

QUADROTOR 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

April 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 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 techniques 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 is tested, using Ziegler-Nichols' PID tuning technique. Finally, a brute force lookup table based PID controller is used. Simulation results of the heuristic method show 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.

Contents

Abstract	i
Acknowledgements	ii
Contents	iii
List of Figures	v
1 Introduction	1
1.1 Motivation	2
1.2 Prior Work	2
1.3 Organization of the Thesis	5
2 Problem Statement	6
3 Quadrotor Dynamic Model	8
3.1 Description of a quad-rotor	8
3.2 Coordinate System Definitions	9
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	18
4.1 Derivation of the Objective Function	19
4.1.1 Lagrangian	20
4.1.2 Hamiltonian	20
4.1.3 The Objective Function	20
4.2 Derivation of the Optimality Conditions	22
4.3 Solving the Boundary Value Problem	25
4.3.1 The Shooting Method	25

4.3.2	The Finite Difference Method	26
5	Quad-rotor Boundary Value Problem	28
5.1	The Co-State equations	29
5.2	Secondary Algebraic Co-state Equations	31
5.3	Stationarity Conditions	34
5.4	Discretization	35
5.5	A Finite Difference Solution to the Quad-rotor Boundary Value Problem	37
6	PID/PD Control	40
6.1	Deriving the Control Expressions	40
6.2	Testing the Control Scheme	46
7	PID Gain Optimization	54
7.1	Initial Simulation Results	55
7.2	Brute Force Simulation Results	58
8	Summary and Future Work	63
8.1	Summary	63
8.2	Further Work	64
A	agentModule.py	67
B	waypointNavigation.py	85
C	bruteForceFunctions.py	89
D	runSimsBruteForce.py	95
E	parseResults.py	98
F	finiteDiffSolution.py	101
	Bibliography	121

List of Figures

3.1	quad-rotor coordinate system	10
6.1	typical run 3D path	48
6.2	typical run time domain	49
6.3	Cube Edges	50
6.4	Cube Edges	51
6.5	largeSetpointDifferencesTest3D path	52
6.6	largeSetpointDifferencesTesttimedomain	53
7.1	ku vs thrust	57
7.2	optimal run 3D path	61
7.3	optimal run time domain	62

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 are a wide range of companies contributing to the development of UAV technology from open-source projects like DIY Drones [[diy](#)] to start-up firms backed by Google, like Airware [[air](#)]. The Federal Aviation Administration in the USA has plans to produce concrete policy regarding the regulation of commercial applications of UAVs by 2015 [[faa](#)]. 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. It can fly autonomously, or be piloted remotely, can be expendable or recoverable, and can carry a lethal or nonlethal payload [1].

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. The single fundamental feature of an UAV is autonomy or built-in intelligence. 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 works 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 [2], [3], and [4]. 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 more complex mechanical complexity. This means that they have lower speeds and shorter ranges, but much greater maneuverability. 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 [5], [6], and [7].

The background material for understanding the dynamical model of the quad-rotor as given by the Euler-Lagrange formulation are explained in [8] and [9]. These references provided detailed derivations and discussion of the Euler-Lagrange equations of motion as well as related topics like Hamiltonian mechanics and the calculus of variations. Also, [9] 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 [5], [6], [7]. 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. [10]. 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 [11], 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 that allows for solving most optimization problems by Pontryagin [12]. Several books on optimal control ([13], [14], [15], [16]) were referenced for the derivations used in this thesis. In order to test the nonlinear optimization, numerical algorithms for the shooting method ([17], [18], [19]) and the finite difference method ([18], [19]) 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 [20]. 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. Designing and tuning a PID controller is more or less intuitive, and there are various tuning techniques [21]. For instance, there is the Ziegler-Nichols method, manual tuning, and software tuning which can be applied to other control

problems. [\[22\]](#), [\[23\]](#)

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 are 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, and formulate a generalized, classical optimal control scheme. Then we solve the boundary value problem generated by two methods and discuss its pros and cons. In Chapter 5, the classical optimal control scheme developed in the previous chapter is applied to the quad-rotor UAV and 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 [7] and [6].

In the classical optimal control approach, as in [14] and [13], 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 non-linear 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 monstrous two point boundary value problem. Solving the theoretical optimal control problem

would likely produce accurate results, but only after an inordinate lapse of time for computation. 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

Quadrotor 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 brush-less 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. 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 [8] and [9].

The linear and angular coordinates are defined as follows. in Figure 3.1 the basic mechanical structure and the relationships between the spatial coordinates are shown.

ψ is the yaw angle around the z-axis

θ is the pitch angle around the y-axis

ϕ is the roll angle around the x-axis

The spatial variables can be grouped into linear and angular components.

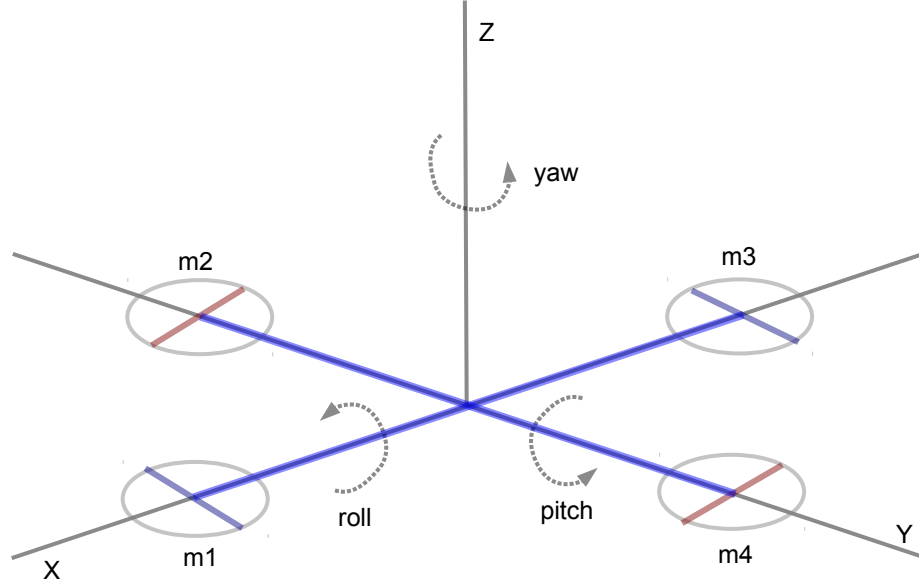


FIGURE 3.1: quad-rotor coordinate system

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

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. The torques due to the rotors about the rotors' axes of rotation are given by $\tau_{M_i} = b\omega_i^2 + I_M\dot{\omega}_i$. 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 torques on the system.

$$\boldsymbol{\tau}_B = \begin{bmatrix} \tau_\phi \\ \tau_\theta \\ \tau_\psi \end{bmatrix} = \begin{bmatrix} ls(-\omega_2^2 + \omega_4^2) \\ ls(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^4 b\omega^2 \end{bmatrix} \quad (3.1)$$

The combined thrust of the rotors in the direction of the quad-rotor frame z axis is

$\boldsymbol{T}_B = [0, 0, T]^T$ where,

$$T = \sum_{i=1}^4 f_i \quad (3.2)$$

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:

$$I = \begin{pmatrix} \frac{1}{12} \left(\frac{m}{2}\right) l^2 & 0 & 0 \\ 0 & \frac{1}{12} \left(\frac{m}{2}\right) l^2 & 0 \\ 0 & 0 & \frac{1}{12} m l^2 \end{pmatrix} = \begin{pmatrix} \frac{\beta}{2} & 0 & 0 \\ 0 & \frac{\beta}{2} & 0 \\ 0 & 0 & \beta \end{pmatrix}. \quad (3.3)$$

In the inertial frame, the kinetic and potential energy of the system are given by

$$T_{\text{trans}} = \frac{1}{2} m \dot{\xi}^T \dot{\xi} \quad (3.4)$$

$$T_{\text{rot}} = \frac{1}{2} \dot{\eta}^T J \dot{\eta} \quad (3.5)$$

$$U = mgz. \quad (3.6)$$

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

$$L = \frac{1}{2} m \dot{\xi}^T \dot{\xi} + \frac{1}{2} \dot{\eta}^T J \dot{\eta} - mgz \quad (3.7)$$

The Jacobian J , is given by

$$J = W_{\eta}^T I W_{\eta} \quad (3.8)$$

$$W_\eta = \begin{bmatrix} 1 & 0 & -\sin(\theta) \\ 0 & \cos(\phi) & \cos(\theta)\sin(\phi) \\ 0 & -\sin(\phi) & \cos(\theta)\cos(\phi) \end{bmatrix} \quad (3.9)$$

The matrix W_η is the matrix transformation which relates the angular velocities from the quad-rotor frame of reference to the inertial frame.

$$J = \begin{pmatrix} \frac{\beta}{2} & 0 & -\frac{\beta}{2}s(\theta) \\ 0 & \frac{\beta}{2}c(\phi)^2 + \beta s(\phi)^2 & \frac{-\beta}{2}c(\phi) s(\phi) c(\theta) \\ -\beta s(\theta) & \frac{-\beta}{2}c(\phi) s(\phi) c(\theta) & \frac{\beta}{2}s(\theta)^2 + \frac{\beta}{2}s(\phi)^2 c(\theta)^2 + \beta c(\phi)^2 c(\theta)^2 \end{pmatrix} \quad (3.10)$$

The dynamics of the system are represented by the Euler - Lagrange differential equations of motion.

$$\frac{d}{dt} \left(\frac{\delta L}{\delta \dot{q}} \right) - \frac{\delta L}{\delta q} = F \quad (3.11)$$

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

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.

$$\begin{pmatrix} f \\ \tau \end{pmatrix} = \frac{d}{dt} \left(\frac{\delta L}{\delta \dot{q}} \right) - \frac{\delta L}{\delta q} \quad (3.12)$$

General derivations of the Euler-Lagrange equations of motion can be found in [8] and [9].

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 below represents an arbitrary rotation transformation from the body frame to the inertial frame is:

$$\mathbf{R} = \begin{bmatrix} c(\psi)c(\theta) & c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi) & c(\psi)s(\theta)c(\phi) + s(\psi)s(\phi) \\ s(\psi)c(\theta) & s(\psi)s(\theta)s(\phi) + c(\psi)c(\phi) & s(\psi)s(\theta)c(\phi) - c(\psi)s(\phi) \\ -s(\theta) & c(\theta)s(\phi) & c(\theta)c(\phi) \end{bmatrix} \quad (3.13)$$

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.

$$\mathbf{f} = \mathbf{R}\mathbf{T}_B = m\ddot{\boldsymbol{\xi}} - \mathbf{G} \quad (3.14)$$

$$m \begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix} = u \begin{pmatrix} \cos \psi \sin \theta \cos \phi + \sin \psi \sin \phi \\ \sin \psi \sin \theta \cos \phi - \cos \psi \sin \phi \\ \cos \theta \cos \phi \end{pmatrix} \quad (3.15)$$

The angular components are expressed as

$$\boldsymbol{\tau} = \boldsymbol{\tau}_b = \frac{d}{dt} \left(\frac{\partial L}{\partial \dot{\boldsymbol{\eta}}} \right) - \frac{\partial L}{\partial \boldsymbol{\eta}} \quad (3.16)$$

$$\boldsymbol{\tau}_b = \frac{d}{dt} (J\dot{\boldsymbol{\eta}}) - \frac{1}{2} \frac{\partial}{\partial \boldsymbol{\eta}} (\dot{\boldsymbol{\eta}}^T J \dot{\boldsymbol{\eta}}) \quad (3.17)$$

$$\tau_b = J\ddot{\eta} + \frac{d}{dt}(J)\dot{\eta} - \frac{1}{2} \frac{\partial}{\partial \eta}(\dot{\eta}^T J \dot{\eta}) \quad (3.18)$$

$$\tau_b = J\ddot{\eta} + \mathfrak{C}(\eta, \dot{\eta})\dot{\eta} \quad (3.19)$$

$$\ddot{\eta} = J^{-1}(\tau_b - \mathfrak{C}(\eta, \dot{\eta})\dot{\eta}) \quad (3.20)$$

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 (3.20)), 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 [24], [25]. For our problem, it is defined as follows.

$$\mathfrak{C}(\eta, \dot{\eta}) = \begin{pmatrix} \mathfrak{C}_{(11)} & \mathfrak{C}_{(12)} & \mathfrak{C}_{(13)} \\ \mathfrak{C}_{(21)} & \mathfrak{C}_{(22)} & \mathfrak{C}_{(23)} \\ \mathfrak{C}_{(31)} & \mathfrak{C}_{(32)} & \mathfrak{C}_{(33)} \end{pmatrix} \quad (3.21)$$

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

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

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

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

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

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

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

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

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

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.

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} + \frac{T}{m} \begin{pmatrix} C_{\psi}S_{\theta}C_{\phi} + S_{\psi}S_{\phi} \\ S_{\psi}S_{\theta}C_{\phi} - C_{\psi}S_{\phi} \\ C_{\theta}C_{\phi} \end{pmatrix} \quad (3.22)$$

$$\begin{pmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{pmatrix} = J^{-1} \left[\begin{pmatrix} ls(-\omega_2^2 + \omega_4^2) \\ ls(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^4 b\omega_i^2 \end{pmatrix} - \mathfrak{C} \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} \right] \quad (3.23)$$

This system model is admittedly rather involved and non-linear. Even in this form there are many important aspects of quadrotor 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 huge one.

Despite the apparent shortcomings of this model, it is very useful as a core component to this thesis. 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 formulation, 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 [14], 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 $L(q(t), u(t), t)$ and the co-state $\lambda(t)$. The objective function defined above 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 F plays the roll of the constraint relationship, and the function $\lambda(t)$ plays the roll of the auxiliary variable. The Lagrangian for our problem is defined as $L[q(t), u(t), t] = u^T I u$ where u is the control input vector and I is the 4×4 identity.

4.1.2 Hamiltonian

For the optimal control formulation, the Hamiltonian can be written as

$$H = L(q(t), u(t), t) + \lambda^T (F(q(t), u(t), t)) \quad (4.1)$$

.

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

4.1.3 The Objective Function

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

$$0 = F - \ddot{q} \quad (4.2)$$

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

$$J = \nu \Psi(q(t_f), t_f) + \int_{t_0}^{t_f} [L(q(t), u(t), t) + \lambda^T (F(q(t), u(t), t) - \ddot{q})] dt. \quad (4.3)$$

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 . The objective function is simplified as:

$$J = \nu \Psi(q(t_f), t_f) + \int_{t_0}^{t_f} H(q(t), u(t), t) - \lambda^T \ddot{q} dt \quad (4.4)$$

The second term in the integrand is integrated by parts.

$$J = \nu \Psi(q(t_f), t_f) + \int_{t_0}^{t_f} H(q(t), u(t), t) dt - \int_{t_0}^{t_f} \lambda^T \ddot{q} dt \quad (4.5)$$

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

$$\int_{t_0}^{t_f} \lambda^T \ddot{q} dt = (\lambda^T \dot{q})|_{t_0}^{t_f} - \int_{t_0}^{t_f} \dot{\lambda}^T \dot{q} dt \quad (4.6)$$

The result is:

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

.

The last term is integrated by parts again.

$$J = \nu \Psi(q(t_f), t_f) - (\lambda^T \dot{q})|_{t_0}^{t_f} + (\dot{\lambda}^T q)|_{t_0}^{t_f} + \int_{t_0}^{t_f} H(q(t), u(t), t) - \ddot{\lambda}^T q dt \quad (4.8)$$

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

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 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 [8] and [9].

