CONTROL OF CRAZYFLIE NANO QUADCOPTER USING SIMULINK

A THESIS

Presented to the Department of Electrical Engineering

California State University, Long Beach

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

Committee Members:

Bahram Shahian, Ph.D. (Chair)
Anastasios Chassiakos, Ph.D.
Parviz Talebi, M.S.

College Designee:

Antonella Sciortino, Ph.D.

By Meghana Gopabhat Madhusudhan

B.E., 2012, Visvesaraya Technological University

May 2016

ABSTRACT

CONTROL OF CRAZYFLIE NANO QUADCOPTER USING SIMULINK

By

Meghana Gopabhat Madhusudhan

May 2016

This thesis focuses on developing a mathematical model in Simulink to Crazyflie, an open source platform. Attitude, altitude and position controllers of a Crazyflie are designed in the mathematical model. The mathematical model is developed based on the quadcopter system dynamics using a non-linear approach. The parameters of translational and rotational dynamics of the quadcopter system are linearized and tuned individually. The tuned attitude and altitude controllers from the mathematical model are implemented on real time Crazyflie Simulink model to achieve autonomous and controlled flight.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor Dr. Bahram Shahian for providing me an opportunity to get hands on experience in the cutting edge technology. I would like to thank him for his continuous support, patience, motivation, knowledge, enthusiasm and guidance during this thesis.

I would like to thank Event 38 Unmanned Systems and K5 Ventures team for providing me an opportunity to intern and explore the essence of control systems and Unmanned Aerial Vehicles.

I thank my parents, all my friends and family for their encouragement.

Last but not the least, I like to thank the immortal for his blessings, love and guidance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

UAV	Unmanned Aerial Vehicle
MAV	Micro Aerial Vehicle
NAV	Nano Aerial Vehicle
MIT	Massachusetts Institute of Technology
AMR	Anisotropic Magnetoresistive
MEMS	Micro Electro Mechanical Sensor
IMU	Inertial Measurement Unit
PWM	Pulse Width Modulation
DC	Direct Current
LiPo	Lithium Polymer
GPS	Global Positioning System
GNNS	Global Navigation Satellite System
UART	Universal Asynchronous Receiver Transmitter
MCU	Micro Controller Unit
PID	Proportional – Integral - Derivative
LQR	Linear Quadratic Regulator
RTOS	Real-Time Operating System
UDP	User Datagram Protocol
ISM	Industrial, Scientific and Medical

CHAPTER 1

INTRODUCTION

Unmanned Aerial Vehicle (UAV) also known as drone is an aircraft without pilot on board. UAVs are motorized aircrafts that fly autonomously or operated by remote control. Based on the aerodynamic principles, UAVs can be classified as fixed-wing and rotary-wing. Fixed-wing UAVs consist of a rigid wing which makes the flight capable to produce lift caused by forward propulsion. The fixed-wing UAV features ailerons, elevator and rudder as its control surfaces to achieve three axis of rotation. In rotary-wing, the constant movement of the propellers produce airflow over the airfoil and generate lift. The rotary-wing UAV has simple construction and three axis of rotation configuration is achieved by varying the thrust of the rotors. The greatest advantage of rotary-wing UAVs is that they can take off and land vertically. The fixed wing and rotary-wing UAVs are shown in fig 1.1.

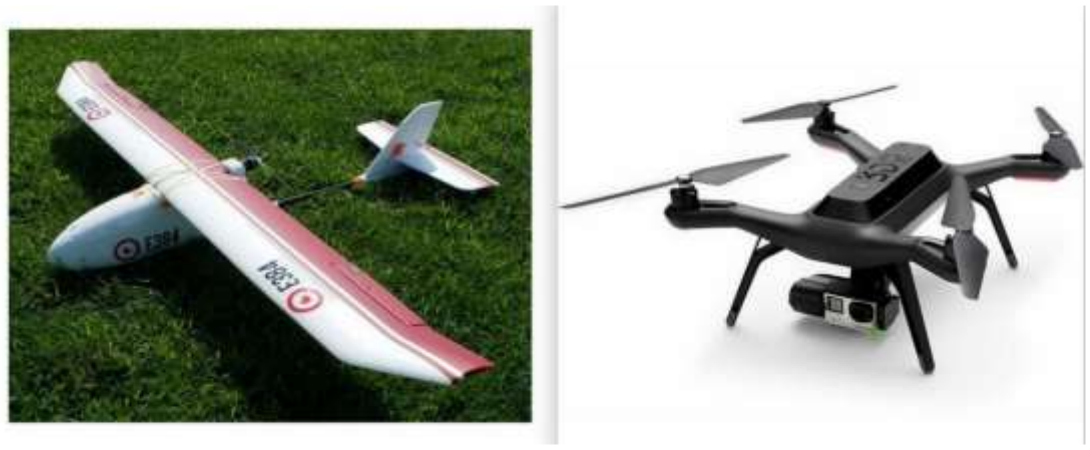


FIGURE 1.1. E384 fixed-wing UAV and 3DR solo rotary-wing UAV.

History of Quadcopter

The multi-rotors have a very rich history. The evolution of multi-rotors began during the early 19th century. The first four-rotor helicopter was built in 1907 by Louis Breguet. The second multi-rotor was designed by a French engineer Etienne Oehmichen and built with steel-tube frame, four rotors and eight propellers as shown in fig 1.2 [1].

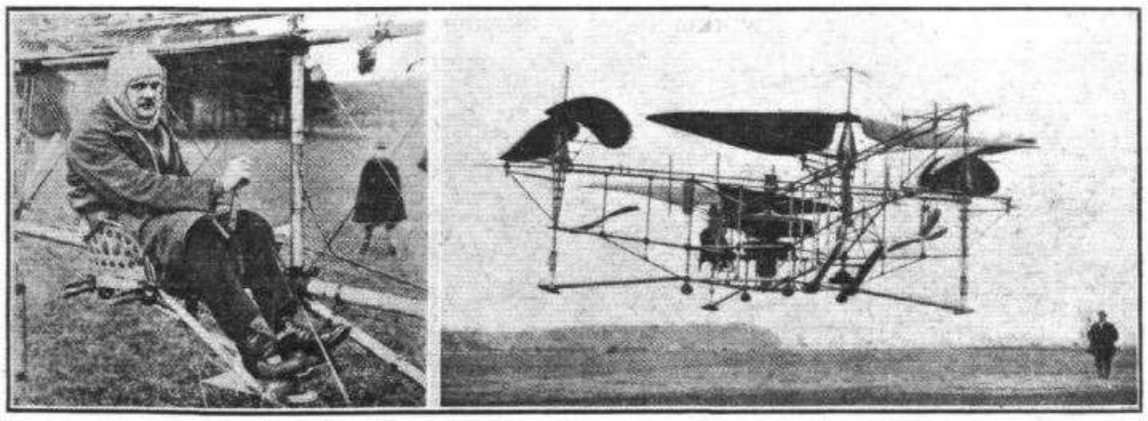


FIGURE 1.2. Oehmichen and his multi-rotor helicopter [1].

In 1950s, a unique quadrotor helicopter prototype fuselage made up of tubular steel, supporting booms of rotors made up of aluminum alloy and control mechanism having two engines driving four rotors. The thrust was controlled by the speed of the rotors and had a very simplified design. Designer and test pilot of Convertawings Model A Quadcopter, D.H. Kaplan demonstrated its first flight at Long Island in March 1956 and proved its design [2]. The Convertawings Model A quadcopter is as shown in fig 1.3.

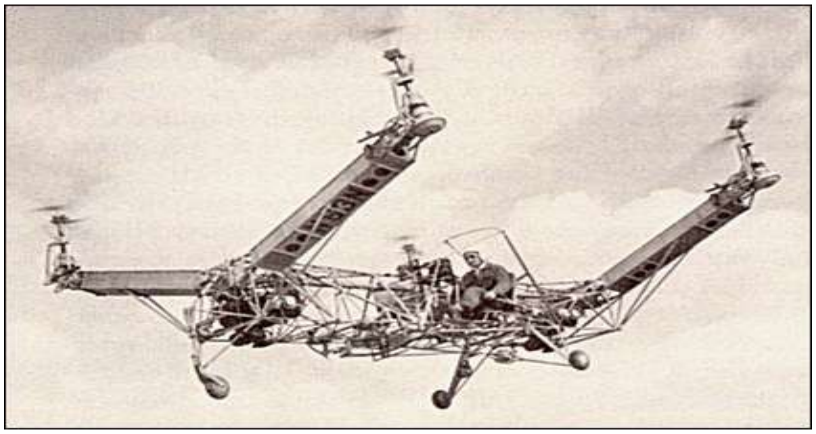


FIGURE 1.3. Convertawings Model A Quadcopter [2].

Due to various technical constraints, multi-rotors helicopters did not gain much significance and were abandoned in favor of single rotor helicopter. With the advent of smart technology, unmanned rotary-wing systems or unmanned aerial vehicles emerged replacing the old control mechanisms with microcontrollers, microeletrical and micromechanical devices. The unmanned rotary-wing aircrafts are classified based on number of rotors namely, helicopter (one), tricopter (three), quadcopter (four), hexacopter (six), octacopter (eight). The multi-rotor system offers a great stability system and control accuracy.

During the past decade, bio-inspired biologists and aerial roboticists introduced micro and nano aerial vehicles showcasing emulative unprecedented flight capabilities of birds and insects [3]. This inspiration has also led to stabilize and control artificial swarms of MAVs and NAVs. Nano Hummingbird was the first experimentation towards NAVs and is as shown in fig 1.4. The MAVs and NAVs were developed by US Army with a motive to reach through diminutive space for survey and spying.

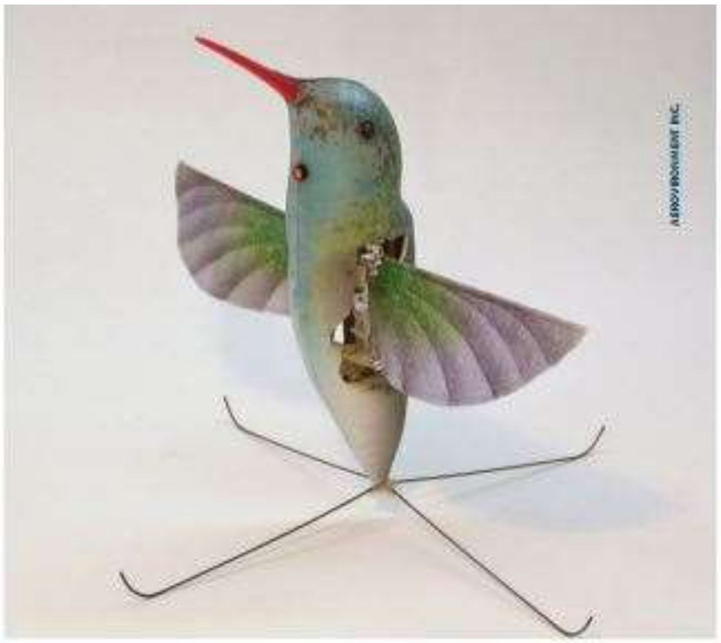


FIGURE 1.4. Nano Hummingbird drone [3].

