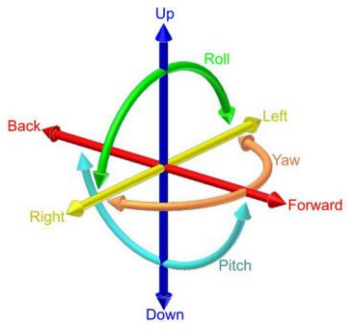


Figure 1: drone movement dimensions

Source: https://emissarydrones.com/what-is-roll-pitch-and-yaw

Problem formulation

what is best control strategy & algorithm for a drone rotation along the horizontal axis (pitch)?

The design process & Time plan:

Start (week 13): System definition and requirements- system's performance is determined (transient response, steady-state error evaluation, stability and etc.).

In our case the hardware specifications are already predetermined:

Sensors: 4 (ultrasound- measures vertical distances, camera- horizontal motion and speed, pressure sensor- altitude, IMU- measures linear acceleration, angular rate (gyroscope)). We are only using IMU sensor for this project.

Propellers: 2; Actuators: Motors- 2.

Steps 1 (week 14): determination of the physical system and specifications from the requirements

Step 2 (week 14): Functional block-diagram &schematic

Step 3 (uge15): Bloch-diagram reduction

Step 4 (week 15): Mathematical model of the system

Step 5 (week 16-17): analyses, design, test.

Step 6 (week 19): simulation of the prototype with Arduino.

Prototype: 2 motors, 2 propellers, 1 wing, 2 speed controllers, 1 IMU sensor (MMA8452), Arduino mega 2650, jump wires, IR sensor; SW**:** Matlab, Simulink

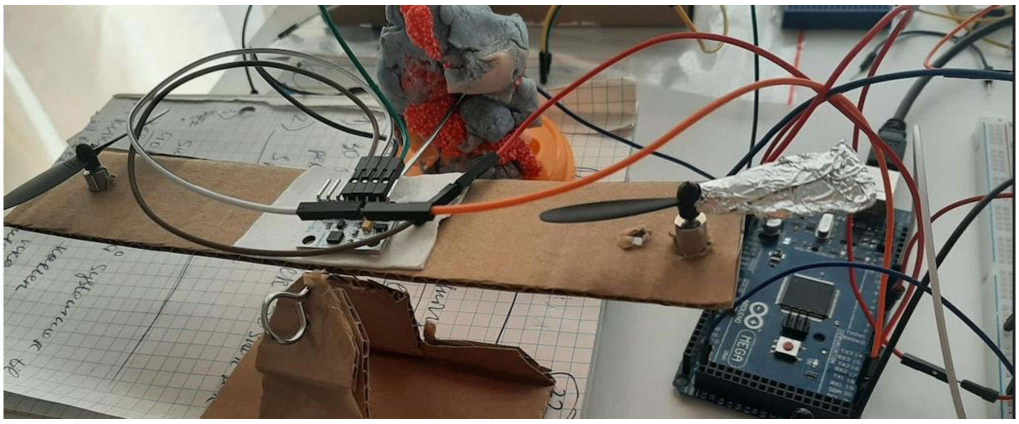


Figure 2. system prototype.

Preliminary system outline:

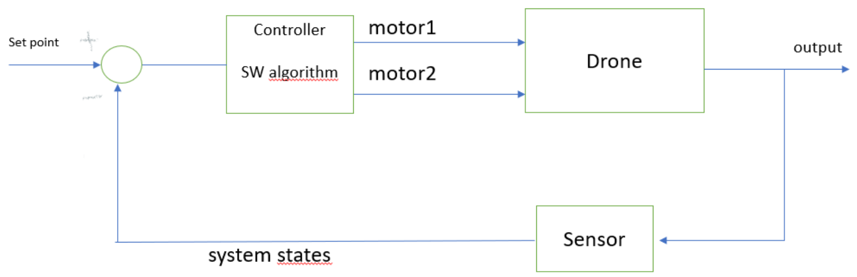


Figure 3. preliminary system outline

Links to videos:

Videos: about experimental set up, accelerometer +IR sensor code:

https://drive.google.com/file/d/1OwvQzRPueJlYtJPDvjNuJCgPmspv8mWv/view?usp=sharing

https://drive.google.com/file/d/1Hk-b0YgO4TUZ7AzfDRPbsHAl_5BaDKGi/view?usp=sharing

System description

A quadcopter has four rotors, each one of which has independent speed, allowing a balanced variation of the rotors' speed and thus generating the thrust and accelerations in the desired directions. Quadcopters are six-degrees-of-freedom (DoF) systems, what means that they can move along the three space axes X, Y and Z, and turn on the aircraft's body axes describing roll (ϕ), pitch (θ) and yaw (ψ) angles. However, in the aircraft, there is only direct control over four DoF, corresponding to the altitude and the three angles: roll, pitch and yaw. Roll and pitch movements also generate motion along the Y and X axes of space, respectively (2.3). Despite the advantages of quadcopters, like other aircrafts, they are extremely unstable systems, and so any imbalance in their motion (especially in roll and pitch) generates angular and linear accelerations, which can cause a collision if not corrected quickly. Moreover, the quadcopter system is nonlinear (2.2) and the movements are coupled with each other. PID controllers offer a simple but effective solution to stabilize the aircraft because they make it possible to treat every variable independently within a limited range in which the behaviour of the quadcopter is approximately linear (Bouabdallah et al., 2005; Castillo et al., 2005).

In this work we are developing a quadcopter's model wing, measuring its' dynamic behaviour in relation to pitch angle framework, after which, based on an analysis of variables, we start implementation of PID controller for pitch angle.

Pitch angle control concept.

Our primary goal is to develop an algorithm for controlling the angle of quadcopter wing. The controller will be designed to determine the magnitude of each motor spinning rate to achieve the desired pitch angle. We set a reference angle max and minimum points (15, -15 degrees) based on the study "Measurement of UAV attitude angles based on a single captured image" (2.4). The major purpose of this controller, like any other controller with feedback, is to constantly calculate the error e(t) as the difference between the required value (set point) and the process variable y(t), and to correct the error by adjusting the control signal u(t). The variable- u(t) is in fact the sum of the proportional, integral and derivative parts. In that way the program will compare the reference angle with the actual pitch angle, calculate the errors and send to the controller which will correct the output.

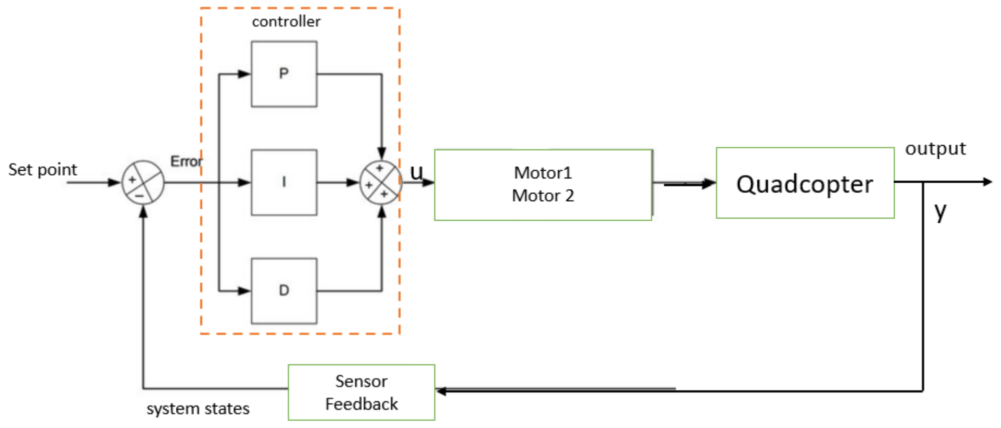


Figure 4. system outline with PID block

Quadcopter prototype.

The physical model for this experiment is based on Z8 smart drones, model Z8W mini quadcopter. Control distance- 60m, frame dimensions -13.5x13.5x3. For the prototype we are using propellers' blades from Z8 (extra parts), 2 – DC 3.7 V Micro Coreless motors with 4800 RPM, 3- axis digital acceleration sensor module MMA-8452.

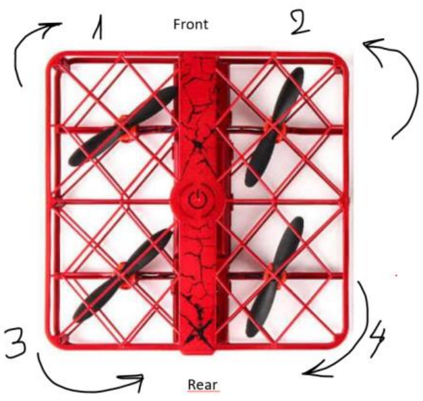


Figure 1. Z8W drone.

Motors configuration –is an “X” configuration, opposing motors spin in the same direction and opposite to another pair. So that roll, pitch, yaw could be commanded independently of each other. To have a balanced quadcopter, the propeller rotation has to be toward the quadcopter main body. To achieve this, the quadcopter motor setup needs to be as follows:

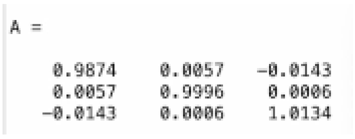
- Front Left – Clockwise motor (CW)
- Front Right – Counter Clockwise motor (CCW)
- Back Left – Counter Clockwise motor (CCW)
- Back Right – Clockwise motor (CW)

In order to fly forward and tilt (pitch), an increase in the quadcopter motor rpm (rotation rate) of rotors 3 and 4 (rear motors) and decrease the rate of rotors 1 and 2 (front motors) is required. The total thrust force will remain equal to the weight, so the drone will stay at the same vertical level (2.1). To simulate quadcopter propeller direction for pitch movement we will be using one wing (13.5 sm.) and two motors- Front left (CW) and Back left (CCW).

Estimating orientation using IMU sensor. Interfacing with Arduino Mega 2560 and MATLAB.

Before going any farther, we started with interfacing Arduino Mega 2560 with Matlab and testing the accelerometer module (MMA8652). The Matlab test for the sensor looks as following: first creating the Arduino object, calling the necessary libraries (I2C and Pololu), creating a sensor object, reading sensor’s data, and converting it into Pith angle with Matlab function (ConvertToPitch). Afterwards we visualized the data acquisition process with live plotting function (AnimatedLine) and saved the data in a matlab and exel file.

The 3-Axis Accelerometer block measures the linear acceleration along the X, Y, and Z axes. The block has one output port, Accel, which outputs the acceleration from the as a [1x3] matrix in g (9.8 m/s^2) (2.3), (table 1).



```
A =

    0.9874    0.0057   -0.0143
    0.0057    0.9996    0.0006
   -0.0143    0.0006    1.0134
```

Table 1. output matrix, acceleration values.

