

Control systems

Lab Project

Modeling and Control of an indoor quadcopter

By

Mohan S P

Kenneth Joel



Department of Electrical and Electronics Engineering

PES University

Bengaluru-560085

Table of Contents

I	Literature review	3
1	Objective	3
2	Introduction	3
3	Methodology	3
4	Assumptions in our model	4
4.1	Plus and cross configuration of the quadcopter	4
4.2	Body axes system	5
II	Mathematical Model	5
5	Motor Equations	5
6	Drag Forces	7
7	Torques	7
8	Euler's Equation for Rotational Motion	8
9	State Space Model	8
III	Simulation	10
10	Attitude command	11
11	Control mixing	12
12	Quadcopter dynamics	12
12.1	S-function code	13
13	Loadable files	18
14	Sample output	18
14.1	OPEN PLOT: State Data	18
14.2	OPEN GUI: Flight Animation	19
IV	Hardware	19
15	Motor characterization	20
16	Inertial Measurement Unit (MPU6050)	21

17 Driver circuit	22
17.1 Justification for the components	23
18 Test rig	24
 V Control	 25
19 Control considerations	25
20 Mathematical Implementation	26
21 Practical implementation	27
21.1 Arduino Code	27
 VI Result	 35
 VII Conclusion	 35
22 Learning experience	35
23 Failure log and future scope	35

Part I

Literature review

1 Objective

To stabilize and control attitude (roll, pitch, yaw) and altitude of a quadcopter.

2 Introduction

A quadrotor helicopter (quadcopter) is a helicopter which has four equally spaced rotors. With four independent rotors, the need for a swash-plate mechanism is alleviated. The swash-plate mechanism was needed to allow the helicopter to utilize more degrees of freedom, but the same level of control can be obtained by adding two more rotors. The development of quadcopters has stalled until very recently, because controlling four independent rotors has proven to be incredibly difficult and impossible without electronic assistance. The decreasing cost of modern microprocessors has made electronic and even completely autonomous control of quadcopters feasible for commercial, military, and even hobbyist purposes. Quadcopter control is a fundamentally difficult and interesting problem. With 6 degrees of freedom (3 translational and 3 rotational) and only 4 independent inputs (rotor speeds), quadcopters are severely under-actuated. In order to achieve 6 degrees of freedom, rotational and translational motion are coupled. The resulting dynamics are highly nonlinear, especially after accounting for the complicated aerodynamic effects. Through this project we intend to implement a simplified controller for the quadcopter that should be capable of orientation stabilization and then orientation control which extends to linear motion as a result of the same.

3 Methodology

- Literature survey
- Obtain mathematical model of the drone considering process variables as roll, pitch and yaw.
- Characterize the core-less DC Motors and design low form factor power circuit accordingly.
- Design control circuit with specified micro-controller and implement the PID algorithm for the same.

4 Assumptions in our model

4.1 Plus and cross configuration of the quadcopter

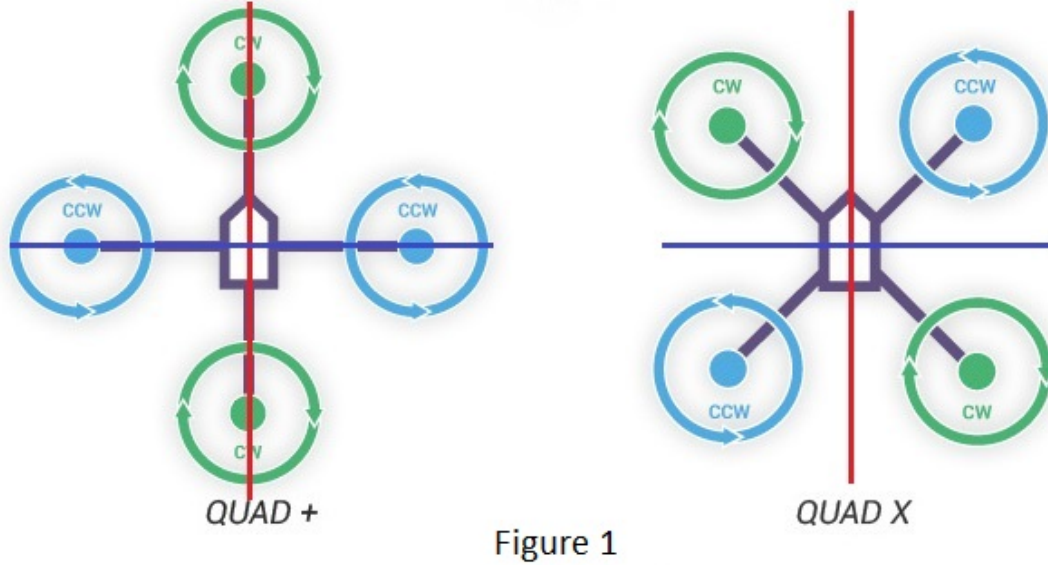


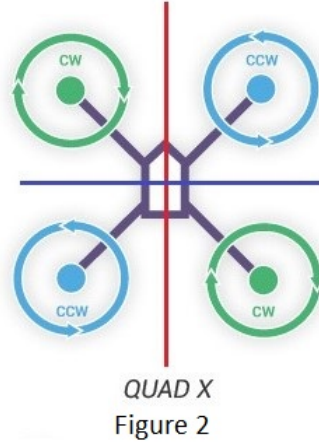
Figure 1

The first step to model a quadcopter would be to decide on a configuration. There are 2 known configurations of quadcopter as shown in Figure 1. Each with their advantages and disadvantages as follows:

- Rotational Moment: The rotational forces about roll and pitch axes of rotation for a + system would be dependent on the thrust from only 1 motor. In the case of the X system the rotational forces would be dependent on any two motors, but the perpendicular distance from the axis of rotation is decreased by a factor of $\cos(\pi/4)$ in the case of a perfectly square arrangement of motors.
- Camera arc clearance and visibility of yaw: These factors do not affect the modeling of the quadrotor but are significant considerations as they affect the user's convenience.

However, there is no common consensus on whether either one of the two models are superior in an absolute sense from a practical standpoint. In terms of modelling the quadcopter there are not too many significant changes to be made as all changes can be accounted for in the thrust equations. The detailed differences between plus and cross configurations can be found in[1]

4.2 Body axes system



The body axis system we consider for this project is shown in Figure 2 and is based on the cross configuration of the quadcopter. Also, the body axis center is assumed to be at the same position as the center of gravity.

Anticlockwise rotations are taken to be positive, the specifications for each axis and the displacements they correspond to are as follows:

Positive Pitch: anticlockwise around X axis \rightarrow positive displacement in Y direction

Positive Roll: anticlockwise around Y axis \rightarrow negative displacement in X direction

Positive Yaw: anticlockwise around Z axis \rightarrow no net displacement

Increase in overall thrust \rightarrow Positive displacement in Z direction

Part II

Mathematical Model

5 Motor Equations

$$\tau = k_t(I - I_o) \quad (1)$$

τ : Torque

k_t : Torque proportionality constant

I : Current through the motor

I_o : No load current

$$V = IR_m + k_v\omega \quad (2)$$

V : Voltage Drop

R_m : Motor resistance

k_v : Back-emf/rpm proportionality constant

$$P = \frac{(\tau R_m + k_t I_o R_m + k_v k_t \omega)(\tau + k_t I_o)}{k_t^2} \quad (3)$$

Approximating this equation by neglecting R_m and considering $k_t I_o \ll \tau$ we get

$$P = \frac{k_v \tau \omega}{k_t} \quad (4)$$

By conservation of energy, we know that the energy the motor expends in a given time period is equal to the force generated on the propeller times the distance that the air it displaces moves. Therefore

$$P \cdot dt = T \cdot dx \quad (5)$$

$$P = T v_h \quad (6)$$

$$\frac{k_v \tau \omega}{k_t} = T v_h \quad (7)$$

v_h : Hover Velocity

From Aerodynamics we have

$$v_h = \sqrt{\frac{T}{2\rho A}} \quad (8)$$

Using this we can rewrite the power equation as

$$P = \frac{T^{3/2}}{\sqrt{2\rho A}} \quad (9)$$

T : Thrust

ρ : Air Density

A : Area swept by the propeller

We know that torque is proportional to thrust. Therefore

$$\tau = k_\tau T \quad (10)$$

k_τ : Thrust proportionality constant

Note: For our project we consider uniform density of air ρ since the drone flies within a limited space.

