

QUAD-ROTOR FLIGHT PATH ENERGY OPTIMIZATION

By Edward Kemper

A Thesis

Submitted in Partial Fulfillment

of the requirements for the degree of

Masters of Science

in Electrical Engineering

Northern Arizona University

May 2014

Approved:

Dr. Niranjan Venkatraman, Ph.D., Chair

Dr. Sheryl Howard, Ph.D.

Dr. Allison Kipple, Ph.D.

ABSTRACT

QUADROTOR FLIGHT PATH ENERGY OPTIMIZATION

by Edward KEMPER

Quad-Rotor unmanned areal vehicles (UAVs) have been a popular area of research and development in the last decade, especially with the advent of affordable microcontrollers like the MSP 430 and the Raspberry Pi. Path-Energy Optimization is an area that is well developed for linear systems. In this thesis, this idea of path-energy optimization is extended to the nonlinear model of the Quad-rotor UAV. The classical optimization technique is adapted to the nonlinear model that is derived for the problem at hand, coming up with a set of partial differential equations and boundary value conditions to solve these equations. Then, different techniques to implement energy optimization algorithms are tested using simulations in Python. First, a purely nonlinear approach is used. This method is shown to be computationally intensive, with no practical solution available in a reasonable amount of time. Second, heuristic techniques to minimize the energy of the flight path are tested, using Ziegler-Nichols' proportional integral derivative (PID) controller tuning technique. Finally, a brute force lookup table based PID controller is used. Simulation results of the heuristic method show that both reliable control of the system and path-energy optimization are achieved in a reasonable amount of time.

ACKNOWLEDGEMENTS

There are a great many people who need to be recognized for their contribution to the success of my academic endeavors. Thank you to my parents for their continued financial and moral support. Thank you to Terry Halm, Dr. Monty Mola, Dr. Wes Bliven, Dr. Robert Zoellner, and the rest of the Physics and Chemistry Faculty at Humboldt State for their work in mentoring and teaching. Thank you to Dr. Allison Kipple, Dr. Sheryl Howard, Dr. Paul Flikkema, Dr. Phillip Mlsna, and the rest of the Engineering Faculty at NAU. Thank you to Emily, my partner, for taking in stride the rich insanity of math and code that sometimes dominates my being. Finally thank you to Dr. Niranjan Venkatraman for his continuous, enthusiastic involvement in the development of this thesis.

Table of Contents

Contents

Abstract	ii
Acknowledgements	iii
Contents	iv
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	2
1.2 Prior Work	2
1.3 Organization of the Thesis	5
2 Problem Statement	6
3 Quad-Rotor Dynamic Model	8
3.1 Description of a Quad-Rotor	8
3.2 Coordinate System Definitions	10
3.3 Motor Speeds, Thrust and Torque	10
3.4 Euler-Lagrange Equations of Motion	11
3.5 A Complete Model	16
4 Classical Optimal Control Formulation	17
4.1 Derivation of the Objective Function	18
4.1.1 Lagrangian	19
4.1.2 Objective Function	19
4.2 Derivation of the Optimality Conditions	21
4.3 Solving the Boundary Value Problem	23

4.3.1	Shooting Method	24
4.3.2	Finite Difference Method	24
5	Quad-rotor Boundary Value Problem	26
5.1	Co-State Equations	27
5.2	Secondary Algebraic Co-State Equations	28
5.3	Stationarity Conditions	30
5.4	Discretization	31
5.5	A Finite Difference Solution to the Quad-Rotor Boundary Value Problem	33
6	PID/PD Control	36
6.1	Deriving the Control Expressions	36
6.2	Testing the Control Scheme	41
7	PID Gain Optimization	46
7.1	Initial Simulation Results	47
7.2	Brute Force Simulation Results	50
8	Summary and Future Work	56
8.1	Summary	56
8.2	Further Work	57
A	Dynamic system model and PID control - agentModule.py	60
B	Module for updating the PID set point - waypointNavigation.py	78
C	Functions used in the brute force method - bruteForceFunctions.py	82
D	Brute force implementation - runSimsBruteForce.py	88
E	Module to sort the results of the brute force method - parseResults.py	91
F	Finite difference method applied to the optimal control BVP - finiteDiffSolution.py	95
	Bibliography	115

List of Figures

3.1	Angular and linear coordinate systems in the quad-rotor dynamical model.	9
6.1	3D path of the quad-rotor for a flight simulation between the points (0,0,0) and (1,1,1). The PID control expressions were tuned experimentally.	42
6.2	Time domain graphs corresponding the 3D path. The linear and angular state variables are shown to stabilize at the desired positions.	42
6.3	3D path of the quad-rotor for an arbitrary path, tracing the edges of a cube.	44
6.4	Dynamics of the rotor during the arbitrary path. The angular positions are shown to change and effect the desired change in linear position	44
6.5	3D path of the quad-rotor for a large set point.	45
6.6	Dynamics of the quad-rotor for a large set point. The overshoot grows proportionally to the set point itself.	45
7.1	The relationship between K_u and the measured total thrust.	49
7.2	3D path simulation of the Quadcopter using the optimal run parameters specified.	54
7.3	Quadcopter dynamics using the optimal run parameters.	55

List of Tables

6.1	Simulation parameters of the quad-rotor model	43
7.1	Numerical descriptors of the optimal run.	52
7.2	Statistics of the brute force simulations.	53

Dedication

For my Parents Jack and Carol...

Chapter 1

Introduction

The technology surrounding Unmanned Aerial Vehicles (UAVs), and in particular quad-rotor devices, has seen tremendous development in recent years. Likewise, the creative application of this technology has expanded into many contexts. Like many new technologies, the early development of UAVs was mostly in a military context. This is not the case any more. The private sector has taken a huge interest in this technology. There is a wide range of companies contributing to the development of UAV technology from open-source projects like DIY Drones [1] to start-up firms backed by Google, such as Airware [2]. The Federal Aviation Administration in the USA has plans to produce concrete policy regarding the regulation of commercial applications of UAVs by 2015 [3]. This will sow the seeds for the rapid growth of a multi-billion dollar industry. There are many applications for this technology which have the potential to save lives and collect scientific data that could inform state and federal legislation. Certainly, the range of potential applications will be further diversified as the technology sees more development.

Unmanned Ariel Vehicles are also called by various other names: remotely piloted vehicles (RPVs), remote controlled drones, robot planes, and pilot-less aircraft. Such vehicles are defined as powered, aerial vehicles that do not carry a human operator and can use aerodynamics forces to provide vehicle lift. They can fly autonomously or be piloted remotely, can be expendable or recoverable, and can carry a lethal or nonlethal payload [4].

1.1 Motivation

With many private organizations making use of UAVs for a variety of applications, one pervasive engineering problem that still exists in general is that of managing the energy usage. Quad-rotors specifically are plagued by very high energy demand. There are two different kinds of UAVs: fixed wing, which have the ability to soar or glide, and multi-rotor systems, which are entirely thrust-driven. It is the natural instability of multi-rotor UAVs which make them extremely maneuverable, but this comes at the cost of high energy expenditure. This provides the motivation for this thesis - to develop an optimal control technique that optimizes the path-energy of a quad-copter UAV.

1.2 Prior Work

There have been work published on the energy optimization and trajectory planning of fixed wing UAVs. Given the ability to soar and utilize thermal gradients in the atmosphere, it is suggested that fixed wing UAVs have the potential to stay aloft almost permanently [5], [6], and [7]. This makes fixed wing UAVs ideal for applications like aerial surveys or surveillance missions. These aircrafts have

the major disadvantage that they are dependent upon some kind of launching mechanism, or a runway, for takeoff and landing.

In contrast, rotary wing UAVs have a higher degree of mechanical complexity. These UAVs can take off and land vertically and have the capacity to hover. This makes rotary wing UAVs, such as the quad-copter, more suitable for short range search and rescue missions, facility inspections, and single-target tracking. Since fixed wing and multi-rotor UAVs are fundamentally different in their physical operation, procedures for managing their respective energy usage are also necessarily unique.

In academic contexts, many advances in UAV and specifically quad-rotor research have provided the seeds for growth for this industry. The problem of basic stability and position control is solved in [8], [9], and [10].

The background material for understanding the dynamical model of the quad-rotor as given by the Euler-Lagrange formulation is explained in [11] and [12]. These references provide detailed derivations and discussions of the Euler-Lagrange equations of motion as well as related topics like Hamiltonian mechanics and the calculus of variations. Also, [12] provides an in-depth review of the historical context surrounding the development of classical mechanics. The derivation of the quad-rotor dynamical model as well as attitude and position control via PD or PID controllers is discussed in [8], [9], and [10]. These papers provide derivations of both the Euler-Lagrange and Newtonian formulations for the quad-rotor.

Optimal control was born in 1697, when Johann Bernoulli published his solution to the Brachystochrone problem in [13]. With the work of Bernoulli, Newton, Leibniz, l'Hopital, and Tschirnhaus, the field of optimal control was clearly defined. This was followed by the works of Euler, Lagrange, and Legendre which

led to the fundamental optimization equations, Euler’s equation [14], the Euler-Lagrange formulation, and Legendre’s necessary condition for a minimum. W. R. Hamilton then came up with an equivalent to the Euler-Lagrange equation that could be used in deriving control equations. This was known as the control Hamiltonian form of the Euler-Lagrange equations. The next development was from Weierstass, who came up with the fundamental path optimization problem in optimal control theory in the late 19th century. This was followed by the fundamental minimization principle by Pontryagin that allows for solving most optimization problems [15]. Several books on optimal control ([16] , [17], [18], [19]) were referenced for the derivations used in this thesis. In order to test the nonlinear optimization, numerical algorithms for the shooting method ([20], [21], [22]) and the finite difference method ([21], [22]) were used.

The Proportional-Integral-Derivative (PID) controller is a control loop feedback mechanism widely used to drive a system to a desired set point. The mechanism uses an error value as the input to the controller. PID controllers are common in industrial applications [23]. In the absence of the knowledge of an underlying process, the PID is considered the best method of control. It must be noted that PID controllers do not necessarily result in optimal control of the system. However, it is possible to achieve a desired system response by adjusting the mathematical parameters of the control expressions. This process is called “tuning”. The tuning must satisfy many criteria within the limitations of PID control and the system itself. There are various tuning techniques [24]. For instance, there are the Ziegler-Nichols, manual tuning, and software tuning methods which can be applied to other control problems [25], [26].

1.3 Organization of the Thesis

The objective of this thesis is to develop a path-energy optimization technique that can operate on a near real-time schedule. Two different methods are discussed. We compare a classical optimal control technique with a simpler heuristic approach involving PID controller tuning. The organization of the chapters is as follows.

In Chapter 2, we present a detailed problem statement where the goal of the research project is defined. Chapter 3 uses the Euler-Lagrange equations of motion to derive a nonlinear dynamical model for the quad-rotor UAV. This mathematical model is the basis of the development of the control and energy optimization algorithms. In chapter 4, we define the various optimality conditions. Then we and formulate a generalized, classical optimal control scheme. Then we solve the boundary value problem generated by two methods and discuss their pros and cons. In Chapter 5, the classical optimal control scheme developed in the previous chapter is applied to the quad-rotor UAV. The resulting boundary value problem and its solution method is discussed. Chapter 6 deals with the PID/PD control technique. The control expressions are derived, and the method is tested. Results from these tests are discussed. Chapter 7 outlines a heuristic approach to the path-energy optimization problem and presents the simulation results of the control algorithm developed. Chapter 8 summarizes the results and proposes avenues for continued research.

Chapter 2

Problem Statement

We wish to find a set of control expressions for a quad-rotor UAV which minimizes the energy expended in flying between two known points in three-dimensional space. In order to maintain focus on a tractable problem, some mathematical assumptions are made about the scenario. First, we assume that the flight path that will be optimized is free of obstacles. Second, we assume only modeled environmental variables. We use a mathematical model of the system derived from a Euler-Lagrange formulation as in [9] and [10].

In the classical optimal control approach, as in [16] and [17], the control of the system and the optimization are represented in a single mathematical formulation. Solving the optimal control problem is achieved by solving a boundary value problem. For a highly nonlinear system such as a quad-rotor, this becomes extremely involved. The classical optimal control approach is shown to be too computationally intensive for a real-time implementation because the result is a substantial two point boundary value problem. Solving the theoretical optimal control problem would likely produce accurate results, but the solution could potentially take

weeks of computation to attain. Also, the convergence of the solution is shown to be intermittent.

For the heuristic approach, full control of the UAV is attained by using PD attitude controllers in conjunction with PID controllers for position. This provides a platform for simulating the UAV as it flies from a known initial position to a desired set point location. The optimization procedure evaluates the results of these simulations for optimality as a function of the PID gains used in the position control expressions.

It is pertinent to define what is meant by near real-time in our somewhat sterile mathematical context. We assume that the set of initial and final locations of the quad-rotor are defined by a user on a human time scale. Imagine a graphical user interface in which the desired location of the UAV is programmed. The quad-rotor then physically traverses the optimal path without more than a second or two of computation before the flight begins. For an autonomous UAV, the on-board computational resources define an upper limit to the computational complexity of the control algorithm. Our aim is to design an energy optimized control scheme which meets these constraints.

Chapter 3

Quad-Rotor Dynamic Model

In this chapter, a mathematical model of the quad-rotor is derived, and the assumptions that go into this derivation are explained in detail. This model will be used as the basis for the optimization techniques outlined in subsequent chapters.

3.1 Description of a Quad-Rotor

A generic model of a quad-rotor is physically composed of a simple frame supporting four brushless motors. Thrust is provided by propellers attached to these motors. The speeds of the rotors are governed by a control algorithm which is implemented on some form of on-board processor.

The stabilization and control of a quad-rotor is accomplished by varying the speeds of the motors. The thrust in the vertical direction is controlled by varying all four motor speeds uniformly. In the quad-rotor frame of reference, the direction of the thrusts from the motors is fixed. This means that in order to produce lateral

motion, the UAV must tilt such that a component of the total thrust vector points in the desired direction of motion.

In Figure 3.1 the basic mechanical structure and the relationships between the spatial coordinates are shown. The linear and angular coordinates are defined as follows. ψ is the yaw angle around the z-axis, θ is the pitch angle around the y-axis and ϕ is the roll angle around the x-axis.

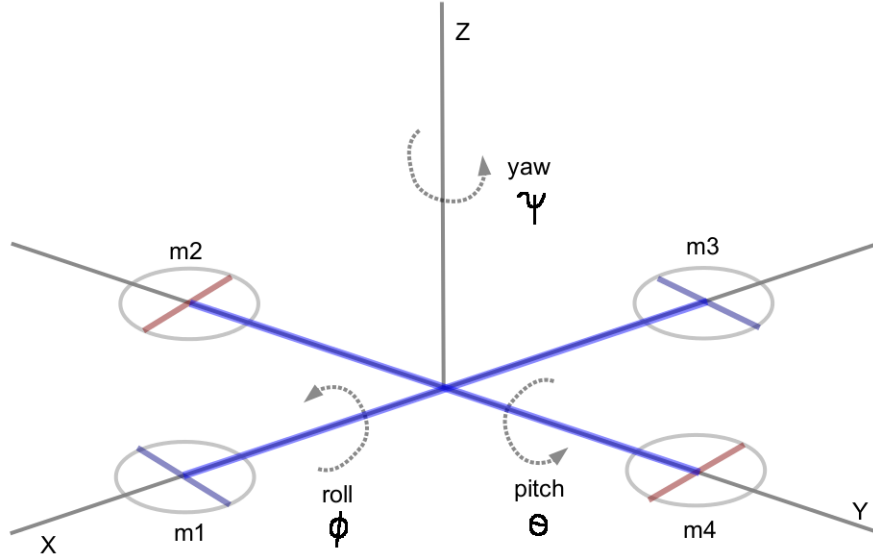


FIGURE 3.1: Angular and linear coordinate systems in the quad-rotor dynamical model.

The pitch and roll angular positions are controlled by driving motors on opposite sides of the frame at different speeds. This produces torque about the center of mass of the quad-rotor. Given non-zero pitch, roll, and total thrust, the UAV

experiences horizontal linear acceleration. The yaw angular position is controlled by driving pairs of opposite motors at the different speeds. This produces a torque about the yaw axis but not about the pitch or roll axes. Also, the two opposite pairs of motors must spin in opposite directions so that when hovering, the net torque about the yaw axis is zero. The details of this description are represented mathematically in the next section.

3.2 Coordinate System Definitions

In order to implement a control algorithm, we must understand the mathematical relationships between the control input and the resulting dynamics of the system. Using the Euler-Lagrange formulation from classical mechanics, we can obtain a nonlinear, deterministic dynamical model. General derivation of the Euler Lagrange differential equations of motion can be found in [11] and [12].

The spatial variables can be grouped into linear and angular components, with ξ representing the linear components and η representing the angular components:

$$\boldsymbol{\xi} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \boldsymbol{\eta} = \begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix}, \boldsymbol{q} = \begin{bmatrix} \xi \\ \eta \end{bmatrix} \quad (3.1)$$

3.3 Motor Speeds, Thrust and Torque

The rotor angular velocities are related to the forces they produce by $f_i = k\omega_i^2$ where k is the constant of proportionality and i is the motor index. The torques due to the rotation of the rotors about their respective axes of rotation are given by

$\tau_i = b\omega_i^2 + I_M\dot{\omega}_i$. The variable τ_i is the torque from the i th motor and the parameter b is a drag coefficient. Note the effect of $\dot{\omega}_i$ is considered to be negligible because the rotational inertia of the rotor itself is negligible.

In the quad-rotor frame of reference, the motors produce the following torques on the system:

$$\boldsymbol{\tau}_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} lk(-\omega_2^2 + \omega_4^2) \\ lk(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^4 b\omega_i^2 \end{bmatrix} \quad (3.2)$$

In the above expression the parameters l and k refer to the length of the quad-rotor arm and an aerodynamic thrust coefficient respectively. The combined thrust of the rotors in the direction of the quad-rotor frame z axis is

$$\mathcal{T}_B = [0, 0, \mathcal{T}]^T \quad (3.3)$$

where,

$$\mathcal{T} = \sum_{i=1}^4 f_i \quad (3.4)$$

3.4 Euler-Lagrange Equations of Motion

The mass of the quad-rotor is m . Each of the moments of inertia i_{xx} and i_{yy} are assumed to be composed of a rod of length l which accounts for half of the mass

of the quad-rotor. Assume the mass is evenly distributed along the two perpendicular rods. Let $\beta = \frac{1}{12}ml^2$. The inertia matrix for the quad-rotor is then:

$$I = \begin{pmatrix} \frac{1}{12} \left(\frac{m}{2}\right) l^2 & 0 & 0 \\ 0 & \frac{1}{12} \left(\frac{m}{2}\right) l^2 & 0 \\ 0 & 0 & \frac{1}{12}ml^2 \end{pmatrix} = \begin{pmatrix} \frac{\beta}{2} & 0 & 0 \\ 0 & \frac{\beta}{2} & 0 \\ 0 & 0 & \beta \end{pmatrix} \quad (3.5)$$

In this section, we will use Newton's notation for time derivatives. For instance, $\dot{\eta} = \frac{\partial \eta}{\partial t}$. In the inertial frame, the kinetic ((translational and rotational) and potential energy of the system are given by

$$KE_{\text{trans}} = \frac{1}{2}m\dot{\xi}^T\dot{\xi} \quad (3.6)$$

$$KE_{\text{rot}} = \frac{1}{2}\dot{\eta}^T J \dot{\eta} \quad (3.7)$$

$$U = mgz. \quad (3.8)$$

The Lagrangian is formed as the difference between kinetic and potential energy:

$$L = \frac{1}{2}m\dot{\xi}^T\dot{\xi} + \frac{1}{2}\dot{\eta}^T J \dot{\eta} - mgz. \quad (3.9)$$

The Jacobian J is given by

$$J = W_{\eta}^T I W_{\eta}, \quad (3.10)$$

where

$$W_{\eta} = \begin{bmatrix} 1 & 0 & -\sin(\theta) \\ 0 & \cos(\phi) & \cos(\theta)\sin(\phi) \\ 0 & -\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \quad (3.11)$$

The matrix W_η is the matrix transformation which relates the angular velocities from the quad-rotor frame of reference to the inertial frame. Substituting equation 3.11 into equation 3.10 then provides:

$$J = \begin{pmatrix} \frac{\beta}{2} & 0 & -\frac{\beta}{2}s(\theta) \\ 0 & \frac{\beta}{2}c(\phi)^2 + \beta s(\phi)^2 & \frac{-\beta}{2}c(\phi) s(\phi) c(\theta) \\ -\beta s(\theta) & \frac{-\beta}{2}c(\phi) s(\phi) c(\theta) & \frac{\beta}{2}s(\theta)^2 + \frac{\beta}{2}s(\phi)^2c(\theta)^2 + \beta c(\phi)^2c(\theta)^2 \end{pmatrix} \quad (3.12)$$

The dynamics of the system are represented by the Euler - Lagrange differential equations of motion, as follows:

$$\frac{d}{dt} \left(\frac{\delta L}{\delta \dot{q}} \right) - \frac{\delta L}{\delta q} = F \quad (3.13)$$

where $q = \{x, y, z, \psi, \theta, \phi\} = \{\xi, \eta\}$.

f is the generalized force vector representing the linear external force acting on the system. τ is the vector of torques acting on the system due to the rotors.

$$\begin{pmatrix} f \\ \tau \end{pmatrix} = \frac{d}{dt} \left(\frac{\delta L}{\delta \dot{q}} \right) - \frac{\delta L}{\delta q} \quad (3.14)$$

General derivations of the Euler-Lagrange equations of motion can be found in [11] and [12].

The coordinates $q_i = \{x, y, z, \psi, \theta, \phi\}$ are in reference to a ground-based inertial coordinate system. The system states and control inputs must be mapped from the quad-rotor frame of reference to the inertial frame in order to express the dynamics of the system. The matrix R below represents an arbitrary rotation

transformation from the body frame to the inertial frame:

$$\mathbf{R} = \begin{bmatrix} c(\psi)c(\theta) & c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi) & c(\psi)s(\theta)c(\phi) + s(\psi)s(\phi) \\ s(\psi)c(\theta) & s(\psi)s(\theta)s(\phi) + c(\psi)c(\phi) & s(\psi)s(\theta)c(\phi) - c(\psi)s(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\theta)c(\phi) \end{bmatrix} \quad (3.15)$$

For simplicity, \cos is denoted by 'c' and \sin is denoted by 's' in the above expression.

The linear components of the generalized forces produce the following equations.

$$\mathbf{f} = \mathbf{R}\mathbf{T}_B = m\ddot{\boldsymbol{\xi}} - \mathbf{G} \quad (3.16)$$

$$m \begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix} = u \begin{pmatrix} \cos \psi \sin \theta \cos \phi + \sin \psi \sin \phi \\ \sin \psi \sin \theta \cos \phi - \cos \psi \sin \phi \\ \cos \theta \cos \phi \end{pmatrix} \quad (3.17)$$

The angular components are expressed as

$$\tau = \tau_b = \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\eta}} \right) - \frac{\partial L}{\partial \eta} \quad (3.18)$$

$$\tau_b = \frac{d}{dt}(J\dot{\eta}) - \frac{1}{2} \frac{\partial}{\partial \eta} (\dot{\eta}^T J \dot{\eta}) \quad (3.19)$$

$$\tau_b = J\ddot{\eta} + \frac{d}{dt}(J)\dot{\eta} - \frac{1}{2} \frac{\partial}{\partial \eta} (\dot{\eta}^T J \dot{\eta}) \quad (3.20)$$

$$\tau_b = J\ddot{\eta} + \mathfrak{C}(\eta, \dot{\eta})\dot{\eta} \quad (3.21)$$

$$\ddot{\eta} = J^{-1}(\tau_b - \mathfrak{C}(\eta, \dot{\eta})\dot{\eta}) \quad (3.22)$$

In the above equations, $\mathfrak{C}(\eta, \dot{\eta})$ is called the Coriolis Matrix. The Coriolis term is a mathematical result of the rotational motion of one coordinate system with respect to another. Since we are considering arbitrary three dimensional motion, there are three orthogonal axes of rotation and the resulting matrix is quite complicated. According to the expression for the angular acceleration (Equation (3.22)), the physical units of the Coriolis term must be torque to maintain algebraic continuity. The Coriolis matrix provides a method to relate the rotational coordinates to the translational coordinates [27], [28]. For our problem, it is defined as follows.

$$\mathfrak{C}(\eta, \dot{\eta}) = \begin{pmatrix} \mathfrak{C}_{(11)} & \mathfrak{C}_{(12)} & \mathfrak{C}_{(13)} \\ \mathfrak{C}_{(21)} & \mathfrak{C}_{(22)} & \mathfrak{C}_{(23)} \\ \mathfrak{C}_{(31)} & \mathfrak{C}_{(32)} & \mathfrak{C}_{(33)} \end{pmatrix} \quad (3.23)$$

$$\mathfrak{C}_{(11)} = 0$$

$$\mathfrak{C}_{(12)} = (I_{yy} - I_{zz})(\dot{\theta}C_{\phi}S_{\phi} + \dot{\psi}C_{\theta}S_{\phi}^2) + (I_{zz} - I_{yy})\dot{\psi}C_{\phi}^2C_{\theta} - I_{xx}\dot{\psi}C_{\theta}$$

$$\mathfrak{C}_{(13)} = (I_{zz} - I_{yy})\dot{\psi}C_{\phi}S_{\phi}C_{\theta}^2$$

$$\mathfrak{C}_{(21)} = (I_{zz} - I_{yy})(\dot{\theta}C_{\phi}S_{\phi} + \dot{\psi}S_{\phi}C_{\theta}) + (I_{yy} - I_{zz})\dot{\psi}C_{\phi}^2C_{\theta} + I_{xx}\dot{\psi}C_{\theta}$$

$$\mathfrak{C}_{(22)} = (I_{zz} - I_{yy})\dot{\phi}C_{\phi}S_{\phi}$$

$$\mathfrak{C}_{(23)} = -I_{xx}\dot{\psi}S_{\theta}C_{\theta} + I_{yy}\dot{\psi}S_{\phi}^2S_{\theta}C_{\theta} + I_{zz}\dot{\psi}C_{\phi}^2S_{\theta}C_{\theta}$$

$$\mathfrak{C}_{(31)} = (I_{yy} - I_{zz})\dot{\psi}C_{\theta}^2S_{\phi}C_{\phi} - I_{xx}\dot{\theta}C_{\theta}$$

$$\mathfrak{C}_{(32)} = (I_{zz} - I_{yy})(\dot{\theta}C_{\phi}S_{\phi}S_{\theta} + \dot{\phi}S_{\phi}^2C_{\theta}) + (I_{yy} - I_{zz})\dot{\phi}C_{\phi}^2C_{\theta} + I_{xx}\dot{\psi}S_{\theta}C_{\theta} - I_{yy}\dot{\psi}S_{\phi}^2S_{\theta}C_{\theta} - I_{zz}\dot{\psi}C_{\phi}^2S_{\theta}C_{\theta}$$

$$\mathfrak{C}_{(33)} = (I_{yy} - I_{zz})\dot{\phi}C_{\phi}S_{\phi}C_{\theta}^2 - I_{yy}\dot{\theta}S_{\phi}^2C_{\theta}S_{\theta} - I_{zz}\dot{\theta}C_{\phi}^2C_{\theta}S_{\theta} + I_{xx}\dot{\theta}C_{\theta}S_{\theta}$$

It is important to keep in mind that the Coriolis term does not represent a real force or torque acting on the system. It is only an artifact which is needed to account for the relative rotation of one coordinate frame with respect to another.