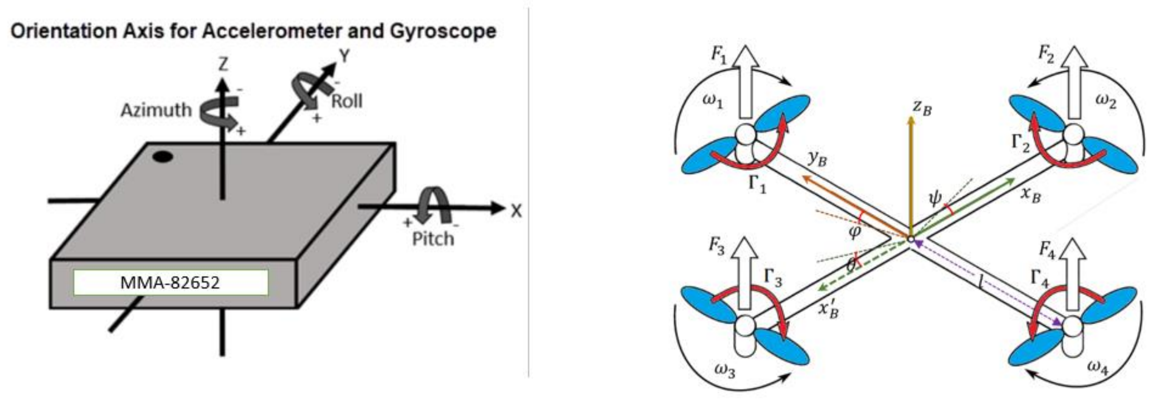
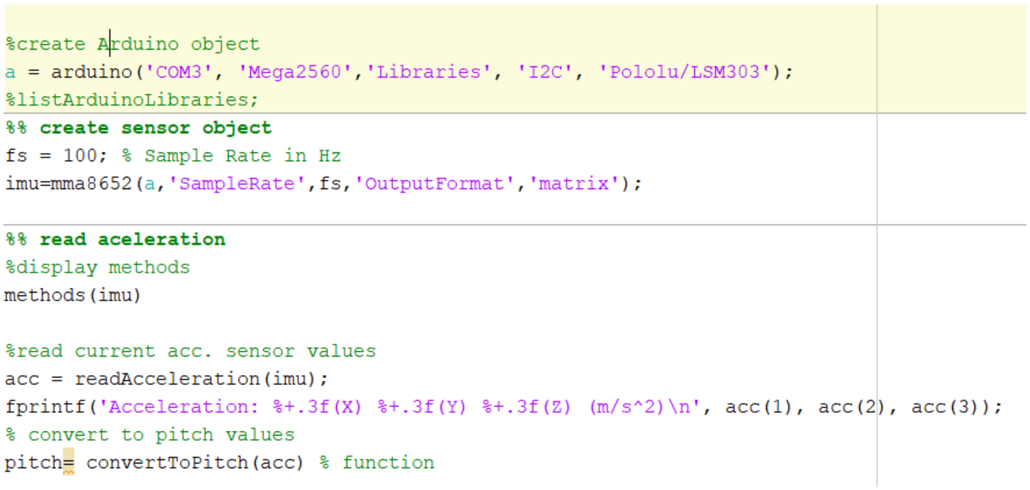


Figure 5. chip line orientation (x,y,z); drone free body diagram. Sources: 1. V. Indragandhi, 2. https://se.mathworks.com/help/supportpkg/microbit/ref/mma8652fc3axisaccelerometer.html

The acceleration values were converted to pitch angle according to this formula: atan(acc_x/(acc_y^2+acc_z^2)).



```
%create Arduino object
a = arduino('COM3', 'Mega2560','Libraries', 'I2C', 'Pololu/LSM303');
%listArduinoLibraries;
%% create sensor object
fs = 100; % Sample Rate in Hz
imu=mma8652(a,'SampleRate',fs,'OutputFormat','matrix');

%% read aceleration
%display methods
methods(imu)

%read current acc. sensor values
acc = readAcceleration(imu);
fprintf('Acceleration: %+.3f(X) %+.3f(Y) %+.3f(Z) (m/s^2)\n', acc(1), acc(2), acc(3));
% convert to pitch values
pitch= convertToPitch(acc) % function
```

Figure 6. Extraction from the Matlab code

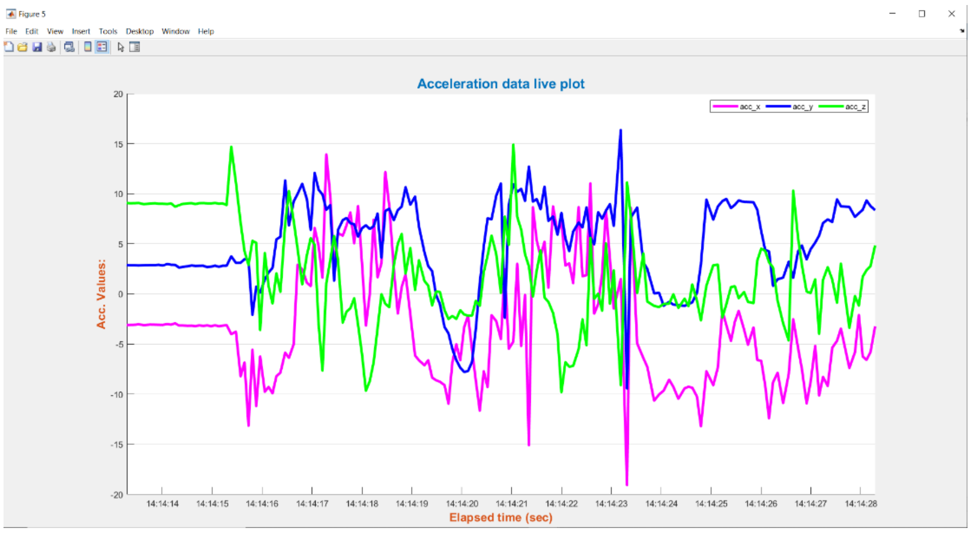


Figure 7. Sensor readings& visualisation

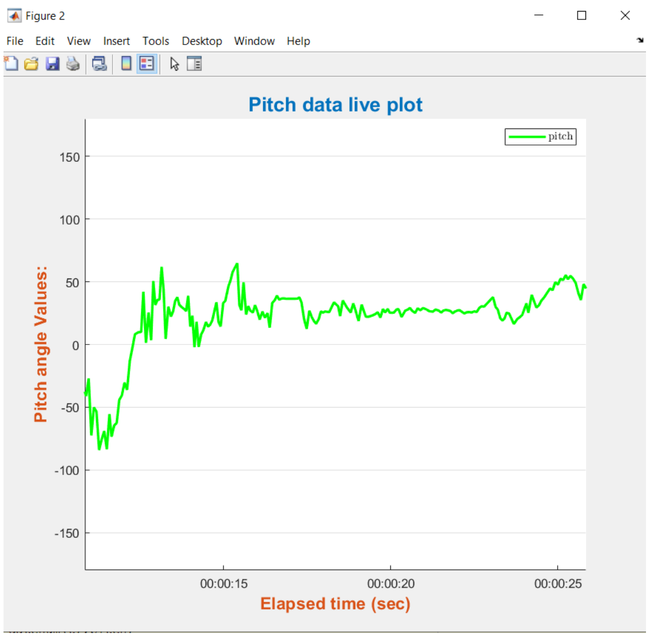


Figure 8. Pitch angle visualisation.

Strategy and Mathematical model.

We will need a mathematical description of the system in order to understand and study its behavior. Usually, these models are equations which describe the relationship between the inputs and outputs of the system (2.2). They can be used to forecasts the behavior of the system, to run simulations and examine the proposed controller. As in this work we focus primary on pitch angle system, the navigation equations derived from kinematics of rigid bodies (3.2), and the equations of motion derived from Newton's laws (3.3) will not be considered. Instead, we will derive a DC motor transfer function by means of system prediction technique and matlab control system toolbox and experimental work.

Model for the experiment.

When we determined the control concept for the project, the next step is to evaluate the performance of the pitch control system. PID controller design requires prior modelling of the system in order to see its behaviour (2.1). The physical model is built up in parameters close to real Z mini drone, although the weight of the prototype is not taken into account. The model is simulated with Arduino mega 2560 and Matlab environment.

In order to design the pitch controller, the transfer function of the motors which are responsible of the quadcopter motion is obtained using MATLAB toolbox-system identification techniques. System identification is a process of using data rather than physical parameters for developing a model of the dynamic system. We will have to start with the experiment and data collection, where one wing model is set up with the accelerometer and IR sensor (figure 9). The various PWM signals will be generated through Arduino, and motors' RMP outputs are registered in respect to pitch angles. The results are presented in a Table 2.

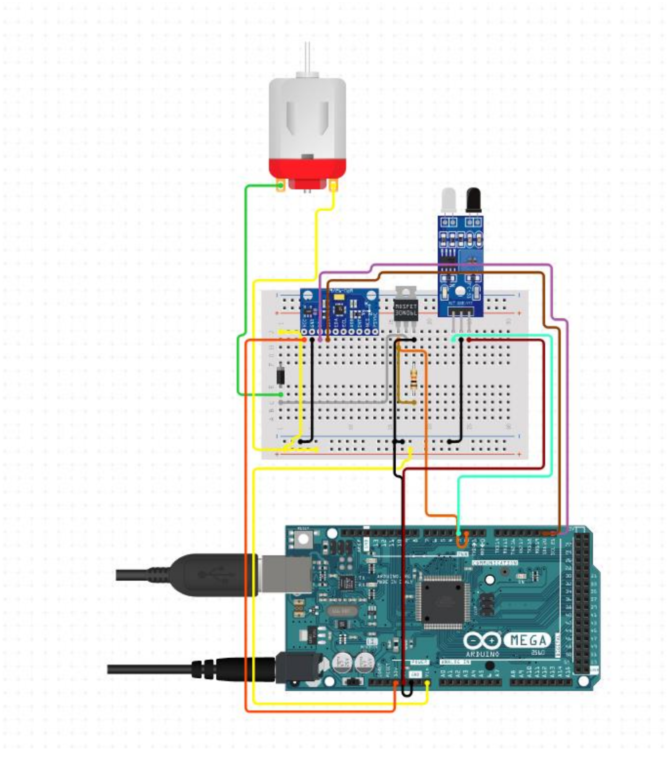


Figure 9. Arduino circuit schematic, with motor, IR sensor, IMU.

Grey- box method.

System identification, with the help of statistical methods builds a mathematical model for a dynamic system using measurements. We are trying to identify the underling dynamics that produces system behaviour by using the inputs and measuring the outputs of the system. The model accuracy highly depends on the accuracy of the measured data (2.3).