Substituting (10) in (4) and equating it to (9) we get

$$T = \left(\frac{\omega k_v k_\tau \sqrt{2\rho A}}{k_t} \right)^2 \quad (11)$$

The only variable is ω . Therefore (11) becomes

$$T = k\omega^2 \quad (12)$$

Since there are 4 motors equation (12) becomes

$$T_B = \begin{bmatrix} 0 \\ 0 \\ k(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \end{bmatrix} \quad (13)$$

6 Drag Forces

The matrix below describes drag forces in 3 directions

$$F_d = \begin{bmatrix} -k_d \dot{x} \\ -k_d \dot{y} \\ -k_d \dot{z} \end{bmatrix} \quad (14)$$

k_d : Drag co-efficient

\dot{x} , \dot{y} , \dot{z} : Linear velocities in x , y , z directions respectively.

From fluid dynamics we have

$$F_D = \frac{\rho C_D A v^2}{2} \quad (15)$$

Therefore τ_D can be written as

$$\tau_D = \frac{\rho R^3 C_D A \omega^2}{2} \quad (16)$$

The only variable is ω . Therefore (16) becomes

$$\tau_D = b \omega^2 \quad (17)$$

C_D : Drag Constant

v : Linear Velocity

7 Torques

According to our model the roll, pitch and yaw torques can be represented as

$$\begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} Lk\sqrt{2}(\omega_2^2 + \omega_3^2 - \omega_1^2 - \omega_4^2) \\ Lk\sqrt{2}(\omega_3^2 + \omega_4^2 - \omega_1^2 - \omega_2^2) \\ b(\omega_1^2 - \omega_2^2 + \omega_3^2 - \omega_4^2) \end{bmatrix} = \tau_B \quad (18)$$

L : Boom length

Expressing linear motion in the inertial frame we get

$$\ddot{a} = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \left[\frac{RT_B}{m} \right] + \left[\frac{F_d}{m} \right] \quad (19)$$

$$\ddot{\mathbf{a}} = \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} \quad (20)$$

$$R = \begin{bmatrix} c_\theta c_\psi & c_\theta s_\psi & -s_\theta \\ s_\phi s_\theta c_\psi - c_\phi s_\psi & c_\phi c_\psi + s_\phi s_\theta s_\psi & c_\theta s_\phi \\ s_\phi s_\psi + c_\phi s_\theta c_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi & c_\phi c_\theta \end{bmatrix} \quad (21)$$

8 Euler's Equation for Rotational Motion

Euler expressed rotational motion in the body frame using the following equation[2]

$$\tau_B = I\dot{\omega} + (\omega \times I\omega) \quad (22)$$

I : Moment of Inertia

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \quad (23)$$

$$\omega = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \quad (24)$$

From (22) we get

$$\dot{\omega} = I^{-1}(\tau - (\omega \times (I\omega))) \quad (25)$$

Therefore by substituting (18) (23) and (24) in (25) we get

$$\dot{\omega} = \begin{bmatrix} \tau_\phi I_{xx}^{-1} - \frac{(I_{yy} - I_{zz})\omega_y \omega_z}{I_{xx}} \\ \tau_\theta I_{yy}^{-1} - \frac{(I_{zz} - I_{xx})\omega_z \omega_x}{I_{yy}} \\ \tau_\psi I_{zz}^{-1} - \frac{(I_{xx} - I_{yy})\omega_x \omega_y}{I_{zz}} \end{bmatrix} \quad (26)$$

9 State Space Model

To describe a quadcopter in space we need the following variables in all 3 axes:

- Position
- Linear velocities
- Orientation
- Angular velocities

This comes upto 12 variables and can be represented as follows

$$\begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \phi \\ \theta \\ \psi \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \quad (27)$$

We can assume 4 vectors x_1 x_2 x_3 x_4 such that

$$x_1 = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (28)$$

$$x_2 = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \quad (29)$$

$$x_3 = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} \quad (30)$$

$$x_4 = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} \quad (31)$$

$$\dot{x}_4 = x_2 \quad (32)$$

$$\dot{x}_2 = \begin{bmatrix} 0 \\ 0 \\ -g \end{bmatrix} + \frac{RT_B}{m} + F_d \quad (33)$$

$$\dot{x}_3 = \begin{bmatrix} 1 & -t_\theta s_\phi & -c_\phi t_\theta \\ 0 & c_\phi & -s_\phi \\ 0 & \frac{s_\phi}{c_\theta} & \frac{c_\phi}{s_\theta} \end{bmatrix} x_4 \quad (34)$$

$$\dot{x}_4 = \begin{bmatrix} \tau_\phi I_{xx}^{-1} - \frac{(I_{yy} - I_{zz})\omega_y \omega_z}{I_{xx}} \\ \tau_\theta I_{yy}^{-1} - \frac{(I_{zz} - I_{xx})\omega_z \omega_x}{I_{yy}} \\ \tau_\psi I_{zz}^{-1} - \frac{(I_{xx} - I_{yy})\omega_x \omega_y}{I_{zz}} \end{bmatrix} \quad (35)$$

Part III

Simulation

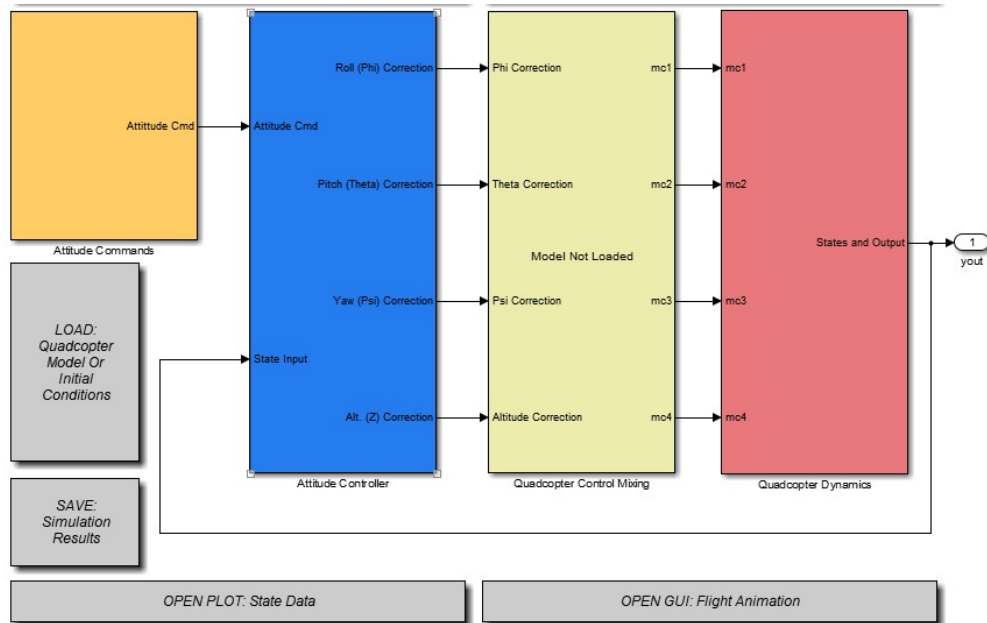
While surveying potential options for simulation of the quadrotor system as a whole the following were considered:

- Using asbQuadCopter Simulink library.
- Importing a 3D CAD model of the quadcopter from solidworks into Simulink as an XML File using sim-import.
- Developing a basic simulation from ground up.
- Adopting some open source models to suit our needs.

We chose to adopt an open source model designed by MEM students of Drexell university to suit our requirements for this project. Doing this we gained significant exposure to tools that we were unfamiliar with as well as a good sense of structure and modularity.

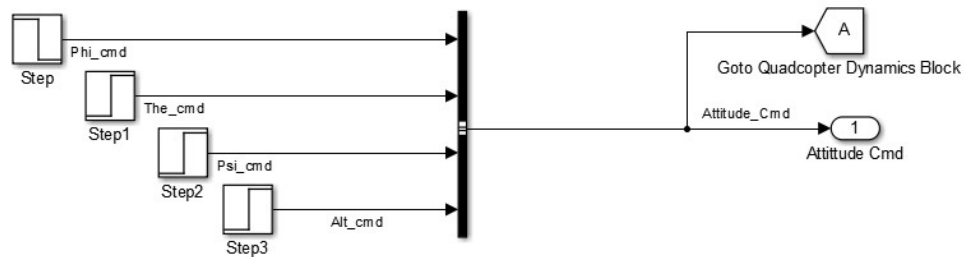
However, the other techniques listed are also quite useful and are adopted for several commercial products and should be considered as future improvements.