3.5 A Complete Model

A complete mathematical representation of the quad-rotor is as follows:

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} + \frac{\mathcal{T}}{m} \begin{pmatrix} C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi S_\theta C_\phi - C_\psi S_\phi \\ C_\theta C_\phi \end{pmatrix} \quad (3.24)$$

$$\begin{pmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{pmatrix} = J^{-1} \left[\begin{pmatrix} ls(-\omega_2^2 + \omega_4^2) \\ ls(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^4 b\omega_i^2 \end{pmatrix} - \mathfrak{C} \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} \right] \quad (3.25)$$

Even in this form there are many important aspects of quad-rotor flight dynamics and environmental variables which are omitted. The general problem of designing a system that is able to understand and adapt to varying goals and circumstances is a large one.

Despite the apparent shortcomings of this model, it is very useful as a core component to this thesis. Appendix A shows the python implementation of the system model. Although the assumed mathematical environment is somewhat of a departure from reality, it allows for a firm theoretical basis which validates the development of the control system and optimization scheme.

Chapter 4

Classical Optimal Control Formulation

In this chapter, we will define a two point boundary value problem and explore the classical optimal control formulation. The solution to this boundary value problem gives the control inputs to the system which produce optimal behavior. The set of mathematical conditions which define the boundary value problem are termed 'optimality conditions'. The content of this chapter is left generalized. It can be applied to any second order dynamic system. In chapter 5, the optimality conditions are applied to the dynamical model of the quad-rotor which was derived in Chapter 3.

Note the names 'Lagrangian' and 'Hamiltonian' are used here in an optimal control context. The multiple use of these names in reference to specific types of expressions is an artifact of the pervasive work of Lagrange and Hamilton. Both optimal control theory and Hamiltonian / Lagrangian mechanics are rooted in the calculus of variations. The dynamic model of the quad-rotor and the optimal

control formulation described below are both results from a form of functional optimization. We rely on a contextual and conceptual separation in our understanding. Specifically, the 'Lagrangian', which is formed as the difference of the expressions for kinetic and potential energies, is unique to the context of classical mechanics. Likewise, the 'Lagrangian' in the optimal control context is a term in the integrand of our objective function which represents a vector of performance metrics. The 'Hamiltonian' from classical mechanics is formed as the sum of kinetic and potential energies. This is different than the Hamiltonian used here in the optimal control context.

4.1 Derivation of the Objective Function

In section 2.3 of Bryson and Ho [17], the conditions for the optimal control of a continuous time system are derived. There, it is presumed that the system is presented as a set of first order differential equations. For a quad-rotor, it is more convenient to leave the system equations as a set of second order differential equations. The motivation for this is as follows. With the finite difference method for solving two point boundary value problems, there are a set of simultaneous algebraic equations that are defined for each point in time for which a solution is desired. If we were to express the system of differential equations that govern the dynamics of a quad-rotor in a first order form, the number of algebraic equations defined by the finite difference method would be effectively doubled. Here, we derive the analogous conditions for optimality for a second order system.

4.1.1 Lagrangian

The real power of the optimal control formulation is in the use of the Lagrangian function (also called the performance index) $L(q(t), u(t), t)$ and the co-state $\lambda(t)$. The objective function in our context is an extension of classical constrained optimization to systems which evolve in time. In our case, the Lagrangian is the function which we wish to minimize, the system model (as derived in chapter 3) F plays the role of the constraint relationship, and the function $\lambda(t)$ plays the same role here as a lagrange muliplier in the context of static optimization. Since our optimization procedure accounts for the time variation of the performance index and the system equations, $\lambda(t)$ is likewise a function of time. The variable $\lambda(t)$ is allowed to vary in time along with the state of the system and is therefore given the name "co-state". In other words it allows for an expression of the criteria for optimallity at every instance in time.

The Lagrangian for our problem is defined as

$$L[q(t), u(t), t] = u^T I u, \quad (4.1)$$

where u is the control input vector and I is the 4×4 identity.

4.1.2 Objective Function

The dynamic equations of motion are appended to the performance index as follows. By definition:

$$F = \ddot{q} \quad (4.2)$$

$$0 = F - \ddot{q} \quad (4.3)$$

Note that F is the vector function representation of the quad-rotor system equations. The components' physical units are linear and angular acceleration, not force. The variable q is a vector of generalized spatial coordinates. F is generally a function of the generalized coordinates, the input to the system $u(t)$, and time. The full objective function can be written as

$$\mathcal{J} = \nu \Psi(q(t_f), t_f) + \int_{t_0}^{t_f} [L(q(t), u(t), t) + \lambda^T (F(q(t), u(t), t) - \ddot{q})] dt. \quad (4.4)$$

The function $\Psi(q(t_f), t_f)$ represents the effect that the final state has on the objective function. In general, $\Psi(q(t_f), t_f)$ is a vector quantity and is scaled by the vector ν .

For the optimal control formulation, the Hamiltonian is defined as a measure of optimality as a function of time. In the next section we'll see that the full objective function involves a time-integration of the Hamiltonian. It is written as:

$$H = L(q(t), u(t), t) + \lambda^T (F(q(t), u(t), t) - \ddot{q}). \quad (4.5)$$

The Hamiltonian allows for a concise expression of the Lagrangian, the co-state function λ and the constraint equations.

The objective function is simplified as:

$$\mathcal{J} = \nu \Psi(q(t_f), t_f) + \int_{t_0}^{t_f} H(q(t), u(t), t) - \lambda^T \ddot{q} dt \quad (4.6)$$

The second term in the integrand is integrated by parts.

$$\mathcal{J} = \nu \Psi(q(t_f), t_f) + \int_{t_0}^{t_f} H(q(t), u(t), t) dt - \int_{t_0}^{t_f} \lambda^T \ddot{q} dt \quad (4.7)$$

In general: $\int u \, dv = (uv)|_{t_0}^{t_f} - \int v \, du$. Using this, the second term is expanded.

$$\int_{t_0}^{t_f} \lambda^T \ddot{q} \, dt = (\lambda^T \dot{q})|_{t_0}^{t_f} - \int_{t_0}^{t_f} \dot{\lambda}^T \dot{q} \, dt \quad (4.8)$$

The result is:

$$\mathcal{J} = \nu \Psi(q(t_f), t_f) + \int_{t_0}^{t_f} H(q(t), u(t), t) \, dt - (\lambda^T \dot{q})|_{t_0}^{t_f} + \int_{t_0}^{t_f} \dot{\lambda}^T \dot{q} \, dt \quad (4.9)$$

The last term is integrated by parts again.

$$\mathcal{J} = \nu \Psi(q(t_f), t_f) - (\lambda^T \dot{q})|_{t_0}^{t_f} + (\dot{\lambda}^T q)|_{t_0}^{t_f} + \int_{t_0}^{t_f} H(q(t), u(t), t) - \ddot{\lambda}^T q \, dt \quad (4.10)$$

$$\mathcal{J} = \nu \Psi(q(t_f), t_f) + [\dot{\lambda}^T q - \lambda^T \dot{q}]|_{t_0}^{t_f} + \int_{t_0}^{t_f} (H(q(t), u(t), t) - \ddot{\lambda}^T q) \, dt \quad (4.11)$$

This result is the objective function which we wish to minimize.

4.2 Derivation of the Optimality Conditions

To find the mathematical conditions necessary for a minimum in \mathcal{J} , the first variation is computed and set equal to 0. In this context, the variation of a function is essentially the same as the total derivative. Further reading on this is found in [11] and [12].

The first variation in \mathcal{J} is given by

$$\delta \mathcal{J} = \frac{\partial \mathcal{J}}{\partial q} \delta q + \frac{\partial \mathcal{J}}{\partial \dot{q}} \delta \dot{q} + \frac{\partial \mathcal{J}}{\partial u} \delta u. \quad (4.12)$$

$$\begin{aligned}\delta\mathcal{J} = & \nu^T \frac{\partial\Psi}{\partial q} \delta q|_{t_f} + \nu^T \frac{\partial\Psi}{\partial \dot{q}} \delta \dot{q}|_{t_f} + [\dot{\lambda}^T \delta q - \lambda^T \delta \dot{q}]_{t_0}^{t_f} \\ & + \int_{t_0}^{t_f} \left[\frac{\partial H}{\partial q} \delta q + \frac{\partial H}{\partial \dot{q}} \delta \dot{q} + \frac{\partial H}{\partial u} \delta u - \ddot{\lambda}^T \delta q \right] dt\end{aligned}\quad (4.13)$$

$$\begin{aligned}\delta\mathcal{J} = & (\nu^T \frac{\partial\Psi}{\partial q} + \dot{\lambda}^T) \delta q|_{t_f} + (\nu^T \frac{\partial\Psi}{\partial \dot{q}} - \lambda^T) \delta \dot{q}|_{t_f} + [\lambda^T \delta \dot{q} - \dot{\lambda}^T \delta q]_{t_0} \\ & + \int_{t_0}^{t_f} \left[\left(\frac{\partial H}{\partial q} - \ddot{\lambda}^T \right) \delta q + \frac{\partial H}{\partial \dot{q}} \delta \dot{q} + \frac{\partial h}{\partial u} \delta u \right] dt.\end{aligned}\quad (4.14)$$

The optimality conditions are found by setting $\delta\mathcal{J} = 0$ and asserting that each of the added terms must therefore go to 0. The results are summarized as follows.

The Co-State equations are

$$\frac{\partial H}{\partial q} = \ddot{\lambda} \quad (4.15)$$

$$\ddot{\lambda} = \left(\frac{\partial L}{\partial q} \right)^T + \left(\frac{\partial F}{\partial q} \right)^T \lambda. \quad (4.16)$$

The Stationarity Conditions are

$$\frac{\partial H}{\partial u} = 0 \quad (4.17)$$

$$\frac{\partial L}{\partial u} + \left(\frac{\partial F}{\partial u} \right)^T \lambda = 0 \quad (4.18)$$

Secondary algebraic Co state condition

$$\frac{\partial H}{\partial \dot{q}} = 0 \quad (4.19)$$

$$\left(\frac{\partial F}{\partial \dot{q}} \right)^T \lambda = 0 \quad (4.20)$$

Terminal Boundary conditions:

$$\nu^T \frac{\partial \Psi}{\partial q} \big|_{t_f} + \dot{\lambda}(t_f)^T = 0 \quad (4.21)$$

$$\nu^T \frac{\partial \Psi}{\partial \dot{q}} \big|_{t_f} - \lambda(t_f)^T = 0 \quad (4.22)$$

Initial Co state conditions

$$(\lambda^T \delta \dot{q} - \dot{\lambda}^T \delta q) \big|_{t_0} = 0 \quad (4.23)$$

$$\lambda(t_0) = 0 \quad (4.24)$$

$$\dot{\lambda}(t_0) = 0 \quad (4.25)$$

Together, the state equations, co-state equations, stationarity equations, secondary algebraic constraints, and boundary conditions form a complete two-point boundary value problem.

4.3 Solving the Boundary Value Problem

Boundary value problems are very common in many science and engineering fields. They can become quite complicated and require significant computation to reach a solution. Two general ways to solve two-point boundary value problems are described next. These are the shooting method and the finite difference method [22],[29]. Both have limitations.

4.3.1 Shooting Method

The shooting method is a relatively straightforward combination of a time marching quadrature method (Runga-Kutta or the like) to solve a set of differential equations and an error minimization technique. The shooting method works by iteratively solving the set of differential equations as an initial value problem and then measuring the error in the final state of the system compared to the desired final state. The shooting method is subject to the stability of the differential equations in question. If the time marching algorithm does not converge, the method will not work. Unfortunately, the boundary value problem for the quad-rotor that is formulated in the next chapter falls into this category. The quad-rotor system model and the coupled optimality conditions are simply too unstable to be solved with the shooting method.

- Advantages : straightforward iterative quadrature method and error minimization,
- Disadvantages : does not always converge

4.3.2 Finite Difference Method

The finite difference method poses another possibility [\[29\]](#). It involves creating a system of algebraic equations to be solved at each instance in time where the solution is desired. For a simulation like ours, this means at least hundreds if not thousands of time steps. The derivatives in the differential equations are expressed as finite differences involving variables at adjacent time steps. The values of each state and co-state variable are defined as unknowns at each time step. This creates a system of equations involving several thousand unknowns that need to be solved

for. For a linear system this is not so bad because the problem is reduced to the inversion of a sparse matrix. For this, there are efficient numerical algorithms that can be used. Since the quad-rotor boundary value problem is nonlinear, it must be solved with a gradient descent technique or something similar.

- Advantages : turns the BVP into a system of algebraic equations, easy to solve for linear systems,
- Disadvantages : hard to solve for nonlinear systems, does not always converge

In the next chapter we derive the set of differential equations which form the boundary value problem defined by our goal of optimizing the energy usage of a quad-rotor.

Chapter 5

Quad-rotor Boundary Value Problem

In this chapter we use the expressions for the dynamic model of the quad-rotor (equations 3.24 and 3.25) and the optimal control formulation (equations 4.15 through 4.25) to derive the optimality conditions for our specific problem.

Recall from chapter 4 that each of the optimality conditions is a mathematical result of setting the first variation of the objective function equal to zero. To maintain algebraic continuity, each additive term must then be zero. Using this logic, each of the optimality conditions is obtained. Given the general form of the optimality conditions, we can introduce the quad-rotor dynamic model. The resulting expressions can be simplified to arrive at specific equations which form our quad-rotor boundary value problem. The optimal flight path and the optimal control input as a function of time form the solution to this boundary value problem.

5.1 Co-State Equations

The co-state equations are expressed as follows where F is our set of system equations and L is the Lagrangian defined in our performance index. Since the Lagrangian does not depend on the state, the co-state differential equation simplifies.

Recall that the Lagrangian for our optimization is $L[q(t), u(t), t] = u^T I u$.

$$\ddot{\lambda} = - \left(\frac{\partial F}{\partial q} \right)^T \lambda - \left(\frac{\partial L}{\partial q} \right)^T \quad (5.1)$$

$$\ddot{\lambda} = - \left(\frac{\partial F}{\partial q} \right)^T \lambda \quad (5.2)$$

The state transition matrix, $\left(\frac{\partial F}{\partial q} \right)$ is tremendous, but there are some simplifications to be made as some of the partial derivatives are zero.

$$\frac{\partial F}{\partial q} = \begin{pmatrix} \frac{\partial F_{(1)}}{\partial x} & \frac{\partial F_{(1)}}{\partial y} & \frac{\partial F_{(1)}}{\partial z} & \frac{\partial F_{(1)}}{\partial \phi} & \frac{\partial F_{(1)}}{\partial \theta} & \frac{\partial F_{(1)}}{\partial \psi} \\ \frac{\partial F_{(2)}}{\partial x} & \frac{\partial F_{(2)}}{\partial y} & \frac{\partial F_{(2)}}{\partial z} & \frac{\partial F_{(2)}}{\partial \phi} & \frac{\partial F_{(2)}}{\partial \theta} & \frac{\partial F_{(2)}}{\partial \psi} \\ \cdot & & & & & \\ \cdot & & & & & \\ \cdot & & & & & \\ \frac{\partial F_{(6)}}{\partial x} & \frac{\partial F_{(6)}}{\partial y} & \frac{\partial F_{(6)}}{\partial z} & \frac{\partial F_{(6)}}{\partial \phi} & \frac{\partial F_{(6)}}{\partial \theta} & \frac{\partial F_{(6)}}{\partial \psi} \end{pmatrix} \quad (5.3)$$

$$\frac{\partial F}{\partial q} = \begin{pmatrix} 0 & 0 & 0 & \frac{\partial F_{(1)}}{\partial \phi} & \frac{\partial F_{(1)}}{\partial \theta} & \frac{\partial F_{(1)}}{\partial \psi} \\ 0 & 0 & 0 & \frac{\partial F_{(2)}}{\partial \phi} & \frac{\partial F_{(2)}}{\partial \theta} & \frac{\partial F_{(2)}}{\partial \psi} \\ 0 & 0 & 0 & \frac{\partial F_{(3)}}{\partial \phi} & \frac{\partial F_{(3)}}{\partial \theta} & 0 \\ 0 & 0 & 0 & \frac{\partial F_{(4)}}{\partial \phi} & \frac{\partial F_{(4)}}{\partial \theta} & \frac{\partial F_{(4)}}{\partial \psi} \\ 0 & 0 & 0 & \frac{\partial F_{(5)}}{\partial \phi} & \frac{\partial F_{(5)}}{\partial \theta} & \frac{\partial F_{(5)}}{\partial \psi} \\ 0 & 0 & 0 & \frac{\partial F_{(6)}}{\partial \phi} & \frac{\partial F_{(6)}}{\partial \theta} & \frac{\partial F_{(6)}}{\partial \psi} \end{pmatrix} \quad (5.4)$$

Each of the elements must be computed numerically. An analytical representation of all the partial derivatives in $\left(\frac{\partial F}{\partial q}\right)$ is possible but the task of computing them all would be too time consuming. For our simulations, a simple backward finite difference is much more appropriate:

$$\frac{\partial F_i}{\partial q_j} \approx \frac{F_i(q_j + \alpha) - F_i(q_j)}{\alpha} \quad (5.5)$$

In the above equation, i the index for the state equations, j is the index for the state variables and α is the discrete step size. The simplified result is:

$$\ddot{\lambda} = \begin{pmatrix} \frac{\partial F_{(1)}}{\partial \phi} & \frac{\partial F_{(1)}}{\partial \theta} & \frac{\partial F_{(1)}}{\partial \psi} \\ \frac{\partial F_{(2)}}{\partial \phi} & \frac{\partial F_{(2)}}{\partial \theta} & \frac{\partial F_{(2)}}{\partial \psi} \\ \frac{\partial F_{(3)}}{\partial \phi} & \frac{\partial F_{(3)}}{\partial \theta} & 0 \\ \frac{\partial F_{(4)}}{\partial \phi} & \frac{\partial F_{(4)}}{\partial \theta} & \frac{\partial F_{(4)}}{\partial \psi} \\ \frac{\partial F_{(5)}}{\partial \phi} & \frac{\partial F_{(5)}}{\partial \theta} & \frac{\partial F_{(5)}}{\partial \psi} \\ \frac{\partial F_{(6)}}{\partial \phi} & \frac{\partial F_{(6)}}{\partial \theta} & \frac{\partial F_{(6)}}{\partial \psi} \end{pmatrix} \begin{pmatrix} \lambda_4 \\ \lambda_5 \\ \lambda_6 \end{pmatrix} \quad (5.6)$$

5.2 Secondary Algebraic Co-State Equations

The secondary algebraic co-state equations are a result of setting the variation of the objective function \mathcal{J} equal to zero. They are unique to the derivation in Chapter 4, which involves a second-order rather than first-order representation of the system equations. Recall that H is the Hamiltonian, q is the state vector, λ is the co-state vector, L is the performance index (also called the Lagrangian),

and F is the dynamic model derived in Chapter 3.

$$\frac{\partial H}{\partial \dot{q}} = 0 \quad (5.7)$$

$$0 = \left(\frac{\partial F}{\partial \dot{q}} \right)^T \lambda + \left(\frac{\partial L}{\partial \dot{q}} \right)^T \quad (5.8)$$

$$0 = \left(\frac{\partial F}{\partial \dot{q}} \right)^T \lambda \quad (5.9)$$

$$0 = \begin{pmatrix} \frac{\partial F_{(1)}}{\partial \dot{x}} & \frac{\partial F_{(1)}}{\partial \dot{y}} & \frac{\partial F_{(1)}}{\partial \dot{z}} & \frac{\partial F_{(1)}}{\partial \dot{\phi}} & \frac{\partial F_{(1)}}{\partial \dot{\theta}} & \frac{\partial F_{(1)}}{\partial \dot{\psi}} \\ \frac{\partial F_{(2)}}{\partial \dot{x}} & \frac{\partial F_{(2)}}{\partial \dot{y}} & \frac{\partial F_{(2)}}{\partial \dot{z}} & \frac{\partial F_{(2)}}{\partial \dot{\phi}} & \frac{\partial F_{(2)}}{\partial \dot{\theta}} & \frac{\partial F_{(2)}}{\partial \dot{\psi}} \\ \frac{\partial F_{(3)}}{\partial \dot{x}} & \frac{\partial F_{(3)}}{\partial \dot{y}} & \frac{\partial F_{(3)}}{\partial \dot{z}} & \frac{\partial F_{(3)}}{\partial \dot{\phi}} & \frac{\partial F_{(3)}}{\partial \dot{\theta}} & \frac{\partial F_{(3)}}{\partial \dot{\psi}} \\ \frac{\partial F_{(4)}}{\partial \dot{x}} & \frac{\partial F_{(4)}}{\partial \dot{y}} & \frac{\partial F_{(4)}}{\partial \dot{z}} & \frac{\partial F_{(4)}}{\partial \dot{\phi}} & \frac{\partial F_{(4)}}{\partial \dot{\theta}} & \frac{\partial F_{(4)}}{\partial \dot{\psi}} \\ \frac{\partial F_{(5)}}{\partial \dot{x}} & \frac{\partial F_{(5)}}{\partial \dot{y}} & \frac{\partial F_{(5)}}{\partial \dot{z}} & \frac{\partial F_{(5)}}{\partial \dot{\phi}} & \frac{\partial F_{(5)}}{\partial \dot{\theta}} & \frac{\partial F_{(5)}}{\partial \dot{\psi}} \\ \frac{\partial F_{(6)}}{\partial \dot{x}} & \frac{\partial F_{(6)}}{\partial \dot{y}} & \frac{\partial F_{(6)}}{\partial \dot{z}} & \frac{\partial F_{(6)}}{\partial \dot{\phi}} & \frac{\partial F_{(6)}}{\partial \dot{\theta}} & \frac{\partial F_{(6)}}{\partial \dot{\psi}} \end{pmatrix}^T \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \lambda_6 \end{pmatrix} \quad (5.10)$$

Again this matrix can simplify considerably because the state equations don't depend on all of the state variable time derivatives.