Steps of system identification:

1. Data collection: u(t), y(t), sampling time.

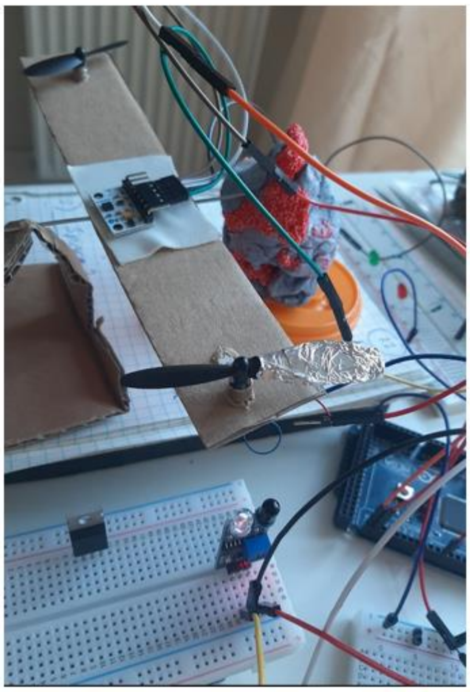
2. Determination of model structure (mathematical relationship between input and output variables that contains unknown parameters).
3. Model evaluation, analyses, validation.

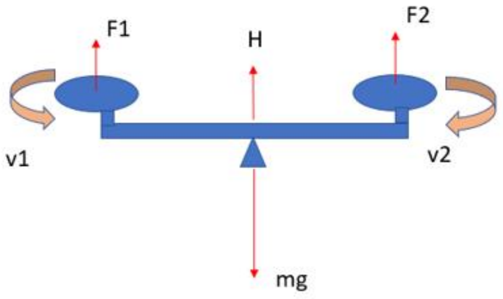
White-box method: the model structure is based upon physical laws and additional relationships+ corresponding parameters. Model obtained directly from physics.

Black-box method: no prior model structure of a system is available; the standard model structure is selected and parameters are tuned to achieve the best model.

We will follow the so -called Grey-box method: provided inputs, measured outputs, some knowledge about the physical parameters of the system.

Test.





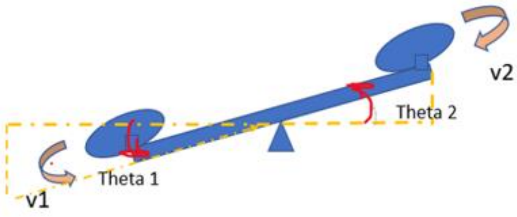


Figure 10. prototype for the test, 1 wing.

A simple rotation is made only by decreasing a motor speed by some value and increasing the diametrically opposed motor speed by the same value. The Input signal is motors' PWM duty cycle in % (with WritePWMVoltage and/or WriteDutyCicle functions of Matlab) output motors' speed (RPM).

The experiment is conducted by going through the following steps:

1. Transmitting variable signals to the motor and measuring the motor's speed in RPM, corresponding pitch angle.
2. The measured input–output data are collected using a data acquisition ports in Arduino MEGA developing board and interfaced with MATLAB.

3. Using system identification toolbox in MATLAB to obtain the transfer function of the used motor.

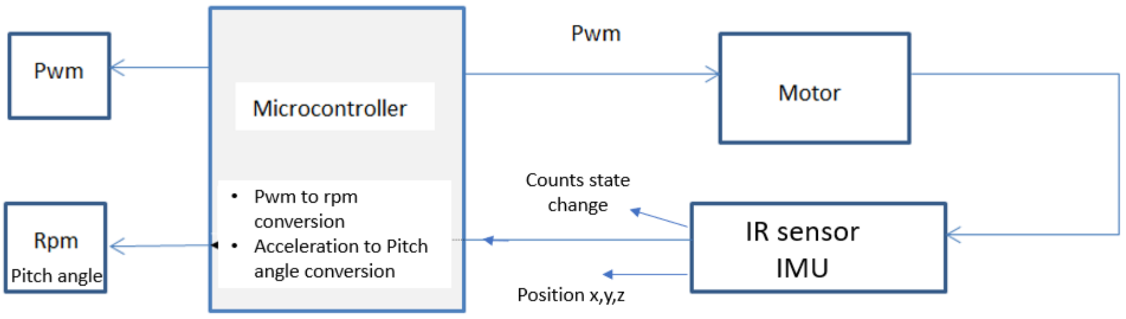
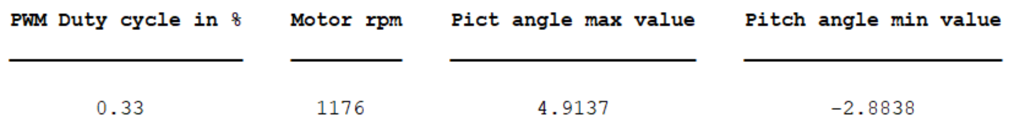


Figure 11. block diagram for the experimental set up.

IR sensor.

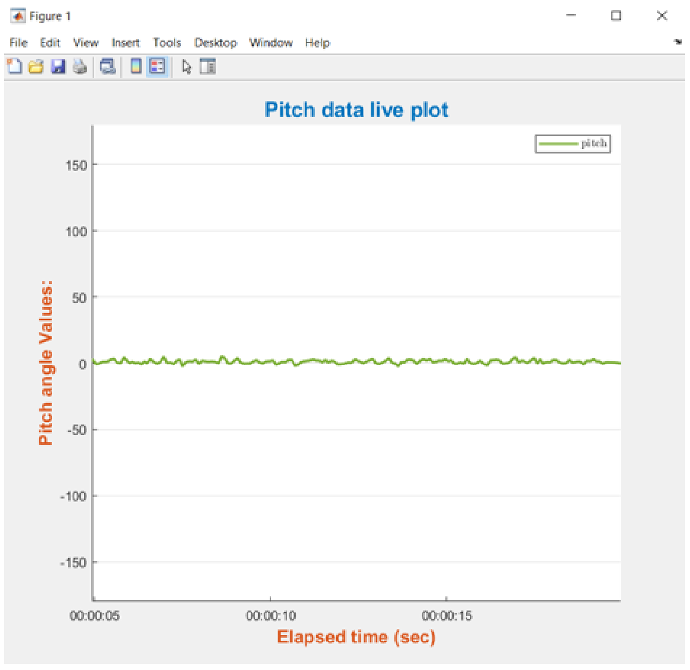
IR Sensor is used for measuring the number of rotations of a motor in RPM. We have interfaced IR sensor module with Arduino, and it works on basic principle of an encoder. When a shaft/body rotates, same point on it repetitively comes to original position after one revolution. By measuring time (in seconds) with Matlab stopwatch timer function tic& toc we will get the motor's rpm per minute. The `tic` function records the current time, and the `toc` function uses the recorded value to calculate the elapsed time. The sampling time is set 20 seconds. A piece of aluminium foil is used to set a reference point on a propeller. Also, it reflects light thus enhances sensitivity of IR sensor. Apposite to Arduino language, Matlab reads "LOW" when the aluminium foil comes in front of the sensor. The program then registers the state change and starts the counter function, the number of motor revolutions per minute is calculated and the value is displayed in a table. For the prototype we are using DC 3.7 V Micro Coreless motors with 4800 RPM max, that gives 80 rps.

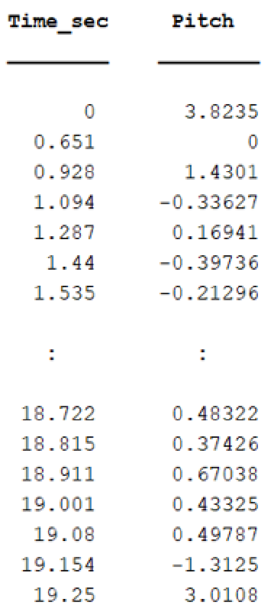


```
PWM Duty cycle in %    Motor rpm    Pict angle max value    Pitch angle min value
___________________    _________    ____________________    _____________________

       0.33              1176              4.9137                  -2.8838
```

Figure 12. Matlab command window output for rpm and pitch angle calculation





```
Time_sec      Pitch
________    ________

       0      3.8235
   0.651           0
   0.928      1.4301
   1.094    -0.33627
   1.287     0.16941
    1.44    -0.39736
   1.535    -0.21296

      :          :

  18.722     0.48322
  18.815     0.37426
  18.911     0.67038
  19.001     0.43325
   19.08     0.49787
  19.154     -1.3125
   19.25      3.0108
```

Figure 12. Pwm 33%, pitch angle plot and pitch data table.

Measuring the motor RPM helps to obtain a relation between input values PWM% with output RPM of the motor and consequently the output pitch angle values.

The experiment results, Table 2:

PWM duty cycle %	RPM	Pitch angle range, degrees
33	1176	(-2.8; 4.9)
48.5	1218	(-10.56; 9.95)
55	1980	(-14. 012; 13.672)
75	2150	(-15.924; 17.749)
85	2456	(-18.57; 21.132)
95	3210	(-21.45; 23.56)

The relation between input signal and pitch angle gives the plot that shows the higher the motor speed, the bigger is the pitch angle. So, by regulating the motor speed, we can control the pitch angle of the drone. In the figure below it can be seen that when motor's duty cycle exceeds 62% the reference pitch angle goes beyond the limits. (-15; 15 degrees). So, we have to make a controller that will (in case the motor goes above 62% PWM) calculate the error and correct the input. It can also been seen that the systems also shows linear behavior.

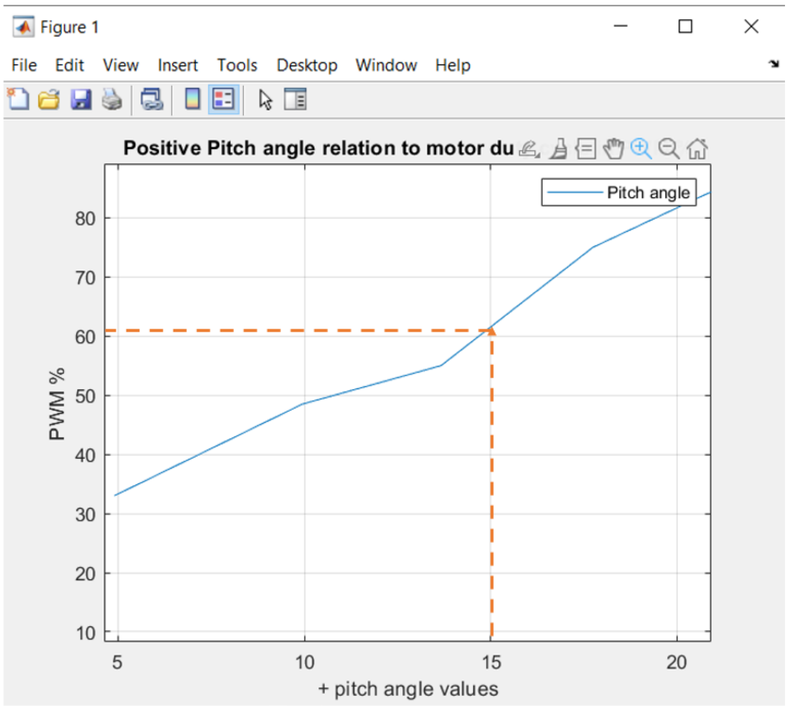
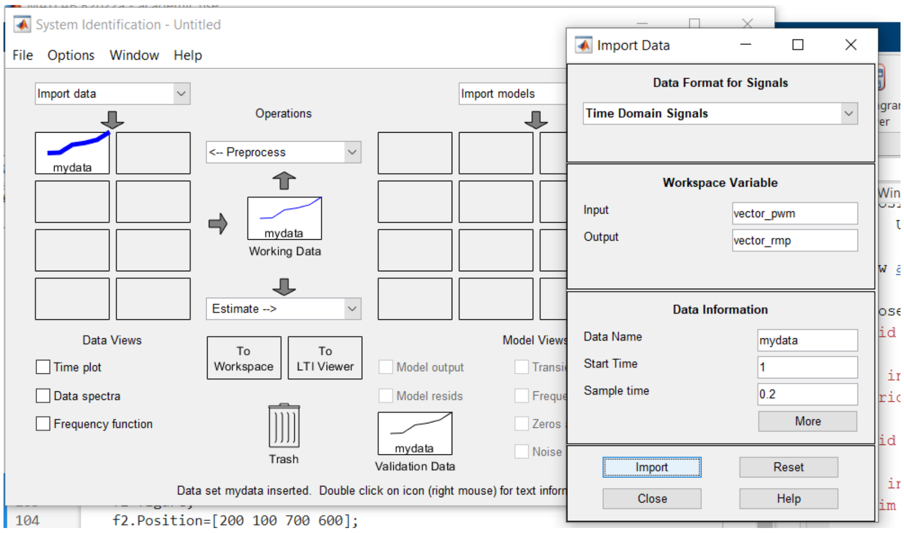


Figure 13. pitch angle data plot to pwm signal inputs.

