AN ABSTRACT OF THE THESIS OF

Cary R. Maunder for the degree of Master of Science in Mechanical Engineering presented on September 22, 2006.

Title: Model Predictive Control For Sagittal Plane Locomotion.

Abstract approved:

John M. Schmitt

A distinct characteristic of legged locomotion is its periodic nature. This periodic motion, in the form of a periodic orbit, has been the target of many walking and running control strategies. The spring loaded inverted pendulum (SLIP) has become a popular model of sagittal plane locomotion, exhibiting behavior characteristic of a variety of legged animals. In this work, a model predictive control scheme is developed for the rigid body SLIP to drive the system to a periodic orbit. This is accomplished by defining a Poincaré map from one stride to the next and using numerical optimization each stride to select a leg touchdown angle that will best deliver the system to a desired fixed point of this map. The scheme is tested on both the point mass and rigid body SLIP models using parameter values that are characteristic of the cockroach, *Blaberus discoidalis*. It is found to increase the region of stability for both, as well as greatly improving the systems ability to recover from energy conservative perturbations.

Model Predictive Control For Sagittal Plane Locomotion

by
Cary R. Maunder

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September 22, 2006
Commencement June 2007

Masters of Science thesis of Cary R. Maunder Presented on September 22, 2006.

APPROVED:

Major Professor, representing Mechanical Engineering

Head of the Department of Mechanical Engineering

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Cary R. Maunder, Author

ACKNOWLEDGEMENTS

I thank my advisor, Dr. John Schmitt for his guidance and support of my research over the past two years, for investing his time and resources into my growth and learning, and for his encouragement or at least tolerance of my “independent” research techniques. I thank Jason Kyle for his companionship in the study of optimization and his advice on my research. I thank my father, Dr. Richard Maunder for his unconditional support of my interests, my mother Lucinda Maunder for helping me through tough times, and my stepmother Karen Maunder for giving me the drive to get things done.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF APPENDICES

1 INTRODUCTION

For centuries humans have looked to biology for inspiration in the development of machinery. The development of walking and running machines is no exception. Of particular interest is the cockroach, which has been found to move extremely fast for its size, taking rough terrain at full speed without falling [1]. Such locomotion characteristics would be very desirable for the creation of nimble robots. Extensive investigation into how the cockroach, *Blaberus descoidalis* walks and runs has shown that although it uses three legs per stance to walk and run, the legs act together to produce force and moment patterns similar to those of a biped [2], [3]. This in conjunction with similar findings from studies of creatures of other morphologies has resulted in the use of reduced order models to describe the motion of more complex systems. These simple models are called templates [4]. Two such templates are the point mass spring loaded inverted pendulum (point mass SLIP) and the rigid body SLIP. Both model sagittal plane locomotion by idealizing the combination of legs used by an animal in each stance to a single effective leg modeled by a spring and have been shown to accurately represent the motion of these animals [5], [6]. The obvious difference is that the point mass SLIP reduces the body to a point mass while the rigid body SLIP goes on to model sagittal rotation.

Extensive research has been done on control of the point mass SLIP. Because the model is unactuated, control is limited to variations in parameters. The most popular parameter to vary has been the leg touchdown angle. Although some control schemes do not directly specify a leg touchdown angle, almost all affect the leg touchdown angle in a way that increases stability. The simplest control scheme is the fixed leg angle reset policy where the touchdown angle is held fixed relative to the inertial frame at each period. This scheme has been shown to have a small region of stability [7],[8]. With model parameters taken from *Blaberus descoidalis* however, this region becomes extremely small [9] indicating that this would be a poor control scheme for robots of this morphology.

An increased region of stability was found using swing leg retraction [10] where the leg is swung toward the ground at a constant angular velocity starting at the apex of the flight phase. This allows the system to counteract disturbances in the touchdown state better than a fixed touchdown angle policy.

Similar methods use prescribed motion to increase stability via open loop control [11], [12]. The advantage of these schemes is that all leg angles are relative to the body so the only sensors needed for implementation are clocks and well-tuned servos. Because of their practicality these schemes have been implemented with great success on the robot, RHex [13].

Neither fixed angle reset, swing leg retraction, nor prescribed motion take advantage of the system's previous behavior to direct its future behavior. Adaptive control schemes have be developed to take advantage of this information [9],[14]. These control schemes rely on the previous leg lift off and touchdown angle to choose its next touchdown angle. They greatly improve the region of stability found for the fixed angle reset policy while requiring little knowledge of the desired gait. Since the input parameter is the desired touchdown angle it is easy to switch between gaits.

Many of the control schemes developed for the point mass case have been extended into rigid body SLIP model with limited success. Partially asymptotically stable gaits having three Floquet multipliers of unity magnitude have been found for the fixed angle reset policy, although the number of such gaits is apparently very small and dependent on the system parameters used in simulation [15]. In addition the prescribed motion scheme developed for RHex has exhibited partially asymptotically stable gaits in certain parameter regions [16].

All the schemes presented thus far use an understanding of the behavior of the system to determine criterion for a leg placement protocol, but none use the model itself in the protocol. The work of Mombaur et al. [17] uses knowledge of the model and numerical optimization to predict optimally stable open-loop gaits for a 4 DOF monopod and a 5 DOF biped. We apply model predictive control to the point mass and rigid body SLIP using the model and the state at lift off to predict the optimal leg touchdown angle to drive the system to a periodic orbit. The feed-back at lift off assists the control system in recovering the system from otherwise catastrophic perturbations. Because control is applied once per stride, this scheme will be well suited for implementation on microcontrollers which operate in a discrete fashion.

The work is structured as follows. In section 2, we review some mathematical concepts that are crucial to development of the control scheme include unconstrained (2.1) and constrained optimization (2.2), distinction between variations and differentials (2.3), Leibnitz' Rule (2.4), Collocation (2.5), Newton-Raphson Routines (2.6), and Poincaré Maps (2.7). In section 3, we

describe the rigid body SLIP model developing its equations of motion for its flight (3.1) and ground phases (3.2). In section 4, we outline the solution, by developing the criterion for an optimal trajectory (4.1), based on the penalization of undesired end states (4.1.1) and the trajectory's adherence to some physical constraints (4.1.2). This constrained penalization function is then differentiated and analyzed to yield the optimal criterion (4.1.3). These are then distilled into boundary conditions for a boundary value ODE (4.1.4). Collocation is employed to turn the boundary value problem into a system on nonlinear equations which must be driven to zero (4.2). A scheme for developing an acceptable guess at the initial conditions of the system of nonlinear equations is developed (4.3). The idea is that the system of nonlinear equations is solved between the lift off and touchdown events to determine an optimal touchdown angle. In section 5 the control scheme is tested numerically for the point mass case (5.1), where the Floquet multipliers of different gait families are determined both with the model predictive control scheme and the fixed leg touchdown angle (5.1.1). The systems ability to return from an energy conservative perturbation is also tested for the nominal gait family (5.1.2). The control scheme is also tested against the fixed angle reset policy in the rigid body case to a lesser extent (5.2). At last, in section 6 we summarize the work and suggest further studies.

2 MATHEMATICAL BACKGROUND

This section reviews and summarizes mathematical concepts that are critical to the development of the research presented in this work. Readers familiar with the material may bypass this section, or refer to it as needed.

2.1 Optimization

Optimization is achieved through the creation of a performance index and its extremization. The performance index consists of a cost function that attains either a maximum or minimum value when a desired result occurs [18]. While many cost functions consist of quadratic forms balancing the weighting of state variables and control parameters, they may also include terms that penalize undesired behavior. Cost functions utilized in this work take the form,

$$J = \phi(t_s) + \int_{t_o}^{t_f} L(t,u)dt\,, \tag{2.1}$$

where ϕ is the performance index applied at a specific time, t_s, and L is the performance index with input, u, applied over an interval of time t_o to t_f. Finding cost function extrema requires equating the total differential of the cost function to zero. Points where the derivative of the cost function is zero represent either extrema, such as minimum or maximum values, or an inflection point of the function. Examining the second derivative of the function at the identified point determines whether the point is a minimum, maximum or an inflection point of the cost function. Minimum, maximum, and inflection points have second derivatives that are positive, negative and zero, respectively. In some cases it is easy to see at what kind of extremum the cost function is. In these cases, the often quite expensive computation of the second derivative is forgone.

2.2 Constrained Optimization

An optimization is often desired that is constrained to an equation or inequality that cannot be substituted into the performance index directly. These constraints can be limited to a specific time or applied over an interval.

2.2.1 Parameter Constraints

To illustrate how constraints are applied to an optimization problem, consider the cost function

$$J = \phi(x,y)\,, \tag{2.2}$$

which is to be minimized constrained to,

$$\psi(x,y) = 0 \tag{2.3}$$

where x and y are optimization parameters and ϕ is continuous in x and y. Consider Figure 2.1.

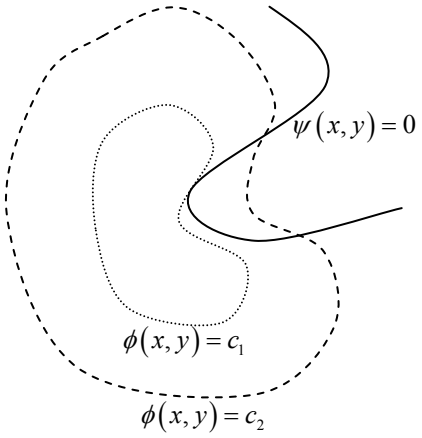

Figure 2.1 Illustration of constrained minimization. The dotted line is the contour of $\phi(x,y)=c_1$. The dashed line is the contour of $\phi(x,y)=c_2$. The solid line is the function $\psi(x,y)=0$.

Let us suppose that ϕ has no local extrema, only a global minimum somewhere within the dotted contour. If at an intersection of a contour of ϕ and ψ, the two functions are not tangent, then movement in the correct direction along ψ will decrease the cost, J. However, if the two are tangent, then movement in any direction along ψ will result in an increase in J. It follows that the minimum of J constrained to ψ is where ϕ and ψ are tangent [18]. That is, their gradients must be parallel. This is expressed as

$$\nabla\left(\phi(x,y)\right)=\nu\nabla\left(\psi(x,y)\right), \tag{2.4}$$

where ν is a constant, denoted as a Lagrange multiplier, which scales the magnitude of $\nabla\psi$ to match the magnitude of $\nabla\phi$. Rearranging eq. (2.4) and absorbing a negative sign into ν yields,

$$\nabla\left(\phi(x,y)+\nu\psi(x,y)\right)=\vec{0}. \tag{2.5}$$

Taking the gradient yields,

$$\begin{Bmatrix}\phi_x+\nu\psi_x\\ \phi_y+\nu\psi_y\\ \psi\end{Bmatrix}=\begin{Bmatrix}0\\0\\0\end{Bmatrix}. \tag{2.6}$$

Creating a constrained cost function J', which equals the quantity inside the gradient operator in eq. (2.5), is equivalent to adding zero to the original unconstrained cost function, J, since ψ is defined to equal zero when the constraint is satisfied. Evaluating the total differential of J', as is done to find the minimum in unconstrained optimization, results in

$$dJ' = \left(\phi_x + \nu\psi_x\right)dx + \left(\phi_y + \nu\psi_y\right)dy + \left(\psi\right)d\nu = 0\,. \tag{2.7}$$

Since dx, dy, and $d\nu$ are arbitrary, the partial derivatives must each independently equal zero [18], resulting in the same equations as presented previously in eq. (2.6). It can be concluded then, that the addition of a constraint to the optimization of a system, is as simple as adding the constraint, in the $\psi = 0$ form, scaled by a Lagrange multiplier, to the unconstrained cost function. That is,

$$J' = J + \nu\psi\,. \tag{2.8}$$

The optimization therefore proceeds as before, by taking the derivative of the new cost function and setting it equal to zero. The time specific part of the cost function, including all the time specific constraints, is sometimes denoted as, G.

2.2.2 Differential Constraints

Differential constraints can be applied to the system in much the same way as parameter constraints. Instead of a constant Lagrange multiplier to scale the gradients to match magnitudes, the constraint is multiplied by a continuous function which varies with time. This function is called a costate. For example in,

$$J = \int_{t_o}^{t_f} L\left(t, y, u\right)dt\,, \tag{2.9}$$

subject to

$$\dot{y} = f\left(t, y, u\right), \tag{2.10}$$

over t_o to t_f, the constrained cost function would be,

$$J' = \int_{t_o}^{t_f} \left[L(t, y, u) + \lambda^T \left(f(t, y) - \dot{y} \right) \right] dt \,, \tag{2.11}$$

where y is a column vector of states, u is a column vector of inputs, and λ is a column vector of costates. For ease of differentiation eq. (2.11) is rearranged to yield

$$J' = \int_{t_o}^{t_f} \left[H(t, y, u, \lambda) - \lambda \dot{y} \right] dt \,, \tag{2.12}$$

where H is called the Hamiltonian and takes the form,

$$H = L(t, y, u) + \lambda^T f(t, y) \,. \tag{2.13}$$

Because L, f, and $\dot{y}$ continuously change with time, so must λ. A general cost function with a time-specific portion as well as an integral portion is usually written in the form,

$$J = G(t_f, y_f) + \int_{t_o}^{t_f} \left[H(t, y, u, \lambda) - \lambda \dot{y} \right] dt \,, \tag{2.14}$$

where G is the time-specific constrained cost and H is the Hamiltonian introduced previously.

Note, the costs and constraints that comprise G need not be applied at the final time or even at a common time. The integral costs and constraints may be applied at any time interval and additional integrals can be added to apply costs and constraints over multiple time intervals. The costs and constraints applied over one interval need not be the same as those applied over another.

2.2.3 Inequality Constraints

It is often necessary to bound optimization parameters to a certain region. This is done using inequality constraints. Inequality constraints require adding an extra parameter to the system. If we desire,

$$x \geq c \,, \tag{2.15}$$

where x is a parameter, input, or state, and c is a constant, then the constraint is,

$$\psi(x,a) = x - a^2 - c\,, \tag{2.16}$$

where a is the extra parameter [18]. Since

$$a^2 \geq 0\,, \tag{2.17}$$

$$x - c \geq 0\,, \tag{2.18}$$

and

$$x \geq c\,, \tag{2.19}$$

therefore bounding x to a specific region, as desired.

2.3 Variation vs. Differential

A variation is a differential taken at a fixed time. Referring to Figure 2.2, if x_f represents a point on the optimal path $x(t)$ where an event, $\psi(x) = 0$, has occurred and x_{*f} is a point on an infinitesimally close neighboring path $x_*(t)$ where the same event has occurred, then dx is the difference between x_{*f} and x_f. However, if $\tilde{x}_{*f}$ represents the point on $x_*(t)$ that occurs at the same time as x_f, then δx is the difference between x_{*f} and $\tilde{x}_f$ because time is fixed [18].

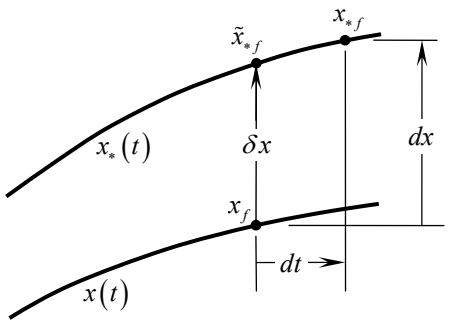

Figure 2.2 The optimal path, $x(t)$, and neighboring path, $x_*(t)$. Paths are infinitesimally close together. The points x_{*f} occurs at a value dx greater than x_f and an infinitesimally small time, dt, after x_f while the point $\tilde{x}_{*f}$ occurs at a value δx greater than x_f but at the same time.

Let us assume now that x_f occurs at time, t_f. Then δx can be expressed,

$$\delta x = x_*\left(t_f\right) - x\left(t_f\right), \tag{2.20}$$

and dx can be expressed

$$dx = x_*\left(t_f + dt\right) - x\left(t_f\right). \tag{2.21}$$

Since dt is infinitesimally small, $x_*\left(t_f + dt\right)$ can be expressed as a first order Taylor series about t_f. Substituting this into eq. (2.21) yields,

$$dx = x_*\left(t_f\right) + \frac{dx}{dt}dt - x\left(t_f\right). \tag{2.22}$$

Substituting in eq. (2.20), eq. (2.22) becomes

$$dx = \delta x + \frac{dx}{dt}dt, \tag{2.23}$$

and we obtain a relationship between differentials and variations.

2.4 Leibnitz' Rule

Since many cost functions contain integrals and the minimization of a cost function requires the evaluation of its derivative, it is often necessary to evaluate the derivative of an integral. For fixed limits the derivative of an integral is simply the integral of the derivative. If

$$I = \int_{t_o}^{t_f} F\left(t, y(t)\right)dt \tag{2.24}$$

then

$$dI = \int_{t_o}^{t_f} \delta F dt, \tag{2.25}$$

for t_o and t_f constant. Note that derivatives taken inside an integral are taken with time fixed and therefore represent variations. Remember an integral is a continuous sum of the integrand evaluated at every time between and including the limits. Even though the limits may not be fixed the individual times at which the integrand is evaluated are.

If the limits are not fixed an extra term must be added to the derivative. The differential of the integral becomes,

$$dI = \left[Fdt \right]_{t_o}^{t_f} + \int_{t_o}^{t_f} \delta F dt \, . \tag{2.26}$$

This is Leibnitz' rule [18].

2.5 Collocation

Collocation is a numerical method for solving boundary value problems. It is carried out by breaking the full time interval into N segments and approximating each state over each time segment as a sum of linearly independent trial functions [19]. See Figure 2.3.

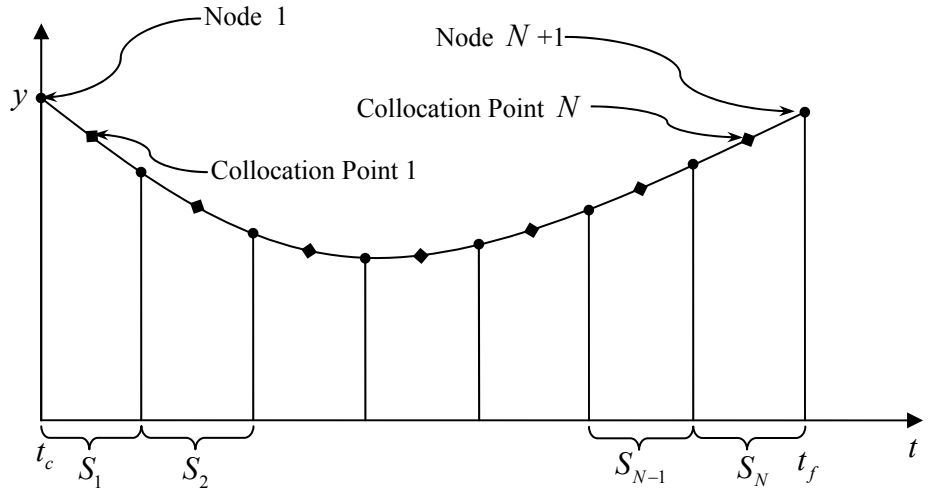

Figure 2.3 Collocation segment break up. State y from t_c to t_f broken into N segments, S_i, with $N+1$ nodes and N collocation points.

In this work a cubic polynomial was used for the sole trial function for each state over each segment because it was the lowest order polynomial that allowed us to enforce constraints at the endpoints through substitution [20]. For the rest of this work we will employ a trial function of this nature. While collocation points may be selected anywhere inside each time segment, we opted to utilize collocation points in the center of the segment in this work. Matching the function with the approximation substituted into it to the derivative of the approximation at these points ensures that the polynomial approximation accurately represents the solution. The resulting cubic polynomial takes the form,

$$y^j(t) = {}^0C^j + {}^1C^j t + {}^2C^j t^2 + {}^3C^j t^3, \tag{2.27}$$

where ${}^iC^j$ is the coefficient in the j^{th} polynomial approximation corresponding with t^i for $i = 0$ to 3. To ensure that the polynomial accurately approximates the real solution, we choose the coefficients such that the resulting polynomials satisfy some conditions that the real solution must also satisfy:

1. The endpoints of each polynomial must coincide with the endpoint of any adjacent polynomial.
2. The polynomials must obey the differential constraints at the end points.
3. The polynomials must obey the differential constraints at the collocation point.

We can use these conditions to write equations that will help us determine the coefficients of our polynomial.

2.5.1 Differential Constraints at the Collocation Point

We can use the polynomial to approximate the state of the system at the collocation points in the middle of each segment. If our polynomial approximation yields the correct state, we should be able to substitute it into the differential constraints and obtain the derivatives of our approximation polynomials [20]. That is,

$$\dot{y}^j_{c_j(i)} = f_{(i)}\left(y^j_{c_j(1)},...,y^j_{c_j(M)}\right) \quad \text{for } i = 1 \text{ to } M \text{ and } j = 1 \text{ to } N\,, \tag{2.28}$$

where $f_{(i)}(x)$ is the differential constraint function for state i, $y^j_{c_j(i)}$ is the j^{th} polynomial approximation of the i^{th} state at the j^{th} collocation point, $\dot{y}^j_{c_j(i)}$ is the derivative of the j^{th} polynomial approximation of the i^{th} state at the j^{th} collocation point, M is the number of states, and N is the number of segments.

The derivative of the polynomial is given by,

$$\dot{y}^j_{(i)}(t) = {}^1C^j_i + 2\,{}^2C^j_i t + 3\,{}^3C^j_i t^2 \qquad \text{for } t_{n_j} \le t \le t_{n_{j+1}}\,, \tag{2.29}$$

with i and j on the same intervals as in eq. (2.28), where $\dot{y}^j_{(i)}$ is the derivative of the polynomial approximation of the i^{th} state in segment j, t_{n_j} is the starting time of the j^{th} segment, the time of the j^{th} node, and the ending time of the $j-1^{\text{th}}$ segment, and the coefficients are as labeled in Figure 2.4.

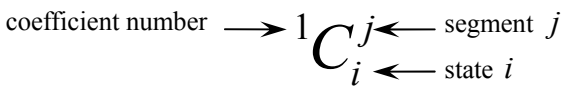

Figure 2.4 Polynomial coefficient labeling scheme.

Substituting eq. (2.29) into (2.28) yields,

$$\begin{aligned} &{}^1C^j_i + 2\,{}^2C^j_i t + 3\,{}^3C^j_i t^2 \\ &\qquad = f_{(i)}\left({}^0C^j_1 + {}^1C^j_1 t + {}^2C^j_1 t^2 + {}^3C^j_1 t^3,..., {}^0C^j_M + {}^1C^j_M t + {}^2C^j_M t^2 + {}^3C^j_M t^3\right), \end{aligned} \tag{2.30}$$

with i, j, and t on the same intervals as in eq. (2.29).

2.5.2 Connection Constraints

As mentioned earlier we would like to be able to constrain the value of polynomial and its derivative at the endpoints. To accomplish this, the polynomial and its derivative must be evaluated at the endpoints, yielding,

$$y^j_{n_j(i)} = {}^0C^j_i + {}^1C^j_i t_{n_j} + {}^2C^j_i {t_{n_j}}^2 + {}^3C^j_i {t_{n_j}}^3, \tag{2.31}$$

$$y^j_{n_{j+1}(i)} = {}^0C^j_i + {}^1C^j_i t_{n_{j+1}} + {}^2C^j_i {t_{n_{j+1}}}^2 + {}^3C^j_i {t_{n_{j+1}}}^3, \tag{2.32}$$

$$\dot{y}^j_{n_j(i)} = {}^1C^j_i + 2\,{}^2C^j_i t_{n_j} + 3\,{}^3C^j_i {t_{n_j}}^2, \tag{2.33}$$

and

$$\dot{y}^j_{n_{j+1}(i)} = {}^1C^j_i + 2\,{}^2C^j_i t_{n_{j+1}} + 3\,{}^3C^j_i {t_{n_{j+1}}}^2, \tag{2.34}$$

where $y^j_{n_j(i)}$ is the j^{th} polynomial approximation of the i^{th} state evaluated at the j^{th} node, $\dot{y}^j_{n_j(i)}$ is the derivative of the j^{th} polynomial approximation of the i^{th} state evaluated at the j^{th} node, and t_{n_j} is the time at the j^{th} node.

Rearranging equations (2.31), (2.32), (2.33), and (2.34) into a single vector equation yields

$$\begin{bmatrix} 1 & t_{n_j} & {t_{n_j}}^2 & {t_{n_j}}^3 \\ 0 & 1 & 2t_{n_j} & 3{t_{n_j}}^2 \\ 1 & t_{n_{j+1}} & {t_{n_{j+1}}}^2 & {t_{n_{j+1}}}^3 \\ 0 & 1 & 2t_{n_{j+1}} & 3{t_{n_{j+1}}}^2 \end{bmatrix} \begin{Bmatrix} {}^0C^j_i \\ {}^1C^j_i \\ {}^2C^j_i \\ {}^3C^j_i \end{Bmatrix} = \begin{Bmatrix} y^j_{n_j(i)} \\ \dot{y}^j_{n_j(i)} \\ y^j_{n_{j+1}(i)} \\ \dot{y}^j_{n_{j+1}(i)} \end{Bmatrix}, \tag{2.35}$$

where the square matrix is called the time matrix whose inverse is denoted by $[K]$ [20]. Multiplying both sides of eq. (2.35) by $[K]$ results in,

$$\begin{Bmatrix} {}^0C^j_i \\ {}^1C^j_i \\ {}^2C^j_i \\ {}^3C^j_i \end{Bmatrix} = [K] \begin{Bmatrix} y^j_{n_j(i)} \\ \dot{y}^j_{n_j(i)} \\ y^j_{n_{j+1}(i)} \\ \dot{y}^j_{n_{j+1}(i)} \end{Bmatrix}, \tag{2.36}$$

an equation for the coefficients of the approximation polynomial in terms of the end states and their derivatives.