$$0 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{\partial F_{(4)}}{\partial \dot{\phi}} & \frac{\partial F_{(4)}}{\partial \dot{\theta}} & \frac{\partial F_{(4)}}{\partial \dot{\psi}} \\ 0 & 0 & 0 & \frac{\partial F_{(5)}}{\partial \dot{\phi}} & \frac{\partial F_{(5)}}{\partial \dot{\theta}} & \frac{\partial F_{(5)}}{\partial \dot{\psi}} \\ 0 & 0 & 0 & \frac{\partial F_{(6)}}{\partial \dot{\phi}} & \frac{\partial F_{(6)}}{\partial \dot{\theta}} & \frac{\partial F_{(6)}}{\partial \dot{\psi}} \end{pmatrix}^T \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \lambda_6 \end{pmatrix} \quad (5.11)$$

$$0 = \begin{pmatrix} \frac{\partial F_{(4)}}{\partial \dot{\phi}} & \frac{\partial F_{(4)}}{\partial \dot{\theta}} & \frac{\partial F_{(4)}}{\partial \dot{\psi}} \\ \frac{\partial F_{(5)}}{\partial \dot{\phi}} & \frac{\partial F_{(5)}}{\partial \dot{\theta}} & \frac{\partial F_{(5)}}{\partial \dot{\psi}} \\ \frac{\partial F_{(6)}}{\partial \dot{\phi}} & \frac{\partial F_{(6)}}{\partial \dot{\theta}} & \frac{\partial F_{(6)}}{\partial \dot{\psi}} \end{pmatrix}^T \begin{pmatrix} \lambda_4 \\ \lambda_5 \\ \lambda_6 \end{pmatrix} \quad (5.12)$$

As with the other optimality conditions, the partial derivatives in this matrix must be computed numerically as follows:

$$\frac{\partial F_i}{\partial \dot{q}_j} \approx \frac{F_i(\dot{q}_j + \alpha) - F_i(\dot{q}_j)}{\alpha} \quad (5.13)$$

5.3 Stationarity Conditions

The stationarity conditions express the relationship between the derivatives of the system equation with respect to the input u , the co-state variable $\lambda(t)$, and the derivative of the Lagrangian with respect to the input.

$$\left(\frac{\partial H}{\partial u} \right)^T = \left(\frac{\partial F}{\partial u} \right)^T \lambda + \left(\frac{\partial L}{\partial u} \right)^T = 0 \quad (5.14)$$

$$\frac{\partial F}{\partial q} = \begin{pmatrix} \frac{\partial F_{(1)}}{\partial u_1} & \frac{\partial F_{(1)}}{\partial u_2} & \frac{\partial F_{(1)}}{\partial u_3} & \frac{\partial F_{(1)}}{\partial u_4} \\ \frac{\partial F_{(2)}}{\partial u_1} & \frac{\partial F_{(2)}}{\partial u_2} & \frac{\partial F_{(2)}}{\partial u_3} & \frac{\partial F_{(2)}}{\partial u_4} \\ \frac{\partial F_{(3)}}{\partial u_1} & \frac{\partial F_{(3)}}{\partial u_2} & \frac{\partial F_{(3)}}{\partial u_3} & \frac{\partial F_{(3)}}{\partial u_4} \\ \cdot & & & \\ \cdot & & & \\ \cdot & & & \\ \frac{\partial F_{(6)}}{\partial u_1} & \frac{\partial F_{(6)}}{\partial u_2} & \frac{\partial F_{(6)}}{\partial u_3} & \frac{\partial F_{(6)}}{\partial u_4} \end{pmatrix} \quad (5.15)$$

$$\frac{\partial L}{\partial u} = 2(u_1, u_2, u_3, u_4) \quad (5.16)$$

$$0 = \begin{pmatrix} \frac{\partial F_{(1)}}{\partial u_1} & \frac{\partial F_{(2)}}{\partial u_1} & \frac{\partial F_{(3)}}{\partial u_1} & \frac{\partial F_{(4)}}{\partial u_1} & \frac{\partial F_{(5)}}{\partial u_1} & \frac{\partial F_{(6)}}{\partial u_1} \\ \frac{\partial F_{(1)}}{\partial u_2} & \frac{\partial F_{(2)}}{\partial u_2} & \frac{\partial F_{(3)}}{\partial u_2} & \frac{\partial F_{(4)}}{\partial u_2} & \frac{\partial F_{(5)}}{\partial u_2} & \frac{\partial F_{(6)}}{\partial u_2} \\ \frac{\partial F_{(1)}}{\partial u_3} & \frac{\partial F_{(2)}}{\partial u_3} & \frac{\partial F_{(3)}}{\partial u_3} & \frac{\partial F_{(4)}}{\partial u_3} & \frac{\partial F_{(5)}}{\partial u_3} & \frac{\partial F_{(6)}}{\partial u_3} \\ \frac{\partial F_{(1)}}{\partial u_4} & \frac{\partial F_{(2)}}{\partial u_4} & \frac{\partial F_{(3)}}{\partial u_4} & \frac{\partial F_{(4)}}{\partial u_4} & \frac{\partial F_{(5)}}{\partial u_4} & \frac{\partial F_{(6)}}{\partial u_4} \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \\ \lambda_6 \end{pmatrix} + 2 \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{pmatrix} \quad (5.17)$$

Again, the partials are computed with a finite difference.

$$\frac{\partial F_i}{\partial u_j} \approx \frac{F_i(u_j + \alpha) - F_i(u_j)}{\alpha} \quad (5.18)$$

5.4 Discretization

In a real implementation the measurements and subsequent state estimates, which are the input to the control algorithm, are made available at discrete time intervals. In order to code a simulation and evaluate the behavior of this system of equations, it is more convenient if these equations are represented in a discrete-time form. First order derivatives are approximated as a first backward finite difference. By using backward finite differences, the causality of the expressions is preserved.

$$\dot{x} \approx \frac{x[k] - x[k-1]}{h} \quad (5.19)$$

The second-order time derivatives are approximated as second-order backward finite differences.

$$\ddot{x} \approx \frac{x[k] - 2x[k-1] + x[k-2]}{h^2} \quad (5.20)$$

The continuous system equations are given as follows.

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} + \frac{\mathcal{T}}{m} \begin{pmatrix} C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi S_\theta C_\phi - C_\psi S_\phi \\ C_\theta C_\phi \end{pmatrix} \quad (5.21)$$

$$\begin{pmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{pmatrix} = J^{-1} \left[\begin{pmatrix} ls(-\omega_2^2 + \omega_4^2) \\ ls(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^4 b\omega_i^2 \end{pmatrix} - \mathfrak{C} \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} \right] \quad (5.22)$$

The discrete-time system equations are

$$\begin{pmatrix} \frac{x[k]-2x[k-1]+x[k-2]}{h^2} \\ \frac{y[k]-2y[k-1]+y[k-2]}{h^2} \\ \frac{z[k]-2z[k-1]+z[k-2]}{h^2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} + \frac{\mathcal{T}[k]}{m} \begin{pmatrix} C_{\psi[k]} S_{\theta[k]} C_{\phi[k]} + S_{\psi[k]} S_{\phi[k]} \\ S_{\psi[k]} S_{\theta[k]} C_{\phi[k]} - C_{\psi[k]} S_{\phi[k]} \\ C_{\theta[k]} C_{\phi[k]} \end{pmatrix} \quad (5.23)$$

$$\begin{pmatrix} \frac{\phi[k]-2\phi[k-1]+\phi[k-2]}{h^2} \\ \frac{\theta[k]-2\theta[k-1]+\theta[k-2]}{h^2} \\ \frac{\psi[k]-2\psi[k-1]+\psi[k-2]}{h^2} \end{pmatrix} = J^{-1}[k] \left[\begin{pmatrix} ls(-\omega_2[k]^2 + \omega_4[k]^2) \\ ls(-\omega_1[k]^2 + \omega_3[k]^2) \\ \sum_{i=1}^4 b\omega_i[k]^2 \end{pmatrix} - \mathfrak{C}[k] \begin{pmatrix} \frac{\phi[k]-\phi[k-1]}{h} \\ \frac{\theta[k]-\theta[k-1]}{h} \\ \frac{\psi[k]-\psi[k-1]}{h} \end{pmatrix} \right] \quad (5.24)$$

The reason for computing all the partial derivatives numerically is now apparent.

To compute the partial derivatives analytically, one would have to deal with the products between the rows of the Coriolis matrix (Equation 3.23) and the columns of the inverse of the Jacobian matrix (Equation 3.12). This sort of computation is the very reason computers were invented in the first place.

5.5 A Finite Difference Solution to the Quad-Rotor Boundary Value Problem

The finite difference method for solving boundary value problems was introduced at the end of Chapter 4. This method reduces our optimal control problem to solving a system of nonlinear algebraic equations. This is a reduction in theoretical complexity but a dramatic increase in computational complexity.

The script 'finiteDiffSolution.py' (Appendix F) implements the finite difference method in an attempt to solve the quad-rotor boundary value problem. Recall that the computational problem is posed as solving a system of nonlinear algebraic equations. This system of equations is composed of the state equations, the co-state equations, the stationarity conditions, the secondary algebraic conditions, and the boundary conditions. Each of these expressions is possibly a function of the state variables, the co-state variables and the control input. Solving this system becomes a significant task since the state, co-state, and control variables become the unknowns for each instance in time for which a solution to the boundary value problem is desired! In order to sufficiently represent the dynamics of the quad-rotor, on the order of thousands of time steps are necessary. To solve this nonlinear system, a straightforward steepest descent technique was used.

Algorithm 5.1 : Steepest Descent Algorithm

1. An objective function is formed out of the sum of the squared residuals of each equation in the system.
2. The gradient is computed as the list of partial derivatives of the objective function with respect to every unknown (every variable defined at each time instance). These partials are approximated as finite differences.
3. The vector of unknowns is 'moved' in the direction of the negative of the gradient.
4. The new value of the objective function as well as the gradient are evaluated with the new vector of unknowns.
5. The state of the minimization process is checked against appropriate convergence criteria.

Conceptually, this algorithm is relatively straightforward but it poses a significant computational challenge. With ten defined time steps, the algorithm ran for several hours before terminating. Additionally, with only ten time steps defined, the dynamics of the system over a period of several seconds of flight are not well represented. In a physical implementation, the motor speeds need to be updated at a rate on the order of 50 Hz at a minimum. Given these constraints, there would be no realistic way to implement the algorithm in this form on an embedded system, which was loosely included as one of our research objectives. The Python Implementation of the finite difference method is shown Appendix [F](#).

Instead, in the next chapter we turn to different methods of control and optimization. Control of the system will be achieved with PID expressions. The

optimization problem will be approached by appropriately manipulating the gains of the PID control laws in order to change the system behavior.

Chapter 6

PID/PD Control

Among the many methods available for mathematical control of the quad-rotor, a well-tuned PID controller offers both relative robustness and a simple mathematical representation. In this chapter we derive and test the PID control scheme for attitude and 3D position control of a quad-rotor.

6.1 Deriving the Control Expressions

The control of the quad-rotor requires three independent PID controllers for the x, y, and z directions. In addition, the attitude stability of the aircraft is accomplished by three independent PD controllers for each of the Euler angles (ϕ, θ, ψ) . It is assumed for the purpose of simulation that the input to the control expressions includes accurate knowledge of the system state. In other words, it is assumed that the process noise and the measurement noise are zero. Given the natural complexity of the system, inclusion of stochastic processes into the model is left for further work. As in [9] and [10], the control algorithm proceeds as follows.

Algorithm 6.1 : PID Control Algorithm

1. The position control expressions give the 'commanded' linear accelerations that are required to drive the system to the desired state.
2. Given the commanded linear accelerations, the necessary total thrust, pitch, and roll are determined.
3. The commanded torques about the three axes of the quad-rotor are given by PD controllers using the commanded yaw, pitch, and roll as angular set points.
4. Given the commanded total thrust and the commanded torques, the motor speeds can be determined.
5. Once the motor speeds are known, the system model can be used to obtain the updated state of the system.
6. Go to step 1.

Our goal in the following derivation is to arrive at expressions for the motor speeds that are required to drive the system to the desired state. The discrete-time PID control expressions are formulated using these vectors.

$$P_c = \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = \text{desired (commanded) set point location}$$
$$P = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \text{actual position at time step } k$$

The vector of commanded accelerations is given by \ddot{P}_c :

$$\ddot{P}_c = K_p(P_c - P) + K_i \sum_k (P_c - P) + K_d(\dot{P}_c - \dot{P}), \quad (6.1)$$

where

$$K_p = \begin{bmatrix} k_{px} \\ k_{py} \\ k_{pz} \end{bmatrix}, K_i = \begin{bmatrix} k_{ix} \\ k_{iy} \\ k_{iz} \end{bmatrix}, K_d = \begin{bmatrix} k_{dx} \\ k_{dy} \\ k_{dz} \end{bmatrix}. \quad (6.2)$$

Recall equation 3.16:

$$f = RT_B = m\ddot{\xi} - G. \quad (6.3)$$

This can be rearranged to give:

$$\ddot{P}_c = -ge_{inz} + \left(\frac{1}{m}\right) (Te_{qrz}) R. \quad (6.4)$$

where,

$$e_{inz} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad (\text{in the inertial reference frame})$$

$$e_{qrz} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \quad (\text{in the quad-rotor reference frame})$$

The angles ϕ and θ , and the total thrust T can be determined algebraically, assuming we know \ddot{P}_c and ψ .

$$R^T \left(\ddot{P}_c + g e_{inz} \right) = \left(\frac{1}{m} \right) (T e_{qrz}) \quad (6.5)$$

$$\begin{pmatrix} C_\psi C_\theta & S_\psi C_\theta & -S_\theta \\ C_\psi S_\theta S_\phi - S_\psi C_\phi & S_\psi S_\theta S_\phi + C_\psi C_\phi & C_\theta S_\phi \\ C_\psi S_\theta C_\phi + S_\psi S_\phi & S_\psi S_\theta C_\phi - C_\psi S_\phi & C_\theta C_\phi \end{pmatrix} \begin{pmatrix} a_x \\ a_y \\ a_z + g \end{pmatrix} X = \frac{1}{m} \begin{pmatrix} 0 \\ 0 \\ T \end{pmatrix} \quad (6.6)$$

The matrix equation above is then written as three independent scalar expressions.

$$a_x C_\psi C_\theta + a_y S_\psi C_\theta - S_\theta (a_z + g) = 0 \quad (6.7)$$

$$a_x (C_\psi S_\theta S_\phi - S_\psi C_\phi) + a_y (S_\psi S_\theta S_\phi + C_\psi C_\phi) + (a_z + g) C_\theta S_\phi = 0 \quad (6.8)$$

$$a_x (C_\psi S_\theta C_\phi + S_\psi S_\phi) + a_y (S_\psi S_\theta C_\phi - C_\psi S_\phi) + (a_z + g) C_\theta C_\phi = \left(\frac{T}{m} \right) \quad (6.9)$$

Next, we divide Equation (6.7) by C_θ and solve for θ as θ_c .

$$a_x C_\psi + a_y S_\psi + (a_z + g)(-\tan(\theta)) = 0 \quad (6.10)$$

$$\theta_c = \arctan \left(\frac{a_x C_\psi + a_y S_\psi}{a_z + g} \right) \quad (6.11)$$

Next, Equation (6.8) $\times S_\phi$ = Equation (6.9) $\times C_\phi$. The result is

$$\phi = \arcsin \left(\frac{a_x S_\psi - a_y C_\psi}{T/m} \right). \quad (6.12)$$

Square both sides of (6.5) and note that $R^T = R^{-1}$.

$$a_x^2 + a_y^2 + (a_z + g)^2 = \left(\frac{T}{m} \right)^2 \quad (6.13)$$

$$\left(\frac{T}{m}\right) = \sqrt{a_x^2 + a_y^2 + (a_z + g)^2} \quad (6.14)$$

This result is then substituted back in Equation (6.12) to give ϕ_c .

$$\phi_c = \arcsin \left(\frac{a_x S_\psi - a_y C_\psi}{\sqrt{a_x^2 + a_y^2 + (a_z + g)^2}} \right) \quad (6.15)$$

Using θ_c and ϕ_c as set points, we can write the PD angular control laws. The subscript ‘c’ stands for ‘commanded’.

$$\tau_{\phi c} = [k_{p\phi}(\phi_c - \phi) + k_{d\phi}(\dot{\phi}_c - \dot{\phi})]I_x \quad (6.16)$$

$$\tau_{\theta c} = [k_{p\theta}(\theta_c - \theta) + k_{d\theta}(\dot{\theta}_c - \dot{\theta})]I_y \quad (6.17)$$

$$\tau_{\psi c} = [k_{p\psi}(\psi_c - \psi) + k_{d\psi}(\dot{\psi}_c - \dot{\psi})]I_z \quad (6.18)$$

Given the commanded torques and the commanded total thrust, the commanded motor speeds can be obtained from the expressions (3.2) and (3.4) from section 3.3:

$$\omega_{1c} = \sqrt{\frac{T_c}{4k} - \frac{\tau_{\theta c}}{2kL} - \frac{\tau_{\psi c}}{4b}} \quad (6.19)$$

$$\omega_{2c} = \sqrt{\frac{T_c}{4k} - \frac{\tau_{\phi c}}{2kL} + \frac{\tau_{\psi c}}{4b}} \quad (6.20)$$

$$\omega_{3c} = \sqrt{\frac{T_c}{4k} + \frac{\tau_{\theta c}}{2kL} - \frac{\tau_{\psi c}}{4b}} \quad (6.21)$$

$$\omega_{4c} = \sqrt{\frac{T_c}{4k} + \frac{\tau_{\phi c}}{2kL} + \frac{\tau_{\psi c}}{4b}} \quad (6.22)$$

With the above results, we can summarize the control loop. The PID control expressions prescribe linear accelerations in each direction (x, y, z) which will drive

the system toward the desired position. The linear accelerations and knowledge of ψ are used to calculate the angles ϕ and θ , and the total thrust T . Given the angles and their time derivatives, the prescribed torques about the quad-rotor center of mass are given by PD control laws. Given the torques and the total thrust, the vector of motor speeds can be calculated. The implementation of the PID control block is listed in Appendix A.

In a physical implementation, after the motor speeds are updated, the state of the system would be estimated from whatever sensor data is available. The environmental context would dictate which type of sensor hardware would be appropriate. In a simulation context, we use the dynamical system model from Chapter 3 to evaluate the resulting motion of the system. In a sense, this process is just the inverse of the control loop. For further work, a random process could be included here to model sensor noise. This would give a nice simulation platform for evaluating the performance of a Kalman filter for estimating the state of the quad-rotor.

6.2 Testing the Control Scheme

With experimentally tuned control expressions, arbitrarily shaped, sub-optimal paths can be formed by updating the desired location periodically. The code which implements this functionality is listed in Appendix B. Figures 6.1 through 6.6 show the utility of the control scheme. It is important to note that in our simulations, the desired velocity at each of the ordered set points is zero. In words, the control algorithm is saying to the quad-rotor: 'Go to the desired location and hover until the set point is updated'. To design a path that includes set points with a non-zero desired velocity vector would require modification of the algorithm. The values of the constants that were used in the simulation are shown in Table 6.2.

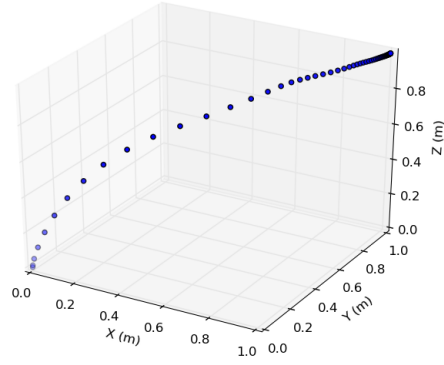


FIGURE 6.1: 3D path of the quad-rotor for a flight simulation between the points $(0,0,0)$ and $(1,1,1)$. The PID control expressions were tuned experimentally.

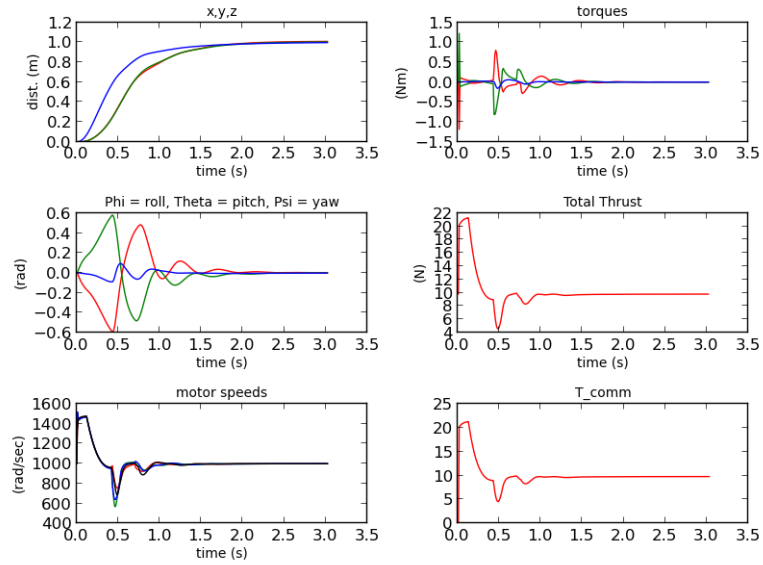


FIGURE 6.2: Time domain graphs corresponding the 3D path. The linear and angular state variables are shown to stabilize at the desired positions.

Simulation		
Parameters	Units	Description
$g = -9.81$	$\frac{m}{s^2}$	acceleration due to gravity
$m = 1$	kg	mass
$L = 1$	m	length of quad-rotor arm
$b = 10^{-6}$	$\frac{Nms^2}{Rad^2}$	aerodynamic torque coefficient
$k = 2.45 * 10^{-6}$	$\frac{Ns^2}{Rad^2}$	aerodynamic thrust coefficient
$Ixx = 5.0 * 10^{-3}$	$\frac{Nms^2}{Rad}$	moments of inertia
$Iyy = 5.0 * 10^{-3}$	$\frac{Nms^2}{Rad}$	
$Izz = 10.0 * 10^{-3}$	$\frac{Nms^2}{Rad}$	

TABLE 6.1: Simulation parameters of the quad-rotor model

Figure 6.3 shows the quad-rotor traversing along the edges of a 4 meter cube. This shows that in the simulation context, we have the ability to precisely locate the quad-rotor in space. The mathematical reality here is that the state of the system is exactly known within the algorithm. For a real implementation, the system model is replaced by the actual system. In this case the validity of the control algorithm is a function of the uncertainty of the state at each instance in time. This can be quantified by the state estimation process by which physical sensor measurements are combined. Figure 6.6 shows that there is an upper limit to the difference in initial and final vector positions. A single PID tuning is only usable up to a certain magnitude of desired displacement. Mathematically, the controller is still stable but the over-shoot of the desired position grows proportionately to the desired position itself. For arbitrarily shaped, long distance flights, the path would have to be composed of incremental pieces which are small enough so that a performance metric for the overshoot for each segment was satisfied. The time

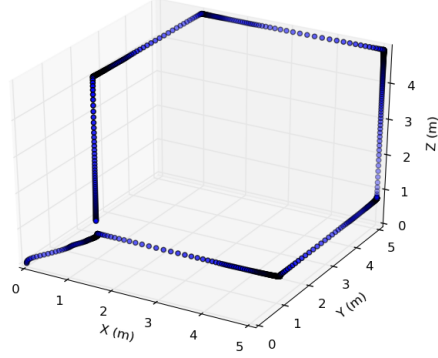


FIGURE 6.3: 3D path of the quad-rotor for an arbitrary path, tracing the edges of a cube.

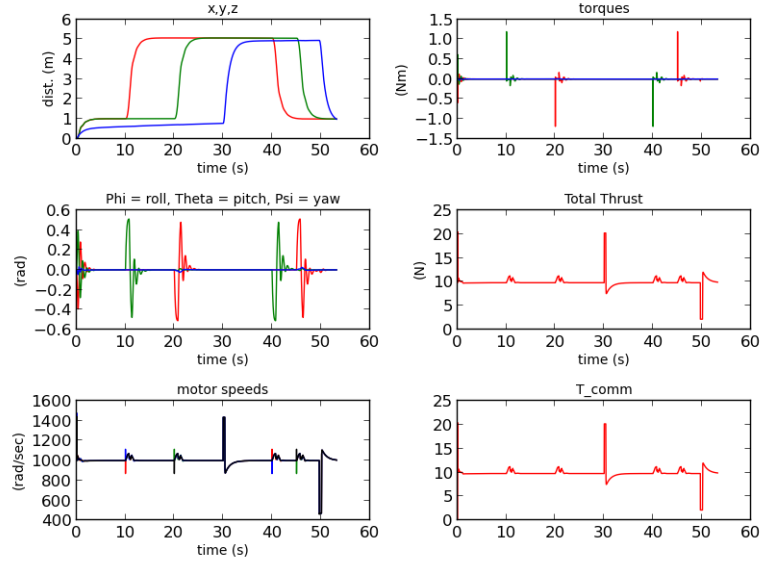


FIGURE 6.4: Dynamics of the rotor during the arbitrary path. The angular positions are shown to change and effect the desired change in linear position

domain plots (Figures 6.2, 6.4, and 6.6) offer information about the stability of the system and the controller. Small oscillations in the linear and angular positions and velocities can be seen in Figure 6.2. These oscillations are an artifact of the coupling between the angular and linear control laws. Intuitively, this makes sense because the control of the linear position requires that the angular state of the quad-rotor be destabilized. In general it is the natural instability of this system

which allows it to be so maneuverable. Also the mathematical complexity of the system makes for a difficult optimization problem. In the next chapter we use the PID tuning as a basis for optimizing the system according to specific performance criteria.