The figure below shows the model we used as displayed in simulink containing the four main masked subsystems and four action buttons. Their roles are as labeled in the image.

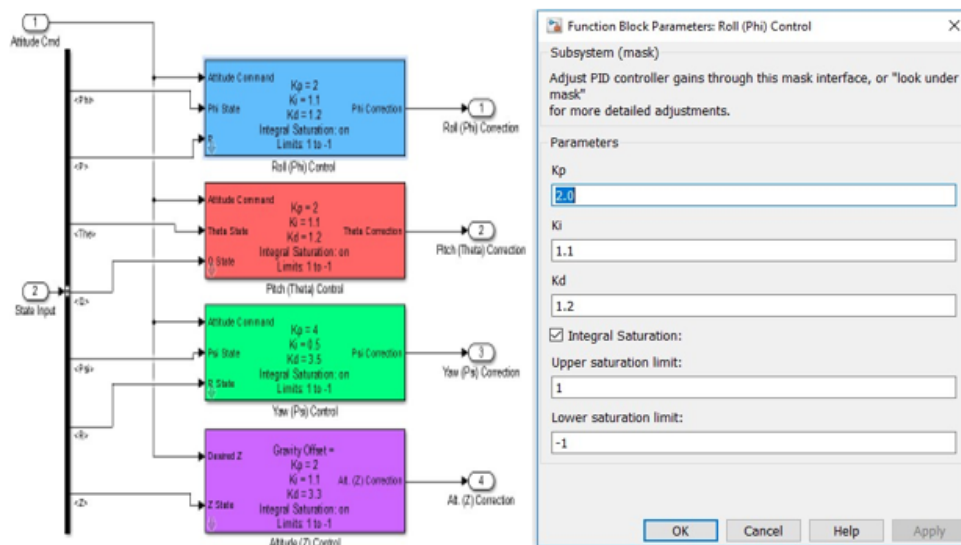


10 Attitude command

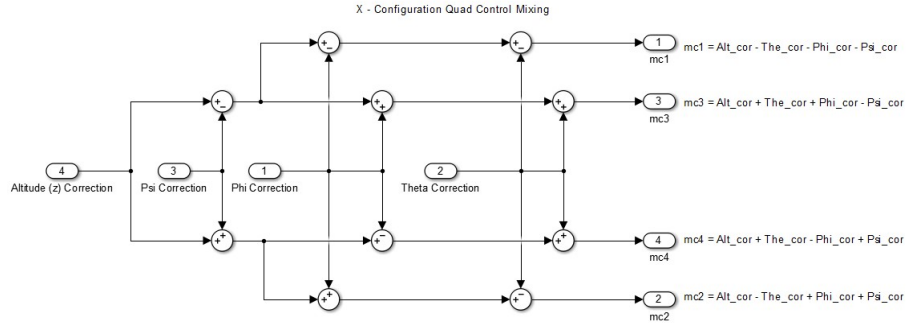
Expanding the first subsystem (yellow) reveals the model as shown in the figure above which represents the inputs specified by the user given by corresponding step inputs for roll pitch yaw and altitude.



Expanding the second subsystem (blue) reveals the model shown in the figure above which comprises of four PID controllers working in parallel to issue correction commands for each corresponding action to the next block. The controllers are implemented as masked function blocks with parameters that can be defined as shown in the figure below.

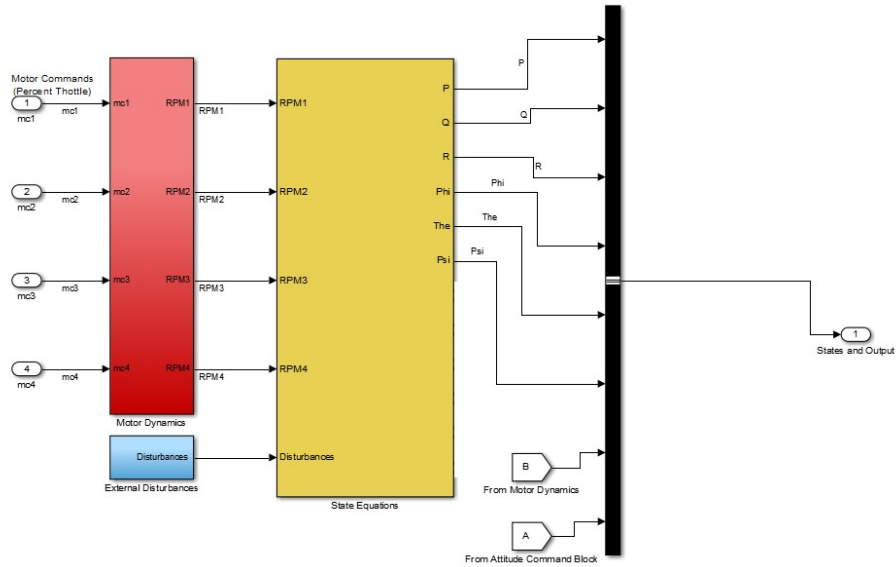


11 Control mixing



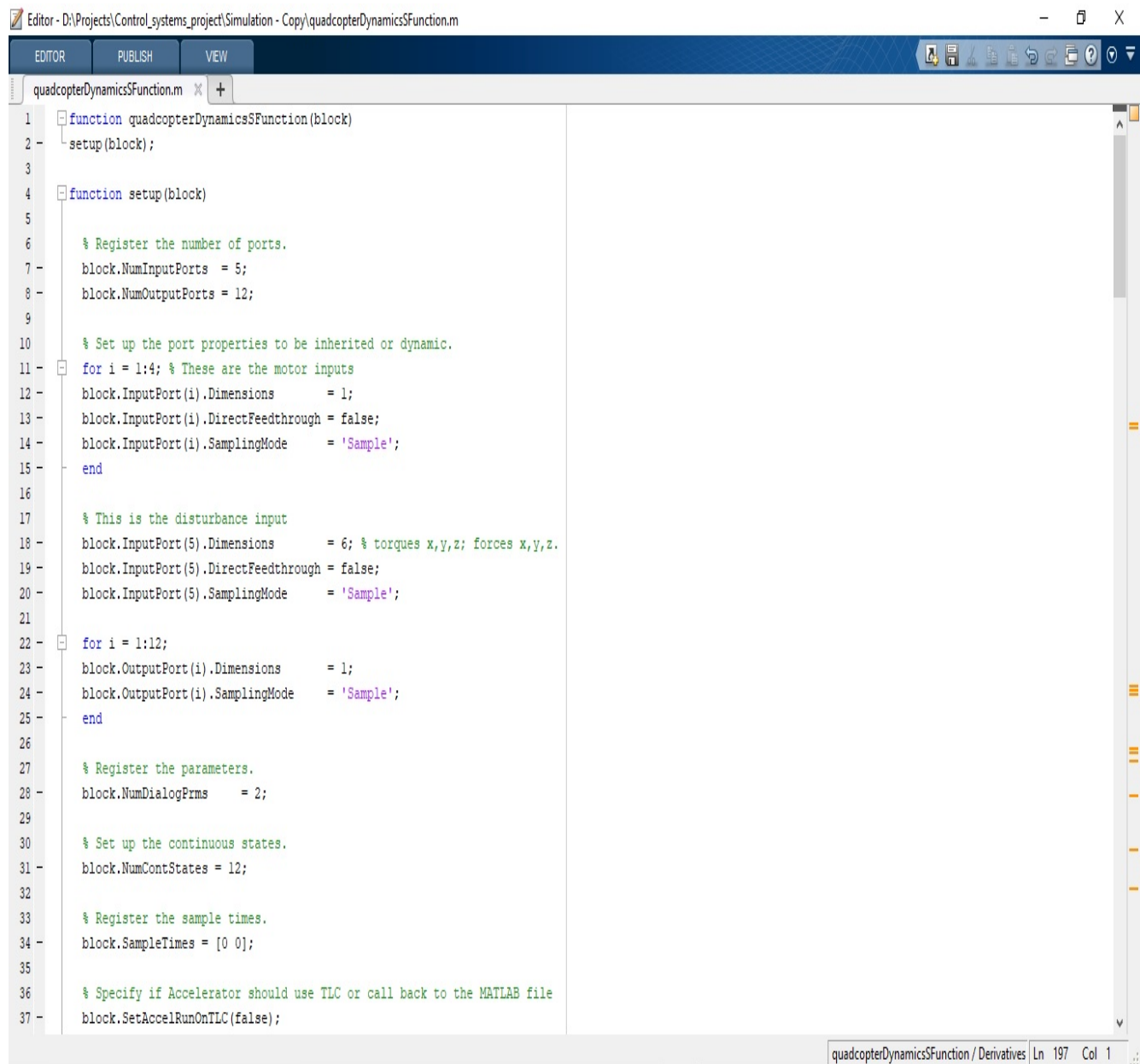
Expanding the third subsystem (blue) reveals the model shown in the figure above which as labeled is the motor control mixing based on correction commands given by the controller for a cross configured quadrotor.