Initially, UAVs took their place in defense to achieve target, decoy, reconnaissance and combat. With subsequent development, UAV application progressed extensively in various fields like logistics-delivering medical supplies to remote areas, aerial surveying, aerial film footage, crop monitoring, search and rescue operations and construction management.

CHAPTER 2

CRAZYFLIE DEVELOPMENT

Software

Bitcraze is offering an interesting open source platform for aerial robotic enthusiasts with a nano quadcopter, Crazyflie. Crazyflie 1.0 was developed and launched in 2010. The open source schematics and firmware source code of Crazyflie helps designers to easily start and expand capabilities. The smaller size makes it user friendly to fly indoors and develop new hardware/software implementations.

Various experiments were performed by researchers and enthusiasts. Programs are developed to check the performance of the hardware and system dynamics. Researchers came up with ideas to control Crazyflie using voice, Kinect and several other electronic gadgets. A software, OS4 simulator was designed to control and check the characteristics of Crazyflie 1.0 [4]. With the development of the firmware and hardware, Bitcraze released Crazyflie 2.0 in late 2013.

A researcher from MIT implemented position hold on Crazyflie 2.0 with VICON sensor [5]. This research involves development of Iterative Regional Inflation by Semidefinite programming algorithm, mixed Integer Semidefinite programs and model based control approaches in Drake – RTOS software. This approach was put forward to enable quadrotors to fly through crowded environments by detecting obstacles [5]. Another research team developed an algorithm to control multiple Crazyflies using single Crazyradio enabling swarm control.

Hardware

Crazyflie is a miniature quadcopter. The development of Crazyflie quadcopter started in late 2009 as a competence development project called Daedalus by Swedish consulting company

Epsilon AB. In 2010, first prototype flight video was sent to Hackaday.com. Eventually, development advanced manufacturing Crazyflie kit and open source development platform. Bitcraze AB was created to finance the development and manufacturing of the Crazyflie kit. The version 1.0 was the first available Crazyflie kit and version 2.0 was launched in December 2015. Crazyflie 1.0 weighs 19gm and measures 9cm motor to motor with a flight time of 7 minutes. The Crazyflie hardware model and hardware block diagrams are shown in fig 2.1 and fig 2.2.

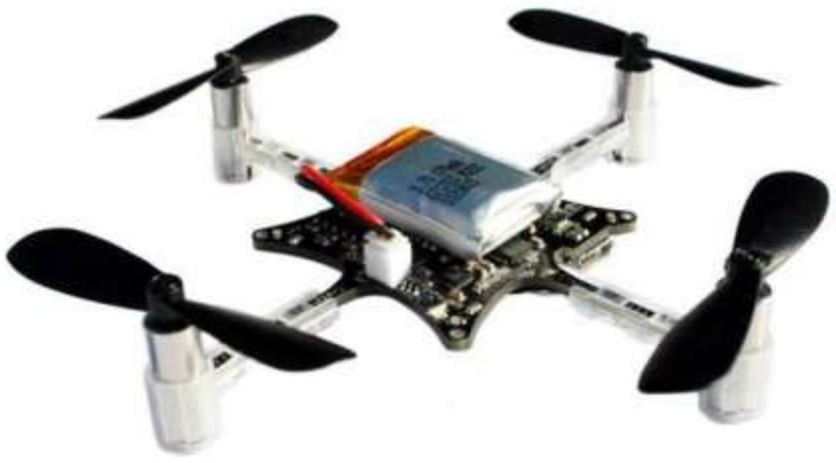


FIGURE 2.1. Crazyflie nano quadcopter version 1.0 [6].

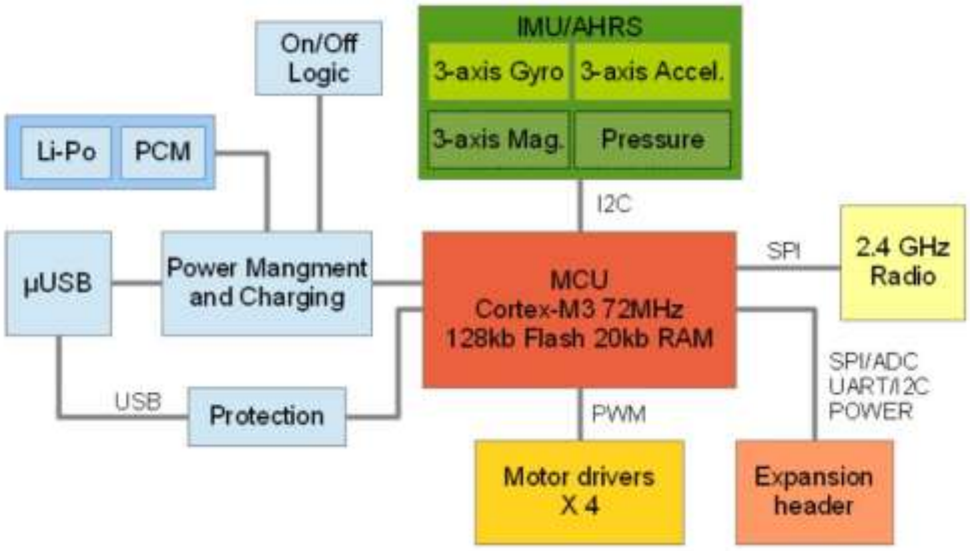


FIGURE 2.2. Hardware overview of Crazyflie nano quadcopter [6].

Microcontroller

To achieve control and stabilization of the system, a control unit is required. A microcontroller is used to make decisions based on the sensor input to stabilize the quadcopter system. Crazyflie 1.0 uses a powerful 32 bit microcontroller unit STM32F103CB operating at 72 MHz to control all the communications and functions onboard.

Accelerometer

The Crazyflie uses a microelectromechanical system (MEMS) accelerometer to measure acceleration in all 3 dimensions. An accelerometer measures translational motion of the Crazyflie quadcopter. The accelerometer used in Crazyflie is a cantilever type with a separate mass for each individual axis. Acceleration is measured by the capacitive sensors when a displacement of the mass occurs.

Gyroscope

Gyroscope is a device which is used to maintain orientation of the object. A MEMS gyroscope measures the rotational motion by vibrating a proof-mass and sensing Coriolis effect caused by angular velocity. A dual-mass are coupled to a sense frame and Coriolis torque moves the four-bar linkage which is sensed by capacitive sensors [7]. The obtained capacitance is then converted into output as binary or an analog voltage. Crazyflie uses a high frequency 3-axis MEMs gyroscope to measure the rotational rate for X, Y, Z axes.

Magnetometer

Crazyflie uses a HMC5883 3-axis magnetometer. It is a MEMS magnetic sensor to detect the direction of magnetic field. Thus, a 3-axis magnetometer can detect magnetic field intensity in all the three axis. The HMC5883L utilizes anisotropic magnetoresistive (AMR) technology and features in-axis sensitivity and linearity [7].

Inertial Measurement Unit (IMU)

An Inertial Measurement Unit is used to determine angular position and attitude of a quadcopter. Crazyflie 1.0 quadcopter is equipped with an IMU, Invensense MPU-6050. The IMU contains 3-axis high performance MEMs gyro with 3-axis accelerometer [5]. An IMU uses accelerometer to detect the rate of acceleration and gyroscopes to detect the change in rotation attributes like roll, pitch and yaw.

Motors

The Crazyflie uses brushed DC motors to drive its propellers. The brushed DC motors are known for their high torque to inertia ratio and provide precision control of speed. The DC motors in the Crazyflie are driven using pulse width modulation (PWM). Motor speed can be varied by PWM without changing the voltage. The speed of the motor is determined by PWM and supply voltage [7].

Propellers

Propeller is a mechanical device which generates thrust by converting its rotational motion. Crazyflie propellers are fragile plastic propellers with a wingspan of 45mm.

Power and Battery

Crazyflie uses a standard 3.7V 170mAh Lithium-Polymer (LiPo) battery. The lower weight and high capacity makes the LiPo battery popular in the radio-controlled cars and radio-controlled aircrafts. The battery accompanies a Protection Circuit Module (PCM) to prevent electrical damage [6]. The battery is generally charged through a micro USB connector and approximately takes 20 minutes to recharge the battery.

Radio

Crazyflie quadcopter uses USB dongle for radio communication and is called Crazyradio. Crazyflie is equipped with nRFM24L01+ chip on board for its radio communication with the Crazyradio which has a Nordic Semiconductor nRF24L01+ chip. The two chips operate and communicate with each other at 2.4GHz ISM band. The communication hardware is shown in fig 2.3.

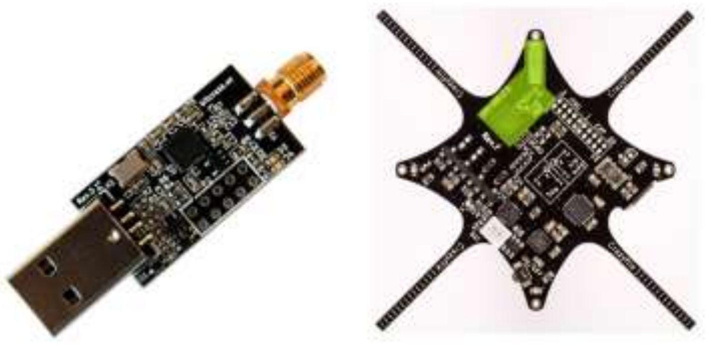


FIGURE 2.3. Crazyradio (left) and Crazyflie (right) hardware communication [6].

Global Positioning System (GPS)

A GPS is a navigational system which involves satellites and computers to determine the latitude, longitude and altitude of a receiver on air, space, ground and sea. Crazyflie is externally equipped with u-blox MAX-M8C GNNS module interfaced by UART. The u-blox MAX-M8C module delivers high sensitivity and minimal acquisition time maintaining low system power. The positioning module is designed to receive and track Legacy Civilian (L1C/A) signals provided at 1575.42MHz by GPS [8]. These features aid Crazyflie to fly autonomously and accurately from one location to another. The fig 2.4 shows u-blox MAX-M8C module integrated with Crazyflie.

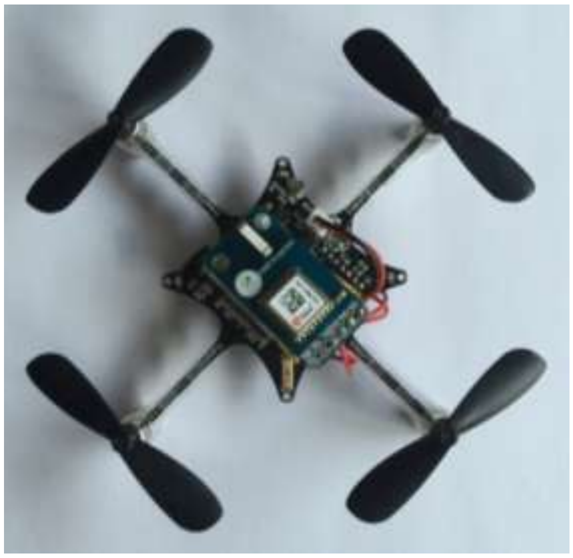


FIGURE 2.4. Crazyflie with u-blox MAX-M8C GPS.