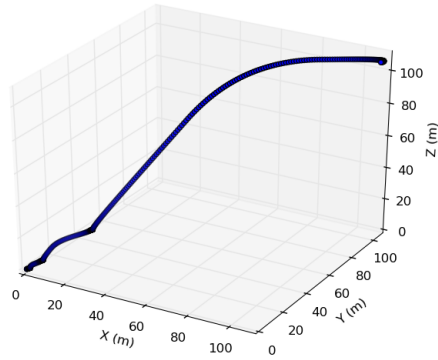


FIGURE 6.5: 3D path of the quad-rotor for a large set point.

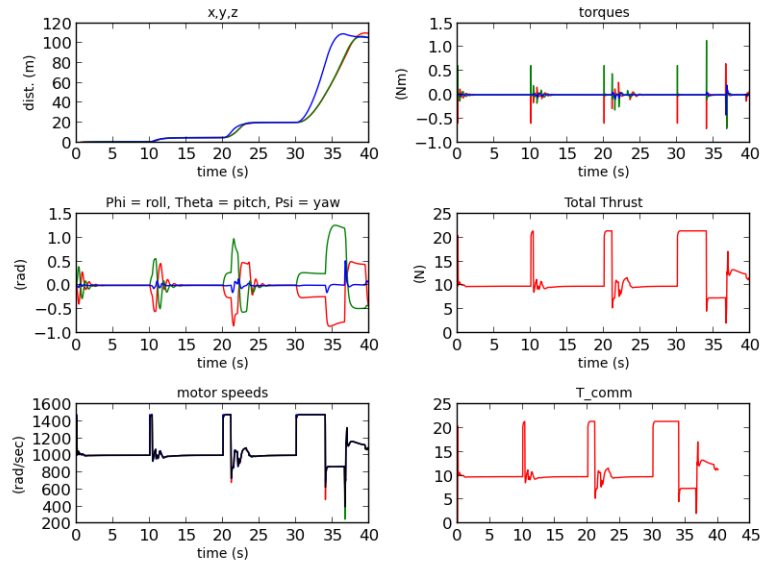


FIGURE 6.6: Dynamics of the quad-rotor for a large set point. The overshoot grows proportionally to the set point itself.

Chapter 7

PID Gain Optimization

In order to arrive at a solution to the quad-rotor energy optimization problem that is closer to running in real-time, a heuristic approach has been adopted. Recall that our general aim is to effectively control the vector position of the quad-rotor and additionally use the least energy in doing so. The relative simplicity of a PID controller makes it a good choice instead of the full nonlinear classical optimal control formulation. Also, if the PID control expressions are tuned well and that tuning is not changed, the control algorithm performs well.

The motivation for our heuristic method is to find the PID controller tuning which uses the least energy to drive the UAV to the desired state. The PID tuning defines the dynamics of the controller. Using the quad-rotor model derived in Chapter 3, the performance of the controller and the dynamics of the system can be evaluated as a function of the tuning. Mathematically, this can be represented as follows.

The aim of the optimization is to find:

$$\arg \min_{K_p, K_i, K_d} \sum_{k,i} \omega_i[k], \quad (7.1)$$

where $\omega_i[k]$ is the i th rotor speed at the k th time step. The variables K_p , K_i and K_d are the vectors of proportional, integral, and derivative gains defined as:

$$K_p = \begin{bmatrix} k_{px} \\ k_{py} \\ k_{pz} \end{bmatrix}, K_i = \begin{bmatrix} k_{ix} \\ k_{iy} \\ k_{iz} \end{bmatrix}, K_d = \begin{bmatrix} k_{dx} \\ k_{dy} \\ k_{dz} \end{bmatrix}$$

In our simulations, the time integral of all four motor speeds is proportional to the total energy used. Calculation of the actual energy used by the UAV in traversing a flight path would require a model for the motor. This is seen as unnecessary for our purposes since the time integral of the motor speeds and the total energy used will have the same effective minimum. Since the control input is calculated as part of the control algorithm anyway, it is used as a performance metric.

In any realistic application of UAV technology, the energy budget is only one important aspect of the control problem. Other important criteria for evaluating the performance of a controller are over-shoot of the desired location, the time of flight, and mathematical resonances or marginal instabilities. These factors must be considered in the design of the system. However, in the initial results described below, the time integral of the motor speeds is used as a single performance metric. The reason for this is to simplify the relationship between the performance metric and the PID gains. This is described in the next section.

7.1 Initial Simulation Results

In the context of the optimization process described in the previous section, there is evidence for the lack of robustness of the PID control. This can be shown by analyzing the relationship between the measured total thrust and the PID gains.

Ideally, a gradient descent method would be used to minimize the thrust as a function of the control gains. The basic flow of the algorithm that we would really like to implement is as follows.

Algorithm 7.1: PID Gain Optimization

1. Choose a set of proportional and derivative gains for each vector direction (x, y, and z);
2. Perform a simulation that controls the quad-rotor from an initial vector position to a desired vector position;
3. Calculate the sum of the four motor speeds over the duration of the simulation;
4. Appropriately change the PID gains such that the sum of the motor speeds decreases;
5. Go to step 1. Repeat until the sum of the motor speeds is found to be a minimum.

In reality, it has been found that the relationship between the measured total thrust and PID gains is not well-behaved. After many days of trying to debug a gradient descent algorithm, and observing inconsistent behavior, it was decided that a brute force method might be the only possibility. A deeper understanding of the objective function was needed. The brute force method simply requires that we simulate the system and determine the value of the performance criteria for each possible set of PID gain vectors within a given range.

In order to limit the number of simulations required to really represent the dynamics of the system, the set point $(0, 0, 1)$ was chosen. By choosing this set point, the

motion of the quad-rotor is intentionally limited to the z -direction which limits the number of possible gain vectors for this test. This makes the tuning of the x and y direction controllers irrelevant. To further limit the number of simulations required, the Ziegler-Nichols PID tuning method was used. This method is discussed in [30]. There are also excellent resources online to explain tuning [31].

The Ziegler-Nichols method specifies simple algebraic relationships between the proportional, integral, and derivative gains. This allows each of the PID gains to be expressed as a function of a single gain variable, k_u , which is allowed to range from 1 to 100.

The results of these simulations are shown in Figure 7.1.

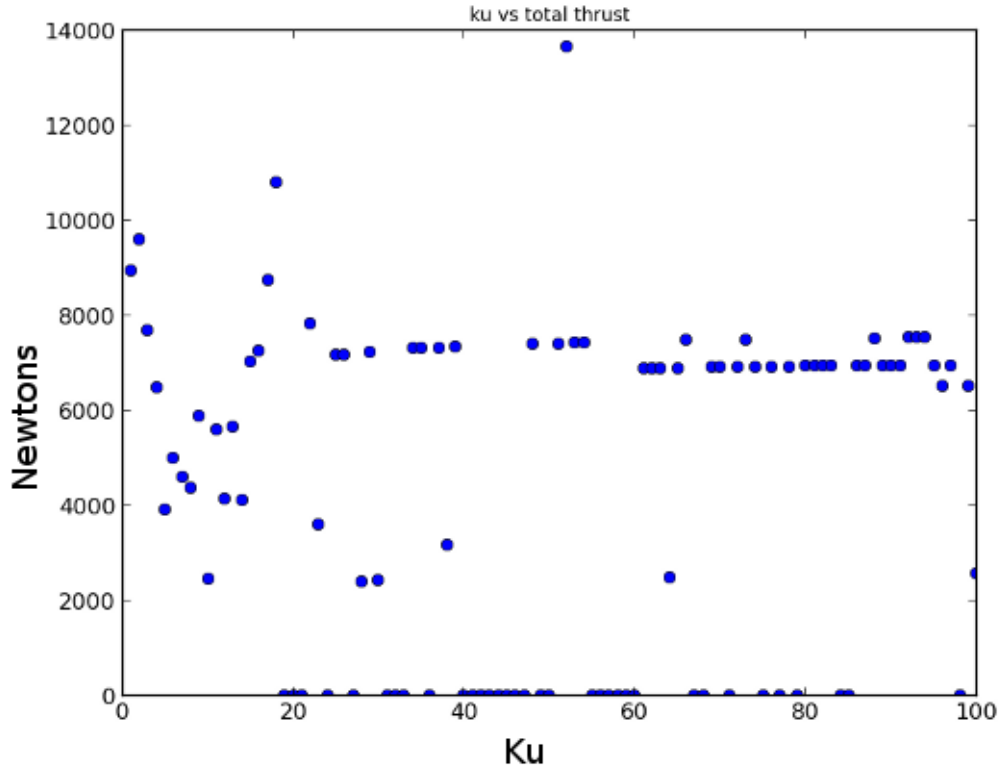


FIGURE 7.1: The relationship between K_u and the measured total thrust.

The relationship between Ku and the total measured thrust depicted in Figure 7.1 is rather disappointing to say the least. The values of zero total thrust are the result of simulations that failed to converge to the set point. Without an objective function that is at least marginally well behaved we have no hope to employ a gradient descent minimization technique. The relationship shown in Figure 7.1 is indeed a product of a deterministic system but displays little to no continuity.

Another detail which cannot be ignored is that the total thrust alone is not sufficient as a performance criteria. Additionally, for appropriate control of the UAV, the overshoot and oscillations which show up in improperly tuned PID controllers must be accounted for. Specifically, as the proportional gain is increased to drive the system to the desired location more quickly, the total thrust for the simulation will decrease but eventually the overshoot and oscillations grow to unacceptable levels. In the opposite fashion, if the differential gain is increased, the overshoot and oscillations will be suppressed but the time required to reach the set point will increase along with the total thrust. The competitive nature of these three (necessary) performance criteria further complicate the relationship between the PID gain vectors and the objective function making a gradient descent minimization technique even less viable.

7.2 Brute Force Simulation Results

Given that a standard mathematical optimization technique is out of the question, what options are left? A nonlinear control law could be implemented and would perhaps make the optimization possible, but this is uncertain and not possible in the current time frame.

The decision was made to try many possible gain vectors and have an appropriate objective function to characterize each of the simulations. Given that the brute force algorithm is easy to write (and massively parallel-izable), around 6000 simulations were performed over a period of several hours. The computation was delegated to six independent instantiations of a python script, each one taking on a subset of the possible gain vectors and a separate CPU. The code used in the brute force implementation is listed in Appendix [C](#), [D](#) and [E](#).

A separate script was used to parse through all the results of the simulations which were stored in many time-stamped files. Of the roughly 6000 simulations, about 1000 of them converged. For these, the set point was reached and the stopping criteria for the simulation were satisfied. Of the 1000 or so good runs, about 80 simulations satisfied the maximum overshoot and oscillation criteria. Only the gain vectors which produced less than ten percent overshoot were accepted. Likewise, only simulations which crossed the desired set point in each direction fewer than four times were deemed acceptable. The remaining 80 simulations were sorted lowest to highest by total measured thrust.

The optimal run that was found is detailed in Table [7.2](#). In addition to the PID gain vectors that produced the optimal system behavior, the measured performance metrics are also shown. The total thrust, set-point over shoot, and marginal instability of the system are quantified. The marginal instability is characterized by how many times the quad-rotor crossed the set-point value. Table [7.2](#) summarizes the statistics of the brute force simulation results. We can readily see that the optimal run is the run requiring the minimum thrust, and hence the minimum energy of all convergent runs. Figures [7.2](#) and [7.3](#) show the dynamics of the system with the optimal PID tuning.

Parameter	Value
kpx	15
kpy	15
kpz	40
kix	0.8
kiz	0.8
kix	0.8
kiz	15
kdx	10
kdy	10
kdz	50
ending iteration	987
discrete time step	0.01 (s)
flight time	9.87 (s)
cpu runtime	11.41 (s)
return value	1 (great success)
initial position	[0, 0, 1] (m)
set point	[1, 1, 2] (m)
total thrust	4607 (Newton seconds)
x crossings	3
x overshoot	0.0249 (m)
y crossings	1
y overshoot	0.0185 (m)
z crossings	1
z overshoot	0.0992 (m)

TABLE 7.1: Numerical descriptors of the optimal run.

Parameter	Value
number of convergent runs	1001
minimum thrust	4607(Newton seconds)
average total thrust	8075(Newton seconds)
number of runs with satisfactory oscillations	86
number of runs with satisfactory overshoot	91
average x crossings	3
average y crossings	3
average z crossings	1
average x overshoot	0.088(m)
average y overshoot	0.112(m)
average z overshoot	0.309(m)

TABLE 7.2: Statistics of the brute force simulations.

Figures 7.2 and 7.3 show the simulation of the quad-rotor flight from the initial point $(0, 0, 1)$ to the desired point $(1, 1, 2)$ using the optimal PID tuning. There are actually two distinct legs to the simulation. The reason for this comes from the fact that there are two types of initial conditions which have fundamental differences. There is a hovering state which we use as initial conditions for the simulations in the optimization procedure. There is another mathematical state which occurs at the very beginning of the simulation. The mathematical initialization of the quad-rotor state is at the origin $(0, 0, 0)$ with zero velocity and acceleration but it is not exactly hovering. This subtle distinction comes from the fact that the force of gravity is not canceled out by the thrust initially. The controller has had no time to act to stabilize the system. The physical scenario that this condition would correspond to is if a person held the quad-rotor at the initial position in free

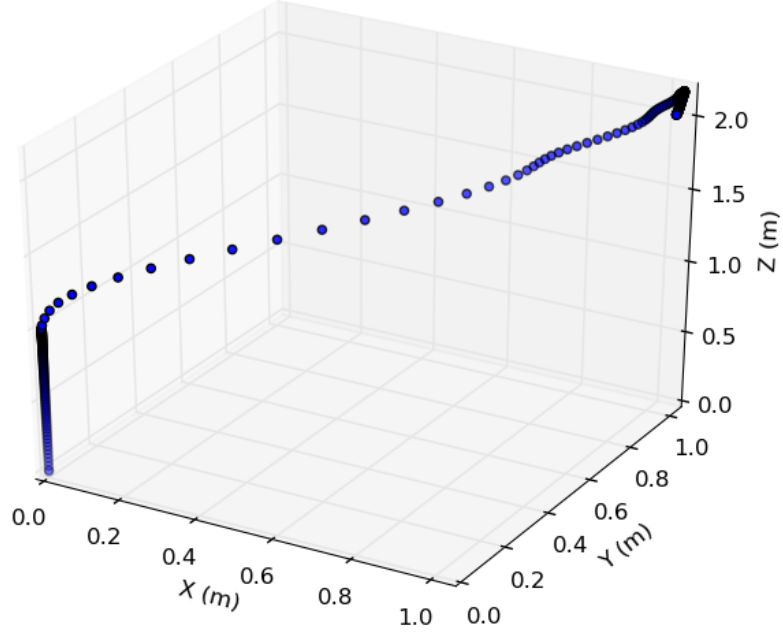


FIGURE 7.2: 3D path simulation of the Quadcopter using the optimal run parameters specified.

space (perhaps designated as the origin) and then at $t = 0$, simply let go. Another way to state this is that the simulations do not account for the normal force which would be imparted to the quad-rotor if it were simply taking off from the ground. To account for this would perhaps require augmentation of the dynamical model of the quad-rotor.

Our aim was to start and end in a hovering state for the optimization. This is why our simulations start with a 'take-off sequence' where the quad-rotor leaves from the origin and goes to the position $(0,0,1)$. After this, the system is stabilized in a hovering state. From there, arbitrary paths can be incrementally constructed by simply redefining the desired location.

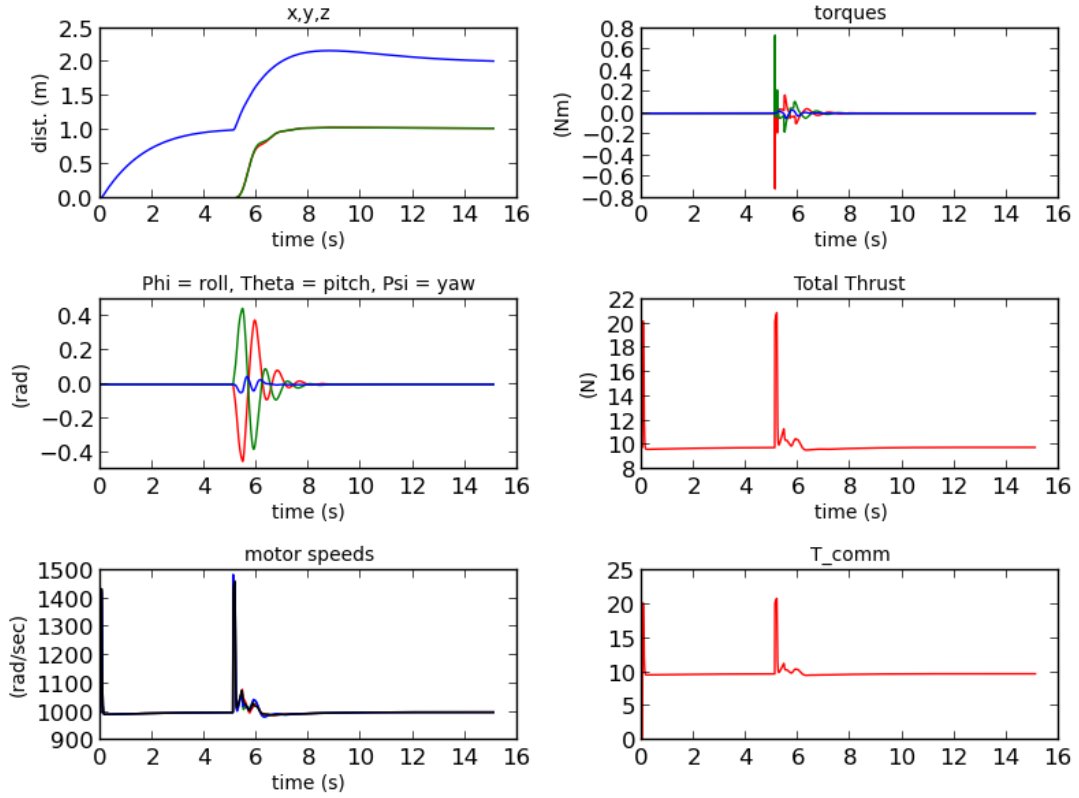


FIGURE 7.3: Quadcopter dynamics using the optimal run parameters.

Despite the mathematical difficulties that were experienced with the various optimization techniques that were explored, this final result is useful. Until a reasonable mathematical optimization technique is found, the optimality of the solution described above is a function of how much time one is willing to run the brute force algorithm.

Chapter 8

Summary and Future Work

8.1 Summary

In this final chapter we review the main points of the paper and propose directions for further work. Our first significant result was the dynamic model of the quadrotor. The Euler-Lagrange formulation was used to derive the dynamic model. The resulting set of nonlinear differential equations formed the basis for both the control and optimization methods which were subsequently derived.

Our first approach to the path-energy optimization problem which incorporated the dynamic model was classical optimal control. Using this formulation, we derived a set of differential and algebraic equations which form a complex boundary value problem. Our initial aim was to develop a method for achieving a path-energy optimization which would perform on a near-real-time-schedule. The computational resources needed to solve the optimal control boundary value problem on this real-time schedule invalidate it as a possible solution.

The next method of optimization which was explored was a heuristic method. Control of the UAV was attained by way of a set of PID and PD controllers. Since the performance of the quad-rotor is defined by the controller tuning, the system can be optimized as a function of this tuning. Relevant criteria for evaluating the performance of the system are the total thrust integrated over the duration of the simulation, oscillations that the system experiences, overshoot of the desired location, and the total time of flight. It was determined experimentally that the relationship between these performance metrics and the controller tuning was not well-behaved mathematically. Without a clean mathematical representation of our objective function, the viability of an efficient optimization method is questioned.

Next, a brute force method was used to determine the optimal controller tuning. Using this method, a controller tuning which is sufficient to perform simulated flights in a relatively efficient manner was determined.

8.2 Further Work

The problem of path-energy optimization as solved in this thesis can be worked on in the future to produce more robust results. Two possible mathematical modifications that could possibly allow for an efficient optimization algorithm are as follows. They both have design trade-offs.

1. *Model Linearization* One approach would be to use a linearized model. The goal there would be to simplify the relationship between the controller tuning and the relevant performance metrics by simplifying the mathematical

representation of the dynamic model. The danger in using a linear approximation is creating an over-simplified model that is divergent from reality to an extent that makes it unusable.

2. *Nonlinear Control* If instead of a linear PID controller, a nonlinear control method was used, the stability of the system could be increased. This would perhaps allow for a well behaved and more well defined relationship between the parameters of the controller and the measurable dynamics of the system. The obvious caveat here is the increase in complexity of the controller. The only real way to know if one of these options would allow for an efficient optimization procedure would be to try them all and characterize them in the context of the goal of the flight.
3. *UAV Swarm Optimization* Another area of research interest that would benefit from an energy optimization procedure is the control of swarms of UAVs. The distributed control of UAVs is a field which offers a wide range of engineering challenges such as collision avoidance, optimization of networked communication, and optimization of workload delegation toward a common goal. The contextual details of the cooperative aim of the swarm would inform the optimization of the system.
4. *Sensor Fusion and State Estimation* Yet another area of possible further research is in the sensor fusion and state estimation problem. For a physical implementation, knowledge of the state of the system is a critical component. Given that there are a very large variety of physical sensors that could contribute to this knowledge, this is an interesting problem. An aspect of this problem that adds richness to this situation is the fact that each type of sensor will have a different rate at which physical information is available. This rate is determined by the physical nature of the quantity being measured.

A good example is the integration of GPS measurements and accelerometer measurements. Values from these sensors are available perhaps at rates of 1 Hz and 100 Hz respectively. The sensor data would be integrated with a Kalman filter to develop situational awareness which allows for effective control of the system.

In general the control and optimization of UAVs is a rich and evolving field of research with many unsolved problems. There will be many commercial applications of this technology appearing in coming years which will be informed by future research.