System identification tool- box.

To process the data into the system identification toolbox in Matlab we created 2 vectors with input and output values. Those were imported into the system and 4 transfer function models were created based on number of poles, zeros and fitting percentage (Table 3).



Input: vector_pwm
Output: vector_rmp
Data Name: mydata
Start Time: 1
Sample time: 0.2

Data set mydata inserted. Double click on icon (right mouse) for text inform

```
104    f2.Position=[200 100 700 600];
```

Figure 14. System identification toolbox.

Poles, zeroes	Transfer Function	Fitting percentage
1,0	127.8 ---------- s + 2.773	79.05%
1,1	From input "u1" to output "y1": 35.64 s + 1.11e24 ----------------- s + 3.111e22	44.92%
2,1	From input "u1" to output "y1": -95.74 s + 653.7 ------------------------- s^2 + 0.0006513 s + 18.52	56.25%
3,2	From input "u1" to output "y1": 8.914 s + 5.959 ---------------------------------- s^3 + 0.4083 s^2 + 1.017 s + 0.1075	100%

Table 3. TF models for the data.

The transfer function corresponding to best fitting identified model is:

$$\frac{\text{Output(rpm)}}{\text{Input(pwm)}} = \frac{8.914\,s + 5.959}{s^3 + 0.4083\,s^2 + 1.017\,s + 0.1075}$$

```
Status:
Estimated using TFEST on time domain data "mydata".
Fit to estimation data: 100% (stability enforced)
MSE: 6.635e-25
```

The obtained transfer function will be used in building the MATLAB/SIMULINK model for analyses. Although the transfer function can be used only for linear systems, it yields more intuitive information then the differential equation (1.3). We will be able to change systems' parameters and rapidly sense the effect of these changes on system response. As we are interested only in performance of an individual subsystem, we can skip steps with block diagram reduction and move into analyses, design, and test.

PID design.

System Identification for PID Control.

We have obtained a dynamical model using identification techniques, by generating measurable signals and the corresponding system responses. The resulting input-output data is then used to obtain a model of the system through the process of system identification. Time-domain data consists of the input and output variables of the system at a uniform sampling interval over a period. Data is stored as vectors of measured values: of motor PWM inputs and RPM outputs.

In the figure below (Figure 15) is illustrated how the created several models and run them through estimation algorithms to choose a model that is the best possible fit between the measured system response to a particular input and the model's response to the same input. Once a suitable plant model is identified, we can impose control objectives on the plant based on knowledge of the desired behavior of the system. The Identification toolbox function tuned a model 2- tf2 (green graph) so it fits best to the data.

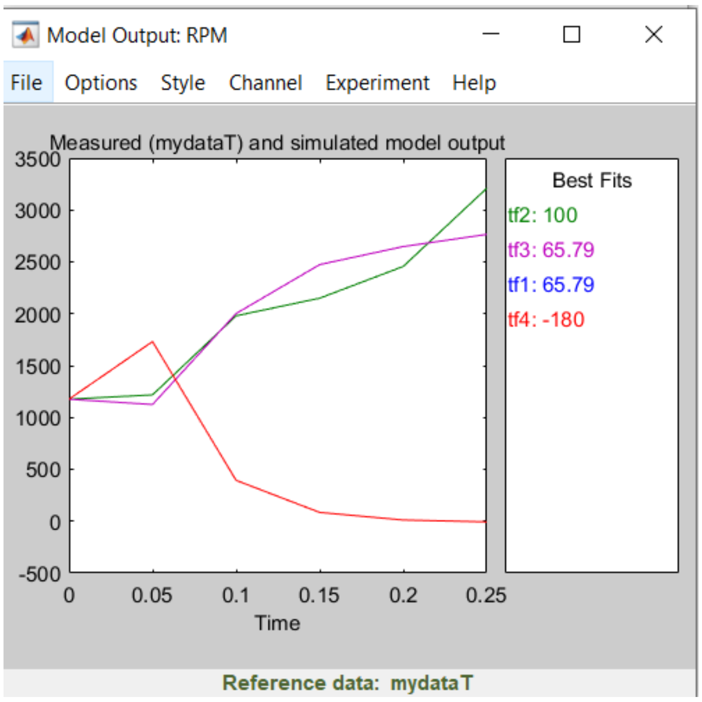


Figure 15. estimating and fitting models using time domain data.

Input and output signals illustrated in System identification toolbox. The system demonstrates linearity (time domain data) as could be seen from the graphs below.

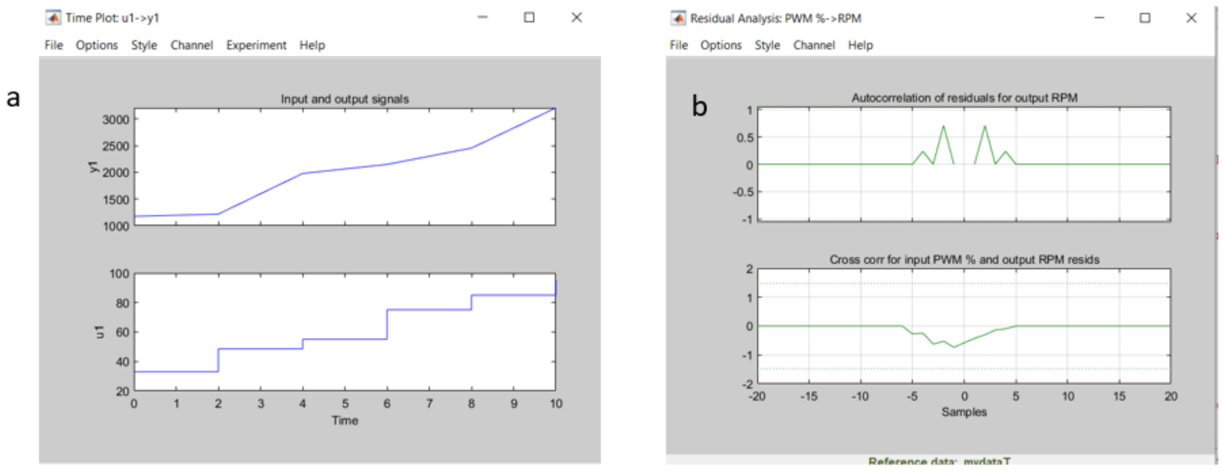
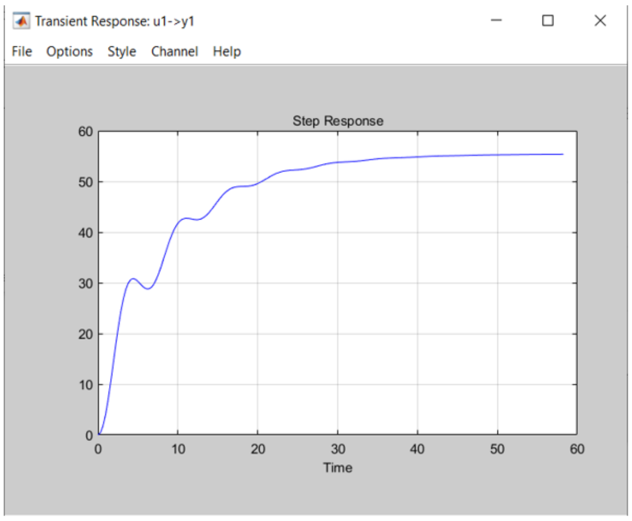


Figure 16. input and outputs signals illustration (a). model residuals autocorrelated (b).

There could be generated plots for transient response, poles- zeroes, autocorrelation for residuals, noise spectrum.



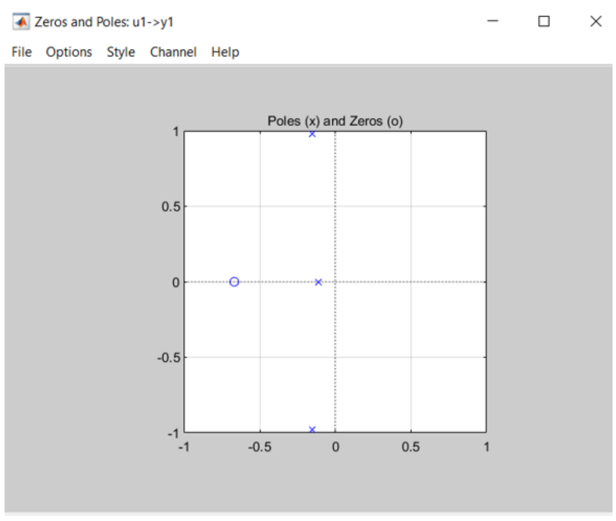
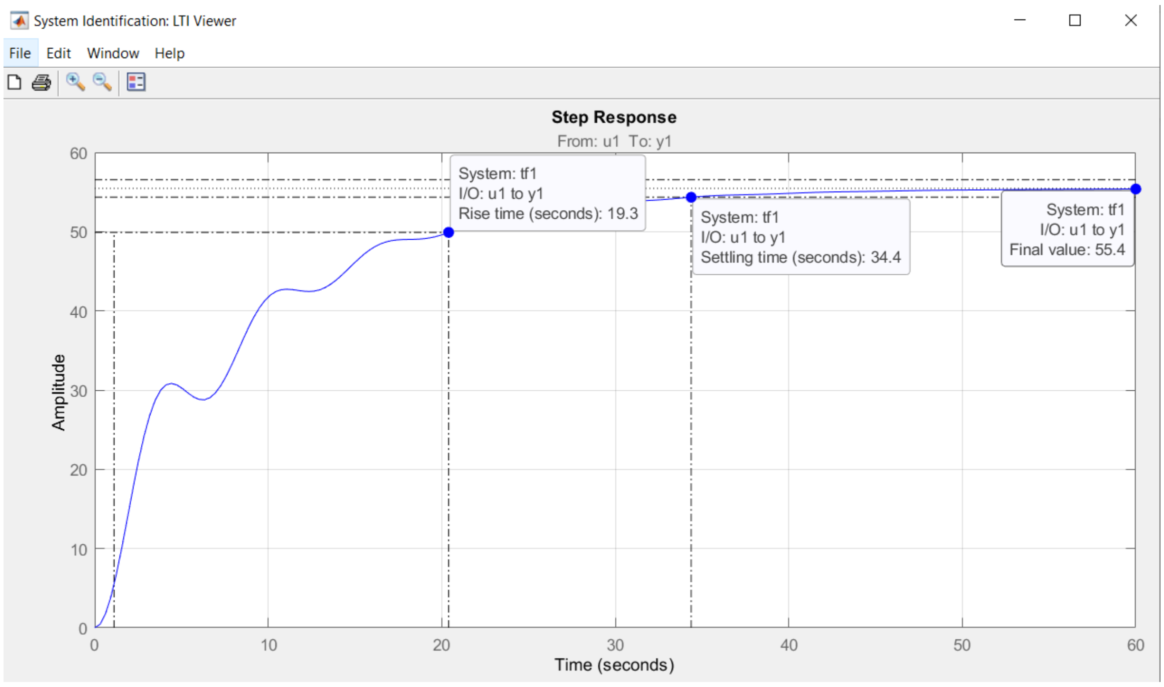