2.5.3 Change of Variables

If the final and initial times are free to change based on geometric events, it follows that the time matrix will also change. To simplify computation, we desire known, fixed values for the time matrix such that K remains invariant with respect to initial or final time changes. This can be achieved through a change in variables. Let us assume a form for the integral part of the derivative of our cost function as

$$\int_{t_o}^{t_f}\left[\left(H_y+\dot{\lambda}^T\right)\delta y+\left(f^T-\dot{y}^T\right)\delta\lambda\right]dt\,, \tag{2.37}$$

where δy is the variation of the state vector, y, $\delta\lambda$ is the variation of the costate vector, λ, $\dot{y}$ is the time derivative of y, $\dot{\lambda}$ is the time derivative of λ, H_y is the differential constraint function for $\dot{\lambda}$, and f is the differential constraint function for y. Since no part of this equation depends explicitly on time, the starting time of the integral is immaterial as long as it is over the same amount of time. So let us define an intermediate time t_n such that,

$$t_n = t_f - t_o\,. \tag{2.38}$$

Then eq. (2.37) becomes,

$$\int_{0}^{t_n}\left[\left(H_y+\dot{\lambda}^T\right)\delta y+\left(f^T-\dot{y}^T\right)\delta\lambda\right]dt\,. \tag{2.39}$$

Let us now define a nondimensionalized time, τ, which is scaled by t_n so that when t is zero, τ is zero, and when t is t_n, τ is 1. This leads to

$$\frac{\tau}{1-0}=\frac{t}{t_n-0}\,. \tag{2.40}$$

Rearranging we obtain

$$t = t_n\tau\,. \tag{2.41}$$

Differentiating eq. (2.41) yields

$$dt = t_n d\tau \,. \tag{2.42}$$

Substituting eq. (2.42) into eq. (2.39) for dt yields,

$$\int_0^1 \left[\left(H_y + \left(\frac{d\lambda}{t_n d\tau} \right)^T \right) \delta y + \left(f^T - \left(\frac{dy}{t_n d\tau} \right)^T \right) \delta\lambda \right] t_n d\tau. \tag{2.43}$$

Distributing we get,

$$\int_0^1 \left[\left(t_n H_y + \left(\frac{d\lambda}{d\tau} \right)^T \right) \delta y + \left(t_n f^T - \left(\frac{dy}{d\tau} \right)^T \right) \delta\lambda \right] d\tau. \tag{2.44}$$

Since we are integrating with respect to τ, we will still obtain values for y and λ, as desired [18]. We define the following quantities for simplicity:

$$\tilde{H}_y = t_n H_y = \lambda^T \tilde{f}_y \,, \tag{2.45}$$

$$\tilde{f} = t_n f \,, \tag{2.46}$$

$$\lambda' = \frac{d\lambda}{d\tau} \,, \tag{2.47}$$

and

$$y' = \frac{dy}{d\tau} \,. \tag{2.48}$$

Our new Euler equations become

$$\lambda' = -\tilde{H}_y \tag{2.49}$$

and

$$y' = \tilde{f}\,. \tag{2.50}$$

Computational simplification would also arise if we did not have to find a unique time matrix inverse for every time segment. Since the polynomials approximate values within the time segment only, it does not matter at what time the segment started, it only matters how much time has passed since the segment began. If we set τ to start at zero at every segment beginning and we restrict the segments to be equally spaced, then our time and K matrix will not change between the segments. Such a protocol would yield

$$\tau_{n_j} = 0 \tag{2.51}$$

and

$$\tau_{n_{j+1}} = \frac{1}{N}\,. \tag{2.52}$$

Substituting eq. (2.41) into (2.27) yields

$$y_{(i)}^{j}(\tau) = {}^{0}C_i^j + {}^{1}C_i^j t_n \tau + {}^{2}C_i^j t_n{}^{2}\tau^2 + {}^{3}C_i^j t_n{}^{3}\tau^3 \tag{2.53}$$

Since t_n does not depend on time, its factors can be absorbed into the constants, yielding,

$$y_{(i)}^{j}(\tau) = {}^{0}C_i^j + {}^{1}\tilde{C}_i^j \tau + {}^{2}\tilde{C}_i^j \tau^2 + {}^{3}\tilde{C}_i^j \tau^3\,. \tag{2.54}$$

Taking the derivative with respect to τ gives,

$$y'^{j}_{(i)}(\tau) = {}^{1}\tilde{C}_i^j + 2\,{}^{2}\tilde{C}_i^j \tau + 3\,{}^{3}\tilde{C}_i^j \tau \tag{2.55}$$

The vector equation therefore becomes,

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & \frac{1}{N} & \frac{1}{N^2} & \frac{1}{N^3} \\ 0 & 1 & 2\frac{1}{N} & 3\frac{1}{N^2} \end{bmatrix} \begin{Bmatrix} {}^0C_i^j \\ {}^1\tilde{C}_i^j \\ {}^2\tilde{C}_i^j \\ {}^3\tilde{C}_i^j \end{Bmatrix} = \begin{Bmatrix} y_{n_j(i)}^j \\ y'^j_{n_j(i)} \\ y_{n_{j+1}(i)}^j \\ y'^j_{n_{j+1}(i)} \end{Bmatrix}. \tag{2.56}$$

Inverting our new time matrix yields,

$$\begin{Bmatrix} {}^0C_i^j \\ {}^1\tilde{C}_i^j \\ {}^2\tilde{C}_i^j \\ {}^3\tilde{C}_i^j \end{Bmatrix} = \left[\tilde{K}\right] \begin{Bmatrix} y_{n_j(i)}^j \\ y'^j_{n_j(i)} \\ y_{n_{j+1}(i)}^j \\ y'^j_{n_{j+1}(i)} \end{Bmatrix}, \tag{2.57}$$

where $\tilde{K}$ is the inverse of the nondimensionalized time matrix. Equation (2.54) can now be written in vector form as,

$$y_{(i)}^j(\tau) = \begin{bmatrix} 1 & \tau & \tau^2 & \tau^3 \end{bmatrix} \begin{Bmatrix} {}^0C_i^j \\ {}^1\tilde{C}_i^j \\ {}^2\tilde{C}_i^j \\ {}^3\tilde{C}_i^j \end{Bmatrix} = \begin{bmatrix} 1 & \tau & \tau^2 & \tau^3 \end{bmatrix} \left[\tilde{K}\right] \begin{Bmatrix} y_{n_j(i)}^j \\ y'^j_{n_j(i)} \\ y_{n_{j+1}(i)}^j \\ y'^j_{n_{j+1}(i)} \end{Bmatrix}. \tag{2.58}$$

We can simply specify a state, $y_{n_j(i)}$, that serves as the end state of the $j-1^{\text{th}}$ segment and the beginning state of the j^{th} segment. Our differential constraints should hold not only at the collocation point but at the nodes too. So we can replace the derivatives of the states with the differential constraints shown in eqs. (2.49) and (2.50) to yield

$$y_{(i)}^j(\tau) = \begin{bmatrix} 1 & \tau & \tau^2 & \tau^3 \end{bmatrix} \left[\tilde{K}\right] \begin{Bmatrix} y_{n_j(i)} \\ \tilde{f}_{(i)}\left(y_{n_j}, t_n\right) \\ y_{n_{j+1}(i)} \\ \tilde{f}_{(i)}\left(y_{n_{j+1}}, t_n\right) \end{Bmatrix}, \tag{2.59}$$

and

$$\lambda_{(i)}^{j}(\tau)=\begin{bmatrix}1 & \tau & \tau^2 & \tau^3\end{bmatrix}\left[\tilde{K}\right]\begin{Bmatrix}\lambda_{n_j(i)}\\ -\tilde{H}_{y(i)}\left(y_{n_j},\lambda_{n_j},t_n\right)\\ \lambda_{n_{j+1}(i)}\\ -\tilde{H}_{y(i)}\left(y_{n_{j+1}},\lambda_{n_{j+1}},t_n\right)\end{Bmatrix}. \tag{2.60}$$

2.5.4 Differential Match With New Polynomials

To satisfy the differential constraint at the collocation point developed in eq. (2.28), we must obtain the derivative of the polynomials in eqs. (2.59) and (2.60). Note, even though eq. (2.28) was only developed for the states, because the costates must follow similar differential constraints, a similar equation can be used. It is,

$$\dot{\lambda}_{c_j(i)}^{j}=-H_{y(i)}\left(y_{c_j}^{j},\lambda_{c_j}\right)\quad \text{for } j=1 \text{ to } N. \tag{2.61}$$

In both instances the derivative with respect to τ is being used instead of t. Because of this $f(y)$ is replaced with $\tilde{f}(y,t_n)$ and $-H_y(y,\lambda)$ is replaced with $-\tilde{H}_y(y,\lambda,t_n)$. The derivatives are

$$y_{(i)}^{\prime j}(\tau)=\begin{bmatrix}0 & 1 & 2\tau & 3\tau^2\end{bmatrix}\left[\tilde{K}\right]\begin{Bmatrix}y_{n_j(i)}\\ \tilde{f}_{(i)}\left(y_{n_j},t_n\right)\\ y_{n_{j+1}(i)}\\ \tilde{f}_{(i)}\left(y_{n_{j+1}},t_n\right)\end{Bmatrix}, \tag{2.62}$$

and

$$\lambda_{(i)}^{\prime j}(\tau)=\begin{bmatrix}0 & 1 & 2\tau & 3\tau^2\end{bmatrix}\left[\tilde{K}\right]\begin{Bmatrix}\lambda_{n_j(i)}\\ -\tilde{H}_{y(i)}\left(y_{n_j},\lambda_{n_j},t_n\right)\\ \lambda_{n_{j+1}(i)}\\ -\tilde{H}_{y(i)}\left(y_{n_{j+1}},\lambda_{n_{j+1}},t_n\right)\end{Bmatrix}. \tag{2.63}$$

These derivatives are evaluated at the collocation points. Tau can be thought of as the percentage of the ground phase that has been completed. So when τ is 1, 100% of the ground phase is complete. However inside each segment τ starts over at zero and ends at what ever percent of the total ground phase that segment represents. Since there are N segments, τ starts at zero and ends the segment at $\frac{1}{N}$. In the middle of the segment where the collocation point is, τ would be $\frac{1}{2N}$. Substituting this into the polynomial derivatives yields,

$$y'_{c_j(i)} = \begin{bmatrix} 0 & 1 & \frac{1}{N} & \frac{3}{4N^2} \end{bmatrix} \begin{bmatrix} \tilde{K} \end{bmatrix} \begin{Bmatrix} y_{n_j(i)} \\ \tilde{f}_{(i)}\left(y_{n_j}, t_n\right) \\ y_{n_{j+1}(i)} \\ \tilde{f}_{(i)}\left(y_{n_{j+1}}, t_n\right) \end{Bmatrix}, \tag{2.64}$$

and

$$\lambda'_{c_j(i)} = \begin{bmatrix} 0 & 1 & \frac{1}{N} & \frac{3}{4N^2} \end{bmatrix} \begin{bmatrix} \tilde{K} \end{bmatrix} \begin{Bmatrix} \lambda_{n_j(i)} \\ -\tilde{H}_{y(i)}\left(y_{n_j}, \lambda_{n_j}, t_n\right) \\ \lambda_{n_{j+1}(i)} \\ -\tilde{H}_{y(i)}\left(y_{n_{j+1}}, \lambda_{n_{j+1}}, t_n\right) \end{Bmatrix} \tag{2.65}$$

for $i = 1$ to 6. The polynomial approximations of the states themselves at the collocation points must also be evaluated. This leads to,

$$y_{c_j(i)} = \begin{bmatrix} 1 & \frac{1}{2N} & \frac{1}{4N^2} & \frac{1}{8N^3} \end{bmatrix} \begin{bmatrix} \tilde{K} \end{bmatrix} \begin{Bmatrix} y_{n_j(i)} \\ \tilde{f}_{(i)}\left(y_{n_j}, t_n\right) \\ y_{n_{j+1}(i)} \\ \tilde{f}_{(i)}\left(y_{n_{j+1}}, t_n\right) \end{Bmatrix}, \tag{2.66}$$

and

$$\lambda_{c_j(i)}(\tau)=\begin{bmatrix}1 & \frac{1}{2N} & \frac{1}{4N^2} & \frac{1}{8N^3}\end{bmatrix}\left[\tilde{K}\right]\begin{Bmatrix}\lambda_{n_j(i)}\\ -\tilde{H}_{y(i)}\left(y_{n_j},\lambda_{n_j},t_n\right)\\ \lambda_{n_{j+1}(i)}\\ -\tilde{H}_{y(i)}\left(y_{n_{j+1}},\lambda_{n_{j+1}},t_n\right)\end{Bmatrix}, \tag{2.67}$$

for $i=1$ to 6. We substitute these values into $\tilde{f}\left(y,t_n\right)$ and $-\tilde{H}_y\left(y,\lambda,t_n\right)$ and set them equal to the polynomial derivative eqs. (2.64) and (2.65). This quantity will go to zero as the polynomial approximations approach the actual solution.

2.6 Newton-Raphson Solver

A typical residual is composed of partial derivatives of the cost function with respect to every variable parameter. Therefore it is the gradient of the cost function. Because the differential constraints are enforced using collocation, the components of the residual vector enforcing the differential constraints are not partial derivatives of the cost function. As a result, the residual is not a pure gradient. In driving the residual to zero, the gradient of each of the residuals must be determined, requiring the determination of the second partial derivatives of the cost function with respect to the every variable parameter. Organizing these gradients into a matrix produces a resultant matrix referred to as the Hessian. Because in this implementation the residual is not exactly the gradient of the cost function, the gradient of the residual vector is not exactly the Hessian of the cost function.

The objective is to drive the elements of the residual, $R_{(k)}$, to zero by correctly selecting the variables on which the residual is dependent , X .

$$R_k(\mathrm{X})=0, \tag{2.68}$$

for $k=1$ to the size of the residual. To do this an initial guess X^o is chosen and refined based on criteria that will be developed shortly. First the X vector must be defined. In the Newton-Raphson routine, the term state vector refers to the state of the routine. That is it refers to all the parameters, states at the nodes, and costates at the nodes assembled into a vector.

By approximating R_k as a truncated Taylor series we find,

$$R_k\left(\mathrm{X}+\delta\mathrm{X}\right)=R_k\left(\mathrm{X}\right)+\sum_{l=1}^{P}\frac{\partial R_k}{\partial \mathrm{X}_l}\delta\mathrm{X}_l\,. \tag{2.69}$$

The goal is to force $R_k\left(\mathrm{X}+\delta\mathrm{X}\right)$ to zero by picking the correct $\delta\mathrm{X}$. If we assume that eq. (2.69) is a good approximation for $R_k\left(\mathrm{X}+\delta\mathrm{X}\right)$, then substituting zero for $R_k\left(\mathrm{X}+\delta\mathrm{X}\right)$ should yield an equation which can be solved for $\delta\mathrm{X}$ [19]. That is,

$$\sum_{l=1}^{P}\frac{\partial R_k}{\partial \mathrm{X}_l}\delta\mathrm{X}_l=-R_k\left(\mathrm{X}\right). \tag{2.70}$$

The sum can be written,

$$\begin{bmatrix}\dfrac{\partial R_k}{\partial \mathrm{X}_1} & \dfrac{\partial R_k}{\partial \mathrm{X}_2} & \cdots & \dfrac{\partial R_k}{\partial \mathrm{X}_P}\end{bmatrix}\begin{Bmatrix}\delta\mathrm{X}_1\\ \delta\mathrm{X}_2\\ \vdots\\ \delta\mathrm{X}_P\end{Bmatrix}=-R_k\left(\mathrm{X}\right). \tag{2.71}$$

This equation can be written for $k=1$ to size of the residual by concatenating partial derivative row vectors such that,

$$\begin{bmatrix}\dfrac{\partial R_1}{\partial \mathrm{X}_1} & \dfrac{\partial R_1}{\partial \mathrm{X}_2} & \cdots & \dfrac{\partial R_1}{\partial \mathrm{X}_P}\\ \dfrac{\partial R_2}{\partial \mathrm{X}_1} & \dfrac{\partial R_2}{\partial \mathrm{X}_2} & \cdots & \dfrac{\partial R_2}{\partial \mathrm{X}_P}\\ \vdots & \vdots & \ddots & \vdots\\ \dfrac{\partial R_P}{\partial \mathrm{X}_1} & \dfrac{\partial R_P}{\partial \mathrm{X}_2} & \cdots & \dfrac{\partial R_P}{\partial \mathrm{X}_P}\end{bmatrix}\begin{Bmatrix}\delta\mathrm{X}_1\\ \delta\mathrm{X}_2\\ \vdots\\ \delta\mathrm{X}_P\end{Bmatrix}=-\begin{Bmatrix}R_1\left(\mathrm{X}\right)\\ R_2\left(\mathrm{X}\right)\\ \vdots\\ R_P\left(\mathrm{X}\right)\end{Bmatrix}. \tag{2.72}$$

The matrix of partial derivatives in eq. (2.72) is the Hessian of the cost function, when the residual is the gradient but is often referred to as the Hessian in other instances as well. Inverting this matrix numerically yields a relationship for $\delta\mathrm{X}$. In those instances where the Hessian is singular, the pseudo-inverse is taken instead. A new guess, X_{new}, is formed by adding $\delta\mathrm{X}$ to the old guess.

$$\mathrm{X}_{new}=\mathrm{X}_{old}+\delta\mathrm{X} \tag{2.73}$$

This is repeated until $R(\mathrm{X})$ is zero.

2.7 Poincaré Map

If $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ is an n-dimentional system, S is an $n-1$ dimensional surface of section, and all trajectories starting at S flow through it, then the Poincaré Map P is the mapping from one of the trajectories intersections of S to the next. Let $\mathbf{y}_k$ denote the k^{th} intersection of S. The Poincaré map is then defined as

$$\mathbf{y}_{k+1} = \mathbf{P}(\mathbf{y}_k). \tag{2.74}$$

For a specific point, $\mathbf{y}^*$, if

$$\mathbf{y}^* = \mathbf{P}(\mathbf{y}^*), \tag{2.75}$$

then $\mathbf{y}^*$ is a fixed point of $\mathbf{P}$. A trajectory starting at $\mathbf{y}^*$ will end up at $\mathbf{y}^*$ in a finite amount of time. This is a closed orbit of the system $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$. The stability of the closed orbit can be determined by examining the behavior of the system in a region around $\mathbf{y}^*$.

To determine the behavior of the system in the region about the fixed point, we perturb the fixed point with a vector resulting in

$$\mathbf{y}^* + \mathbf{v}_1 = \mathbf{P}(\mathbf{y}^* + \mathbf{v}_0), \tag{2.76}$$

where $\mathbf{v}_0$ is an $(n-1)$-dimentional vector. Expanding eq. (2.76) in a first order Taylor series expansion yields,

$$\mathbf{y}^* + \mathbf{v}_1 = \mathbf{P}(\mathbf{y}^*) + \left(D\mathbf{P}(\mathbf{y}^*)\right)\mathbf{v}_0 \tag{2.77}$$

for $\mathbf{v}_0$ of small magnitude where $D\mathbf{P}$ is an $(n-1)\,\mathrm{x}\,(n-1)$ matrix. Substituting eq. (2.75) into eq. (2.77) yields

$$\mathbf{v}_1 = \left(D\mathbf{P}(\mathbf{y}^*)\right)\mathbf{v}_0. \tag{2.78}$$

The stability of fixed point $\mathbf{y}^*$ is determined by the eigenvalues, λ_i, of $D\mathbf{P}$ [21]. The eigenvalues of this matrix are called the *Floquet multipliers* of the periodic orbit. Technically there is one extra unity Floquet multiplier associated with a perturbation directly along the periodic orbit. This multiplier is trivial since a perturbation along it would just amount to a translation in time. Because of this it is ignored.

The matrix $D\mathbf{P}$ can be determined by perturbing each state individually and using a difference formula to obtain the column vector of partial derivatives taken with respect to the individual state [19]. These vectors are then concatenated so they form $D\mathbf{P}$.

3 MODEL

The model considered in this work is illustrated in Figure 3.1. It consists of a rigid body of mass m and moment of inertia I_{yy}, with a spring attached at point A, a distance d above the center of mass in the negative $\vec{k}_B$ direction. The spring makes contact with the ground intermittently at its end point, labeled C.

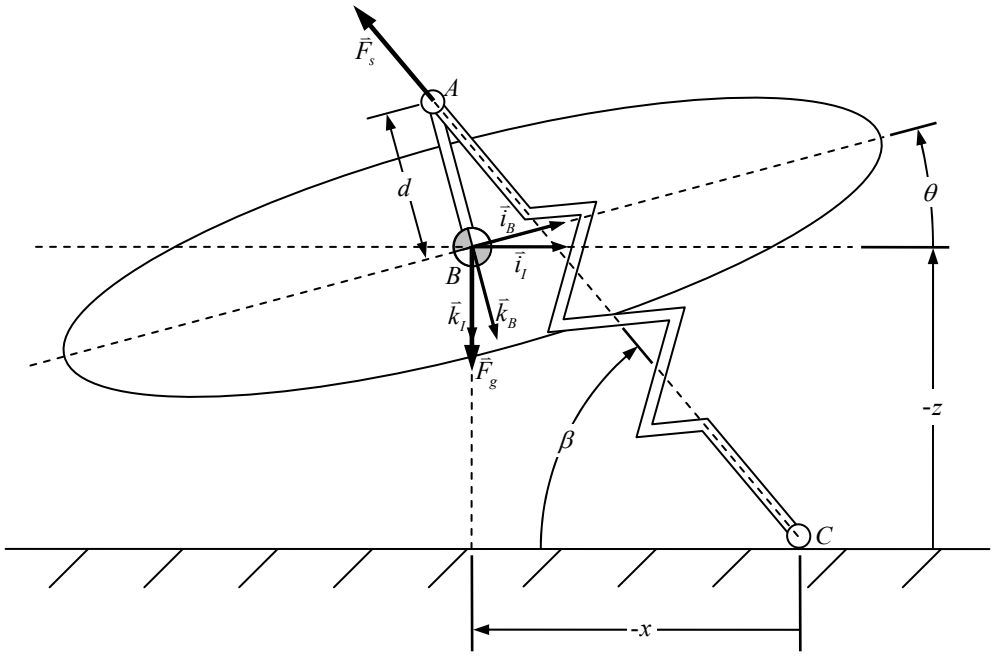

Figure 3.1 The system at spring touchdown.

Unit vectors $\vec{i}_I$, $\vec{j}_I$, and $\vec{k}_I$ are mutually perpendicular and inertially fixed with $\vec{i}_I$ pointing to the right, $\vec{j}_I$ pointing out of the page and $\vec{k}_I$ point directly downward. Unit vectors $\vec{i}_B$, $\vec{j}_B$, and $\vec{k}_B$ are also mutually perpendicular, but are fixed to the rigid body and aligned with its principal axes. Unit vectors $\vec{j}_I$ and $\vec{j}_B$ are aligned and $\vec{i}_B$ makes an angle, θ, with $\vec{i}_I$. The center of mass, is a distance $-z$ and $-x$, above the ground and to the left of the foot placement respectively. These coordinates are assigned in this way so z and x increase in the positive $\vec{k}_I$ and $\vec{i}_I$ directions respectively. The constant gravitational force, $\vec{F}_g$, acts in the $\vec{k}_I$ direction.

Because the governing equations of motion of the body change depending upon whether or not the leg is in contact with the ground, the model is a hybrid system. As a result, the equations of motion are defined in a piecewise manner with discrete events that determine when the reign of one set stops and the next starts. We will refer to these segments of continuity as phases. The events that switch between phases are spring lift off and touchdown.

The flight phase starts when the spring lifts off the ground and ends when it first makes contact again. During the flight phase, the spring remains an angle β from horizontal in the inertial frame. The only force acting on the body is the gravitational force, $\vec{F}_g$. The ground phase starts, with the spring undeflected, when the spring makes contact with the ground, and ends when the spring returns to its initial length and lifts off from the ground. At touchdown the spring remains at the angle, β, with respect to the horizontal but after the first instant, the angle changes as required by the dynamics of the system. During this phase, in addition to the gravitational force, a spring force, $\vec{F}_s$, is present.

3.1 Flight Phase Dynamics

In flight, the gravitational force is the only force on the body. It given by

$$\vec{F}_g = mg\vec{k}_I , \tag{3.1}$$

where g is the gravitational constant. Applying Newton's Second law yields

$$m\left(\ddot{x}\vec{i}_I + \ddot{z}\vec{k}_I\right) = mg\vec{k}_I , \tag{3.2}$$

where $\ddot{x}$ and $\ddot{z}$ are the second derivatives with respect to time of x and z. Canceling the m on both sides of the equation and dotting with $\vec{i}_I$ and $\vec{k}_I$ yields

$$\ddot{x} = 0\,, \tag{3.3}$$

and

$$\ddot{z} = g\,, \tag{3.4}$$

respectively.

Since there are no applied couples and gravity acts only on the center of mass, there is no angular acceleration. So

$$\ddot{\theta} = 0\,. \tag{3.5}$$

Organizing x, z, θ, and their derivatives in a vector yields

$$\boldsymbol{y} = \begin{Bmatrix} x \\ z \\ \theta \\ \dot{x} \\ \dot{z} \\ \dot{\theta} \end{Bmatrix}. \tag{3.6}$$

This is the state space representation of the system. We can take the derivative of eq. (3.6) and substitute in eqns. (3.3) through (3.6), to obtain

$$\dot{\boldsymbol{y}} = \begin{Bmatrix} y_{(4)} \\ y_{(5)} \\ y_{(6)} \\ 0 \\ g \\ 0 \end{Bmatrix}. \tag{3.7}$$

These are the first order equations of motion. Equation (3.7) can be integrated analytically resulting in,

$$\boldsymbol{y}(t) = \begin{Bmatrix} y_{o(1)} + y_{o(4)}t \\ y_{o(2)} + y_{o(5)}t + \frac{1}{2}gt^2 \\ y_{o(3)} + y_{o(6)}t \\ y_{o(4)} \\ y_{o(5)} + gt \\ y_{o(6)} \end{Bmatrix}, \tag{3.8}$$

where $y_{o(i)}$ is the initial value of $y_{(i)}$ for $i = 1$ to 6.

3.2 Ground Phase Dynamics

During the ground phase, the spring force, $\vec{F}_s$, acts on the body in addition to the gravitational force. The spring is linear so $\left|\vec{F}_s\right|$ will be proportional to the change in spring length. We will refer to the spring length as η and the uncompressed spring length as η_o. To obtain equations of motion utilizing $\vec{F}_s$, we must define η in terms of our generalized coordinates, x, z, and θ. Referring to Figure 3.1, since C and A are the endpoints of the spring, it follows that the length of the spring, η, is the magnitude of the position vector from C to A. That is,

$$\eta = \left|\vec{r}_{C\to A}\right|. \tag{3.9}$$

This position vector can be expressed as

$$\vec{r}_{C\to A} = \vec{r}_{C\to \bar{A}} + \vec{r}_{\bar{A}\to A}, \tag{3.10}$$

where $\vec{r}_{C\to \bar{A}}$ is the position vector from C to the center of mass and $\vec{r}_{\bar{A}\to A}$ is the position vector from the center of mass to A.

The generalized coordinate, x, is defined as the distance in the $\vec{i}_I$ direction from the leg touchdown point to the center of mass. The generalized coordinate, z, is defined as the distance in the $\vec{k}_I$ direction from the leg touchdown point to the center of mass. Since the vectors are being

defined for the ground phase we can assume the leg touchdown point and the spring end point, C, to be coincident. So the position vector from C to the center of mass is,

$$\vec{r}_{C\to\text{A}} = x\vec{i}_I + z\vec{k}_I \,. \tag{3.11}$$

As mentioned earlier, the spring attachment point, A, is held a fixed distance d above the center of mass. Above, in this case, means above relative to the body. So,

$$\vec{r}_{\text{A}\to A} = -d\vec{k}_B \,. \tag{3.12}$$

Converting this to the inertial reference frame so we can combine it with eq. (3.11) yields,

$$\vec{r}_{\text{A}\to A} = -d\vec{k}_B = -d\left(\sin(\theta)\vec{i}_I + \cos(\theta)\vec{k}_I\right). \tag{3.13}$$

Substituting eqs. (3.11) and (3.13) into (3.10), we obtain,

$$\vec{r}_{C\to A} = \left(x - d\sin(\theta)\right)\vec{i}_I + \left(z - d\cos(\theta)\right)\vec{k}_I \,. \tag{3.14}$$

The leg length is just the magnitude of this vector, which may be computed as

$$\eta = \sqrt{\left(x - d\sin(\theta)\right)^2 + \left(z - d\cos(\theta)\right)^2} \,. \tag{3.15}$$

We represent the leg with a linear spring such that the magnitude of the spring force is a function of the leg length and the spring constant, k. So,

$$\left|\vec{F}_s\right| = k\left(\eta_o - \eta\right). \tag{3.16}$$

Substituting eq. (3.15) into eq. (3.16) yields,

$$\left|\vec{F}_s\right| = k\left(\eta_o - \eta\right) = k\left(\eta_o - \sqrt{\left(x - d\sin(\theta)\right)^2 + \left(z - d\cos(\theta)\right)^2}\right). \tag{3.17}$$

The spring force always acts along the spring. That is $\vec{F}_s$ acts in the direction of the position vector from C to A, $\vec{r}_{C\to A}$. Multiplying the $\left|\vec{F}_s\right|$ with a unit vector in the direction of $\vec{r}_{C\to A}$ yields $\vec{F}_s$. Unitizing $\vec{r}_{C\to A}$ and multiplying by eq. (3.16) yields

$$\vec{F}_s = k\left(\eta_o - \eta\right)\left(\frac{\vec{r}_{C\to A}}{\left|\vec{r}_{C\to A}\right|}\right). \tag{3.18}$$

Substituting eqs. (3.14) and (3.9) in for the position vector and its magnitude respectively yields,

$$\vec{F}_s = k\left(\frac{\eta_o}{\eta} - 1\right)\left(\left(x - d\sin\left(\theta\right)\right)\vec{i}_I + \left(z - d\cos\left(\theta\right)\right)\vec{k}_I\right). \tag{3.19}$$

While η can be calculated in terms of the generalized coordinates with eq. (3.15), for simplicity, we refrain from utilizing this relationship in the next calculations. Using Newton's Second Law to obtain the equations of motion, we sum the two forces on the body, $\vec{F}_s$ and $\vec{F}_g$, and divide by the mass yielding

$$\ddot{x}\vec{i}_I + \ddot{z}\vec{k}_I = \frac{k}{m}\left(\frac{\eta_o}{\eta} - 1\right)\left(x - d\sin\left(\theta\right)\right)\vec{i}_I + \left(\frac{k}{m}\left(\frac{\eta_o}{\eta} - 1\right)\left(z - d\cos\left(\theta\right)\right) + g\right)\vec{k}_I \tag{3.20}$$

Evaluating the dot product of this expression with $\vec{i}_I$ and $\vec{k}_I$ yields

$$\ddot{x} = \frac{k}{m}\left(\frac{\eta_o}{\eta} - 1\right)\left(x - d\sin\left(\theta\right)\right) \tag{3.21}$$

and

$$\ddot{z} = \frac{k}{m}\left(\frac{\eta_o}{\eta} - 1\right)\left(z - d\cos\left(\theta\right)\right) + g \tag{3.22}$$

respectively. We identify the last equation of motion using the change in angular momentum. Since we have the moment of inertia about the center of mass, it would be easiest to sum the moments

about the center of mass as well. Since, spring force, $\vec{F}_s$, is the only force that is not applied at the center of mass and there are no applied couples, the total moment about the center of mass is,

$$\vec{M} = \vec{r}_{\mathrm{A}\to\mathrm{A}} \times \vec{F}_s \,. \tag{3.23}$$

Substituting eqs (3.13) and (3.19) into eqn. (3.23) yields,

$$\vec{M} = -dk\left(\frac{\eta_o}{\eta} - 1\right)\begin{bmatrix} 0 & -\cos(\theta) & 0 \\ \cos(\theta) & 0 & -\sin(\theta) \\ 0 & \sin(\theta) & 0 \end{bmatrix}\begin{Bmatrix} x - d\sin(\theta) \\ 0 \\ z - d\cos(\theta) \end{Bmatrix}. \tag{3.24}$$

Multiplying the skewsymmetric cross product with the vector and simplifying yields,

$$\vec{M} = -dk\left(\frac{\eta_o}{\eta} - 1\right)\left(x\cos(\theta) - z\sin(\theta)\right)\vec{j}_I \,. \tag{3.25}$$

The change in angular momentum with respect to time is,

$$\frac{d\vec{H}_{B/I}}{dt} = I_{yy}\ddot{\theta}\vec{j}_I \,. \tag{3.26}$$

Equating (3.26) to the moment in eq. (3.25), dotting both sides with $\vec{j}_I$, and dividing by the moment of inertia yields the rotational equation of motion,

$$\ddot{\theta} = \frac{dk}{I_{yy}}\left(\frac{\eta_o}{\eta} - 1\right)\left(z\sin(\theta) - x\cos(\theta)\right). \tag{3.27}$$

Arranging the ground phase equations of motion into state space form yields,

$$\dot{y} = \begin{Bmatrix} y_{(4)} \\ y_{(5)} \\ y_{(6)} \\ \frac{k}{m}\left(\frac{\eta_o}{\eta} - 1\right)\left(y_{(1)} - d\sin\left(y_{(3)}\right)\right) \\ \frac{k}{m}\left(\frac{\eta_o}{\eta} - 1\right)\left(y_{(2)} - d\cos\left(y_{(3)}\right)\right) + g \\ \frac{dk}{I_{yy}}\left(\frac{\eta_o}{\eta} - 1\right)\left(y_{(2)}\sin\left(y_{(3)}\right) - y_{(1)}\cos\left(y_{(3)}\right)\right) \end{Bmatrix}, \tag{3.28}$$

where

$$\eta = \sqrt{\left(y_{(1)} - d\sin\left(y_{(3)}\right)\right)^2 + \left(y_{(2)} - d\cos\left(y_{(3)}\right)\right)^2}\ . \tag{3.29}$$

4 SOLUTION

The ultimate goal is for the system to move forward with an asymptotically stable periodic gait, such that perturbations applied to the system that shift it away from the periodic orbit simply result in the system returning to the original periodic gait. Because a periodic orbit ends where it began, we can represent the continuous periodic orbit discretely with a single fixed point of the associated Poincaré map. The fixed point contains all the states sampled at a particular instant in the periodic orbit. For our system, the sampling is taken when the leg lifts off the ground and the flight phase begins. It is called the lift off state.