Appendix A

Dynamic system model and PID control - agentModule.py

```
breakatwhitespace
1
2 '''
3 here is an outline of the program flow:
4
5     1) initialize lists for state variables and appropriate derivatives
6         as well as constants.
7     2) calculate total thrust T and the torques using the state variables from
8         the [k]th time step
9     3) using T[k] and tao_[k] calculate new motor speed vector for time [k]
10    4) using T[k] and tao_[k] calculate new state variables at time [k+1]
11    5) repeat
12 '''
13
14 from numpy import cos as c, sin as s , sqrt, array, dot, arctan2, arcsin, sign , around
15 from numpy.linalg import inv
16 from wind import *
17 import sys
18 import math
19
20 #####
21
22 class agent:
23
24     def __init__(self, x_0, y_0, z_0,
25                  initial_setpoint_x, initial_setpoint_y, initial_setpoint_z,
26                  agent_priority = 10 ):
27
```

```

29
31 #----- physical constants
self.g = -9.81 #[m/s^2]
33 self.m = 1 #[kg]
self.L = 1 #[m]
35 self.b = 10**-6
self.k = self.m*abs(self.g)/(4*(1000**2))
37 #print 'self.k = ',self.k
#raw_input()
39 #-----moments of inertia
self.Ixx = 5.0*10**-3
41 self.Iyy = 5.0*10**-3
self.Izz = 10.0*10**-3
43
45 #-----directional drag coefficients
self.Ax = 0.25
self.Ay = 0.25
47 self.Az = 0.25
49
self.hover_at_setpoint = True
51
# a list of distances to each other agent sorted nearest to farthest
self.distance_vectors = []
53
self.agent_priority = agent_priority
55
# state vector and derivative time series' list initializations
57 self.x = [x_0]
self.y = [y_0]
59 self.z = [z_0]
61
self.xdot = [0]
self.ydot = [0]
63 self.zdot = [0]
65
self.xddot = [0]
self.yddot = [0]
67 self.zddot = [0]
69
self.phi = [0]
self.theta = [0]
71 self.psi = [0]
73
self.phidot = [0]
self.thetadot = [0]
75 self.psidot = [0]
77
self.phiddot = [0]
self.thetaddot = [0]
79 self.psiddot = [0]
81
self.tao_qr_frame = []
self.etadot = []

```

```

83     self.etaddot =[]

85

87     #-----CONTROLLER GAINS

89     self.kpx = 0.          # PID proportional gain values
90     self.kpy = 0.
91     self.kpz = 0.

93     self.kdx = 0.          # PID derivative gain values
94     self.kdy = 0.
95     self.kdz = 0.

97     self.kddx = 0.
98     self.kddy = 0.
99     self.kddz = 0.

101    self.kix = 0.           # PID integral gain values
102    self.kiy = 0.
103    self.kiz = 0.

105    self.kpphi  = 4  # gains for the angular pid control laws
106    self.kptheta = 4
107    self.kppsi  = 4

109    self.kiphi  = 0.
110    self.kitheta = 0.
111    self.kipsi  = 0.

113    self.kdphi  = 5
114    self.kdtheta = 5
115    self.kdpsi  = 5

117    self.xdd_des = 0
118    self.ydd_des = 0
119    self.zdd_des = 0

121    self.x_integral_error = [0]
122    self.y_integral_error = [0]
123    self.z_integral_error = [0]

125    # angular set points
126    self.phi_des = 0
127    self.theta_des = 0
128    self.psi_des = 0

129    self.psidot_des = 0

131

133    self.phi_comm = [0,0]
134    self.theta_comm = [0,0]
135    self.T_comm = [0,0]

137    self.tao_phi_comm = [0,0]

```

```

139     self.tao_theta_comm = [0,0]
140     self.tao_psi_comm = [0,0]
141
142     # force, torque, and motor speed list initializations
143
144     self.T = [9.81]
145     self.tao_phi = [0]
146     self.tao_theta = [0]
147     self.tao_psi = [0]
148
149     self.w1 = [1000]
150     self.w2 = [1000]
151     self.w3 = [1000]
152     self.w4 = [1000]
153
154     self.etaddot = []
155     #-----
156
157     self.max_iterations = 10000
158     self.h = 0.01
159     self.ending_iteration = 0
160
161     #-----wind!!
162     self.wind_duration = self.max_iterations
163     self.max_gust = 0.1
164
165     # generate the wind for the entire simulation beforehand
166     self.wind_data = wind_vector_time_series(self.max_gust,self.wind_duration)
167
168     self.wind_x = self.wind_data[0]
169     self.wind_y = self.wind_data[1]
170     self.wind_z = self.wind_data[2]
171
172     #-----
173
174     #TODO: NEED TO SORT OUT THE MIN AND MAX THRUST PARAMETERS AND CORRELATE
175     #       THIS PHYSICAL LIMITATION WITH THE MAX VALUES FOR THE PROPORTIONAL
176     #       GAIN TERMS IN EACH CONTROL LAW.
177
178     #self.max_total_thrust = 50.0 # [newtons]
179     #self.min_total_thrust = 1.0
180
181     self.x_des = initial_setpoint_x
182     self.y_des = initial_setpoint_y
183     self.z_des = initial_setpoint_z
184
185     self.xdot_des = 0
186     self.ydot_des = 0
187     self.zdot_des = 0
188
189     self.initial_setpoint_x = initial_setpoint_x
190     self.initial_setpoint_y = initial_setpoint_y
191     self.initial_setpoint_z = initial_setpoint_z

```



```

193
195
197     self.w1_arg = [0,0]
199     self.w2_arg = [0,0]
201     self.w3_arg = [0,0]
203     self.w4_arg = [0,0]
205
207
209     self.xacc_comm = [0]
211     self.yacc_comm = [0]
213     self.zacc_comm = [19.62]
215
217
219
221
223
225
227
229
231
233
235
237
239
241
243
245
247
#-----
##### END " __INIT__() "
# the Jacobian for transforming from body frame to inertial frame
def J(self):
    ixx = self.Ixx
    iyy = self.Iyy
    izz = self.Izz
    th = self.theta[-1]
    ph = self.phi[-1]
    j11 = ixx
    j12 = 0
    j13 = -ixx * s(th)
    j21 = 0
    j22 = iyy*(c(ph)**2) + izz * s(ph)**2
    j23 = (iyy-izz)*c(ph)*s(ph)*c(th)
    j31 = -ixx*s(th)
    j32 = (iyy-izz)*c(ph)*s(ph)*c(th)
    j33 = ixx*(s(th)**2) + iyy*(s(th)**2)*(c(th)**2) + izz*(c(ph)**2)*(c(th)**2)
    return array([
        [j11, j12, j13],

```

```

249         [j21, j22, j23],
        [j31, j32, j33]
251     ])
253
255
257
259     #-----Coriolis matrix
261
263     def coriolis_matrix(self):
265
267         ph = self.phi[-1]
269         th = self.theta[-1]
271
273         phd = self.phidot[-1]
275         thd = self.thetadot[-1]
277         psd = self.psidot[-1]
279
281         ix = self.Ixx
283         iy = self.Iyy
285         iz = self.Izz
287
289         c11 = 0
291
293         # break up the large elements in to bite size chunks and then add each term ...
295
297         c12_term1 = (iy-iz) * ( thd*c(ph)*s(ph) + psd*c(th)*s(ph)**2 )
299
301         c12_term2and3 = (iz-iy)*psd*(c(ph)**2)*c(th) - ix*psd*c(th)
303
305         c12 = c12_term1 + c12_term2and3
307
309         c13 = (iz-iy) * psd * c(ph) * s(ph) * c(th)**2
311
313         c21_term1 = (iz-iy) * ( thd*c(ph)*s(ph) + psd*s(ph)*c(th) )
315
317         c21_term2and3 = (iy-iz) * psd * (c(ph)**2) * c(th) + ix * psd * c(th)
319
321         c21 = c21_term1 + c21_term2and3
323
325         c22 = (iz-iy)*phd*c(ph)*s(ph)
327
329         c23 = -ix*psd*s(th)*c(th) + iy*psd*(s(ph)**2)*s(th)*c(th)
331
333         c31 = (iy-iz)*phd*(c(th)**2)*s(ph)*c(ph) - ix*thd*c(th)
335
337
339
341
343
345
347
349
351
353
355
357
359
361
363
365
367
369
371
373
375
377
379
381
383
385
387
389
391
393
395
397
399
401

```

```

303     c32_term1      = ( izz-iyy)*( thd*c(ph)*s(ph)*s(th) + phd*(s(ph)**2)*c(th) )
305
306     c32_term2and3 = ( iyy-izz)*phd*(c(ph)**2)*c(th) + ixx*psd*s(th)*c(th)
307
308     c32_term4 = - iyy*psd*(s(ph)**2)*s(th)*c(th)
309
310     c32_term5 = - izz*psd*(c(ph)**2)*s(th)*c(th)
311
312     c32 = c32_term1 + c32_term2and3 + c32_term4 + c32_term5
313
314
315     c33_term1 = (iyy-izz) * phd *c(ph)*s(ph)*(c(th)**2)
316
317     c33_term2 = - iyy * thd*(s(ph)**2) * c(th)*s(th)
318
319     c33_term3and4 = - izz*thd*(c(ph)**2)*c(th)*s(th) + ixx*thd*c(th)*s(th)
320
321     c33 = c33_term1 + c33_term2 + c33_term3and4
322
323
324     return array([
325         [c11,c12,c13],
326         [c21,c22,c23],
327         [c31,c32,c33]
328     ])
329
330
331     #-----
332
333
334     def control_block(self):
335
336
337         # calculate the integral of the error in position for each direction
338
339         self.x_integral_error.append( self.x_integral_error[-1] + (self.x_des - self.x[-1])*self.h )
340         self.y_integral_error.append( self.y_integral_error[-1] + (self.y_des - self.y[-1])*self.h )
341         self.z_integral_error.append( self.z_integral_error[-1] + (self.z_des - self.z[-1])*self.h )
342
343
344         # compute the comm linear accelerations needed to move the system from present location to the
345         desired location
346
347         self.xacc_comm.append( self.kdx * (self.xdot_des - self.xdot[-1])
348                                + self.kpx * ( self.x_des - self.x[-1] )
349                                + self.kddx * (self.xdd_des - self.xddot[-1] )
350                                + self.kix * self.x_integral_error[-1] )
351
352         self.yacc_comm.append( self.kdy * (self.ydot_des - self.ydot[-1])
353                                + self.kpy * ( self.y_des - self.y[-1] )
354                                + self.kddy * (self.ydd_des - self.yddot[-1] )
355                                + self.kiy * self.y_integral_error[-1] )

```

```

357     self.zacc_comm.append( self.kdz * (self.zdot_des - self.zdot[-1])
359                             + self.kpz * ( self.z_des - self.z[-1] )
361                             + self.kddz * (self.zdd_des - self.zddot[-1] )
363                             + self.kiz * self.z_integral_error[-1] )
365
366     # need to limit the max linear acceleration that is perscribed by the control law
367
368     # as a meaningful place to start, just use the value '10m/s/s' , compare to g = -9.8 ...
369
370     max_latt_acc = 5
371
372     max_z_acc = 30
373
374     if abs(self.xacc_comm[-1]) > max_latt_acc: self.xacc_comm[-1] = max_latt_acc * sign(self.xacc_comm
375 [-1])
376     if abs(self.yacc_comm[-1]) > max_latt_acc: self.yacc_comm[-1] = max_latt_acc * sign(self.yacc_comm
377 [-1])
378     if abs(self.zacc_comm[-1]) > max_z_acc: self.zacc_comm[-1] = max_z_acc * sign(self.zacc_comm[-1])
379
380     min_z_acc = 12
381
382     if self.zacc_comm[-1] < min_z_acc: self.zacc_comm[-1] = min_z_acc
383
384     # using the comm linear accelerations, calc theta_c, phi_c and T_c
385
386     theta_numerator = (self.xacc_comm[-1] * c(self.psi[-1]) + self.yacc_comm[-1] * s(self.psi[-1]) )
387
388     theta_denominator = float( self.zacc_comm[-1] + self.g )
389
390     if theta_denominator <= 0:
391
392         theta_denominator = 0.1          # don't divide by zero !!!
393
394     self.theta_comm.append(arctan2( theta_numerator , theta_denominator ))
395
396     self.phi_comm.append(arcsin( (self.xacc_comm[-1] * s(self.psi[-1]) - self.yacc_comm[-1] * c(self.psi
397 [-1]) ) / float(sqrt( self.xacc_comm[-1]**2 +
398
399         self.yacc_comm[-1]**2 +
400
401         (self.zacc_comm[-1] + self.g)**2 )) ))
402
403     self.T_comm.append(self.m * ( self.xacc_comm[-1] * ( s(self.theta[-1])*c(self.psi[-1])*c(self.phi
404 [-1]) + s(self.psi[-1])*s(self.phi[-1]) ) +
405
406         self.yacc_comm[-1] * ( s(self.theta[-1])*s(self.psi[-1])*c(self.phi
407 [-1]) - c(self.psi[-1])*s(self.phi[-1]) ) +
408
409         (self.zacc_comm[-1] + self.g) * ( c(self.theta[-1])*c(self.phi[-1]) )
410
411     ))
412
413     if self.T_comm[-1] < 1.0:
414
415         self.T_comm = self.T_comm[:-1]
416
417         self.T_comm.append(1.0)

```

```

405     # we will need the derivatives of the comanded angles for the torque control laws.
407     self.phidot_comm = (self.phi_comm[-1] - self.phi_comm[-2])/self.h

409     self.thetadot_comm = (self.theta_comm[-1] - self.theta_comm[-2])/self.h

411

413     # solve for torques based on theta_c, phi_c and T_c , also psi_des , and previous values of theta,
    phi, and psi

415     tao_phi_comm_temp = ( self.kpphi*(self.phi_comm[-1] - self.phi[-1]) + self.kdphi*(self.phidot_comm -
    self.phidot[-1]) ) * self.Ixx

417     tao_theta_comm_temp = ( self.kptheta*(self.theta_comm[-1] - self.theta[-1]) + self.kdtheta*(self.
    thetadot_comm - self.thetadot[-1]) ) * self.Iyy

419     tao_psi_comm_temp = ( self.kppsi*(self.psi_des - self.psi[-1]) + self.kdpsi*( self.psidot_des - self.
    psidot[-1] ) ) * self.Izz

421     self.tao_phi_comm.append(tao_phi_comm_temp )
    self.tao_theta_comm.append(tao_theta_comm_temp )
423     self.tao_psi_comm.append(tao_psi_comm_temp )

425     #-----solve for motor speeds, eq 24

427     self.w1_arg.append( (self.T_comm[-1] / (4.0*self.k)) - ( self.tao_theta_comm[-1] / (2.0*self.k*self.L
    ) ) - ( self.tao_psi_comm[-1] / (4.0*self.b) ) )
    self.w2_arg.append( (self.T_comm[-1] / (4.0*self.k)) - ( self.tao_phi_comm[-1] / (2.0*self.k*self.L
    ) ) + ( self.tao_psi_comm[-1] / (4.0*self.b) ) )
429     self.w3_arg.append( (self.T_comm[-1] / (4.0*self.k)) + ( self.tao_theta_comm[-1] / (2.0*self.k*self.L
    ) ) - ( self.tao_psi_comm[-1] / (4.0*self.b) ) )
    self.w4_arg.append( (self.T_comm[-1] / (4.0*self.k)) + ( self.tao_phi_comm[-1] / (2.0*self.k*self.L
    ) ) + ( self.tao_psi_comm[-1] / (4.0*self.b) ) )

431     self.w1.append( sqrt( self.w1_arg[-1] ) )
433     self.w2.append( sqrt( self.w2_arg[-1] ) )
    self.w3.append( sqrt( self.w3_arg[-1] ) )
435     self.w4.append( sqrt( self.w4_arg[-1] ) )

437     # IMPORTANT!!! THIS ENDS THE 'CONTROLLER BLOCK' IN A REAL IMPLEMENTATION, WE WOULD NOW TAKE
    MEASUREMENTS AND ESTIMATE THE STATE and then start over...

439     def system_model_block(self):

441         # BELOW ARE THE EQUATIONS THAT MODEL THE SYSTEM,
        # FOR THE PURPOSE OF SIMULATION, GIVEN THE MOTOR SPEEDS WE CAN CALCULATE THE STATES OF THE SYSTEM

443

445         self.tao_qr_frame.append( array([
            self.L*self.k*( -self.w2[-1]**2 + self.w4[-1]**2 ) ,
            self.L*self.k*( -self.w1[-1]**2 + self.w3[-1]**2 ) ,
447             self.b*( -self.w1[-1]**2 + self.w2[-1]**2 - self.w3[-1]**2 + self.w4[-1]**2
        )

        ]) )
449

```

```

451 self.tao_phi.append(self.tao_qr_frame[-1][0])
452 self.tao_theta.append(self.tao_qr_frame[-1][1])
453 self.tao_psi.append(self.tao_qr_frame[-1][2])
454
455 self.T.append(self.k*( self.w1[-1]**2 + self.w2[-1]**2 + self.w3[-1]**2 + self.w4[-1]**2 ) )
456
457 # use the previous known angles and the known thrust to calculate the new resulting linear
458 # accelerations
459 # remember this would be measured ...
460 # for the purpose of modeling the measurement error and testing a kalman filter, inject noise here...
461 # perhaps every 1000ms substitute an artificial gps measurement (and associated uncertainty) for the
462 # double integrated imu value
463
464 self.xddot.append( (self.T[-1]/self.m)*( c(self.psi[-1])*s(self.theta[-1])*c(self.phi[-1])
465 + s(self.psi[-1])*s(self.phi[-1]) )
466 - self.Ax * self.xdot[-1] / self.m )
467
468 self.yddot.append( (self.T[-1]/self.m)*( s(self.psi[-1])*s(self.theta[-1])*c(self.phi[-1])
469 - c(self.psi[-1])*s(self.phi[-1]) )
470 - self.Ay * self.ydot[-1] / self.m )
471
472 self.zddot.append( self.g + (self.T[-1]/self.m)*( c(self.theta[-1])*c(self.phi[-1]) ) - self.Az *
473 self.zdot[-1] / self.m )
474
475 # calculate the new angular accelerations based on the known values
476 self.etadot.append( array( [self.phidot[-1], self.thetadot[-1], self.psidot[-1] ] ) )
477
478 self.etaddot.append( dot(inv( self.J() ), self.tao_qr_frame[-1] - dot(self.coriolis_matrix() , self
479 .etadot[-1]) ) )
480
481 # parse the etaddot vector of the new accelerations into the appropriate time series'
482 self.phiddot.append(self.etaddot[-1][0])
483
484 self.thetaddot.append(self.etaddot[-1][1])
485
486 self.psidot.append(self.etaddot[-1][2])
487
488 #----- integrate new acceleration values to obtain velocity values
489
490 self.xdot.append( self.xdot[-1] + self.xddot[-1] * self.h )
491 self.ydot.append( self.ydot[-1] + self.yddot[-1] * self.h )
492 self.zdot.append( self.zdot[-1] + self.zddot[-1] * self.h )
493
494 self.phidot.append( self.phidot[-1] + self.phiddot[-1] * self.h )
495 self.thetadot.append( self.thetadot[-1] + self.thetaddot[-1] * self.h )
496 self.psidot.append( self.psidot[-1] + self.psiddot[-1] * self.h )
497
498 #----- integrate new velocity values to obtain position / angle values
499
500 self.x.append( self.x[-1] + self.xdot[-1] * self.h )
501 self.y.append( self.y[-1] + self.ydot[-1] * self.h )
502 self.z.append( self.z[-1] + self.zdot[-1] * self.h )
503
504 self.phi.append( self.phi[-1] + self.phidot[-1] * self.h )
505 self.theta.append( self.theta[-1] + self.thetadot[-1] * self.h )

```

```

501         self.psi.append( self.psi[-1] + self.psidot[-1] * self.h )
503
505         #####
507
509     def plot_results(self, show_plot = True, save_plot = False, fig1_file_path = None, fig2_file_path = None):
511
512         timeSeries = [self.h*i for i in range(len(self.x))]
513
514         from mpl_toolkits.mplot3d import Axes3D
515         import matplotlib.pyplot as plt
516         from pylab import title
517         import matplotlib.gridspec as gridspec
518
519         fig0 = plt.figure()
520
521         ax = fig0.add_subplot(111, projection='3d')
522
523         ax.scatter(
524             self.x[0:len(self.x):5],
525             self.y[0:len(self.y):5],
526             self.z[0:len(self.z):5])
527
528         ax.set_xlabel('X (m)')
529         ax.set_ylabel('Y (m)')
530         ax.set_zlabel('Z (m)')
531
532
533         fig1 = plt.figure()
534         gs1 = gridspec.GridSpec(3, 2)
535
536         #-----linear displacements
537         xx = fig1.add_subplot(gs1[0,0])
538         plt.plot(timeSeries, self.x, 'r',
539                 timeSeries, self.y, 'g',
540                 timeSeries, self.z, 'b')
541         title('x,y,z', fontsize=10)
542         plt.xlabel('time (s)', fontsize=10)
543         plt.ylabel('dist. (m)', fontsize=10)
544
545         #-----angles
546         thth = fig1.add_subplot(gs1[1,0])
547         plt.plot(timeSeries, self.phi, 'r',
548                 timeSeries, self.theta, 'g',
549                 timeSeries, self.psi, 'b')
550         title('Phi = roll, Theta = pitch, Psi = yaw', fontsize=10)
551         plt.xlabel('time (s)', fontsize=10)
552         plt.ylabel('(rad)', fontsize=10)
553         #-----motor speeds
554
555

```

```

557     spd = fig1.add_subplot(gs1[2,0])
    plt.plot([self.h*i for i in range(len(self.w1))], self.w1,'r',
559             [self.h*i for i in range(len(self.w2))], self.w2,'g',
             [self.h*i for i in range(len(self.w3))], self.w3,'b',
561             [self.h*i for i in range(len(self.w4))], self.w4,'k')
    title('motor speeds',fontsize=10)
    plt.xlabel('time (s)',fontsize=10)
563     plt.ylabel('(rad/sec)',fontsize=10)
    #-----torque
565     spd = fig1.add_subplot(gs1[0,1])
    plt.plot([self.h*i for i in range(len(self.tao_phi))], self.tao_phi,'r',
567             [self.h*i for i in range(len(self.tao_theta))], self.tao_theta,'g',
             [self.h*i for i in range(len(self.tao_psi))], self.tao_psi,'b')
569     title('torques ',fontsize=10)
    plt.xlabel('time (s)',fontsize=10)
571     plt.ylabel('(Nm)',fontsize=10)
    #-----thrust
573     spd = fig1.add_subplot(gs1[1,1])
    plt.plot([self.h*i for i in range(len(self.T))], self.T,'r',)
575     title('Total Thrust',fontsize=10)
    plt.xlabel('time (s)',fontsize=10)
577     plt.ylabel('(N)',fontsize=10)

579
    #-----commanded total thrust
581     t_comm = fig1.add_subplot(gs1[2,1])
    plt.plot([self.h*i for i in range(len(self.T_comm))], self.T_comm,'r')
583     title('T_comm',fontsize=10)

585
587     '''
    #-----wind velocities
589     wind_plot = fig1.add_subplot( gs1[ 8:10 ,0:3 ] )

591     plt.plot([self.h*i for i in range(len(self.wind_x))], self.wind_x,'-r',
             [self.h*i for i in range(len(self.wind_y))], self.wind_y,'-g',
593             [self.h*i for i in range(len(self.wind_z))], self.wind_z,'-b' )

595     title('wind velocities, rgb = xyz', fontsize=10)
    '''
597     fig1.tight_layout()

599     #####
601
603     fig2 = plt.figure()
    gs2 = gridspec.GridSpec(4, 2)
605
    #-----linear velocities
607     lin_vel = fig2.add_subplot(gs2[0,0])
    plt.plot(timeSeries, self.xdot,'r',
609             timeSeries, self.ydot,'g',
             timeSeries, self.zdot,'b')

```



```

611     title('x_dot, y_dot, self.z_dot',fontSize=10)

613     #-----linear accelerations
lin_acc = fig2.add_subplot(gs2[1,0])
615     plt.plot(timeSeries, self.xddot,'r',
617             timeSeries, self.yddot,'g',
619             timeSeries, self.zddot,'b')
        title('xddot, yddot, self.zddot',fontSize=10)

621     #-----angular velocities
ang_vel = fig2.add_subplot(gs2[2,0])
623     plt.plot(timeSeries, self.phidot,'r',
625             timeSeries, self.thetadot,'g',
627             timeSeries, self.psidot,'b')
        title('phidot, thetadot, self.psidot',fontSize=10)

629     #-----commanded torques
t_comm = fig2.add_subplot(gs2[3,0])
631     plt.plot([self.h*i for i in range(len(self.tao_phi_comm))], self.tao_phi_comm,'r',
633             [self.h*i for i in range(len(self.tao_theta_comm))], self.tao_theta_comm,'g',
635             [self.h*i for i in range(len(self.tao_psi_comm))], self.tao_psi_comm,'b',
637             )
        title('tao_phi_comm,tao_theta_comm,tao_psi_comm',fontSize=10)

639     #-----angular velocities
ang_acc = fig2.add_subplot(gs2[0,1])
641     plt.plot(timeSeries, self.phiddot,'r',
643             timeSeries, self.thetaddot,'g',
645             timeSeries, self.psiddot,'b')
        title('phiddot, thetaddot, self.psiddot',fontSize=10)

647     #-----integral errors
integral_errors = fig2.add_subplot(gs2[1,1])
649     plt.plot(timeSeries, self.x_integral_error,'r',
651             timeSeries, self.y_integral_error,'g',
653             timeSeries, self.z_integral_error,'b')
        title('x_integral_error, y_integral_error, z_integral_error',fontSize=10)

655     #-----w_args
integral_errors = fig2.add_subplot(gs2[2,1])
657     plt.plot([self.h*i for i in range(len(self.w1_arg))], self.w1_arg,'r',
659             [self.h*i for i in range(len(self.w2_arg))], self.w2_arg,'g',
661             [self.h*i for i in range(len(self.w3_arg))], self.w3_arg,'b',
663             [self.h*i for i in range(len(self.w4_arg))], self.w4_arg,'k'
665             )
        title('w1_arg, w2_arg, w3_arg, w4_arg ',fontSize=10)

        #-----commanded phi and theta
integral_errors = fig2.add_subplot(gs2[3,1])

```

```

667     plt.plot([self.h*i for i in range(len(self.theta_comm))], self.theta_comm,'r',
669             [self.h*i for i in range(len(self.phi_comm))], self.phi_comm,'g'
671             )
673     title('theta_com ,phi_com ',fontsize=10)
675
677     fig2.tight_layout()
679
681     if save_plot == True:
683
685         fig1.savefig(fig1_file_path)
687         fig2.savefig(fig2_file_path)
689
691     if show_plot == True:
693
695         plt.show()
697
699 #-----
701
703     def print_dump(self,n=5):
705
707         print '\n\nself.xacc_comm[-n:] = ',around(self.xacc_comm[-n:], decimals=5)
709         print '\n\nself.yacc_comm[-n:] = ',around(self.yacc_comm[-n:], decimals=5)
711         print '\n\nself.zacc_comm[-n:] = ',around(self.zacc_comm[-n:], decimals=5)
713
715         print '\n\ntheta_com[-n:] = ',around(self.theta_comm[-n:], decimals=5)
717
719         print '\n\nphi_com[-n:] = ',around(self.phi_comm[-n:], decimals=5)
721
723         print '\n\nT_comm[-n:] = ',around(self.T_comm[-n:], decimals=5)
725
727         print '\n\nself.tao_phi_comm = ',around(self.tao_phi_comm[-n:], decimals=5)
729         print '\n\nself.tao_theta_comm = ',around(self.tao_theta_comm[-n:], decimals=5)
731         print '\n\nself.tao_psi_comm = ',around(self.tao_psi_comm[-n:], decimals=5)
733
735         print '\n\nw1_arg[-n:] = ',around(self.w1_arg[-n:], decimals=0)
737         print '\n\nw2_arg[-n:] = ',around(self.w2_arg[-n:], decimals=0)
739         print '\n\nw3_arg[-n:] = ',around(self.w3_arg[-n:], decimals=0)
741         print '\n\nw4_arg[-n:] = ',around(self.w4_arg[-n:], decimals=0)
743
745         print '\n\nw1[-n:] = ',around(self.w1[-n:], decimals=1)
747         print '\n\nw2[-n:] = ',around(self.w2[-n:], decimals=1)
749         print '\n\nw3[-n:] = ',around(self.w3[-n:], decimals=1)
751         print '\n\nw4[-n:] = ',around(self.w4[-n:], decimals=1)
753
755         print '\n\nself.tao_qr_frame[-n:] = ',around(self.tao_qr_frame[-n:], decimals=5)
757
759         print '\n\nself.T[-n:] = ',around(self.T[-n:], decimals=5)
761
763         print '\n\nself.phi[-n:] = ',around(self.phi[-n:], decimals=5)
765         print '\n\nself.theta[-n:] = ',around(self.theta[-n:], decimals=5)
767         print '\n\nself.psi[-n:] = ',around(self.psi[-n:], decimals=5)