The first variation in J is given by

$$\delta J = \frac{\partial J}{\partial q} \delta q + \frac{\partial J}{\partial \dot{q}} \delta \dot{q} + \frac{\partial J}{\partial u} \delta u. \quad (4.10)$$

$$\begin{aligned} \delta J = & \nu^T \frac{\partial \Psi}{\partial q} \delta q|_{t_f} + \nu^T \frac{\partial \Psi}{\partial \dot{q}} \delta \dot{q}|_{t_f} + [\dot{\lambda}^T \delta q - \lambda^T \delta \dot{q}]_{t_0}^{t_f} \\ & + \int_{t_0}^{t_f} \left[\frac{\partial H}{\partial q} \delta q + \frac{\partial H}{\partial \dot{q}} \delta \dot{q} + \frac{\partial H}{\partial u} \delta u - \ddot{\lambda}^T \delta q \right] dt \end{aligned} \quad (4.11)$$

$$\begin{aligned} \delta J = & (\nu^T \frac{\partial \Psi}{\partial q} + \dot{\lambda}^T) \delta q|_{t_f} + (\nu^T \frac{\partial \Psi}{\partial \dot{q}} - \lambda^T) \delta \dot{q}|_{t_f} + [\lambda^T \delta \dot{q} - \dot{\lambda}^T \delta q]_{t_0} \\ & + \int_{t_0}^{t_f} \left[\left(\frac{\partial H}{\partial q} - \ddot{\lambda}^T \right) \delta q + \frac{\partial H}{\partial \dot{q}} \delta \dot{q} + \frac{\partial H}{\partial u} \delta u \right] dt. \end{aligned} \quad (4.12)$$

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

The Co state equations are

$$\frac{\partial H}{\partial q} = \ddot{\lambda} \quad (4.13)$$

$$\ddot{\lambda} = \left(\frac{\partial L}{\partial q} \right)^T + \left(\frac{\partial F}{\partial q} \right)^T \lambda. \quad (4.14)$$

The Stationarity Conditions are

$$\frac{\partial H}{\partial u} = 0 \quad (4.15)$$

$$\frac{\partial L}{\partial u} + \left(\frac{\partial F}{\partial u}\right)^T \lambda = 0 \quad (4.16)$$

Secondary algebraic Co state condition

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

$$\left(\frac{\partial F}{\partial \dot{q}}\right)^T \lambda = 0 \quad (4.18)$$

Terminal Boundary conditions:

$$\nu^T \frac{\partial \Psi}{\partial q} \Big|_{t_f} + \dot{\lambda}(t_f)^T = 0 \quad (4.19)$$

$$\nu^T \frac{\partial \Psi}{\partial \dot{q}} \Big|_{t_f} - \lambda(t_f)^T = 0 \quad (4.20)$$

Initial Co state conditions

$$(\lambda^T \delta \dot{q} - \dot{\lambda}^T \delta q) \Big|_{t_0} = 0 \quad (4.21)$$

$$\lambda(t_0) = 0 \quad (4.22)$$

$$\dot{\lambda}(t_0) = 0 \quad (4.23)$$

Together, the state equations, the co-state equations, the stationarity equations, the secondary algebraic constraints, and the 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 the shooting method and the finite difference method [19],[26]. Both have limitations.

4.3.1 The Shooting Method

The shooting method is a relatively straightforward combination of a time marching quadrature method (Runge-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 The Finite Difference Method

The finite difference method poses another possibility [26]. 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 non linear, 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 system
- Disadvantages
 - hard to solve for nonlinear system
 - 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 [3.22](#) and [3.23](#)) and the optimal control formulation (equations [4.13](#) through [4.23](#)) 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 are 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 The 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 partials 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 partials in $\left(\frac{\partial F}{\partial q}\right)$ is possible but the task of computing them all would push the limits of human endurance and patience. For our simulations, a simple backward finite difference is much easier.

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

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

These algebraic conditions are a result of setting the variation of 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.

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

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

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

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

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

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

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

Like with the other optimality conditions, the partials in this matrix must be computed numerically

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

5.3 Stationarity Conditions

The stationarity conditions express the relationship between the derivatives of the system equation with respect to the input u , the costate 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 they 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.

$$\frac{\partial \phi}{\partial t} \approx \frac{\phi[k] - \phi[k-1]}{h}$$

The second order time derivatives are approximated as a second order backward finite difference.

$$\frac{\partial^2 x}{\partial t^2} \approx \frac{x[k]-2x[k-1]+x[k-2]}{h^2}$$

The continuous system equations are given as follows.

$$\begin{pmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} + \frac{T}{m} \begin{pmatrix} C_\psi S_\theta C_\phi + S_\psi S_\phi \\ S_\psi S_\theta C_\phi - C_\psi S_\phi \\ C_\theta C_\phi \end{pmatrix} \quad (5.19)$$

$$\begin{pmatrix} \ddot{\phi} \\ \ddot{\theta} \\ \ddot{\psi} \end{pmatrix} = J^{-1} \left[\begin{pmatrix} ls(-\omega_2^2 + \omega_4^2) \\ ls(-\omega_1^2 + \omega_3^2) \\ \sum_{i=1}^4 b\omega_i^2 \end{pmatrix} - \mathfrak{C} \begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} \right] \quad (5.20)$$

The discrete-time system equations are

$$\begin{pmatrix} \frac{x[k+1]-2x[k]+x[k-1]}{h^2} \\ \frac{y[k+1]-2y[k]+y[k-1]}{h^2} \\ \frac{z[k+1]-2z[k]+z[k-1]}{h^2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ g \end{pmatrix} + \frac{T[k]}{m} \begin{pmatrix} C_{\psi[k]} S_{\theta[k]} C_{\phi[k]} + S_{\psi[k]} S_{\phi[k]} \\ S_{\psi[k]} S_{\theta[k]} C_{\phi[k]} - C_{\psi[k]} S_{\phi[k]} \\ C_{\theta[k]} C_{\phi[k]} \end{pmatrix} \quad (5.21)$$

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

The reason for computing all the partial derivatives numerically is now apparent. To compute the partials analytically, one would have to deal with the products between the rows of the Coriolis matrix (Equation 3.21) and the columns of the inverse of the Jacobian matrix (Equation 3.10). 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 are possibly functions of the state variables, the co-state variables and the control input. Solving this system becomes a tremendous 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 quadrotor, on the order of thousands of time steps are necessary. To solve this nonlinear system, a straightforward steepest descent technique was used:

- 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 partials 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. Computationally, it is pretty overbearing. The debug cycle was terribly slow even with only ten defined

time steps. The potential of this type of solution to provide realistic results is overshadowed by the exorbitant time requirement. There would be no realistic way to implement this algorithm in this form on an embedded system, which was loosely included as one of our research objectives.

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 quadrotor 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 [7] and [6], the control algorithm proceeds as follows (Algorithm 6.1).

- Algorithm 6.1

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.

$$P_c = \begin{pmatrix} x_c \\ y_c \\ z_c \end{pmatrix} = \text{desired (commanded) set point location}$$

$$P = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \text{actual position at time step } k$$

The discrete-time PID control expressions are formulated using these vectors.

$$\ddot{P}_c = k_p(P_c - P) + k_i \sum_k (P_c - P) + k_d(\dot{P}_c - \dot{P}) \quad (6.1)$$

\ddot{P}_c is the vector of commanded accelerations.

$$\ddot{P}_c = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} \quad (6.2)$$

Equation 3.14 can be rearranged to give

$$\ddot{P}_c = -ge_{inz} + \left(\frac{1}{m}\right)(Te_{qrz})R \quad (6.3)$$

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

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

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

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

$$\begin{aligned}
& \begin{pmatrix} c(\psi)c(\theta) & s(\psi)c(\theta) & -s(\theta) \\ c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi) & s(\psi)s(\theta)s(\phi) + c(\psi)c(\phi) & c(\theta)s(\phi) \\ c(\psi)s(\theta)c(\phi) + s(\psi)s(\phi) & s(\psi)s(\theta)c(\phi) - c(\psi)s(\phi) & c(\theta)c(\phi) \end{pmatrix} \begin{pmatrix} a_x \\ a_y \\ a_z + g \end{pmatrix} \\
&= \frac{1}{m} \begin{pmatrix} 0 \\ 0 \\ T \end{pmatrix}
\end{aligned} \tag{6.5}$$

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

$$a_x c(\psi)c(\theta) + a_y s(\psi)c(\theta) - s(\theta)(a_z + g) = 0 \tag{6.6}$$

$$\begin{aligned}
& a_x (c(\psi)s(\theta)s(\phi) - s(\psi)c(\phi)) \\
& + a_y (s(\psi)s(\theta)s(\phi) + c(\psi)c(\phi)) \\
& + (a_z + g)c(\theta)s(\phi) = 0
\end{aligned} \tag{6.7}$$

$$\begin{aligned}
& a_x (c(\psi)s(\theta)c(\phi) + s(\psi)s(\phi)) \\
& + a_y (s(\psi)s(\theta)c(\phi) - c(\psi)s(\phi)) \\
& + (a_z + g)c(\theta)c(\phi) = \left(\frac{T}{m}\right)
\end{aligned} \tag{6.8}$$

Next, we divide Equation (6.6) by $c(\theta)$ and solve for θ .

$$a_x c(\psi) + a_y s(\psi) + (a_z + g)(-\tan(\theta)) = 0 \quad (6.9)$$

$$\theta_c = \arctan\left(\frac{a_x c(\psi) + a_y s(\psi)}{a_z + g}\right) \quad (6.10)$$

Next: Equation (6.7) x $s(\phi)$ - Equation (6.8) x $c(\phi)$. The result is

$$\phi = \arcsin\left(\frac{a_x s(\psi) - a_y c(\psi)}{T/m}\right). \quad (6.11)$$

Square both sides of (6.4) and note that $R^T = R^{-1}$.

$$a_x^2 + a_y^2 + (a_z + g)^2 = \left(\frac{T}{m}\right)^2 \quad (6.12)$$

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

This result is then substituted back in Equation (6.11) to give

$$\phi_c = \arcsin\left(\frac{a_x s(\psi) - a_y c(\psi)}{\sqrt{a_x^2 + a_y^2 + (a_z + g)^2}}\right) \quad (6.14)$$

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

$$\tau_{\phi_c} = [k_{p\phi}(\phi_c - \phi) + k_{d\phi}(\dot{\phi}_c - \dot{\phi})]I_x \quad (6.15)$$

$$\tau_{\theta c} = [k_{p\theta}(\theta_c - \theta) + k_{d\theta}(\dot{\theta}_c - \dot{\theta})]I_y \quad (6.16)$$

$$\tau_{\psi c} = [k_{p\psi}(\psi_c - \psi) + k_{d\psi}(\dot{\psi}_c - \dot{\psi})]I_z \quad (6.17)$$

Given the commanded torques and the commanded total thrust, the commanded motor speeds can be obtained from the expressions (3.1) and (3.2) from section 3.3

$$\omega_{1c} = \sqrt{\frac{T_c}{4k} - \frac{\tau_{\theta c}}{2kL} - \frac{\tau_{\psi c}}{4b}} \quad (6.18)$$

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

$$\omega_{3c} = \sqrt{\frac{T_c}{4k} + \frac{\tau_{\theta c}}{2kL} - \frac{\tau_{\psi c}}{4b}} \quad (6.20)$$

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

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 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.

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. 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 quadrotor: '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 6.3 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 overshoot 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 over-shoot for each segment was satisfied.

The time domain plots (Figures 6.2, 6.4 , 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 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.

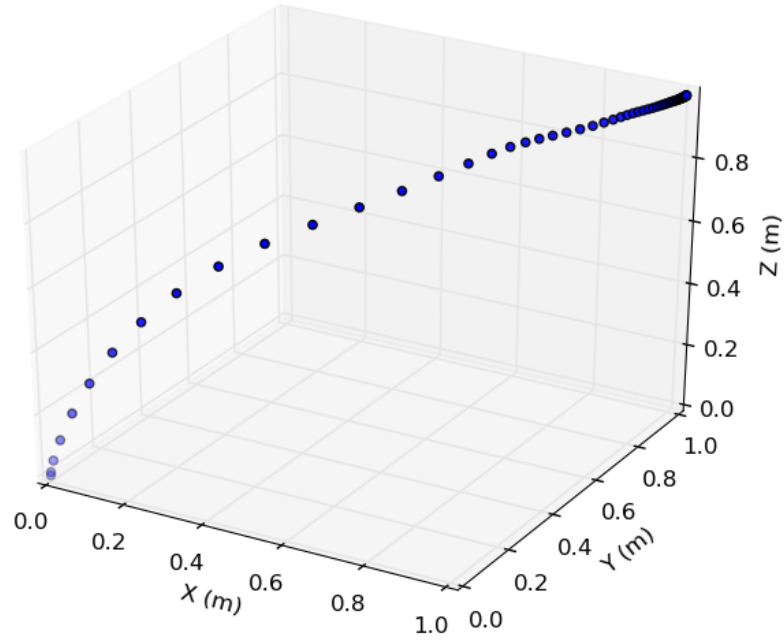


FIGURE 6.1: A typical run - the 3D path

Simulation Parameters

$g = -9.81$	$\frac{m}{s^2}$	acceleration due to gravity
$m = 1$	kg	mass
$L = 1$	m	length of quadrotor 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
$I_{xx} = 5.0 * 10^{-3}$	$\frac{Nms^2}{Rad}$	moments of inertia
$I_{yy} = 5.0 * 10^{-3}$	$\frac{Nms^2}{Rad}$	
$I_{zz} = 10.0 * 10^{-3}$	$\frac{Nms^2}{Rad}$	