Figure 17. Step response. Pole-zeroes plot.

Open loop response.

The open-loop response of a system is the combined response of the plant and the controller, excluding the effect of the feedback loop (2.3). The uncompensated open-loop system perform can be seen in figures below. We used MATLAB command `step` to analyse the open-loop step response, where we have scaled the input to represent a pitch angle input of 0.26 radians (15 degrees) which is 10% rise in duty cycle PWM for the motor.



System: tf1
I/O: u1 to y1
Rise time (seconds): 19.3

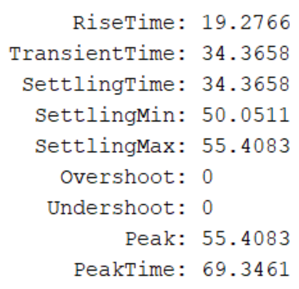
System: tf1
I/O: u1 to y1
Settling time (seconds): 34.4

System: tf1
I/O: u1 to y1
Final value: 55.4

Figure 18. open loop, step response.

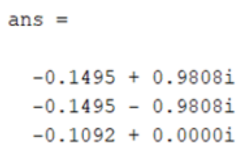
From the above plot, we see that the open-loop response does not satisfy the design criteria (Rise time too large, Settling time as well, Peak time and etc.), and the open-loop response is unstable. The Rise time, peak time, and settling time yield information about the speed of the transient response. This information can help to improve the performance of the system (1.3).

With the Matlab function stepinfo(sys) can we see systems characteristics.



```
       RiseTime: 19.2766
  TransientTime: 34.3658
   SettlingTime: 34.3658
    SettlingMin: 50.0511
    SettlingMax: 55.4083
      Overshoot: 0
     Undershoot: 0
           Peak: 55.4083
       PeakTime: 69.3461
```

Stability of a system can be determined by examining the poles of the transfer function where the poles can be identified using the MATLAB command `pole` as shown below.



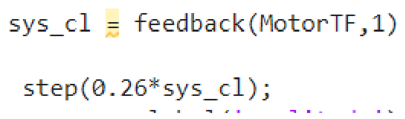
```
ans =

  -0.1495 + 0.9808i
  -0.1495 - 0.9808i
  -0.1092 + 0.0000i
```

As indicated by this function, one of the poles of the open-loop transfer function is on the real axis while the other two poles are in the left-half of the complex *s*-plane. The system has 3 poles and 1 zero (figure 17, pole-zero plot). The underdamped system with an extra pole and a zero under certain condition can be approximated as a second- order system that has just 2 complex dominant poles (1.3, page 155). An extra zero of a response affect the residue, or amplitude, of a response component but do not affect the nature of the response- exponential, damped sinusoid, step or so on (1.3, page 159).

Close loop response.

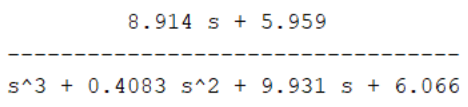
To stabilize this system, we will add a feedback controller. The closed-loop transfer function with the feedback set equal to one can be generated using the MATLAB command `feedback`:



```
sys_cl = feedback(MotorTF,1)

 step(0.26*sys_cl);
```

The response is scaled to model the fact that the pitch angle reference is a 0.26 radian (15 degree, plus 10% to PWM input) step.



```
          8.914 s + 5.959
 ----------------------------------
 s^3 + 0.4083 s^2 + 9.931 s + 6.066
```

The above results demonstrate that the closed-loop transfer function is third order with a zero.

Most of the research in control systems assume a standard underdamped second-order system with no zeros (2.1). Therefore, we cannot rely on these relationships for this system. We can, however, transform the output back to the time domain to generate a time function for the system's response to get some insight into how the poles and zeros of the closed-loop transfer function affect the system's response. Assuming the closed-loop transfer function has the form *Y*(*s*) / *R*(*s*), the output *Y*(*s*) in the Laplace domain is calculated as follows where *R*(*s*) is a step of magnitude 0.26 . We can then perform a partial fraction expansion in order to break this expression into simpler terms that we are able to covert from the Laplace domain back to the time domain. First, we will use the MATLAB command `zpk` to factor the numerator and denominator of our output *Y*(*s*) into simpler terms (2.1).

```
R = 0.26/s;
Y = zpk(sys_cl*R);
```

```
Y =

            2.3176 (s+0.6685)
  -------------------------------------
  s (s+0.6037) (s^2 - 0.1954s + 10.05)

Continuous-time zero/pole/gain model.
```

The denominator of the output *Y*(*s*) can be factored into a first-order term for the real pole of the transfer function, a second-order term for the complex conjugate poles of the transfer function, and a pole at the origin for the step input. Therefore, it is desired to expand *Y*(*s*) as shown below (2.1).

$$Y = \frac{A}{s} + \frac{B}{s + 0.6037} + \frac{Cs + \mathrm{D}}{s^2 - 0.1954\, s + 10.05}$$

The specific values of the constants *A*, *B*, *C*, and *D* can be determined by using the MATLAB command residue to perform the partial fraction expansion. In Matlab the syntax is [r,p,k] = residue(num,den) where num and den are arrays containing the coefficients of the numerator and denominator, respectively, of the Laplace function being expanded.

```
[r,p,k] = residue(0.26*[8.914 5.959],[1 0.4083 9.931 9.931 6.066]);
```

```
r =

  -0.0071 - 0.3673i
  -0.0071 + 0.3673i
   0.0143 + 0.0000i


p =

   0.0977 + 3.1685i
   0.0977 - 3.1685i
  -0.6036 + 0.0000i


k =

     []
```

The coefficients *C* and *D* can be determined by combining the terms for the complex conjugate poles back into a single expression:

```
[num,den] = residue(r(1:2),p(1:2),k);
tf(num,den)
```

```
ans =

   -0.01427 s + 2.329
  ----------------------
  s^2 - 0.1953 s + 10.05
```

Based on the above, *C* = -0.0127 and *D* = 2.329 and our resulting partial fraction expansion can be expressed as follows.

$$Y = \frac{0.0143}{s} - \frac{0.6036}{s + 0.6037} + \frac{-0.01427\, s + 2.329}{s^2 - 0.1953\, s + 10.05}$$

Employing a Laplace transform table, the inverse Laplace transform of the above expression can be taken to generate the corresponding time domain expression. And plot the results. Plot matches the one for LTI viewer.

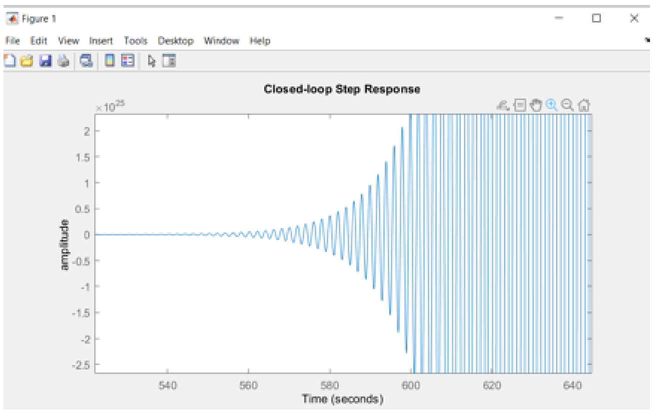
Inverse Laplace brings us back into time domain where we can plot (time, system).

```
syms s
F = (0.0143*1/s) - (0.6036/(s + 0.6037)) + ((-0.01427*s + 2.329)/...
    (s^2 - 0.1953*s + 10.05));

ilaplace(F)
```

$$0.01430000000 - 0.6036000000\, e^{-0.6037000000\, t} + 2.489924650\; 10^{-15}\, e^{0.09765000000\, t} \left(2.950166051\; 10^{14} \sin(3.168669196\, t) - 5.731097124\; 10^{12} \cos(3.168669196\, t)\right)$$

System model with a feedback loop gets even more unstable and the response looks like this:



```
        RiseTime: 220.5582
   TransientTime: 5.9530e+03
    SettlingTime: 5.9535e+03
     SettlingMin: 9.2073e+24
     SettlingMax: 9.5709e+24
       Overshoot: 0
      Undershoot: 90.7628
            Peak: 9.5709e+24
        PeakTime: 5954
```

Figure 19. Close-loop step response.

PID control overview and design requirements.

The output of a PID controller, which is equal to the control input to the plant, is calculated in the time domain from the feedback error as follows (2.2):

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

The variable (e) represents the tracking error, the difference between the desired output (r) and the actual output (y). This error signal (e) is fed to the PID controller, and the controller computes both the derivative and the integral of this error signal with respect to time. The control signal (u) to the plant is equal to the proportional gain (K_p) times the magnitude of the error plus the integral gain (K_i) times the integral of the error plus the derivative gain (K_d) times the derivative of the error. This control signal (u) is fed to the plant and the new output (y) is obtained. The new output (y) is then fed back and compared to the reference to find the new error signal (e). The controller takes this new error signal and computes an update of the control input. The transfer function of a PID controller is found by taking the Laplace transform of Equation (2)

$$K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s}$$ eq.2.

where K_p= proportional gain, K_i= integral gain, and K_d= derivative gain.

Pitch angle control concept.