```

```

721     print '\n\nself.phidot[-n:] = ',around(self.phidot[-n:], decimals=5)
723     print '\n\nself.thetadot[-n:] = ',around(self.thetadot[-n:], decimals=5)
725     print '\n\nself.psidot[-n:] = ',around(self.psidot[-n:], decimals=5)
727
729     print '\n\nself.phiddot[-n:] = ',around(self.phiddot[-n:], decimals=5)
731     print '\n\nself.thetaddot[-n:] = ',around(self.thetaddot[-n:], decimals=5)
733     print '\n\nself.psidot[-n:] = ',around(self.psidot[-n:], decimals=5)
735
737     print '\n\nself.x[-n:] = ',around(self.x[-n:], decimals=5)
739     print '\n\nself.y[-n:] = ',around(self.y[-n:], decimals=5)
741     print '\n\nself.z[-n:] = ',around(self.z[-n:], decimals=5)
743
745     print '\n\nself.xdot[-n:] = ',around(self.xdot[-n:], decimals=5)
747     print '\n\nself.ydot[-n:] = ',around(self.ydot[-n:], decimals=5)
749     print '\n\nself.zdot[-n:] = ',around(self.zdot[-n:], decimals=5)
751
753     print '\n\nself.xddot[-n:] = ',around(self.xddot[-n:], decimals=5)
755     print '\n\nself.yddot[-n:] = ',around(self.yddot[-n:], decimals=5)
757     print '\n\nself.zddot[-n:] = ',around(self.zddot[-n:], decimals=5)
759
761     print '\n\nself.x_integral_error[-n:] = ',around(self.x_integral_error[-n:], decimals=5)
763     print '\n\nself.y_integral_error[-n:] = ',around(self.y_integral_error[-n:], decimals=5)
765     print '\n\nself.z_integral_error[-n:] = ',around(self.z_integral_error[-n:], decimals=5)
767
769     print '\n\n'
771
773     #-----
775
777 if __name__ == "__main__":
779
781     a = agent(x_0 = 0,
783             y_0 = 0,
785             z_0 = 0,
787             initial_setpoint_x = 1,
789             initial_setpoint_y = 1,
791             initial_setpoint_z = 1,
793             agent_priority = 1)
795
797
799     #the following gains worked well for a setpoint of (1,1,1)
801
803     a.kpx = 40      # -----PID proportional gain values
805     a.kpy = 40
807     a.kpz = 40
809
811     a.kdx = 20      #-----PID derivative gain values
813     a.kdy = 20
815     a.kdz = 10
817
819     a.kix = .2
821     a.kiy = .2
823     a.kiz = 40
825
827     a.kpphi = 4 # gains for the angular pid control laws

```

```

777     a.kptheta = 4
       a.kppsi  = 4

779     a.kdphi   = 10
       a.kdtheta = 10
781     a.kdpsi   = 5

783
785
787
789
       for i in range(a.max_iterations):

791
           a.ending_iteration = i # preemptively...

793
           a.system_model_block()

795
           a.control_block()

797
           x_ave = sum(a.x[-100:])/100.0
           y_ave = sum(a.y[-100:])/100.0
799           z_ave = sum(a.z[-100:])/100.0

801
           xerr = a.x_des - x_ave
           yerr = a.y_des - y_ave
803           zerr = a.z_des - z_ave

805
           #if i%50 == 0:
807           print 'x_ave = ',x_ave
           print 'y_ave = ',y_ave
809           print 'z_ave = ',z_ave

           print 'i = ',i
           print 'xerr, yerr, zerr = ',xerr,',',yerr,',',zerr
813           print 'sqrt( xerr**2 + yerr**2 + zerr**2 ) = ',sqrt( xerr**2 + yerr**2 + zerr**2 )
           #a.print_dump(3)

815
817
819           # Stopping Criteria: if the agent is within a 5 cm error sphere for 200 time steps ( .2 sec )

821           if ( sqrt( xerr**2 + yerr**2 + zerr**2 ) < 10**-2) and (i>50):

823
               print 'set point reached!!'

825
               print 'i = ', i

827
               break

829           if ( sqrt( xerr**2 + yerr**2 + zerr**2 ) > 200) and (i>50):

```

```

831         print 'you are lost!!'

833     print 'i = ', i

835     break

837     k_th_variable_list = [
838         a.xacc_comm[-1],
839         a.yacc_comm[-1],
840         a.zacc_comm[-1],
841         a.theta_comm[-1],
842         a.phi_comm[-1],
843         a.T_comm[-1],
844         a.tao_phi_comm[-1],
845         a.tao_theta_comm[-1],
846         a.tao_psi_comm[-1],
847         a.w1_arg[-1],
848         a.w2_arg[-1],
849         a.w3_arg[-1],
850         a.w4_arg[-1],
851         a.w1[-1],
852         a.w2[-1],
853         a.w3[-1],
854         a.w4[-1],
855         a.tao_qr_frame[-1][0],
856         a.tao_qr_frame[-1][1],
857         a.tao_qr_frame[-1][2],
858         a.T[-1],
859         a.phi[-1],
860         a.theta[-1],
861         a.psi[-1],
862         a.phidot[-1],
863         a.thetadot[-1],
864         a.psidot[-1],
865         a.phiddot[-1],
866         a.thetaddot[-1],
867         a.psiddot[-1],
868         a.x[-1],
869         a.y[-1],
870         a.z[-1],
871         a.xdot[-1],
872         a.ydot[-1],
873         a.zdot[-1],
874         a.xddot[-1],
875         a.yddot[-1],
876         a.zddot[-1],
877         a.x_integral_error[-1],
878         a.y_integral_error[-1],
879         a.z_integral_error[-1],
880     ]

881     for k in k_th_variable_list:
882
883         if math.isnan(k):
884
885

```

```

887         a.print_dump(1)
889
891         break
893
895         if a.phi[-1] > 5: break
897         if a.theta[-1] > 5: break
899         if a.psi[-1] > 5: break
901
903         if math.isnan(xerr):
905
907             break
909
911     print '#####\n\n'
913
915     a.print_dump(10)
917
919     fig1_file_path = '/home/ek/Dropbox/THESIS/python_scripts/fig1_agent_module.png'
921     fig2_file_path = '/home/ek/Dropbox/THESIS/python_scripts/fig2_agent_module.png'
923
925     a.plot_results(False, True, fig1_file_path, fig2_file_path)

```

/home/ek/Dropbox/THESIS/python_scripts/agent_module.py

Appendix B

Module for updating the PID set point - waypointNavigation.py

```
breakatwhitespace

2 from agent_module import *

4 def go(agent):

6     for i in range(agent.max_iterations):

8         a.system_model_block()

10        a.control_block()

12        retval = stopping_criteria(agent)

14        if (retval == 0) or (retval == 1):

16            print 'i = ', i

18            break

20 #-----

22 def stopping_criteria(agent):

24     x_ave = sum(agent.x[-100:])/100.0
25     y_ave = sum(agent.y[-100:])/100.0
26     z_ave = sum(agent.z[-100:])/100.0
```

```

28     xerr = agent.x_des - x_ave
    yerr = agent.y_des - y_ave
30     zerr = agent.z_des - z_ave

32     #if i%50 == 0:
        #print 'x_ave = ',x_ave
34        #print 'y_ave = ',y_ave
        #print 'z_ave = ',z_ave
36
    print 'xerr, yerr, zerr = ',xerr,', ',yerr,', ',zerr
38    print 'sqrt( xerr**2 + yerr**2 + zerr**2 ) = ',sqrt( xerr**2 + yerr**2 + zerr**2 )

40    if ( sqrt( xerr**2 + yerr**2 + zerr**2 ) < 10**-2 ) and (len(agent.x) >50):

42        print 'set point reached!!'

44        #print 'i = ', i

46        return 1

48    if ( sqrt( xerr**2 + yerr**2 + zerr**2 ) > 200 ) and (i>50):

50        print 'you are lost!!'

52
54        return 0

    k_th_variable_list = [
56        a.xacc_comm[-1],a.yacc_comm[-1],a.zacc_comm[-1],
        a.theta_comm[-1],a.phi_comm[-1],a.T_comm[-1],
58        a.tao_phi_comm[-1],a.tao_theta_comm[-1],a.tao_psi_comm[-1],
        a.w1_arg[-1],a.w2_arg[-1],a.w3_arg[-1],a.w4_arg[-1],
60        a.w1[-1],a.w2[-1],a.w3[-1],a.w4[-1],
        a.tao_qr_frame[-1][0],a.tao_qr_frame[-1][1],a.tao_qr_frame[-1][2],
62        a.T[-1],
        a.phi[-1],a.theta[-1],a.psi[-1],
64        a.phidot[-1],a.thetadot[-1],a.psidot[-1],
        a.phiddot[-1],a.thetaddot[-1],a.psidot[-1],
66        a.x[-1],a.y[-1],a.z[-1],
        a.xdot[-1],a.ydot[-1],a.zdot[-1],
68        a.xddot[-1],a.yddot[-1],a.zddot[-1],
        a.x_integral_error[-1],a.y_integral_error[-1],a.z_integral_error[-1],
70        ]

72    for k in k_th_variable_list:

74        if math.isnan(k):

76            a.print_dump(1)

78            return 0

80
82    if a.phi[-1] > 5: return 0

```



```

84     if a.theta[-1] > 5: return 0

86     if a.psi[-1] > 5: return 0

88     if math.isnan(xerr): return 0

90     #####

92     if __name__ == '__main__':

94         a = agent(x_0 = 0,
95                 y_0 = 0,
96                 z_0 = 0,
97                 initial_setpoint_x = 1,
98                 initial_setpoint_y = 1,
99                 initial_setpoint_z = 1,
100                 agent_priority = 1)

102     #the following gains worked well for a setpoint of (1,1,1)

104     a.kpx = 15      # -----PID proportional gain values
105     a.kpy = 15
106     a.kpz = 50

108     a.kdx = 10      #-----PID derivative gain values
109     a.kdy = 10
110     a.kdz = 50

112     a.kix = 0.8
113     a.kiy = 0.8
114     a.kiz = 15

116     a.kpphi = 4 # gains for the angular pid control laws
117     a.kptheta = 4
118     a.kppsi = 4

120     a.kdphi = 6
121     a.kdtheta = 6
122     a.kdpsi = 6
123     '''
124     u'a0.ending_iteration': 257,          the optimal run
125     u'a0.kdx': 10,
126     u'a0.kdy': 10,
127     u'a0.kdz': 50,
128     u'a0.kix': 0.8,
129     u'a0.kiy': 0.8,
130     u'a0.kiz': 20,
131     u'a0.kpx': 15,
132     u'a0.kpy': 15,
133     u'a0.kpz': 40,
134     u'ith_runtime': 11.414505958557129,
135     u'return_val2': 1,
136     u'setpoint': [1, 1, 2],
137     u'total_thrust': 4969.888344602483,
138     u'x_crossings': 3,

```

```

138     u'x_over_shoot': 0.024969492375947366,
140     u'y_crossings': 1,
142     u'y_over_shoot': 0.01858534082026475,
144     u'z_crossings': 1,
146     u'z_over_shoot': 0.09928387955818296}
148
149
150
151     a.max_iterations = 1000
152
153     position_setpoint_list = [[0,0,1],[1,1,2]] # [[1,1,1],[5,5,5],[20,20,20],[100,100,100]]
154     # [[1,1,1],[5,1,1],[5,5,1],[5,5,5],[1,5,5],[1,1,5],[1,1,1]] #
155
156     for ss in range(len(position_setpoint_list)):
157
158         a.x_des = position_setpoint_list[ss][0]
159         a.y_des = position_setpoint_list[ss][1]
160         a.z_des = position_setpoint_list[ss][2]
161
162         go(a)
163
164     print '#####\n\n'
165
166     a.print_dump(10)
167
168     fig1_file_path = '/home/ek/Dropbox/THESIS/python_scripts/fig1_agent_module.png'
169     fig2_file_path = '/home/ek/Dropbox/THESIS/python_scripts/fig2_agent_module.png'
170
171     a.plot_results()

```

/home/ek/Dropbox/THESIS/python_scripts/waypoint_navigation.py

Appendix C

Functions used in the brute force method - bruteForceFunctions.py

```
breakatwhitespace
1
2 from agent_module import *
3
4 from numpy import mean
5
6 import json
7 #-----
8 #-----
9
10 def run(agent, plots = False):
11
12     for i in range(agent.max_iterations):
13
14         agent.ending_iteration = i
15
16         agent.system_model_block()
17
18         agent.control_block()
19
20
21         x_ave = sum(agent.x[-100:])/100.0
22         y_ave = sum(agent.y[-100:])/100.0
23         z_ave = sum(agent.z[-100:])/100.0
24
25         xerr = agent.x_des - x_ave
26         yerr = agent.y_des - y_ave
27         zerr = agent.z_des - z_ave
```

```

29     # Stopping Criteria: if the agent is within a n cm error sphere for 200 time steps ( .2 sec )
31
32     if ( sqrt( xerr**2 + yerr**2 + zerr**2 ) <10**-2 ) and (i>50):
33
34         print 'i = ', i
35
36         print 'set point reached'
37
38         return 1
39
40     if ( abs(zerr) > 200) and (i>50):
41
42         print 'i = ', i
43
44         print 'you are lost'
45
46         return 'err'
47
48     k_th_variable_list = [ agent.xacc_comm[-1],agent.yacc_comm[-1],agent.zacc_comm[-1],
49                           agent.theta_comm[-1],agent.phi_comm[-1],agent.T_comm[-1],
50                           agent.tao_phi_comm[-1],agent.tao_theta_comm[-1],agent.tao_psi_comm[-1],
51                           agent.w1_arg[-1],agent.w2_arg[-1],agent.w3_arg[-1],agent.w4_arg[-1],
52                           agent.w1[-1],agent.w2[-1],agent.w3[-1],agent.w4[-1],
53                           agent.tao_qr_frame[-1][0],agent.tao_qr_frame[-1][1],agent.tao_qr_frame
54                           [-1][2],
55                           agent.T[-1],
56                           agent.phi[-1],agent.theta[-1],agent.psi[-1],
57                           agent.phidot[-1],agent.thetadot[-1],agent.psidot[-1],
58                           agent.phiddot[-1],agent.thetaddot[-1],agent.psiddot[-1],
59                           agent.x[-1],agent.y[-1],agent.z[-1],
60                           agent.xdot[-1],agent.ydot[-1],agent.zdot[-1],
61                           agent.xddot[-1],agent.yddot[-1],agent.zddot[-1],
62                           agent.x_integral_error[-1],agent.y_integral_error[-1],agent.z_integral_error
63                           [-1],
64                           ]
65
66     for k in k_th_variable_list:
67
68         if math.isnan(k):
69
70             agent.print_dump(1)
71
72             return 'err'
73
74         if agent.phi[-1] > 5: break
75         if agent.theta[-1] > 5: break
76         if agent.psi[-1] > 5: break
77
78         if math.isnan(xerr):
79             print 'math.isnan(xerr) = True'
80
81             return 'err'
82
83     #-----

```

```

81 #-----
83 def take_off():
85     agent0 = agent(x_0 = 0,
86                     y_0 = 0,
87                     z_0 = 0,
88                     initial_setpoint_x = 0,
89                     initial_setpoint_y = 0,
90                     initial_setpoint_z = 1,
91                     agent_priority = 1)
92
93     agent0.max_iterations = 800
94
95     #the following gains worked well for a setpoint of (1,1,1)
96
97     agent0.kpx = 40      # -----PID proportional gain values
98     agent0.kpy = 40
99     agent0.kpz = 40
100
101     agent0.kdx = 25      #-----PID derivative gain values
102     agent0.kdy = 25
103     agent0.kdz = 40
104
105     agent0.kix = .2
106     agent0.kiy = .2
107     agent0.kiz = 40
108
109     run(agent0)
110
111     return agent0      # return the agent instance hovering at (0,0,1)
112
113 #-----
114 #-----
115 def test_gain_vector(a0, set_point, gain_dictionary):
116
117     a0.x_des = set_point[0]
118     a0.y_des = set_point[1]
119     a0.z_des = set_point[2]
120
121     a0.max_iterations = 1000
122
123     a0.kpx = gain_dictionary['kpxy']
124     a0.kix = gain_dictionary['kixy']
125     a0.kdx = gain_dictionary['kdx']
126
127     a0.kpy = gain_dictionary['kpy']
128     a0.kiy = gain_dictionary['kiy']
129     a0.kdy = gain_dictionary['kdy']
130
131     a0.kpz = gain_dictionary['kpz']
132     a0.kiz = gain_dictionary['kiz']
133     a0.kdz = gain_dictionary['kdz']
134
135     return_val2 = run(a0)

```

```

137     '''
138     #-----
139     variable_dictionary = {
140         'a0.xacc_comm' : a0.xacc_comm, 'a0.yacc_comm' : a0.yacc_comm, 'a0.zacc_comm' : a0.zacc_comm,
141         'a0.theta_comm' : a0.theta_comm, 'a0.phi_comm' : a0.phi_comm, 'a0.T_comm' : a0.T_comm,
142         'a0.tao_phi_comm' : a0.tao_phi_comm, 'a0.tao_theta_comm' : a0.tao_theta_comm, 'a0.tao_psi_comm'
143         : a0.tao_psi_comm,
144         'a0.w1_arg' : a0.w1_arg, 'a0.w2_arg' : a0.w2_arg, 'a0.w3_arg' : a0.w3_arg, 'a0.w4_arg' : a0.
145         w4_arg,
146         'a0.w1' : a0.w1, 'a0.w2' : a0.w2, 'a0.w3' : a0.w3, 'a0.w4' : a0.w4,
147         'a0.tao_qr_frame[0]' : a0.tao_qr_frame[0].tolist(), 'a0.tao_qr_frame[1]' : a0.tao_qr_frame[1].
148         tolist(), 'a0.tao_qr_frame[2]' : a0.tao_qr_frame[2].tolist(),
149         'a0.T' : a0.T,
150         'a0.phi' : a0.phi, 'a0.theta' : a0.theta, 'a0.psi' : a0.psi,
151         'a0.phidot' : a0.phidot, 'a0.thetadot' : a0.thetadot, 'a0.psidot' : a0.psidot,
152         'a0.phiddot' : a0.phiddot, 'a0.thetaddot' : a0.thetaddot, 'a0.psiddot' : a0.psiddot,
153         'a0.x' : a0.x, 'a0.y' : a0.y, 'a0.z' : a0.z,
154         'a0.xdot' : a0.xdot, 'a0.ydot' : a0.ydot, 'a0.zdot' : a0.zdot,
155         'a0.xddot' : a0.xddot, 'a0.yddot' : a0.yddot, 'a0.zddot' : a0.zddot,
156         'a0.x_integral_error' : a0.x_integral_error,
157         'a0.y_integral_error' : a0.y_integral_error,
158         'a0.z_integral_error' : a0.z_integral_error,
159     }
160     #-----
161     '''
162     if return_val2 != 1:
163
164         test_run_dictionary = {'setpoint':set_point,
165                                'a0.kpx' : a0.kpx,
166                                'a0.kix' : a0.kix,
167                                'a0.kdx' : a0.kdx,
168                                'a0.kpy' : a0.kpy,
169                                'a0.kiy' : a0.kiy,
170                                'a0.kdy' : a0.kdy,
171                                'a0.kpz' : a0.kpz,
172                                'a0.kiz' : a0.kiz,
173                                'a0.kdz' : a0.kdz,
174                                'return_val2' : return_val2,
175                                'a0.ending_iteration' : a0.ending_iteration
176                                }# 'variable_dictionary':variable_dictionary
177                                #}
178
179         return test_run_dictionary
180
181     elif return_val2 ==1:
182
183         #need to calculate the number of times the state crosses the setpoint value:
184
185         x_crossings = 0
186         y_crossings = 0
187         z_crossings = 0

```

```

189     for i in range(len(a0.x)-1):
191         if sign( a0.x[i] - a0.x_des ) != sign( a0.x[i+1] - a0.x_des ) :
193             x_crossings += 1
195
196         if sign( a0.y[i] - a0.y_des ) != sign( a0.y[i+1] - a0.y_des ) :
197             y_crossings += 1
199
200         if sign( a0.z[i] - a0.z_des ) != sign( a0.z[i+1] - a0.z_des ) :
201             z_crossings += 1
203
204
205
206
207
208     #-----
209
210
211     if (max(a0.x) - a0.x_des) > 0: x_over_shoot = max(a0.x) - a0.x_des
212
213     if (max(a0.y) - a0.y_des) > 0: y_over_shoot = max(a0.y) - a0.y_des
214
215     if (max(a0.z) - a0.z_des) > 0: z_over_shoot = max(a0.z) - a0.z_des
216
217
218     #-----
219
220
221     test_run_dictionary = {'setpoint':set_point,
222
223                           'a0.kpx' : a0.kpx,
224                           'a0.kix' : a0.kix,
225                           'a0.kdx' : a0.kdx,
226                           'a0.kpy' : a0.kpy,
227                           'a0.kiy' : a0.kiy,
228                           'a0.kdy' : a0.kdy,
229                           'a0.kpz' : a0.kpz,
230                           'a0.kiz' : a0.kiz,
231                           'a0.kdz' : a0.kdz,
232                           'return_val2' : return_val2,
233                           'a0.ending_iteration' : a0.ending_iteration,
234                           'total_thrust' : sum(a0.T),
235                           'x_over_shoot':x_over_shoot,
236                           'y_over_shoot':y_over_shoot,
237                           'z_over_shoot':z_over_shoot,
238                           'x_crossings':x_crossings,
239                           'y_crossings':y_crossings,
240                           'z_crossings':z_crossings
241                           }

```

```

243         return test_run_dictionary

245 #-----
246 '''
247 {   u'a0.ending_iteration': 266,
248     u'a0.kdx': 5,
249     u'a0.kdy': 5,
250     u'a0.kdz': 20,
251     u'a0.kix': 0.8,
252     u'a0.kiy': 0.8,
253     u'a0.kiz': 15,
254     u'a0.kpx': 5,
255     u'a0.kpy': 5,
256     u'a0.kpz': 30,
257     u'ith_runtime': 7.14291787147522,
258     u'return_val2': 1,
259     u'setpoint': [1, 1, 2],
260     u'total_thrust': 4973.812742102159,
261     u'x_crossings': 1,
262     u'x_over_shoot': 0.06537601013649552,
263     u'y_crossings': 1,
264     u'y_over_shoot': 0.0706587304875288,
265     u'z_crossings': 1,
266     u'z_over_shoot': 0.03570385076740079}
267 '''
268 #-----
269
271 if __name__ == '__main__':
272
273     gain_dictionary = {
274         'kpxy' : 5,
275         'kpz'  : 30,
276         'kdxy' : 5,
277         'kdz'  : 20,
278         'kixy' : 0.8,
279         'kiz'  : 15}
280
281     agent = take_off() # ----> returns the agent instance hovering at (0,0,1)
282
283     set_point = [1,1,2]
284
285     test_run_dictionary = test_gain_vector(agent, set_point, gain_dictionary)
286
287     print 'test_run_dictionary = ',test_run_dictionary
288
289     agent.plot_results()