TABLE 6.1: Simulation Parameters

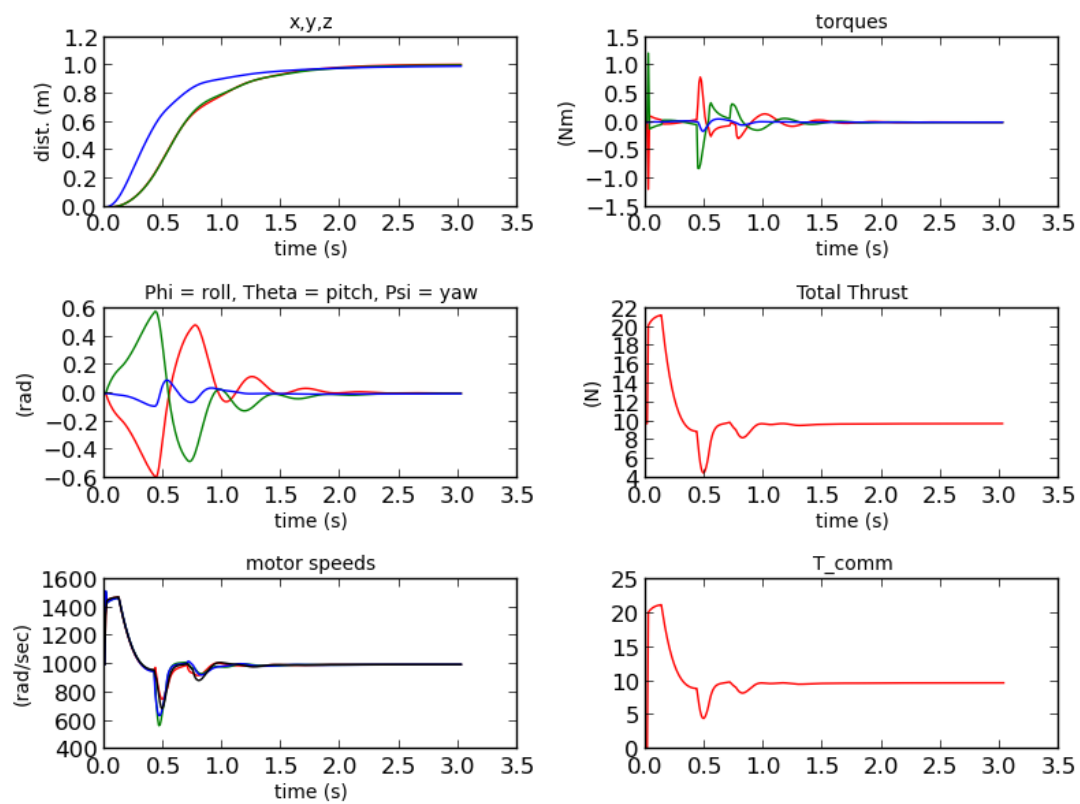


FIGURE 6.2: A typical run - time domain

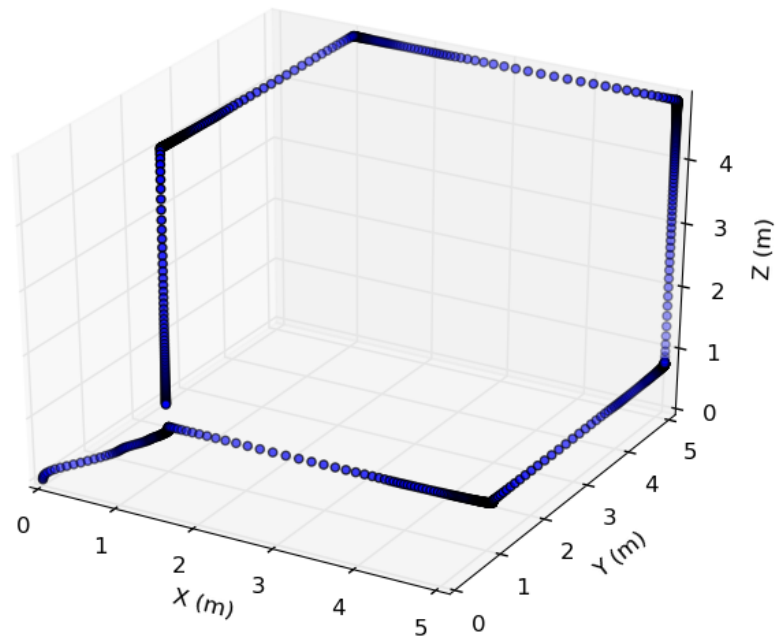


FIGURE 6.3: Tracing some of the edges of a 4m cube - the 3D path

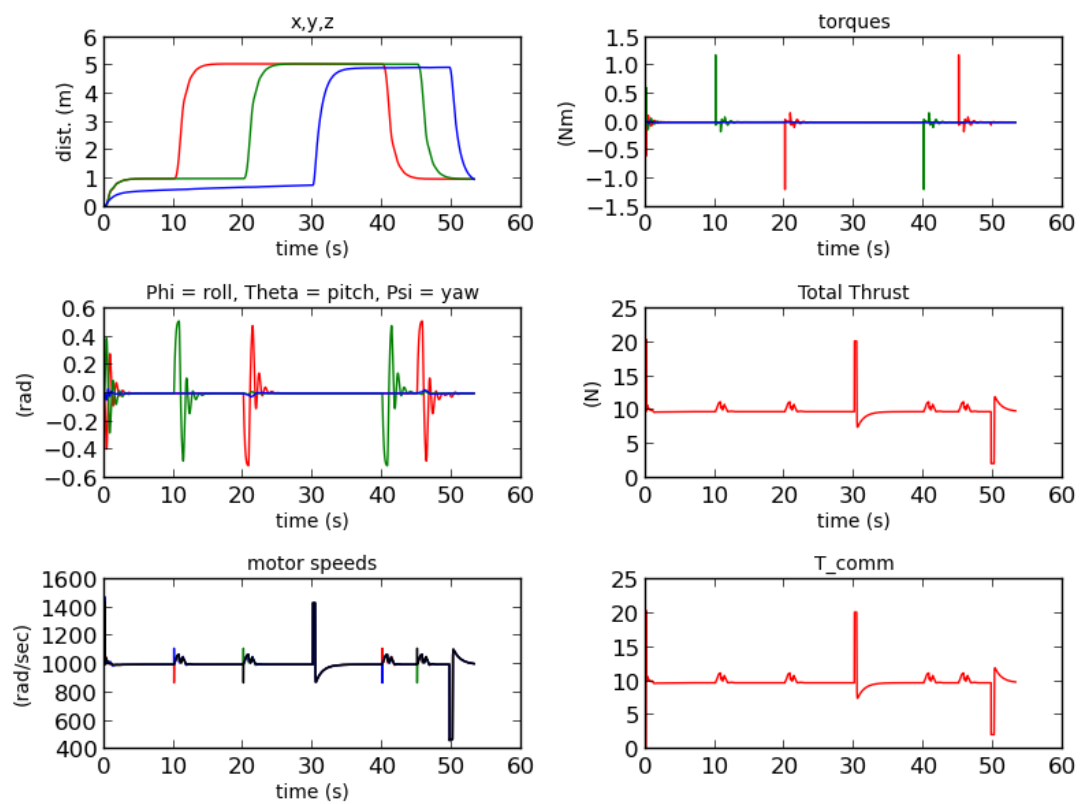


FIGURE 6.4: Tracing some of the edges of a 4m cube - time domain plots

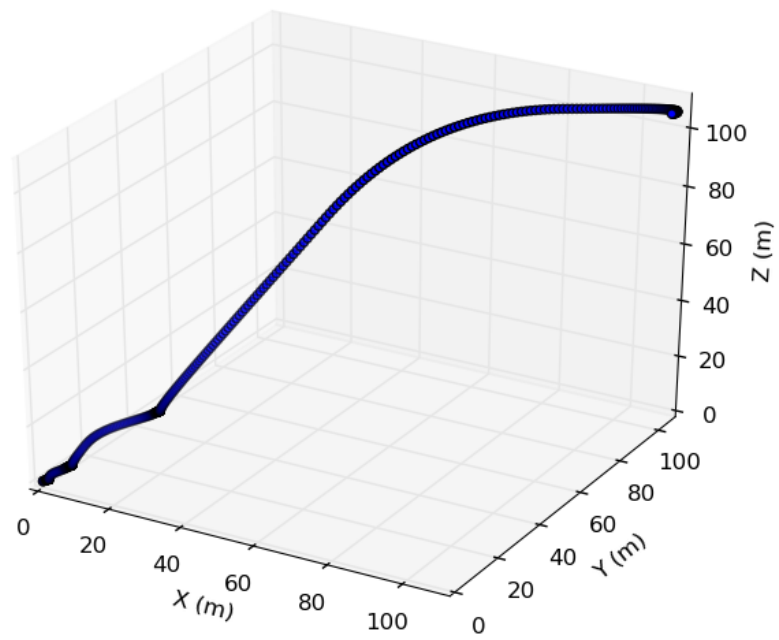


FIGURE 6.5: Testing the control with larger set points - the 3D path

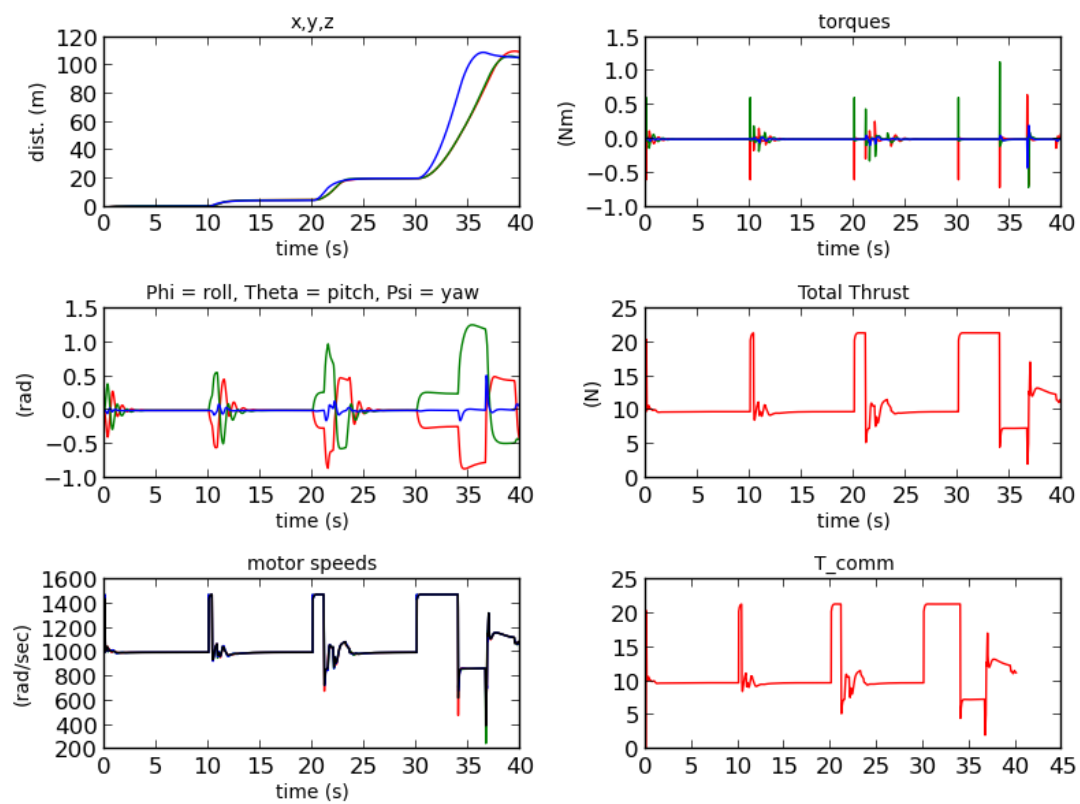


FIGURE 6.6: Testing the control with larger set points - time domain plots

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: $\operatorname{argmin} \left[\sum_{k,i} \omega_i[k] \mid K_p, K_i, K_d \right]$, 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.

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 [27]. There are also excellent resources online to explain tuning [znw].

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.

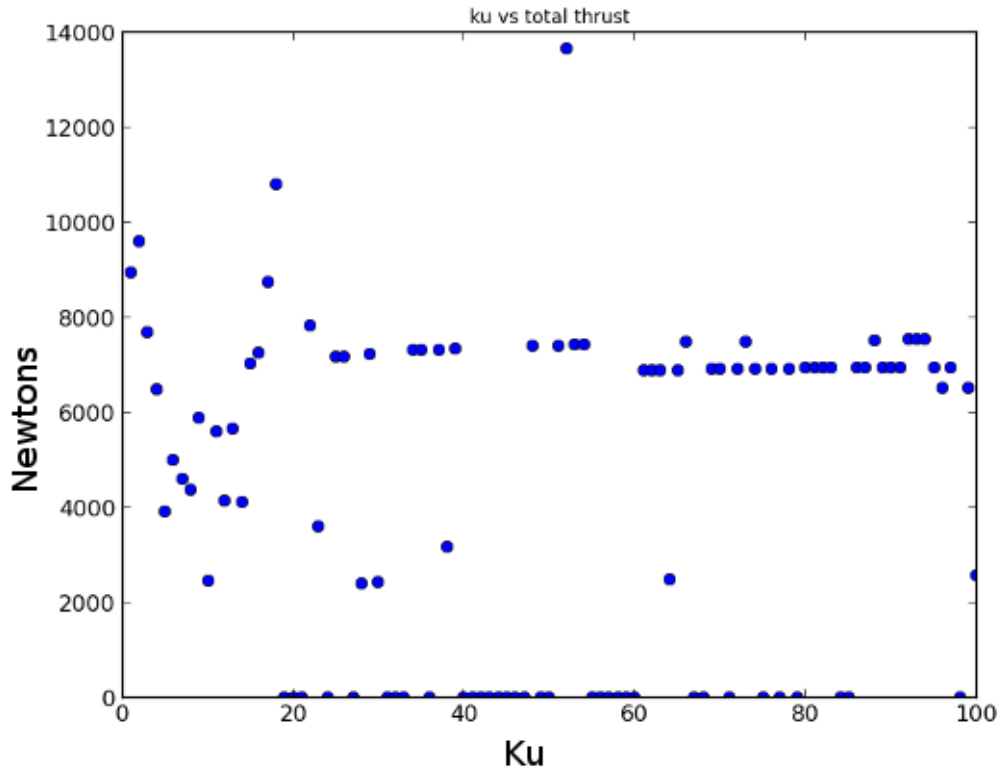


FIGURE 7.1: The relationship between K_u and the measured total thrust.

The relationship between K_u 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. There are a few outliers in the data which are substantially lower in measured total thrust but the reason for their presence in the data as outliers is glib.

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? With another year of research, a nonlinear control law could be implemented and would perhaps make the optimization possible, but this is uncertain. Sadly a naive, brute force approach is actually *MORE* time efficient when compared to another year of research! For the sake of learning, a brute force

algorithm is not so valuable. However, another year of research is simply not an option.

The decision was thus made to simply try many (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 parallelizable), 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.

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 ran to completion without a numerical explosion. 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 Figures 7.2 and 7.3 the dynamics of the system with optimal PID tuning are shown.

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

The Optimal Run

kpx	15
kpy	15
kpz	40
kix	0.8
kiy	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	4969.8 (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)

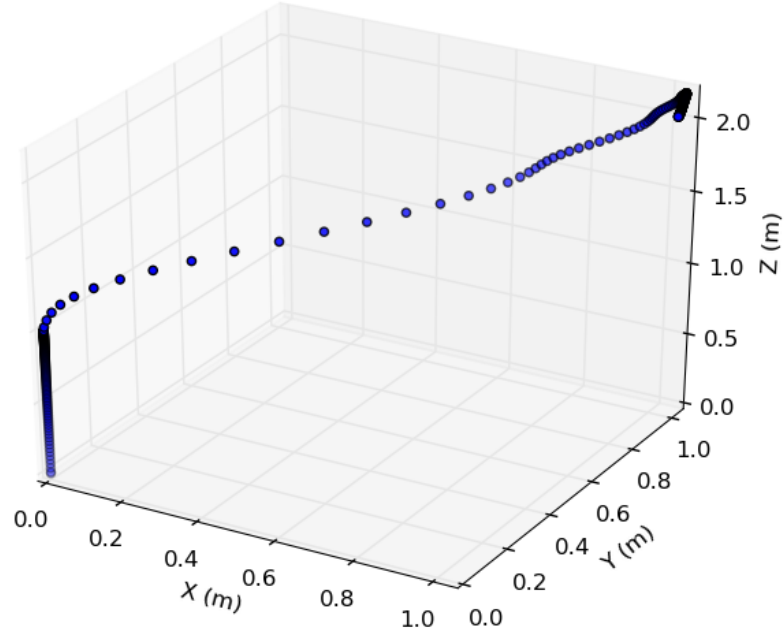


FIGURE 7.2: The Optimal Run - 3D path

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 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.