Our primary goal is to develop an algorithm for controlling the angle of quadcopter wing. The controller will be designed to determine the magnitude of each motor spinning rate to achieve the desired pitch angle. We set a reference angle max and minimum points (15, -15 degrees) based on the study "Measurement of UAV attitude angles based on a single captured image" (2.4). The major purpose of this controller, like any other controller with feedback, is to constantly calculate the error e(t) as the difference between the required value (set point) and the process variable y(t), and to correct the error by adjusting the control signal u(t). The variable- u(t) is in fact the sum of the proportional, integral, and derivative parts. In that way the program will compare the reference angle with the actual pitch angle, calculate the errors and send to the controller which will correct the output. It is important to note here, that our input and output signals are measured due to motors' PWM input and RPM outcome and related pitch angle for certain input.

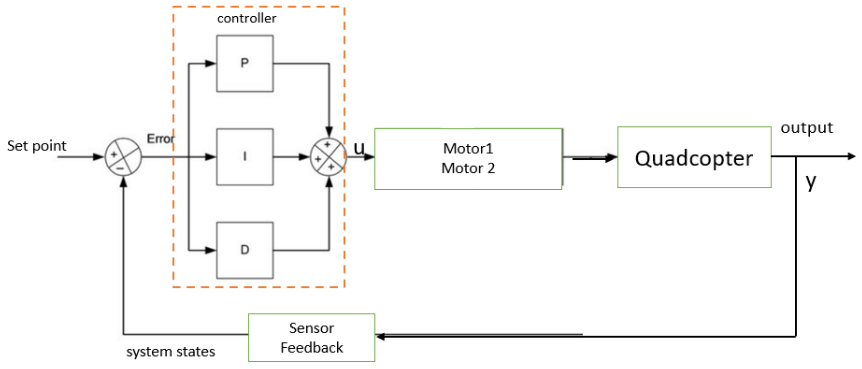


Figure 20. the systems' outline with the PID block.

The basic idea behind this PID controller is to read a sensor, then compute the desired actuator output by calculating proportional, integral, and derivative responses and summing those three components. In the Matlab code the reference pitch angle vector (control vector) is compared with the actual angle (process vector) measured by a IMU sensor. The measurements tables for each signals' value can be seen in Appendix 1.

In summary, the design requirements are the following and suggested by Control Tutorials (source: https://ctms.engin.umich.edu/CTMS/index.php?example=AircraftPitch&section=SystemAnalysis)

- Overshoot less than 10%
- Rise time less than 2 seconds
- Settling time less than 10 seconds
- Steady-state error less than 2% (reference 1)

PID design.

From the main problem, the transfer function for the drone pitch control is:

```
From input "u1" to output "y1":
          8.914 s + 5.959
-----------------------------------
s^3 + 0.4083 s^2 + 1.017 s + 0.1075
```

Where input is motors' PWM signals and output motors' RPM.

The transfer function for the PID controller is:

$$C(s) = K_p + \frac{K_i}{s} + K_d s = \frac{K_d s^2 + K_p s + K_i}{s}$$

We will implement combinations of proportional Kp, integral Ki, and derivative Kd control in order to achieve the desired system behaviour and use automated tuning capabilities of the Control System Designer within MATLAB to design our PID controller for drone pitch control.

The steady state error for the system is calculated by applying MTLAB function file (Appendix 1) GetEss, and is within acceptable frames:

```
e_s_s = 0.01772 ;
```

Proportional control.

We begin by designing a proportional controller of the form C(s)=Kp. The Control System Designer window opens with the root locus plot, open-loop Bode plot, and closed-loop step response plot displayed for the provided plant transfer function with controller C(s)=1 (figure 21).

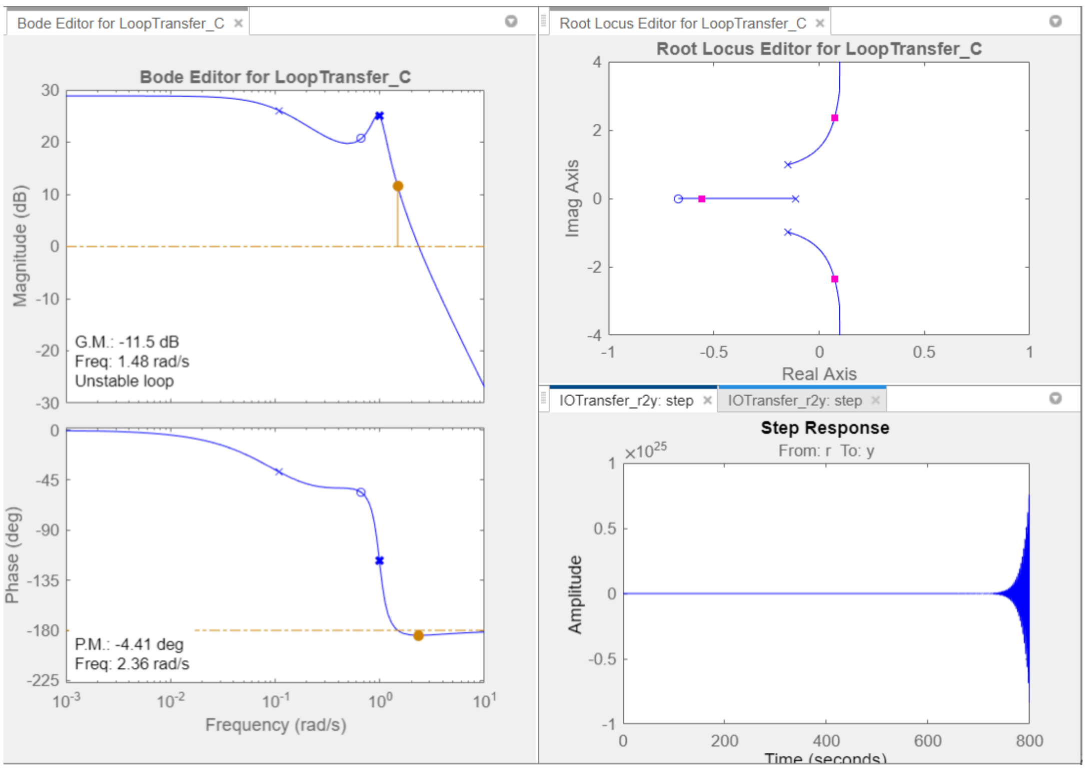
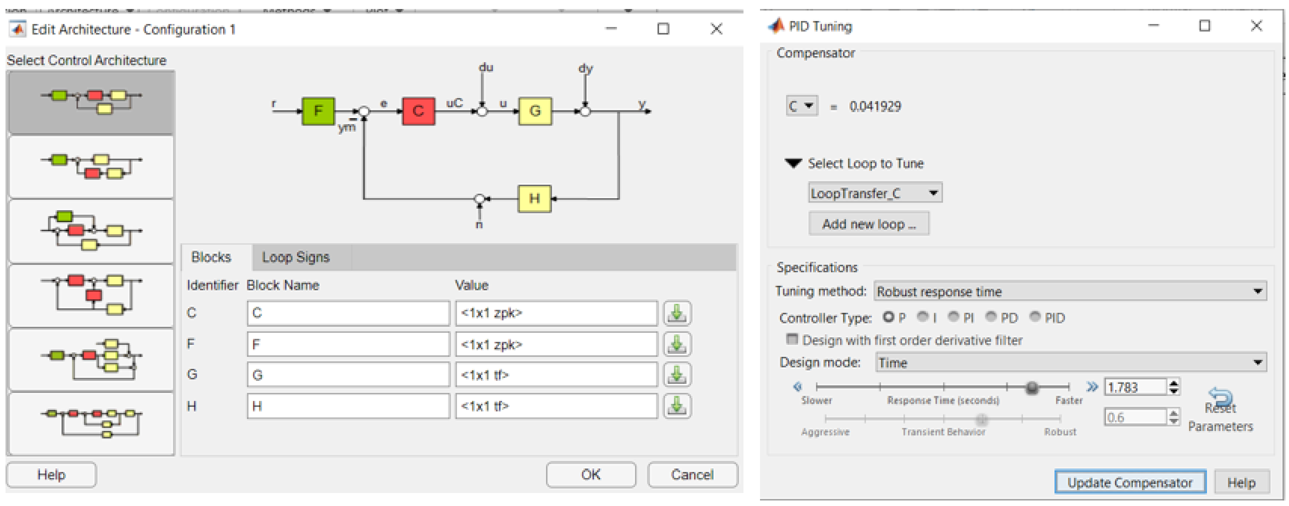


Figure 21. the bode plot, step response, pole-zero, phase response for the system.



Identifier	Block Name	Value
C	C	<1x1 zpk>
F	F	<1x1 zpk>
G	G	<1x1 tf>
H	H	<1x1 tf>

C = 0.041929

LoopTransfer_C

Tuning method: Robust response time

Controller Type: [✓ P] I PI PD PID

Design with first order derivative filter

Design mode: Time

1.783

0.6

Figure 22. The Control System Designer window(left), PID tuning tool (right).

We start with adding a proportional controller Kp to the system and see how the system behaves. Since our reference is a step function scaled to 0.26, we can set the precompensator block F(s) equal to 0.26 to scale a unit step input to our system. Also, adding a precompensator block often helps to reduce the steady state error (1.3). The Kp is set equal to 2. And the compensator C(s) defined from drop-down menu in Compensator editor in Matlab. We set the compensator equal to 2 and generate new plot. The resulting plot can be seen here:

```
C =

  Kp = 0.0419

P-only controller.
```

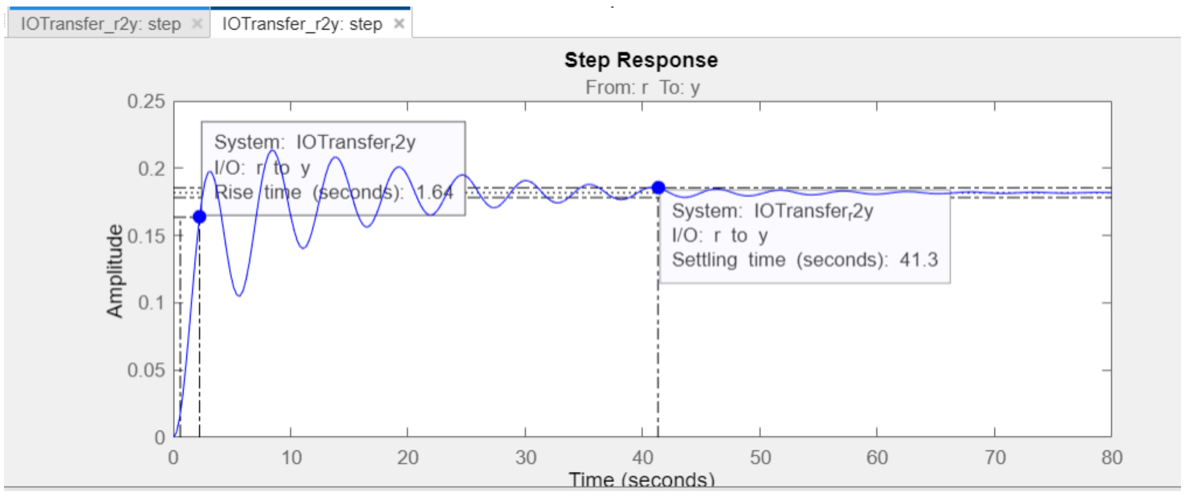


Figure 23. system response with proportional control block.