```

/home/ek/Dropbox/THESIS/python_scripts/brute_force_functions.py

Appendix D

Brute force implementation - runSimsBruteForce.py

```
breakatwhitespace
'''
2 This is a last resort , brute force approach to finding the gain vector that
  produces the lowest objective function value for a set point of (1,1,2)
4
  each run will start with the state variable and input lists produced by the take_off
6 function . for speed this data will be read from a json file which is produced beforehand
8
  for each run the ku gain variable will be incremented by 5 and the objective function measured
  '''
10 import sys
   from numpy import arange
12 from brute_force_functions import *
   from datetime import datetime
14 import json
   import time
16 '''
   gain_dictionary = {
18         'kpxy' : 10,
           'kpz'  : 40,
20         'kdx'  : 10,
           'kdz'  : 40,
22         'kixy' : 0.5,
           'kiz'  : 25}
24 '''
26 global_start_time = time.time()
```

```

28 kpxy_range = arange(5,30,5)
30
32 kpz_range = arange(20,70,10)
34
36 kdx_range = arange(5,30,5)
38
40 kdz_range = arange(20,70,10)
42
44 kixy_range = arange(0.2,1.0,0.2)
46
48 kiz_range = arange(15,45,5)
50
52 print 'kpxy_range = ',kpxy_range
54
56 print 'kpz_range = ',kpz_range
58
60 print 'kdx_range = ',kdx_range
62
64 print 'kdz_range = ',kdz_range
66
68 print 'kixy_range = ',kixy_range
70
72 print 'kiz_range = ',kiz_range
74
76 number_of_sims = len(kpxy_range)*len(kpz_range)*len(kdx_range)*len(kdz_range)*len(kixy_range)*len(kiz_range)
78
80 print 'number_of_sims = ',number_of_sims
82
84 index = int(sys.argv[1])
86
88 runtimes = []
90 run_dictionaries = []
92
94 for kpxy in [kpxy_range[index]]:
96     for kpz in kpz_range:
98         for kdx in kdx_range:
100             for kdz in kdz_range:
102                 for kixy in kixy_range:
104
106                     date_and_time = datetime.now().strftime('%Y-%m-%d_%H.%M.%S')
108
109                     filepath = '/home/ek/Dropbox/THESIS/python_scripts/brute_force_output_data/
110 brute_force_output_index'+str(index)+'_' + date_and_time+ '.json'
112
113                     with open(filepath, 'wb') as fp:
114                         json.dump(run_dictionaries, fp)
115                         fp.close()
116
117                     run_dictionaries = []
118
119                     for kiz in kiz_range:

```

```

82         gain_dictionary = {
83             'kpxy' : kpxy,
84             'kpz'  : kpz,
85             'kdx'  : kdx,
86             'kdz'  : kdz,
87             'kixy' : kixy,
88             'kiz'  : kiz}

90     print '\ngain_dictionary = ',gain_dictionary

92     ith_starttime = time.time()

94     agent = take_off() # ----> returns the agent instance hovering at (0,0,1)

96     set_point = [1,1,2]

98     test_run_dictionary = test_gain_vector(agent, set_point, gain_dictionary)

100    test_run_dictionary['ith_runtime'] = time.time() - ith_starttime

102    print 'test_run_dictionary = ',test_run_dictionary

104    run_dictionaries.append( test_run_dictionary )

106    #-----

108

110    total_run_time = time.time() - global_start_time

112

114    date_and_time = datetime.now().strftime('%Y-%m-%d_%H.%M.%S')

116    filepath = '/home/ek/Dropbox/THESIS/python_scripts/brute_force_output_data/brute_force_output_index'+str(index
117              )+'_' + date_and_time+ '.json'

118    with open(filepath, 'wb') as fp:
119        json.dump(run_dictionaries, fp)
120        fp.close()

122    '''
123    for r in run_dictionaries:
124
125        for kee in r.keys():
126
127            if kee != 'variable_dictionary':
128
129                print kee,r[kee]
130    '''

```

/home/ek/Dropbox/THESIS/python_scripts/run_sims_brute_force.py

Appendix E

Module to sort the results of the brute force method - parseResults.py

```
breakatwhitespace
2
import os
4 import json
import itertools
6 from operator import itemgetter
import pprint
8 pp = pprint.PrettyPrinter(indent=4)
import numpy
10
'''
12 def obj_fun(d):
14     of = d['total_thrust']
16     return of
'''
18
20
22 # need to go through all the output files and make a list of all the sims that still need to be run:
```

```

24 output_dir = '/home/ek/Dropbox/THESIS/python_scripts/brute_force_output_data/'

26 output_file_names = [fn for fn in os.listdir(output_dir)]

28 output_file_paths = [output_dir + ofn for ofn in output_file_names]

30
31 data = []
32
33 for ofp in output_file_paths:
34
35     with open(ofp, 'rb') as fp:
36         output_data = json.load(fp)
37
38         data = data + output_data
39
40 #-----collect the runs that actually converged
41
42 good_runs = []
43
44 for d in data:
45
46     if d['return_val2'] == 1:
47
48         good_runs.append(d)
49
50 number_of_convergent_runs = len(good_runs)
51
52 # -----create a list of total thrust values from the convergent runs
53
54 T_list = [g['total_thrust'] for g in good_runs]
55
56 T_min = numpy.amin(T_list)
57 T_ave = numpy.mean(T_list)
58
59
60 # -----for the runs that actually converged, find the ones that satisfied
    the overshoot criteria
61
62 min_overshoot_runs = []
63
64 for g in good_runs:
65
66     if ( g['x_over_shoot'] < 0.1 ) and ( g['y_over_shoot'] < 0.1 ) and ( g['z_over_shoot'] < 0.1 ):
67
68         min_overshoot_runs.append(g)
69
70
71
72 min_oscillation_runs = []
73
74 for r in min_overshoot_runs:
75
76     if ( r['x_crossings'] < 4 ) and ( r['y_crossings'] < 4 ) and ( r['z_crossings'] < 4 ) :

```

```

78         min_oscillation_runs.append(r)
80
82 thrust_sorted_good_runs = sorted(min_oscillation_runs, key=itemgetter('total_thrust'))
84 for t in thrust_sorted_good_runs[:20]:
85     print '\n'
86     pp.pprint(t)
88
89     #####
90
91     print "\n\nnumber_of_convergent_runs = ", number_of_convergent_runs
92
93     print "minimum thrust = ", T_min
94
95     print "average total thrust = " , T_ave
96
97     print "number_of_runs_with_satisfactory_overshoot = ", len(min_overshoot_runs)
98
99     ave_x_crossings = numpy.mean([g['x_crossings'] for g in good_runs])
100    ave_y_crossings = numpy.mean([g['y_crossings'] for g in good_runs])
101    ave_z_crossings = numpy.mean([g['z_crossings'] for g in good_runs])
102
103    print 'ave_x_crossings = ', ave_x_crossings
104    print 'ave_y_crossings = ', ave_y_crossings
105    print 'ave_z_crossings = ', ave_z_crossings
106
107    ave_x_overshoot = numpy.mean([g['x_over_shoot'] for g in good_runs])
108    ave_y_overshoot = numpy.mean([g['y_over_shoot'] for g in good_runs])
109    ave_z_overshoot = numpy.mean([g['z_over_shoot'] for g in good_runs])
110
111    print 'ave_x_overshoot = ' , ave_x_overshoot
112    print 'ave_y_overshoot = ' , ave_y_overshoot
113    print 'ave_z_overshoot = ' , ave_z_overshoot
114
115    print "number_of_runs_with_satisfactory_oscillations = ", len(min_oscillation_runs)
116
117
118
119
120    # according to the available data, here is the best run...
121    '''
122    {   u'a0.ending_iteration': 266,
123        u'a0.kdx': 5,
124        u'a0.kdy': 5,
125        u'a0.kdz': 20,
126        u'a0.kix': 0.8,
127        u'a0.kiy': 0.8,
128        u'a0.kiz': 15,
129        u'a0.kpx': 5,
130        u'a0.kpy': 5,
131        u'a0.kpz': 30,
132        u'ith_runtime': 7.14291787147522,

```

```
134     u'return_val2': 1,  
135     u'setpoint': [1, 1, 2],  
136     u'total_thrust': 4973.812742102159,  
137     u'x_crossings': 1,  
138     u'x_over_shoot': 0.06537601013649552,  
139     u'y_crossings': 1,  
140     u'y_over_shoot': 0.0706587304875288,  
141     u'z_crossings': 1,  
142     u'z_over_shoot': 0.03570385076740079}  
143  
144  
145  
146 '''
```

/home/ek/Dropbox/THESIS/python_scripts/parse_results.py

Appendix F

Finite difference method applied to the optimal control BVP - finiteDiffSolution.py

```
breakatwhitespace
from os import system
2
from numpy import cos as c , sin as s , array as a , concatenate , arange , sqrt , reshape , log
4
from numpy import dot
6 from numpy.linalg import inv
from numpy.linalg import norm
8 from numpy import transpose
global ixx
10 global iyy
global izz
12
ixx = 5.0*10**-3
14 iyy = 5.0*10**-3
izz = 10.0*10**-3
16
global g
18 global s
global l
20 global b
global m
22 global h
global d
```



```

24 |
25 | g = -9.8
26 | alpha = 0.001
27 | l = 0.25 # m
28 | b = 0.001
29 | m = 1.
30 |
31 | h = 0.1
32 |
33 | d = 0.0001 # the value for adding to the input variables of f to express the finite differences
34 |
35 | #-----the jacobian for transforming from body frame to
36 | inertial frame
37 |
38 | def J(ph,th):
39 |
40 |     jac = a([
41 |         [ixx , 0 , -ixx * s(th)
42 |         ],
43 |         [0 , iyy*(c(ph)**2) + izz * s(ph)**2 , (iyy-izz)*c(ph)*s(ph)*c(th)
44 |         ],
45 |         [-ixx*s(th) , (iyy-izz)*c(ph)*s(ph)*c(th) , ixx*(s(th)**2) + iyy*(s(th)**2)*(c(th)**2) + izz*(c(ph)
46 |         **2)*(c(th)**2)]
47 |     ])
48 |
49 |     #print '\n\njac = \n',jac
50 |
51 |     return jac
52 |
53 |
54 |
55 |
56 | #
57 | -----
58 |
59 |
60 | def coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 ,
61 | partial_with_respect_to ):
62 |
63 |     # the argument 'partial_with_respect_to' specifies the angular quantity for which the derivative is bein
64 |     computed
65 |
66 |     # note that 'partial_with_respect_to' MUST be a string
67 |
68 |     # if the standard coriolis matrix is needed, the argument: 'partial_with_respect_to' should be set to
69 |     None
70 |
71 |     if partial_with_respect_to == 'phd': phd = ( ( ph_k - ph_k_minus_1 ) / h ) + d

```

```

70     else: phd = ( ph_k - ph_k_minus_1 ) / h

72

74     if partial_with_respect_to == 'thd': thd = ( ( th_k - th_k_minus_1 ) / h ) + d

76     else: thd = ( th_k - th_k_minus_1 ) / h

78

80     if partial_with_respect_to == 'psd': psd = ( ( ps_k - ps_k_minus_1 ) / h ) + d

82     else: psd = ( ps_k - ps_k_minus_1 ) / h

84

86     # here are the elements in the matrix

88     c11 = 0

90     c12 = (iyy-izz) * ( thd*c(ph_k)*s(ph_k) + psd*c(th_k)*s(ph_k)**2 ) + (izz-iyy)*psd*(c(ph_k)**2)*c(th_k)
        - ixx*psd*c(th_k)

92     c13 = (izz-iyy) * psd * c(ph_k) * s(ph_k) * c(th_k)**2

94     c21 = (izz-iyy) * ( thd*c(ph_k)*s(ph_k) + psd*s(ph_k)*c(th_k) ) + (iyy-izz) * psd * (c(ph_k)**2) * c(th_k)
        ) + ixx * psd * c(th_k)

96     c22 = (izz-iyy)*phd*c(ph_k)*s(ph_k)

98     c23 = -ixx*psd*s(th_k)*c(th_k) + iyy*psd*(s(ph_k)**2)*s(th_k)*c(th_k)

100    c31 = (iyy-izz)*phd*(c(th_k)**2)*s(ph_k)*c(ph_k) - ixx*thd*c(th_k)

102    c32 = (izz-iyy)*( thd*c(ph_k)*s(ph_k)*s(th_k) + phd*(s(ph_k)**2)*c(th_k) ) + (iyy-izz)*phd*(c(ph_k)**2)*c
        (th_k) + ixx*psd*s(th_k)*c(th_k) - iyy*psd*(s(ph_k)**2)*s(th_k)*c(th_k) - izz*psd*(c(ph_k)**2)*s(th_k)*
        c(th_k)

104    c33 = (iyy-izz) * phd *c(ph_k)*s(ph_k)*(c(th_k)**2) - iyy * thd*(s(ph_k)**2) * c(th_k)*s(th_k) - izz*thd
        *(c(ph_k)**2)*c(th_k)*s(th_k) + ixx*thd*c(th_k)*s(th_k)

106

108    cm = a([[c11,c12,c13],
        [c21,c22,c23],
        [c31,c32,c33]])

110

112    #print '\n\ncm = \n',cm

114

116

118    return cm

```

```

120
122
124
126 #
-----

128 # the function f will be used as the state equations as well as for the many partial derivatives

130 def f(x , x_k_minus_1 , x_k_minus_2 ,
132       y , y_k_minus_1 , y_k_minus_2 ,
134       z , z_k_minus_1 , z_k_minus_2 ,
136       ph_k , ph_k_minus_1 , ph_k_minus_2 ,
138       th_k , th_k_minus_1 , th_k_minus_2 ,
140       ps_k , ps_k_minus_1 , ps_k_minus_2 ,
142       u1_k , u2_k , u3_k , u4_k ):

138     T = alpha * (u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2)
140     #print '\nT = ',T

142     xdd = (1/h**2) * ( x - 2*x_k_minus_1 - x_k_minus_2 )

144     x_residual = T/m * ( c(ps_k) * s(th_k) * c(ph_k) + s(ps_k) * s(ph_k) ) # F1

146     ydd = (1/h**2) * ( x - 2*y_k_minus_1 - y_k_minus_2 )

148     y_residual = T/m * ( s(ps_k) * s(th_k) * c(ph_k) + c(ps_k) * s(ph_k) ) # F2

150     zdd = (1/h**2) * ( x - 2*z_k_minus_1 - z_k_minus_2 )

152     z_residual = T/m * c(th_k) * c(ph_k) + g # F3

154

156     # the angular equations are better kept as vectors and matrices

158

160     input_func_vector = a([
162         1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
164         1 * alpha * ( -u1_k**2 + u3_k**2 ) ,
166         b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
168     ])
170     #print '\ninput_func_vector = ',input_func_vector

172     coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1, None )
174     #print '\ncoriolis_k = ',coriolis_k

176     ang_vel_vector = a( [ (ph_k - ph_k_minus_1)/h , (th_k - th_k_minus_1)/h , ( ps_k - ps_k_minus_1)/h ] )
178     #print '\nang_vel_vector = ',ang_vel_vector

```

```

174
176 # this is the large bracketed factor which is multiplied into the inverse of the jacobian
178 temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )
180 #print '\ntemp_factor = ',temp_factor
182
184 etadd = ( 1/h**2 ) * a([
186     ph_k - 2* ph_k_minus_1 - ph_k_minus_2,
188     th_k - 2* th_k_minus_1 - th_k_minus_2,
190     ps_k - 2* ps_k_minus_1 - ps_k_minus_2,
192     ])
194
196 angular_residual = dot( inv( J(ph_k,th_k) ) , temp_factor ) - etadd
198 #print '\netadd = ',etadd
200
202 state_equations_residual = a( [ x_residual , y_residual , z_residual ,angular_residual[0] ,
204     angular_residual[1] , angular_residual[2] ] )
206
208 #print '\n\nstate_equations_residual = \n',state_equations_residual
210
212
214
216
218
220
222
224

```

```

206 # The costate
208 def costate(x_k , x_k_minus_1 , x_k_minus_2 ,
210     y_k , y_k_minus_1 , y_k_minus_2 ,
212     z_k , z_k_minus_1 , z_k_minus_2 ,
214     ph_k , ph_k_minus_1 , ph_k_minus_2,
216     th_k , th_k_minus_1 , th_k_minus_2,
218     ps_k , ps_k_minus_1 , ps_k_minus_2,
220     u1_k , u2_k , u3_k , u4_k ,
222     la1_k , la1_k_minus_1 , la1_k_minus_2,
224     la2_k , la2_k_minus_1 , la2_k_minus_2,
226     la3_k , la3_k_minus_1 , la3_k_minus_2,
228     la4_k , la4_k_minus_1 , la4_k_minus_2,
230     la5_k , la5_k_minus_1 , la5_k_minus_2,
232     la6_k , la6_k_minus_1 , la6_k_minus_2
234 ):
236
238 # this is the list of state equations differenciated with respect to phi:
240

```

```

226     ff = f(x , x_k_minus_1 , x_k_minus_2 ,
227           y , y_k_minus_1 , y_k_minus_2 ,
228           z , z_k_minus_1 , z_k_minus_2 ,
229           ph_k , ph_k_minus_1 , ph_k_minus_2,
230           th_k , th_k_minus_1 , th_k_minus_2,
231           ps_k , ps_k_minus_1 , ps_k_minus_2,
232           u1_k , u2_k , u3_k , u4_k )

233
234     f_at_phi_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
235                       y , y_k_minus_1 , y_k_minus_2 ,
236                       z , z_k_minus_1 , z_k_minus_2 ,
237                       ph_k + d, ph_k_minus_1 , ph_k_minus_2,
238                       th_k , th_k_minus_1 , th_k_minus_2,
239                       ps_k , ps_k_minus_1 , ps_k_minus_2,
240                       u1_k , u2_k , u3_k , u4_k )

241
242     del_f_d_phi = ( f_at_phi_plus_d - ff )/d

243
244
245
246     f_at_theta_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
247                          y , y_k_minus_1 , y_k_minus_2 ,
248                          z , z_k_minus_1 , z_k_minus_2 ,
249                          ph_k , ph_k_minus_1 , ph_k_minus_2,
250                          th_k + d, th_k_minus_1 , th_k_minus_2,
251                          ps_k , ps_k_minus_1 , ps_k_minus_2,
252                          u1_k , u2_k , u3_k , u4_k )

253
254     del_f_d_theta = ( f_at_theta_plus_d - ff )/d

255
256
257
258
259     f_at_psi_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
260                       y , y_k_minus_1 , y_k_minus_2 ,
261                       z , z_k_minus_1 , z_k_minus_2 ,
262                       ph_k , ph_k_minus_1 , ph_k_minus_2,
263                       th_k , th_k_minus_1 , th_k_minus_2,
264                       ps_k + d, ps_k_minus_1 , ps_k_minus_2,
265                       u1_k , u2_k , u3_k , u4_k )

266
267     del_f_d_psi = ( f_at_psi_plus_d - ff )/d

268
269
270     state_transition_matrix = a( [ del_f_d_phi ,
271                                  del_f_d_theta,
272                                  del_f_d_psi   ] )

273
274     #print '\n\nstate_transition_matrix = ',state_transition_matrix

275
276
277     lam = a( [ la1_k , la2_k , la3_k , la4_k , la5_k , la6_k ] )

278
279     la_k_double_dot = (1/h)*a([

```

```

280         la1_k - 2* la1_k_minus_1 - la1_k_minus_2 ,
281         la2_k - 2* la2_k_minus_1 - la2_k_minus_2 ,
282         la3_k - 2* la3_k_minus_1 - la3_k_minus_2 ,
283         la4_k - 2* la4_k_minus_1 - la4_k_minus_2 ,
284         la5_k - 2* la5_k_minus_1 - la5_k_minus_2 ,
285         la6_k - 2* la6_k_minus_1 - la6_k_minus_2
286     ])
287
288
289     costate_residual_vector = dot( state_transition_matrix , lam ) - la_k_double_dot[3:]
290     #print '\n\ncostate_residual_vector = \n',costate_residual_vector
291     return costate_residual_vector
292
293
294     #
295     -----
296
297     # we must compute the partials of the angular state equations ( etadd ) with respect to the angular
298     # velocities ( phi_dot , theta_dot , psi_dot )
299
300
301     # this function just evaluates the angular state equations replacing "(ph_k - ph_k_minus_1)/h " with "( (
302     # ph_k - ph_k_minus_1)/h ) + d"
303
304     def etadd_with_phi_plus_d( ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 , u1_k , u2_k ,
305                               u3_k , u4_k ):
306
307         input_func_vector = a([
308             1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
309             1 * alpha * ( -u1_k**2 + u3_k**2 ) ,
310             b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
311         ])
312
313         coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 , 'phd' )
314
315         ang_vel_vector = a( [ ( (ph_k - ph_k_minus_1)/h ) + d , (th_k - th_k_minus_1)/h , ( ps_k - ps_k_minus_1)/
316                               h ] )
317
318         temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )
319
320         etadd_phi_plus_d = dot( inv( J(ph_k,th_k) ) , temp_factor )
321         #print '\n\netadd_phi_plus_d = \n',etadd_phi_plus_d
322         return a( etadd_phi_plus_d )
323
324
325     def etadd_with_theta_plus_d( ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 , u1_k , u2_k ,
326                                u3_k , u4_k ):
327
328         input_func_vector = a([
329             1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
330             1 * alpha * ( -u1_k**2 + u3_k**2 ) ,

```

```

328         b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
329     ])
330
331     coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 , 'thd' )
332
333     ang_vel_vector = a( [ (ph_k - ph_k_minus_1)/h , ( (th_k - th_k_minus_1)/h ) + d , ( ps_k - ps_k_minus_1)/
334         h ] )
335
336     temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )
337
338     etadd_theta_plus_d = dot( inv( J(ph_k,th_k) ) , temp_factor )
339     #print '\n\netadd_theta_plus_d = \n',etadd_theta_plus_d
340     return a( etadd_theta_plus_d )
341
342
343 def etadd_with_psi_plus_d( ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 , u1_k , u2_k ,
344     u3_k , u4_k ):
345
346     input_func_vector = a([
347         1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
348         1 * alpha * ( -u1_k**2 + u3_k**2 ) ,
349         b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
350     ])
351
352     coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 , 'psd' )
353
354     ang_vel_vector = a( [ (ph_k - ph_k_minus_1)/h , (th_k - th_k_minus_1)/h , ( ( ps_k - ps_k_minus_1)/h ) + d
355         ] )
356
357     temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )
358
359     etadd_psi_plus_d = dot( inv( J(ph_k,th_k) ) , temp_factor )
360     #print '\n\netadd_psi_plus_d = \n',etadd_psi_plus_d
361     return a( etadd_psi_plus_d )
362
363
364 # this is just the plain ole angular state equation vector
365
366 def etadd( ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 , u1_k , u2_k , u3_k , u4_k ):
367
368     input_func_vector = a([
369         1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
370         1 * alpha * ( -u1_k**2 + u3_k**2 ) ,
371         b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
372     ])
373
374     coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 , None )
375
376     ang_vel_vector = a( [ (ph_k - ph_k_minus_1)/h , (th_k - th_k_minus_1)/h , ( ps_k - ps_k_minus_1)/h ] )
377
378     temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )

```

```

380     etadd_theta_plus_d = dot( inv( J(ph_k,th_k) ) , temp_factor )
381     #print '\n\etadd_theta_plus_d = \n',etadd_theta_plus_d
382     return a( etadd_theta_plus_d )
383
384
385 #
386 -----
387     print '\n\n = \n',
388
389 # The algebraic costate equations
390
391 def algebraic_costate_equations(
392     ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 ,
393     u1_k , u2_k , u3_k , u4_k ,
394     la4_k ,
395     la5_k ,
396     la6_k
397 ):
398
399     # the pnumeonic for the following three assignments is ppd -> phi plus d
400     ppd = etadd_with_phi_plus_d( ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 , u1_k ,
401     u2_k , u3_k , u4_k )
402
403     tpd = etadd_with_theta_plus_d( ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 , u1_k ,
404     u2_k , u3_k , u4_k )
405
406     ppd = etadd_with_psi_plus_d( ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 , u1_k ,
407     u2_k , u3_k , u4_k )
408
409     eta_double_dot = etadd( ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 , u1_k , u2_k ,
410     u3_k , u4_k )
411
412     del_f_d_phi_dot = (ppd-eta_double_dot)/d
413
414     del_f_d_theta_dot = (tpd-eta_double_dot)/d
415
416     del_f_d_psi_dot = (ppd-eta_double_dot)/d
417
418     algebraic_transition_matrix = a( [ del_f_d_phi_dot , del_f_d_theta_dot , del_f_d_psi_dot ] )#.reshape
419     ([3,3])
420
421     #print '\n\nalgebraic_transition_matrix = ',algebraic_transition_matrix
422
423     la4_through_6 = a( [ la4_k , la5_k , la6_k ] )
424     #print '\n\nla4_through_6 = ',la4_through_6
425
426     algebraic_costate_equations_residual_vector = dot( algebraic_transition_matrix , la4_through_6 )
427     #print '\n\nalgebraic_costate_equations_residual_vector = \n',algebraic_costate_equations_residual_vector
428     return algebraic_costate_equations_residual_vector