CHAPTER 3

SYSTEM MODELING

A quadcopter consists of four rotors at the end of its four arms, configured as twin motors. A pair of opposite motors rotating clockwise (M2 and M4) and counter-clockwise (M1 and M3) and the flight dynamics is as shown in fig 3.1. The three axis of rotation Roll (x-axis), Pitch (y-axis) and Yaw (z-axis) being controlled by adjusting the thrust of the motors. The quadcopter system modeling is done with Euler-Lagrange Formalism.

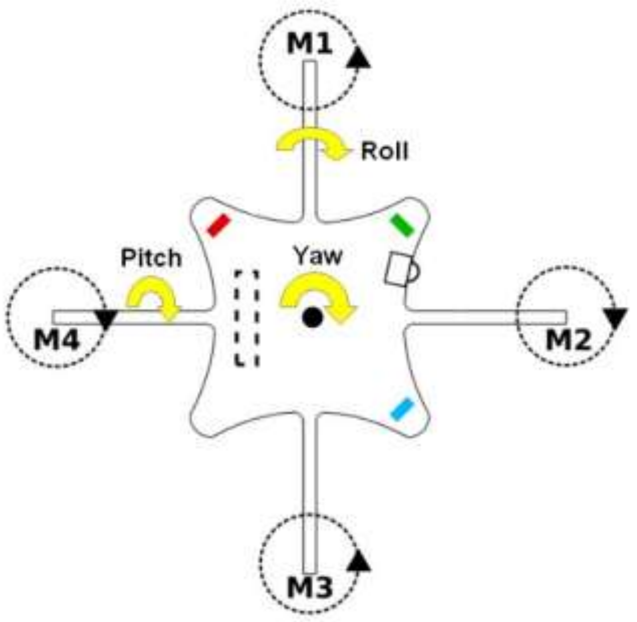


FIGURE 3.1. Crazyflie quadcopter flight dynamics [6].

A point of reference is required on the quadcopter to track its rotation and movement. Generally, physical center of the quadcopter or center of gravity is used as a point of reference. The physical center of quadcopter is chosen since,it is easily determined. Once the center of body is chosen as a reference, arms of the quadcopter can be used as x and y axis with z-axis being orthogonal to it.

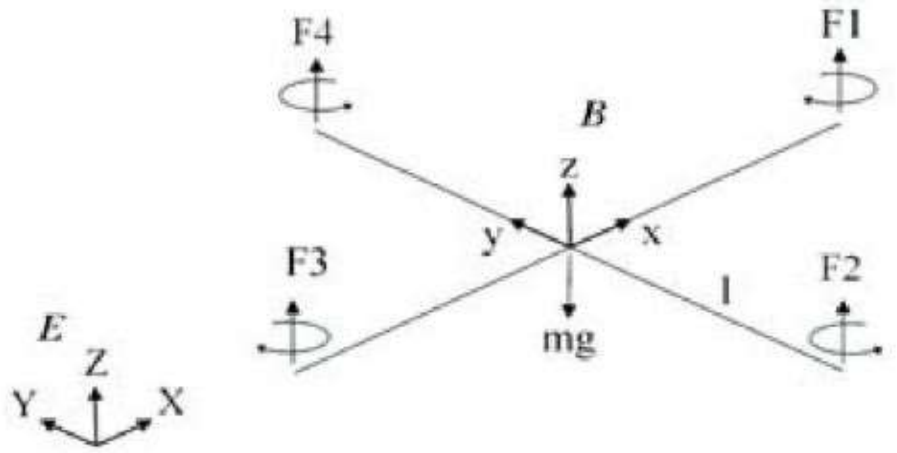


FIGURE 3.2. Frames of reference.

In fig 3.2, E represents the Earth frame of reference, B represents the quadcopter body frame of reference, F1, F2, F3, F4 represent torque and thrust generated by motors, mg is the effect of gravity on quadcopter and L represents distance from the center of quadcopter to motor axis [7].

Kinematics

The kinematics for any point of frame is expressed in fixed frame as shown in Equations 1, 2 and 3.

$$r_x = (cos\psi cos\theta)x + (cos\psi sin\theta sin\phi - sin\psi cos\phi)y + (cos\psi sin\theta cos\phi + sin\psi sin\phi)z \tag{1}$$

$$r_y = (sin\psi cos\theta)x + (sin\psi sin\theta cos\phi + cos\psi cos\phi)y + (sin\psi sin\theta cos\phi - cos\psi sin\theta)z \tag{2}$$

$$r_z = (-sin\theta)x + (cos\theta sin\phi)y + (cos\theta cos\phi)z \tag{3}$$

Dynamics

The dynamics of quadcopter depends on speed of motors and thrust coefficient. The quadcopter is controlled by varying speed of each motor individually. The dynamics of quadcopter are represented in Equations 4, 5, 6 and 7.

$$U_1 = b(W_1^2 + W_2^2 + W_3^2 + W_4^2) \tag{4}$$

$$U_2 = b(-W_2^2 + W_4^2) \tag{5}$$

$$U_3 = b(-W_1^2 + W_3^2) \tag{6}$$

$$U_4 = b(-W_1^2 + W_2^2 - W_3^2 + W_4^2) \tag{7}$$

(W_1, W_2, W_3, W_4) are the angular rate of propellers, b is the thrust coefficient and (U_1, U_2, U_3, U_4) are speed of the motors.

Moment of Inertia

The Moment of inertia of a quadcopter depends on distribution of mass of body with respect to its axis of rotation. Moment of inertia is represented by a unit matrix. Moment of inertia for x and y axis are equal for a quadcopter system and moment of inertia matrix is shown in Equation 8.

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{8}$$

The quadcopter moment of inertia is calculated using Equations 9 and 10, respectively,

$$I_{xx} = I_{yy} = \left(\frac{2}{3} m_h {r_h}^2\right) + 2\left(\frac{1}{12} m_r (3r^2 + h^2) + m_r l^2\right) \tag{9}$$

$$I_{zz} = 4\left(\frac{1}{2} m_r r^2 + m_r l^2\right) \tag{10}$$

(I_{xx}, I_{yy}, I_{zz}) are the moment of inertia of x, y and z axis in kg-m^2, (m_r) is mass of the motor in kg, (r) is radius of the motor in m, (h) is height of the motor in m, (m_h) is mass of the quadcopter hub in kg, (r_h) is radius of the central hub in m and (l) is length of the quadcopter arm in m.

Equations of Motion

Equations of motion is derived using the Euler-Lagrangian formalism. Considering quadcopter aerodynamic forces and moments, the equations of motion is derived and represented in Equations 11, 12 and 13.

$$\ddot{\phi} = \frac{1}{I_{xx}}\left(I_{yy} - I_{zz}\right)qr + J_r q(W_1 - W_2 + W_3 - W_4) + lb(W_4^2 - W_2^2) \quad (11)$$

$$\ddot{\theta} = \frac{1}{I_{yy}}(I_{zz} - I_{xx})pr + J_r p(W_1 - W_2 + W_3 - W_4) + lb(W_1^2 - W_3^2) \quad (12)$$

$$\ddot{\psi} == \frac{1}{I_{zz}}\left(I_{xx} - I_{yy}\right) + d(W_1^2 - W_2^2 + W_3^2 - W_4^2) \quad (13)$$

$(\ddot{\phi}, \ddot{\theta}, \ddot{\psi})$ are the angular acceleration of quadcopter, (p, q, r) are the angular rates of roll, pitch and yaw angles, (W_1, W_2, W_3, W_4) are the angular rate of propellers, b is the thrust coefficient, d is the drag coefficient, l is length of the quadcopter from the center of gravity, (I_{xx}, I_{yy}, I_{zz}) are the moment of inertia of x, y and z axis and J_z is the moment of inertia of motors.

Thrust Coefficient

Thrust coefficient is an aerodynamic factor of quadcopter and depends on the angular rate of propeller and mass of the quadcopter [9]. Thrust coefficient is calculated while hovering and is calculated using the Equation 14, respectively:

$$b = \frac{mg}{4\Omega_o^2} \quad (14)$$

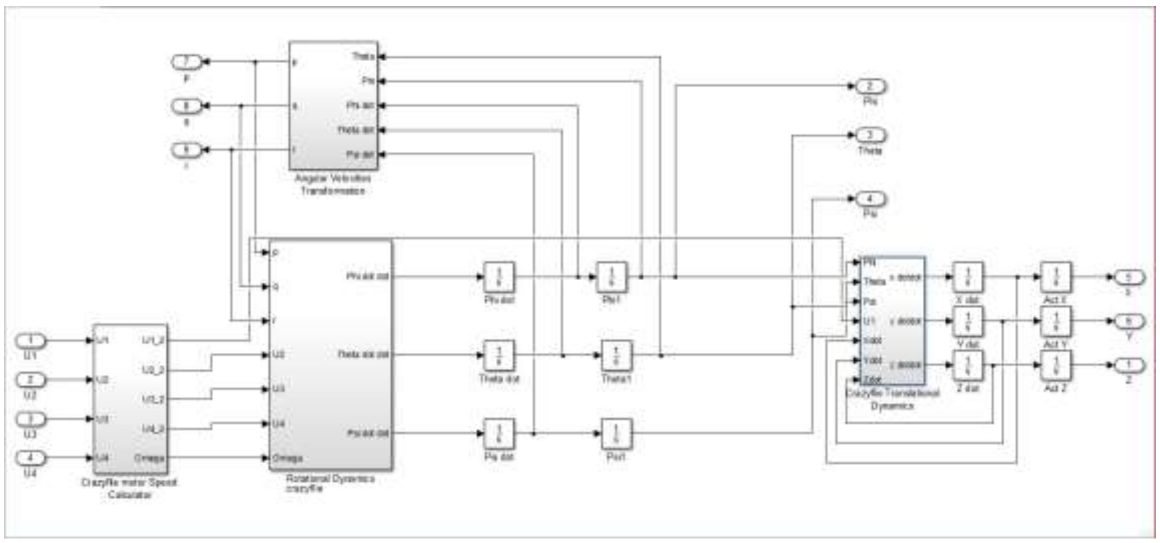
b is the thrust coefficient, m is mass of the quadcopter, g is gravity and Ω is angular rate of quadcopter propellers.

CHAPTER 4

SIMULATIONS

Crazyflie Mathematical Model in Simulink

A mathematical model is developed for Crazyflie in Simulink. The mathematical model emulates Crazyflie behavior with its system dynamics equations. The model is designed with a position controller, altitude controller, attitude controller and PID rate controller. Crazyflie system dynamics includes motor speed calculator, rotational dynamics, translational dynamics and angular velocities as shown in fig 4.1 [10].