4.1 Optimal Criterion

Our goal is to drive the lift off state to the fixed point each period. Since our only control over the system is selection of the leg touchdown angle, β, we look into the future one stride and determine the β that sends the lift off state, y_f, the closest to the fixed point, y_{fixed}. We accomplish this by writing a cost function that penalizes deviations of y_f from y_{fixed} and finding the β which minimizes this cost. The cost function is minimized by taking its derivative and driving it to zero.

4.1.1 Unconstrained Cost Function

As detailed previously, the cost function must penalize the final state if it deviates from the fixed point. It will take the form,

$$J = \phi\left(y_f\right), \tag{4.1}$$

where $\phi\left(y_f\right)$ is a positive semidefinite function which is zero when y_f is equal to y_{fixed} and greater than zero when y_f is not equal to y_{fixed}. The latter is accomplished by using,

$$\Delta Y = y_{fixed} - y_f . \tag{4.2}$$

We nondimensionalize this quantity so that deviations of different quantities are treated equally. This yields,

$$\Delta\overline{Y} = \left\{ \begin{array}{c} \dfrac{\Delta Y_{(1)}}{\eta_o} \\ \dfrac{\Delta Y_{(2)}}{\eta_o} \\ \Delta Y_{(3)} \\ \dfrac{\Delta Y_{(4)}}{v_o} \\ \dfrac{\Delta Y_{(5)}}{v_o} \\ \dfrac{\Delta Y_{(6)}\eta_o}{v_o} \end{array} \right\}, \tag{4.3}$$

where η_o is the uncompressed spring length and v_o is the speed of the center of mass at the beginning of the period. Positive definiteness of ϕ is then assured with

$$\phi\left(y_f\right) = \left(\Delta\overline{Y}\right)^T Q\left(\Delta\overline{Y}\right), \tag{4.4}$$

where Q is a diagonal weighting matrix. The elements of Q are positive and act as tuners for the control system. Note that since the system will move forward with every stride, the first state, x or $y_{f(1)}$, will not be periodic. Therefore it is taken out of the cost function by setting $Q_{(1,1)}$ to zero.

4.1.2 Constrained Cost Function

Unfortunately the minimization of eq. (4.1) cannot help us in its current form since the system has constraints it must follow. It must start at the initial conditions, follow the equations of motion for flight, touch down at the right time, follow equations of motion for the ground phase, and lift off at the correct time. These constraints are broken up into two categories.

The first category includes the time specific constraints. Together with ϕ, from eq. (4.4), these constraints comprise G, the time specific cost. They include the touchdown condition constraints and lift-off constraints. Since the equations of motion for the flight phase can be integrated analytically, instead of using an integral constraint to constrain this motion, we constrain the touchdown state, y_c, to equal the preintegrated function for y in terms of the initial conditions, y_o, and the touchdown time, t_c.

The second category includes the integral constraints. The only integral constraint that needs to be applied is the one constraining the ground phase to follow its equations of motion. There are no unconstrained costs applied over a time interval.

The constrained cost function takes the form,

$$J' = G + \int_{t_c}^{t_f} \left[H - \lambda^T \dot{y} \right] dt \tag{4.5}$$

where G is the time specific constrained cost, $\dot{y}$ is a 6 x 1 vector of the derivatives of the states, y, λ is a 6 x 1 vector of the costates associated with the states, and H is the Hamiltonian given by

$$H = \lambda^T f(t, y), \tag{4.6}$$

where $f(t, y)$ the a 6 x 1 vector of the equations of motion for the ground phase shown in eq. (3.28).

As mentioned previously, the time specific function, G, is broken into the end cost, ϕ, touchdown state constraints, θ, the touchdown condition constraints, χ, and the lift off condition constraints, ψ.

The touchdown state constraints form a 6 x 1 column vector. They constrain the states at the touchdown time to equal the analytical solution to the equations of motion of the flight phase, presented in eq. (3.8), evaluated at the touchdown time, t_c. That is,

$$y_c = y(t_c). \tag{4.7}$$

This ensures that the equations of motion for the flight phase are obeyed. Note in Figure 3.1, that the coordinate, x, which corresponds to $y_{(1)}$, is measured from the foot placement. This definition simplifies the ground phase equations of motion, but it makes it difficult to define $y_{(1)}$ when the leg has not touched down. Since $y_{(1)}$ does not appear in the unconstrained cost function and this calculation is only being done over the ground phase, we do not care how $y_{(1)}$ is defined as long as the rest of the states satisfy the equations of motion. So even though $y_{o(1)}$ may be defined from some other point, we redefine $y_{(1)}$ for the ground phase to be measured from the foot placement. Constraining $y_{c(1)}$ is done by defining $\vec{r}_{C\to A}$ evaluated at the touchdown state in two different ways. The first is using eq. (3.14). The second is using the angle β and the length η_o to express $\vec{r}_{C\to A}$ in the I frame. That is,

$$\vec{r}_{C\to A} = \eta_o\left(-\vec{i}_I\cos(\beta) - \vec{k}_I\sin(\beta)\right). \tag{4.8}$$

Equating these expressions yields,

$$\left(y_{c(1)} - d\sin\left(y_{c(3)}\right)\right)\vec{i}_I + \left(y_{c(2)} - d\cos\left(y_{c(3)}\right)\right)\vec{k}_I = -\eta_o\cos(\beta)\vec{i}_I - \eta_o\sin(\beta)\vec{k}_I. \tag{4.9}$$

Evaluating the dot product of this expression with $\vec{i}_I$ and rearranging yields,

$$\theta_{(1)} = y_{c(1)} - d\sin\left(y_{c(3)}\right) + \eta_o\cos(\beta) = 0. \tag{4.10}$$

The whole initial state constraint vector then becomes,

$$\theta\left(y_c, t_c, \beta\right) = \begin{Bmatrix} y_{c(1)} - d\sin\left(y_{c(3)}\right) + \eta_o \cos\left(\beta\right) \\ y_{c(2)} - y_{o(2)} - y_{o(5)} t_c - \frac{1}{2} g t_c^{\,2} \\ y_{c(3)} - y_{o(3)} - y_{o(6)} t_c \\ y_{c(4)} - y_{o(4)} \\ y_{c(5)} - y_{o(5)} - g t_c \\ y_{c(6)} - y_{o(6)} \end{Bmatrix}. \tag{4.11}$$

The touchdown condition constraint vector, χ, determines when the system must switch from the flight phase to the ground phase in addition to limiting the touchdown time to be positive. The first condition that must be satisfied for the system to go from flight phase to ground phase is the height of the foot from the ground must be zero, indicating foot touch-down. This condition is satisfied by expressing $\vec{r}_{C\to A}$ in two different and equating as before in eq. (4.9). We then dot both sides with $\vec{k}_I$ and reverse signs to obtain

$$\eta_o \sin\left(\beta\right) = d\cos\left(y_{c(3)}\right) - y_{c(2)}. \tag{4.12}$$

Bringing everything to one side yields,

$$\chi_{(1)}\left(y_c, \beta\right) = d\cos\left(y_{c(3)}\right) - y_{c(2)} - \eta_o \sin\left(\beta\right). \tag{4.13}$$

The second condition that must be satisfied is the leg must be entering compression. If, for example, the foot was below the ground at the beginning of the flight phase and moved up so it was at the same height as the ground, the first constraint would hold, but if the center of mass is not moving forward fast enough, the distance between the foot placement and the leg attachment point would increase. If the ground phase starts, this will put the leg in tension. Since the foot must never grip the ground, leg tension should be impossible. In order to eliminate this problem we will specify that the time derivative of the leg length be negative at touchdown. That is the leg length must be decreasing as the leg touches down, forcing compression to occur. Equation (3.15) shows the leg length in terms of the states. We square this quantity for simplicity. This is an acceptable step because the time derivative of the square of a positive definite real quantity has the same sign as the derivative of the quantity itself. This yields,

$$\eta^2 = \left(x - d\sin\left(\theta\right)\right)^2 + \left(z - d\cos\left(\theta\right)\right)^2 . \tag{4.14}$$

Taking the time derivative yields,

$$\frac{d\left(\eta^2\right)}{dt} = 2\left(x - d\sin\left(\theta\right)\right)\left(\dot{x} - d\cos\left(\theta\right)\dot{\theta}\right) + 2\left(z - d\cos\left(\theta\right)\right)\left(\dot{z} + d\sin\left(\theta\right)\dot{\theta}\right). \tag{4.15}$$

This quantity must be less than or equal to zero. Dividing by 2 and substituting the state space variables in eq. (3.6) evaluated at touchdown yields,

$$\begin{aligned}&\left(y_{c(1)} - d\sin\left(y_{c(3)}\right)\right)\left(y_{c(4)} - d\cos\left(y_{c(3)}\right)y_{c(6)}\right)\\ &+\left(y_{c(2)} - d\cos\left(y_{c(3)}\right)\right)\left(y_{c(5)} + d\sin\left(y_{c(3)}\right)y_{c(6)}\right) \le 0\end{aligned} . \tag{4.16}$$

Using the eq. (2.16) we form the constraint,

$$\begin{aligned}\chi_{(2)}\left(y_c, a\right) = &\left(y_{c(1)} - d\sin\left(y_{c(3)}\right)\right)\left(y_{c(4)} - d\cos\left(y_{c(3)}\right)y_{c(6)}\right)\\ &+\left(y_{c(2)} - d\cos\left(y_{c(3)}\right)\right)\left(y_{c(5)} + d\sin\left(y_{c(3)}\right)y_{c(6)}\right) + {a_{(1)}}^2\end{aligned}, \tag{4.17}$$

where $a_{(1)}$ is the first element of a 4 x 1 bounding vector, a. The touchdown time, t_c, must be constrained to be positive. Again using eq. (2.16) we form the constraint,

$$\chi_{(3)}\left(t_c, a\right) = t_c - {a_{(2)}}^2 , \tag{4.18}$$

where $a_{(2)}$ is the second element of the 4 x 1 bounding vector, a. Equations (4.13), (4.17), and (4.18) form the 3 x 1 vector of touchdown condition constraints,

$$\chi\left(y_c, t_c, \beta, a\right) = \left\{\begin{array}{c} d\cos\left(y_{c(3)}\right) - y_{c(2)} - \eta_o\sin\left(\beta\right)\\ {a_{(1)}}^2 + \left(y_{c(1)} - d\sin\left(y_{c(3)}\right)\right)\left(y_{c(4)} - d\cos\left(y_{c(3)}\right)y_{c(6)}\right)\\ +\left(y_{c(2)} - d\cos\left(y_{c(3)}\right)\right)\left(y_{c(5)} + d\sin\left(y_{c(3)}\right)y_{c(6)}\right)\\ t_c - {a_{(2)}}^2 \end{array}\right\} . \tag{4.19}$$

The lift off condition constraint vector, ψ, determines when the system must switch from the ground phase to the flight phase in addition to limiting the ground phase time, t_n, to be positive. The ground phase time is,

$$t_n = t_f - t_c . \tag{4.20}$$

In order for the system to switch from the ground phase to the flight phase, the leg must be fully extended. That is

$$\eta = \eta_o . \tag{4.21}$$

Substituting this into the left side of eq. (3.15), squaring both sides for simplicity, and arranging everything on one side yields the constraint equation,

$$\psi_{(1)}\left(y_f\right) = \eta_o{}^2 - \left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)^2 - \left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)^2 . \tag{4.22}$$

The next condition is the leg must be leaving compression. The first condition, $\psi_{(1)}$, is satisfied both when the leg touches down and when it lifts off. So if we do not specify that the derivative of the spring length is positive, then the ground phase could be cut off where it started. We already calculated the derivative of the leg length in eq. (4.15). Dividing by 2 and substituting the state space coordinates in eq. (3.6) evaluated at the end state yields,

$$\begin{aligned} &\left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)\left(y_{f(4)} - d\cos\left(y_{f(3)}\right)y_{f(6)}\right) \\ &+\left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)\left(y_{f(5)} + d\sin\left(y_{f(3)}\right)y_{f(6)}\right) \geq 0. \end{aligned} \tag{4.23}$$

Using the eq. (2.16) we form the constraint,

$$\begin{aligned} \psi_{(2)}\left(y_f, a\right) = &\left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)\left(y_{f(4)} - d\cos\left(y_{f(3)}\right)y_{f(6)}\right) \\ &+\left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)\left(y_{f(5)} + d\sin\left(y_{f(3)}\right)y_{f(6)}\right) - a_{(3)}{}^2 , \end{aligned} \tag{4.24}$$

where $a_{(3)}$ is the third element of the 4 x 1 bounding vector, a.

The ground phase time, t_n, must be constrained to be positive. Again using eq. (2.16) we form the constraint,

$$\psi_{(3)}\left(t_n, a\right) = t_n - a_{(4)}{}^2, \tag{4.25}$$

where $a_{(4)}$ is the fourth element of the 4 x 1 bounding vector, a.

Equations (4.22), (4.24), and (4.25) form the 3x1 vector of lift off condition constraints,

$$\psi\left(y_f, t_n, a\right) = \left\{ \begin{array}{c} \eta_o{}^2 - \left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)^2 - \left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)^2 \\ -a_{(3)}{}^2 + \left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)\left(y_{f(4)} - d\cos\left(y_{f(3)}\right)y_{f(6)}\right) \\ \quad + \left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)\left(y_{f(5)} + d\sin\left(y_{f(3)}\right)y_{f(6)}\right) \\ t_n - a_{(4)}{}^2 \end{array} \right\}. \tag{4.26}$$

The time-specific function is assembled to yield,

$$G = \phi\left(y_f\right) + \xi^T\theta\left(y_c, t_c, \beta\right) + \upsilon^T\chi\left(y_c, t_c, \beta, a\right) + \nu^T\psi\left(y_f, t_n, a\right). \tag{4.27}$$

where ξ is a 6 x 1 vector of the Lagrange multipliers associated with the touchdown state constraint vector, θ, υ is a 3 x 1 vector of the Lagrange multipliers associated with the touchdown condition constraint vector, χ, and ν is a 3 x 1 vector of Lagrange multipliers associated with the lift off condition constraint vector, ψ. The constraint equations are zero when the constraints are satisfied. Now that each part of the constrained cost function has been derived, it can be assembled displaying its parameter dependencies. This will make is easy to take its total derivative. The final constrained cost function is,

$$J' = \quad G\left(y_c, y_f, t_c, t_n, \beta, a, \xi, \upsilon, \nu\right) + \int_{t_c}^{t_f}\left[H\left(y, \lambda\right) - \lambda^T\dot{y}\right]dt. \tag{4.28}$$

4.1.3 First Derivative

The minimum of a function is found where the functions total derivative is zero. To find the minimum of our cost function, eq. (4.28), we take its total derivative. Using Leibniz' Rule for differentiating the integral, this yields

$$\begin{aligned} dJ' = \; & G_{y_c} dy_c + G_{y_f} dy_f + G_{t_c} dt_c + G_{t_f} dt_f + G_\beta d\beta + G_\xi d\xi + G_\upsilon d\upsilon + G_\nu d\nu \\ & + G_a da + \left[\left(H - \lambda^T \dot{y} \right) dt \right]_{t_c}^{t_f} + \int_{t_c}^{t_f} \left[H_y \delta y + H_\lambda \delta\lambda - \dot{y}^T \delta\lambda - \lambda^T \delta\dot{y} \right] dt, \end{aligned} \tag{4.29}$$

where

$$G_x = \frac{\partial G}{\partial x} \tag{4.30}$$

for x equal to every variable on which G is dependent and

$$H_x = \frac{\partial H}{\partial x} \tag{4.31}$$

for x equal to every variable on which H is dependent. See eq. (4.28) for dependencies. Note from eq. (4.27) that

$$G_\xi = \theta^T, \tag{4.32}$$

$$G_\upsilon = \chi^T, \tag{4.33}$$

and

$$G_\nu = \psi^T. \tag{4.34}$$

Since θ, χ, and ψ were defined to be zero vectors, the $G_\xi d\xi$, $G_\nu d\nu$, and $G_\upsilon d\upsilon$ terms are zero. Even though these terms are zero and they will be removed from dJ' for simplicity, they are still requirements for the total derivative to be zero.

Note from eq. (4.6) that

$$H_\lambda = f^T . \tag{4.35}$$

Substituting eq. (4.35) into eq. (4.29) as well as eliminating the terms we established as zero leaves

$$\begin{aligned} dJ' = \; & G_{y_c} dy_c + G_{y_f} dy_f + G_{t_c} dt_c + G_{t_f} dt_f + G_\beta d\beta + G_a da \\ & + \left[\left(H - \lambda^T \dot{y} \right) dt \right]_{t_c}^{t_f} + \int_{t_c}^{t_f} \left[H_y \delta y - \lambda^T \delta \dot{y} + \left(f^T - \dot{y}^T \right) \delta\lambda \right] dt. \end{aligned} \tag{4.36}$$

We now integrate the $\lambda^T \delta \dot{y}$ term by parts by selecting,

$$u^T = \lambda^T \tag{4.37}$$

and

$$\frac{dv}{dt} = \delta \frac{dy}{dt} = \frac{d}{dt}(\delta y) . \tag{4.38}$$

This results in

$$\int_{t_1}^{t_2} \left[-\lambda^T \delta \dot{y} \right] dt = -\left[\lambda^T \delta y \right]_{t_1}^{t_2} + \int_{t_1}^{t_2} \left[\dot{\lambda}^T \delta y \right] dt . \tag{4.39}$$

Substituting this into eq. (4.36) yields,

$$\begin{aligned} dJ' = \; & G_{y_c} dy_c + G_{y_f} dy_f + G_{t_c} dt_c + G_{t_f} dt_f + G_\beta d\beta + G_a da \\ & + \left[\left(H - \lambda^T \dot{y} \right) dt - \lambda^T \delta y \right]_{t_c}^{t_f} + \int_{t_c}^{t_f} \left[\left(H_y + \dot{\lambda}^T \right) \delta y + \left(f^T - \dot{y}^T \right) \delta\lambda \right] dt. \end{aligned} \tag{4.40}$$

Since $\dot{y} = f$, the term $\left(f^T - \dot{y}^T \right) \delta\lambda$ is zero. Substituting in the limits we obtain

$$\begin{aligned} dJ' = \; & G_{y_c} dy_c + G_{y_f} dy_f + G_{t_c} dt_c + G_{t_f} dt_f + G_\beta d\beta + G_a da \\ & + \left(H_f - \lambda_f{}^T \dot{y}_f \right) dt_f - \lambda_f{}^T \delta y_f - \left(H_c - \lambda_c{}^T \dot{y}_c \right) dt_c + \lambda_c{}^T \delta y_c + \int_{t_c}^{t_f} \left[\left(H_y + \dot{\lambda}^T \right) \delta y \right] dt. \end{aligned} \tag{4.41}$$

Using eq. (2.23) we relate δy to dy eliminating δy outside of the integral to obtain,

$$\begin{aligned} dJ' = \quad & G_{y_c} dy_c + G_{y_f} dy_f + G_{t_c} dt_c + G_{t_f} dt_f + G_\beta d\beta + G_a da \\ & + H_f dt_f - \lambda_f{}^T dy_f - H_c dt_c + \lambda_c{}^T dy_c + \int_{t_c}^{t_f} \left[\left(H_y + \dot{\lambda}^T \right) \delta y \right] dt. \end{aligned} \tag{4.42}$$

Rearranging yields,

$$\begin{aligned} dJ' = \quad & \left(G_{y_c} + \lambda_c{}^T \right) dy_c + \left(G_{y_f} - \lambda_f{}^T \right) dy_f + \left(G_{t_c} - H_c \right) dt_c \\ & + \left(G_{t_f} + H_f \right) dt_f + G_\beta d\beta + G_a da + \int_{t_c}^{t_f} \left[\left(H_y + \dot{\lambda}^T \right) \delta y \right] dt. \end{aligned} \tag{4.43}$$

Since dJ' must equal zero and dy_c, dy_f, dt_c, dt_f, $d\beta$, da, and δy are arbitrary, their coefficients must be zero yielding the boundary conditions

$$\begin{gathered} \lambda_f = G_{y_f}{}^T, \quad \lambda_c = -G_{y_c}{}^T, \quad G_\beta = 0, \quad G_a = 0, \\ H_c = G_{t_c}, \quad H_f = -G_{t_f}, \\ \theta = 0, \quad \psi = 0, \quad \chi = 0, \end{gathered} \tag{4.44}$$

and differential constraint equations,

$$\begin{aligned} \dot{y} &= f, \\ \dot{\lambda} &= -H_y{}^T. \end{aligned} \tag{4.45}$$

4.1.4 Evaluation of Boundary Conditions

We have established all the conditions that must be satisfied for dJ' to be zero but further computation must be done to make these conditions useful. We begin by evaluating the partial derivatives in eq. (4.44). In evaluating G_{y_f} we refer to eq. (4.27) to see that

$$G_{y_f} = \phi_{y_f} + \nu^T \psi_{y_f}. \tag{4.46}$$

Referring to eq. (4.4) we observe that

$$\phi_{y_f} = \left(\Delta \bar{Y} \right)^T Q \left(\Delta \bar{Y}_{y_f} \right). \tag{4.47}$$

Equation (4.2) establishes that ΔY is linear in y_f and each element of ΔY is only dependent on the corresponding element in y_f. As a result ΔY_{y_f} is a negative identity matrix. We can convert ΔY_{y_f} to $\Delta \overline{Y}_{y_f}$ by simply multiplying by a diagonal matrix of the nondimensionalization parameters. Substituting this into eq. (4.47) yields,

$$\phi_{y_f}{}^T = \left\{ \begin{array}{c} 0 \\ \dfrac{2q_2\left(y_{f(2)} - y_{fixed(2)}\right)}{\eta_o{}^2} \\ 2q_3\left(y_{f(3)} - y_{fixed(3)}\right) \\ \dfrac{2q_4\left(y_{f(4)} - y_{fixed(4)}\right)}{v_o{}^2} \\ \dfrac{2q_5\left(y_{f(5)} - y_{fixed(5)}\right)}{v_o{}^2} \\ \dfrac{2q_6\left(y_{f(5)} - y_{fixed(5)}\right)\eta_o{}^2}{v_o{}^2} \end{array} \right\}, \tag{4.48}$$

where q_i for $i = 2$ through 6 are the diagonal elements of the weighting matrix Q. Referring to eq. (4.26) we see that

$$\psi_{y_f}^{\ T} = \begin{bmatrix} 2\left(d\sin\left(y_{f(3)}\right)-y_{f(1)}\right) & \left(y_{f(4)}-d\cos\left(y_{f(3)}\right)y_{f(6)}\right) & 0 \\ 2\left(d\cos\left(y_{f(3)}\right)-y_{f(2)}\right) & y_{f(5)}+d\sin\left(y_{f(3)}\right)y_{f(6)} & 0 \\ \begin{matrix}2d\left(y_{f(1)}-d\sin\left(y_{f(3)}\right)\right)\cos\left(y_{f(3)}\right) \\ -2d\left(y_{f(2)}-d\cos\left(y_{f(3)}\right)\right)\sin\left(y_{f(3)}\right)\end{matrix} & \begin{matrix}d\left(y_{f(2)}y_{f(6)}-y_{f(4)}\right)\cos\left(y_{f(3)}\right) \\ +d\left(y_{f(1)}y_{f(6)}+y_{f(5)}\right)\sin\left(y_{f(3)}\right)\end{matrix} & 0 \\ 0 & y_{f(1)}-d\sin\left(y_{f(3)}\right) & 0 \\ 0 & y_{f(2)}-d\cos\left(y_{f(3)}\right) & 0 \\ 0 & d\left(y_{f(2)}\sin\left(y_{f(3)}\right)-y_{f(1)}\cos\left(y_{f(3)}\right)\right) & 0 \end{bmatrix}. \quad (4.49)$$

In evaluating G_{y_c} we refer to eq. (4.27) to observe that

$$G_{y_c} = \xi^T \theta_{y_c} + \upsilon^T \chi_{y_c}. \quad (4.50)$$

Differentiating eq. (4.11) with respect to y_c yields

$$\theta_{y_c} = \begin{bmatrix} 1 & 0 & -d\cos\left(y_{c(3)}\right) & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \quad (4.51)$$

Differentiating eq. (4.19) with respect to y_c yields

$$\chi_{y_c}{}^T = \begin{bmatrix} 0 & \left(y_{c(4)} - d\cos\left(y_{c(3)}\right)y_{c(6)}\right) & 0 \\ -1 & y_{c(5)} + d\sin\left(y_{c(3)}\right)y_{c(6)} & 0 \\ -d\sin\left(y_{c(3)}\right) & \begin{matrix} d\left(y_{c(2)}y_{c(6)} - y_{c(4)}\right)\cos\left(y_{c(3)}\right) \\ +d\left(y_{c(1)}y_{c(6)} + y_{c(5)}\right)\sin\left(y_{c(3)}\right) \end{matrix} & 0 \\ 0 & y_{c(1)} - d\sin\left(y_{c(3)}\right) & 0 \\ 0 & y_{c(2)} - d\cos\left(y_{c(3)}\right) & 0 \\ 0 & d\left(y_{c(2)}\sin\left(y_{c(3)}\right) - y_{c(1)}\cos\left(y_{c(3)}\right)\right) & 0 \end{bmatrix}. \tag{4.52}$$

In evaluating G_β we refer to eq. (4.27) to see that

$$G_\beta = \xi^T \theta_\beta + \upsilon^T \chi_\beta . \tag{4.53}$$

Differentiating eq. (4.11) with respect to β yields

$$\theta_\beta = \begin{Bmatrix} -\eta_o \sin\left(\beta\right) \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{Bmatrix}. \tag{4.54}$$

Differentiating eq. (4.19) with respect to β yields

$$\chi_\beta = \begin{Bmatrix} -\eta_o \cos\left(\beta\right) \\ 0 \\ 0 \end{Bmatrix}. \tag{4.55}$$

Substituting eqs. (4.54) and (4.55) into (4.53) yields

$$G_\beta = -\xi_{(1)}\eta_o \sin(\beta) - \upsilon_{(1)}\eta_o \cos(\beta). \tag{4.56}$$

In evaluating G_a we refer to eq. (4.27) to see

$$G_a = \upsilon^T \chi_a + \nu^T \psi_a. \tag{4.57}$$

Differentiating eq. (4.19) with respect to a yields

$$\chi_a = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 2a_{(1)} & 0 & 0 & 0 \\ 0 & -2a_{(2)} & 0 & 0 \end{bmatrix}. \tag{4.58}$$

Differentiating eq. (4.26) with respect to a yields

$$\psi_a = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & -2a_{(3)} & 0 \\ 0 & 0 & 0 & -2a_{(4)} \end{bmatrix}. \tag{4.59}$$

Substituting eqs. (4.58) and (4.59) into (4.57) yields

$$G_a = \begin{bmatrix} 2\upsilon_{(2)}a_{(1)} & -2\upsilon_{(3)}a_{(2)} & -2\nu_{(2)}a_{(3)} & -2\nu_{(3)}a_{(4)} \end{bmatrix}. \tag{4.60}$$

In evaluating G_{t_c} we refer to eq. (4.27) to see

$$G_{t_c} = \xi^T \theta_{t_c} + \upsilon^T \chi_{t_c}. \tag{4.61}$$

Differentiating eq. (4.11) with respect to t_c yields

$$\theta_{t_c} = \begin{Bmatrix} 0 \\ -y_{o(5)} - gt_c \\ -y_{o(6)} \\ 0 \\ -g \\ 0 \end{Bmatrix}. \tag{4.62}$$

Differentiating eq. (4.19) with respect to t_c yields

$$\chi_{t_c} = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}. \tag{4.63}$$

Substituting eqs. (4.62) and (4.63) into (4.61) yields

$$G_{t_c} = \xi_{(2)}\left(-y_{o(5)} - gt_c\right) - \xi_{(3)} y_{o(6)} - \xi_{(5)} g + \upsilon_{(3)}. \tag{4.64}$$

We now evaluate the Hamiltonian at touchdown to obtain

$$H_c = \lambda_c^{\ T} f_c. \tag{4.65}$$

Since at the touch down state the spring remains uncompressed, the equations of motion for the flight phase can still be used for f_c. Therefore

$$H_c = \lambda_{c(1)} y_{c(4)} + \lambda_{c(2)} y_{c(5)} + \lambda_{c(3)} y_{c(6)} + \lambda_{c(5)} g. \tag{4.66}$$

In evaluating G_{t_f} we refer to eq. (4.27) to see that

$$G_{t_f} = \nu^T \psi_{t_f}. \tag{4.67}$$

Differentiating eq. (4.26) with respect to t_f yields

$$\psi_{t_f} = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix}. \tag{4.68}$$

Substituting eqs. (4.68) into (4.67) yields

$$G_{t_f} = \nu_{(3)}. \tag{4.69}$$

We can evaluate the Hamiltonian at lift off in a manner similar to that at touchdown. That is,

$$H_f = \lambda_f{}^T f_f \,. \tag{4.70}$$

Since at the final state the spring is uncompressed, the equation of motion vector f reduces to the equations of motion for the flight phase again yielding,

$$H_f = \lambda_{f(1)} y_{f(4)} + \lambda_{f(2)} y_{f(5)} + \lambda_{f(3)} y_{f(6)} + \lambda_{f(5)} g \,. \tag{4.71}$$

4.2 Structure of the Numerical Method

Now that we have developed the necessary conditions for the minimization of our cost function, we must develop a method to drive the system to these conditions. To do this we form a vector called a residual which is driven to zero using a least squares method. This vector contains all the boundary conditions presented in eq. (4.44) with all the terms moved to one side so that the equation is zero when the conditions are satisfied. In addition to this, the residual vector contains the polynomial approximation constraints developed with collocation to enforce the differential constraints in eq. (4.45). These constraints were developed in section 2.5.