12 Quadcopter dynamics



Expanding the third subsystem (blue) reveals the model shown in the figure above. The red block represents the motor dynamics and the yellow block is defined by a level 2 S-function which implements the mathematical model of the system. We have used the mathematical model we derived in Part II Section 9 in this S function. However, we have not considered states involving position and velocity since for this project we are primarily concerned with the orientation of the quadcopter. Majority of the modifications made were in this section of the model.

12.1 S-function code



```
Editor - D:\Projects\Control_systems_project\Simulation - Copy\quadcopterDynamicsSFunction.m
EDITOR PUBLISH VIEW
quadcopterDynamicsSFunction.m
1 function quadcopterDynamicsSFunction(block)
2     setup(block);
3
4 function setup(block)
5
6     % Register the number of ports.
7     block.NumInputPorts = 5;
8     block.NumOutputPorts = 12;
9
10    % Set up the port properties to be inherited or dynamic.
11    for i = 1:4; % These are the motor inputs
12        block.InputPort(i).Dimensions = 1;
13        block.InputPort(i).DirectFeedthrough = false;
14        block.InputPort(i).SamplingMode = 'Sample';
15    end
16
17    % This is the disturbance input
18    block.InputPort(5).Dimensions = 6; % torques x,y,z; forces x,y,z.
19    block.InputPort(5).DirectFeedthrough = false;
20    block.InputPort(5).SamplingMode = 'Sample';
21
22    for i = 1:12;
23        block.OutputPort(i).Dimensions = 1;
24        block.OutputPort(i).SamplingMode = 'Sample';
25    end
26
27    % Register the parameters.
28    block.NumDialogPrms = 2;
29
30    % Set up the continuous states.
31    block.NumContStates = 12;
32
33    % Register the sample times.
34    block.SampleTimes = [0 0];
35
36    % Specify if Accelerator should use TLC or call back to the MATLAB file
37    block.SetAccelRunOnTLC(false);
quadcopterDynamicsFunction / Derivatives Ln 197 Col 1
```

Editor - D:\Projects\Control_systems_project\Simulation - Copy\quadcopterDynamicsFunction.m

EDITOR PUBLISH VIEW

quadcopterDynamicsFunction.m

```

38 % Specify the block simStateCompliance. The allowed values are:
39 % 'UnknownSimState', < The default setting; warn and assume DefaultSimState
40 % 'DefaultSimState', < Same SimState as a built-in block
41 % 'HasNoSimState', < No SimState
42 % 'CustomSimState', < Has GetSimState and SetSimState methods
43 % 'DisallowSimState' < Errors out when saving or restoring the SimState
44 block.SimStateCompliance = 'DefaultSimState';
45
46 % CheckParameters:
47 %   Functionality : Called in order to allow validation of the
48 %                   block dialog parameters. You are
49 %                   responsible for calling this method
50 %                   explicitly at the start of the setup method.
51 block.RegBlockMethod('CheckParameters', @CheckPrms);
52
53 % InitializeConditions:
54 %   Functionality : Called in order to initialize the
55 %                   conditions of the quad.
56 block.RegBlockMethod('InitializeConditions', @InitializeConditions);
57
58 % Outputs:
59 %   Functionality : Call to generate the block outputs during a
60 %                   simulation step.
61 block.RegBlockMethod('Outputs', @Outputs);
62
63 % Derivatives:
64 %   Functionality : Call to update the derivatives of the
65 %                   continuous states during a simulation step.
66 %   C-Mex counterpart: mdlDerivatives
67 block.RegBlockMethod('Derivatives', @Derivatives);
68
69 function CheckPrms(block)
70     quad = block.DialogPrm(1).Data;
71     IC   = block.DialogPrm(2).Data;
72
73 function InitializeConditions(block)
74     % Initialize 12 States

```

quadcopterDynamicsFunction / Derivatives Ln 197 Col 1

```
Editor - D:\Projects\Control_systems_project\Simulation - Copy\quadcopterDynamicsFunction.m
EDITOR PUBLISH VIEW
quadcopterDynamicsFunction.m
75 IC = block.DialogPrm(2).Data;
76
77 % IC.P, IC.Q, IC.R converted to rad/s
78 P = IC.P*pi/180; Q = IC.Q*pi/180; R = IC.R*pi/180;
79
80 % IC.Phi, IC.The, IC.Psi converted to rads
81 Phi = IC.Phi*pi/180; The = IC.The*pi/180; Psi = IC.Psi*pi/180;
82
83 U = IC.U; V = IC.V; W = IC.W;
84 X = IC.X; Y = IC.Y; Z = IC.Z;
85
86 init = [P,Q,R,Phi,The,Psi,U,V,W,X,Y,Z];
87
88 for i=1:12
89     block.OutputPort(i).Data = init(i);
90     block.ContStates.Data(i) = init(i);
91 end
92
93 function Outputs(block)
94 for i = 1:12;
95     block.OutputPort(i).Data = block.ContStates.Data(i);
96 end
97
98 function Derivatives(block)
99
100 % Load model data selected in parameter block
101 quad = block.DialogPrm(1).Data;
102
103 % P Q R in units of rad/sec
104 P = block.ContStates.Data(1);
105 Q = block.ContStates.Data(2);
106 R = block.ContStates.Data(3);
107
108 % Phi The Psi in radians
109 Phi = block.ContStates.Data(4);
110 The = block.ContStates.Data(5);
111 Psi = block.ContStates.Data(6);
```

quadcopterDynamicsFunction / Derivatives Ln 197 Col 1


```

Editor - D:\Projects\Control_systems_project\Simulation - Copy\quadcopterDynamicsFunction.m
EDITOR PUBLISH VIEW
quadcopterDynamicsFunction.m
112 % U V W in units of m/s
113 %Uncomment for position control and team_37 control
114 %U = block.ContStates.Data(7);
115 %V = block.ContStates.Data(8);
116 %W = block.ContStates.Data(9);
117
118 %Set to 0 because we are looking at only orientation of the quad
119 %Comment if not using attitude control
120 U = 0;
121 V = 0;
122 W = 0;
123
124 % X Y Z in units of m
125 %Uncomment for position control and team_37 control
126 %X = block.ContStates.Data(10);
127 %Y = block.ContStates.Data(11);
128 %Z = block.ContStates.Data(12);
129
130 %Set to 0 because we are looking at only orientation of the quad
131 %Comment if not using attitude control
132 X = 0;
133 Y = 0;
134 Z = 0;
135
136 % w values in rev/min
137 w1 = block.InputPort(1).Data;
138 w2 = block.InputPort(2).Data;
139 w3 = block.InputPort(3).Data;
140 w4 = block.InputPort(4).Data;
141 w = [w1; w2; w3; w4];
142
143 Dist_tau = block.InputPort(5).Data(1:3);
144 Dist_F = block.InputPort(5).Data(4:6);
145
146 % Calculate moment and thrust forces
147 tau_motorGyro = [Q*quad.Jm*2*pi/60*(-w1-w3+w2+w4); P*quad.Jm*2*pi/60*(w1+w3-w2-w4); 0]; % Note: 2*pi/60 required to convert from RPM to radians/s
148 Mb = (quad.dctcq*(w.^2))+ tau_motorGyro + (Dist_tau); % Mb = [taul tau2 tau3]'
149
quadcopterDynamicsFunction / Derivatives Ln 197 Col 1