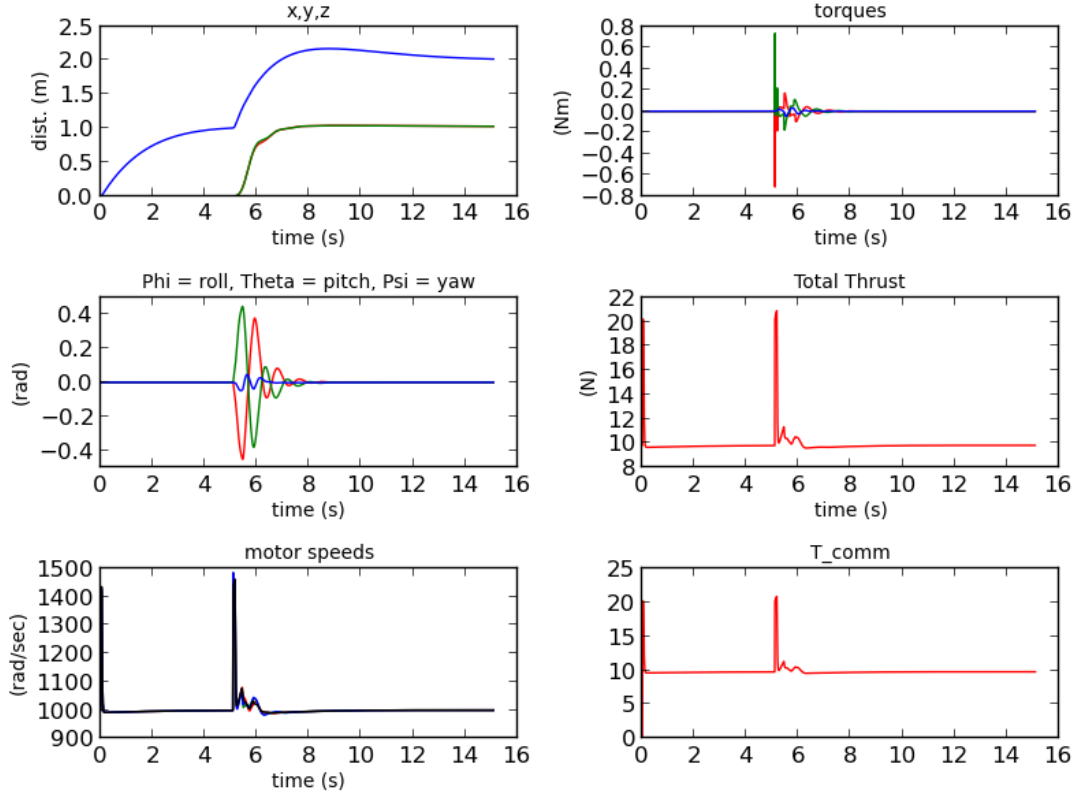


FIGURE 7.3: The Optimal Run - time domain

It is 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.

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. With limited time, the true optimality of the solution is not certain. Even with limited resources we were able to determine a controller tuning which is good enough to perform simulated flights in a relatively efficient manner.

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.

One possibility is to linearize the model in some manner. 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.

On the other hand, 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 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 both and characterize them in the context of the goal of the flight.

Another area of research interest that would benefit from an energy optimization procedure is the control of swarms of UAV's. The distributed control of UAV's is a field which offers a wide range of engineering challenges such as collision avoidance, optimization of networked communication, and optimization of work load delegation toward a common goal. The contextual details of the cooperative aim of the swarm would inform the optimization of the system.

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 sensors are integrated with a Kalman filter to develop situational awareness which allows for effective control of the system.

In general the control and optimization of UAV's 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

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
28
29
30
31     #----- physical constants
32     self.g = -9.81 #[m/s^2]
```

```

33     self.m = 1          #[kg]
34     self.L = 1          #[m]
35     self.b = 10**-6
36     self.k = self.m*abs(self.g)/(4*(1000**2))
37     #print 'self.k = ',self.k
38     #raw_input()
39     #-----moments of inertia
40     self.Ixx = 5.0*10**-3
41     self.Iyy = 5.0*10**-3
42     self.Izz = 10.0*10**-3
43
44     #-----directional drag coefficients
45     self.Ax = 0.25
46     self.Ay = 0.25
47     self.Az = 0.25
48
49     self.hover_at_setpoint = True
50
51     # a list of distances to each other agent sorted nearest to farthest
52     self.distance_vectors = []
53
54     self.agent_priority = agent_priority
55
56     # state vector and derivative time series' list initializations
57     self.x = [x_0]
58     self.y = [y_0]
59     self.z = [z_0]
60
61     self.xdot = [0]
62     self.ydot = [0]
63     self.zdot = [0]
64
65     self.xddot = [0]
66     self.yddot = [0]
67     self.zddot = [0]
68
69     self.phi = [0]
70     self.theta = [0]
71     self.psi = [0]
72
73     self.phidot = [0]
74     self.thetadot = [0]
75     self.psidot = [0]
76
77     self.phiddot = [0]
78     self.thetaddot = [0]
79     self.psidotdot = [0]
80
81     self.tao_qr_frame = []
82     self.etadot = []
83     self.etaddot = []
84
85     #-----CONTROLLER GAINS
86
87

```

```

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

91
    self.kdx = 0.          # PID derivative gain values
93     self.kdy = 0.
    self.kdz = 0.

95
    self.kddx = 0.
97     self.kddy = 0.
    self.kddz = 0.

99
    self.kix = 0.          # PID integral gain values
101    self.kiy = 0.
    self.kiz = 0.

103
    self.kpphi = 4 # gains for the angular pid control laws
105    self.kptheta = 4
    self.kppsi = 4

107
    self.kiphi = 0.
109    self.kitheta = 0.
    self.kipsi = 0.

111
    self.kdphi = 5
113    self.kdtheta = 5
    self.kdpsi = 5

115
    self.xdd_des = 0
117    self.ydd_des = 0
    self.zdd_des = 0

119
    self.x_integral_error = [0]
121    self.y_integral_error = [0]
    self.z_integral_error = [0]

123
    # angular set points
125    self.phi_des = 0
    self.theta_des = 0
127    self.psi_des = 0

129
    self.psidot_des = 0

131
    self.phi_comm = [0,0]
133    self.theta_comm = [0,0]
    self.T_comm = [0,0]

135
    self.tao_phi_comm = [0,0]
137    self.tao_theta_comm = [0,0]
    self.tao_psi_comm = [0,0]

139
    # force, torque, and motor speed list initializations
141

```

```

143         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
192
193         #-----
194
195
196         self.w1_arg = [0,0]

```

```

199     self.w2_arg = [0,0]
200     self.w3_arg = [0,0]
201     self.w4_arg = [0,0]
202
203     self.xacc_comm = [0]
204     self.yacc_comm = [0]
205     self.zacc_comm = [19.62]
206
207
208
209     ##### END " __INIT__ "
210
211
212
213
214
215
216     # the Jacobian for transforming from body frame to inertial frame
217
218     def J(self):
219
220         ix = self.Ixx
221         iy = self.Iyy
222         iz = self.Izz
223
224         th = self.theta[-1]
225         ph = self.phi[-1]
226
227
228         j11 = ix
229
230         j12 = 0
231
232         j13 = -ix * s(th)
233
234         j21 = 0
235
236         j22 = iy*(c(ph)**2) + iz * s(ph)**2
237
238         j23 = (iy-iz)*c(ph)*s(ph)*c(th)
239
240         j31 = -ix*s(th)
241
242         j32 = (iy-iz)*c(ph)*s(ph)*c(th)
243
244         j33 = ix*(s(th)**2) + iy*(s(th)**2)*(c(th)**2) + iz*(c(ph)**2)*(c(th)**2)
245
246         return array([
247             [j11, j12, j13],
248             [j21, j22, j23],
249             [j31, j32, j33]
250         ])

```

```

253
255
257 #-----Coriolis matrix
259
261 def coriolis_matrix(self):
263
265     ph = self.phi[-1]
267     th = self.theta[-1]
269
271     phd = self.phidot[-1]
273     thd = self.thetadot[-1]
275     psd = self.psidot[-1]
277
279     ix = self.Ixx
281     iy = self.Iyy
283     iz = self.Izz
285
287     c11 = 0
289
291     # break up the large elements in to bite size chunks and then add each term ...
293
295     c12_term1 = (iy-iz) * ( thd*c(ph)*s(ph) + psd*c(th)*s(ph)**2 )
297
299     c12_term2and3 = (iz-iy)*psd*(c(ph)**2)*c(th) - ix*psd*c(th)
301
303     c12 = c12_term1 + c12_term2and3
305
307     c13 = (iz-iy) * psd * c(ph) * s(ph) * c(th)**2
309
311     c21_term1 = (iz-iy) * ( thd*c(ph)*s(ph) + psd*s(ph)*c(th) )
313
315     c21_term2and3 = (iy-iz) * psd * (c(ph)**2) * c(th) + ix * psd * c(th)
317
319     c21 = c21_term1 + c21_term2and3
321
323     c22 = (iz-iy)*phd*c(ph)*s(ph)
325
327     c23 = -ix*psd*s(th)*c(th) + iy*psd*(s(ph)**2)*s(th)*c(th)
329
331     c31 = (iy-iz)*phd*(c(th)**2)*s(ph)*c(ph) - ix*thd*c(th)
333
335     c32_term1 = (iz-iy)*( thd*c(ph)*s(ph)*s(th) + phd*(s(ph)**2)*c(th) )
337
339     c32_term2and3 = (iy-iz)*phd*(c(ph)**2)*c(th) + ix*psd*s(th)*c(th)
341
343     c32_term4 = - iy*psd*(s(ph)**2)*s(th)*c(th)

```

```

309         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)
317
318         c33_term2 = - iyy * thd*(s(ph)**2) * c(th)*s(th)
319
320         c33_term3and4 = - izz*thd*(c(ph)**2)*c(th)*s(th) + ixx*thd*c(th)*s(th)
321
322         c33 = c33_term1 + c33_term2 + c33_term3and4
323
324         return array([
325             [c11,c12,c13],
326             [c21,c22,c23],
327             [c31,c32,c33]
328         ])
329
330
331     #-----
332
333     def control_block(self):
334
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
353         self.yacc_comm.append( self.kdy * (self.ydot_des - self.ydot[-1])
354                                + self.kpy * ( self.y_des - self.y[-1] )
355                                + self.kddy * (self.ydd_des - self.yddot[-1] )
356                                + self.kiy * self.y_integral_error[-1] )
357
358
359         self.zacc_comm.append( self.kdz * (self.zdot_des - self.zdot[-1])
360                                + self.kpz * ( self.z_des - self.z[-1] )
361                                + self.kddz * (self.zdd_des - self.zddot[-1] )
362                                + self.kiz * self.z_integral_error[-1] )

```



```

363         # need to limit the max linear acceleration that is perscribed by the control law
365
367         # as a meaningful place to start, just use the value '10m/s/s' , compare to g = -9.8 ...
369
371         max_latt_acc = 5
373
375         max_z_acc = 30
377
379         if abs(self.xacc_comm[-1]) > max_latt_acc: self.xacc_comm[-1] = max_latt_acc * sign(self.xacc_comm
381 [-1])
383         if abs(self.yacc_comm[-1]) > max_latt_acc: self.yacc_comm[-1] = max_latt_acc * sign(self.yacc_comm
385 [-1])
387         if abs(self.zacc_comm[-1]) > max_z_acc: self.zacc_comm[-1] = max_z_acc * sign(self.zacc_comm[-1])
389
391         min_z_acc = 12
393
395         if self.zacc_comm[-1] < min_z_acc: self.zacc_comm[-1] = min_z_acc
397
399         # using the comm linear accelerations, calc theta_c, phi_c and T_c
401
403         theta_numerator = (self.xacc_comm[-1] * c(self.psi[-1]) + self.yacc_comm[-1] * s(self.psi[-1]) )
405
407         theta_denominator = float( self.zacc_comm[-1] + self.g )
409
411         if theta_denominator <= 0:
413
415             theta_denominator = 0.1             # don't divide by zero !!!
417
419         self.theta_comm.append(arctan2( theta_numerator , theta_denominator ))
421
423         self.phi_comm.append(arcsin( (self.xacc_comm[-1] * s(self.psi[-1]) - self.yacc_comm[-1] * c(self.psi
425 [-1]) ) / float(sqrt( self.xacc_comm[-1]**2 +
427
429             self.yacc_comm[-1]**2 +
431
433             (self.zacc_comm[-1] + self.g)**2 ))))
435
437         self.T_comm.append(self.m * ( self.xacc_comm[-1] * ( s(self.theta[-1])*c(self.psi[-1])*c(self.phi
439 [-1]) + s(self.psi[-1])*s(self.phi[-1]) ) +
441             self.yacc_comm[-1] * ( s(self.theta[-1])*s(self.psi[-1])*c(self.phi
443 [-1]) - c(self.psi[-1])*s(self.phi[-1]) ) +
445             (self.zacc_comm[-1] + self.g) * ( c(self.theta[-1])*c(self.phi[-1]) )
447             ))
449
451         if self.T_comm[-1] < 1.0:
453             self.T_comm = self.T_comm[:-1]
455             self.T_comm.append(1.0)
457
459         # we will need the derivatives of the comanded angles for the torque control laws.
461         self.phidot_comm = (self.phi_comm[-1] - self.phi_comm[-2])/self.h
463
465         self.thetadot_comm = (self.theta_comm[-1] - self.theta_comm[-2])/self.h

```

```

411         # solve for torques based on theta_c, phi_c and T_c , also psi_des , and previous values of theta,
412         phi, and psi
413
414
415         tao_phi_comm_temp = ( self.kpphi*(self.phi_comm[-1] - self.phi[-1]) + self.kdphi*(self.phidot_comm -
416         self.phidot[-1]) )*self.Ixx
417
418         tao_theta_comm_temp = ( self.kptheta*(self.theta_comm[-1] - self.theta[-1]) + self.kdtheta*(self.
419         thetadot_comm - self.thetadot[-1]) )*self.Iyy
420
421         tao_psi_comm_temp = ( self.kppsi*(self.psi_des - self.psi[-1]) + self.kdpsi*( self.psidot_des - self.
422         psidot[-1] ) )*self.Izz
423
424         self.tao_phi_comm.append(tao_phi_comm_temp )
425         self.tao_theta_comm.append(tao_theta_comm_temp )
426         self.tao_psi_comm.append(tao_psi_comm_temp )
427
428         #-----solve for motor speeds, eq 24
429
430         self.w1_arg.append( (self.T_comm[-1] / (4.0*self.k)) - ( self.tao_theta_comm[-1] / (2.0*self.k*self.L
431         ) ) - ( self.tao_psi_comm[-1] / (4.0*self.b) ) )
432         self.w2_arg.append( (self.T_comm[-1] / (4.0*self.k)) - ( self.tao_phi_comm[-1] / (2.0*self.k*self.L
433         ) ) + ( self.tao_psi_comm[-1] / (4.0*self.b) ) )
434         self.w3_arg.append( (self.T_comm[-1] / (4.0*self.k)) + ( self.tao_theta_comm[-1] / (2.0*self.k*self.L
435         ) ) - ( self.tao_psi_comm[-1] / (4.0*self.b) ) )
436         self.w4_arg.append( (self.T_comm[-1] / (4.0*self.k)) + ( self.tao_phi_comm[-1] / (2.0*self.k*self.L
437         ) ) + ( self.tao_psi_comm[-1] / (4.0*self.b) ) )
438
439         self.w1.append( sqrt( self.w1_arg[-1] ) )
440         self.w2.append( sqrt( self.w2_arg[-1] ) )
441         self.w3.append( sqrt( self.w3_arg[-1] ) )
442         self.w4.append( sqrt( self.w4_arg[-1] ) )
443
444         # IMPORTANT!!! THIS ENDS THE 'CONTROLLER BLOCK' IN A REAL IMPLEMENTATION, WE WOULD NOW TAKE
445         MEASUREMENTS AND ESTIMATE THE STATE and then start over...
446
447         def system_model_block(self):
448
449             # BELOW ARE THE EQUATIONS THAT MODEL THE SYSTEM,
450             # FOR THE PURPOSE OF SIMULATION, GIVEN THE MOTOR SPEEDS WE CAN CALCULATE THE STATES OF THE SYSTEM
451
452             self.tao_qr_frame.append( array([
453                 self.L*self.k*( -self.w2[-1]**2 + self.w4[-1]**2 ) ,
454                 self.L*self.k*( -self.w1[-1]**2 + self.w3[-1]**2 ) ,
455                 self.b*( -self.w1[-1]**2 + self.w2[-1]**2 - self.w3[-1]**2 + self.w4[-1]**2
456             )
457
458             ]) )
459
460             self.tao_phi.append(self.tao_qr_frame[-1][0])
461             self.tao_theta.append(self.tao_qr_frame[-1][1])
462             self.tao_psi.append(self.tao_qr_frame[-1][2])
463
464             self.T.append(self.k*( self.w1[-1]**2 + self.w2[-1]**2 + self.w3[-1]**2 + self.w4[-1]**2 ) )

```