Angular Velocities Transformation
Crazyflie motor Speed Calculator
Rotational Dynamics crazyflie
Crazyflie Translational Dynamics

FIGURE 4.1. Crazyflie system dynamics model.

The motor speed calculator block configures quadrotor system with a "+" configuration and generates speed according to requirement of the system. Generated speed is calculated based on the thrust force coefficient, drag force coefficient and moment of inertia of speed. Newly calculated speed feeds rotational dynamics block and translational dynamics block.

The rotational dynamics block is modeled based on moment of inertia of all the three x, y, z axes including motor speed and generates angular acceleration $\ddot{\phi}, \ddot{\theta}, \ddot{\psi}$. Angular velocity is fed into angular velocity transformation block and rotational angles are fed into translational dynamics. The rotational dynamics block is modeled using Equations 15, 16 and 17.

$$\ddot{\phi} = \left(\frac{I_{yy}-I_{zz}}{I_{xx}}\right) qr - \left(\frac{I_p}{I_{xx}}\right) q\Omega + \left(\frac{l}{I_{xx}}\right) U_2 \tag{15}$$

$$\ddot{\theta} = \left(\frac{I_{zz}-I_{xx}}{I_{yy}}\right) pr + \left(\frac{I_p}{I_{yy}}\right) p\Omega + \left(\frac{l}{I_{yy}}\right) U_3 \tag{16}$$

$$\ddot{\psi} = \left(\frac{I_{xx}-I_{yy}}{I_{zz}}\right) pq + \left(\frac{1}{I_{zz}}\right) U_4 \tag{17}$$

$(\ddot{\phi}, \ddot{\theta}, \ddot{\psi})$ are the angular acceleration of Roll, Pitch and Yaw. (p, q, r) are the angular rates of Roll, Pitch and Yaw. (U_2, U_3, U_4) are the motor speeds of x-axis, y-axis and z-axis. (I_{xx}, I_{yy}, I_{zz}) are the moment of inertia of x, y and z axes and Ω is angular rate of motors [10]. The block diagram of Crazyflie rotational dynamics is illustrated in fig 4.2.

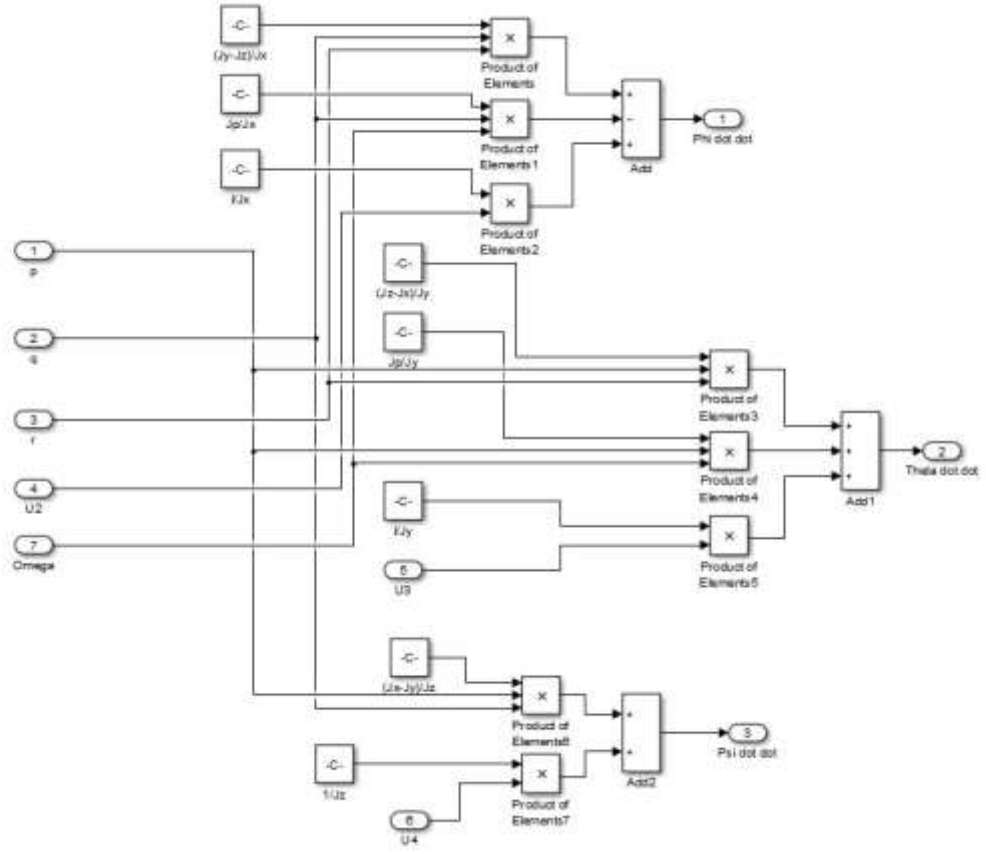


FIGURE 4.2. Crazyflie rotational dynamics model.

The rotational angles ϕ, θ, ψ and the motor speed are fed in to the translational dynamics block. The translational dynamics is mainly responsible for the position control of the quadcopter system. This block calculates the position of the quadcopter based on the translational drag force coefficient, mass, gravity and the speed. The translational dynamics is modeled using Equations 18, 19 and 20.

$$\ddot{X} = -(\cos\phi \sin\theta \cos\psi + \sin\phi \sin\psi)\frac{U_1}{m} - \dot{X}_d \frac{k_{dx}}{m} \tag{18}$$

$$\ddot{Y} = -(\cos\phi \sin\theta \sin\psi - \sin\phi \cos\psi)\frac{U_1}{m} - \dot{Y}_d \frac{k_{dy}}{m} \tag{19}$$

$$\ddot{Z} = -(\cos\phi \cos\theta)\frac{U_1}{m} - \dot{Z}\frac{k_{dz}}{m} + g \tag{20}$$

$(\ddot{X}, \ddot{Y}, \ddot{Z})$ are the angular positions of the Crazyflie translational dynamics, (k_{dx}, k_{dy}, k_{dz}) are the translational drag coefficients of the system, (m) is the mass of the quadcopter, (g) is the gravity and (U_1) is the thrust [10]. Crazyflie translational dynamics model is illustrated in fig 4.3.

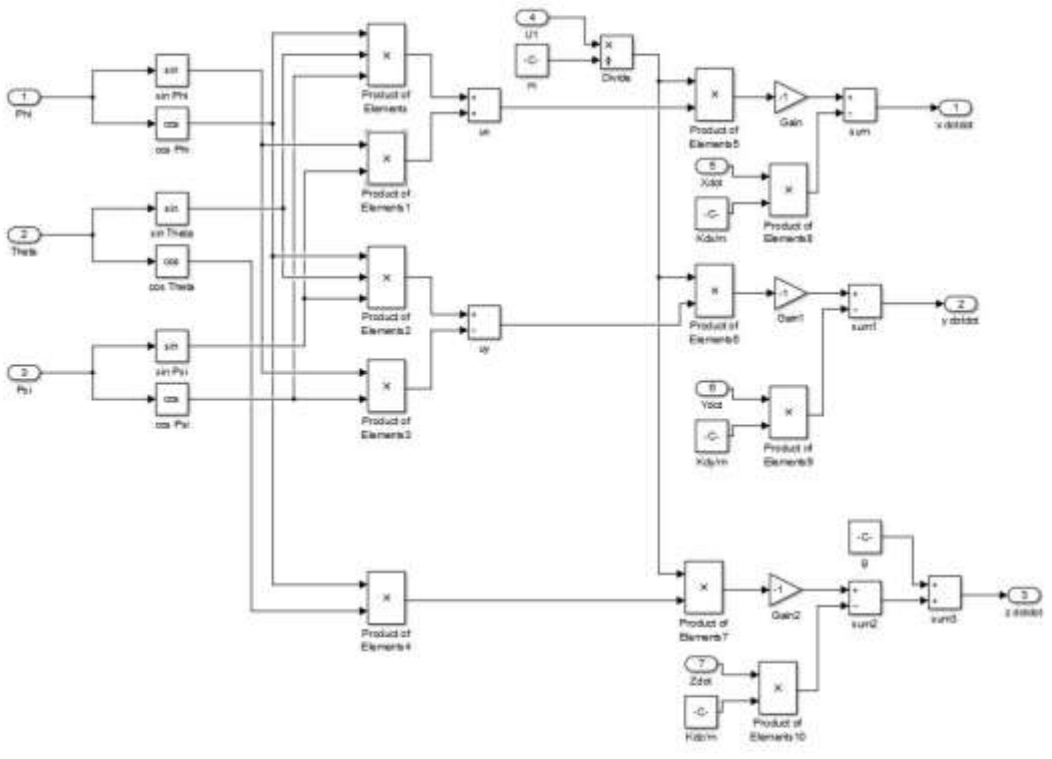


FIGURE 4.3. Crazyflie translational dynamics model.

The Angular velocity transformation block is the basic unit of the quadcopter system that is controlled by the angular rate controller. The angular velocity transformation block is modeled using Equations 21, 22 and 23.

$$p = \dot{\phi} - \dot{\psi}\sin\theta \qquad (21)$$

$$q = \dot{\theta}\cos\phi + \dot{\psi}\sin\phi\cos\theta \qquad (22)$$

$$r = \dot{\theta}\sin\phi + \dot{\psi}\cos\phi\cos\theta \qquad (23)$$

(p, q, r) are the angular rates, (ϕ, θ, ψ) the rotational angles and $(\ddot{\phi}, \ddot{\theta}, \ddot{\psi})$ are the angular velocity. The angular velocity model is illustrated in fig 4.4.