```

```

Editor - D:\Projects\Control_systems_project\Simulation - Copy\quadcopterDynamicsFunction.m
EDITOR PUBLISH VIEW
quadcopterDynamicsFunction.m
149 % Thrust due to motor speed
150 Fb = [0; 0; sum(quad.ct*(w.^2))];
151 % Obtain dP dQ dR
152 omb_bi = [P; Q; R];
153 Omb_bi = [ 0, -R, Q; R, 0, -P; -Q, P, 0];
154
155 b_omdotb_bi = quad.Jbinv*(Mb-(cross(omb_bi, (quad.Jb*omb_bi))));
156 H_Phi = [1, tan(The)*sin(Phi), tan(The)*cos(Phi); 0, cos(Phi), -sin(Phi); 0, sin(Phi)/cos(The), cos(Phi)/cos(The)];
157 Phidot = H_Phi*omb_bi;
158
159 % We use a Z-Y-X rotation matrix
160 Rib = [cos(Psi)*cos(The) cos(Psi)*sin(The)*sin(Phi)-sin(Psi)*cos(Phi) cos(Psi)*sin(The)*cos(Phi)+sin(Psi)*sin(Phi);
161 sin(Psi)*cos(The) sin(Psi)*sin(The)*sin(Phi)+cos(Psi)*cos(Phi) sin(Psi)*sin(The)*cos(Phi)-cos(Psi)*sin(Phi);
162 -sin(The) cos(The)*sin(Phi) cos(The)*cos(Phi)];
163 Rbi = Rib';
164 ge = [0; 0; -quad.g];
165 gb = Rbi*ge;
166 Dist_Fb = Rbi*Dist_F;
167
168 % Compute Velocity and Position derivatives of body frame
169 vb = [U;V;W];
170 % Acceleration in body frame (For velocity)
171 b_dv = (1/quad.mass)*Fb+gb+Dist_Fb-Omb_bi*vb;
172 % Units OK SI: Velocity of body frame w.r.t inertial frame (FOR POSITION)
173 i_dp = Rib*vb;
174 dP = b_omdotb_bi(1);
175 dQ = b_omdotb_bi(2);
176 dR = b_omdotb_bi(3);
177 dPhi = Phidot(1);
178 dTheta = Phidot(2);
179 dPsi = Phidot(3);
180 dU = b_dv(1);
181 dV = b_dv(2);
182 dW = b_dv(3);
183 dX = i_dp(1);
184 dY = i_dp(2);
185 dZ = i_dp(3);
...
quadcopterDynamicsFunction / Derivatives Ln 168 Col 1

```

13 Loadable files

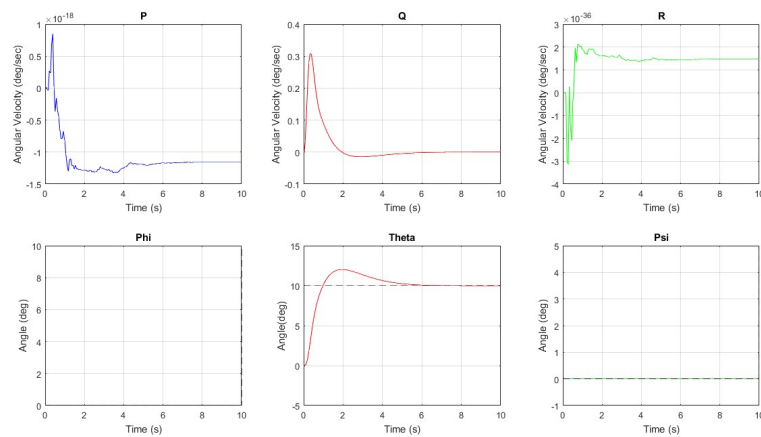
Name	Date modified	Type
All_zero.mat	4/2/2018 2:37 PM	Microsoft Access ...
Dynamics Test.mat	7/13/2014 7:02 PM	Microsoft Access ...
Hover.mat	7/13/2014 7:02 PM	Microsoft Access ...
IC.mat	7/13/2014 7:02 PM	Microsoft Access ...
IC_45yawonground.mat	7/13/2014 7:02 PM	Microsoft Access ...
IC_attErrors.mat	7/13/2014 7:02 PM	Microsoft Access ...
IC_attErrors_w4000.mat	7/13/2014 7:02 PM	Microsoft Access ...
IC_attMoreErrorsYaw_w4000.mat	7/13/2014 7:02 PM	Microsoft Access ...
IC_HoverAt10ft.mat	7/13/2014 7:02 PM	Microsoft Access ...
IC onground w4000.mat	7/13/2014 7:02 PM	Microsoft Access ...
quadModel_+.mat	7/13/2014 7:02 PM	Microsoft Access ...
quadModel_X.mat	7/13/2014 7:02 PM	Microsoft Access ...

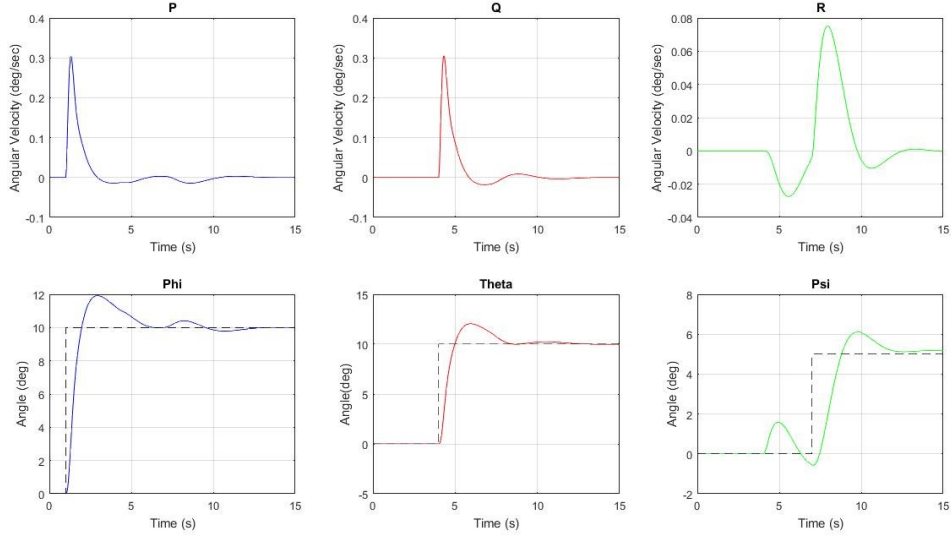
As seen in the left half of the above figure the simulation allows us to load predefined configuration files or initial condition files to the model. These files are used either to define a quadcopter model or initialize the state variables to some initial conditions before the execution of the simulation. For our purposes we use only `quadModel_X.mat` since we have modeled a cross configured quadcopter. Initial conditions were defined as per our needs in the simulation, note that even here we define initial conditions that affect the orientation of the quadcopter and nothing that affects translation or absolute position as that is out of the scope of this project.

14 Sample output

Action Buttons ‘OPEN PLOT: State Data’ and ‘OPEN GUI: Flight Animation’ give us the plots and 3D animation respectively.

14.1 OPEN PLOT: State Data

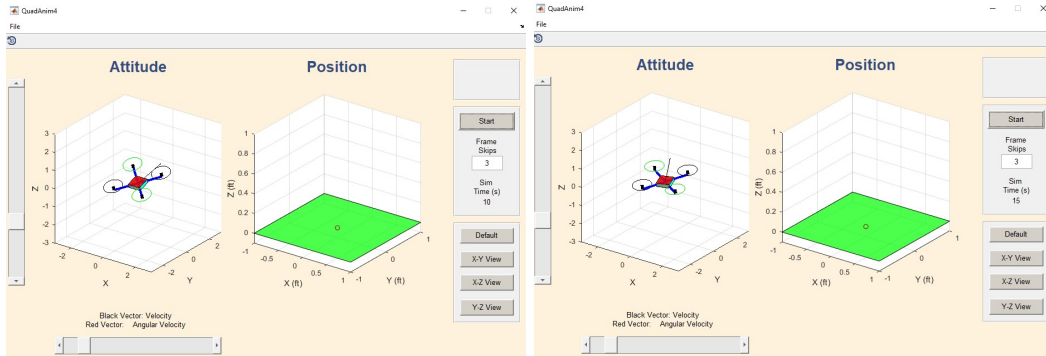




The plots shown above correspond to angle $[\phi, \theta, \psi]$ and angular velocity $[P, Q, R]$ of the quadcopter about all the axes. This is a result of the output obtained for the previous block which gave the state data. We are primarily concerned with the plots of angle vs time and hope for a slightly underdamped response in these cases.

The first plot shows the response response of the system to a 10° step input to θ at $t = 0$
The second plot shows the response of the system to hybrid inputs at different points in time.