```

455         # use the previous known angles and the known thrust to calculate the new resulting linear
accelerations
457         # remember this would be measured ...
459         # for the purpose of modeling the measurement error and testing a kalman filter, inject noise here...
459         # perhaps every 1000ms substitute an artificial gps measurement (and associated uncertainty) for the
double integrated imu value

461         self.xddot.append( (self.T[-1]/self.m)*( c(self.psi[-1])*s(self.theta[-1])*c(self.phi[-1])
+ s(self.psi[-1])*s(self.phi[-1]) )
463         - self.Ax * self.xdot[-1] / self.m )

465         self.yddot.append( (self.T[-1]/self.m)*( s(self.psi[-1])*s(self.theta[-1])*c(self.phi[-1])
- c(self.psi[-1])*s(self.phi[-1]) )
467         - self.Ay * self.ydot[-1] / self.m )

469         self.zddot.append( self.g + (self.T[-1]/self.m)*( c(self.theta[-1])*c(self.phi[-1]) ) - self.Az *
self.zdot[-1] / self.m )

471         # calculate the new angular accelerations based on the known values
self.etadot.append( array( [self.phidot[-1], self.thetadot[-1], self.psidot[-1] ] ) )

473

475         self.etaddot.append( dot(inv( self.J() ), self.tao_qr_frame[-1] - dot(self.coriolis_matrix() , self
.etadot[-1]) ) )

477         # parse the etaddot vector of the new accelerations into the appropriate time series'
self.phiddot.append(self.etaddot[-1][0])

479         self.thetaddot.append(self.etaddot[-1][1])

481         self.psidot.append(self.etaddot[-1][2])

483         #----- integrate new acceleration values to obtain velocity values

485         self.xdot.append( self.xdot[-1] + self.xddot[-1] * self.h )
self.ydot.append( self.ydot[-1] + self.yddot[-1] * self.h )
487         self.zdot.append( self.zdot[-1] + self.zddot[-1] * self.h )

489         self.phidot.append( self.phidot[-1] + self.phiddot[-1] * self.h )
self.thetadot.append( self.thetadot[-1] + self.thetaddot[-1] * self.h )
491         self.psidot.append( self.psidot[-1] + self.psidot[-1] * self.h )

493         #----- integrate new velocity values to obtain position / angle values

495         self.x.append( self.x[-1] + self.xdot[-1] * self.h )
self.y.append( self.y[-1] + self.ydot[-1] * self.h )
497         self.z.append( self.z[-1] + self.zdot[-1] * self.h )

499         self.phi.append( self.phi[-1] + self.phidot[-1] * self.h )
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
513         timeSeries = [self.h*i for i in range(len(self.x))]
515
517         from mpl_toolkits.mplot3d import Axes3D
519         import matplotlib.pyplot as plt
521         from pylab import title
523         import matplotlib.gridspec as gridspec
525
527         fig0 = plt.figure()
529
531         ax = fig0.add_subplot(111, projection='3d')
533
535         ax.scatter(
537             self.x[0:len(self.x):5],
539             self.y[0:len(self.y):5],
541             self.z[0:len(self.z):5])
543
545         ax.set_xlabel('X (m)')
547         ax.set_ylabel('Y (m)')
549         ax.set_zlabel('Z (m)')
551
553         fig1 = plt.figure()
555         gs1 = gridspec.GridSpec(3, 2)
557
559         #-----linear displacements
561         xx = fig1.add_subplot(gs1[0,0])
563         plt.plot(timeSeries, self.x, 'r',
565                 timeSeries, self.y, 'g',
567                 timeSeries, self.z, 'b')
569         title('x,y,z', fontsize=10)
571         plt.xlabel('time (s)', fontsize=10)
573         plt.ylabel('dist. (m)', fontsize=10)
575
577         #-----angles
579         thth = fig1.add_subplot(gs1[1,0])
581         plt.plot(timeSeries, self.phi, 'r',
583                 timeSeries, self.theta, 'g',
585                 timeSeries, self.psi, 'b')
587         title('Phi = roll, Theta = pitch, Psi = yaw', fontsize=10)
589         plt.xlabel('time (s)', fontsize=10)
591         plt.ylabel('(rad)', fontsize=10)
593
595         #-----motor speeds
597
599         spd = fig1.add_subplot(gs1[2,0])
601         plt.plot([self.h*i for i in range(len(self.w1))], self.w1, 'r',
603                 [self.h*i for i in range(len(self.w2))], self.w2, 'g',
605                 [self.h*i for i in range(len(self.w3))], self.w3, 'b',
607                 [self.h*i for i in range(len(self.w4))], self.w4, 'k')

```

```

561     title('motor speeds',fontsize=10)
562     plt.xlabel('time (s)',fontsize=10)
563     plt.ylabel('(rad/sec)',fontsize=10)
564     #-----torque
565     spd = fig1.add_subplot(gs1[0,1])
566     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',
568             [self.h*i for i in range(len(self.tao_psi))], self.tao_psi,'b')
569     title('torques ',fontsize=10)
570     plt.xlabel('time (s)',fontsize=10)
571     plt.ylabel('(Nm)',fontsize=10)
572     #-----thrust
573     spd = fig1.add_subplot(gs1[1,1])
574     plt.plot([self.h*i for i in range(len(self.T))], self.T,'r',)
575     title('Total Thrust',fontsize=10)
576     plt.xlabel('time (s)',fontsize=10)
577     plt.ylabel('(N)',fontsize=10)
578
579     #-----commanded total thrust
580     t_comm = fig1.add_subplot(gs1[2,1])
581     plt.plot([self.h*i for i in range(len(self.T_comm))], self.T_comm,'r')
582     title('T_comm',fontsize=10)
583
584
585
586     '''
587     #-----wind velocities
588     wind_plot = fig1.add_subplot( gs1[ 8:10 ,0:3 ] )
589
590     plt.plot([self.h*i for i in range(len(self.wind_x))], self.wind_x,'-r',
591             [self.h*i for i in range(len(self.wind_y))], self.wind_y,'-g',
592             [self.h*i for i in range(len(self.wind_z))], self.wind_z,'-b' )
593
594     title('wind velocities, rgb = xyz', fontsize=10)
595     '''
596     fig1.tight_layout()
597
598     #####
599
600
601
602
603     fig2 = plt.figure()
604     gs2 = gridspec.GridSpec(4, 2)
605
606     #-----linear velocities
607     lin_vel = fig2.add_subplot(gs2[0,0])
608     plt.plot(timeSeries, self.xdot,'r',
609             timeSeries, self.ydot,'g',
610             timeSeries, self.zdot,'b')
611     title('xdot, ydot, self.zdot',fontsize=10)
612
613     #-----linear accelerations
614     lin_acc = fig2.add_subplot(gs2[1,0])
615     plt.plot(timeSeries, self.xddot,'r',

```

```

        timeSeries, self.yddot,'g',
        timeSeries, self.zddot,'b')
    title('xddot, yddot, self.zddot',fontsize=10)

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

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

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

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

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

#-----commanded phi and theta
integral_errors = fig2.add_subplot(gs2[3,1])
plt.plot([self.h*i for i in range(len(self.theta_comm))], self.theta_comm,'r',
         [self.h*i for i in range(len(self.phi_comm))], self.phi_comm,'g'
         )
title('theta_com ,phi_com ',fontsize=10)

```

```

671
673     fig2.tight_layout()
675
677     if save_plot == True:
679
681         fig1.savefig(fig1_file_path)
683         fig2.savefig(fig2_file_path)
685
687
689     if show_plot == True:
691
693         plt.show()
695
697
699
701
703
705
707
709
711
713
715
717
719
721
723
725

```

```

def print_dump(self,n=5):

    print '\n\nself.xacc_comm[-n:] = ',around(self.xacc_comm[-n:], decimals=5)
    print '\n\nself.yacc_comm[-n:] = ',around(self.yacc_comm[-n:], decimals=5)
    print '\n\nself.zacc_comm[-n:] = ',around(self.zacc_comm[-n:], decimals=5)

    print '\n\nself.theta_comm[-n:] = ',around(self.theta_comm[-n:], decimals=5)

    print '\n\nself.phi_comm[-n:] = ',around(self.phi_comm[-n:], decimals=5)

    print '\n\nself.T_comm[-n:] = ',around(self.T_comm[-n:], decimals=5)

    print '\n\nself.tao_phi_comm = ',around(self.tao_phi_comm[-n:], decimals=5)
    print '\n\nself.tao_theta_comm = ',around(self.tao_theta_comm[-n:], decimals=5)
    print '\n\nself.tao_psi_comm = ',around(self.tao_psi_comm[-n:], decimals=5)

    print '\n\nnw1_arg[-n:] = ',around(self.w1_arg[-n:], decimals=0)
    print '\n\nnw2_arg[-n:] = ',around(self.w2_arg[-n:], decimals=0)
    print '\n\nnw3_arg[-n:] = ',around(self.w3_arg[-n:], decimals=0)
    print '\n\nnw4_arg[-n:] = ',around(self.w4_arg[-n:], decimals=0)

    print '\n\nnw1[-n:] = ',around(self.w1[-n:], decimals=1)
    print '\n\nnw2[-n:] = ',around(self.w2[-n:], decimals=1)
    print '\n\nnw3[-n:] = ',around(self.w3[-n:], decimals=1)
    print '\n\nnw4[-n:] = ',around(self.w4[-n:], decimals=1)

    print '\n\nself.tao_qr_frame[-n:] = ',around(self.tao_qr_frame[-n:], decimals=5)

    print '\n\nself.T[-n:] = ',around(self.T[-n:], decimals=5)

    print '\n\nself.phi[-n:] = ',around(self.phi[-n:], decimals=5)
    print '\n\nself.theta[-n:] = ',around(self.theta[-n:], decimals=5)
    print '\n\nself.psi[-n:] = ',around(self.psi[-n:], decimals=5)

    print '\n\nself.phidot[-n:] = ',around(self.phidot[-n:], decimals=5)
    print '\n\nself.thetadot[-n:] = ',around(self.thetadot[-n:], decimals=5)
    print '\n\nself.psidot[-n:] = ',around(self.psidot[-n:], decimals=5)

```

```

727     print '\n\nself.phiddot[-n:] = ',around(self.phiddot[-n:], decimals=5)
728     print '\n\nself.thetaddot[-n:] = ',around(self.thetaddot[-n:], decimals=5)
729     print '\n\nself.psiddot[-n:] = ',around(self.psiddot[-n:], decimals=5)
730
731     print '\n\nself.x[-n:] = ',around(self.x[-n:], decimals=5)
732     print '\n\nself.y[-n:] = ',around(self.y[-n:], decimals=5)
733     print '\n\nself.z[-n:] = ',around(self.z[-n:], decimals=5)
734
735     print '\n\nself.xdot[-n:] = ',around(self.xdot[-n:], decimals=5)
736     print '\n\nself.ydot[-n:] = ',around(self.ydot[-n:], decimals=5)
737     print '\n\nself.zdot[-n:] = ',around(self.zdot[-n:], decimals=5)
738
739     print '\n\nself.xddot[-n:] = ',around(self.xddot[-n:], decimals=5)
740     print '\n\nself.yddot[-n:] = ',around(self.yddot[-n:], decimals=5)
741     print '\n\nself.zddot[-n:] = ',around(self.zddot[-n:], decimals=5)
742
743     print '\n\nself.x_integral_error[-n:] = ',around(self.x_integral_error[-n:], decimals=5)
744     print '\n\nself.y_integral_error[-n:] = ',around(self.y_integral_error[-n:], decimals=5)
745     print '\n\nself.z_integral_error[-n:] = ',around(self.z_integral_error[-n:], decimals=5)
746
747     print '\n\n'
748     #-----
749 if __name__ == "__main__":
750
751     a = agent(x_0 = 0,
752             y_0 = 0,
753             z_0 = 0,
754             initial_setpoint_x = 1,
755             initial_setpoint_y = 1,
756             initial_setpoint_z = 1,
757             agent_priority = 1)
758
759
760
761     #the following gains worked well for a setpoint of (1,1,1)
762
763     a.kpx = 40      # -----PID proportional gain values
764     a.kpy = 40
765     a.kpz = 40
766
767     a.kdx = 20      #-----PID derivative gain values
768     a.kdy = 20
769     a.kdz = 10
770
771     a.kix = .2
772     a.kiy = .2
773     a.kiz = 40
774
775     a.kpphi = 4     # gains for the angular pid control laws
776     a.kptheta = 4
777     a.kppsi = 4
778
779     a.kdphi = 10
780     a.kdtheta = 10

```



```

781     a.kdpsi    = 5
783
785
787
789     for i in range(a.max_iterations):
791
792         a.ending_iteration = i    # preemptively...
793
794         a.system_model_block()
795
796         a.control_block()
797
798         x_ave = sum(a.x[-100:])/100.0
799         y_ave = sum(a.y[-100:])/100.0
800         z_ave = sum(a.z[-100:])/100.0
801
802         xerr = a.x_des - x_ave
803         yerr = a.y_des - y_ave
804         zerr = a.z_des - z_ave
805
806
807         #if i%50 == 0:
808         print 'x_ave = ',x_ave
809         print 'y_ave = ',y_ave
810         print 'z_ave = ',z_ave
811
812         print 'i = ',i
813         print 'xerr, yerr, zerr = ',xerr,',',yerr,',',zerr
814         print 'sqrt( xerr**2 + yerr**2 + zerr**2 ) = ',sqrt( xerr**2 + yerr**2 + zerr**2 )
815         #a.print_dump(3)
816
817
818
819     # Stopping Criteria: if the agent is within a 5 cm error sphere for 200 time steps ( .2 sec )
820
821     if ( sqrt( xerr**2 + yerr**2 + zerr**2 ) < 10**-2) and (i>50):
822
823         print 'set point reached!!'
824
825         print 'i = ', i
826
827         break
828
829     if ( sqrt( xerr**2 + yerr**2 + zerr**2 ) > 200) and (i>50):
830
831         print 'you are lost!!'
832
833         print 'i = ', i
834
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
882     for k in k_th_variable_list:
883
884         if math.isnan(k):
885
886             a.print_dump(1)
887
888             break
889

```