Examination of the above shows that the given design requirements have not been met. we will use the Control System Designer to automatically tune our proportional compensator Kp with the setting “Robust response time”, the algorithm that automatically tunes the PID parameters to balance speed of response and robustness (2.1). Since, our rise time is expected to be less than 2 seconds, we are specifying a Response Time of 1.7 seconds. The algorithm then chooses a proportional gain of Kp= 0.0419 (figure 22, right).

.

```
        RiseTime: 1.6447
   TransientTime: 41.3115
    SettlingTime: 41.3115
     SettlingMin: 0.4033
     SettlingMax: 0.8211
       Overshoot: 17.4612
      Undershoot: 0
            Peak: 0.8211
        PeakTime: 8.3745
```

Steady state error- 0.5886.

This controller meets the rise time requirement, but the settle time is much too large- 41.3 seconds. When we tried to adjust a response time manually by moving a slider, this resulted in oscillation and increase of overshoot. So, the proportional controller does not provide a sufficient degree of freedom in this tuning, we need to add integral and/or derivative terms to our controller to meet the given requirements.

PI control.

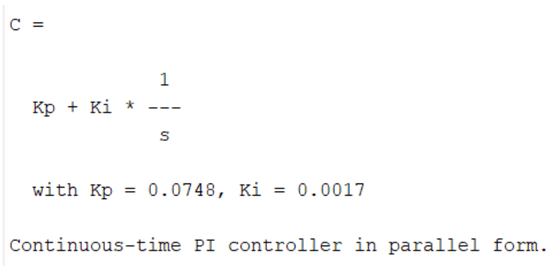
C(s)=Kp+Ki/s

The integral control is often helpful in reducing steady-state error(2.2), the difference between actual and commanded position after the controller has finished applying corrections. Though for the model the SSE is equal to 0.017 % and fits to the requirements.

After applying the PI filter to the tuner, the algorithm gave following values:

$$C = 0.0017487 \times \frac{(1 + 44s)}{s}$$

This transfer function is a PI compensator with Kp=0.0748 and Ki= 0.0017. The resulting closed-loop step response is shown below. The adding of integral control helped reduce the overshoot (2.87%), but it didn't help reduce the oscillation.



```
C =

 
              1
  Kp + Ki * ---
              s

  with Kp = 0.0748, Ki = 0.0017
 
Continuous-time PI controller in parallel form.
```

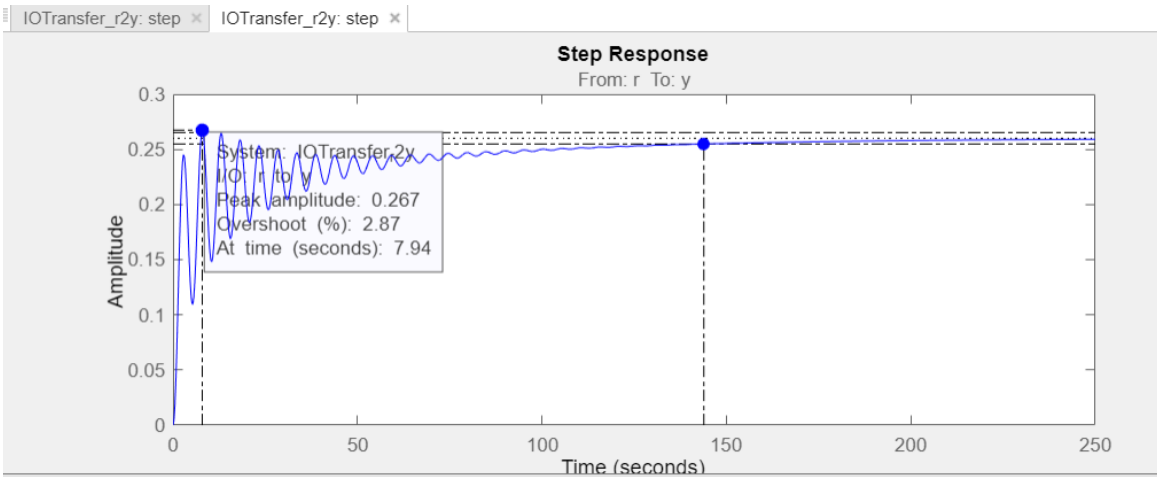


Figure 24. PI controller tuning.

PID control.

C(s)=Kp+Ki/s+ Kd*s

$$C = 0.037871 \times \frac{(1 + 1.3s)\,(1 + 3.7s)}{s}$$

```
              1
  Kp + Ki * --- + Kd * s
              s

  with Kp = 0.189, Ki = 0.0379, Kd = 0.182

Continuous-time PID controller in parallel form.
```

After applying the full PID filter to the system, this transfer function is a PID compensator with Kp=0.1894, Ki= 0.03787, Kd=0.1822. Increasing the derivative gain Kd in a PID controller can often help reduce overshoot (2.1). By adding derivative control, we are able to reduce the oscillation in the response and we can then increase the other gains to reduce the settling time.

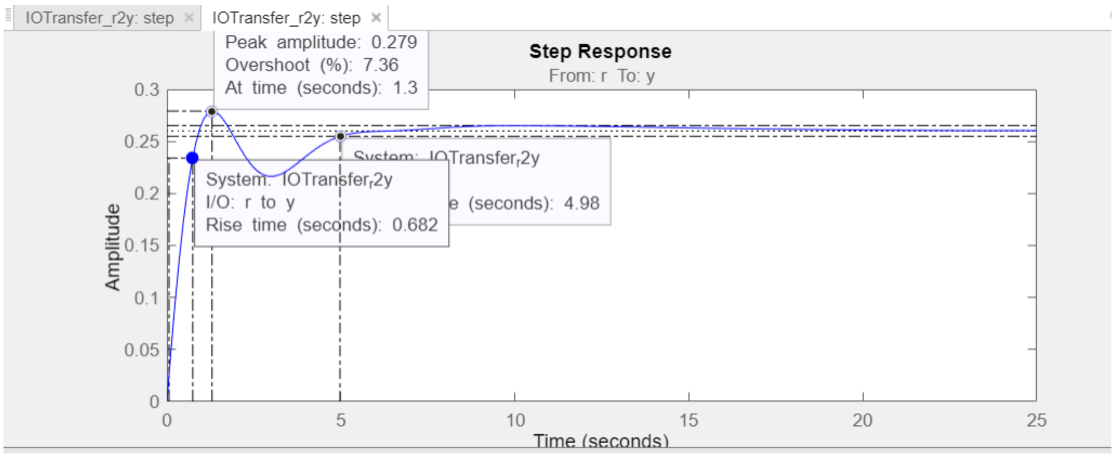


Figure 25. full PID tuner applied.

Rise time: 0.68 seconds, Settling time: 4.98 seconds, Overshoot: 7.36%. the system response meets the requirements, Steady-state error =0.5 % (was calculated separately with matlab function file GetEss, appendix 2).

```
        RiseTime: 0.6845
   TransientTime: 4.9553
    SettlingTime: 4.9553
     SettlingMin: 0.8311
     SettlingMax: 1.0751
       Overshoot: 7.5085
      Undershoot: 0
            Peak: 1.0751
        PeakTime: 1.3159
```

Simulink model.

Uncompensated model and the response simulated in Simulink.

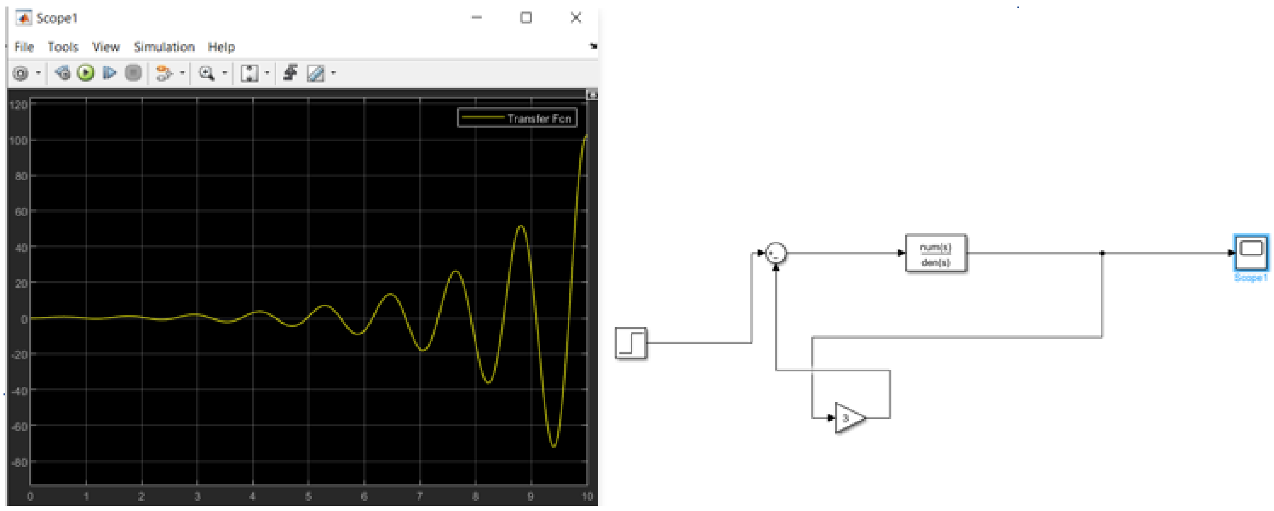
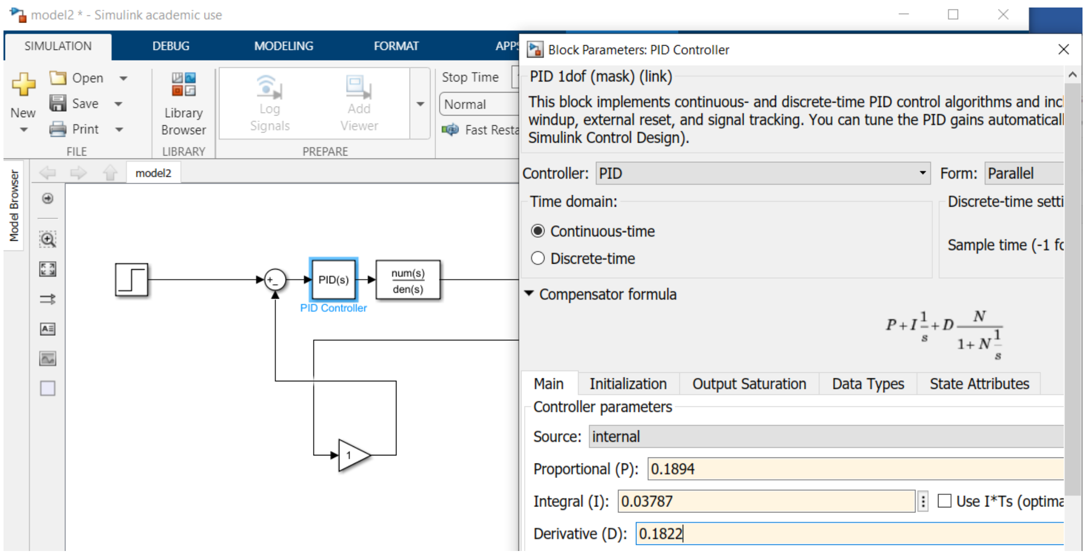


Figure 26. Simulink model, without controller block.