14.2 OPEN GUI: Flight Animation



The 3D animation gives us visual information about the orientation and position of the quadcopter in 3D space, for this project we have not allowed the quadcopter to translate in space so we are primarily concerned with the orientation (shown on the left side of the above figures).

The first figure shows the response of the system to a 10° step input to θ at $t = 0$
The second figure shows the response of the system to hybrid inputs at different points in time.

Part IV

Hardware

15 Motor characterization

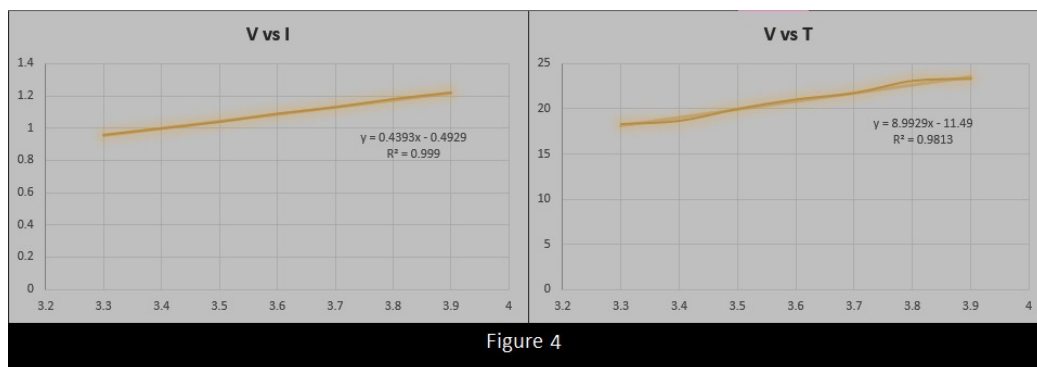
The motor we will be using for this project is the one shown in Figure 3. These type of motors are called core-less motor. The main reason behind using these motors is because they can generate very high rpm at very low voltages.



Figure 3

The characterization of this motor was done to figure out the relation between Voltage and Current and Voltage and Thrust. The results obtained are as shown below.

<i>Volts(V)</i>	<i>Amps(A)</i>	<i>Thrust(gcm)</i>
3.3	0.96	18.36
3.4	1.00	18.71
3.5	1.04	20.01
3.6	1.09	21.04
3.7	1.13	21.73
3.8	1.18	23.06
3.9	1.22	23.28



From Figure 4 we can conclude the relation between the analyzed quantities as follows.

$$I = (0.4393 * V) - 0.4929 \quad (36)$$

$$T = (8.9929 * V) - 11.49 \quad (37)$$

These equations were obtained using Microsoft Excels built-in curve fitting feature.

16 Inertial Measurement Unit (MPU6050)



This sensor is intended to be used to measure the absolute orientation of an object. MPU6050 is an integration of these 2 sensors:

- Accelerometer
- Gyroscope

It has an on board Digital Motion Processor to evaluate motion fusion operations. To summarize their individual operations:

- Accelerometer
Used to measure the components of linear acceleration of the object in 3 dimensions with respect to the body axes of the sensor.
- Gyroscope
Used to measure the angular rate of change of the object with respect to the body axes of the sensor.

The unique combination of these 2 sensors is what enables us to obtain the absolute orientation of the drone in 3D space, this introduces feedback to the plant by giving us real time information about the orientation of the drone. Our PID algorithm will evaluate errors based on outputs from this sensor.

17 Driver circuit

The driver circuit that works best for the coreless motors we are using, is the one shown in the figure above. The circuit allows unidirectional control of the motor using the PWM signal from the Arduino that is then amplified by the MOSFET.

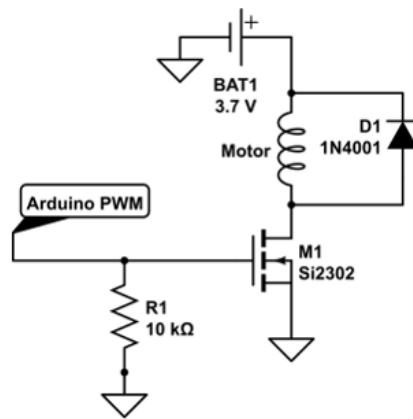


Figure 5

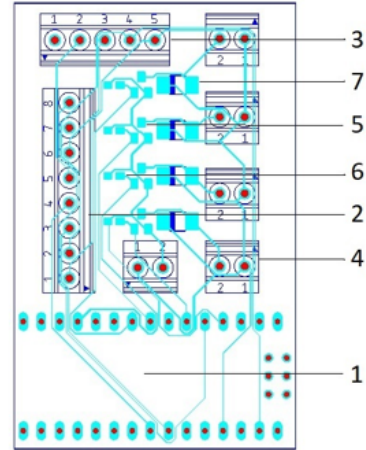


Figure 6

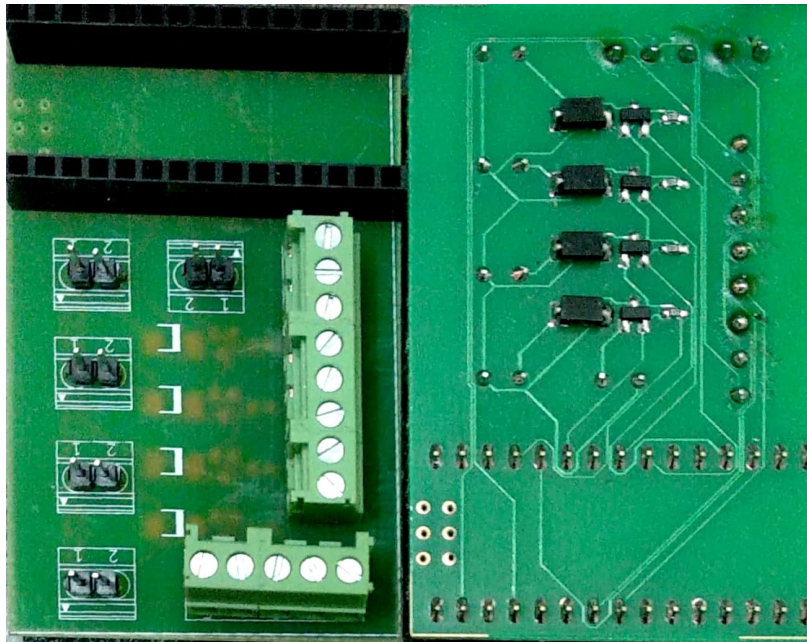


Figure 7

Figure 5 shows the schematic for a single brushless motor. For our application we need 4 identical circuits to control the four motors independently. To facilitate this a PCB was fabricated that

accommodates the controller, sensors and driver circuit on the same board (as shown in Figure 6).

Figure 7 shows the fabricated board. The components used in the fabricated board are as follows:

1. Main Controller (Arduino Nano)
2. Accelerometer + Gyroscope (MPU6050)
3. Compass (HMC5883)
4. Motor Output Pins
5. SMD Logic Level MOSFET (Si2302, SOT-23/TO-236)
6. 10k resistors (0603)
7. Freewheeling Diode (1N4001, DO-214AC)

17.1 Justification for the components

- **Arduino Nano**
We chose this microcontroller to carry out the task of stabilization of drone. We have implemented a PID control for attitude stabilization of the drone. This was the best microcontroller which satisfied our memory, power and form factor requirements.
- **MPU6050**
We first experimented with MPU9250 which had Accelerometer+Gyroscope+Magnetometer. We realized that the yaw given by the sensor was not reliable. Therefore we decided to go for MPU6050 and use relative yaw instead of absolute yaw.
- **Si2302**
The reason why we went for this logic level MOSFET is because it can be easily gated by the output pins of Arduino Nano without the help of any external gating circuit. The current rating of this MOSFET was also appropriate for our application. The reason for choosing the SMD version is to reduce the form-factor of the PCB.
- **10k resistor**
This resistor is used to limit the gate current flowing into the MOSFET incase of an overshoot. SMD version has been chosen to reduce the form-factor of the PCB.
- **1N4001**
This diode is used a free-wheeling diode to protect the circuit from inductive fly-back effect of the motor. SMD version has been chosen to reduce the form-factor of the PCB.

Efforts were made to reduce the form factor by as much as possible. To make the circuit board more robust, tracks that carry high current were made twice as thick as tracks use for logic connections.