```
891         if a.phi[-1] > 5: break
892         if a.theta[-1] > 5: break
893         if a.psi[-1] > 5: break
894
895         if math.isnan(xerr):
896
897             break
898
899     print '#####\n\n'
900
901     a.print_dump(10)
902
903     fig1_file_path = '/home/ek/Dropbox/THESIS/python_scripts/fig1_agent_module.png'
904     fig2_file_path = '/home/ek/Dropbox/THESIS/python_scripts/fig2_agent_module.png'
905
906     a.plot_results(False, True, fig1_file_path, fig2_file_path)
```

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

Appendix B

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
      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:
```

```

34     #print 'x_ave = ',x_ave
35     #print 'y_ave = ',y_ave
36     #print 'z_ave = ',z_ave

37
38     print 'xerr, yerr, zerr = ',xerr,', ',yerr,', ',zerr
39     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):

41
42         print 'set point reached!!'

43
44         #print 'i = ', i

45
46         return 1

47
48     if ( sqrt( xerr**2 + yerr**2 + zerr**2 ) > 200) and (i>50):

49
50         print 'you are lost!!'

51
52
53         return 0

54
55     k_th_variable_list = [
56         a.xacc_comm[-1],a.yacc_comm[-1],a.zacc_comm[-1],
57         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],
59         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],
61         a.tao_qr_frame[-1][0],a.tao_qr_frame[-1][1],a.tao_qr_frame[-1][2],
62         a.T[-1],
63         a.phi[-1],a.theta[-1],a.psi[-1],
64         a.phidot[-1],a.thetadot[-1],a.psidot[-1],
65         a.phiddot[-1],a.thetaddot[-1],a.psidot[-1],
66         a.x[-1],a.y[-1],a.z[-1],
67         a.xdot[-1],a.ydot[-1],a.zdot[-1],
68         a.xddot[-1],a.yddot[-1],a.zddot[-1],
69         a.x_integral_error[-1],a.y_integral_error[-1],a.z_integral_error[-1],
70     ]

71
72     for k in k_th_variable_list:

73
74         if math.isnan(k):

75
76             a.print_dump(1)

77
78             return 0

79
80
81     if a.phi[-1] > 5: return 0

82
83     if a.theta[-1] > 5: return 0

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

86
87     if math.isnan(xerr): return 0

```

```

88 #####
90
91 if __name__ == '__main__':
92
93     a = agent(x_0 = 0,
94               y_0 = 0,
95               z_0 = 0,
96               initial_setpoint_x = 1,
97               initial_setpoint_y = 1,
98               initial_setpoint_z = 1,
99               agent_priority = 1)
100
101     #the following gains worked well for a setpoint of (1,1,1)
102
103     a.kpx = 15      # -----PID proportional gain values
104     a.kpy = 15
105     a.kpz = 50
106
107     a.kdx = 10      #-----PID derivative gain values
108     a.kdy = 10
109     a.kdz = 50
110
111     a.kix = 0.8
112     a.kiy = 0.8
113     a.kiz = 15
114
115     a.kpphi = 4 # gains for the angular pid control laws
116     a.kptheta = 4
117     a.kppsi = 4
118
119     a.kdphi = 6
120     a.kdtheta = 6
121     a.kdpsi = 6
122     '''
123     u'a0.ending_iteration': 257,                the optimal run
124     u'a0.kdx': 10,
125     u'a0.kdy': 10,
126     u'a0.kdz': 50,
127     u'a0.kix': 0.8,
128     u'a0.kiy': 0.8,
129     u'a0.kiz': 20,
130     u'a0.kpx': 15,
131     u'a0.kpy': 15,
132     u'a0.kpz': 40,
133     u'ith_runtime': 11.414505958557129,
134     u'return_val2': 1,
135     u'setpoint': [1, 1, 2],
136     u'total_thrust': 4969.888344602483,
137     u'x_crossings': 3,
138     u'x_over_shoot': 0.024969492375947366,
139     u'y_crossings': 1,
140     u'y_over_shoot': 0.01858534082026475,
141     u'z_crossings': 1,
142     u'z_over_shoot': 0.09928387955818296}

```

```
144
146
148
150
152
154
156
158
160
162
164
166
168

'''
a.max_iterations = 1000

position_setpoint_list = [[0,0,1],[1,1,2]] # [[1,1,1],[5,5,5],[20,20,20],[100,100,100]]
# [[1,1,1],[5,1,1],[5,5,1],[5,5,5],[1,5,5],[1,1,5],[1,1,1]] #

for ss in range(len(position_setpoint_list)):

    a.x_des = position_setpoint_list[ss][0]
    a.y_des = position_setpoint_list[ss][1]
    a.z_des = position_setpoint_list[ss][2]

    go(a)

print '#####\n\n'

a.print_dump(10)

fig1_file_path = '/home/ek/Dropbox/THESIS/python_scripts/fig1_agent_module.png'
fig2_file_path = '/home/ek/Dropbox/THESIS/python_scripts/fig2_agent_module.png'

a.plot_results()
```

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

Appendix C

bruteForceFunctions.py

```
breakatwhitespace
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
from agent_module import *

from numpy import mean

import json

#-----
#-----

def run(agent, plots = False):

    for i in range(agent.max_iterations):

        agent.ending_iteration = i

        agent.system_model_block()

        agent.control_block()

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

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

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

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



```

33         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     #-----
84     #-----
85
86 def take_off():
87
88     agent0 = agent(x_0 = 0,

```

```

87         y_0 = 0,
        z_0 = 0,
        initial_setpoint_x = 0,
89         initial_setpoint_y = 0,
        initial_setpoint_z = 1,
91         agent_priority = 1)

93     agent0.max_iterations = 800

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

97     agent0.kpx = 40      # -----PID proportional gain values
    agent0.kpy = 40
99     agent0.kpz = 40

101     agent0.kdx = 25      #-----PID derivative gain values
    agent0.kdy = 25
103     agent0.kdz = 40

105     agent0.kix = .2
    agent0.kiy = .2
107     agent0.kiz = 40

109     run(agent0)

111     return agent0      # return the agent instance hovering at (0,0,1)

113 #-----
114 #-----
115 def test_gain_vector(a0, set_point, gain_dictionary):

117     a0.x_des = set_point[0]
    a0.y_des = set_point[1]
119     a0.z_des = set_point[2]

121     a0.max_iterations = 1000

123     a0.kpx = gain_dictionary['kpxy']
    a0.kix = gain_dictionary['kixy']
125     a0.kdx = gain_dictionary['kdxxy']

127     a0.kpy = gain_dictionary['kpxy']
    a0.kiy = gain_dictionary['kixy']
129     a0.kdy = gain_dictionary['kdxxy']

131     a0.kpz = gain_dictionary['kpz']
    a0.kiz = gain_dictionary['kiz']
133     a0.kdz = gain_dictionary['kdz']

135     return_val2 = run(a0)
    ,,,

137     #-----

    variable_dictionary = {

139         'a0.xacc_comm' : a0.xacc_comm, 'a0.yacc_comm' : a0.yacc_comm, 'a0.zacc_comm' : a0.zacc_comm,
        'a0.theta_comm' : a0.theta_comm, 'a0.phi_comm' : a0.phi_comm, 'a0.T_comm' : a0.T_comm,

```

```

141         'a0.tao_phi_comm' : a0.tao_phi_comm, 'a0.tao_theta_comm' : a0.tao_theta_comm, 'a0.tao_psi_comm'
      : a0.tao_psi_comm,
      'a0.w1_arg' : a0.w1_arg, 'a0.w2_arg' : a0.w2_arg, 'a0.w3_arg' : a0.w3_arg, 'a0.w4_arg' : a0.
143 w4_arg,
      'a0.w1' : a0.w1, 'a0.w2' : a0.w2, 'a0.w3' : a0.w3, 'a0.w4' : a0.w4,
      'a0.tao_qr_frame[0]' : a0.tao_qr_frame[0].tolist(), 'a0.tao_qr_frame[1]' : a0.tao_qr_frame[1].
145 tolist(), 'a0.tao_qr_frame[2]' : a0.tao_qr_frame[2].tolist(),
      'a0.T' : a0.T,
      'a0.phi' : a0.phi, 'a0.theta' : a0.theta, 'a0.psi' : a0.psi,
147 'a0.phidot' : a0.phidot, 'a0.thetadot' : a0.thetadot, 'a0.psidot' : a0.psidot,
      'a0.phiddot' : a0.phiddot, 'a0.thetaddot' : a0.thetaddot, 'a0.psidot' : a0.psidot,
149 'a0.x' : a0.x, 'a0.y' : a0.y, 'a0.z' : a0.z,
      'a0.xdot' : a0.xdot, 'a0.ydot' : a0.ydot, 'a0.zdot' : a0.zdot,
151 'a0.xddot' : a0.xddot, 'a0.yddot' : a0.yddot, 'a0.zddot' : a0.zddot,
      'a0.x_integral_error' : a0.x_integral_error,
153 'a0.y_integral_error' : a0.y_integral_error,
      'a0.z_integral_error' : a0.z_integral_error,
155     }

#-----
157 '''
158
159 if return_val2 != 1:

161     test_run_dictionary = {'setpoint':set_point,
      'a0.kpx' : a0.kpx,
163     'a0.kix' : a0.kix,
      'a0.kdx' : a0.kdx,
165     'a0.kpy' : a0.kpy,
      'a0.kiy' : a0.kiy,
167     'a0.kdy' : a0.kdy,
      'a0.kpz' : a0.kpz,
169     'a0.kiz' : a0.kiz,
      'a0.kdz' : a0.kdz,
171     'return_val2' : return_val2,
      'a0.ending_iteration' : a0.ending_iteration
173     }# 'variable_dictionary':variable_dictionary
      #}

175
176     return test_run_dictionary
177
178
179
180 elif return_val2 ==1:
181
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
188
189     for i in range(len(a0.x)-1):
190
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
198             y_crossings += 1
199
200         if sign( a0.z[i] - a0.z_des ) != sign( a0.z[i+1] - a0.z_des ) :
201
202             z_crossings += 1
203
204
205
206
207     #-----
208
209
210     if (max(a0.x) - a0.x_des) > 0: x_over_shoot = max(a0.x) - a0.x_des
211
212     if (max(a0.y) - a0.y_des) > 0: y_over_shoot = max(a0.y) - a0.y_des
213
214     if (max(a0.z) - a0.z_des) > 0: z_over_shoot = max(a0.z) - a0.z_des
215
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                           }
242
243     return test_run_dictionary
244
245 #-----
246
247 { u'a0.ending_iteration': 266,

```

```

249     u'a0.kdx': 5,
250     u'a0.kdy': 5,
251     u'a0.kdz': 20,
252     u'a0.kix': 0.8,
253     u'a0.kiy': 0.8,
254     u'a0.kiz': 15,
255     u'a0.kpx': 5,
256     u'a0.kpy': 5,
257     u'a0.kpz': 30,
258     u'ith_runtime': 7.14291787147522,
259     u'return_val2': 1,
260     u'setpoint': [1, 1, 2],
261     u'total_thrust': 4973.812742102159,
262     u'x_crossings': 1,
263     u'x_over_shoot': 0.06537601013649552,
264     u'y_crossings': 1,
265     u'y_over_shoot': 0.0706587304875288,
266     u'z_crossings': 1,
267     u'z_over_shoot': 0.03570385076740079}
268
269     '''
270
271     #-----
272
273     if __name__ == '__main__':
274
275         gain_dictionary = {
276             'kpxy' : 5,
277             'kpz' : 30,
278             'kdx' : 5,
279             'kdz' : 20,
280             'kixy' : 0.8,
281             'kiz' : 15}
282
283         agent = take_off() # ----> returns the agent instance hovering at (0,0,1)
284
285         set_point = [1,1,2]
286
287         test_run_dictionary = test_gain_vector(agent, set_point, gain_dictionary)
288
289         print 'test_run_dictionary = ',test_run_dictionary
290
291         agent.plot_results()

```

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

Appendix D

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
30 kpxy_range = arange(5,30,5)
32 kpx_range = arange(20,70,10)
```

```

kdx_range = arange(5,30,5)
34
kdz_range = arange(20,70,10)
36
kixy_range = arange(0.2,1.0,0.2)
38
kiz_range = arange(15,45,5)
40
print 'kpxy_range = ',kpxy_range
42
print 'kpz_range = ',kpz_range
44
print 'kdx_range = ',kdx_range
46
print 'kdz_range = ',kdz_range
48
print 'kixy_range = ',kixy_range
50
print 'kiz_range = ',kiz_range
52
number_of_sims = len(kpxy_range)*len(kpz_range)*len(kdx_range)*len(kdz_range)*len(kixy_range)*len(kiz_range)
54
print 'number_of_sims = ',number_of_sims
56
58
index = int(sys.argv[1])
60
runtimes = []
run_dictionaries = []
62
for kpxy in [kpxy_range[index]]:
64     for kpz in kpz_range:
66         for kdx in kdx_range:
68             for kdz in kdz_range:
70                 for kixy in kixy_range:

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

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

72
                    with open(filepath, 'wb') as fp:
74                        json.dump(run_dictionaries, fp)
76                        fp.close()

78
                    run_dictionaries = []

80
                    for kiz in kiz_range:

82                        gain_dictionary = {
84                            'kpxy' : kpxy,
86                            'kpz' : kpz,
                            'kdx' : kdx,
                            'kdz' : kdz,

```

```

88         'kixy' : kixy,
          '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
        )+'_ ' + 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

parseResults.py

```
breakatwhitespace
2
import os
4 import json
import itertools
6 from operator import itemgetter
import pprint
8 pp = pprint.PrettyPrinter(indent=4)
10 '''
def obj_fun(d):
12
    of = d['total_thrust']
14
    return of
16 '''
18
20
# need to go through all the output files and make a list of all the sims that still need to be run:
22
output_dir = '/home/ek/Dropbox/THESIS/python_scripts/brute_force_output_data/'
24
output_file_names = [fn for fn in os.listdir(output_dir)]
26
output_file_paths = [output_dir + ofn for ofn in output_file_names]
28
30 data = []
32 for ofp in output_file_paths:
```

```

34     with open(ofp, 'rb') as fp:
35         output_data = json.load(fp)
36
37         data = data + output_data
38
39
40 good_runs = []
41
42 for d in data:
43
44     if d['return_val2'] == 1:
45
46         good_runs.append(d)
47
48
49 min_overshoot_runs = []
50
51 for g in good_runs:
52
53     if ( g['x_over_shoot'] < 0.1 ) and ( g['y_over_shoot'] < 0.1 ) and ( g['z_over_shoot'] < 0.1 ):
54
55         min_overshoot_runs.append(g)
56
57
58 min_oscillation_runs = []
59
60 for r in min_overshoot_runs:
61
62     if (r['x_crossings'] < 4) and (r['y_crossings'] < 4) and (r['z_crossings'] < 4) :
63
64         min_oscillation_runs.append(r)
65
66
67 thrust_sorted_good_runs = sorted(min_oscillation_runs, key=itemgetter('total_thrust'))
68
69 for t in thrust_sorted_good_runs[:20]:
70     print '\n'
71     pp.pprint(t)
72
73
74
75
76 # according to the available data, here is the best run...
77 '''
78 {   u'a0.ending_iteration': 266,
79     u'a0.kdx': 5,
80     u'a0.kdy': 5,
81     u'a0.kdz': 20,
82     u'a0.kix': 0.8,
83     u'a0.kiy': 0.8,
84     u'a0.kiz': 15,
85     u'a0.kpx': 5,
86     u'a0.kpy': 5,

```

```
88     u'a0.kpz': 30,  
    u'ith_runtime': 7.14291787147522,  
90     u'return_val2': 1,  
    u'setpoint': [1, 1, 2],  
92     u'total_thrust': 4973.812742102159,  
    u'x_crossings': 1,  
94     u'x_over_shoot': 0.06537601013649552,  
    u'y_crossings': 1,  
96     u'y_over_shoot': 0.0706587304875288,  
    u'z_crossings': 1,  
98     u'z_over_shoot': 0.03570385076740079}  
100  
102  
'''
```

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

Appendix F

finiteDiffSolution.py

```
breakatwhitespace
1 from os import system

3 from numpy import cos as c , sin as s , array as a , concatenate , arange , sqrt , reshape , log

5 from numpy import dot
  from numpy.linalg import inv
7 from numpy.linalg import norm
  from numpy import transpose
9 global ixx
  global iyy
11 global izz

13 ixx = 5.0*10**-3
  iyy = 5.0*10**-3
15 izz = 10.0*10**-3

17 global g
  global s
19 global l
  global b
21 global m
  global h
23 global d

25 g = -9.8
  alpha = 0.001
27 l = 0.25 # m
  b = 0.001
29 m = 1.