4.2.1 Residual

We will now use the constraints that have been developed to form the residual vector, where the constraint equation is satisfied when its corresponding component in the residual vector is zero. In order to impose some kind of order to the residual we will try to organize the components in order of what time they are evaluated. The first part of the residual will be the boundary conditions that are applied at touchdown. These take the form

$$R_{(1)} = H_c\left(\lambda_c, y_c\right) - G_{t_c}\left(\xi, t_c, \upsilon\right), \tag{4.72}$$

$$R_{(2:3)} = G_{a(1:2)}\left(a_{(1:2)}, \upsilon\right), \tag{4.73}$$

$$R_{(4)} = G_{\beta}\left(\xi, \beta, \upsilon\right), \tag{4.74}$$

$$R_{(5:7)} = \chi\left(y_c, t_c, \beta, a_{(1:2)}\right), \tag{4.75}$$

$$R_{(8:13)} = \theta\left(y_c, t_c, \beta\right), \tag{4.76}$$

and

$$R_{(14:19)} = \lambda_c + G_{y_c}\left(\xi, \upsilon, y_{c(3)}\right)^T. \tag{4.77}$$

The next section of the residual vector is made up of the differential constraints at the collocation points of each section for every state. We developed these constraint equations in such a way that they automatically satisfy the differential constraints at the nodes and continuity between sections. These residuals take the form

$$R_{(12j+8:12j+13)} = y'_{c_j}\left(y_{n_j}, y_{n_{j+1}}, t_n\right) - \tilde{f}\left(y_{c_j}\left(y_{n_j}, y_{n_{j+1}}, t_n\right), t_n\right), \tag{4.78}$$

and

$$\begin{aligned} R_{(12j+14:12j+19)} = {} & \lambda'_{c_j}\left(y_{n_j}, \lambda_{n_j}, y_{n_{j+1}}, \lambda_{n_{j+1}}, t_n\right) \\ & + \tilde{H}_y\left(y_{c_j}\left(y_{n_j}, y_{n_{j+1}}, t_n\right), \lambda_{c_j}\left(y_{n_j}, \lambda_{n_j}, y_{n_{j+1}}, \lambda_{n_{j+1}}, t_n\right), t_n\right), \end{aligned} \tag{4.79}$$

for $j = 1$ to N where y'_{c_j} is the $6x1$ vector found using eq. (2.64), $\tilde{f}$ is the modified equation of motion vector found in eq. (2.46) where f is the original equation of motion vector found in eq. (3.28), y_{c_j} is the $6x1$ vector found using eq. (2.66), λ'_{c_j} is the $6x1$ vector found using eq. (2.65), $\tilde{H}_y$ is the modified differential constraint vector for the costates found in eq. (2.45) where H_y is the derivative of the Hamiltonian, found in eq. (4.6), with respect to the states, λ_{c_j} is the $6x1$ vector given by eq. (2.67), and t_n is the ground phase time given by eq. (4.20).

The last elements of the residual are the boundary conditions applied at t_f. They take the form

$$R_{(12N+20:12N+25)} = \lambda_f - G_{y_f}\left(\nu, y_f\right)^T, \tag{4.80}$$

$$R_{(12N+26:12N+28)} = \psi\left(y_f, t_n, a_{(3:4)}\right), \tag{4.81}$$

$$R_{(12N+29)} = H_f\left(\lambda_f, y_f\right), \tag{4.82}$$

and

$$R_{(12N+30:12N+31)} = G_{a(3:4)}\left(a_{(3:4)}, \nu\right). \tag{4.83}$$

The MATLAB code for developing the three pieces of the residual is presented in appendix A.

4.2.2 Collocation State Vector

In a dynamic system the state vector refers to the vector of parameters defining its position and velocity at some instant in time. In a collocation scheme the state vector refers to a vector of all the parameters that are varied in the scheme. In our case this means all the parameters, the Lagrange multiplier, and the states and costates evaluated at the nodes. This vector is then updated to drive the residual vector to zero. Just like the residual vector, the state vector is broken up into three sections. The first contains quantities that are used in the flight phase and at touchdown. They are

$$\mathrm{X}_{(1)} = t_c, \tag{4.84}$$

$$\mathrm{X}_{(2:3)} = a_{(1:2)}, \tag{4.85}$$

$$\mathrm{X}_{(4)} = \beta, \tag{4.86}$$

$$\mathrm{X}_{(5:7)} = \upsilon, \tag{4.87}$$

and

$$\mathrm{X}_{(8:13)} = \xi \,. \tag{4.88}$$

The next part of the state vector contains quantities that are used during the ground phase. These are the states and costates at every node. They are

$$\mathrm{X}_{(12j+2:12j+7)} = y_{n_j} \tag{4.89}$$

and

$$\mathrm{X}_{(12j+8:12j+13)} = \lambda_{n_j} \tag{4.90}$$

for $j = 1$ to $N+1$. The last part of the state vector contains quantities that are used at lift off. These are,

$$\mathrm{X}_{(12N+26:12N+28)} = \nu \,, \tag{4.91}$$

$$\mathrm{X}_{(12N+29)} = t_n \,, \tag{4.92}$$

and

$$\mathrm{X}_{(12N+30:12N+31)} = a_{(3:4)} \,. \tag{4.93}$$

4.2.3 Nondimensionalization of the Residual Vector

To improve the convergence properties of the Newton Raphson routine it can be advantageous to nondimensionalize the residual. We do this by multiplying each residual by a nondimensionalization parameter that has units that are the inverse of that of the residual. For organizational purposes, these nondimensionalization parameters are put together to form nondimensionalization vector, κ.

Equation (4.72) says that the first residual has units equal to those of G_{t_c}. Since G is unitless and t_c has units of time, G_{t_c} has units of the inverse of time. This leaves the nondimensionalization parameter

$$\kappa_{(1)} = \frac{\eta_o}{v_o}, \tag{4.94}$$

where η_o is the uncompressed leg length and v_o is the speed of the center of mass at lift off. Equation (4.73) says that the second residual is equal to $G_{a(1)}$. The constraint parameter, $a_{(1)}$, has units of the square root of velocity. The derivative of G with respect to $a_{(1)}$ has units of the inverse of the square root of velocity. The nondimensionalization parameter then becomes,

$$\kappa_{(2)} = \sqrt{v_o}\ . \tag{4.95}$$

Equation (4.73) also says that the third residual is equal to $G_{a(2)}$. Since the constraint parameter, $a_{(2)}$, has units of the square root of time, the derivative of G with respect to $a_{(2)}$ has units of the inverse of the square root of time. The nondimensionalization parameter then becomes,

$$\kappa_{(3)} = \sqrt{\frac{\eta_o}{v_o}}\ . \tag{4.96}$$

According to eq. (4.74) the next residual is equal to G_β. Since both G and β have no units G_β is unitless. The nondimensionalization constant for $R_{(4)}$ is then,

$$\kappa_{(4)} = 1\ . \tag{4.97}$$

Equation (4.75) shows that $R_{(5)} = \chi_{(1)}$ which as shown in eq. (4.19) has units of length. So,

$$\kappa_{(5)} = \frac{1}{\eta_o}\ . \tag{4.98}$$

Similarly $R_{(6)} = \chi_{(2)}$ which has units of velocity. So,

$$\kappa_{(6)} = \frac{1}{v_o}. \tag{4.99}$$

Finally $R_{(7)} = \chi_{(3)}$ which has units of time. So,

$$\kappa_{(7)} = \frac{v_o}{\eta_o}. \tag{4.100}$$

According to eq. (4.76), $R_{(8)}$ through $R_{(13)}$ have the same units as the constraint equation vector, θ. Since θ constrains the states directly, it will have units equal to those of the states. The nondimensionalization constants corresponding to these residuals will have the inverse units. That is

$$\begin{Bmatrix} \kappa_{(8)} \\ \kappa_{(9)} \\ \kappa_{(10)} \\ \kappa_{(11)} \\ \kappa_{(12)} \\ \kappa_{(13)} \end{Bmatrix} = \begin{Bmatrix} 1/\eta_o \\ 1/\eta_o \\ 1 \\ 1/v_o \\ 1/v_o \\ \eta_o/v_o \end{Bmatrix}. \tag{4.101}$$

According to eq. (4.77), $R_{(14)}$ through $R_{(19)}$ have the same units as λ. The costates have units inverse to those of the states. The nondimensionalization constants corresponding to these residuals will have units equal to those of the states. That is,

$$\begin{Bmatrix} \kappa_{(14)} \\ \kappa_{(15)} \\ \kappa_{(16)} \\ \kappa_{(17)} \\ \kappa_{(18)} \\ \kappa_{(19)} \end{Bmatrix} = \begin{Bmatrix} \eta_o \\ \eta_o \\ 1 \\ v_o \\ v_o \\ v_o / \eta_o \end{Bmatrix}. \tag{4.102}$$

The next residuals are matching the state derivatives at the collocation points. The nondimensionalization constants will be defined for a general collocation point. Equation (4.78) shows that the residuals have units of the time normalized equations of motion. This means they have units the same as the states. So,

$$\begin{Bmatrix} \kappa_{(12j+8)} \\ \kappa_{(12j+9)} \\ \kappa_{(12j+10)} \\ \kappa_{(12j+11)} \\ \kappa_{(12j+12)} \\ \kappa_{(12j+13)} \end{Bmatrix} = \begin{Bmatrix} 1/\eta_o \\ 1/\eta_o \\ 1 \\ 1/v_o \\ 1/v_o \\ \eta_o / v_o \end{Bmatrix}, \tag{4.103}$$

for j =1 to N.

Equation (4.79) shows that the residuals have units of the derivative of the costates with respect to τ. Since τ is unitless, these residuals have the same units as the costates. So,

$$\begin{Bmatrix} \kappa_{(12j+14)} \\ \kappa_{(12j+15)} \\ \kappa_{(12j+16)} \\ \kappa_{(12j+17)} \\ \kappa_{(12j+18)} \\ \kappa_{(12j+19)} \end{Bmatrix} = \begin{Bmatrix} \eta_o \\ \eta_o \\ 1 \\ v_o \\ v_o \\ v_o / \eta_o \end{Bmatrix}. \tag{4.104}$$

for j =1 to N.

Equation (4.80) shows that the next 6 residuals have the same units as λ. So,

$$\begin{Bmatrix} \kappa_{(12N+20)} \\ \kappa_{(12N+21)} \\ \kappa_{(12N+22)} \\ \kappa_{(12N+23)} \\ \kappa_{(12N+24)} \\ \kappa_{(12N+25)} \end{Bmatrix} = \begin{Bmatrix} \eta_o \\ \eta_o \\ 1 \\ v_o \\ v_o \\ v_o / \eta_o \end{Bmatrix}. \tag{4.105}$$

Equation (4.81) shows that the next 3 residuals have units the same as the ψ vector. Since the first, second, and third components of ψ have units of length squared, velocity, and time, respectively, we obtain,

$$\begin{Bmatrix} \kappa_{(12N+26)} \\ \kappa_{(12N+27)} \\ \kappa_{(12N+28)} \end{Bmatrix} = \begin{Bmatrix} \dfrac{1}{\eta_o^2} \\ \dfrac{1}{v_o} \\ \dfrac{v_o}{\eta_o} \end{Bmatrix}. \tag{4.106}$$

According to eq. (4.82), the next residual has the same units as the Hamiltonian, which has units of the inverse of time. So,

$$\kappa_{(12N+29)} = \frac{\eta_o}{v_o} . \tag{4.107}$$

From eq. (4.83) we see that since G is unitless and a_3 has units of the square root of velocity, the next residual has units of the inverse of the square root of velocity. So,

$$\kappa_{12N+30} = \sqrt{v_o} . \tag{4.108}$$

Also from eq. (4.83) we see that since a_4 has units of the square root of time, the last residual has units of the inverse of the square root of time. So,

$$\kappa_{12N+31} = \sqrt{\frac{\eta_o}{v_o}} . \tag{4.109}$$

4.3 Initial Conditions

We are using a Newton-Raphson solver drive the parameters to the optimal solution. The performance of such a solver is closely linked with how close the initial guess is to the solution. It will be beneficial to make this guess as educated as possible.

There is no physical intuition to aid in determining initial assumptions of the magnitude of the costates or Lagrange multipliers. However, we do have information regarding the values of our states. Formulating the unoptimized system as an initial value problem provides an approximation for the states. The only parameter we must pick is the leg touch down angle, β. This will be done by calculating a range in which β could be and calculating the value of the unconstrained cost function for β at a few values inside of that range. Assuming the body is supposed to move in the positive x direction there are three cases for which a range must be developed. The three cases are set apart by the initial velocity in the $\vec{i}_I$ direction, $\dot{x}_o$. The first case is if $\dot{x}_o$ is positive. In such a situation the velocity is forward and should remain forward at the end of the stride. The second case

is if $\dot{x}_o$ is zero. The velocity in the $\vec{i}_I$ direction must be increased so that it is going forward. The third case is if $\dot{x}_o$ is negative. The velocity in the $\vec{i}_I$ direction must be reduced if not reversed.

4.3.1 Bounds on Leg Touchdown Angle for Zero Horizontal Velocity

The middle case is easiest to deal with. If $\dot{x}_o$ is zero, a leg angle of less than $\frac{\pi}{2}$ radians will yield a negative horizontal body velocity. Since we are assuming a target horizontal velocity that is positive, this is counter productive. So for this case, β must be at least $\frac{\pi}{2}$ radians. Every angle greater than this value would yield some velocity in the $\vec{i}_I$ direction until β was increased to π radians. Touch down at this angle would yield no compression in the spring leading to no change in motion and falling. So for this case, β must be no more than π radians.

4.3.2 Bounds on Leg Touchdown Angle for Positive Horizontal Velocity

In the case where $\dot{x}_o$ is positive, if β is too small, the body will not make it over the foot placement point. The angle the velocity vector of the center of mass makes with the horizontal, $\vec{i}_I$, (see Figure 3.1) measured positive counterclockwise will be defined as δ and δ_c will be δ at touchdown. In the point mass case, if β is less than $-\delta_c$, then $\dot{x}$ will reverse direction and the system is in danger of falling. The minimum quantity that β can be is where it equals $-\delta_c$ at touchdown. This quantity is determined by writing both β and δ_c in terms of t_c, setting the negative of one equal to the positive of the other, and solving for t_c. The goal is,

$$\beta_{\min} = -\delta_c \,. \tag{4.110}$$

It is easier to express the sines of the angles rather than the angles themselves in terms of t_c. Taking the sine of both sides yields,

$$\sin\left(\beta_{\min}\right) = -\sin\left(\delta_c\right). \tag{4.111}$$

Expressing the sine of δ_c in terms of a ratio of speeds at touchdown yields,

$$\sin(\delta_c) = \frac{-\dot{z}_c}{\sqrt{\dot{x}_c^{\,2} + \dot{z}_c^{\,2}}}. \tag{4.112}$$

Substituting quantities from eq. (3.8) into eq. (4.112) yields,

$$\sin(\delta) = \frac{-\dot{z}_o - gt_c}{\sqrt{\dot{x}_o^{\,2} + (\dot{z}_o + gt_c)^2}} \tag{4.113}$$

We would like to express the sine of β in terms of a ratio of lengths. The first length we need is the distance between the foot placement, C, and the leg attachment point, A, in the $-\vec{k}_I$ direction. This can be found by evaluating the dot product of eq. (3.14) and $-\vec{k}_I$, yielding,

$$-\vec{k}_I \cdot \vec{r}_{C\to A} = d\cos(\theta_c) - z_c \tag{4.114}$$

Substituting the quantities from eq. (3.8) into eq. (4.114) yields

$$-\vec{k}_I \cdot \vec{r}_{C\to A} = -z_o - \dot{z}_o t_c - \frac{1}{2} g t_c^{\,2} + d\cos(\theta_o + \dot{\theta}_o t_c) \tag{4.115}$$

The second length is the uncompressed leg length, η_o. Expressing the sine as ratio of these lengths yields,

$$\sin(\beta) = \frac{-2z_o - 2\dot{z}_o t_c - g t_c^{\,2} + 2d\cos(\theta_o + \dot{\theta}_o t_c)}{2\eta_o}. \tag{4.116}$$

Combining eqs. (4.113) and (4.116) with eq. (4.111) and rearranging yields,

$$0 = \frac{-2z_o - 2\dot{z}_o t_c - g t_c^{\,2} + 2d\cos(\theta_o + \dot{\theta}_o t_c)}{2\eta_o} - \frac{\dot{z}_o + gt_c}{\sqrt{\dot{x}_o^{\,2} + (\dot{z}_o + gt_c)^2}}. \tag{4.117}$$

While we cannot solve this equation explicitly for t_c, by using some bounds that we will develop later on t_c we can use the False Position method to solve for it numerically [19]. Since this equation is defined for a case when $\dot{x}_o$ is nonzero there is no danger attaining an infinite value.

Because the False Position method requires a bound on either side of the root it is trying to find, we must find a lower limit and an upper limit on t_c. The leg should never touch down while the attachment point is still moving upward. So the minimum t_c is where the leg attachment point, A (see Figure 3.1), is at the maximum height. The right side of eq. (4.115) is equal to the height of A. Assuming that $\dot{\theta}$ is small enough in magnitude that point A does not make a full rotation about the center of mass, B, while it is in the air, the minimum t_c is where the derivative of this equation is zero. This value will be called t_t. Taking the derivative yields,

$$0 = -\dot{z}_o - gt_t - d\dot{\theta}_o \sin\left(\theta_o + \dot{\theta}_o t_t\right). \tag{4.118}$$

Taking a second order Taylor series expansion of the sine term yields the quadratic polynomial,

$$0 = -\dot{z}_o - gt_t - d\dot{\theta}_o \sin\left(\theta_o\right) - d\dot{\theta}_o^{\,2} \cos\left(\theta_o\right) t_t + \frac{d}{2}\dot{\theta}_o^{\,3} \sin\left(\theta_o\right) t_t^{\,2}. \tag{4.119}$$

The solution to this is found using the quadratic formula. Organizing the terms yields,

$$0 = \underbrace{\frac{d}{2}\dot{\theta}_o^{\,3} \sin\left(\theta_o\right)}_{a} t_t^{\,2} \underbrace{-\left(g + d\dot{\theta}_o^{\,2} \cos\left(\theta_o\right)\right)}_{b} t_t \underbrace{-\dot{z}_o - d\dot{\theta}_o \sin\left(\theta_o\right)}_{c}. \tag{4.120}$$

Because when $d = 0$, the coefficient of $t_t^{\,2}$ goes to zero and the typical quadratic formula goes to infinity, we use the rationalized quadratic equation given by,

$$t_t = -\frac{2c}{b \pm \sqrt{b^2 - 4ac}} \text{ [19].} \tag{4.121}$$

Substituting the quantities from eq. (4.120) into eq. (4.121) yields,

$$t_t = -\frac{2\left(\dot{z}_o + d\dot{\theta}_o \sin\left(\theta_o\right)\right)}{\left(g + d\dot{\theta}_o^{\,2} \cos\left(\theta_o\right)\right) \pm \sqrt{\left(g + d\dot{\theta}_o^{\,2} \cos\left(\theta_o\right)\right)^2 + 2\left(d\dot{\theta}_o^{\,3} \sin\left(\theta_o\right)\right)\left(\dot{z}_o + d\dot{\theta}_o \sin\left(\theta_o\right)\right)}}. \tag{4.122}$$

If $\dot{\theta}_o$ is set to zero the subtraction of the square root leads to an infinite value for t_t. This means that adding the square root is the correct choice for calculation of t_t. The final expression is,

$$t_t = -\frac{2\left(\dot{z}_o + d\dot{\theta}_o \sin\left(\theta_o\right)\right)}{g + d\dot{\theta}_o{}^2 \cos\left(\theta_o\right) + \sqrt{\left(g + d\dot{\theta}_o{}^2 \cos\left(\theta_o\right)\right)^2 + 2\left(d\dot{\theta}_o{}^3 \sin\left(\theta_o\right)\right)\left(\dot{z}_o + d\dot{\theta}_o \sin\left(\theta_o\right)\right)}}. \quad (4.123)$$

We must also establish a maximum bound on t_c such that eq. (4.117) is satisfied. If the attachment point was at ground level, β would be zero. This would mean that δ_c would have to be zero to satisfy eq. (4.110), but such a situation would be impossible at this height because there would have to be some vertical velocity for the system to reach this state. If the leg attachment point were slightly above the ground, then β would be slightly greater than zero and δ_c would be slightly less than zero. This situation would certainly be possible. The equation we have for height of the leg attachment point is eq. (4.115). Setting this to zero and solving for t will give us the time at which the leg attachment point is at zero height. This time will be referred to as t_g. Taking a second order Taylor series expansion of the cosine term yields,

$$0 = \left(-\frac{1}{2}g - \frac{1}{2}d\cos\left(\theta_o\right)\dot{\theta}_o{}^2\right)t_g{}^2 + \left(-\dot{z}_o - d\sin\left(\theta_o\right)\dot{\theta}_o\right)t_g + d\cos\left(\theta_o\right) - z_o. \quad (4.124)$$

Plugging this into the quadratic formula yields,

$$t_g = \frac{-\left(\dot{z}_o + d\sin\left(\theta_o\right)\dot{\theta}_o\right) \pm \sqrt{\left(\dot{z}_o + d\sin\left(\theta_o\right)\dot{\theta}_o\right)^2 + 2\left(g + d\cos\left(\theta_o\right)\dot{\theta}_o{}^2\right)\left(d\cos\left(\theta_o\right) - z_o\right)}}{\left(g + d\cos\left(\theta_o\right)\dot{\theta}_o{}^2\right)}. \quad (4.125)$$

The term $d\cos\left(\theta_o\right) - z_o$ represents the attachment height at lift off. This must be positive. As long as θ_o remains between $\frac{\pi}{2}$ and -$\frac{\pi}{2}$ radians, $\cos\left(\theta_o\right)$ is positive. Since $\dot{\theta}_o{}^2$ is positive, g is positive and d is positive, $g + d\cos\left(\theta_o\right)\dot{\theta}_o{}^2$ is positive. This means the magnitude of the square root term is greater than the magnitude of $\dot{z}_o + d\sin\left(\theta_o\right)\dot{\theta}_o$. Since t_g must be positive and the denominator is positive, the positive root is the correct value of t_g. That is,

$$t_g = \frac{-\left(\dot{z}_o + d\sin\left(\theta_o\right)\dot{\theta}_o\right) + \sqrt{\left(\dot{z}_o + d\sin\left(\theta_o\right)\dot{\theta}_o\right)^2 + 2\left(g + d\cos\left(\theta_o\right)\dot{\theta}_o{}^2\right)\left(d\cos\left(\theta_o\right) - z_o\right)}}{\left(g + d\cos\left(\theta_o\right)\dot{\theta}_o{}^2\right)}. \quad (4.126)$$

Using the False Position method with t_t and t_g as bounds, eq. (4.117) is solved for t_c. This is then substituted into eq. (4.116) and the arcsine is taken to get the minimum touchdown angle, $\beta_{\min}$.

The maximum angle that β can be must also be established for this case. If the touchdown angle is such that the leg is perpendicular to the direction of the velocity, the leg will not be compressed at touchdown. Instead it will simply touch and lift off again immediately. If β is any greater than this, it will do the same thing. If it is less, it will touch down. This state can be quantified using β and δ_c. See Figure 4.1.

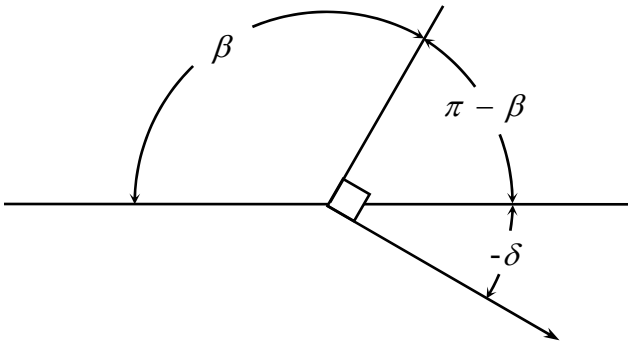

Figure 4.1 Relation of maximum β to δ at touchdown. The angle β is measured between the horizontal and the leg at touchdown in a clockwise direction. The angle δ is measured between the velocity vector and the horizontal counterclockwise direction.

The equation relating these quantities is,

$$\frac{\pi}{2} = \pi - \beta - \delta_c . \tag{4.127}$$

Simplifying yields,

$$\beta = \frac{\pi}{2} - \delta . \tag{4.128}$$

As before we write these both in terms of t_c and solve. It is easier to write both in terms of their sine. Taking the sine of both sides yields,

$$\sin(\beta) = \sin\left(\frac{\pi}{2} - \delta_c\right). \tag{4.129}$$

Simplifying using trigonometry gives,

$$\sin(\beta) = \cos(\delta_c). \tag{4.130}$$

Expressing the cosine in terms of a ratio of lengths gives,

$$\cos(\delta) = \frac{\dot{x}_o}{\sqrt{\dot{x}_o^{\,2} + (\dot{z}_o + gt_c)^2}} \tag{4.131}$$

Substituting eqs. (4.116) and (4.131) into eq. (4.130) and rearranging yields,

$$0 = \frac{-2z_o - 2\dot{z}_o t_c - gt_c^{\,2} + 2d\cos(\theta_o + \dot{\theta}_o t_c)}{2\eta_o} - \frac{\dot{x}_o}{\sqrt{\dot{x}_o^{\,2} + (\dot{z}_o + gt_c)^2}} \tag{4.132}$$

Since this equation is defined for a case when $\dot{x}_o$ is nonzero there is no danger attaining an infinite value. False Position method must again be used to calculate the proper touchdown time. The bounds from before (eqs. (4.123) and (4.126)) are still valid in this situation. The touchdown time, t_c, is then substituted into eq. (4.116) and the arcsine is taken to get the maximum leg touchdown angle, $\beta_{\max}$.

4.3.3 Bounds on Leg Touchdown Angle for Negative Horizontal Velocity

For the d equal zero case, if $\dot{x}_o$ is negative, touchdown angles of greater than or equal to $-\delta$ and less than π will reverse the velocity in the x direction in one step. But this could be at such a cost to the lift off height and vertical velocity that it is better to slow the horizontal velocity in one step and change its direction in the next. For this reason, the lower bound on β is left as low as $\frac{\pi}{2}$ radians. The upper bound remains π radians.

4.3.4 Modified Limits Due to Insufficient Height

In any of the three cases, if the height of the leg attachment point at the top of the flight phase is such that it cannot accommodate the full range of values, the angle at which the uncompressed leg will fit is the maximum value of β in the positive $\dot{x}_o$ case and the minimum value of β in the cases where $\dot{x}_o$ is less than or equal to zero. To determine if the height at the top of the flight phase will be sufficient, the right side of eq. (4.116) is evaluated at t_t. If this is greater than or equal to one, the full range of β can be used as defined previously. If it is less than one, a new limit must be found. It is either a new maximum or new minimum depending of the sign of $\dot{x}_o$.

For the $\dot{x}_o$ positive case, eq. (4.116) is solved for β and evaluated at $t_c = t_t$. That is,

$$\beta_{\max} = \sin^{-1}\left(\frac{-2z_o - 2\dot{z}_o t_t - g t_t^{\,2} + 2d\cos\left(\theta_o + \dot{\theta}_o t_t\right)}{2\eta_o}\right). \tag{4.133}$$

For the $\dot{x}_o$ less than or equal to zero case, eq. (4.133) is subtracted from π to get the minimum touchdown angle that is possible. That is,

$$\beta_{\min} = \pi - \sin^{-1}\left(\frac{-2z_o - 2\dot{z}_o t_t - g t_t^{\,2} + 2d\cos\left(\theta_o + \dot{\theta}_o t_t\right)}{2\eta_o}\right). \tag{4.134}$$

The MATLAB code that calculates the limits on β and selects the range of leg angles to test is presented in appendix B.2.1.