18 Test rig

The quadcopter is connected to a mount which is connected to a spherical joint which is connected to a base gives it all required degrees of freedom (as shown in Figure 8). However our quad was too heavy to use this mount. Therefore in order to test the quad we used the same mount but suspended using a thread (as shown in Figure 9) giving us the ability to test yaw. We were able to test roll and pitch also to a certain extent.

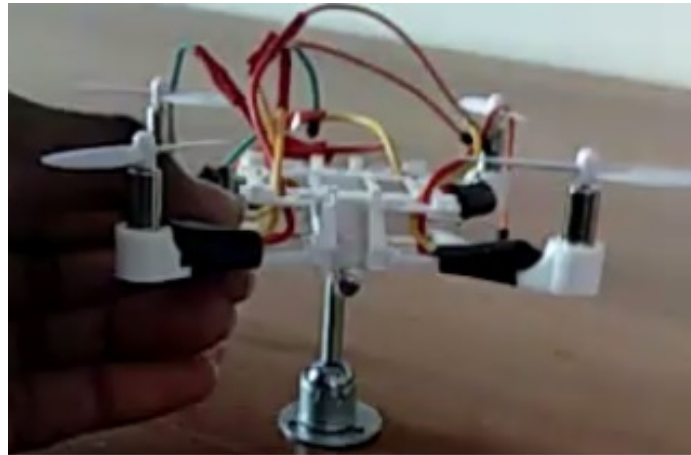


Figure 8



Figure 9

Part V

Control

19 Control considerations

A typical global coordinate system based trajectory planning set up would have goals specified according to a start and end point, or some curve expressing the trajectory to be followed. To successfully implement such a system we would need to design a controller with the following process variables:

- Position
- Linear velocity
- Angular velocity
- Linear acceleration
- Angular acceleration
- Jerk
- Snap

This results in a very complex model that is hard to compute in real time. Also, some of these parameters cannot even be accurately measured with existing sensor technology. This makes the approach practically unfeasible though mathematically robust.

It is very common to find roll, pitch, yaw and altitude control in a parallel PID setup, as this is the most feasible in terms of physical implementation, due to its inherent simplicity and the coupling that these rotational changes and translation motion have. The disadvantage of this is that the operator of the quadcopter would now need to be consciously manipulating these parameters in order to obtain his desired trajectory.

However, in autonomous systems it is common to find another layer of control implemented to plan trajectories by manipulating roll pitch yaw and altitude. A sort of multilevel structure simplifies the problem mathematically while giving us the same results as the case described previously. From a control systems perspective we can look at this system as one that is controlled by a cascaded PID.

Though most of this is out of the scope of what we intend to achieve we think it is important to understand this distinction and the limitations associated with it.

20 Mathematical Implementation

Here we use a simple PID controller to control $[\phi \ \theta \ \psi]$. These values are read in real time from MPU6050.

Note: If required altitude control can also be implemented in this by using BMI180 pressure sensor to detect the z co-ordinate in an absolute sense.

We know that

$$\tau = Iu(t) \quad (38)$$

$u(t)$: Input to the system

Since we are implementing a PID control the input depends on the error. According to the equations of the PID controller we get

$$\begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} -I_{xx}(k_p e_\phi + k_d \dot{e}_\phi + k_i \int e_\phi dt) \\ -I_{yy}(k_p e_\theta + k_d \dot{e}_\theta + k_i \int e_\theta dt) \\ -I_{zz}(k_p e_\psi + k_d \dot{e}_\psi + k_i \int e_\psi dt) \end{bmatrix} \quad (39)$$

For simplicity assume

$$\gamma_i = \omega_i^2 \quad (40)$$

From (18) and (39) we get

$$\begin{bmatrix} \gamma_2 + \gamma_3 - \gamma_1 - \gamma_4 \\ \gamma_3 + \gamma_4 - \gamma_1 - \gamma_2 \\ \gamma_1 - \gamma_2 + \gamma_3 - \gamma_4 \end{bmatrix} = \begin{bmatrix} \frac{I_{xx}(k_p e_\phi + k_d \dot{e}_\phi + k_i \int e_\phi dt)}{Lk\sqrt{2}} \\ \frac{I_{yy}(k_p e_\theta + k_d \dot{e}_\theta + k_i \int e_\theta dt)}{Lk\sqrt{2}} \\ \frac{I_{zz}(k_p e_\psi + k_d \dot{e}_\psi + k_i \int e_\psi dt)}{b} \end{bmatrix} \quad (41)$$

From the above equations we have 4 unknowns and 3 equations. Therefore in order to calculate the 4 unknowns we can add a constraint $T=mg$.

Projecting it onto the inertial frame we get

$$T = \frac{mg}{c_\theta c_\phi} \quad (42)$$

This implies

$$\gamma_1 + \gamma_2 + \gamma_3 + \gamma_4 = \frac{mg}{c_\theta c_\phi} \quad (43)$$

In order to obtain the input equations for the speed of the four motors equations (41) and (43) has to be solved.

21 Practical implementation

The PID controller is implemented on a micro-controller. This project requires 3 parallel PIDs for roll pitch and yaw respectively followed by control mixing for a cross configuration quadcopter. The error variable considered here is the R-P-Y angles in degrees obtained from the sensor (MPU6050). The initial calibration routine runs for 15 seconds during which the yaw drift settles and the current position of the quadcopter corresponds to 0 degrees in all degrees of freedom. In the implementation of the PID time taken for each iteration is measured and used to calculate differential and integral error terms.

21.1 Arduino Code



```
Attitude_control | Arduino 1.8.4
File Edit Sketch Tools Help

1 #include "I2Cdev.h"
2 #include "MPU6050_6Axis_MotionApps20.h"
3
4 MPU6050 mpu;
5
6 #define INTERRUPT_PIN 2
7 #define LED_PIN 13
8 bool blinkState = false;
9
10 // MPU control/status variables
11 bool dmpReady = false; // set true if DMP init was successful
12 uint8_t mpuIntStatus; // holds actual interrupt status byte from MPU
13 uint8_t devStatus; // return status after each device operation (0 = success, != 0 = error)
14 uint16_t packetSize; // expected DMP packet size (default is 42 bytes)
15 uint16_t fifoCount; // count of all bytes currently in FIFO
16 uint8_t fifoBuffer[64]; // FIFO storage buffer
17
18 // orientation/motion variables
19 Quaternion q; // [w, x, y, z] quaternion container
20 VectorInt16 aa; // [x, y, z] accel sensor measurements
21 VectorInt16 aaReal; // [x, y, z] gravity-free accel sensor measurements
22 VectorInt16 aaWorld; // [x, y, z] world-frame accel sensor measurements
23 VectorFloat gravity; // [x, y, z] gravity vector
24 float euler[3]; // [psi, theta, phi] Euler angle container
25 float ypr[3]; // [yaw, pitch, roll] yaw/pitch/roll container
26 float ypri[3]; // [yaw, pitch, roll] initial yaw/pitch/roll container
27
28 // =====
29 // === INTERRUPT DETECTION ROUTINE ===
30 // =====
31
```

141 Arduino Nano, ATmega328P on COM12

```
Attitude_control$
32 volatile bool mpuInterrupt = false;    // indicates whether MPU interrupt pin has gone high
33
34 void dmpDataReady()
35 {
36     mpuInterrupt = true;
37 }
38
39 // =====
40 // ===                INITIAL SETUP                ===
41 // =====
42
43 void setup()
44 {
45     pinMode(5, OUTPUT);
46     pinMode(6, OUTPUT);
47     pinMode(10, OUTPUT);
48     pinMode(11, OUTPUT);
49
50     Serial.begin(115200);
51     while (!Serial);
52
53     // initialize device
54     mpu.initialize();
55     pinMode(INTERRUPT_PIN, INPUT);
56
57     // verify connection
58     Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") : F("MPU6050 connection failed"));
59
60     // empty buffer
61     while (Serial.available() && Serial.read());
62
63     // ... ..
```

```
Attitude_control $
63 // load and configure the DMP
64 devStatus = mpu.dmpInitialize();
65
66 // supply your own gyro offsets here, scaled for min sensitivity
67 mpu.setXGyroOffset(220);
68 mpu.setYGyroOffset(76);
69 mpu.setZGyroOffset(-85);
70 mpu.setZAccelOffset(1788); // 1688 factory default for my test chip
71
72 // make sure it worked (returns 0 if so)
73 if (devStatus == 0)
74 {
75     // turn on the DMP, now that it's ready
76     mpu.setDMPEEnabled(true);
77
78     // enable Arduino interrupt detection
79     attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), dmpDataReady, RISING);
80     mpuIntStatus = mpu.getIntStatus();
81
82     // set our DMP Ready flag so the main loop() function knows it's okay to use it
83     dmpReady = true;
84
85     // get expected DMP packet size for later comparison
86     packetSize = mpu.dmpGetFIFOPacketSize();
87 }
88
89 else
90 {
91     Serial.print(F("DMP Initialization failed"));
92 }
93
```