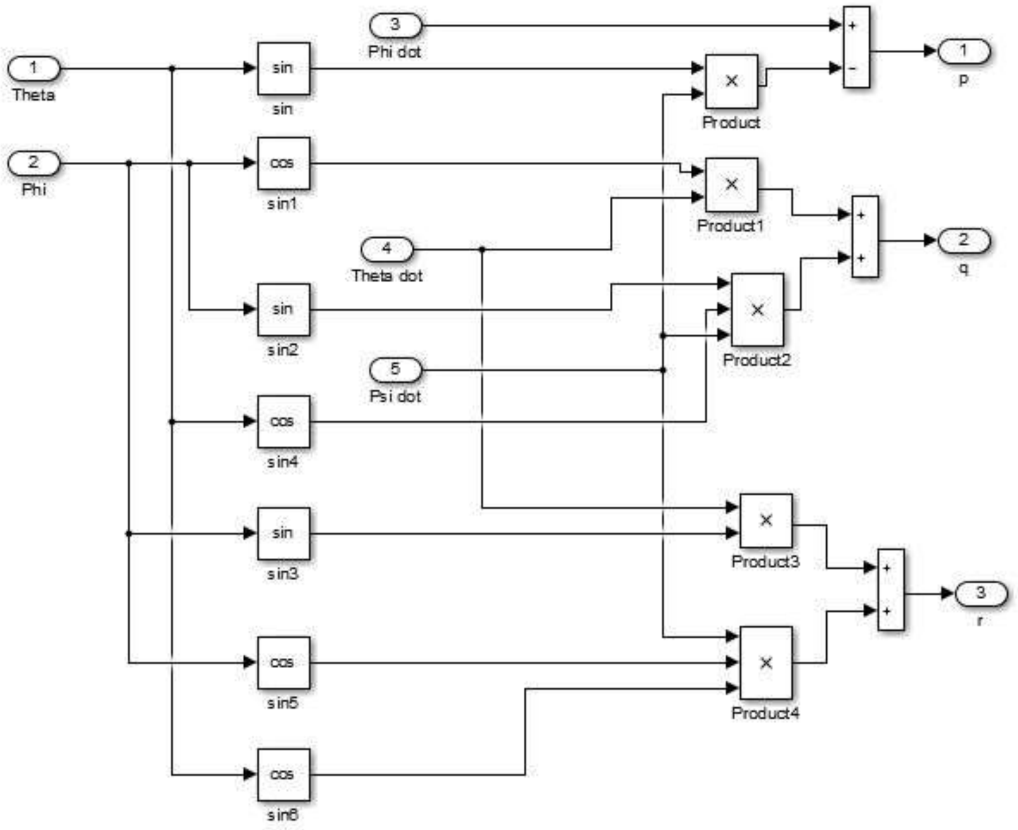


FIGURE 4.4. Crazyflie angular velocity model.

Controllers are designed around the above quadcopter system dynamics block. The quadcopter dynamics system is designed on non-linear approach and hence, the system must be linearized to tune controllers. Each equation from the dynamic system is treated individually and linearized around a set point. Consider a rotational dynamic system generating Phi, Theta and Psi. The system dynamics to obtain Theta is chosen (as shown in fig 4.5).

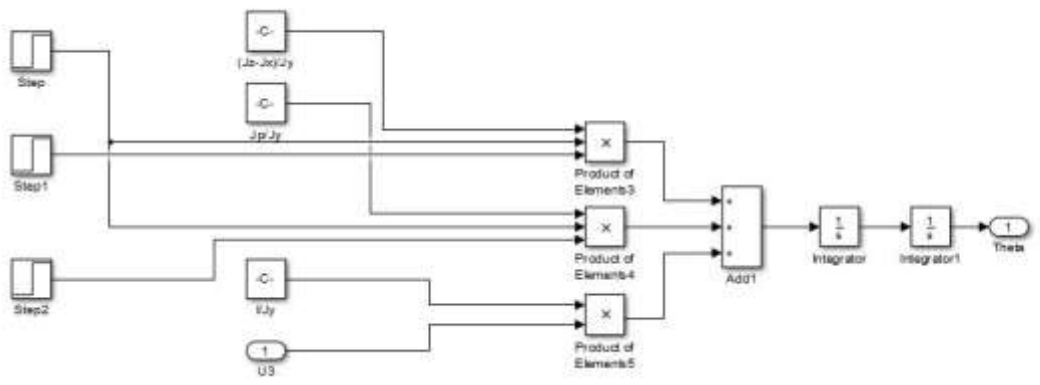


FIGURE 4.5. Linearizing rotational dynamics system for variable theta.

The above system is linearized using the following commands to obtain the transfer function and is as follows.

```
[num,den]=dlinmod('Theta_rotational');

sys=tf(num,den);
```

Result:

```
sys =


    5.6 s + 5.6
  -------------
  s^2 - 2 s + 1

Continuous-time transfer function.
```

A closed loop system was built around the above transfer function for rotational system Theta parameter. The transfer function is treated as a plant and PI controller is tuned until a steady state response is obtained (shown in fig 4.6 and fig 4.7). The control gain values are Kp=5, Ki=2.5.

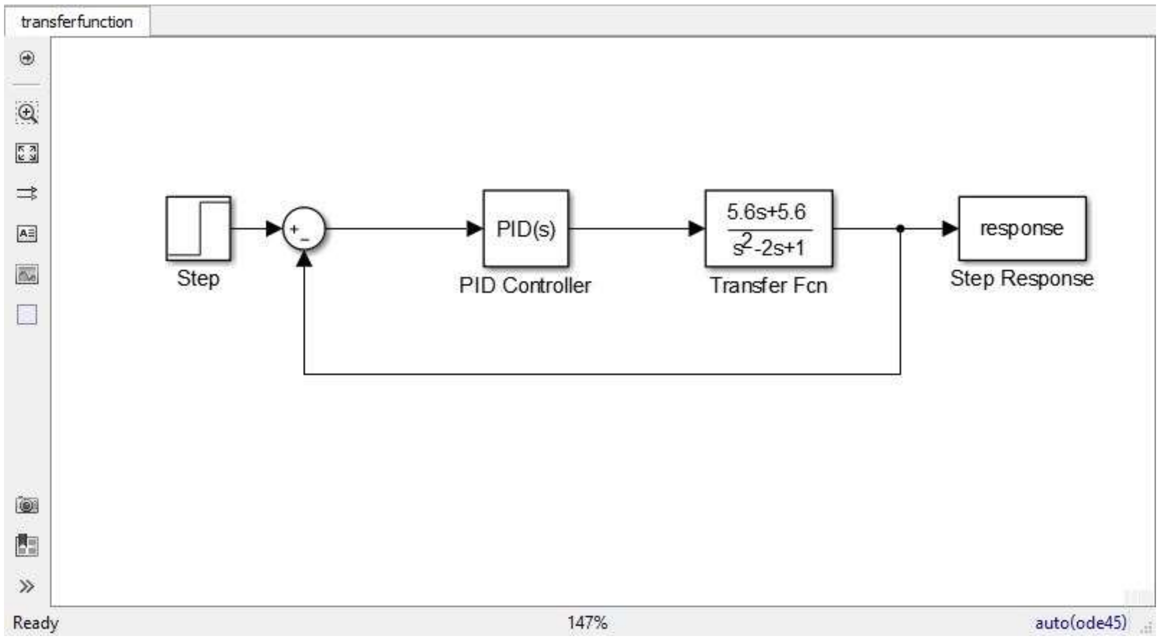


FIGURE 4.6. Closed loop system of the rotational system variable theta.

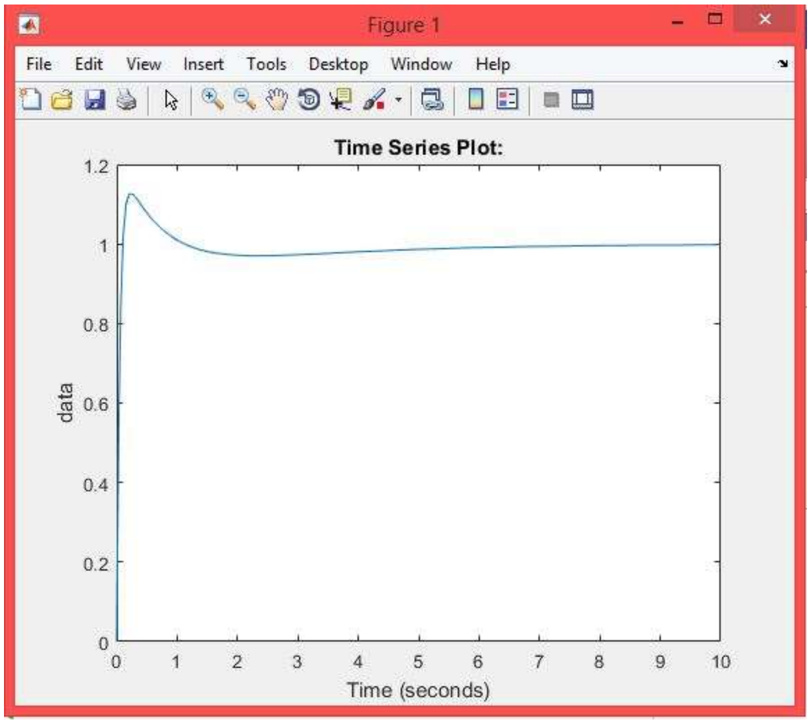


FIGURE 4.7. Steady state response of variable theta in rotational system.

Similarly, Phi, Psi in rotational and X, Y, Z in translational systems are individually linearized around a set point to get transfer function and PI controller is tuned to stabilize the system. The controller gains values for the translational and rotational system are shown in Table 1.

TABLE 1. Controller Gains

Dynamics System	Kp	Ki	Kd
Phi	5	2.5	0
Theta	5	2.5	0
Psi	8.2	4.3	0
X	.3	0.15	0
Y	.3	0.15	0
Z	.5	0.1	0
p	70	0	0
q	70	0	0
r	25	50	0

Thus, attitude (Phi, Theta, Psi), altitude (Z) and position (X, Y) controllers are designed using the tuned controller gain values (Table 1). The Crazyflie physical parameters are measured, calculated and utilized in the mathematical modeling of the system. Crazyflie system physical parameters are shown in Table 2.

TABLE 2. Crazyflie Physical Parameters

Parameters	Value	Unit
mass	0.022	kg
Length	0.042	m
thickness of arm	0.003	m
Moment of Inertia , I_{xx}	9.1914e-06	kg-m^2
Moment of Inertia , I_{yy}	9.1914e-06	kg-m^2
Moment of Inertia , I_{zz}	2.2800e-05	kg-m^2
Moment of Inertia of motor	2.97e^{-5}	kg-m^2
Thrust force coefficient, K_t	2.75e^{-11}	kg-m
Drag force coefficient, K_d	1e^{-9}	kg-m^2

Crazyflie mathematical model is designed with feedback altitude, attitude, angular rate and position controllers. The angular rate controller is applied to a quadcopter to control the angular velocities (p, q, r) in order to achieve controlled heading and avoid drifts. Attitude controller is implemented around angular rate controller to maintain orientation of the inertial angles (ϕ, θ, ψ) with respect to the body frame. Altitude controller is deigned to maintain the Z position or altitude of the quadcopter during its flight. A position controller controls the positions (X, Y) with respect to the desired positions. All the feedback controllers are equipped with a PID controller. PID means Proportional - Integral – Derivative. The PID controller calculates state error between the set point and the process variables. The non-negative gains $K_p, K_i,, K_d$ minimizes the error values $e(t)$ according to Equation 24. The block diagram of Equation 24 is illustrated in fig 4.8.

$$u(t) = K_p e(t) + K_i \int_0^t e(t) + K_d \frac{de(t)}{dt} \tag{24}$$

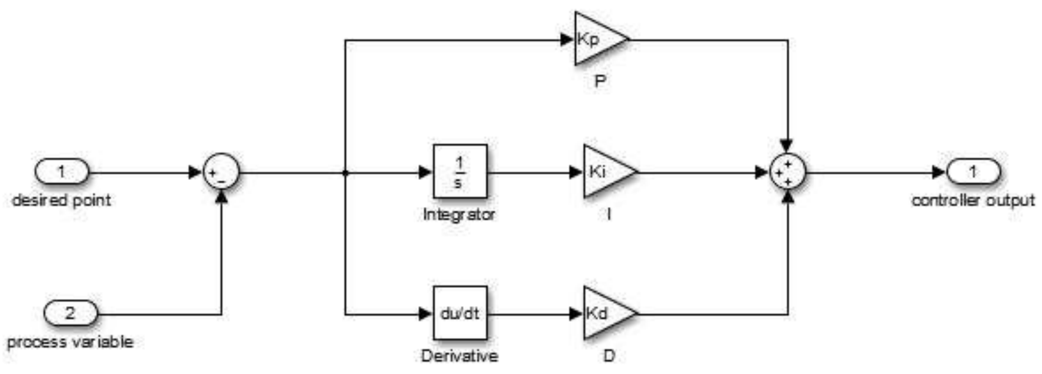


FIGURE 4.8. PID block diagram.