4.3.5 Calculation of Other States

Simulations are evaluated using the range of β defined previously and the initial conditions. Since the flight phase is integrated analytically, an estimation of the touchdown time, t_c, is determined and substituted into the integrated equations, eq. (3.8). The estimation of t_c is found by rearranging eq. (4.116) to yield,

$$0 = -\eta_o \sin\left(\beta\right) - z_o - \dot{z}_o t - \frac{1}{2} g t^2 + d\cos\left(\theta_o + \dot{\theta}_o t\right). \tag{4.135}$$

Determining a second order Taylor series expansion of the cosine term about $t = 0$ and evaluating it at t_c, yields,

$$0 = t_c^{\,2}\underbrace{\left(-\frac{1}{2}g - \frac{1}{2}\dot{\theta}_o^{\,2} d\cos(\theta_o)\right)}_{a} + t_c\underbrace{\left(-\dot{z}_o - \dot{\theta}_o d\sin(\theta_o)\right)}_{b}\underbrace{-z_o + d\cos(\theta_o) - \eta_o\sin(\beta)}_{c} \quad (4.136)$$

This can be substituted into the quadratic formula to obtain,

$$t_c = \frac{-\left(\dot{z}_o + d\sin(\theta_o)\dot{\theta}_o\right) \pm \sqrt{\left(\dot{z}_o + d\sin(\theta_o)\dot{\theta}_o\right)^2 + 2\left(g + d\cos(\theta_o)\dot{\theta}_o^{\,2}\right)\left(d\cos(\theta_o) - z_o - \eta_o\sin(\beta)\right)}}{\left(g + d\cos(\theta_o)\dot{\theta}_o^{\,2}\right)}. \quad (4.137)$$

Since the flight time should be a positive quantity, we must select the sign in the quadratic formula to ensure that this happens. Since the denominator of eq. (4.137) is positive, the only way to use the negative root and still have positive t_c is if b in eq. (4.136) is positive and the square root term in eq. (4.137) is less than b. The square root term can only be less than b if the product of a and c is positive. As long as θ_o remains between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, a is guaranteed to be negative. So c must also be negative. However looking more closely at c we can see that $-z_o + d\cos(\theta_o)$ is simply the height of the leg attachment point at the initial time and $\eta_o\sin(\beta)$ is the vertical distance from the foot to the leg attachment point. The subtraction of the two yields the height of the foot above the ground at the initial time. If c is negative then when the foot first reaches the ground, it will be coming up through the ground. We would like the foot to continue past this until it is coming down to the ground from above. This means that even if we could have a positive t_c by subtracting the square root we would rather have the greater positive t_c. So,

$$t_c = \frac{-\left(\dot{z}_o + d\sin(\theta_o)\dot{\theta}_o\right) + \sqrt{\left(\dot{z}_o + d\sin(\theta_o)\dot{\theta}_o\right)^2 + 2\left(g + d\cos(\theta_o)\dot{\theta}_o^{\,2}\right)\left(d\cos(\theta_o) - z_o - \eta_o\sin(\beta)\right)}}{\left(g + d\cos(\theta_o)\dot{\theta}_o^{\,2}\right)}. \quad (4.138)$$

The ground phase is simply integrated using a Runge-Kutta routine. This Runge-Kutta routine is presented in appendix B.2.4. The final conditions, y_f, from the simulations that provide the lowest unconstrained cost are then used to calculate quantities at the final time. The MATLAB code that is used to find the leg angle with the lowest unconstrained cost is presented in appendix

B.2.2. The ground phase time, t_n, can be found from the simulation. We have a boundary condition,

$$\lambda_f = G_{y_f}\left(y_f, \nu\right)^T . \tag{4.139}$$

We have another condition,

$$H_f - G_{t_f} = 0 , \tag{4.140}$$

where

$$H_f = \lambda_{f(1)} y_{f(4)} + \lambda_{f(2)} y_{f(5)} + \lambda_{f(3)} y_{f(6)} + \lambda_{f(5)} g \tag{4.141}$$

and

$$G_{t_f} = \nu_{(3)} . \tag{4.142}$$

Substituting $G_{y_f(1)}$ through $G_{y_f(3)}$ and $G_{y_f(5)}$ into eq. (4.141) for the costates yeilds

$$H_f\left(y_f, \nu_{(1)}, \nu_{(2)}\right) - \nu_{(3)} = 0 . \tag{4.143}$$

We still do not know the value of $\nu_{(1)}$, $\nu_{(2)}$ and $\nu_{(3)}$. From equation (4.60) we obtain the relationships

$$-2\nu_{(2)} a_{(3)} = 0 , \tag{4.144}$$

and

$$-2\nu_{(3)} a_{(4)} = 0 . \tag{4.145}$$

As a result, if $a_{(3)}$ is nonzero then $\nu_{(2)}$ must be zero and if $a_{(4)}$ is nonzero then $\nu_{(3)}$ must equal zero. From eq. (4.24) we get

$$a_{(3)} = \sqrt{\begin{array}{l}\left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)\left(y_{f(4)} - d\cos\left(y_{f(3)}\right)y_{f(6)}\right) \\ +\left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)\left(y_{f(5)} + d\sin\left(y_{f(3)}\right)y_{f(6)}\right)\end{array}} \quad . \tag{4.146}$$

That is $a_{(3)}$ is the square root of the derivative of the leg length at lift off. As long as the leg length is increasing at lift off (which it should) then $a_{(3)}$ is nonzero. From eq. (4.25) we get

$$a_{(4)} = \sqrt{t_n} \ . \tag{4.147}$$

If the ground phase time, t_n, was zero then there would be no ground phase. This would not be acceptable so $a_{(4)}$ is also nonzero. Since $\nu_{(2)}$ and $\nu_{(3)}$ are zero, eq. (4.143) reduces to

$$\begin{aligned} 0 = \ & \nu_{(1)}\left(\left(d\sin\left(y_{f(3)}\right) - y_{f(1)}\right)y_{f(4)} + \left(d\cos\left(y_{f(3)}\right) - y_{f(2)}\right)y_{f(5)}\right) \\ & + \nu_{(1)}y_{f(6)}d\left(\left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)\cos\left(y_{f(3)}\right) - \left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)\sin\left(y_{f(3)}\right)\right) \\ & + \frac{q_2\left(y_{f(2)} - y_{fixed(2)}\right)}{\eta_o^{\,2}}y_{f(5)} + q_3\left(y_{f(3)} - y_{fixed(3)}\right)y_{f(6)} + \frac{q_5\left(y_{f(5)} - y_{fixed(5)}\right)}{v_o^{\,2}}g \end{aligned} \tag{4.148}$$

From this reduced form $\nu_{(1)}$ can be solved for to yield

$$\nu_{(1)} = \frac{q_2\left(y_{f(2)} - y_{fixed(2)}\right)v_o^{\,2}y_{f(5)} + q_3\left(y_{f(3)} - y_{fixed(3)}\right)y_{f(6)}\eta_o^{\,2}v_o^{\,2} + q_5\left(y_{f(5)} - y_{fixed(5)}\right)g\eta_o^{\,2}}{\left(\begin{array}{l}\left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)y_{f(4)} + \left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)y_{f(5)} \\ -d\left(\left(y_{f(1)} - d\sin\left(y_{f(3)}\right)\right)\cos\left(y_{f(3)}\right) - \left(y_{f(2)} - d\cos\left(y_{f(3)}\right)\right)\sin\left(y_{f(3)}\right)\right)y_{f(6)}\end{array}\right)\eta_o^{\,2}v_o^{\,2}} \ . \tag{4.149}$$

Substituting $\nu_{(1)}$ into eq. (4.139) provides an initial guess for the final costates. We can use the differential constraint for the costates to numerically integrate backwards, yielding initial guesses for all the costates at the nodes. The numerical integration technique used is the same Runge-Kutta routine used to integrate the states forward in time before. It is presented in appendix B.2.4. While we are integrating the costates backward, we also integrate the states backward using a time step that ensures that the states are evaluated at each node. These evaluations are used as the initial guesses for the states at the nodes.

The costates at the touchdown node are, λ_c. The boundary condition from eq. (4.44) is

$$\lambda_c = -G_{y_c}\left(\xi, \upsilon_{(1:2)}, y_c\right)^T . \tag{4.150}$$

From eq. (4.60) we get the relationship

$$2\upsilon_{(2)} a_{(1)} = 0 . \tag{4.151}$$

From eq. (4.17) we get,

$$a_{(1)} = \sqrt{\begin{matrix} -\left(y_{c(1)} - d\sin\left(y_{c(3)}\right)\right)\left(y_{c(4)} - d\cos\left(y_{c(3)}\right)y_{c(6)}\right) \\ -\left(y_{c(2)} - d\cos\left(y_{c(3)}\right)\right)\left(y_{c(5)} + d\sin\left(y_{c(3)}\right)y_{c(6)}\right) \end{matrix}} . \tag{4.152}$$

This means if the derivative of the leg length is nonzero at touchdown, then $\upsilon_{(2)}$ is zero. If the derivative of the leg length at touchdown is zero, then $\upsilon_{(2)}$ is nonzero. Since the leg must compress to touchdown, the leg length should be decreasing from uncompressed to compressed and $\upsilon_{(2)}$ is zero.

We have another constraint equation,

$$G_\beta\left(\xi_{(1)}, \beta, \upsilon_{(1)}\right) = 0 . \tag{4.153}$$

Because

$$G_{y_c(1)} = \xi_{(1)} , \tag{4.154}$$

we know from eq. (4.150) that

$$\xi_{(1)} = -\lambda_{c(1)} . \tag{4.155}$$

Equation (4.56) can be solved for $\upsilon_{(1)}$ to yield,

$$\upsilon_{(1)} = \lambda_{c(1)} \tan\left(\beta\right) . \tag{4.156}$$

This is then substituted into eq. (4.150) and we can solve for the remaining initial guesses for the elements of the ξ. This yields,

$$\xi = \begin{Bmatrix} -\lambda_{c(1)} \\ \lambda_{c(1)} \tan(\beta) - \lambda_{c(2)} \\ \lambda_{c(1)} d \tan(\beta) \sin\left(y_{c(3)}\right) - \lambda_{c(1)} d \cos\left(y_{c(3)}\right) - \lambda_{c(3)} \\ -\lambda_{c(4)} \\ -\lambda_{c(5)} \\ -\lambda_{c(6)} \end{Bmatrix}. \tag{4.157}$$

Equation (4.60) shows us,

$$-2\upsilon_{(3)} a_{(2)} = 0\,, \tag{4.158}$$

and eq. (4.18) shows us that

$$a_{(2)} = \sqrt{t_c}\,. \tag{4.159}$$

In most cases the flight time is nonzero so $\upsilon_{(3)}$ is zero, but in some cases the optimal trajectory does not leave room for a flight phase. In this case $a_{(2)}$ is zero and $\upsilon_{(3)}$ can be solved for using the constraint,

$$H_c = G_{t_c} \tag{4.160}$$

where H_c is given by eq. (4.66) and G_{t_c} is given by eq. (4.64). This yields,

$$\upsilon_{(3)} = \lambda_{c(1)} y_{c(4)} + \lambda_{c(2)} y_{c(5)} + \lambda_{c(3)} y_{c(6)} + \lambda_{c(5)} g + \xi_{(2)} y_{o(5)} + \xi_{(3)} y_{o(6)} + \xi_{(5)} g\,. \tag{4.161}$$

The MATLAB code used to calculate all the states presented in this section is in appendix B.2.3.

5 RESULTS

A requirement of the control system is that it must have known fixed points to which it drives the system. Since the model conserves energy, the control system must drive the system to a fixed point that is at the same energy level as its current state. This means that the system must have a continuum of known fixed points to choose from, spanning a range of energies. Such a map could easily be created and a fixed point picking code, written, to select a fixed point based on the energy of the system at the time of selection.

The goal of this work was to develop and test the control system over a range of fixed points to determine its effect on gait stability. The control system was tested on both the point mass SLIP model and the rigid body SLIP model using parameters similar to those of *Blaberus discoidalis*, the death head cockroach. These parameters included a spring stiffness, k, of 20 N/m, a leg length, η_o, of 0.015 m [9], leg attachment distance, d, of 0.004 m, a body mass, m, of 0.0025 kg, a sagital moment of inertia, I_{yy}, of $1.86 \times 10^{-7}\,\mathrm{kg \cdot m^2}$ [22]. The range of leg touchdown angles, β, tested, were centered around the leg touchdown angle that allowed the system to match experimental stride lengths in [9], [23]. This will be referred to as the nominal touchdown angle, β_n and was 1.2 radians.

5.1 Point Mass SLIP

For the point mass SLIP model, fixed points were found for 21 gait families where the touchdown β was held fixed at 21 different values between 1.1 and 1.3 radians. The initial velocity angle, δ_o, was then varied between 0 and 1 radians. An initial speed, v_o, and initial height, z_o, were found for the specific δ_o that yielded the same state at the end of a stride as at the beginning. See Figure 5.1. This resulted in 101 fixed points for every touchdown angle, β. The periodic orbits found had a lift off height equal to their touchdown height. Because the spring was attached at the center of mass, this meant that the lift off angle was equal to the touchdown angle.

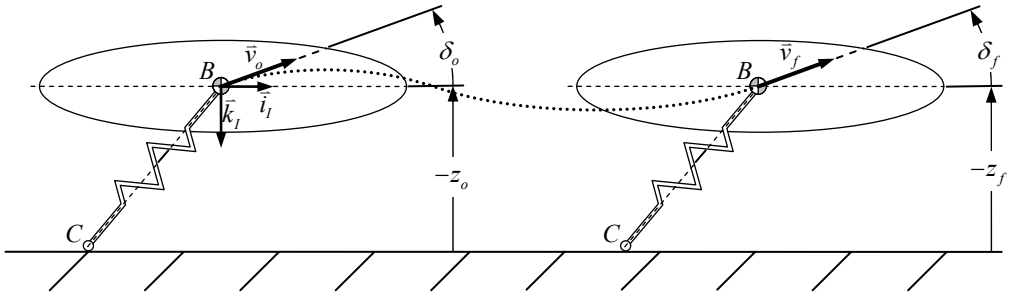

Figure 5.1 Periodic orbit's relation to its fixed point. On the left the point mass SLIP is at lift off entering the stride. On the right the point mass SLIP is at lift off leaving the stride. The combination of z_o, δ_o, and v_o is a fixed point because $z_f = z_o$, $\delta_f = \delta_o$, and $v_f = v_o$.

5.1.1 Stability of Point Mass Fixed Points

The eigenvalues of the Poincaré map linearized about these fixed points were tested for a fixed touchdown angle reset policy and then again using the control system to calculate the optimal touchdown angle. The eigenvalues were determined as described in section 2.7. The Poincaré section used to determine the Poincaré Map was

$$\eta_o = \sqrt{x^2 + z^2} \ . \tag{5.1}$$

Since all points on the Poincaré Map are on this surface, x_o could be determined by z_o with eq. (5.1) and was therefore omitted from all the points in the map. The difference formula used to find $D\mathbf{P}$ was,

$$d\mathbf{P}_i = \frac{\mathbf{P}\left(\mathbf{y}^* - 2\Delta\mathbf{v}_i\right) - 8\mathbf{P}\left(\mathbf{y}^* - \Delta\mathbf{v}_i\right) + 8\mathbf{P}\left(\mathbf{y}^* + \Delta\mathbf{v}_i\right) - \mathbf{P}\left(\mathbf{y}^* + 2\Delta\mathbf{v}_i\right)}{12\Delta y_i} \tag{5.2}$$

for i = 1 to 3 where $d\mathbf{P}_i$ is the column vector of partial derivatives with respect to the i^{th} state, $\mathbf{P}$ is the Poincaré Map, $\mathbf{y}^*$ is the fixed point, and $\Delta\mathbf{v}_i$ a vector with all elements zero except the i^{th} element which is a small nonzero quantity, Δy_i [19]. The column vectors, $d\mathbf{P}_i$ for i = 1 to 3, were then concatenated to form $D\mathbf{P}$ and the eigenvalues of this matrix were found. As mentioned in

section 2.7 the eigenvalues of this matrix are the eigenvalues of the Poincaré map linearized about the fixed point, $\mathbf{y}^*$ and the nontrivial *Floquet multipliers* of the periodic orbit associated with $\mathbf{y}^*$.

The Floquet multipliers of the periodic orbits for the β_n gait family, using the fixed angle reset policy where the leg touchdown angle is set fixed to the leg angle associated with the gait family are shown in Fig. 5.2.

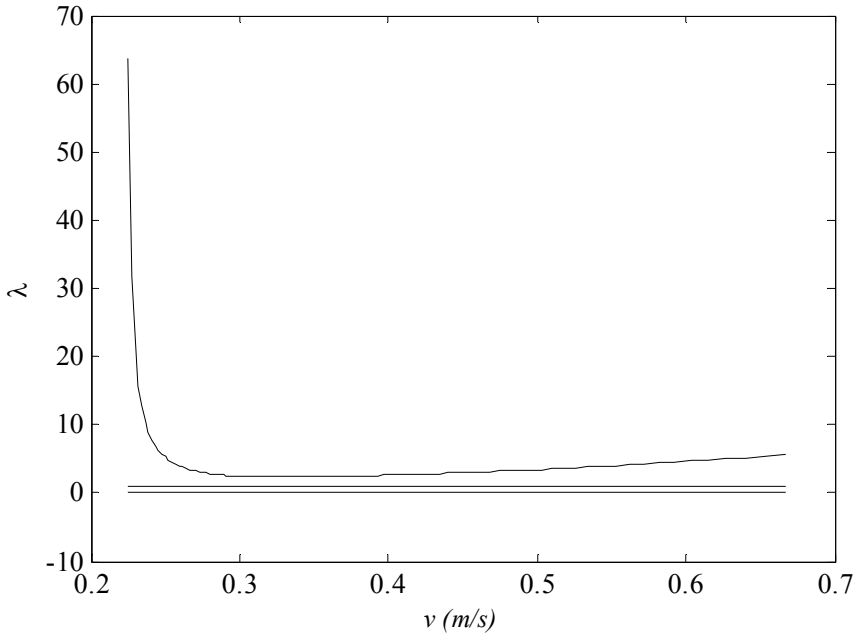

Figure 5.2 Floquet multipliers of the periodic orbits in the nominal gait family for the point mass SLIP with a fixed angle reset policy.

For a periodic orbit to be asymptotically stable, the magnitudes of all its nontrivial Floquet multipliers must be below one. However, because the system is energy conservative there will be at least one nontrivial Floquet multiplier equal to one. This is apparent when looking at Fig. 5.2. In addition there is one Floquet multiplier that is very close to zero and another that is consistently above one, rendering the system at this gait family completely unstable.

The Floquet multipliers of the periodic orbits in the β_n gait family, using model predictive control, are shown in Fig. 5.3. They were found using control weightings $Q_{(1,1)}$ through $Q_{(6,6)}$, of 0, 4, 0, 1, 3, and 0. These weightings appeared to work well in initial tests. Once again, since the system is energy conservative, there is a nontrivial Floquet multiplier of one. Since the rest of the multipliers have magnitude of less than one, the system displays partial asymptotic stability, as has

been found in the SLIP model for several different leg touchdown protocols [10], [15]. The eigenvector associated with the unity eigenvalue of the Poincaré map points in the direction of increasing energy. The plane that is perpendicular to this eigenvector is a constant energy surface. If the system is perturbed from the surface it cannot get back to the energy surface with the target fixed point because it can neither dissipate nor generate the energy required to get there. Because model predictive control exhibits partial asymptotic stability for many energy levels the system could easily assume a new partially asymptotically stable gait at its new energy surface.

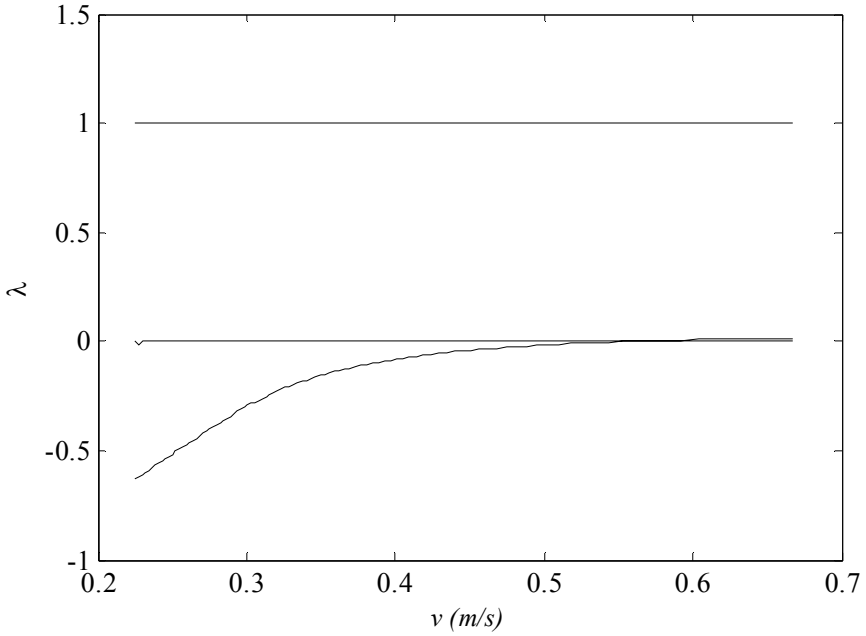

Figure 5.3 Floquet multipliers of the periodic orbits in the nominal gait family for the point mass SLIP with model predictive control.

As outlined above, the stability of each gait was determined by the magnitude of the eigenvalues of the Poincaré map linearized about fixed point associated with the gait. If there was a single eigenvalue of magnitude greater than one, the gait was said to be unstable. The stability of the gaits in all 21 of the gait families tested using a fixed angle reset policy can be found in Fig. 5.4. This figure shows a small area of stable gaits for the β =1.1 and 1.11 gait families using a small δ_o.

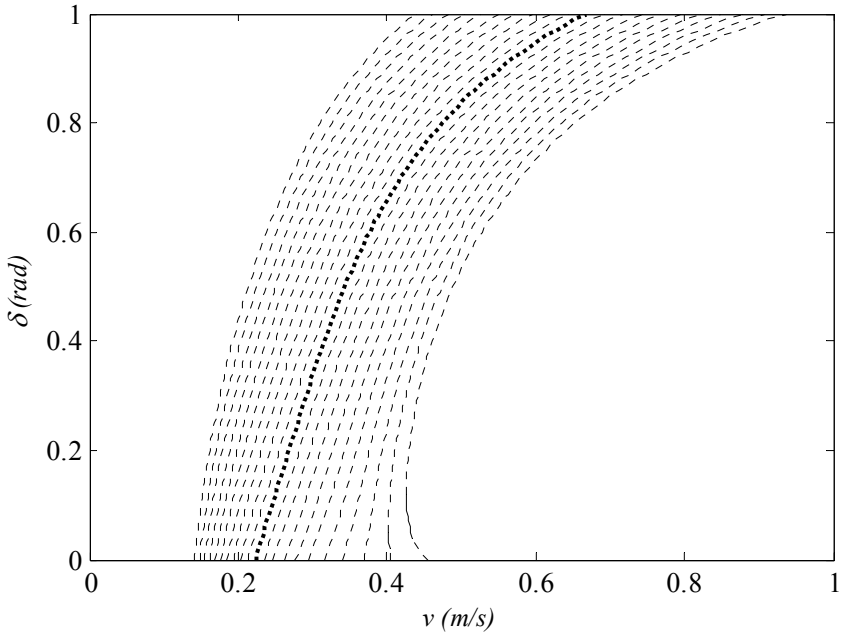

Figure 5.4 Stability of point mass SLIP gait families with a fixed angle reset policy for $\beta = 1.1$ to 1.3 radians. The dotted lines represent unstable gaits. The solid lines represent stable gaits. The right most gait family is that associated with $\beta = 1.1$. Gait families associated with increasing β are found by moving toward the left. The gait family associated with the nominal leg touchdown angle, $\beta = 1.2$ is indicated by the thicker line.

While this is impressive because with essentially no control applied, the system can maintain a forward pace without falling, it does not leave much freedom in terms of choice of speed or energy surface. Also note that for the nominal leg angle the system is never stable.

The stability of the gaits in all 21 of the gait families tested using model predictive control can be found in Fig. 5.5. The figure shows all gait families to be stable provided a sufficient velocity angle, δ_o, is used. This allows for great freedom in speed and energy surface.

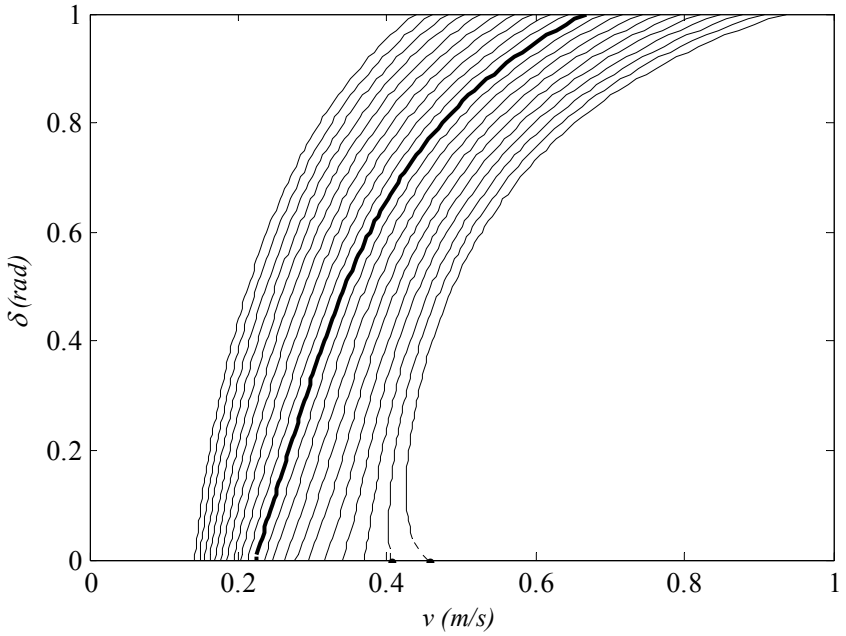

Figure 5.5 Stability of point mass SLIP gait families with model predictive control for $\beta = 1.1$ to 1.3 radians. The dotted lines represent unstable gaits. The solid lines represent stable gaits. The right most gait family is that associated with $\beta = 1.1$. Gait families associated with increasing β are found by moving toward the left. The gait family associated with the nominal leg touchdown angle, $\beta = 1.2$ is indicated by the thicker line.

5.1.2 Perturbation Returnability of Point Mass System

The control scheme was also tested to see if it could return the system from large perturbations to the fixed points. Because the system conserves energy the perturbations had to leave the system at the same energy level as the fixed point itself or there would be no hope of return. Perturbations were made to the initial velocity angle, δ_o, and it was left to the control scheme to pick touchdown angles that drove the system back to the target fixed point and of course the periodic orbit associated with it. Fig. 5.6 shows a map of the returnability. All the fixed points in the nominal gait family were tested, from δ_o=0 to 1 radian. The initial conditions given to the system were the fixed points with δ_o perturbed from $-\frac{31\pi}{32}$ to π radians in increments $\frac{\pi}{32}$ radians. The system was able to return from almost any perturbation to δ_o, even in some cases where the initial conditions sent the body directly into the ground. The vein of no return, shown in black, was due to a combination of the initial velocity angle sending the body into the ground and the initial speed being too great. It

can be seen in Fig. 5.5 that as δ_o increases so does the initial speed, v_o. This combination led the system to fall before it could return itself to its fixed point.

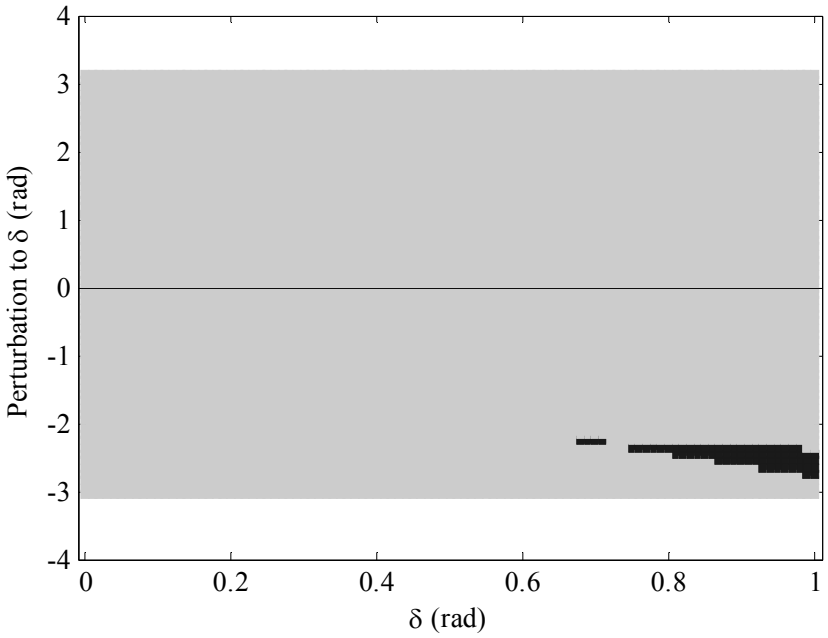

Figure 5.6 The control system's ability to return from a perturbation to δ_o. Returnability was measured for the gate family associated with β=1.2. The system was perturbed from the velocity angle associated with the fixed point (shown on the abscissa) by the angle shown on the ordinate. In grey is the region for which the system was returned to the fixed point. In black is the region for which the system fell.