252 Arduino Nano, ATmega328P on COM12

Attitude_control

```
94 // configure LED for output
95 pinMode(LED_PIN, OUTPUT);
96 }
97
98 float t,t1;
99 //Initialing PID Gains
100 float kp[3]={0.7,0.7,1};
101 float ki[3]={0,0,0};
102 float kd[3]={0.1,0.1,0.12};
103 float u_roll,u_pitch,u_yaw;
104 float ud_roll,ud_pitch,ud_yaw;
105 float ui_roll,ui_pitch,ui_yaw;
106 float e_roll,e_pitch,e_yaw;
107 float ui_roll_prev,ui_pitch_prev,ui_yaw_prev;
108 float e_roll_prev,e_pitch_prev,e_yaw_prev;
109 int m1,m2,m3,m4;
110 //Variables used in PID controller
111 float roll_meas=0,roll_req=0,offset_r=0;
112 float pitch_meas=0,pitch_req=0,offset_p=0;
113 float yaw_meas=0,yaw_req=0,offset_y=0;
114
115 // =====
116 // ===          MAIN PROGRAM LOOP          ===
117 // =====
118
119 void loop()
120 {
121   t=millis();
122
123   while(1)
124   {
```

Done Saving.

Attitude_control

```
125 while(millis()-t<15000)
126 {
127   IMU();
128 }
129
130 while(millis()-t>15000 && millis()-t<15100)
131 {
132   IMU();
133
134   //Converting radians to degrees
135   ypri[0]=ypr[0] * 180/M_PI;
136   ypri[1]=ypr[1] * 180/M_PI;
137   ypri[2]=ypr[2] * 180/M_PI;
138 }
139
140 while(millis()-t>15200)
141 {
142   IMU();
143   ypr[0]=ypr[0] * 180/M_PI;
144   ypr[1]=ypr[1] * 180/M_PI;
145   ypr[2]=ypr[2] * 180/M_PI;
146
147   //Simple check to make sure the system does not go uncontrollable
148   if((abs(ypri[1]-ypr[1])<30) && (abs(ypri[2]-ypr[2])<30))
149   {
150     PID();
151     delay(3);
152   }
153
154   else
155   {
```

Done Saving.

Attitude_control

```
156
157     }
158   }
159 }
160 }
161
162 // =====
163 // ===          GETTING IMU DATA          ===
164 // =====
165
166 void IMU()
167 {
168   if (!dmpReady) return;
169
170   while (!mpuInterrupt && fifoCount < packetSize)
171   {
172   }
173 }
174
175 // reset interrupt flag and get INT_STATUS byte
176 mpuInterrupt = false;
177 mpuIntStatus = mpu.getIntStatus();
178
179 // get current FIFO count
180 fifoCount = mpu.getFIFOCount();
181
182 // check for overflow (this should never happen unless our code is too inefficient)
183 if ((mpuIntStatus & 0x10) || fifoCount == 1024)
184 {
185   // reset so we can continue cleanly
186   mpu.resetFIFO();
187   delay(7);
```

Done Saving.

Attitude_control

```
187     delay(7);
188 }
189
190 else if (mpuIntStatus & 0x02)
191 {
192     while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();
193
194     // read a packet from FIFO
195     mpu.getFIFOBytes(fifoBuffer, packetSize);
196
197     // track FIFO count here in case there is > 1 packet available
198     // (this lets us immediately read more without waiting for an interrupt)
199     fifoCount -= packetSize;
200
201     // blink LED to indicate activity
202     blinkState = !blinkState;
203     digitalWrite(LED_PIN, blinkState);
204 }
205 }
206
207 // =====
208 // ===                PID CONTROL                ===
209 // =====
210
211 void PID()
212 {
213     //Get errors
214     e_roll=(ypri[2]-ypr[2]);
215     e_pitch=(ypri[1]-ypr[1]);
216     e_yaw=(ypri[0]-ypr[0]);
217 }
```

Done Saving.



Attitude_control\$

```

218   tl=(millis()-t)/1000;
219
220   // Calculating I and D correction terms
221   ui_roll = ui_roll_prev + (ki[0] * tl*e_roll);
222   ud_roll = kd[0] * (e_roll - e_roll_prev)/tl;
223   ui_pitch = ui_pitch_prev + (ki[1] * (tl/10)*e_pitch);
224   ud_pitch = kd[1] * (e_pitch - e_pitch_prev)/tl;
225   ui_yaw = ui_yaw_prev + (ki[2] * (tl/10)*e_yaw);
226   ud_yaw = kd[2] * (e_yaw - e_yaw_prev)/tl;
227
228   //Retaining previous values for Integrator
229   e_roll_prev = e_roll;
230   ui_roll_prev = ui_roll;
231   e_pitch_prev = e_pitch;
232   ui_pitch_prev = ui_pitch;
233   e_yaw_prev = e_yaw;
234   ui_yaw_prev = ui_yaw;
235
236   //Evaluating PID correction term
237   u_roll = ((kp[0] * e_roll) + ui_roll + ud_roll);
238   u_pitch = ((kp[1] * e_pitch) + ui_pitch + ud_pitch);
239   u_yaw = ((kp[2] * e_yaw) + ui_yaw + ud_yaw);
240
241   //Control mixing for 'X' configuration
242   m1=150+u_roll+u_pitch-u_yaw;
243   m2=150+u_roll-u_pitch+u_yaw;
244   m3=150-u_roll-u_pitch-u_yaw;
245   m4=150-u_roll+u_pitch+u_yaw;
246   //Writing values to the motors
247   analogWrite(6,m1); analogWrite(5,m2); analogWrite(11,m3); analogWrite(10,m4);
248 }

```

Done Saving.

245

Arduino Nano, ATmega328P on COM12

Part VI

Result

We successfully

- Modeled quadcopter for attitude control with graphs and animation as output.
- Implemented parallel PID using Arduino Nano as microcontroller and MPU6050 as sensor for feedback.
- Designed 3 DOF test rig.
- Verified working for yaw stabilization to counter disturbance inputs.

We failed to

- Verify working for roll and pitch stabilization to counter disturbance inputs.

Part VII

Conclusion

22 Learning experience

This project helped us understand and appreciate the process of designing and implementing a controller in a practical scenario. We fully understood the working of a quadcopter which is one of the most famous applications of control theory. We got to work with features in MATLAB and Simulink that we were previously unaware of such as the S-Function Builder and the ability to represent output as an animation. Working with the MPU6050 and comparing its performance with other IMUs taught us many new things about them. Designing the test rig for the quadcopter was an interesting experience. Designing and experimenting with different driver circuits for coreless motors was a good learning experience during which we had our first attempt at SMD level fabrication. Seeing the controller in action was also a very satisfying result.

23 Failure log and future scope

We have identified the main difficulty to be a disproportionate power to weight ratio. This was expected to be a challenge as designing a micro quad rotor is much harder than a regular sized one but we definitely underestimated its complexity. The hardest part was designing a driver circuit that was small, light and still capable of powering 4 motors that draw reasonably high currents and this is where we failed since we were fixated on having a driver and controller that could be mounted on a small 5x5 cm form factor. The L293 is slightly under-powered for this application and the Si2302 despite being ideal for this use case is available only in surface mount package so testing the design before printing was impossible, but we went ahead with it anyway. The driver worked but did not perform any better than the L293. This could be due to some mistake in design or the tracks on the PCB being too thin for the high current the motors drew (despite us taking that into consideration). In conclusion, a perfect driver circuit would have led to a perfect execution of this

project and that is the primary scope for improvement. Once that is done the next step would be to replace the Arduino Nano with a NodeMCU and send commands to the drone wirelessly. Once satisfied with this, an RC application can be designed and the drone can be removed from the test rig and will be capable of flight.

References

- [1] Shariq Neshat Akhtar, "The use of modern tools for modelling and simulation of UAV with haptic."
- [2] Zaid Tahir", "State Space System Modelling of a Quad Copter UAV."