To this model designed in Simulink, we added the PID block and input values for Kp, Ki and Kd obtained before. The accelerometer response is made as a gain K, with element wise multiplication algorithm K*u.



Block Parameters: PID Controller

PID 1dof (mask) (link)

This block implements continuous- and discrete-time PID control algorithms and inc
windup, external reset, and signal tracking. You can tune the PID gains automaticall
Simulink Control Design).

Controller: PID Form: Parallel

Time domain: Discrete-time setti

(●) Continuous-time

( ) Discrete-time

Sample time (-1 fo

Compensator formula

$$P+I\frac{1}{s}+D\frac{N}{1+N\frac{1}{s}}$$

Main | Initialization | Output Saturation | Data Types | State Attributes

Controller parameters

Source: internal

Proportional (P): 0.1894

Integral (I): 0.03787 ☐ Use I*Ts (optima

Derivative (D): 0.1822

Figure 27. with the PID block.

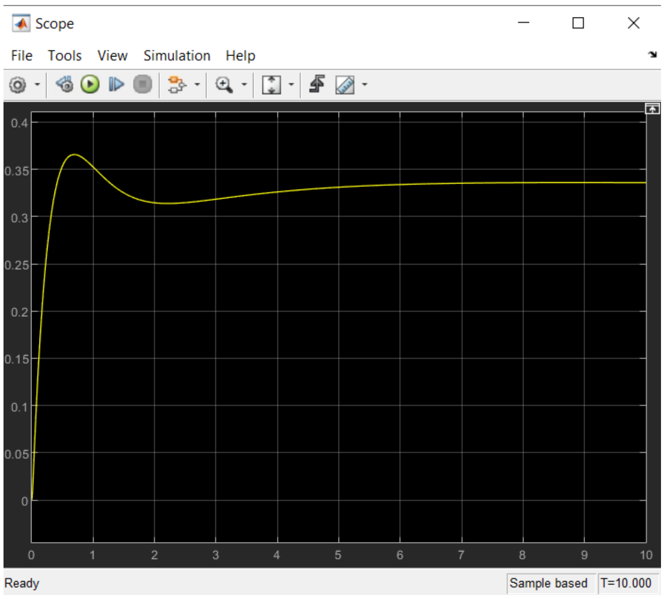


Figure 28. Simulation with the PID block.

The closed-loop response can be seen from the Scope block, that demonstrates that the rise time, settle time, and overshoot requirements are all met. Although the steady state error was not an issue for this model, as it was calculated <2%, the study suggests (1.3) that adding an extra pre-compensator block can help to reduce SSE and smooth the response. The pre-compensator can be implemented by adding a Gain block from the Simulink/Math Operations library to the Simulink model above (between the Step block and the Sum block). As can be seen on the figure below.

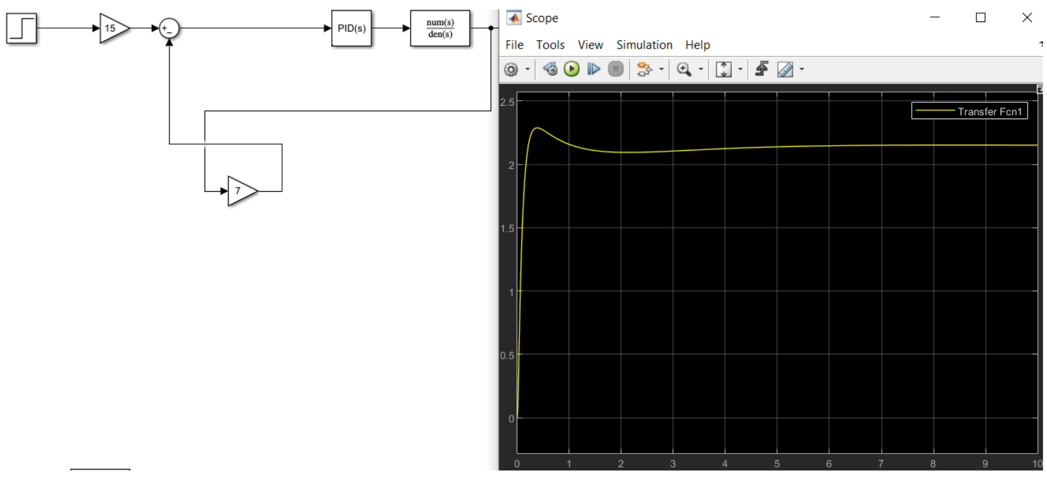


Figure 29. extra gain block added.

Implementation. MATLAB code generator.

The reading from the accelerometer is being converted into pitch angle with the Matlab function ConvertToPitch (figure 31). For the Simulink model we copy the code of the function into empty Matlab function block and include it into our system. The motor duty cycle block generates user defined pulse width, Reference pitch block contained a control vector for the pitch angle, actual pitch block reads sensor measurements. The model from figure 29 is converted into sub- system in figure 30 and extra parts are added.

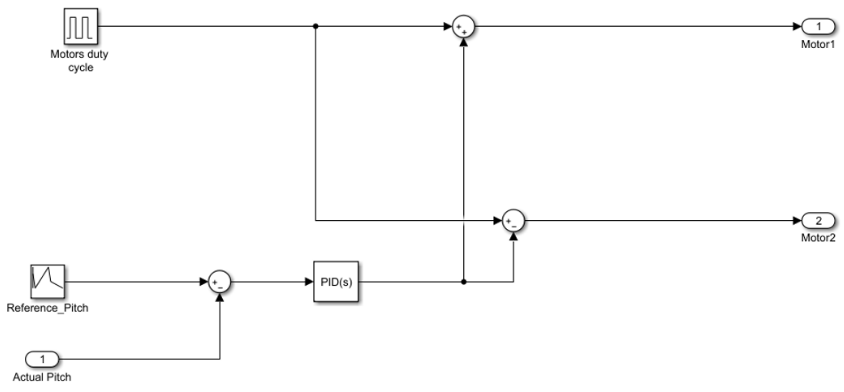


Figure 30. preliminary Simulink model outline. With control.

Converting to sub- system and adding extra blocks for code generator.

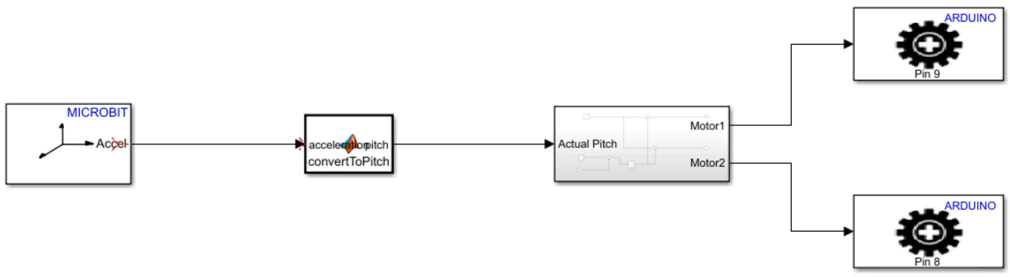


Figure 31. final implementation for Simulink code generator with the sub-system included from figure 29.

So, the Code generator in Simulink builds the model, and generates the code, which can control Arduino-prototype for the drone system. The system is tuned, as we added PID block to it. So, the IMU sensor is used to measure the process variable and provide feedback to the control system. The set point is the desired value for the process variable, such as 15 and -15 degrees. At any given moment, the difference between the process variable and the set point is used by the control system algorithm (compensator), to determine the desired actuator output to drive the system (plant). This process: reading sensor, providing constant feedback and calculating the desired actuator output is repeated continuously.

Conclusion:

During this project work we build a drone wing prototyped, interfaced with Matlab and Simulink, measured systems' behavior and tried to create based on that data the controller to regulate drone's pitch angle. Due to the time limit, we did not manage to run a physical test on the system again with the PID controller algorithm included. The transfer function for the system was obtained using system identification toolbox MATLAB via an experiment. After first analyses in Matlab, the model seemed to have a long settling time and large overshoot, as well as rising time. Therefore, the PID controller was created to improve the performance. Afterwards the PID was tuned and tested in Simulink and the system with the controller showed satisfying results.

References:

References: part 1

1.1. https://www.techtarget.com/iotagenda/definition/drone

1.2.https://botlink.com/blog/2017/12/18/10-terms-and-abbreviations-that-every-drone-enthusiast-should-know

1.3. Norman Nisse Control Systems Engineering, 7th Edition, ISBN: 9781119590132. John Wiley & Sons Inc. 2019.

References, part 2.

2.1. https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID

2.2. https://asat.journals.ekb.eg/article_23032_ed5def262af973aee764d51ec186bf32.pdf
2.3. Stevens, B.L. and F.L. Lewis, Aircraft control and simulation. 2003: John Wiley & Sons.
2.4. https://www.instructables.com/Simple-Motor-Speed-Tester-Tachometer/)
2.5. https://www.dronezon.com/learn-about-drones-quadcopters/how-a-quadcopter-works-with-propellers-and-motors-direction-design-explained/
2.6 https://ctms.engin.umich.edu/CTMS/index.php?example=Introduction&section=ControlPID
2.7. https://se.mathworks.com/help/supportpkg/microbit/ref/mma8652fc3axisaccelerometer.html
2.8. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6111754/

Part 3 references

3.1 https://ctms.engin.umich.edu/CTMS/index.php?example=AircraftPitch&section=SystemModeling
3.2 https://se.mathworks.com/help/slcontrol/ug/system-identification-of-plant-models.html
3.3. https://se.mathworks.com/help/slcontrol/ug/open-loop-response-of-control-system-for-stability-margin-analysis.html
3.4. https://se.mathworks.com/help/ident/gs/about-system-identification.html
3.5 https://ctms.engin.umich.edu/CTMS/index.php?example=AircraftPitch&section=SystemAnalysis

Appendix 1 experimental measurements. MATLAB codes.

Due to large volume the measurement data is zipped and attached as separate Excel files for pitch angle measurements and rpm speed measurements.

Also, MATLAB and Simulink files are zipped and attached to the report in a separate folder.