Position controller in the Crazyflie mathematical model is designed to obtain a position hold for X and Y axes to a desired X and Y position. The Z axis is controlled by altitude controller to obtain desired Z position or altitude. And hence, a position controller is designed for the control of X and Y axes. The controllers are given a step response to check the characteristics of the entire system. The X and Y desired position values are calculated using Equations 25 and 26.

$$X_d = X_d \cos\psi + Y_d \sin\psi \tag{25}$$

$$Y_d = Y_d \cos\psi + X_d \sin\psi \tag{26}$$

The above calculated desired velocity is fed into PID controller. The current X and Y values are given as a feedback to the PID block to generate error and the error is fed into a PID block. The PID values are tuned until X and Y positions responses are critically damped and steady state is attained. Crazyflie mathematical model with angular rate, altitude, attitude and position controllers are shown in fig 4.9.

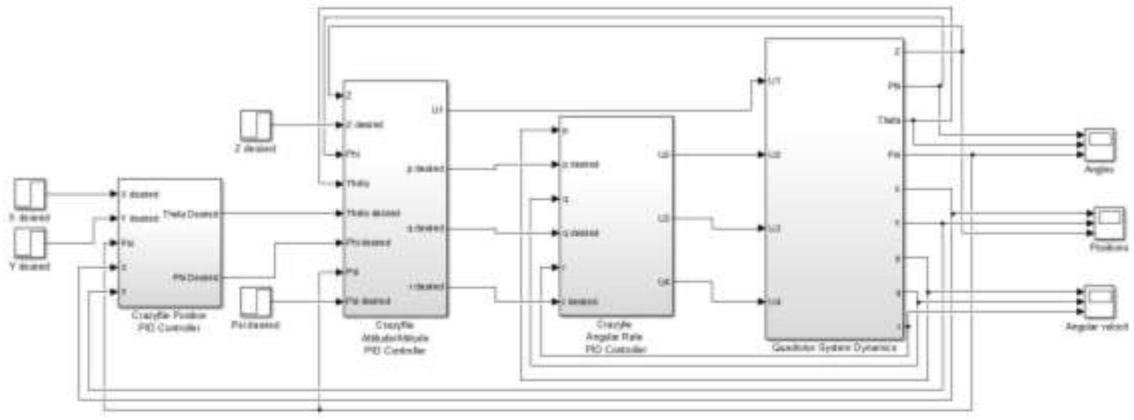


FIGURE 4.9. Crazyflie mathematical model.

Controller Implementation in real time Simulink model

The Crazyflie nano quadcopter is interfaced with Simulink using an open-source client library “libcflie”. Libcflie is a communication protocol that connects the Crazyflie firmware to Simulink using “S-function” block. The S-function includes a set of header files that defines the physical layout of Crazyflie program data. The header files are customized according to the requirements to design the controllers. CCrazyflie.h, CCrazyRadio.h, CTOC.h and CCRTPPacket.h are the dependencies for S-function. In CCrazyflie.h, the required set of sensor loggings are enabled and set points are sent to the controller.

CCrazyflie.h

```
#ifndef __C_CRAZYFLIE_H__
#define __C_CRAZYFLIE_H__
#define NSEC_PER_SEC 1000000000L


// System
#include <cmath>


// Private
#include "CCrazyRadio.h"
#include "CTOC.h"
enum State {
  STATE_ZERO = 0,
  STATE_READ_PARAMETERS_TOC = 1,
  STATE_READ_LOGS_TOC = 2,
  STATE_START_LOGGING = 3,
  STATE_ZERO_MEASUREMENTS = 4,
  STATE_NORMAL_OPERATION = 5
};
```

```
class CCrazyflie {
 private:
  // Variables
  int m_nAckMissTolerance;
  int m_nAckMissCounter;
  CCrazyRadio *m_crRadio;
  /*! \brief The current thrust to send as a set point to the
      copter. */
  int m_nThrust;
  /*! \brief The current roll to send as a set point to the copter. */
  float m_fRoll;
  /*! \brief The current pitch to send as a set point to the
      copter. */
  float m_fPitch;
  /*! \brief The current yaw to send as a set point to the copter. */
  float m_fYaw;
  /*! \brief The current desired control set point (position/yaw to
      reach) */
float m_fMaxAbsRoll;
  /*! \brief Maximum absolute value for the pitch that will be sent to
      the copter. */
  float m_fMaxAbsPitch;
  /*! \brief Maximum absolute value for the yaw that will be sent to
```

```
    the copter. */
float m_fMaxYaw;
/*! \brief Maximum thrust that will be sent to the copter. */
int m_nMaxThrust;
/*! \brief Minimum thrust that will be sent to the copter. */
int m_nMinThrust;
double m_dSendSetpointPeriod;
double m_dSetpointLastSent;
bool m_bSendsSetpoints;
CTOC *m_tocParameters;
CTOC *m_tocLogs;
enum State m_enumState;
// Functions
bool readTOCParameters();
bool readTOCLogs();
bool sendSetpoint(float fRoll, float fPitch, float fYaw, short sThrust);
void disableLogging();
void enableStabilizerLogging();
void enableGyroscopeLogging();
void enableAccelerometerLogging();
void disableStabilizerLogging();
void disableGyroscopeLogging();
void disableAccelerometerLogging();
```

```
 void enableBatteryLogging();
 void disableBatteryLogging();
 bool startLogging();
 bool stopLogging();
 void enableMagnetometerLogging();
 void disableMagnetometerLogging();
 void enableAltimeterLogging();
 void disableAltimeterLogging();
 double currentTime();
public:
 CCrazyflie(CCrazyRadio *crRadio);
  ~CCrazyflie();
 void setThrust(int nThrust);
 int thrust();
 void setRoll(float fRoll);
 float roll();
 void setPitch(float fPitch);
 float pitch();
 void setYaw(float fYaw);
 float yaw();
 bool cycle();
 bool copterInRange();
 bool isInitialized();
```

```
  void setSendSetpoints(bool bSendSetpoints);
  bool sendsSetpoints();
  double sensorDoubleValue(std::string strName);
  double batteryLevel();
  float accX();
  float accY();
  float accZ();
  float accZW();
  float asl();
  float aslLong();
  float temperature();
  float pressure();
  float gyroX();
  float gyroY();
  float gyroZ();
  float batteryState();
  float magX();
  float magY();
  float magZ();
};
```

The CCrazyRadio.h detects USB dongle Crazyradio, starts the Crazyradio, maintains connection, sends and receives data using Crazy Radio Transfer protocol while communicating with Crazyflie. CCRTPpacket.h is responsible for maintaining and processing data, data length related to Crazyradio and Crazyflie communication during the Crazy Radio Transfer Protocol. The logged Crazyflie parameters are stored in table of contents in its respective identities using CTOC.h.

CCrazyRadio.h

```
#ifndef __C_TOC_H__
#define __C_TOC_H__
// System
#include <list>
#include <string>
#include <cstdlib>
#include <iostream>
// Private
#include "CCrazyRadio.h"
#include "CCRTPPacket.h"
/*! \brief Storage element for logged variable identities */
struct TOCElement {
  /*! \brief The numerical ID of the log element on the copter's
      internal table */
  int nID;
  /*! \brief The (ref) type of the log element */
  int nType;
  /*! \brief The string group name of the log element */
  std::string strGroup;
  /*! \brief The string identifier of the log element */
  std::string strIdentifier;
  bool bIsLogging;
  double dValue;
};
struct LoggingBlock {
  std::string strName;
  int nID;
  double dFrequency;
  std::list<int> lstElementIDs;
};
class CTOC {
 private:
  int m_nPort;
  CCrazyRadio *m_crRadio;
  int m_nItemCount;
  std::list<struct TOCElement> m_lstTOCElements;
  std::list<struct LoggingBlock> m_lstLoggingBlocks;
  bool requestInitialItem();
  bool requestItem(int nID, bool bInitial);
  bool requestItem(int nID);
  bool processItem(CCRTPPacket* crtpItem);
  CCRTPPacket* sendAndReceive(CCRTPPacket* crtpSend, int nChannel);
```

```
public:
 CTOC(CCrazyRadio* crRadio, int nPort);
 ~CTOC();
 bool sendTOCPointerReset();
 bool requestMetaData();
 bool requestItems();
 struct TOCElement elementForName(std::string strName, bool& bFound);
 struct TOCElement elementForID(int nID, bool &bFound);
 int idForName(std::string strName);
 int typeForName(std::string strName);
 // For loggable variables only
 bool registerLoggingBlock(std::string strName, double dFrequency);
 bool unregisterLoggingBlock(std::string strName);
 struct LoggingBlock loggingBlockForName(std::string strName, bool& bFound);
 struct LoggingBlock loggingBlockForID(int nID, bool& bFound);
 bool startLogging(std::string strName, std::string strBlockName);
 bool stopLogging(std::string strName);
 bool isLogging(std::string strName);
 double doubleValue(std::string strName);
 bool enableLogging(std::string strBlockName);
 void processPackets(std::list<CCRTPPacket*> lstPackets);
 int elementIDinBlock(int nBlockID, int nElementIndex);
 bool setFloatValueForElementID(int nElementID, float fValue);
```

```
  bool addElementToBlock(int nBlockID, int nElementID);

  bool unregisterLoggingBlockID(int nID);

}
```

CCRTPPacket.h

```
#ifndef __C_CRTP_PACKET_H__

#define __C_CRTP_PACKET_H__

// System

#include <cstring>

/*! \brief Class to hold and process communication-related data for

  the CRTProtocol */

class CCRTPPacket {

 private:

  char *m_cData;

  /*! \brief The length of the data pointed to by m_cData */

  int m_nDataLength;

  /*! \brief The copter port the packet will be delivered to */

  int m_nPort;

  /*! \brief The copter channel the packet will be delivered to */

  int m_nChannel;

  bool m_bIsPingPacket;

  void basicSetup();

  void clearData();

 public:
  CCRTPPacket(int nChannel);

  CCRTPPacket(char *cData, int nDataLength, int nChannel);

  CCRTPPacket(char cData, int nPort);

   ~CCRTPPacket();

  void setData(char *cData, int nDataLength);

  char *data();

  int dataLength();

  char *sendableData();

  int sendableDataLength();

  void setPort(int nPort);

  int port();

  void setChannel(int nChannel);

  int channel();

  void setIsPingPacket(bool bIsPingPacket);

  bool isPingPacket();

};

#endif /* __C_CRTP_PACKET_H__ */
```