5.2 Rigid Body SLIP

For the rigid body SLIP model, fixed points were found in the nominal gait family. This was done using the fsolve function in MATLAB in conjunction with a simulation of the system over one stride. A code for the simulation of the system is presented in appendix C.1. First fsolve was run with the simulation starting with the ground phase and ending with the flight phase. In this case the touchdown angle, β, was held at 1.2 radians and the touchdown velocity angle, δ_c, was held at values between 0 and 1 radian, while the touchdown speed, v_c, body pitch, θ_c, and angular velocity, $\dot{\theta}_c$, were varied to produce a fixed point. Since the touchdown height, z_c, can be calculated from β and θ_c this quantity did not need to be varied. Then fsolve was run again with the simulation starting with the flight phase and ending with the ground phase. The initial conditions for the routine were obtained from the lift off conditions of the results of the first fsolve routine. Here the

touchdown angle was held fixed again but the velocity angle at lift off instead of touchdown was held fixed, with β the same as before and δ_o equal to $-\delta_c$. The parameters that were varied were v_o, θ_o, $\dot{\theta}_o$. Since β does not necessarily constrain z_o, it was varied as well. Since the periodic orbits found were very close to symmetric about the middle of the ground phase and flight phase, the first ground phase – flight phase routine got the fixed point very close to the flight phase – ground phase fixed point. The second fsolve routine was more of a refinement of the fixed point than anything else. The MATLAB code used to find the fixed points is presented in appendix C.2.1.

5.2.1 Stability of Rigid Body Fixed Points

The Floquet multipliers of the periodic orbits in the nominal gait family were tested for the rigid body case using the same method as the point mass case with the addition of, θ and $\dot{\theta}$ to the fixed point vectors. The MATLAB code used to calculate the Floquet multipliers is presented in appendix C.2.3. This resulted in five nontrivial Floquet multipliers for every periodic orbit. The magnitudes of the Floquet multipliers of the periodic orbits in the nominal gait family using the fixed angle reset policy can be found in Fig. 5.7.

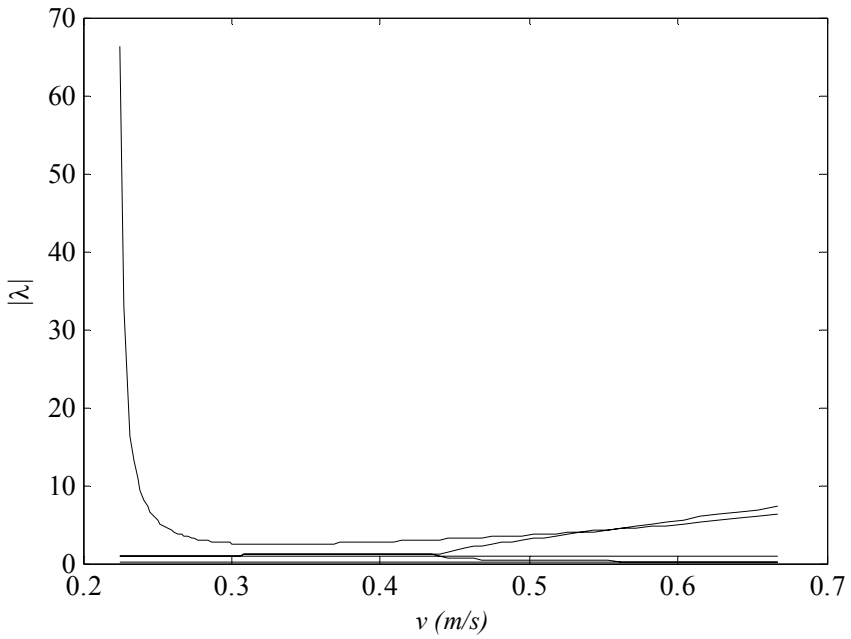

Figure 5.7 The magnitudes of the Floquet multiplier of the periodic orbits in the nominal gait family for the rigid body SLIP with a fixed angle reset policy.

The most noticeable feature is the sweeping Floquet multiplier that seems to reach an asymptote at around v=0.22. This is reminiscent of Fig. 5.2 where the same thing occurred for the point mass case. This Floquet multiplier again maintains a magnitude greater than one for the entire gait family, rendering the entire gait family unstable.

The Floquet multipliers of the periodic orbits in the nominal gait family were also tested using model predictive control. Their magnitudes can be found in Fig. 5.8.

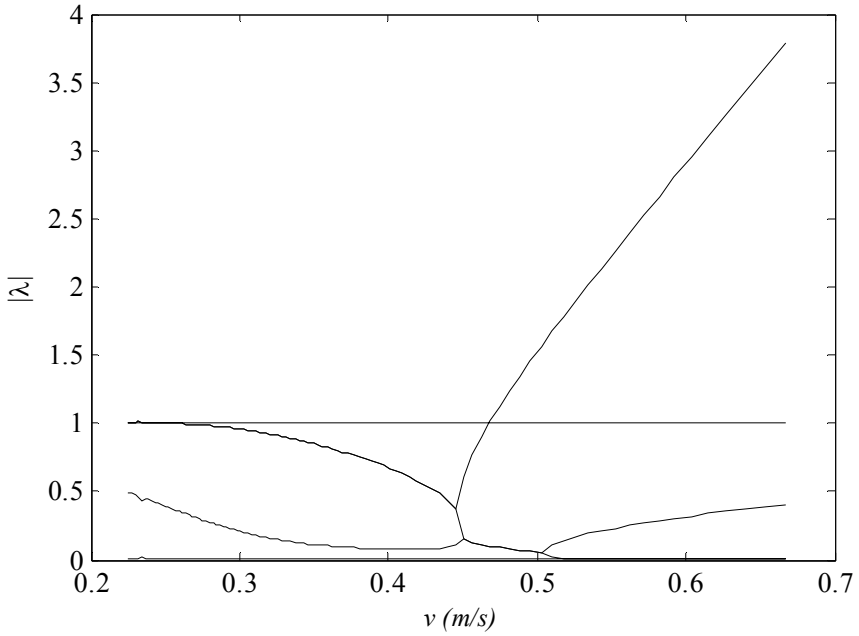

Figure 5.8 The magnitudes of the Floquet multiplier of the periodic orbits in the nominal gait family for the rigid body SLIP with model predictive control.

The Floquet multipliers were found using control weightings $Q_{(1,1)}$ through $Q_{(6,6)}$, of 0, 4, 8, 1, 3, and 7. These weightings appeared to work the best in initial tests but further study should include a more thorough investigation of how the weightings affect performance. From this plot it is clear that many of the Floquet multipliers were complex for portions of the gait family. Just as before, the unity Floquet multiplier is present throughout the gait family. For v values less than 0.26 m/s the magnitude of a second Floquet multiplier is just above one, rendering all gaits below v=0.26 m/s unstable. Another Floquet multiplier shoots past the unity marker at v=0.47 m/s, making

everything above it unstable. This leaves stable gaits in the nominal gait family for lift off speeds from 0.26 to 0.47 m/s.

5.2.2 Perturbation Returnability of Rigid Body System

Returnability was tested in the rigid body case for a gait in the nominal gait family with a δ of 0.5 radians. The MATLAB code for this test is presented in appendix C.2.4. The control system recovered consistently from perturbations to δ as low as $-\frac{\pi}{2}$ and as high as $\frac{35\pi}{32}$ but fell intermittently for perturbations outside that region.

6 CONCLUSION

In this work we used the point mass and rigid body spring loaded inverted pendulums to model sagittal plane locomotion. Since the key to stable locomotion is running the system at a periodic orbit, we developed a cost function that achieved a minimum when the lift off state was at a desired fixed point of the Poincaré map and increased as the lift off state moved away from the fixed point. We constrained the cost function to follow the model and developed a boundary value problem through its minimization. We then converted the boundary value problem into a system of nonlinear equation using Collocation. Each stride this set of nonlinear equations was solved using Newton's method to determine the next optimal touchdown angle. The performance of this model predictive control scheme along with a fixed leg touchdown angle reset policy was investigated when applied to the SLIP with parameters typical of the cockroach *Blaberus discoidalis*.

In the point mass case, we found that for the parameter range tested, the fixed angle reset policy yielded gaits that were widely unstable for a large range of leg touchdown angles and gait speeds, although a limited number of stable gaits were found. When the model predictive control scheme was applied, almost all the gaits in every gait family were stable, although a limited number of unstable gaits were found. The control scheme's ability to return the point mass SLIP from large energy conservative perturbations was found to be very impressive. It could return the system from every perturbation tested that it was physically possible to return from.

The gait family tested in the rigid body case with the fixed angle reset policy showed no stable gaits while the same gait family tested with the model predictive control scheme showed

stable gaits for roughly half the gait family. For the gait tested, the control scheme consistently returned the rigid body SLIP from large energy conservative perturbations within a region but only had intermittent success outside that region.

It is unfair to compare model predictive control to other control schemes developed for the SLIP, because this formulation requires greatly increased computation over other schemes. Boundary value problems can be extremely hard to solve and the one we ask the controller to solve every flight phase is not trivial. The computation of an optimal leg angle in MATLAB takes about 25 seconds on average (although it can be much more), while for the parameters found in a cockroach, the flight phase is about 35 milliseconds (although it can be zero). The code must be optimized and translated into compiled language but a huge gap remains between theory and implementation. Using Moore's Law we can predict that if the code is optimized and translated into C, this control method will be viable for use on robots of cockroach morphology in 15 years. This time would be significantly reduced if it were implemented on a larger robot.

The solution of the set of nonlinear equations not only yields the optimal leg touchdown angle but also the lift off state. This could be used to run Newton iterations during the ground phase to predict better initial conditions for the Newton iterations in the flight phase using the real lift off state. In addition the flight time could be constrained to be at least a certain length so as to allow time for computation, although the longer the flight phase is constrained to be, the less robust the system will be. This is a necessary step however because aside from computational time, an issue that was not addressed in this formulation was how the spring gets from its position at lift off to the next touchdown angle. Because the spring was considered to be massless, theoretically it would be able to move instantaneously from one place to another. Although relative to the body, the mass of the spring is negligible, trying to move it from one position to the next in zero time is not realistic. In addition to giving the spring some time to position itself for the next stride, a method for it to do so must be developed.

Another drawback is that this scheme requires full state feedback at the lift off state. Although this is better than requiring continuous full state feedback, for something this small moving this rapidly, this is not very realistic. With additional formulation an observer could be used to eliminate some of the feedback requirements but a control system that does not require so much feedback to begin with would be much easier to implement.

Given that this scheme will not be practical for implementation for quite some time, and that it outperforms most other schemes in terms of leg angle choice, we suggest that it be used as a target from which to gain insight in developing other control schemes, rather than being taken seriously as a control scheme itself. It certainly turns the point mass SLIP into a savvy monopode, and although more development should be done for the rigid body implementation, it has shown more than limited success in stabilizing rigid body gaits. This scheme should be easily extendable to the three dimensional spring loaded inverted pendulums, known as the spatial SLIP, and the rigid body spatial SLIP, which includes rolling, pitching and yawing of the body. These models better approximate the gaits of higher dimensional robots at the expense of being more complicated. Because of the latter, there has been little success in controlling such models. The application of model predictive control to these models would help to give some insight into how a successful leg angle reset policy should act even though it would have the same if not greater computational drawbacks.

BIBLIOGRAPHY

[1] R.J. Full, K. Autumn, J.I. Chung, and A. Ahn, "Rapid negotiation of rough terrain by the death-head cockroach," *American Zoologist*, Vol. 38, pp. 81A, 1998

[2] Full, R.J., Blickhan, R., Ting, L.H., "Leg Design In Hexapedal Runners", *The Journal of Experimental Biology*, Vol. 158, pp. 369-390, 1991.

[3] Full, R.J., Tu, M.S., "Mechanics of Six-Legged Runners", *The Journal of Experimental Biology*, Vol. 148, pp. 129-146, 1990.

[4] Full, R.J., Koditschek, D.E., "Templates and Anchors: Neuromechanical Hypotheses of Legged Locomotion On Land", *The Journal of Experimental Biology*, Vol. 202, pp. 3325-3332, 1999.

[5] Blickhan, R., "The spring-mass model for running and hopping", *Journal of Biomechanics*, Vol. 22, pp. 1217-1227, 1989

[6] Blickhan, R., Full, R.J. "Similarity in multi-legged locomotion: bouncing like a monopode", *Journal of Comparative Physiology A*, Vol. 173, pp. 509-517, 1993

[7] Seyfarth, A., Geyer, H., Günther, M., Blickhan, R., "A Movement Criterion For Running", *Journal of Biomechanics*, Vol. 35, pp. 649-655, 2002.

[8] Geyer, H., Seyfarth, A., Blickhan, R., "Spring-mass running: simple approximate solution and application to gait stability", *Journal of Theoretical Biology*, Vol. 232, pp. 315-328, 2005.

[9] Schmitt, J.M., "A Simple Stabilizing Control for Sagittal Plane Locomotion", *Journal of Computational and Nonlinear Dynamics*, in press, Oct. 2006.

[10] Sayfarth, A., Geyer, H., Herr, H., "Swing-leg Retraction: A Simple Control Model for Stable Running", *The Journal of Experimental Biology*, Vol. 206, pp. 2547-2555, 2003.

[11] Saranli, U., Schwind, W.J., Koditschek, D.E., "Toward the Control of a Multi-Jointed, Monoped Runner" Proceedings of IEEE International Conference on Robotics and Automation, Leuven, Belgium, Vol. 3, pp. 2676-2682.

[12] Saranli U., Koditschek, D.E., "Template Based Control of Hexapedal Running", Proceedings of 2003 IEEE International Conference on Robotics and Automation, Taipei, Taiwan, Vol. 1, pp. 1374-1379.

[13] Saranli, U., Buehler, M., Koditschek, D.E., "RHex - A Simple Highly Mobile Hexapod Robot", *The International Journal of Robotics Research*, Vol. 20, pp. 616-631, 2001.

[14] Schmitt, J.M., "Simple Feedback Control of Running", *Lecture Notes on Control and Information Sciences*, to appear.

[15] Ghigliazza, R.M., Altendorfer, R., Holmes, P., Koditschek, D., "A Simply Stabilized Running Model", *SIAM Review*, Vol. 47, pp. 519-549, 2005.

[16] Altendorfer, R., Koditscheck, D.E., Holmes, P., "Stability Analysis of Legged Locomotion Models, by Symmetry-Factored Return Maps", *The International Journal of Robotics Research*, Vol. 23, pp. 979-999, 2004.

[17] Mombaur, K.D., Longman, R.W., Bock, H.G., Schlöder, J.P., "Open-Loop Stable Running", *Robotica*, Vol. 23, pp. 21-33, 2005.

[18] Hull, D.G., *Optimal Control Theory for Applications*, Springer-Verlag Inc., New York, NY, 2003.

[19] Hoffman, J.D., *Numerical Methods for Engineers and Scientists*, Marcel Dekker, Inc., New York, NY, 2001.

[20] Costello, M.F. Personal Communication, 2005.

[21] Strogatz, S. H., *Nonlinear Dynamics and Chaos*, Perseus Books Publishing, LLC., Cambridge, MA, 1994.

[22] Kram, R., Wong, B., Full, R.J., "Three-Dimensional Kinematics And Limb Kinetic Energy Of Running Cockroaches", *The Journal of Experimental Biology*, Vol. 200, pp. 1919-1929, 1997.

[23] Ting, L., Blickhan, R., and Full, R.J., "Dynamic and Static Stability in Hexapedal Runners", *The Journal of Experimental Biology*, Vol. 197, pp. 251-269, 1994.

[24] Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T., Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, New York, NY, 1992.

APPENDICES

APPENDIX A

Residual and Hessian Development Code

Because they were so complicated, the residual and the Hessian were calculated using a code. This code not only calculated them, but put them in a form that could easily be augmented by a formatting code which made them into pieces of MATLAB code themselves to be used in the residual8 and analytical_hessian1 codes themselves.

A.1 euler_param_calc_ad1

The function euler_param_calc_ad1 calculates the residual vector and Hessian matrix. Since this task takes lots of time and memory, the code only calculates one section of the residual and the Hessian at a time. The calculation of the middle of the residual and the Hessian was the most daunting task. These were defined iteratively so that the collocation scheme could be broken up into any number of segments.

```
function euler_param_calc_ad1(calc)
%=========================================================================
% euler_param_calc
%
% Calculates one of the three pieces of the residual vector
% as well as the hessian matrix and saves them to text files for
% formatting.
%
% Inputs:
%   calc   Depending on the value the function calculates the begining,
%          middle, or end of the residual and hessian:  1 for begining, 2
%          for middle, and 3 for end.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================

  clc

%-------------------------------------------------------------------------
% Declare the symbolic parameters used in the euler parameter calculation
%-------------------------------------------------------------------------
```

```
% Render all symbolic parameters that are used in a subfunction global
  global g m I k tc tn d etao vo betaTD dHdy

% Model Constant Parameters
  syms g m I k d etao vo betaTD positive

% Nondimensionalization parameter - initial speed of the body at the last
% lift off.
% Note, this is constant for each application of the control.
  syms vo positive

% Variable Parameters
  syms tc tn betaTD positive

% Numerical Approximation quantities.
  syms N integer

% Defining States, Coestates, and Lagrange Multipliers
  for i = 1:6
    %Physical States
      y(i,1)       = sym(['y' num2str(i)],'real');
      ym(i,1)      = sym(['ym' num2str(i)],'real');
      yp(i,1)      = sym(['yp' num2str(i)],'real');
      yo(i,1)      = sym(['yo' num2str(i)],'real');
      yc(i,1)      = sym(['yc' num2str(i)],'real');
      yf(i,1)      = sym(['yf' num2str(i)],'real');
      yfixed(i,1)  = sym(['yfixed' num2str(i)],'real');
      lambda(i,1)  = sym(['lambda' num2str(i)],'real');
      lambdam(i,1) = sym(['lambdam' num2str(i)],'real');
      lambdap(i,1) = sym(['lambdap' num2str(i)],'real');
      lambdac(i,1) = sym(['lambdac' num2str(i)],'real');
      lambdaf(i,1) = sym(['lambdaf' num2str(i)],'real');
      xi(i,1)      = sym(['xi' num2str(i)],'real');
  end

% Define constraint parameters a
  for i = 1:4
      a(i,1) = sym(['a' num2str(i)],'real');

  end

% Define the Lagrange multiplier vectors upsilon and nu
  for i = 1:3
      upsilon(i,1) = sym(['upsilon' num2str(i)],'real');
      nu(i,1)      = sym(['nu' num2str(i)],'real');
  end


% The weighting matrix is mostly zeros
  Q   = sym(zeros(6,6));

% Defining the weightings of the penalties for each state's deviation
% and putting them in their place in the weighting matrix.
% Note q1 equals 0 because we do not want x to return to the same value
  for i = 2:6
      Q(i,i)    = sym(['q' num2str(i)],'real');
  end

% The deviation of the final state from the desired fixed state yfixed
  devYlo      = yf-yfixed
```

```
% nondimensionalizing
  devYlobar   = nondimensionalizer(devYlo)

% penalty function
  phi         = devYlobar'*Q*devYlobar

% Starting State Constraints.
  theta = yc-[d*sin(yc(3))-etao*cos(betaTD);...
              yo(2)+yo(5)*tc+1/2*g*tc^2;...
              yo(3)+yo(6)*tc;...
              yo(4);...
              yo(5)+g*tc;...
              yo(6)]

% End State Constraint
  psi   = [etao^2-(yf(1)-d*sin(yf(3)))^2-(yf(2)-d*cos(yf(3)))^2;...
           -a(3)^2+(yf(1)-d*sin(yf(3)))*(yf(4)-d*cos(yf(3))*yf(6))+...
                   (yf(2)-d*cos(yf(3)))*(yf(5)+d*sin(yf(3))*yf(6));...
           tn-a(4)^2];

% Parameter Constraint
  chi   = [d*cos(yc(3))-yc(2)-etao*sin(betaTD);...
           a(1)^2+(yc(1)-d*sin(yc(3)))*(yc(4)-d*cos(yc(3))*yc(6))+...
                  (yc(2)-d*cos(yc(3)))*(yc(5)+d*sin(yc(3))*yc(6));...
           tc-a(2)^2];

% The Time-specific function
  G     = phi+xi.'*theta+nu.'*psi+upsilon.'*chi

% Hamiltonian
  H     = lambda.'*fg(y)

% Hamiltonian evaluated at the touchdown point
  Hc    = lambdac.'*ff(yc)

% Hamiltonian evaluated at the lift of point
  Hf    = lambdaf.'*ff(yf)

% Taking partial derivatives for the boundary conditions
  Gyf   = jake(G,yf)

  Gyc   = jake(G,yc)

  Gbeta = jake(G,betaTD)

  Ga    = jake(G,a)

  Gtc   = jake(G,tc)

% Euler equation
  dHdy    = jake(H,y).'

%-------------------------------------------------------------------------
% Develop Differential Constraints
%-------------------------------------------------------------------------
% Make Ktilda matrix
  TM = [1 0 0 0; 0 1 0 0; 1 1/N 1/N^2 1/N^3; 0 1 2/N 3/N^2]
  Ktilda = inv(TM)
```

```
% Make the time vectors for creation of the polynomial approximation
  tauc          = 1/(2*N)
  tau_vec       = [1 tauc tauc^2 tauc^3]
  tau_vec_prime = [0 1 2*tauc 3*tauc^2]

% Evaluate the derivative of the polynomial approximation at the
% collocation point.
  ycol_prime      = (tau_vec_prime*Ktilda*[ym.';...
                                           tn*fg(ym).';...
                                           yp.';...
                                           tn*fg(yp).']).';

  lambdacol_prime = (tau_vec_prime*Ktilda*[lambdam.';...
                                           -tn*Hy(ym,lambdam).';...
                                           lambdap.';...
                                           -tn*Hy(yp,lambdap).']).';


% Evaluate the polynomial approximation at the collocation point.
  ycol      = (tau_vec*Ktilda*[ym.';...
                               tn*fg(ym).';...
                               yp.';...
                               tn*fg(yp).']).';

  lambdacol = (tau_vec*Ktilda*[lambdam.';...
                               -tn*Hy(ym,lambdam).';...
                               lambdap.';...
                               -tn*Hy(yp,lambdap).']).';

% Plug the polynomial approximations into the the differential constraints.
% Subract the differential constraints from the derivative of the
% polynomial constraint found above.
  Ryi       = ycol_prime-tn*fg(ycol);
  Rlambdai  = lambdacol_prime+tn*Hy(ycol,lambdacol);



%-------------------------------------------------------------------------
% Substitution Definitions for Residual Vector
%-------------------------------------------------------------------------
  subsvec1     = {'tc','a1','a2','betaTD','upsilon1','upsilon2',...
                  'upsilon3'};
  subsvec2     = {'X1','X2','X3','X4','X5','X6','X7'};
  subsvec3     = {'X(1)','X(2)','X(3)','X(4)','X(5)','X(6)','X(7)'};
  for i = 1:6
      subsvec1(i+7)  = xi(i);
      subsvec2(i+7)  = {['X' num2str(i+7)]};
      subsvec3(i+7)  = {['X(' num2str(i+7) ')']};

      %State at connection time substitution
      subsvec1(i+13) = yc(i);
      subsvec2(i+13) = {['X' num2str(i+13)]};
      subsvec3(i+13) = {['X(' num2str(i+13) ')']};

      subsvec1(i+19) = lambdac(i);
      subsvec2(i+19) = {['X' num2str(i+19)]};
      subsvec3(i+19) = {['X(' num2str(i+19) ')']};

      %State at first node substitution.
      subsvec1(i+25) = ym(i);
```

```
    subsvec2(i+25) = {['X12ip' num2str(i+1)]};
    subsvec3(i+25) = {['X(12*i+' num2str(i+1) ')']};

    subsvec1(i+31) = lambdam(i);
    subsvec2(i+31) = {['X12ip' num2str(i+7)]};
    subsvec3(i+31) = {['X(12*i+' num2str(i+7) ')']};

    %State at second node substitution
    subsvec1(i+37) = yp(i);
    subsvec2(i+37) = {['X12ip' num2str(i+13)]};
    subsvec3(i+37) = {['X(12*i+' num2str(i+13) ')']};

    subsvec1(i+43) = lambdap(i);
    subsvec2(i+43) = {['X12ip' num2str(i+19)]};
    subsvec3(i+43) = {['X(12*i+' num2str(i+19) ')']};

    %State at final time substitution
    subsvec1(i+49)  = yf(i);
    subsvec2(i+49)  = {['X12Np' num2str(i+13)]};
    subsvec3(i+49)  = {['X(12*N+' num2str(i+13) ')']};

    subsvec1(i+55)  = lambdaf(i);
    subsvec2(i+55)  = {['X12Np' num2str(i+19)]};
    subsvec3(i+55)  = {['X(12*N+' num2str(i+19) ')']};

    %Initial state substitution
    subsvec1(i+61)  = yo(i);
    subsvec2(i+61)  = yo(i);
    subsvec3(i+61)  = {['yo(' num2str(i) ')']};

    %Fixed point state substitution
    subsvec1(i+67)  = yfixed(i);
    subsvec2(i+67)  = yfixed(i);
    subsvec3(i+67)  = {['yfix(' num2str(i) ')']};

end
for i = 2:6
    subsvec1(i+72)  = Q(i,i);
    subsvec2(i+72)  = Q(i,i);
    subsvec3(i+72)  = {['q(' num2str(i) ')']};
end
%Last Lagrange Multiplier substitution
for i = 1:3
    subsvec1(i+78) = nu(i);
    subsvec2(i+78) = {['X12Np' num2str(i+25)]};
    subsvec3(i+78) = {['X(12*N+' num2str(i+25) ')']};
end
subsvec1(82) = {'tn'};
subsvec2(82) = {'X12Np29'};
subsvec3(82) = {'X(12*N+29)'};
%Last constraint parameter substitution
for i = 1:2
    subsvec1(i+82)  = a(i+2);
    subsvec2(i+82)  = {['X12Np' num2str(i+29)]};
    subsvec3(i+82)  = {['X(12*N+' num2str(i+29) ')']};
end
%Substitution of Model Constants
subsvec1(85:91) = {k,etao,d,m,I,g,vo};
subsvec2(85:91) = {k,etao,d,m,I,g,vo};
subsvec3(85:91) = {'c.k','c.etao','c.d','c.m','c.I','c.g','c.vo'};
```

```
%-------------------------------------------------------------------------
% Nondimensionalization Vector for the Residual
%-------------------------------------------------------------------------
  dimstate      = [etao; etao; 1; vo; vo; vo/etao];
  dimcoestate   = 1./dimstate;
  kappao = [etao/vo; sqrt(vo); sqrt(etao/vo); 1; 1/etao; 1/vo; vo/etao;...
            dimcoestate; dimstate];

  kappai = [dimcoestate; dimstate];

  kappaf = [dimstate; 1/(etao^2); 1/vo; vo/etao; etao/vo; sqrt(vo);...
            sqrt(etao/vo)];

  switch calc
      case 1
        % The first part of the residual vector consists of the boundary
        % conditions at the touchdown time.
          R(1,1)     = subs(kappao(1).*(Hc-Gtc),subsvec1,subsvec2,0);
          R(2:3,1)   = subs(kappao(2:3).*Ga(1:2).',subsvec1,subsvec2,0);
          R(4,1)     = subs(kappao(4).*Gbeta,subsvec1,subsvec2,0);
          R(5:7,1)   = subs(kappao(5:7).*chi,subsvec1,subsvec2,0);
          R(8:13,1)  = subs(kappao(8:13).*theta,subsvec1,subsvec2,0);
          R(14:19,1) = subs(kappao(14:19).*(lambdac+Gyc.'),subsvec1,...
                            subsvec2,0);

        % The first section of the Hessian.
          varvec = [];
          for i = 1:25
              varvec = [varvec;sym(['X' num2str(i)])];
          end
          heso = jake(R,varvec);
          delete('Ro.txt')
          diary('Ro.txt')
          Ro    = subs(R,subsvec2,subsvec3,0)
          diary off
          delete('heso.txt')
          diary('heso.txt')
          heso = subs(heso,subsvec2,subsvec3,0)
          diary off
      case 2
        % The middle of the residual vector is defined using a for loop.
        % It consists of the differential constraints applied at each
        % collocation point for all the states and coestates.
          Ri        = subs(kappai.*[Ryi;Rlambdai],subsvec1,subsvec2,0);

        % The middle section of the residual is dependent on only the
        % states and coestates at the endpoints of each segment and the the
        % ground phase time, tn, known as X(14) in the collocation state
        % vector.  A column vector of the derivatives of this part of the
        % residual with respect to tn will be made and a jacobian will be
        % made of this part of the residual with respect to the endpoint
        % state.
          varvec = sym('X12Np29');
          hestni = jake(Ri,varvec);
          varvec = [];
          for i = 1:24
              varvec = [varvec;sym(['X12ip' num2str(i+1)])];
          end
          hesi   = jake(Ri,varvec);
```

```
    delete('Ri.txt')
    diary('Ri.txt')
    Ri     = subs(Ri,subsvec2,subsvec3,0)
    diary off
    delete('hestni.txt')
    diary('hestni.txt')
    hestni = subs(hestni,subsvec2,subsvec3,0)
    diary off
    delete('hesi.txt')
    diary('hesi.txt')
    hesi   = subs(hesi,subsvec2,subsvec3,0)
    diary off
case 3
  % The last part of the residual vector consists of the boundary
  % conditions at the liftoff time.
    Rf(1:6,1)   = subs(kappaf(1:6).*(lambdaf-Gyf.'),subsvec1,...
                                                    subsvec2,0);
    Rf(7:9,1)   = subs(kappaf(7:9).*psi,subsvec1,subsvec2,0);
    Rf(10,1)    = subs(kappaf(10).*Hf,subsvec1,subsvec2,0);
    Rf(11:12,1) = subs(kappaf(11:12).*Ga(3:4).',subsvec1,subsvec2,0);
  % Last part of the Hessian.
    varvec = [];
    for i = 1:18
        varvec = [varvec;sym(['X12Np' num2str(i+13)])];
    end
    hesf   = jake(Rf,varvec);
    delete('Rf.txt')
    diary('Rf.txt')
    Rf     = subs(Rf,subsvec2,subsvec3,0)
    diary off
    delete('hesf.txt')
    diary('hesf.txt')
    hesf   = subs(hesf,subsvec2,subsvec3,0)
    diary off
otherwise
  %--------------------------------------------------------------------
  % Prepare the Equations of motion for ic_prep3
  %--------------------------------------------------------------------
    clear subsvec1 subsvec2
  % Substitution Defenition
    for i = 1:6
        subsvec1(i)   = y(i);
        subsvec2(i)   = {['y(' num2str(i) ')']};

        subsvec1(i+6) = lambda(i);
        subsvec2(i+6) = {['y(' num2str(i+6) ')']};
    end
  % Substitution of Model Constants
    subsvec1(13:18) = [k,etao,d,m,I,g];
    subsvec2(13:18) = {'c.k','c.etao','c.d','c.m','c.I','c.g'};

    func(1:6,1)     = subs(fg(y),subsvec1,subsvec2,0);
    func(7:12,1)     = subs(-dHdy,subsvec1,subsvec2,0)

  %--------------------------------------------------------------------
  % Prepare equation for nu and -Gyc.' for calculating lambdaf in
  % ic_prep3
  %--------------------------------------------------------------------
    clear subsvec1 subsvec2
  % Substitution Defenition
```

```
          for i = 1:6
            % State at final time substitution
              subsvec1(i)   = yf(i);
              subsvec2(i)   = {['Y(' num2str(i) ',N+1)']};
            % Fixed point state substitution
              subsvec1(i+6)  = yfixed(i);
              subsvec2(i+6)  = {['yfix(' num2str(i) ')']};
          end

        % Final State Weightings
          for i = 2:6
              subsvec1(i+11)  = Q(i,i);
              subsvec2(i+11)  = {['q(' num2str(i) ')']};
          end
        % Substitution of Model Constants
          subsvec1(18:21) = [etao,d,g,vo];
          subsvec2(18:21) = {'c.etao','c.d','c.g','c.vo'};
          nusubbed  = subs(nusolved,subsvec1,subsvec2,0)
          Gyfsubbed = subs(Gyf.',subsvec1,subsvec2,0)
  end
end


function fybar = nondimensionalizer(y)
%==========================================================================
% fybar = nondimensionalizer(y))
%
% This function nondimensionalizes a 6x1 state vector
%
% Inputs:
%   y   6x1 state vector.
%
% Output:
%   fybar  6x1 nondimensionalized state vector.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    global etao vo
    fybar   = [y(1)/etao; y(2)/etao; y(3); y(4)/vo; y(5)/vo; y(6)*etao/vo];
end


function f = ff(y)
%==========================================================================
% f = ff(y)
%
% This calcualates the flight phase equations of motion evaluated at the
% state y.
%
% Inputs:
%   y   6x1 state vector.
%
% Output:
%   f   6x1 vector of the derivative of y.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    global g m I k tc tn
    f   = [y(4);y(5);y(6);0;g;0];
end
```

```
function f = fg(y)
%==========================================================================
% f = fg(y)
%
% This calcualates the ground phase equations of motion evaluated at the
% state y.
%
% Inputs:
%   y   6x1 state vector.
%
% Output:
%   f   6x1 vector of the derivative of y.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    global g m I k tc tn d etao betaTD
    eta = sqrt((y(1)-d*sin(y(3)))^2+(y(2)-d*cos(y(3)))^2);
    f   = [y(4); y(5); y(6); k/m*(etao/eta-1)*(y(1)-d*sin(y(3)));...
                       k/m*(etao/eta-1)*(y(2)-d*cos(y(3)))+g;...
                       d*k/I*(etao/eta-1)*(y(2)*sin(y(3))-y(1)*cos(y(3)))];
end


function f = Hy(y_of_t,lambda_of_t)
%==========================================================================
% Hy(y_of_t,lambda_of_t)
%
% Takes the derivative of the Hamiltonian with respect to the physical
% variables as defined by the code and returns the Hamiltonian derivative
% evaluated for a specific set of parameters y_of_t and lambda_of_t.
%
% Inputs:
%   y_of_t       6x1 state vector.
%
%   lambda_of_t  6x1 coestate vector.
%
% Output:
%   f   Derivative of the hamiltonian evaluated at y_of_t and lambda_of_t.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    global dHdy
    for i = 1:6
        y(i,1)       = sym(['y' num2str(i)],'real');
        lambda(i,1)  = sym(['lambda' num2str(i)],'real');
    end
    f=subs(dHdy,[y;lambda],[y_of_t;lambda_of_t],0);
end


function fy=jake(f,y)
%==========================================================================
% fy=jake(f,y)
%
% Takes the jacobian of a vector.
%
% Inputs:
%   f   an Nx1 vector of functions of the parameters in vector y where N
```

```
%       may equal 1.
%
%   y   an Mx1 vector of parameters where M may equal 1.
%
% Output:
%   fy  an NxM matrix of derivatives where fy(i,j) is the derivative of
%       f(i) with respect to y(j).
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================

    for i=1:length(f)
        for j = 1:length(y)
            fy(i,j)=diff(f(i),y(j));
        end
    end
end
```