31 h = 0.1
```

```

33 d = 0.0001 # the value for adding to the input variables of f to express the finite differences
35 #-----the jacobian for transforming from body frame to
    inertial frame
37 def J(ph,th):
39     jac = a([
41         [ixx          , 0                                , -ixx * s(th)
          ],
         [0            , iyy*(c(ph)**2) + izz * s(ph)**2  , (iyy-izz)*c(ph)*s(ph)*c(th)
          ],
         [-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)
          **2)*(c(th)**2)]
43     ])
45     #print '\n\njac = \n',jac
47
48     return jac
49
50
51
52
53
54
55
56
57 #
    -----
58
59 def coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 ,
    partial_with_respect_to ):
61
62     # the argument 'partial_with_respect_to' specifies the angular quantity for which the derivative is being
    computed
63
64     # note that 'partial_with_respect_to' MUST be a string
65
66     # if the standard coriolis matrix is needed, the argument: 'partial_with_respect_to' should be set to
    None
67
68     if partial_with_respect_to == 'phd':    phd = ( ( ph_k - ph_k_minus_1 ) / h ) + d
69
70     else: phd = ( ph_k - ph_k_minus_1 ) / h
71
72
73
74     if partial_with_respect_to == 'thd':    thd = ( ( th_k - th_k_minus_1 ) / h ) + d
75
76     else: thd = ( th_k - th_k_minus_1 ) / h
77

```

```

79
80
81     if partial_with_respect_to == 'psd':      psd = ( ( ps_k - ps_k_minus_1 ) / h ) + d
82
83
84
85
86
87     # here are the elements in the matrix
88
89     c11 = 0
90
91     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)
92         - ixx*psd*c(th_k)
93
94     c13 = (izz-iyy) * psd * c(ph_k) * s(ph_k) * c(th_k)**2
95
96     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)
97         + ixx * psd * c(th_k)
98
99     c22 = (izz-iyy)*phd*c(ph_k)*s(ph_k)
100
101     c23 = -ixx*psd*s(th_k)*c(th_k) + iyy*psd*(s(ph_k)**2)*s(th_k)*c(th_k)
102
103     c31 = (iyy-izz)*phd*(c(th_k)**2)*s(ph_k)*c(ph_k) - ixx*thd*c(th_k)
104
105     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
106         (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)*
107         c(th_k)
108
109     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
110         *(c(ph_k)**2)*c(th_k)*s(th_k) + ixx*thd*c(th_k)*s(th_k)
111
112
113     cm = a([[c11,c12,c13],
114             [c21,c22,c23],
115             [c31,c32,c33]])
116
117
118     #print '\n\ncm = \n',cm
119
120
121
122
123
124
125     return cm
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

```

127 # the function f will be used as the state equations as well as for the many partial derivatives
129
130 def f(x , x_k_minus_1 , x_k_minus_2 ,
131      y , y_k_minus_1 , y_k_minus_2 ,
132      z , z_k_minus_1 , z_k_minus_2 ,
133      ph_k , ph_k_minus_1 , ph_k_minus_2 ,
134      th_k , th_k_minus_1 , th_k_minus_2 ,
135      ps_k , ps_k_minus_1 , ps_k_minus_2 ,
136      u1_k , u2_k , u3_k , u4_k ):
137
138     T = alpha * (u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2)
139     #print '\nT = ',T
140
141     xdd = (1/h**2) * ( x - 2*x_k_minus_1 - x_k_minus_2 )
142
143     x_residual = T/m * ( c(ps_k) * s(th_k) * c(ph_k) + s(ps_k) * s(ph_k) ) # F1
144
145
146     ydd = (1/h**2) * ( x - 2*y_k_minus_1 - y_k_minus_2 )
147
148     y_residual = T/m * ( s(ps_k) * s(th_k) * c(ph_k) + c(ps_k) * s(ph_k) ) # F2
149
150
151     zdd = (1/h**2) * ( x - 2*z_k_minus_1 - z_k_minus_2 )
152
153     z_residual = T/m * c(th_k) * c(ph_k) + g # F3
154
155
156 # the angular equations are better kept as vectors and matrices
157
158
159 input_func_vector = a([
160     1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
161     1 * alpha * ( -u1_k**2 + u3_k**2 ) ,
162     b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
163 ])
164
165 #print '\ninput_func_vector = ',input_func_vector
166
167 coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1, None )
168 #print '\ncoriolis_k = ',coriolis_k
169
170
171 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 ] )
172 #print '\nang_vel_vector = ',ang_vel_vector
173
174
175
176 # this is the large bracketed factor which is multiplied into the inverse of the jacobian
177
178 temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )
179 #print '\ntemp_factor = ',temp_factor
180
181

```

```

183     etadd = ( 1/h**2 ) * a([
184         ph_k - 2* ph_k_minus_1 - ph_k_minus_2,
185         th_k - 2* th_k_minus_1 - th_k_minus_2,
186         ps_k - 2* ps_k_minus_1 - ps_k_minus_2,
187     ])

188
189     angular_residual = dot( inv( J(ph_k,th_k) ) , temp_factor ) - etadd
190     #print '\netadd = ',etadd
191
192
193     state_equations_residual = a( [ x_residual , y_residual , z_residual ,angular_residual[0] ,
194         angular_residual[1] , angular_residual[2] ] )
195
196     #print '\n\nstate_equations_residual = \n',state_equations_residual
197
198
199     return state_equations_residual
200
201
202
203     #
204     -----
205
206     # The costate
207
208     def costate(x_k , x_k_minus_1 , x_k_minus_2 ,
209         y_k , y_k_minus_1 , y_k_minus_2 ,
210         z_k , z_k_minus_1 , z_k_minus_2 ,
211         ph_k , ph_k_minus_1 , ph_k_minus_2,
212         th_k , th_k_minus_1 , th_k_minus_2,
213         ps_k , ps_k_minus_1 , ps_k_minus_2,
214         u1_k , u2_k , u3_k , u4_k ,
215         la1_k , la1_k_minus_1 , la1_k_minus_2,
216         la2_k , la2_k_minus_1 , la2_k_minus_2,
217         la3_k , la3_k_minus_1 , la3_k_minus_2,
218         la4_k , la4_k_minus_1 , la4_k_minus_2,
219         la5_k , la5_k_minus_1 , la5_k_minus_2,
220         la6_k , la6_k_minus_1 , la6_k_minus_2
221     ):
222
223         # this is the list of state equations differenciated with respect to phi:
224
225         ff = f(x , x_k_minus_1 , x_k_minus_2 ,
226             y , y_k_minus_1 , y_k_minus_2 ,
227             z , z_k_minus_1 , z_k_minus_2 ,
228             ph_k , ph_k_minus_1 , ph_k_minus_2,
229             th_k , th_k_minus_1 , th_k_minus_2,
230             ps_k , ps_k_minus_1 , ps_k_minus_2,
231             u1_k , u2_k , u3_k , u4_k )
232
233

```



```

235     f_at_phi_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
236                        y , y_k_minus_1 , y_k_minus_2 ,
237                        z , z_k_minus_1 , z_k_minus_2 ,
238                        ph_k + d, ph_k_minus_1 , ph_k_minus_2,
239                        th_k      , th_k_minus_1 , th_k_minus_2,
240                        ps_k      , ps_k_minus_1 , ps_k_minus_2,
241                        u1_k , u2_k , u3_k , u4_k )
242
243     del_f_d_phi = ( f_at_phi_plus_d - ff )/d
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     f_at_psi_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
259                        y , y_k_minus_1 , y_k_minus_2 ,
260                        z , z_k_minus_1 , z_k_minus_2 ,
261                        ph_k      , ph_k_minus_1 , ph_k_minus_2,
262                        th_k      , th_k_minus_1 , th_k_minus_2,
263                        ps_k + d, ps_k_minus_1 , ps_k_minus_2,
264                        u1_k , u2_k , u3_k , u4_k )
265
266     del_f_d_psi = ( f_at_psi_plus_d - ff )/d
267
268
269     state_transition_matrix = a( [ del_f_d_phi ,
270                                  del_f_d_theta,
271                                  del_f_d_psi      ] )
272
273     #print '\n\nstate_transition_matrix = ',state_transition_matrix
274
275
276     lam = a( [ la1_k , la2_k , la3_k , la4_k , la5_k , la6_k ] )
277
278     la_k_double_dot = (1/h)*a([
279
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                                ])

```

```

289     costate_residual_vector = dot( state_transition_matrix , lam ) - la_k_double_dot[3:]
    #print '\n\ncostate_residual_vector = \n',costate_residual_vector
291     return costate_residual_vector

293
295 #
    -----

297 # we must compute the partials of the angular state equations ( etadd ) with respect to the angular
    velocities ( phi_dot, theta_dot, psi_dot )

299
301 # this function just evaluates the angular state equations replacing "(ph_k - ph_k_minus_1)/h " with "( (
    ph_k - ph_k_minus_1)/h ) + d"

303 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 ,
    u3_k , u4_k ):

305     input_func_vector = a([
        1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
307         1 * alpha * ( -u1_k**2 + u3_k**2 ) ,
        b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
309     ])

311     coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 , 'phd' )

313     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)/
        h ] )

315     temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )

317     etadd_phi_plus_d = dot( inv( J(ph_k,th_k) ) , temp_factor )
    #print '\n\netadd_phi_plus_d = \n',etadd_phi_plus_d
319     return a( etadd_phi_plus_d )

321
323 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 ,
    u3_k , u4_k ):

325     input_func_vector = a([
        1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
327         1 * alpha * ( -u1_k**2 + u3_k**2 ) ,
        b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
329     ])

331     coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 , 'thd' )

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)/
        h ] )

335     temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )

```

```

337     etadd_theta_plus_d = dot( inv( J(ph_k,th_k) ) , temp_factor )
338     #print '\n\netadd_theta_plus_d = \n',etadd_theta_plus_d
339     return a( etadd_theta_plus_d )
340
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 ,
    u3_k , u4_k ):
344
345     input_func_vector = a([
346         1 * alpha * ( -u2_k**2 + u4_k**2 ) ,
347         1 * alpha * ( -u1_k**2 + u3_k**2 ) ,
348         b*(u1_k**2 + u2_k**2 + u3_k**2 + u4_k**2) ,
349     ])
350
351     coriolis_k = coriolis_matrix( ph_k , th_k , ps_k , ph_k_minus_1 , th_k_minus_1 , ps_k_minus_1 , 'psd' )
352
353     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
    ] )
354
355     temp_factor = input_func_vector - dot( coriolis_k , ang_vel_vector )
356
357     etadd_psi_plus_d = dot( inv( J(ph_k,th_k) ) , temp_factor )
358     #print '\n\netadd_psi_plus_d = \n',etadd_psi_plus_d
359     return a( etadd_psi_plus_d )
360
361
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 )
379
380     etadd_theta_plus_d = dot( inv( J(ph_k,th_k) ) , temp_factor )
381     #print '\n\netadd_theta_plus_d = \n',etadd_theta_plus_d
382     return a( etadd_theta_plus_d )
383
384
385
386 #
387
388 -----
389 print '\n\n = \n',

```

```

387
388 # The algebraic costate equations
389
390 def algebraic_costate_equations(
391     ph_k , ph_k_minus_1 , th_k , th_k_minus_1 , ps_k , ps_k_minus_1 ,
392     u1_k , u2_k , u3_k , u4_k ,
393     la4_k ,
394     la5_k ,
395     la6_k
396 ):
397
398     # the pnumeonic for the following three assignments is ppd -> phi plus d
399     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 ,
400     u2_k , u3_k , u4_k )
401
402     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 ,
403     u2_k , u3_k , u4_k )
404
405     pps = 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 ,
406     u2_k , u3_k , u4_k )
407
408     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 ,
409     u3_k , u4_k )
410
411     del_f_d_phi_dot = (ppd-eta_double_dot)/d
412
413     del_f_d_theta_dot = (tpd-eta_double_dot)/d
414
415     del_f_d_psi_dot = (pps-eta_double_dot)/d
416
417     algebraic_transition_matrix = a( [ del_f_d_phi_dot , del_f_d_theta_dot , del_f_d_psi_dot ] )#.reshape
418     ([3,3])
419
420     #print '\n\algebraic_transition_matrix = ',algebraic_transition_matrix
421
422     la4_through_6 = a( [ la4_k , la5_k , la6_k ] )
423     #print '\n\nla4_through_6 = ',la4_through_6
424
425     algebraic_costate_equations_residual_vector = dot( algebraic_transition_matrix , la4_through_6 )
426     #print '\n\nalgebraic_costate_equations_residual_vector = \n',algebraic_costate_equations_residual_vector
427     return algebraic_costate_equations_residual_vector
428
429 #-----# stationarity
430     conditions:
431
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 ,

```

```

437         ph_k , ph_k_minus_1 , ph_k_minus_2,
         th_k , th_k_minus_1 , th_k_minus_2,
439         ps_k , ps_k_minus_1 , ps_k_minus_2,
         u1_k , u2_k , u3_k , u4_k,
         la1_k , la2_k , la3_k , la4_k , la5_k , la6_k ):
441
443     ff = f(x , x_k_minus_1 , x_k_minus_2 ,
445           y , y_k_minus_1 , y_k_minus_2 ,
447           z , z_k_minus_1 , z_k_minus_2 ,
449           ph_k , ph_k_minus_1 , ph_k_minus_2,
           th_k , th_k_minus_1 , th_k_minus_2,
           ps_k , ps_k_minus_1 , ps_k_minus_2,
           u1_k , u2_k , u3_k , u4_k )
451
453     ff_u1_k_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
455           y , y_k_minus_1 , y_k_minus_2 ,
457           z , z_k_minus_1 , z_k_minus_2 ,
           ph_k , ph_k_minus_1 , ph_k_minus_2,
           th_k , th_k_minus_1 , th_k_minus_2,
           ps_k , ps_k_minus_1 , ps_k_minus_2, u1_k + d, u2_k , u3_k , u4_k )
459
461     ff_u2_k_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
463           y , y_k_minus_1 , y_k_minus_2 ,
465           z , z_k_minus_1 , z_k_minus_2 ,
           ph_k , ph_k_minus_1 , ph_k_minus_2,
           th_k , th_k_minus_1 , th_k_minus_2,
           ps_k , ps_k_minus_1 , ps_k_minus_2, u1_k , u2_k + d , u3_k , u4_k )
467
469     ff_u3_k_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
471           y , y_k_minus_1 , y_k_minus_2 ,
           z , z_k_minus_1 , z_k_minus_2 ,
           ph_k , ph_k_minus_1 , ph_k_minus_2,
           th_k , th_k_minus_1 , th_k_minus_2,
           ps_k , ps_k_minus_1 , ps_k_minus_2, u1_k , u2_k , u3_k + d , u4_k )
473
475     ff_u4_k_plus_d = f(x , x_k_minus_1 , x_k_minus_2 ,
477           y , y_k_minus_1 , y_k_minus_2 ,
           z , z_k_minus_1 , z_k_minus_2 ,
           ph_k , ph_k_minus_1 , ph_k_minus_2,
           th_k , th_k_minus_1 , th_k_minus_2,
           ps_k , ps_k_minus_1 , ps_k_minus_2, u1_k , u2_k , u3_k , u4_k + d )
479
481     # note this is the TRANSPOSE of the matrix of partials of f WRT u
483     dfdq1 = (ff_u1_k_plus_d - ff)/d
485     dfdq2 = (ff_u2_k_plus_d - ff)/d
487     dfdq3 = (ff_u3_k_plus_d - ff)/d
489     dfdq4 = (ff_u4_k_plus_d - ff)/d
491
493     lambda_k = [ la1_k , la2_k , la3_k , la4_k , la5_k , la6_k ]
495
497     individ_rows = a([ dot( dfdq1 , lambda_k ),