CTOC.h

```
#ifndef __C_TOC_H__

#define __C_TOC_H__

// System

#include <list>

#include <string>
```

```
#include <cstdlib>

#include <iostream>

// Private

#include "CCrazyRadio.h"

#include "CCRTPPacket.h"

/*! \brief Storage element for logged variable identities */

struct TOCElement {

  /*! \brief The numerical ID of the log element on the copter's

      internal table */

  int nID;

  /*! \brief The (ref) type of the log element */

  int nType;

  /*! \brief The string group name of the log element */

  std::string strGroup;

  /*! \brief The string identifier of the log element */

  std::string strIdentifier;

  bool bIsLogging;

  double dValue;

};

struct LoggingBlock {

  std::string strName;

  int nID;

  double dFrequency;
  std::list<int> lstElementIDs;
};
class CTOC {
 private:
  int m_nPort;
  CCrazyRadio *m_crRadio;
  int m_nItemCount;
  std::list<struct TOCElement> m_lstTOCElements;
  std::list<struct LoggingBlock> m_lstLoggingBlocks;
  bool requestInitialItem();
  bool requestItem(int nID, bool bInitial);
  bool requestItem(int nID);
  bool processItem(CCRTPPacket* crtpItem);
  CCRTPPacket* sendAndReceive(CCRTPPacket* crtpSend, int nChannel);
 public:
  CTOC(CCrazyRadio* crRadio, int nPort);
  ~CTOC();
  bool sendTOCPointerReset();
  bool requestMetaData();
  bool requestItems();
  struct TOCElement elementForName(std::string strName, bool& bFound);
  struct TOCElement elementForID(int nID, bool &bFound);
  int idForName(std::string strName);
  int typeForName(std::string strName);
  // For loggable variables only
  bool registerLoggingBlock(std::string strName, double dFrequency);
  bool unregisterLoggingBlock(std::string strName);
  struct LoggingBlock loggingBlockForName(std::string strName, bool& bFound);
  struct LoggingBlock loggingBlockForID(int nID, bool& bFound);
  bool startLogging(std::string strName, std::string strBlockName);
  bool stopLogging(std::string strName);
  bool isLogging(std::string strName);
  double doubleValue(std::string strName);
  bool enableLogging(std::string strBlockName);
  void processPackets(std::list<CCRTPPacket*> lstPackets);
  int elementIDinBlock(int nBlockID, int nElementIndex);
  bool setFloatValueForElementID(int nElementID, float fValue);
  bool addElementToBlock(int nBlockID, int nElementID);
  bool unregisterLoggingBlockID(int nID);
};
#endif /* __C_TOC_H__ */
```

In S-function, the number of input and output ports are initialized and a function to start the Crazyradio is called with the Crazyradio address. The input port signal values are read with the help of CCrazyflie.h and respective commands are sent using CCrazyRadio.h, CCRTPPacket.h to the Crazyflie through Crazyradio and sensor loggings are obtained using CCTOC.h.

S-Function

```
    #include <iostream>
    #include "mex.h"
    #include "CCrazyflie.h"
    using namespace std;
    #define S_FUNCTION_LEVEL 2
    #define S_FUNCTION_NAME sfun_cflie_simple
    #include "simstruc.h"
    #define IS_PARAM_DOUBLE(pVal) (mxIsNumeric(pVal) && !mxIsLogical(pVal)
&&\
    !mxIsEmpty(pVal) && !mxIsSparse(pVal) && !mxIsComplex(pVal) &&
mxIsDouble(pVal))
    /*===================*
     *S-function methods *
     *===================*/
    #define MDL_CHECK_PARAMETERS
    #if defined(MDL_CHECK_PARAMETERS)  && defined(MATLAB_MEX_FILE)
    /*
     * Check to make sure that each parameter is 1-d and positive
     */
    static void mdlCheckParameters(SimStruct *S)
```

```
{
   const mxArray *pVal0 = ssGetSFcnParam(S,0);
   if ( !IS_PARAM_DOUBLE(pVal0)) {
      ssSetErrorStatus(S, "Parameter to S-function must be a double scalar");
      return;
   }
}
#endif
/* Function: mdlInitializeSizes
=========================================
 * Abstract:
 *
 *
 */
static void mdlInitializeSizes(SimStruct *S)
{
   ssSetNumSFcnParams(S,1); //Numer of expected parameters
   #if defined(MATLAB_MEX_FILE)
   if (ssGetNumSFcnParams(S) == ssGetSFcnParamsCount(S)) {
      mdlCheckParameters(S);
      if (ssGetErrorStatus(S) != NULL) {
         return;
      }
```

```
  } else {
      return; /* Parameter mismatch will be reported by Simulink */
  }
  #endif

  ssSetSFcnParamTunable(S,0,0);
  ssSetNumContStates(S,0);
  ssSetNumDiscStates(S,0);

  int NUM_INPUT_PORTS = 4;
  if(!ssSetNumInputPorts(S,NUM_INPUT_PORTS)) return;

for (int i=0; i<NUM_INPUT_PORTS; i++) {
      ssSetInputPortWidth(S,i,1);
      ssSetInputPortDirectFeedThrough(S, i, 1);
      //ssSetInputPortRequiredContiguous(S,i,1);
  }
  int NUM_OUTPUT_PORTS = 1;
  if(!ssSetNumOutputPorts(S,NUM_OUTPUT_PORTS)) return;
  ssSetOutputPortWidth(S,0,4);
  ssSetOutputPortDataType(S,0,SS_DOUBLE);
  ssSetNumSampleTimes(S,1);
  ssSetNumRWork(S,0);
```

```
    ssSetNumIWork(S,0);

    ssSetNumPWork(S,2); // reserve element in the pointers vector to store a C++ object

    ssSetNumModes(S,0);

    ssSetNumNonsampledZCs(S,0);

    ssSetSimStateCompliance(S, USE_CUSTOM_SIM_STATE);

    ssSetOptions(S,0);

}

/* Function: mdlInitializeSampleTimes
=====================================

 * Abstract:

 *

 *

 */

static void mdlInitializeSampleTimes(SimStruct *S)

{

    ssSetSampleTime(S,0, mxGetScalar(ssGetSFcnParam(S,0)));

    ssSetOffsetTime(S,0, 0.0);

    ssSetModelReferenceSampleTimeDefaultInheritance(S);

}

#define MDL_START   /* Change to #undef to remove function */

#if defined(MDL_START)

/* Function: mdlStart
=====================================================
 * Abstract:
 *
 *
 */
static void mdlStart(SimStruct *S)
{
   int_T i;
   int_T nInputPorts = ssGetNumInputPorts(S);
   InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
   CCrazyRadio *crRadio = new CCrazyRadio("radio://0/10/250K");
   ssGetPWork(S)[0] = crRadio;
     if(crRadio->startRadio()){
      mexPrintf("\n I'm insider if");
      CCrazyflie *cflieCopter = new CCrazyflie(crRadio);
                  cflieCopter->setSendSetpoints(true);
      ssGetPWork(S)[1] = cflieCopter;
   }
   else {
      mexPrintf("\n Radio could not be started!");
      ssSetErrorStatus(S, "Radio could not be started!");
   }
}
#endif /* MDL_START */
```

```
/* Function: mdlOutputs
=================================================
 * Abstract:
 *
 *
 */
static void mdlOutputs(SimStruct *S, int_T tid)
{
   double *output = (double *)ssGetOutputPortSignal(S,0);
   //output[0] = 5;
   UNUSED_ARG(tid);
   // Read input port signals
   int_T nInputPorts = ssGetNumInputPorts(S);
   InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
   int_T nu = ssGetInputPortWidth(S,0);
   ssPrintf("Port width: %d\n", nu);
   int_T thrust = *uPtrs[0];
   int_T roll = *uPtrs[1];
   int_T pitch = *uPtrs[2];
   int_T yaw = *uPtrs[3];
   CCrazyflie *Copter = (CCrazyflie *) ssGetPWork(S)[1];
   if(Copter->cycle()){
    // Send command
```

```
                Copter->setThrust(thrust);

                Copter->setRoll(roll);

                Copter->setPitch(pitch);

                Copter->setYaw(yaw);

                ssPrintf("Sending command ---> thrust: %d, roll: %d, pitch: %d, yaw: %d\n",thrust,
roll, pitch, yaw);

                float lon = Copter->gpsLon();

                float lat = Copter->gpsLat();


                // Get sensor data

                int tr = Copter->thrust();


            //Copter->enablebarometertLogging("baro.asl", "baro");


                float asl = Copter->asl();

                float aslL = Copter->aslLong();

                float r = Copter->roll();

                float p = Copter->pitch();

                float y = Copter->yaw();


                output[0] = Copter->roll();

                output[1] = Copter->pitch();

                output[2] = Copter->yaw();
```

```
        output[3] = Copter->asl();


    }
}

/* Function: mdlTerminate
===================================================
 * Abstract:
 *
 *
 */
static void mdlTerminate(SimStruct *S)
{
    CCrazyRadio *radio = (CCrazyRadio *) ssGetPWork(S)[0];
    //CCrazyflie *copter = (CCrazyflie *) ssGetPWork(S)[2];


    delete radio;
    //delete copter;
}

/*==============================*
 * Required S-function trailer *
 *==============================*/
```

```
#ifdef MATLAB_MEX_FILE
#include "simulink.c"      /* MEX-file interface mechanism */
#else
#include "cg_sfun.h"       /* Code generation registration function */
#endif
```

A mex binding file for S-function is generated using “mex” command.

```
mex -v CFLAGS='$CFLAGS -fPIC'-L/Cflie -lcflie -lusb-1.0 sfun_cflie_simple.cpp
```

Thus, a feedback controller is built around the Crazyflie S-function block. The Crazyflie firmware has an in-built angular velocity controller which is called using stabilizer logging function in CCrazyflie.h. In this model, attitude and altitude controllers are implemented. Output variables from the quadcopter system are fed into the controllers. For attitude control roll, pitch and yaw values are fed into the controller block. For altitude control, altimeter value is fed into the controller block and is as shown in fig 4.10.