A.2 residual_formatter

The script residual_formatter formatted the residual output of euler_param_calc_ad1 to be pasted into a residual code which calculated the residual for a given collocation state.

```
%==========================================================================
% residual_formatter
%
% This script formats symbolic column vector MATLAB outputs written to a
% text file.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
%This script formats stuff
clear all
width_of_line = [58,61,65];

if(1)
% start pre-search
    frid = fopen('hestni.txt', 'r');
    fwid = fopen('temp.txt', 'w');
    subsvec1 = {'yo(','q(','yfix('};
    subsvec2 = {'c.yo(','c.q(','c.yfix('};
    while(~feof(frid))
        A = fread(frid, 1, 'char');
        i=1;
        whole_file = [];
        while (~feof(frid))&&(A~=')')&&(A~=13)
            whole_file(i) = char(A);
            A = fread(frid, 1, 'char');
            i = i + 1;
        end
        if ~feof(frid)
            whole_file(i) = char(A);
        end
        whole_file = char(whole_file);
        for i = 1:3
```

```
            whole_file = strrep(whole_file,subsvec1(i),subsvec2(i));
            whole_file = whole_file{1};
        end
        fwrite(fwid,whole_file,'char');
    end
    fclose('all');
% end pre-search
end

frid = fopen('temp.txt', 'r');
fwid = fopen('hestni_f.txt', 'w');

operators = {'^', '/', '*', '-', '+', '(', ',','{'};

in_vector = 0;
after_eq_sgn = 0;
col_count = 0;
eq_count = 0;
stack_count = 1;
burst = [];

while(~feof(frid))
    A = fread(frid, 1, 'char');
    if in_vector
        if after_eq_sgn
            if isstrprop(A, 'wspace')
            else
                after_eq_sgn = 0;
                col_count    = 1;
                fwrite(fwid,A,'char');
            end
        elseif stack_count~=1|A==char(13)
            stack(stack_count) = A;
            if stack_count == 4
                fwrite(fwid,burst,'char');
                burst = [];
                if stack == [char(13)  char(10) char(32) char(13)]
                    fprintf(fwid,'];\r\n\r\n\r\n');
                    in_vector = 0;
                else
                    fprintf(fwid,';...\r\n\r\n');
                    for i = 1:(72-width_of_line(eq_count))
                        fwrite(fwid,' ','uchar');
                    end
                    if ~isstrprop(A, 'wspace')
                        fwrite(fwid,A,'char');
                        col_count    = 1;
                    end
                end
                col_count = 0;
                stack_count = 1;
            else
                stack_count = stack_count+1;
            end
        elseif isstrprop(A, 'wspace')
        elseif isempty(strmatch(char(A),operators,'exact'))
            burst = [burst A];
        else
            burst = [burst A];
            if col_count+length(burst) >= width_of_line(eq_count)
```

```
                fprintf(fwid,'...\r\n');
                for i = 1:(72-width_of_line(eq_count))
                     fwrite(fwid,' ','uchar');
                end
                col_count = 0;
            end
            col_count = col_count+length(burst);
            fwrite(fwid,burst,'char');
            burst = [];
        end
    else
        if isstrprop(A, 'wspace')
        elseif A=='='
            in_vector    = 1;
            after_eq_sgn = 1;
            eq_count     = eq_count+1;
            fprintf(fwid,' = [');
        else
            fwrite(fwid,A,'char');
        end
    end
end
fclose('all');
```

A.3 hessian_ formatter

The script hessian_formatter formatted the Hessian output of euler_param_calc_ad1 to be pasted into a Hessian code which calculated the Hessian for a given collocation state.

```
%=========================================================================
% hessian_formatter
%
% This script formats symbolic matrix MATLAB outputs written to a text
% file.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
%This script formats stuff
clear all
width_of_line = [62,61,65];
if(1)
%start pre-search
    frid = fopen('hesf.txt', 'r');
    fwid = fopen('tempf.txt', 'w');
    subsvec1 = {'yo(','q(','yfix('};
    subsvec2 = {'c.yo(','c.q(','c.yfix('};
    while(~feof(frid))
        A = fread(frid, 1, 'char');
        i=1;
        whole_file = [];
        while (~feof(frid))&&(A~=')')&&(A~=13)
            whole_file(i) = char(A);
            A = fread(frid, 1, 'char');
            i = i + 1;
        end
```

```
        if ~feof(frid)
            whole_file(i) = char(A);
        end
        whole_file = char(whole_file);
        for i = 1:3
            whole_file = strrep(whole_file,subsvec1(i),subsvec2(i));
            whole_file = whole_file{1};
        end
        fwrite(fwid,whole_file,'char');
    end
    fclose('all');
%end pre-search
end
frid = fopen('tempf.txt', 'r');
fwid = fopen('hesf_f.txt', 'w');

operators = {'^', '/', '*', '-', '+', '(', ',', '{'};

in_vector = 0;
after_eq_sgn = 0;
col_count = 0;
eq_count = 0;
stack_count = 1;
burst = [];

while(~feof(frid))
    A = fread(frid, 1, 'char');
    if in_vector
        if after_eq_sgn
            if isstrprop(A, 'wspace')
            else
                after_eq_sgn = 0;
                col_count    = 1;
                fwrite(fwid,A,'char');
            end
        elseif stack_count~=1|A==']'
            stack(stack_count) = A;
            if stack_count == 4
                fwrite(fwid,burst,'char');
                burst = [];
                if stack == [']' char(13) char(10) '[']
                    fprintf(fwid,';...\r\n\r\n');
                    for i = 1:(72-width_of_line(eq_count))
                        fwrite(fwid,' ','uchar');
                    end
                else
                    fprintf(fwid,'];\r\n\r\n\r\n');
                    in_vector = 0;
                end
                col_count = 0;
                stack_count = 1;
            else
                stack_count = stack_count+1;
            end
        elseif isstrprop(A, 'wspace')
        elseif isempty(strmatch(char(A),operators,'exact'))
            burst = [burst A];
        else
            burst = [burst A];
            if col_count+length(burst) >= width_of_line(eq_count)
```

```
                fprintf(fwid,'...\r\n');
                for i = 1:(72-width_of_line(eq_count))
                     fwrite(fwid,' ','uchar');
                end
                col_count = 0;
            end
            col_count = col_count+length(burst);
            fwrite(fwid,burst,'char');
            burst = [];
        end
    else
        if isstrprop(A, 'wspace')
        elseif A=='='
        elseif A=='['
            in_vector    = 1;
            after_eq_sgn = 1;
            eq_count     = eq_count+1;
            fprintf(fwid,' = [');
        else
            fwrite(fwid,A,'char');
        end
    end
end
fclose('all');
```

APPENDIX B

Control Scheme Code

The control scheme was set up to run separately from the simulation so as to simulate its use on a real system. The main code, collocation4, is called by the simulation at the end of the ground phase. It returns its choice of leg angles as well as the time of touchdown to the simulation and the flight phase simulation is started.

B.1 collocation4

The collaction4 function calculates the optimal touchdown angle from lift off conditions. It calls many sub-functions to accomplish this task. These are presented in subsequent sections.

```
function [beta, tc] = collocation4(yo,c)
%=========================================================================
% [beta, tc] = collocation4(yo,c)
%
% This function calculates the optimal touchdown for the SLIP based on the
% lift off conditons.
%
% Inputs:
%   yo    6x1 lift off conditions vector
%         [x; z; theta; xdot; zdot; thetadot];
%
%   c     The model constants.  These will stay constant throughout the
%         applicatoin of the control.  See rb_slip_sim for structure
%         explanation.
%
% Outputs:
%   beta    The optimal touchdown angle
%
%   tc      The optimal touchdown time
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    c.yo    = yo;
    c.vo    = sqrt(c.yo(4)^2+c.yo(5)^2);
    c.nondim = [1/c.etao;1/c.etao;1;1/c.vo;1/c.vo;c.etao/c.vo];
    c.max_error      = 1e-6;
  % Program Constants
```

```
  N = 5
  h = 1/N;
  itteration_max   = 100;
  max_small_lambda = 8;
% Progam Counters
  lambda_too_small  = 0;
  refine_count     = 0;
  i                = 1;
% Find General Area of Minimum Through Several Short Simulations
% Use Initial Conditions of best short Simulation.
% Initial Conditions
  rdat.refine = 0;
  [X{i},rdat] = ic_prep5(N,c,rdat);
% Calculate Residual
  R{i}    = residual8(X{i},N,c);
% Square of the norm of the residual is an indication of convergence.
  RTR = (R{i}.'*R{i})/2;  %Square of norm.
  fprintf('RTR is %9.8g. Beta is %9.8g.\n',RTR,X{i}(4))
% Test if routine has converged
  not_done =  unsatesfactroy(R{i},c);
% Start itterations
  while(not_done & i<=itteration_max)
      refine_time = 1;
    % Runs iteration once unless the initial conditions had to be refined
      while refine_time
          refine_time = 0;
        % Compute the Hessian analytically
          dRa = analytical_hessian1(X{i},N,c);
        % Newton step taken
          [X{i+1},R{i+1},lambda] = gc_newton(dRa,R{i},X{i},N,c);
        % If step is too small, take note.
          if lambda < 1e-3
              lambda_too_small = lambda_too_small + 1;
              fprintf('%i small steps until refine.\n',...
                                    max_small_lambda-lambda_too_small+1)
            % If too many small steps were taken, refine ic's
              if lambda_too_small > max_small_lambda
                  refine_time      = 1;
                  lambda_too_small = 0;
                  rdat.refine      = 1;
                % Recompute initial conditions for more leg angles
                  [X{i},rdat]      = ic_prep5(N,c,rdat);
                % Recompute residual with better initial conditions
                  R{i}             = residual8(X{i},N,c);
                  RTR = (R{i}.'*R{i})/2;  %Square of norm.
                  fprintf('Refining initial conditions.\n')
                  fprintf('RTR is %9.8g. Beta is %9.8g.\n',RTR,X{i}(4))
              end
          end
      end
      i = i+1;
    % Sets tc to 0 when it is negative
      if X{i}(1)<0
         X{i}(1) = 0;
         X{i}(3) = 0;
         X{i}(7) = X{i}(20)*X{i}(17)+X{i}(21)*X{i}(18)+X{i}(22)*...
                   X{i}(19)+X{i}(24)*c.g-X{i}(9)*(-c.yo(5)-c.g*...
                   X{i}(1))+X{i}(10)*c.yo(6)+X{i}(12)*(c.g);
         R{i}    = residual8(X{i},N,c);
         disp('Switching to tc = 0')
```

```
        end
      % Since angles of greater than 2pi are really angles of less than 2pi
      % plus a full rotation, the full rotation is taken out.
        if X{i}(4)>=2*pi||X{i}(4)<0
           X{i}(4) = mod(X{i}(4),2*pi);
           disp('Truncating beta')
        end
      % Report on Progress
        RTR = R{i}'*R{i}/2;
        fprintf('RTR is %9.8f. Beta is %9.8f. Omega is %9.8f.\n',...
                RTR,X{i}(4),lambda);
        not_done =  unsatesfactroy(R{i},c);
    end
    if not_done
      % If Newton routine did not converge in itteration_max itterations
      % use the best guess so far.
        beta = rdat.betas(rdat.min.i)
        tc   = rdat.min.tc
    elseif (X{i}(1) <= 1e-6)
      % Corrects for numirical error if flight time is intended to be zero
        if X{i}(4)<pi/2
            beta = asin((c.d*cos(c.yo(3))-c.yo(2))/c.etao)
        else
            beta = pi-asin((c.d*cos(c.yo(3))-c.yo(2))/c.etao)
        end
        tc = 0
    else
      % If everything goes as planned routine returns the optimal leg angle
        beta = real(X{i}(4))
      % And the optimal flight time
        tc   = X{i}(1);
    end
end


function yes_or_no = unsatesfactroy(R,c)
%=========================================================================
% yes_or_no = unsatesfactroy(R,c)
%
% This function checks to see if the residual meets the tollerance
% requirements to end the routine
%
% Inputs:
%   R     Residual vector.
%
%   c     The model constants.  These will stay constant throughout the
%         applicatoin of the control.  See rb_slip_sim for structure
%         explanation.
%
% Outputs:
%   yes_or_no   If unsatesfactory 1. If satesfactory 0.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    yes_or_no = 0;
  % Test every part of the residual to make sure it is within tolerance
    for j=1:length(R)
        if abs(R(j))> c.max_error
            yes_or_no = 1;
        end
```

```
    end
end
```

B.2 ic_prep5

The function ic_prep5 is called in collocation4 to prepare an initial guess of the collocation state vector.

```
function [X,rdat] = ic_prep5(N,c,rdat)
%=========================================================================
% [X,rdat] = ic_prep5(N,c)
%
% This function sets up the necessary variables for collocation4.
% It makes an initial guess at the values of the parameters in the X
% vector. These initial guesses are called the initial conditions of the
% collocation scheme even though they are not just for the state at the
% initial time.
%
% This funciton uses a folder of functions ic_prep.  It cannot run without
% them.
%
% Inputs:
%   N     Number of segments into which the collocation scheme will be
%         broken.
%
%   c     The model constants.  These will stay constant throughout the
%         applicatoin of the control.  See rb_slip_sim for structure
%         explanation.
%
%   rdat  This is a structure containing everything ic_prep5 needs to know
%         about its previous calls.
%           rdat.betas  Is a vector of the angles that have already been
%                       explored by ic_prep
%
%           rdat.min    Is a structure containing data about initial
%                       conditions for the colloaction function that yield
%                       the minimum end cost, J.  It also includes the cost
%                       itself.
%
%               rdat.min.i   This is the index number of the touchdown
%                            angle that yields the minimum cost in rdat.J
%
%               rdat.min.J   Is a vector of the final costs associated with
%                            the angles in rdat.betas
%
%               rdat.min.Y   This is the state of the collocation function
%                            for which the cost is rdat.J
%
%               rdat.min.tc  Is the flight time of the collocation state
%                            rdat.min.Y.
%
%               rdat.min.tn  Is the ground phase time of the collocation
%                            state rdat.min.Y.
%
```

```
%           rdat.refine Is a variable simply telling the ic_prep funciton
%                       if it is creating initial condition for a new
%                       collocation scheme or refining some initial
%                       conditons that have already been developed.
%
% Outputs:
%   X       The initial guess for the collocation state vector.
%
%   rdat    This is a structure containing everything ic_prep5 will need to
%           know if it is called again to refine the initial conditons
%           about this and previous calls this function calls.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================

  % Error Acceptable in the initial conditions
    c.ic_error = 0.0001;
  % Change to the ic_prep directory
    cd ic_prep
    if ~rdat.refine
      % If ic_prep is starting form scratch, it must calculate a range to
      % test in.
        rdat.betas = ic_prep_range_test(c);
        rdat.min = ic_prep_cost_finder(rdat.betas,inf,c,c.nondim);
    else
      % If ic_prep has been called before it mus refine its search for the
      % best initial guess of an optimal leg angle because the original
      % guess converged to a local minimum.
        minn.i = 0;
        while minn.i == 0
            M = length(rdat.betas);
          % Tests leg angles half way in between all the previously tested
          % leg angles and half again as close to to the max and min beta
          % values.
            beta(1)=.25*(rdat.betas(1)-rdat.betas(2))+rdat.betas(1);
            for i = 1:M-1
                beta(i+1) = (rdat.betas(i+1)+rdat.betas(i))/2;
            end
            beta(M+1) = 0.25*(rdat.betas(M)-rdat.betas(M-1))+...
                        rdat.betas(M);
            minn = ic_prep_cost_finder(beta,rdat.min.J,c,c.nondim);
          % Organize Betas.
            betas(1) = beta(1);
            for i = 1:length(rdat.betas)
                betas(2*i)   = rdat.betas(i);
                betas(2*i+1) = beta(i+1);
            end
            rdat.betas = betas;
        end
        rdat.min = minn;
        rdat.min.i = 2*minn.i-1;
    end
    rdat.refine = 0;
    X = ic_final_prep1(N,c,rdat);
    cd ..
end
```

B.2.1 ic_prep_range_test

The function ic_prep_range_test calculates the range in which it would be acceptable to find a leg touchdown angle and proceeds to select the leg angles to try.

```
function beta = ic_prep_range_test(c)
%=========================================================================
% beta = ic_prep_range_test(c)
%
% This function calculates the initial guesses of leg angles that should be
% tested.
%
% Inputs:
%
%   c     The model constants.  These will stay constant throughout the
%         applicatoin of the control.  See rb_slip_sim for structure
%         explanation.
%
% Outputs:
%   beta  Vector of leg angles to try.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    M = 7;
  % Time of Max height of leg attachment point (Min Time)
    tmin    = -2*(c.yo(5)+c.d*c.yo(6)*sin(c.yo(3)))/...
              ((c.g+c.d*c.yo(6)^2*cos(c.yo(3)))+...
              sqrt((c.g+c.d*c.yo(6)^2*cos(c.yo(3)))^2+...
              2*c.d*c.yo(6)^3*sin(c.yo(3))*...
              (c.yo(5)+c.d*c.yo(6)*sin(c.yo(3)))));
    if tmin < 0
        tmin = 0;
    end
    hmax = -c.yo(2)-c.yo(5)*tmin-1/2*c.g*tmin^2+...
               c.d*cos(c.yo(3)+c.yo(6)*tmin)-c.etao;
  % Determin the bounds for beta.
    if c.yo(4)>0
  % Time of Min height of leg attachment point (Max Time)
    tmax = (-(c.yo(5)+c.d*c.yo(6)*sin(c.yo(3)))+...
           sqrt((c.yo(5)+c.d*c.yo(6)*sin(c.yo(3)))^2+...
           2*(c.g+c.d*c.yo(6)^2*cos(c.yo(3)))*...
           (c.d*cos(c.yo(3))-c.yo(2))))/...
           (c.g+c.d*c.yo(6)^2*cos(c.yo(3)));
    if hmax < 0
        beta_max = asin((-2*c.yo(2)-2*c.yo(5)*tmin-c.g*tmin^2+...
                   2*c.d*cos(c.yo(3)+c.yo(6)*tmin))/(2*c.etao));
    else
  % Find the time at which the leg touch at touchdown is
  % perpendicular to the direction of the touchdown velocity.
    tbetamax = RegulaFalsi(tmin,tmax,@MaxBetat,c);
    beta_max = pi-asin((-c.yo(2)-c.yo(5)*tbetamax-1/2*c.g*...
               tbetamax^2+c.d*cos(c.yo(3)+c.yo(6)*tbetamax))...
               /c.etao);
    end
  % Find the time at which the leg touch at touchdown is
  % parallel to the direction of the touchdown velocity.
    tbetamin  = RegulaFalsi(tmin,tmax,@MinBetat,c);
```

```
if tbetamin < 0
    tbetamin = 0;
end
beta_min  = asin((-c.yo(2)-c.yo(5)*tbetamin-1/2*c.g*...
            tbetamin^2+c.d*cos(c.yo(3)+c.yo(6)*...
            tbetamin))/c.etao);
else
    if hmax < 0
        beta_min = pi-asin((-2*c.yo(2)-2*c.yo(5)*tmin-c.g*tmin^...
                   2+2*c.d*cos(c.yo(3)+c.yo(6)*tmin))/...
                   (2*c.etao));
    else
        beta_min = pi/2;
    end
    beta_max = pi;
end
mid = pi/2;
test_pi_over_2 = 1;
if test_pi_over_2
  % Cost Test Aligns Spread of Beta's to hit pi/2 in the middle.
    if beta_min >= beta_max
        beta(1) = beta_min;
        M = 1;
    elseif c.yo(4)>0 && beta_max>mid
        Mf = ceil((M-1)/2);
        Mb = floor((M-1)/2);
        scalef = (mid-beta_min)/(Mf+.5);
        scaleb = (beta_max-mid)/(Mb+.5);
        beta(1) = beta_min+scalef*.5;
        for i=2:Mf+1
            beta(i) = beta(i-1)+scalef;
        end
        for i=Mf+2:M
            beta(i) = beta(i-1)+scaleb;
        end
    else
        scale   = (beta_max-beta_min)/M;
        beta(1) = beta_min + scale*.5;
        for i=2:M
            beta(i) = beta(i-1)+scale;
        end
    end
else
  % Cost Test Aligns Spread of Beta's to hit before and after pi/2.
    if beta_min >= beta_max
        beta(1) = beta_min;
        M = 1;
    elseif c.yo(4)>0 && beta_max>mid
        Mf = ceil(M/2);
        Mb = floor(M/2);
        scalef = (mid-beta_min)/(Mf);
        scaleb = (beta_max-mid)/(Mb);
        beta(1) = beta_min+scalef*.5;
        for i=2:Mf
            beta(i) = beta(i-1)+scalef;
        end
        beta(Mf+1) = mid+scaleb*.5;
        for i=Mf+2:M
            beta(i) = beta(i-1)+scaleb;
        end
```

```
        else
            scale   = (beta_max-beta_min)/M;
            beta(1) = beta_min + scale*.5;
            for i=2:M
                beta(i) = beta(i-1)+scale;
            end
        end
    end
end

function tmaybe = RegulaFalsi(tmin,tmax,func,c)
%==========================================================================
% tmaybe = RegulaFalsi(tmin,tmax,func,c)
%
% This function finds the input value, tmaybe, for which func is 0.
%
% Inputs:
%
%   tmin   The minimum input value for which func could be 0.
%
%   tmax   The maximum input value for which func could be 0.
%
%   func   A function handle for a function.  The input of this function
%          must be found such that the function's value is 0.
%
%   c      The model constants.  These will stay constant throughout the
%          applicatoin of the control.  See rb_slip_sim for structure
%          explanation.
%
% Outputs:
%   tmaybe  The input value for which func is 0.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    ftmax   = func(tmax,c);
    ftmin   = func(tmin,c);
    ftmaybe = ftmin;
    tmaybe  = tmin;
    while  abs(ftmaybe)>c.ic_error
        gprime  = (ftmax-ftmin)/(tmax-tmin);
        tmaybe  = tmax-ftmax/gprime;
        ftmaybe = func(tmaybe,c);
        if ftmaybe < 0
            tmax  = tmaybe;
            ftmax = ftmaybe;
        else
            tmin  = tmaybe;
            ftmin = ftmaybe;
        end
    end
end

function f = MinBetat(t,c)
%==========================================================================
% f = MinBetat(t,c)
%
% This function is 0 when the input is the time at which the system would
% touchdown if it were touching down at the minimum leg touchdown angle.
%
% Inputs:
```

```
%
%   t   Input time.
%
%   c   The model constants.  These will stay constant throughout the
%       applicatoin of the control.  See rb_slip_sim for structure
%       explanation.
%
% Outputs:
%   f   Function value.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    if c.yo(4) == 0
        f = (-2*c.yo(2)-2*c.yo(5)*t-c.g*t^2+2*c.d*...
            cos(c.yo(3)+c.yo(6)*t))/(2*c.etao)-1;
    else
        f = (-2*c.yo(2)-2*c.yo(5)*t-c.g*t^2+2*c.d*...
            cos(c.yo(3)+c.yo(6)*t))/(2*c.etao)-...
            (c.yo(5)+c.g*t)/sqrt(c.yo(4)^2+(c.yo(5)+c.g*t)^2);
    end
end

function f = MaxBetat(t,c)
%==========================================================================
% f = MaxBetat(t,c)
%
% This function is 0 when the input is the time at which the system would
% touchdown if it were touching down at the maximum leg touchdown angle.
%
% Inputs:
%
%   t   Input time.
%
%   c   The model constants.  These will stay constant throughout the
%       applicatoin of the control.  See rb_slip_sim for structure
%       explanation.
%
% Outputs:
%   f   Function value.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    f = (-2*c.yo(2)-2*c.yo(5)*t-c.g*t^2+2*c.d*...
        cos(c.yo(3)+c.yo(6)*t))/(2*c.etao)-...
        c.yo(4)/sqrt(c.yo(4)^2+(c.yo(5)+c.g*t)^2);
end
```