```



```

428 #-----# stationarity
430 conditions:
432
433 def stationarity_conditions(x_k , x_k_minus_1 , x_k_minus_2 ,
434     y_k , y_k_minus_1 , y_k_minus_2 ,
435     z_k , z_k_minus_1 , z_k_minus_2 ,
436     ph_k , ph_k_minus_1 , ph_k_minus_2 ,
437     th_k , th_k_minus_1 , th_k_minus_2 ,
438     ps_k , ps_k_minus_1 , ps_k_minus_2 ,
439     u1_k , u2_k , u3_k , u4_k ,
440     la1_k , la2_k , la3_k , la4_k , la5_k , la6_k ):
442
443     ff = f(x , x_k_minus_1 , x_k_minus_2 ,
444         y , y_k_minus_1 , y_k_minus_2 ,
445         z , z_k_minus_1 , z_k_minus_2 ,
446         ph_k , ph_k_minus_1 , ph_k_minus_2 ,
447         th_k , th_k_minus_1 , th_k_minus_2 ,
448         ps_k , ps_k_minus_1 , ps_k_minus_2 ,
449         u1_k , u2_k , u3_k , u4_k )
450
451     ff_u1_k_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
452         y , y_k_minus_1 , y_k_minus_2 ,
453         z , z_k_minus_1 , z_k_minus_2 ,
454         ph_k , ph_k_minus_1 , ph_k_minus_2 ,
455         th_k , th_k_minus_1 , th_k_minus_2 ,
456         ps_k , ps_k_minus_1 , ps_k_minus_2 , u1_k + d , u2_k , u3_k , u4_k )
457
458     ff_u2_k_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
459         y , y_k_minus_1 , y_k_minus_2 ,
460         z , z_k_minus_1 , z_k_minus_2 ,
461         ph_k , ph_k_minus_1 , ph_k_minus_2 ,
462         th_k , th_k_minus_1 , th_k_minus_2 ,
463         ps_k , ps_k_minus_1 , ps_k_minus_2 , u1_k , u2_k + d , u3_k , u4_k )
464
465     ff_u3_k_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
466         y , y_k_minus_1 , y_k_minus_2 ,
467         z , z_k_minus_1 , z_k_minus_2 ,
468         ph_k , ph_k_minus_1 , ph_k_minus_2 ,
469         th_k , th_k_minus_1 , th_k_minus_2 ,
470         ps_k , ps_k_minus_1 , ps_k_minus_2 , u1_k , u2_k , u3_k + d , u4_k )
471
472     ff_u4_k_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
473         y , y_k_minus_1 , y_k_minus_2 ,
474         z , z_k_minus_1 , z_k_minus_2 ,
475         ph_k , ph_k_minus_1 , ph_k_minus_2 ,
476         th_k , th_k_minus_1 , th_k_minus_2 ,
477         ps_k , ps_k_minus_1 , ps_k_minus_2 , u1_k , u2_k , u3_k , u4_k + d )
478
479 # note this is the TRANSPOSE of the matrix of partials of f WRT u

```

```

482     dfdq1 = (ff_u1_k_plus_d - ff)/d
484     dfdq2 = (ff_u2_k_plus_d - ff)/d
486     dfdq3 = (ff_u3_k_plus_d - ff)/d
488     dfdq4 = (ff_u4_k_plus_d - ff)/d

490     lambda_k = [ la1_k , la2_k , la3_k , la4_k , la5_k , la6_k ]

492     individ_rows = a([ dot( dfdq1 ,lambda_k ),
494                       dot( dfdq2 ,lambda_k ),
496                       dot( dfdq3 ,lambda_k ),
498                       dot( dfdq4 ,lambda_k )
500                       ])

502
504
506     stationarity_conditions_residual_vector = individ_rows + 2* a( [ u1_k , u2_k , u3_k , u4_k ] )
508     #print '\n\stationarity_conditions_residual_vector = \n',stationarity_conditions_residual_vector
510     return stationarity_conditions_residual_vector

512
514
516
518
520
522 #-----# the objective
524     function at the kth timestep

526
528
530 def G_at_k(x_k, x_k_minus_1 , x_k_minus_2 ,
532     y_k , y_k_minus_1 , y_k_minus_2 ,
534     z_k , z_k_minus_1 , z_k_minus_2 ,
536     ph_k , ph_k_minus_1 , ph_k_minus_2 ,
538     th_k , th_k_minus_1 , th_k_minus_2 ,
540     ps_k , ps_k_minus_1 , ps_k_minus_2 ,
542     u1_k , u2_k , u3_k , u4_k ,
544     la1_k , la1_k_minus_1 , la1_k_minus_2 ,
546     la2_k , la2_k_minus_1 , la2_k_minus_2 ,
548     la3_k , la3_k_minus_1 , la3_k_minus_2 ,
550     la4_k , la4_k_minus_1 , la4_k_minus_2 ,
552     la5_k , la5_k_minus_1 , la5_k_minus_2 ,
554     la6_k , la6_k_minus_1 , la6_k_minus_2
556 ):

558     state_vector_residual = f(x_k , x_k_minus_1 , x_k_minus_2 ,
560     y_k , y_k_minus_1 , y_k_minus_2 ,
562     z_k , z_k_minus_1 , z_k_minus_2 ,
564     ph_k , ph_k_minus_1 , ph_k_minus_2 ,
566     th_k , th_k_minus_1 , th_k_minus_2 ,
568     ps_k , ps_k_minus_1 , ps_k_minus_2 , u1_k , u2_k , u3_k , u4_k )
570     #print '\n\state_vector_residual = \n',state_vector_residual

572
574
576
578
580
582
584
586
588
590
592
594
596
598
600
602
604
606
608
610
612
614
616
618
620
622
624
626
628
630
632
634
636
638
640
642
644
646
648
650
652
654
656
658
660
662
664
666
668
670
672
674
676
678
680
682
684
686
688
690
692
694
696
698
700
702
704
706
708
710
712
714
716
718
720
722
724
726
728
730
732
734
736
738
740
742
744
746
748
750
752
754
756
758
760
762
764
766
768
770
772
774
776
778
780
782
784
786
788
790
792
794
796
798
800
802
804
806
808
810
812
814
816
818
820
822
824
826
828
830
832
834
836
838
840
842
844
846
848
850
852
854
856
858
860
862
864
866
868
870
872
874
876
878
880
882
884
886
888
890
892
894
896
898
900
902
904
906
908
910
912
914
916
918
920
922
924
926
928
930
932
934
936
938
940
942
944
946
948
950
952
954
956
958
960
962
964
966
968
970
972
974
976
978
980
982
984
986
988
990
992
994
996
998
1000

```

```

536         ph_k , ph_k_minus_1 , ph_k_minus_2,
          th_k , th_k_minus_1 , th_k_minus_2,
538         ps_k , ps_k_minus_1 , ps_k_minus_2,
          u1_k , u2_k , u3_k , u4_k,
540         la1_k , la1_k_minus_1 , la1_k_minus_2,
          la2_k , la2_k_minus_1 , la2_k_minus_2,
542         la3_k , la3_k_minus_1 , la3_k_minus_2,
          la4_k , la4_k_minus_1 , la4_k_minus_2,
544         la5_k , la5_k_minus_1 , la5_k_minus_2,
          la6_k , la6_k_minus_1 , la6_k_minus_2
546     )

    #print '\n\ncostate_residual = \n',costate_residual
548
550
552    algebraic_costate_equations_residual_vector = algebraic_costate_equations(
          ph_k , ph_k_minus_1, th_k , th_k_minus_1 , ps_k , ps_k_minus_1,
554          u1_k , u2_k , u3_k , u4_k,
          la4_k ,
556          la5_k ,
          la6_k
558    )

    #print '\n\nalgebraic_costate_equations_residual_vector = \n',algebraic_costate_equations_residual_vector
560
562
564    stationarity_conditions_residual_vector = stationarity_conditions(x_k , x_k_minus_1 , x_k_minus_2 ,
          y_k , y_k_minus_1 , y_k_minus_2 ,
566          z_k , z_k_minus_1 , z_k_minus_2 ,
          ph_k , ph_k_minus_1 , ph_k_minus_2,
568          th_k , th_k_minus_1 , th_k_minus_2,
          ps_k , ps_k_minus_1 , ps_k_minus_2,
570          u1_k , u2_k , u3_k , u4_k,
          la1_k , la2_k , la3_k , la4_k , la5_k , la6_k )
572

    #print '\n\nstationarity_conditions_residual_vector = \n',stationarity_conditions_residual_vector
574
576
578    temp_array = a( concatenate([
          state_vector_residual,
580          costate_residual,
          algebraic_costate_equations_residual_vector,
582          stationarity_conditions_residual_vector])
    )

584

    kth_residual = sum( temp_array )
586    #print '\n\nkth_residual = \n',kth_residual
588
590    return kth_residual

```

```

592
594
596
#-----
598     full_objective_function
600
# the full objective function sums up all the contributions from each time step
602
def full_objective_function(N,
604     x,y,z,
606     ph,th,ps,
608     u1,u2,u3,u4,
610     la1,la2,la3,la4,la5,la6):
612
614     glist = []
616
618     for k in arange(2,N):
620
622         g_k = G_at_k(
624             x[k] , x[k-1], x[k-2],
626             y[k] , y[k-1], y[k-2],
628             z[k] , z[k-1], z[k-2],
630             ph[k] , ph[k-1], ph[k-2],
632             th[k] , th[k-1], th[k-2],
634             ps[k] , ps[k-1], ps[k-2],
636             u1[k] , u2[k] , u3[k] , u4[k],
638             la1[k] , la1[k-1] , la1[k-2],
640             la2[k] , la2[k-1] , la2[k-2],
642             la3[k] , la3[k-1] , la3[k-2],
644             la4[k] , la4[k-1] , la4[k-2],
646             la5[k] , la5[k-1] , la5[k-2],
648             la6[k] , la6[k-1] , la6[k-2]
650         )
652
654         glist.append(g_k)
656
658     objective_function_residual = sum(glist)
660
662     #print '\n\nobjective_function_residual = \n',objective_function_residual
664
666     return objective_function_residual
668
#----- gradient
670
def gradient(N,
672     x,y,z,

```

```

646         ph,th,ps,
        u1,u2,u3,u4,
        la1,la2,la3,la4,la5,la6
648     ):

650     grad = []

652     input_vars_1d = concatenate([
        x[2:-1], y[2:-1], z[2:-1],
654         ph[2:-1], th[2:-1], ps[2:-1],
        u1[2:-1], u2[2:-1], u3[2:-1], u4[2:-1],
656         la1[2:-1], la2[2:-1], la3[2:-1], la4[2:-1], la5[2:-1], la6[2:-1]
        ])

658     obj_func_res = full_objective_function(N,
660         x,y,z,
        ph,th,ps,
662         u1,u2,u3,u4,
        la1,la2,la3,la4,la5,la6)

664     delta = 0.0001

666     for i in range(len(input_vars_1d)):

668         aug_input = []
670         #print 'aug_input = ',aug_input

672         for j in range(len(input_vars_1d)):
674             if j == i:
676                 aug_input.append(a(input_vars_1d[j] + delta) )

678                 else: aug_input.append(input_vars_1d[j])

680         aug_input_parsed = reshape(aug_input, ( 16 , N ))

682         #print 'aug_input_parsed = ',aug_input_parsed

684         x_aug = aug_input_parsed[0]
686         y_aug = aug_input_parsed[1]
688         z_aug = aug_input_parsed[2]
690         ph_aug = aug_input_parsed[3]
692         th_aug = aug_input_parsed[4]
694         ps_aug = aug_input_parsed[5]
696         u1_aug = aug_input_parsed[6]
698         u2_aug = aug_input_parsed[7]
        u3_aug = aug_input_parsed[8]
        u4_aug = aug_input_parsed[9]
        la1_aug = aug_input_parsed[10]
        la2_aug = aug_input_parsed[11]
        la3_aug = aug_input_parsed[12]
        la4_aug = aug_input_parsed[13]
        la5_aug = aug_input_parsed[14]
        la6_aug = aug_input_parsed[15]

```

```

700     aug_obj_func_res = full_objective_function(N,
702         x_aug,y_aug,z_aug,
703         ph_aug,th_aug,ps_aug,
704         u1_aug,u2_aug,u3_aug,u4_aug,
705         la1_aug,la2_aug,la3_aug,la4_aug,la5_aug,la6_aug)
706
707
708     dg = ( aug_obj_func_res - obj_func_res )/delta
709
710     #print '\n\ndg = ',dg
711
712     grad.append(dg)    # grad returns as a one d list...
713
714     return a(grad)
715
716 #####
717
718 if __name__ == '__main__':
719
720     from time import time
721
722     t1 = time()
723
724     N = 10 # the number of timesteps
725
726     # initialize the lists that will contain the solutions for each variable
727
728     init = a( [ 1 for i in range(N) ] ) # note this list does not include the boundary values
729
730     xterm = a([10])
731     yterm = a([10])
732     zterm = a([10])
733
734     x = concatenate([ a([0,0]) , 5 * init , xterm ])
735     y = concatenate([ a([0,0]) , 5 * init , yterm ])
736     z = concatenate([ a([0,0]) , 5 * init , zterm ])
737
738     ph = concatenate([ a([0,0]) , 0.1 * init , a([0])])
739     th = concatenate([ a([0,0]) , 0.1 * init , a([0])])
740     ps = concatenate([ a([0,0]) , 0.1 * init , a([0])])
741
742
743
744     # the terminal conditions for the control inputs are defined by the fact that we want the quadrotor to
745     # end in a hovering state
746
747     # this means that the total thrust must equal g, and that all the inputs (motor speeds) must be the same
748     '''
749     g = T
750
751     g = alpha * (u1**2 + u2**2 + u3**2 + u4**2)
752
753     g = alpha*4*u**2

```

```

754     uterm = sqrt(g)/(4*alpha)
755     '''
756     uhover = sqrt(abs(g))/(4*alpha)
757     #print 'uterm = ',uterm
758
759
760     u1 = concatenate([ a([uhover,uhover]) , 100 * init , a([uhover])])
761     u2 = concatenate([ a([uhover,uhover]) , 100 * init , a([uhover])])
762     u3 = concatenate([ a([uhover,uhover]) , 100 * init , a([uhover])])
763     u4 = concatenate([ a([uhover,uhover]) , 100 * init , a([uhover])])
764
765
766     la1 = concatenate( [ a([0,0]) , init , a([0]) ] )
767
768     la2 = concatenate( [ a([0,0]) , init , a([0]) ] )
769
770     la3 = concatenate( [ a([0,0]) , init , a([0]) ] )
771
772     la4 = concatenate( [ a([0,0]) , init , a([0]) ] )
773
774     la5 = concatenate( [ a([0,0]) , init , a([0]) ] )
775
776     la6 = concatenate( [ a([0,0]) , init , a([0]) ] )
777
778     '''
779     for i in [x,y,z,ph,th,ps,u1,u2,u3,u4,la1,la2,la3,la4,la5,la6]:
780         print len(i)
781     '''
782
783
784     #-----
785
786
787     initial_objective_function_residual = full_objective_function(N,
788
789
790         x,y,z,
791         ph,th,ps,
792         u1,u2,u3,u4,
793         la1,la2,la3,la4,la5,la6)
794
795     #print '\n\nobjective_function_residual= \n',objective_function_residual
796
797
798
799
800
801
802     obj_func_res_list = [initial_objective_function_residual]
803     grad_norm_list = []
804
805
806     max_iterations = 10
807     tol = 1
808
809     for iteration_number in range(max_iterations):
810
811         grad = gradient(N,
812
813             x,y,z,
814             ph,th,ps,

```

```

810         u1,u2,u3,u4,
            la1,la2,la3,la4,la5,la6
812     )

814     grad_norm = norm( a( grad ) )

816     print '-----iteration_number = ',iteration_number
817     print '\n\ngrad = \n',grad
818
819
820     grad_norm_list.append( grad_norm )
821
822     normalized_gradient = grad/grad_norm
823
824     step_size = 0.5
825     '''
826     if iteration_number > 10:
827         step_size = 0.01
828     elif iteration_number > 40:
829         step_size = 0.001
830     elif iteration_number > 60:
831         step_size = 0.0001
832     '''
833
834     # step in the direction opposite of the gradient
835     step = a( ( step_size ) * normalized_gradient )
836
837     print '\n\nstep = ',step
838
839
840     input_vars_1d = concatenate([
841         x[2:-1], y[2:-1], z[2:-1],
842         ph[2:-1], th[2:-1], ps[2:-1],
843         u1[2:-1], u2[2:-1], u3[2:-1], u4[2:-1],
844         la1[2:-1], la2[2:-1], la3[2:-1], la4[2:-1], la5[2:-1], la6[2:-1]
845     ])
846
847
848
849
850     new_partial_input_vector = reshape( input_vars_1d - step , ( 16 , N ) ) # this does not
851     contain the boundary conditions hence the 'partial'
852
853
854
855
856     x = concatenate([ a([0,0]) , new_partial_input_vector[0] , xterm ])
857     y = concatenate([ a([0,0]) , new_partial_input_vector[1] , yterm ])
858     z = concatenate([ a([0,0]) , new_partial_input_vector[2] , zterm ])
859     ph = concatenate([ a([0,0]) , new_partial_input_vector[3] , a([0]) ])
860     th = concatenate([ a([0,0]) , new_partial_input_vector[4] , a([0]) ])
861     ps = concatenate([ a([0,0]) , new_partial_input_vector[5] , a([0]) ])
862     u1 = concatenate([ a([uhover,uhover]) , new_partial_input_vector[6] , a([uhover]) ])

```



```

864     u2 = concatenate([ a([uhover,uhover]) , new_partial_input_vector[7] , a([uhover]) ])
      u3 = concatenate([ a([uhover,uhover]) , new_partial_input_vector[8] , a([uhover]) ])
866     u4 = concatenate([ a([uhover,uhover]) , new_partial_input_vector[9] , a([uhover]) ])
      la1 = concatenate([ a([0,0]) , new_partial_input_vector[10], a([0]) ])
868     la2 = concatenate([ a([0,0]) , new_partial_input_vector[11], a([0]) ])
      la3 = concatenate([ a([0,0]) , new_partial_input_vector[12], a([0]) ])
870     la4 = concatenate([ a([0,0]) , new_partial_input_vector[13], a([0]) ])
      la5 = concatenate([ a([0,0]) , new_partial_input_vector[14], a([0]) ])
872     la6 = concatenate([ a([0,0]) , new_partial_input_vector[15], a([0]) ])

874     print '\n\nx = \n',x
      print '\n\nphi = \n',ph
876     print '\n\nu1 = \n',u1
      print '\n\nla1 = \n',la1

878     objective_function_residual = full_objective_function(N,
880                                     x,y,z,
                                     ph,th,ps,
882                                     u1,u2,u3,u4,
                                     la1,la2,la3,la4,la5,la6)

884     print '\n\nobjective_function_residual = ',objective_function_residual

886     obj_func_res_list.append(objective_function_residual)

888

890     # -----TEST FOR CONVERGENCE

892

894     if objective_function_residual > obj_func_res_list[0]:

896         print '\n\n ERROR : the new value of the objective function has exceeded the initial value'
898         print '\n\n objective_function_residual = ',objective_function_residual
900         print '\n\n obj_func_res_list[0] = ',obj_func_res_list[0]
          break
          #step_size = step_size*0.5

902

904     elif grad_norm < tol:

906         print 'norm_del_G < tolerance.....the process has converged!!!!'
          break

908

910     #wait = raw_input('\n\npress space to continue...')

912

914     t2 = time()

916     delta_t = t2-t1

```

```

918     print '-----'
920     print '\n\n\ndelta_t = ',delta_t
922     #####
924
926
928
930     '''
932     delta = 0.0001
934
936     for i in range(len(input_vars_1d)):
938
939         aug_input = []
940         #print 'aug_input = ',aug_input
942
943         for j in range(len(input_vars_1d)):
944             if j == i:
945                 aug_input.append(a(input_vars_1d[j] + delta) )
947
948             else: aug_input.append(input_vars_1d[j])
950
951         aug_input_parsed = reshape(aug_input, ( 16 , N ))
952
953         #print 'aug_input_parsed = ',aug_input_parsed
954
955         x_aug = aug_input_parsed[0]
956         y_aug = aug_input_parsed[1]
957         z_aug = aug_input_parsed[2]
958         ph_aug = aug_input_parsed[3]
959         th_aug = aug_input_parsed[4]
960         ps_aug = aug_input_parsed[5]
961         u1_aug = aug_input_parsed[6]
962         u2_aug = aug_input_parsed[7]
963         u3_aug = aug_input_parsed[8]
964         u4_aug = aug_input_parsed[9]
965         la1_aug = aug_input_parsed[10]
966         la2_aug = aug_input_parsed[11]
967         la3_aug = aug_input_parsed[12]
968         la4_aug = aug_input_parsed[13]
969         la5_aug = aug_input_parsed[14]
970         la6_aug = aug_input_parsed[15]
972
973         aug_obj_func_res = full_objective_function(N,
974             x_aug,y_aug,z_aug,
975             ph_aug,th_aug,ps_aug,
976             u1_aug,u2_aug,u3_aug,u4_aug,
977             la1_aug,la2_aug,la3_aug,la4_aug,la5_aug,la6_aug)

```

```
974         dg = ( aug_obj_func_res - obj_func_res )/delta
976         #print '\n\ndg = ',dg
978         grad.append(dg)      # grad returns as a one d list...
980
982     , , ,
```

/home/ek/Dropbox/THESIS/quadrotor_optimal_control/finiteDiffSolution.py

Bibliography

- [1] diydrones. diydrones@ONLINE, 2014. URL <http://diydrones.com/>.
- [2] Airware. Airware@ONLINE, 2014. URL <http://www.airware.com/>.
- [3] faa. faa@ONLINE, 2014. URL <http://www.faa.gov/about/initiatives/uas/>.
- [4] Elizabeth Bone and Christopher C Bolkcom. *Unmanned aerial vehicles: background and issues*. Nova Science Pub Incorporated, 2004.
- [5] Jack W Langelaan. Long distance/duration trajectory optimization for small uavs. In *AIAA Guidance, Navigation and Control Conference and Exhibit*, 2007.
- [6] Andrew T Klesh and Pierre T Kabamba. Solar-powered aircraft: Energy-optimal path planning and perpetual endurance. *Journal of guidance, control, and dynamics*, 32(4):1320–1329, 2009.
- [7] Nicholas RJ Lawrance and Salah Sukkarieh. A guidance and control strategy for dynamic soaring with a gliding uav. In *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pages 3632–3637. IEEE, 2009.
- [8] Bora Erginer and Erdinc Altug. Modeling and pd control of a quadrotor vtol vehicle. pages 894–899, 2007.

- [9] Samir Bouabdallah, Andre Noth, and Roland Siegwart. Pid vs lq control techniques applied to an indoor micro quadrotor. 3:2451–2456, 2004. URL <http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.149.6169>.
- [10] Teppo Luukkonen. Modelling and control of quadcopter. *Aalto University*, August 2011.
- [11] Jerry B Marion and Stephen T Thornton. *Classical dynamics of particles and systems*. Saunders College Pub., 1995.
- [12] Cornelius Lanczos. *The variational principles of mechanics*, volume 4. Courier Dover Publications, 1970.
- [13] J. Bernoulli. *Bending of light rays in transparent non-uniform media and the Solution to the problem of determing the Brachistochrone curve*. 1697.
- [14] L.Euler. *Methodus Inveniendi Lineas Curvas Maximi Minimive Proprietate Gaudentes sive Solutio Problematis Isoperimetrici Latissimo Sensu Accepti (The Method of Finding Plane Curves that Show Some Property of Maximum or Minimum*. 1744.
- [15] L.S. Pontryagin V.G. Boltyanskii, R.V. Gamkrelidze. *Towards a theory of optimal processes, (Russian)*. Reports Acad. Sci. USSR, vol.110(1), 1956.
- [16] Frank L Lewis, Draguna Vrabie, and Vassilis L Syrmos. *Optimal control*. Wiley. com, 2012.
- [17] A. E. Bryson and Y. C. Ho. *Applied Optimal Control*. Blaisdell, New York, 1969.
- [18] Arturo Locatelli. *Optimal control: An introduction*. Springer, 2001.

- [19] Michael Athans and Peter L Falb. *Optimal control: an introduction to the theory and its applications*. Courier Dover Publications, 2006.
- [20] L Richard and J Burden. Douglas faires, numerical analysis, 1988.
- [21] Singiresu S Rao and SS Rao. *Engineering optimization: theory and practice*. John Wiley & Sons, 2009.
- [22] Herbert Bishop Keller. *Numerical methods for two-point boundary-value problems*. Dover Publications New York, NY, 1992.
- [23] Aidan O'Dwyer. Reducing energy costs by optimizing controller tuning. In *Conference papers*, page 66, 2006.
- [24] B Wayne Bequette. *Process control: modeling, design, and simulation*. Prentice Hall Professional, 2003.
- [25] Chong G.C.Y. Ang, K.H. and Y. Li. Pid control system analysis, design, and technology. 2005.
- [26] Stuart Bennett. A history of control engineering, 1800-1930. 1986.
- [27] G.G. Coriolis. Mémoire sur les équations du mouvement relatif des systèmes de corps. 1835.
- [28] David Hestenes. *New Foundations for Classical Mechanics*. The Netherlands: Kluwer Academic Publishers. p. 312. ISBN 90-277-2526-8., 1990.
- [29] Singiresu S Rao. *Applied numerical methods for engineers and scientists*. Prentice Hall Professional Technical Reference, 2001.
- [30] JG Ziegler and NB Nichols. Optimum settings for automatic controllers. *trans. ASME*, 64(11), 1942.

- [31] Ziegler-nichols-method @ONLINE, 2014. URL http://en.wikipedia.org/wiki/Ziegler-Nichols_method.