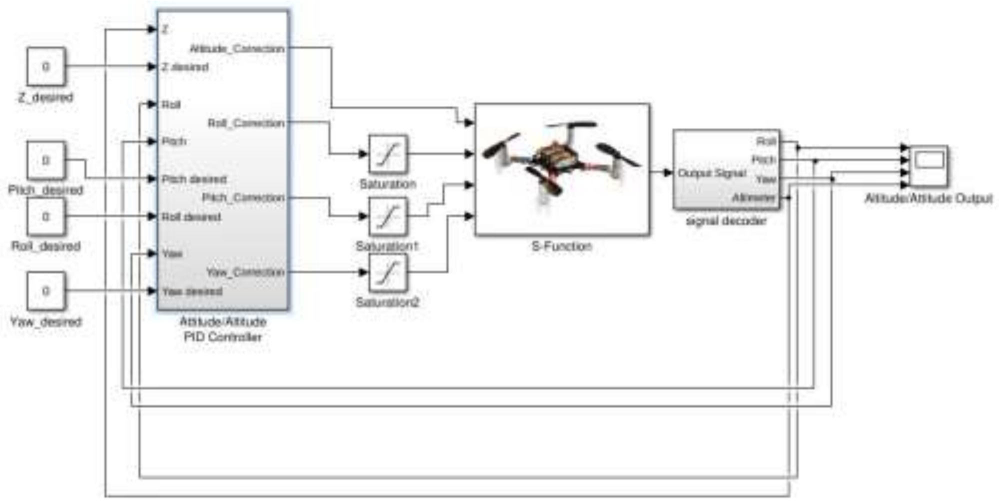


FIGURE 4.10. Attitude/Altitude controller implementation in real time Simulink model.

Desired values are fed into the respective PID blocks. The tuned PID values for attitude and altitude controller from the mathematical model are implemented in real time Simulink model. The output from attitude controller serves as the new input values for roll, pitch and yaw parameters in Crazyflie. The PID controlled output of altitude controller feeds the thrust input for Crazyflie.

CHAPTER 5

RESULTS

The tuned position control of Crazyflie mathematical model shows a steady state response. Also attitude and altitude controller responses are plotted. The roll, pitch, yaw, altitude/Z, X and Y positions are shown in fig 5.1 and 5.2.

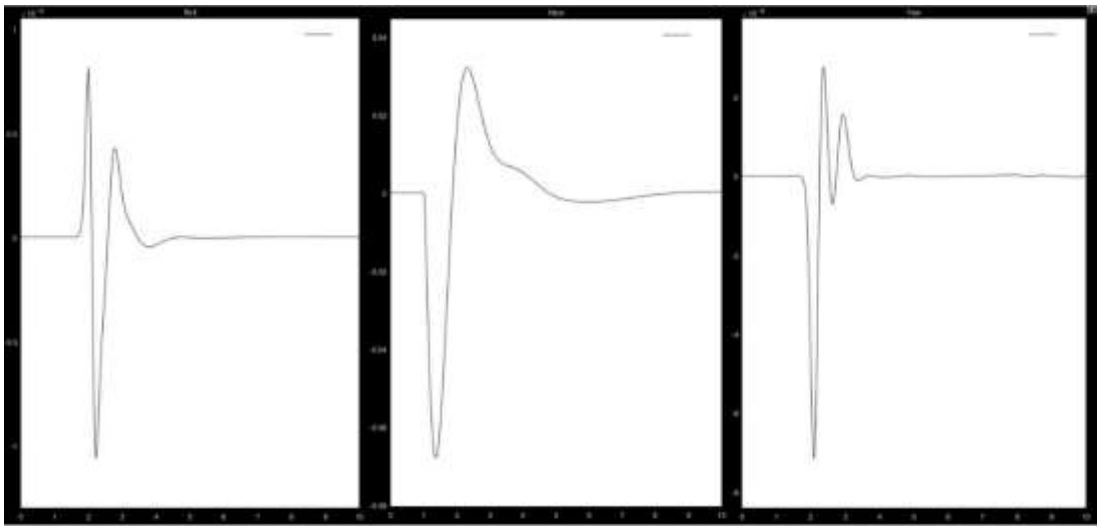


FIGURE 5.1. Steady state responses of roll, pitch and yaw angles from attitude controller.

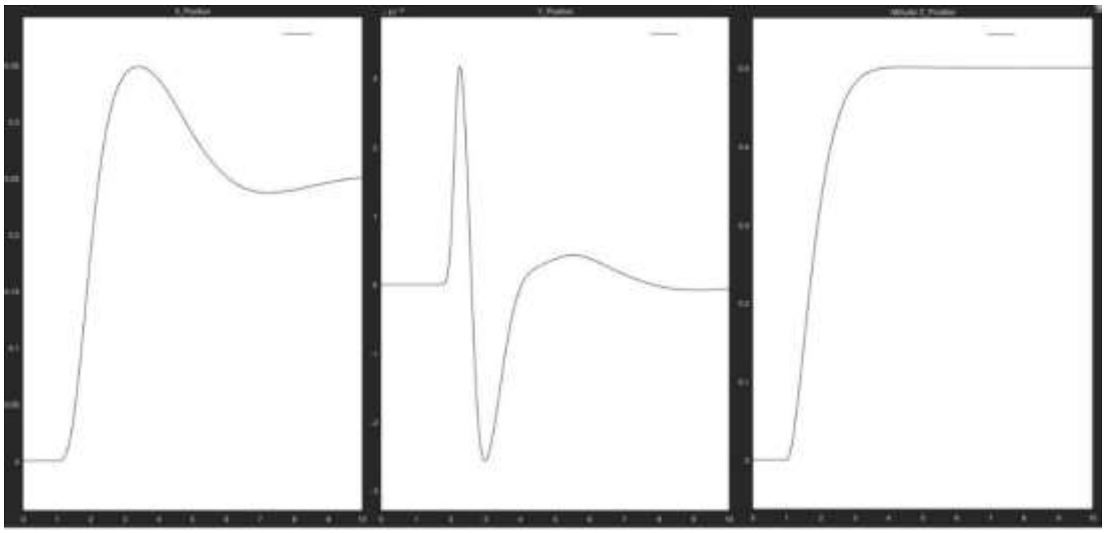


FIGURE 5.2. Steady state response of X position, Y position and Z/altitude.

Autonomous and controlled flight of the Crazyflie is achieved using Simulink. The response from the Crazyflie real time interface Simulink model designed with attitude and altitude controller depicts substantial behavior (as shown in fig 5.3).

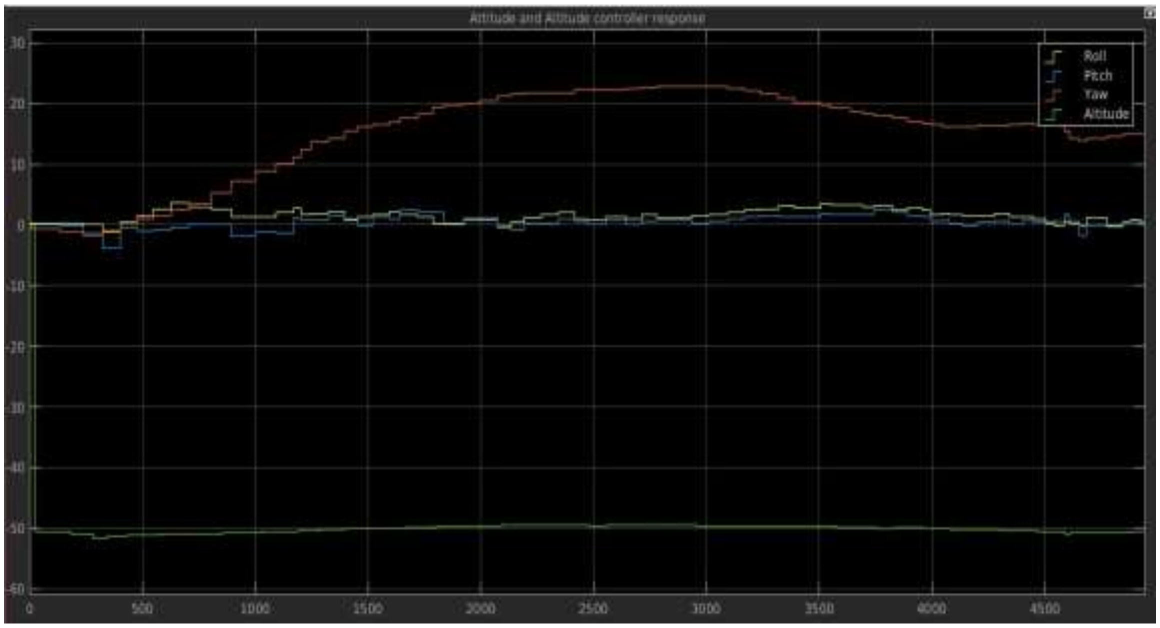


FIGURE 5.3. Attitude and altitude controller response of Crazyflie real time interface with Simulink.

CHAPTER 6

CONCLUSION AND FUTURE IMPLEMENTATION

In the past, implementation of controllers on Crazyflie nano quadcopter was achieved through Firmware and lacked a real time controller. The method presented in this project introduced mathematical modeling of the Crazyflie system dynamics and a new technique of controlling Crazyflie autonomously in real time using Simulink.

The Crazyflie mathematical model exhibits a stable response. The model provides a better understanding of the quadcopter system design and its operations in a simple manner. Tuning of the controllers in the mathematical model proves stability and controllability with the application of the controllers in real time. The binding library for the Simulink real time interface acts as a bridge between the Simulink and the Firmware.

The binding library provides access to only specific parameters to control the hardware. The latency between Simulink and Crazyflie is very high, and this aspect makes it difficult to tune a controller in real mechanical system. Also, the Crazyflie 1.0 version shows its unreliability with upgrade in its firmware.

In future works, position controller could be used by implementing sensors like GPS, Camera and Motion Capture by improving the binding library and the Crazyflie firmware. Advanced controllers like LQR and H infinity loop controllers can be implemented for Crazyflie. Direct interface with the Simulink must be made with the Crazyradio or the Crazyflie hardware using UDP.

REFERENCES

REFERENCES

[1] “*A successful French helicopter*,” Flight pp. 47, Jan 1924.

[2] Fightglobal. (1956). FLIGHTGLOBAL.com. Convertawings Inc. [Online].Available: URL: https://www.flightglobal.com

[3] L. Sabbapathy, "*Hummingbird's-eye view for the US military*," Leonardo Times, vol. 17, no.3, 2013.

[4] S. Bouabdallah, “*Design and control of quadrotors with application to autonomous flying*,” Ph.D. dissertation, École Polytechnique federale de Lausanne, 2007.

[5] B. Landry, “*Planning and control for quadrotor flight through cluttered environments*,” Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[6] Bitcraze. (2014). WIKI.BITCRAZE.io. Bitcraze. Oct 2013. [Online].Available: URL: http://wiki.bitcraze.se

[7] W. Hanna, “*Modelling and control of an unmanned aerial vehicle*,” B.S. Thesis, Charles Darwin University, 2014.

[8] Ublox. (2015). U-BLOX.com. U-blox. [Online].Available: URL: https://www.u-blox.com

[9] T. Jirinec, "*Stabilization and control of unmanned quadcopter,”* M.S. Thesis, Lulea University of Technology, 2011.

[10] W.C. Selby, “*Autonomous navigation and tracking of dynamic surface targets on-board a computationally impoverished aerial vehicle,”* Ph.D. dissertation, Massachusetts Institute of Technology, 2011.