```

```

491         dot( dfdq2 ,lambda_k ),
           dot( dfdq3 ,lambda_k ),
493         dot( dfdq4 ,lambda_k )
           ])
495
497
stationarity_conditions_residual_vector = individ_rows + 2* a( [ u1_k , u2_k , u3_k , u4_k ] )
499 #print '\n\nstationarity_conditions_residual_vector = \n',stationarity_conditions_residual_vector
return stationarity_conditions_residual_vector
501
503
505
#-----# the objective
function at the kth timestep
507
509 def G_at_k(x_k, x_k_minus_1 , x_k_minus_2 ,
           y_k , y_k_minus_1 , y_k_minus_2 ,
511           z_k , z_k_minus_1 , z_k_minus_2 ,
           ph_k , ph_k_minus_1 , ph_k_minus_2,
513           th_k , th_k_minus_1 , th_k_minus_2,
           ps_k , ps_k_minus_1 , ps_k_minus_2,
515           u1_k , u2_k , u3_k , u4_k,
           la1_k , la1_k_minus_1 , la1_k_minus_2,
517           la2_k , la2_k_minus_1 , la2_k_minus_2,
           la3_k , la3_k_minus_1 , la3_k_minus_2,
519           la4_k , la4_k_minus_1 , la4_k_minus_2,
           la5_k , la5_k_minus_1 , la5_k_minus_2,
521           la6_k , la6_k_minus_1 , la6_k_minus_2
           ):
523
state_vector_residual = f(x_k , x_k_minus_1 , x_k_minus_2 ,
525           y_k , y_k_minus_1 , y_k_minus_2 ,
           z_k , z_k_minus_1 , z_k_minus_2 ,
527           ph_k , ph_k_minus_1 , ph_k_minus_2,
           th_k , th_k_minus_1 , th_k_minus_2,
529           ps_k , ps_k_minus_1 , ps_k_minus_2, u1_k , u2_k , u3_k , u4_k )
#print '\n\nstate_vector_residual = \n',state_vector_residual
531
533 costate_residual = costate(x_k , x_k_minus_1 , x_k_minus_2 ,
           y_k , y_k_minus_1 , y_k_minus_2 ,
535           z_k , z_k_minus_1 , z_k_minus_2 ,
           ph_k , ph_k_minus_1 , ph_k_minus_2,
537           th_k , th_k_minus_1 , th_k_minus_2,
           ps_k , ps_k_minus_1 , ps_k_minus_2,
539           u1_k , u2_k , u3_k , u4_k,
           la1_k , la1_k_minus_1 , la1_k_minus_2,
541           la2_k , la2_k_minus_1 , la2_k_minus_2,
           la3_k , la3_k_minus_1 , la3_k_minus_2,
543           la4_k , la4_k_minus_1 , la4_k_minus_2,
           la5_k , la5_k_minus_1 , la5_k_minus_2,

```

```

545         la6_k , la6_k_minus_1 , la6_k_minus_2
546     )
547     #print '\n\ncostate_residual = \n',costate_residual
548
549
550
551
552     algebraic_costate_equations_residual_vector = algebraic_costate_equations(
553         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 ,
555         la4_k ,
556         la5_k ,
557         la6_k
558     )
559     #print '\n\nalgebraic_costate_equations_residual_vector = \n',algebraic_costate_equations_residual_vector
560
561
562
563
564     stationarity_conditions_residual_vector = stationarity_conditions(x_k , x_k_minus_1 , x_k_minus_2 ,
565         y_k , y_k_minus_1 , y_k_minus_2 ,
566         z_k , z_k_minus_1 , z_k_minus_2 ,
567         ph_k , ph_k_minus_1 , ph_k_minus_2 ,
568         th_k , th_k_minus_1 , th_k_minus_2 ,
569         ps_k , ps_k_minus_1 , ps_k_minus_2 ,
570         u1_k , u2_k , u3_k , u4_k ,
571         la1_k , la2_k , la3_k , la4_k , la5_k , la6_k )
572
573     #print '\n\nstationarity_conditions_residual_vector = \n',stationarity_conditions_residual_vector
574
575
576
577
578     temp_array = a( concatenate([
579         state_vector_residual ,
580         costate_residual ,
581         algebraic_costate_equations_residual_vector ,
582         stationarity_conditions_residual_vector])
583     )
584
585     kth_residual = sum( temp_array )
586     #print '\n\nkth_residual = \n',kth_residual
587
588
589     return kth_residual
590
591
592
593
594
595
596
597 #-----
    full_objective_function

```

```

599 # the full objective function sums up all the contributions from each time step
601
602 def full_objective_function(N,
603                             x,y,z,
604                             ph,th,ps,
605                             u1,u2,u3,u4,
606                             la1,la2,la3,la4,la5,la6):
607
608     glist = []
609
610     for k in arange(2,N):
611
612         g_k = G_at_k(
613             x[k] , x[k-1], x[k-2],
614             y[k] , y[k-1], y[k-2],
615             z[k] , z[k-1], z[k-2],
616             ph[k] , ph[k-1], ph[k-2],
617             th[k] , th[k-1], th[k-2],
618             ps[k] , ps[k-1], ps[k-2],
619             u1[k] , u2[k] , u3[k] , u4[k],
620             la1[k] , la1[k-1] , la1[k-2],
621             la2[k] , la2[k-1] , la2[k-2],
622             la3[k] , la3[k-1] , la3[k-2],
623             la4[k] , la4[k-1] , la4[k-2],
624             la5[k] , la5[k-1] , la5[k-2],
625             la6[k] , la6[k-1] , la6[k-2]
626         )
627
628         glist.append(g_k)
629
630     objective_function_residual = sum(glist)
631
632     #print '\n\nobjective_function_residual = \n',objective_function_residual
633
634     return objective_function_residual
635
636
637
638 #----- gradient
639
640
641
642 def gradient(N,
643             x,y,z,
644             ph,th,ps,
645             u1,u2,u3,u4,
646             la1,la2,la3,la4,la5,la6
647             ):
648
649     grad = []
650
651     input_vars_1d = concatenate([
652         x[2:-1], y[2:-1], z[2:-1],

```



```

655         ph[2:-1], th[2:-1], ps[2:-1],
        u1[2:-1], u2[2:-1], u3[2:-1], u4[2:-1],
657         la1[2:-1], la2[2:-1], la3[2:-1], la4[2:-1], la5[2:-1], la6[2:-1]
        ])

659     obj_func_res = full_objective_function(N,
        x,y,z,
661         ph,th,ps,
        u1,u2,u3,u4,
663         la1,la2,la3,la4,la5,la6)

665     delta = 0.0001

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

669         aug_input = []
        #print 'aug_input = ',aug_input

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

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

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

679         #print 'aug_input_parsed = ',aug_input_parsed

681         x_aug = aug_input_parsed[0]
683         y_aug = aug_input_parsed[1]
        z_aug = aug_input_parsed[2]
685         ph_aug = aug_input_parsed[3]
        th_aug = aug_input_parsed[4]
687         ps_aug = aug_input_parsed[5]
        u1_aug = aug_input_parsed[6]
689         u2_aug = aug_input_parsed[7]
        u3_aug = aug_input_parsed[8]
691         u4_aug = aug_input_parsed[9]
        la1_aug = aug_input_parsed[10]
693         la2_aug = aug_input_parsed[11]
        la3_aug = aug_input_parsed[12]
695         la4_aug = aug_input_parsed[13]
        la5_aug = aug_input_parsed[14]
697         la6_aug = aug_input_parsed[15]

699

701         aug_obj_func_res = full_objective_function(N,
        x_aug,y_aug,z_aug,
703         ph_aug,th_aug,ps_aug,
        u1_aug,u2_aug,u3_aug,u4_aug,
705         la1_aug,la2_aug,la3_aug,la4_aug,la5_aug,la6_aug)

707         dg = ( aug_obj_func_res - obj_func_res )/delta

```

```

709         #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     # the terminal conditions for the control inputs are defined by the fact that we want the quadrotor to
744     # end in a hovering state
745
746     # this means that the total thrust must equal g, and that all the inputs (motor speeds) must be the same
747     '''
748     g = T
749
750     g = alpha * (u1**2 + u2**2 + u3**2 + u4**2)
751
752     g = alpha*4*u**2
753
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])])
765
767     la1 = concatenate( [ a([0,0]) , init , a([0]) ] )
769
771     la2 = concatenate( [ a([0,0]) , init , a([0]) ] )
773
775     la3 = concatenate( [ a([0,0]) , init , a([0]) ] )
777
779     la4 = concatenate( [ a([0,0]) , init , a([0]) ] )
781
783     la5 = concatenate( [ a([0,0]) , init , a([0]) ] )
785
787     la6 = concatenate( [ a([0,0]) , init , a([0]) ] )
789
791     '''
793     for i in [x,y,z,ph,th,ps,u1,u2,u3,u4,la1,la2,la3,la4,la5,la6]:
795         print len(i)
797     '''
799
801     #-----
803
805     initial_objective_function_residual = full_objective_function(N,
807         x,y,z,
809         ph,th,ps,
811         u1,u2,u3,u4,
813         la1,la2,la3,la4,la5,la6)
815
817     #print '\n\nobjective_function_residual= \n',objective_function_residual
819
821     obj_func_res_list = [initial_objective_function_residual]
823     grad_norm_list = []
825
827     max_iterations = 10
829     tol = 1
831
833     for iteration_number in range(max_iterations):
835
837         grad = gradient(N,
839             x,y,z,
841             ph,th,ps,
843             u1,u2,u3,u4,
845             la1,la2,la3,la4,la5,la6
847         )
849
851         grad_norm = norm( a( grad ))
853
855         print '-----iteration_number = ',iteration_number
857         print '\n\ngrad = \n',grad

```

```

819         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\ntstep = ',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         new_partial_input_vector = reshape( input_vars_1d - step , ( 16 , N ) )      # this does not
850         contain the boundary conditions hence the 'partial'
851
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]) ])
863         u2 = concatenate([ a([uhover,uhover]) , new_partial_input_vector[7] , a([uhover]) ])
864         u3 = concatenate([ a([uhover,uhover]) , new_partial_input_vector[8] , a([uhover]) ])
865         u4 = concatenate([ a([uhover,uhover]) , new_partial_input_vector[9] , a([uhover]) ])
866         la1 = concatenate([ a([0,0]) , new_partial_input_vector[10] , a([0])    ])
867         la2 = concatenate([ a([0,0]) , new_partial_input_vector[11] , a([0])    ])
868         la3 = concatenate([ a([0,0]) , new_partial_input_vector[12] , a([0])    ])
869         la4 = concatenate([ a([0,0]) , new_partial_input_vector[13] , a([0])    ])
870         la5 = concatenate([ a([0,0]) , new_partial_input_vector[14] , a([0])    ])
871         la6 = concatenate([ a([0,0]) , new_partial_input_vector[15] , a([0])    ])

```

```

873     print '\n\nx = \n',x
875     print '\n\nphi = \n',ph
877     print '\n\nu1 = \n',u1
877     print '\n\nla1 = \n',la1

879     objective_function_residual = full_objective_function(N,
881                                     x,y,z,
881                                     ph,th,ps,
883                                     u1,u2,u3,u4,
883                                     la1,la2,la3,la4,la5,la6)

885     print '\n\nobjective_function_residual = ',objective_function_residual

887     obj_func_res_list.append(objective_function_residual)

889
891     # -----TEST FOR CONVERGENCE

893
895     if objective_function_residual > obj_func_res_list[0]:

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

901
903
905     elif grad_norm < tol:

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

909
911     #wait = raw_input('\n\npress space to continue...')

913
915     t2 = time()

917     delta_t = t2-t1

919     print '-----'

921     print '\n\n\ndelta_t = ',delta_t

923     #####

925

```

```

927
929     '''
931     delta = 0.0001
933
934     for i in range(len(input_vars_1d)):
935
936         aug_input = []
937         #print 'aug_input = ',aug_input
938
939         for j in range(len(input_vars_1d)):
940             if j == i:
941                 aug_input.append(a(input_vars_1d[j] + delta) )
942
943             else: aug_input.append(input_vars_1d[j])
944
945         aug_input_parsed = reshape(aug_input, ( 16 , N ))
946
947         #print 'aug_input_parsed = ',aug_input_parsed
948
949         x_aug = aug_input_parsed[0]
950         y_aug = aug_input_parsed[1]
951         z_aug = aug_input_parsed[2]
952         ph_aug = aug_input_parsed[3]
953         th_aug = aug_input_parsed[4]
954         ps_aug = aug_input_parsed[5]
955         u1_aug = aug_input_parsed[6]
956         u2_aug = aug_input_parsed[7]
957         u3_aug = aug_input_parsed[8]
958         u4_aug = aug_input_parsed[9]
959         la1_aug = aug_input_parsed[10]
960         la2_aug = aug_input_parsed[11]
961         la3_aug = aug_input_parsed[12]
962         la4_aug = aug_input_parsed[13]
963         la5_aug = aug_input_parsed[14]
964         la6_aug = aug_input_parsed[15]
965
966
967         aug_obj_func_res = full_objective_function(N,
968             x_aug,y_aug,z_aug,
969             ph_aug,th_aug,ps_aug,
970             u1_aug,u2_aug,u3_aug,u4_aug,
971             la1_aug,la2_aug,la3_aug,la4_aug,la5_aug,la6_aug)
972
973
974         dg = ( aug_obj_func_res - obj_func_res )/delta
975
976         #print '\n\ndg = ',dg
977
978         grad.append(dg)      # grad returns as a one d list...
979
981

```

983

'''

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

Bibliography

[diy] URL <http://diydrones.com/>.

[air] URL <http://www.airware.com/>.

[faa] URL <http://www.faa.gov/about/initiatives/uas/>.

- [1] Elizabeth Bone and Christopher C Bolkcom. *Unmanned aerial vehicles: background and issues*. Nova Science Pub Incorporated, 2004.
- [2] Jack W Langelaan. Long distance/duration trajectory optimization for small uavs. In *AIAA Guidance, Navigation and Control Conference and Exhibit*, 2007.
- [3] 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.
- [4] 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.
- [5] Bora Erginer and Erdinc Altug. Modeling and pd control of a quadrotor vtol vehicle. pages 894–899, 2007.

-
- [6] 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>.
- [7] Teppo Luukkonen. Modelling and control of quadcopter. *Aalto University*, August 2011.
- [8] Jerry B Marion and Stephen T Thornton. *Classical dynamics of particles and systems*. Saunders College Pub., 1995.
- [9] Cornelius Lanczos. *The variational principles of mechanics*, volume 4. Courier Dover Publications, 1970.
- [10] J. Bernoulli. *Bending of light rays in transparent non-uniform media and the Solution to the problem of determing the Brachistochrone curve*. 1697.
- [11] 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.
- [12] L.S. Pontryagin V.G. Boltyanskii, R.V. Gamkrelidze. *Towards a theory of optimal processes, (Russian)*. Reports Acad. Sci. USSR, vol.110(1), 1956.
- [13] Frank L Lewis, Draguna Vrabie, and Vassilis L Syrmos. *Optimal control*. Wiley. com, 2012.
- [14] A. E. Bryson and Y. C. Ho. *Applied Optimal Control*. Blaisdell, New York, 1969.
- [15] Arturo Locatelli. *Optimal control: An introduction*. Springer, 2001.

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