B.2.2 ic_prep_cost_finder

The ic_prep_cost_finder function determines the unconstrained cost associated with each leg angle suggested by ic_prep_range_test and passes on the leg angle with the lowest cost associated with it.

```
function minn = ic_prep_cost_finder(beta,jmin,c,nondim)
%==========================================================================
% minn = ic_prep_cost_finder(beta,jmin,c,nondim)
```

```
%
% This function selects the leg angle out of an input vector of leg angles
% that produces the lowest cost as determined by the unconstrained cost
% function.  It then records information about its findings.
%
% Inputs:
%
%   beta    A vector of leg angles to test.
%
%   jmin    The best cost found so far.
%
%   c       The model constants.  These will stay constant throughout the
%           applicatoin of the control.  See rb_slip_sim for structure
%           explanation.
%
%   nondim  6x1 vector of nondimensionalization parameters for the states.
%
% Outputs:
%   minn    A structure containing data about about the best beta chosen.
%           The elements are:
%             minn.i   The index number of the best leg angle chosen in the
%                      vector.
%
%             minn.J   The unconstrained cost associated with the best
%                      choice of leg angle.
%
%             minn.Y   6x2 touchdown state and end state associated with
%                      the best leg angle found concatenated together in
%                      that order.
%
%             minn.tc  The touchdown time associated with the best leg
%                      angle found.
%
%             minn.tn  The ground phase time associated with the best leg
%                      angle found.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================

  % Angles to test, Cost to test against,
  % For each new angle find the cost.
    minn.J = jmin;
    minn.i = 0;
    for i = 1:length(beta)
      % Approximate the touchdown time.
        tc(i)   = (-(c.yo(5)+c.d*c.yo(6)*sin(c.yo(3)))+...
                  sqrt((c.yo(5)+c.yo(6)*c.d*sin(c.yo(3)))^2+...
                  2*(c.g+c.yo(6)^2*c.d*cos(c.yo(3)))*...
                  (c.d*cos(c.yo(3))-c.yo(2)-c.etao*sin(beta(i)))))/...
                  (c.g+c.yo(6)^2*c.d*cos(c.yo(3)));
      % Use the Approximated touchdown time to get the touchdown state.
        Y{i}(3,1) = c.yo(3)+c.yo(6)*tc(i);
        Y{i}(1,1) = c.d*sin(Y{i}(3,1))-c.etao*cos(beta(i));
        Y{i}(2,1) = c.yo(2)+c.yo(5)*tc(i)+1/2*c.g*tc(i)^2;
        Y{i}(4,1) = c.yo(4);
        Y{i}(5,1) = c.yo(5)+c.g*tc(i);
        Y{i}(6,1) = c.yo(6);
      % Loosely approximated the ground phase time so the time step for the
      % rk4 is not too big.
        tn_est    = c.etao/c.vo;
```

```
      % For approximating tn and Yn use runge_N steps.
        runge_N = 50;
      % Make sure runge_tn_find runs once
        true_ratio = 0;
      % Rerun runge_tn_find if the time step was too large last time
        while true_ratio < .5
            [Y{i}(:,2),tn(i)] =...
                        runge_tn_find(Y{i}(:,1),tn_est,runge_N,c);
            true_ratio = tn(i)/tn_est;
            tn_est = tn(i)*1.5;
        end
      % Calculate Cost of end State
        deltaYbar = [Y{i}(:,2)-c.yfix].*nondim;
        J(i)      = deltaYbar.'*(c.q.*deltaYbar);
        if J(i)<minn.J
            minn.i   = i;
            minn.J   = J(i);
            minn.Y   = Y{i};
            minn.tc  = tc(i);
            minn.tn  = tn(i);
        end
    end
end


function [endspresh,tn] = runge_tn_find(Y,tn_est,N,c)
%=========================================================================
% [endspresh,tn] = runge_tn_find(Y,tn_est,N,c)
%
% This function uses a 4th order runge-kutta scheme to numerically
% integrate forward in time until the leg is no longer in compression.  It
% procedes finds the exact lift off state and returns it with the ground
% phase time.
%
% Inputs:
%
%   Y       6x1 touchdown state vector.
%
%   tn_est  A guess of what the final time will be so a good step size can
%           be picked.
%
%   N       The number of time steps to shoot for.
%
%   c       The model constants.  These will stay constant throughout the
%           applicatoin of the control.  See rb_slip_sim for structure
%           explanation.
%
% Outputs:
%   endsprech  6x1 lift off state vector.
%
%   tn         Ground phase time.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================

  % Step size
    h = tn_est/N;
  % Make sure integration starts
    eta = 0;
    etaprime = -1;
```

```
  % Initialize index
    i=1;
  % Start time
    tn = 0;
    while (eta <= c.etao || etaprime < 0)
      % Take Step
        [Y(:,i+1),tn] = rkstep(Y(:,i),tn,h,'half',c);
        i = i+1;
      % Calculate conditions for end
        eta = sqrt((Y(1,i)-c.d*sin(Y(3,i)))^2+(Y(2,i)-c.d*cos(Y(3,i)))^2);
        etaprime = 1/(2*eta)*...
            (2*(Y(1,i)-c.d*sin(Y(3,i)))*(Y(4,i)-c.d*cos(Y(3,i))*Y(6,i))+...
             2*(Y(2,i)-c.d*cos(Y(3,i)))*(Y(5,i)+c.d*sin(Y(3,i))*Y(6,i)));
    end
  % Lower bracket end condtion
    eta2 = sqrt((Y(1,i-1)-c.d*sin(Y(3,i-1)))^2+(Y(2,i-1)-c.d*...
           cos(Y(3,i-1)))^2);
  % False Position method to find end.
    gprime = (eta-eta2)/h;
    h   = -(eta-c.etao)/gprime;
    while (1-eta/c.etao)^2 > c.ic_error^2
        [Y(:,i+1),tn] = rkstep(Y(:,i),tn,h,'half',c);
        eta2 = sqrt((Y(1,i+1)-c.d*sin(Y(3,i+1)))^2+(Y(2,i+1)-c.d*...
               cos(Y(3,i+1)))^2);
        i   = i+1;
        gprime = (eta2-eta)/h;
        eta=eta2;
        h   = -(eta-c.etao)/gprime;
    end
    endspresh = Y(:,i);
end
```

B.2.3 ic_final_prep1

The function ic_final_prep1 uses the best leg angle determined by ic_prep_cost_finder and calculates guesses for the values of all the collocation states based on it.

```
function X = ic_final_prep1(N,c,rdat)
%=========================================================================
% X = ic_final_prep1(N,c,rdat)
%
% This function uses the best leg angle found, and the lift off state and
% ground phase time associated with it to determine estimates for the rest
% of the parameters in the collocation scheme.
%
% Inputs:
%   N     Number of segments into which the collocation scheme will be
%         broken.
%
%   c     The model constants.  These will stay constant throughout the
%         applicatoin of the control.  See rb_slip_sim for structure
%         explanation.
%
%   rdat  This is a structure containing everything ic_final_prep needs to
%         know about the findings of the current call the ic_prep function
```

```
%         and its previous calls.
%
% Outputs:
%   X     The initial guess for the collocation state vector.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================

    betaTD   = rdat.betas(rdat.min.i);
    Y(:,1)   = rdat.min.Y(:,1);
    Y(:,N+1) = rdat.min.Y(:,2);
    tc       = rdat.min.tc;
    tn       = rdat.min.tn;
  % Calculate nu from end states
    nu(1,1)   = (Y(5,N+1)*(c.vo)^2*c.q(2)*Y(2,N+1)-Y(5,N+1)*(c.vo)^2*...
                 c.q(2)*c.yfix(2)+Y(6,N+1)*(c.etao)^2*(c.vo)^2*c.q(3)*Y(...
                 3,N+1)-Y(6,N+1)*(c.etao)^2*(c.vo)^2*c.q(3)*c.yfix(3)+...
                 c.q(5)*(c.g)*(c.etao)^2*Y(5,N+1)-c.q(5)*(c.g)*(c.etao)^...
                 2*c.yfix(5))/(c.etao)^2/(c.vo)^2/((Y(4,N+1)-Y(6,N+1)*(...
                 c.d)*cos(Y(3,N+1)))*Y(1,N+1)-Y(4,N+1)*(c.d)*sin(Y(3,N+...
                 1))+Y(5,N+1)*Y(2,N+1)-Y(5,N+1)*(c.d)*cos(Y(3,N+1))+Y(6,...
                 2)*(c.d)*sin(Y(3,N+1))*Y(2,N+1));

    nu(2:3,1) = [0;0];
  % Plug nu into Gyf.' to find the final coestates
    Y(7:12,N+1) = [nu(1)*(-2*Y(1,N+1)+2*(c.d)*sin(Y(3,N+1)));...

                   2*(Y(2,N+1)-c.yfix(2))/(c.etao)^2*c.q(2)+nu(1)*(-2*...
                   Y(2,N+1)+2*(c.d)*cos(Y(3,N+1)));...

                   2*(Y(3,N+1)-c.yfix(3))*c.q(3)+nu(1)*(2*(Y(1,N+1)-...
                   c.d*sin(Y(3,N+1)))*(c.d)*cos(Y(3,N+1))-2*(...
                   Y(2,N+1)-c.d*cos(Y(3,N+1)))*(c.d)*sin(Y(3,N+1)));...

                   2*(Y(4,N+1)-c.yfix(4))/(c.vo)^2*c.q(4);...

                   2*(Y(5,N+1)-c.yfix(5))/(c.vo)^2*c.q(5);...

                   2*(Y(6,N+1)-c.yfix(6))*(c.etao)^2/(c.vo)^2*c.q(6)];

  % Set the time step to go backwards
    h       = -tn/N;
    ti(N+1) = tn;
  % Integrate backward until a step before the touch down time, tc
    for i=N:-1:2
        [Y(:,i),ti(i)] = rkstep(Y(:,i+1),ti(i+1),h,'full',c);
    end
    [temp1,ti(1)] = rkstep(Y(:,2),ti(2),h,'full',c);
  % Fill in the touchdown coestates in the Y matrix (we already have
  % the state).
    Y(7:12,1) = temp1(7:12);

  % Using A polynominal to carry it over the asimptote.
    max_upsilon = 50;
    beta_region = pi/2-atan(max_upsilon);
    min_reg = pi/2-beta_region;
    max_reg = pi/2+beta_region;
    if min_reg <= betaTD && betaTD <= max_reg
        K  = [1 min_reg min_reg^2 min_reg^3; 0 1 2*min_reg 3*min_reg^2;...
              1 max_reg max_reg^2 max_reg^3; 0 1 2*max_reg 3*max_reg^2];
```

```
        Cs = inv(K)*[ max_upsilon;1/(cos(min_reg))^2;...
                     -max_upsilon;1/(cos(max_reg))^2];
        upsilon(1) = Y(7,1)*[1 betaTD betaTD^2 betaTD^3]*Cs;
    else
        upsilon(1) = Y(7,1)*tan(betaTD);
    end
  % Gyc.' = -lambdac
    xi(1:6,1) = -Y(7:12,1)+[0;upsilon(1);upsilon(1)*c.d*sin(Y(3,1))-...
                 Y(7,1)*c.d*cos(Y(3,1));0;0;0];
  % Find upsilon(2)
    upsilon(2,1) = 0;
  % Find a's
    asqrd(1) = -(Y(1,1)-c.d*sin(Y(3,1)))*(Y(4,1)-c.d*cos(Y(3,1))*Y(6,1))...
               -(Y(2,1)-c.d*cos(Y(3,1)))*(Y(5,1)+c.d*sin(Y(3,1))*Y(6,1));
    asqrd(2) = tc;
    asqrd(3) = (Y(1,N+1)-c.d*sin(Y(3,N+1)))*...
               (Y(4,N+1)-c.d*cos(Y(3,N+1))*Y(6,N+1))+...
               (Y(2,N+1)-c.d*cos(Y(3,N+1)))*...
               (Y(5,N+1)+c.d*sin(Y(3,N+1))*Y(6,N+1));
    asqrd(4) = tn;
    for i = 1:4
        if asqrd(i) > 0
            a(i,1) = sqrt(asqrd(i));
        else
            a(i,1) = 0;
        end
    end
  % find Best upsilon(3)
    upsilon(3,1) = Y(7,1)*Y(4,1)+Y(8,1)*Y(5,1)+Y(9,1)*Y(6,1)+Y(11,1)*...
                   c.g-xi(2)*(-c.yo(5)-c.g*tc)+xi(3)*c.yo(6)+xi(5)*c.g;
    res1 = 1/(c.vo)*(c.etao)*upsilon(3);
    res2 = -2*(1/(c.vo)*(c.etao))^(1/2)*a(2)*upsilon(3);...
    if abs(res2) >= abs(res1)
        upsilon(3,1) = 0;
    end
  % Now asign the variables to the collocation state vector X
    X = [tc;a(1:2);betaTD;upsilon;xi];
    for i = 1:N+1
        X(12*i+2:12*i+13) = Y(:,i);
    end
    X(12*N+26:12*N+31) = [nu; tn; a(3:4)];
end
```

B.2.4 rk_step

The function rk_step is called in some of the sub-functions of ic_prep5. It moves the system forward of backward the amount specified in its input.

```
function [yplus,tn] = rkstep(y,tn,h,func_style,c)
%=========================================================================
% [yplus,tn] = rkstep(y,tn,h,func_style,c)
%
% This function integrates the equations of motion of the system forward or
% backward in time the amount specified by h.
%
```

```
% Inputs:
%   y           6x1 or 12x1 state or state and costate vector at time tn.
%
%   tn          Time passed since the start of the ground phase.
%
%   h           Time step (positive or negative).
%
%   func_style  String containing directions to either integrate both the
%               states and the costates or just the states.
%
%   c           The model constants.  See rb_slip_sim for structure
%               explanation.
%
% Outputs:
%   yplus       6x1 or 12x1 state or state and costate vector and the new
%               time tn.
%
%   tn          New time passed since the ground phase started.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    if func_style == 'half'
        k1 = h*funcG(y,c);
        k2 = h*funcG(y+k1/2,c);
        k3 = h*funcG(y+k2/2,c);
        k4 = h*funcG(y+k3,c);
    elseif func_style == 'full'
        k1 = h*fullfuncG(y,c);
        k2 = h*fullfuncG(y+k1/2,c);
        k3 = h*fullfuncG(y+k2/2,c);
        k4 = h*fullfuncG(y+k3,c);
    end
    tn = tn+h;
    yplus = y+(k1+2*k2+2*k3+k4)/6;
end


function dy = funcG(y,c)
%==========================================================================
% dy = funcG(y,c)
%
% This function contains the ground phase equations of motion for the
% states.  It evaluates these given the state of the system.
%
% Inputs:
%   y   6x1 state vector.
%
%   c   The model constants.  See rb_slip_sim for structure explanation.
%
% Outputs:
%   dy  6x1 vector of the EOMs evaluated at state y.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    dy = [y(4);...

          y(5);...

          y(6);...
```

```
          c.k/(c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(...
          3)))^2)^(1/2)-1)*(y(1)-c.d*sin(y(3)));...

          c.k/(c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(...
          3)))^2)^(1/2)-1)*(y(2)-c.d*cos(y(3)))+c.g;...

          c.d*(c.k)/(c.I)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*...
          cos(y(3)))^2)^(1/2)-1)*(y(2)*sin(y(3))-y(1)*cos(y(3)))];
end


function dy = fullfuncG(y,c)
%=========================================================================
% dy = fullfuncG(y,c)
%
% This function contains the ground phase equations of motion for the
% states and costates.  It evaluates these given the state of the system.
%
% Inputs:
%   y   12x1 state and costate vector.
%
%   c   The model constants.  See rb_slip_sim for structure explanation.
%
% Outputs:
%   dy  12x1 vector of the EOMs evaluated at state y.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    dy = [y(4);...

          y(5);...

          y(6);...

          c.k/(c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(...
          3)))^2)^(1/2)-1)*(y(1)-c.d*sin(y(3)));...

          c.k/(c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(...
          3)))^2)^(1/2)-1)*(y(2)-c.d*cos(y(3)))+c.g;...

          c.d*(c.k)/(c.I)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*...
          cos(y(3)))^2)^(1/2)-1)*(y(2)*sin(y(3))-y(1)*cos(y(3)));...

          1/2*y(10)*(c.k)/(c.m)*(c.etao)/((y(1)-c.d*sin(y(3)))^2+(y(2)-...
          c.d*cos(y(3)))^2)^(3/2)*(2*y(1)-2*(c.d)*sin(y(3)))*(y(1)-c.d*...
          sin(y(3)))-y(10)*(c.k)/(c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+...
          (y(2)-c.d*cos(y(3)))^2)^(1/2)-1)+1/2*y(11)*(c.k)/(c.m)*(...
          c.etao)/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(3)))^2)^(3/...
          2)*(2*y(1)-2*(c.d)*sin(y(3)))*(y(2)-c.d*cos(y(3)))+1/2*y(12)*...
          (c.d)*(c.k)/(c.I)*(c.etao)/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*...
          cos(y(3)))^2)^(3/2)*(2*y(1)-2*(c.d)*sin(y(3)))*(y(2)*sin(y(...
          3))-y(1)*cos(y(3)))+y(12)*(c.d)*(c.k)/(c.I)*(c.etao/((y(1)-...
          c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(3)))^2)^(1/2)-1)*cos(y(3));...

          1/2*y(10)*(c.k)/(c.m)*(c.etao)/((y(1)-c.d*sin(y(3)))^2+(y(2)-...
          c.d*cos(y(3)))^2)^(3/2)*(2*y(2)-2*(c.d)*cos(y(3)))*(y(1)-c.d*...
          sin(y(3)))-y(11)*(-1/2*(c.k)/(c.m)*(c.etao)/((y(1)-c.d*sin(y(...
          3)))^2+(y(2)-c.d*cos(y(3)))^2)^(3/2)*(2*y(2)-2*(c.d)*cos(y(...
          3)))*(y(2)-c.d*cos(y(3)))+c.k/(c.m)*(c.etao/((y(1)-c.d*sin(y(...
          3)))^2+(y(2)-c.d*cos(y(3)))^2)^(1/2)-1))+1/2*y(12)*(c.d)*(...
```

```
            c.k)/(c.I)*(c.etao)/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(...
            3)))^2)^(3/2)*(2*y(2)-2*(c.d)*cos(y(3)))*(y(2)*sin(y(3))-y(...
            1)*cos(y(3)))-y(12)*(c.d)*(c.k)/(c.I)*(c.etao/((y(1)-c.d*sin(...
            y(3)))^2+(y(2)-c.d*cos(y(3)))^2)^(1/2)-1)*sin(y(3));...

            1/2*y(10)*(c.k)/(c.m)*(c.etao)/((y(1)-c.d*sin(y(3)))^2+(y(2)-...
            c.d*cos(y(3)))^2)^(3/2)*(-2*(y(1)-c.d*sin(y(3)))*(c.d)*cos(y(...
            3))+2*(y(2)-c.d*cos(y(3)))*(c.d)*sin(y(3)))*(y(1)-c.d*sin(y(...
            3)))+y(10)*(c.k)/(c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-...
            c.d*cos(y(3)))^2)^(1/2)-1)*(c.d)*cos(y(3))-y(11)*(-1/2*(c.k)/...
            (c.m)*(c.etao)/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(3)))^...
            2)^(3/2)*(-2*(y(1)-c.d*sin(y(3)))*(c.d)*cos(y(3))+2*(y(2)-...
            c.d*cos(y(3)))*(c.d)*sin(y(3)))*(y(2)-c.d*cos(y(3)))+c.k/(...
            c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(3)))^2)^...
            (1/2)-1)*(c.d)*sin(y(3)))+1/2*y(12)*(c.d)*(c.k)/(c.I)*(...
            c.etao)/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(3)))^2)^(3/...
            2)*(-2*(y(1)-c.d*sin(y(3)))*(c.d)*cos(y(3))+2*(y(2)-c.d*cos(...
            y(3)))*(c.d)*sin(y(3)))*(y(2)*sin(y(3))-y(1)*cos(y(3)))-y(...
            12)*(c.d)*(c.k)/(c.I)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-...
            c.d*cos(y(3)))^2)^(1/2)-1)*(y(2)*cos(y(3))+y(1)*sin(y(3)));...

            -y(7);...

            -y(8);...

            -y(9)];
end
```

B.3 gc_newton

The function gc_newton takes the system of nonlinear equations one step in the Newton direction. It decides the size of the step by making sure that the residual is reduced sufficiently by it. The algorithm was inspired by [24].

```
function [Xp,Rp,lambda1] = gc_newton(dR,R,X,N,c)
%======================================================================
% [Xp,Rp,lambda1] = gc_newton(dR,R,X,N,c)
%
% This function moves a system of nonlinear equations one step in the
% Newton direction.  It decides how far to move in the step by ensuring
% that the residual is decreased sufficiently by the step.
%
% Inputs:
%   dR    Hessian matrix.
%
%   R     Old residual vector.
%
%   X     Old state vector
%
%   N     Number of collocation segments
%
%   c     The model constants.  These will stay constant throughout the
%         applicatoin of the control.  See collocation4 for structure
%         explanation.
```

```
%
% Outputs:
%   Xp       New state vector
%
%   Rp       New residual vector
%
%   lambda1  Newton step size
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    g0      = R'*R/2;
    gp0     = -2*g0;
    alpha   = 1e-4;
    M       = length(R);
    [U,S,V] = svd(dR);
    dRinv   = V*inv(S)*U';
    dX      = -dRinv*R;
    Xp      = X+dX;
    Rp      = residual8(Xp,N,c);
    g1      = Rp'*Rp/2;
    lambda1 = 1;
    if g1 > g0-2*alpha*g0
        lambda2 = lambda1;
        lambda1 = min(max(-gp0/(2*(g1+g0)),0.1*lambda2),0.5*lambda2);
        Xp      = X+lambda1*dX;
        Rp      = residual8(Xp,N,c);
        g2      = g1;
        g1      = Rp'*Rp/2;
        while g1 > g0-2*alpha*lambda1*g0 && lambda1 >= 1e-3
            a = 1/(lambda1-lambda2)*[1/lambda1^2, -1/lambda2^2;...
                        -lambda2/lambda1^2, lambda1/lambda2^2]*...
                        [g1-gp0*lambda1-g0; g2-gp0*lambda2-g0];
            lambda2 = lambda1;
            lambda1 = min(max((-a(2)+sqrt(a(2)^2-3*a(1)*gp0))/(3*a(1)),...
                                                0.1*lambda2),0.5*lambda2);
            Xp      = X+lambda1*dX;
            Rp      = residual8(Xp,N,c);
            g2      = g1;
            g1      = Rp'*Rp/2;
        end
    end
end
```

B.4 Residual and Hessian Codes

The residual and Hessian codes were too long to include in this text. Since they were both written by a series of MATLAB codes those were included in appendix A.

APPENDIX C

Simulation Code

A general simulation code was written which could simulate with model predictive control, with fixed leg angle touchdown, or with a leg angle prescribed by a vector. It could run for a prescribed number of steps or until a certain criterion was met. Various front ends were written to accomplish all the tests that were done in this work.

C.1 rb_slip_sim

The function rb_slip_sim is the general simulation code. It works for both the point mass SLIP and the rigid body SLIP.

```
function [v,vlo,vtd,return_status] = rb_slip_sim(yo,sp,c)
%=========================================================================
% [v,vlo,vtd,return_status] = rb_slip_sim(yo,fp,sp,c)
%
% This function simulates the rigid body SLIP.
%
% Inputs:
%   yo    6x1 lift off conditions vector
%         [x; z; theta; xdot; zdot; thetadot];
%
%   sp    Simulation parameter structure containting:
%           sp.control          Switches control on and off (1/0)
%
%           sp.max_steps        The maximum number of steps the system
%                               takes before stopping.
%
%           sp.max_step_size       Maximum stepsize the simulation will take
%
%           sp.beta             Scalar or vector of leg angles
%
%           sp.beta_const       Determines if the leg angle should be held
%                               fixed at sp.beta(1) for every stride or
%                               change each step.
%
%           sp.end_criterion    Function handle for a function to determine
%                               when to end the simulaiton
%
```

```
%           sp.skip_flight_1    Switch to tell the simulation to skip the
%                               first flight phase
%
%   c     The model constants.  These will stay constant throughout the
%         applicatoin fo the control.  This is a structure containing:
%           c.k       Spring constant
%
%           c.etao    Nominal leg length
%
%           c.d       The distance above the center of mass at which the
%                     spring is attached
%
%           c.m       The mass of the body
%
%           c.I       Iyy moment of inertia
%
%           c.g       Gravitational constant
%
%           c.vo      The speed of the body at initial lift off
%
%           c.yfix    The desired lift off conditions
%
%           c.yo      The initial lift off conditions
%
%           c.q       The weighting vector
%
%           c.nondim  6x1 vector of nondimensionalization parameters for
%                     the states
%
% Outputs:
%   v              Vector of structures containing system data for a each
%                  time step.  The elements of these structures are:
%                    v.t    Time of data
%
%                    v.y    6x1 vector of states at time v.t
%
%                    v.fp   2x1 vector giving the x and z position of the
%                           foot
%
%   vtd            Vector of structures containing system data at each
%                  touchdown condition. This structures has the same
%                  elements as v.
%
%   vlo            Vector of structures containing system data at each lift
%                  off condition. This structures has the same
%                  elements as v.
%
%   return_status  How the simulaiton ended. (returned, unretruned, fallen)
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    global beta_now tc_now i controlled nv returned unreturned fallen;
  % Handy constants
    returned   = 1;
    unreturned = 2;
    fallen     = 3;
  % if c is not specified use default
    if isnumeric(c)
        clear c
        global c
```

```
    c.k     = 20;       %spring constant
    c.etao  = 0.015;    %nominal leg length
    c.d     = 0.004;    %distance of leg attachment point from COM 0.004
    c.m     = 0.0025;   %mass
    c.I     = 1.86e-7;  %moment of inertia Iyy 1.86e-7
    c.g     = 9.81;         %gravitational acceleration
    c.yfix  = [0.0059; -0.0107; 0.0051; 0.1289; -0.0984; -0.5056];
    c.q     = [0; 4; 8; 1; 3; 7];
    c.max_period = 0.5;
  end
% If sp is not specified, use default
  if isnumeric(sp)
    clear sp
    sp.control       = 1;                %use controler y/n?
    sp.max_steps     = 10;               %number of steps to take
    sp.max_step_size = 1e-4;             %max step size
    sp.beta          = [1.2, 1.2, 1.2];  %leg angles if uncontroled
    sp.beta_const    = 1;                %vary leg angle?
    sp.end_criterion = @default_end_criterion;
    sp.skip_flight_1 = 0;
  end
  controlled = sp.control;
% Set simulation options
  ground_options = odeset('Events',@lift_off_event,'Refine',2,...
                          'AbsTol',1e-12,'RelTol',1e-12,'InitialStep',...
                          1e-15,'MaxStep',sp.max_step_size);
  flight_options = odeset('Events',@touch_down_event,'Refine',2,...
                          'AbsTol',1e-12,'RelTol',1e-12,'InitialStep',...
                          1e-15,'MaxStep',sp.max_step_size);

  i     = 1;
% State recording variables
  v.y   = yo;
  v.t   = 0;
  vlo   = v;
  return_status = unreturned;
  keep_it_up    = 1;
% Simulate stride if
  while(keep_it_up)
    if sp.control
      [beta(i),tc] = collocation4(vlo(i).y,c);
    else
      if sp.beta_const
        beta(i)  = sp.beta(1);
      else
        beta(i)  = sp.beta(i);
      end
     % tc not 0
      tc           = 1;
    end
   % Foot placement point
    vlo(i,1).fp = [vlo(i).y(1)-c.d*sin(vlo(i).y(3))+...
                   c.etao*cos(beta(i));...
                   vlo(i).y(2)-c.d*cos(vlo(i).y(3))+...
                   c.etao*sin(beta(i))];
    v(length(v)).fp = vlo(i,1).fp;
   % If there is a flight phase
    if (tc > 0)&~(sp.skip_flight_1&i==1)
      % Set global variables
      beta_now = beta(i);
```

```
            tc_now   = tc;
          % Flight Phase
            clear vadd
            vadd     = flight_phase(vlo(i),beta(i),c,flight_options);
            vtd(i,1) = vadd(length(vadd));
            v        = [v;vadd];
        else
      % If there is no flight phase
            vtd(i,1) = vlo(i);
        end
        if((vtd(i).y(2)>=0)&(vtd(i).y(5)>0))
                return_status = fallen;
        else
          % Ground Phase
            clear vadd
            vadd   = ground_phase(vtd(i),c,ground_options);
            v      = [v;vadd];
            i = i+1;
            vlo(i,1)  = vadd(length(vadd));
        end
      % Decide if simulation should stop running.
        [keep_it_up,return_status] = ...
                            sp.end_criterion(v,vlo,vtd,sp,c,return_status);
    end
end

function v = flight_phase(vlo,beta,c,flight_options)
%=========================================================================
% function v = flight_phase(vlo,beta,c,flight_options)
%
% This function runs flight phase simulations.
%
% Inputs:
%   vlo             Structure containing system data at the lift off
%                   condition. See rb_slip_sim for structure explanation.
%
%   beta            Leg touchdown angle.
%
%   c               Structure model constants.  See rb_slip_sim for
%                   structure info.
%
%   flight_options  Vector of settings for ode45.  See odeset of more info.
%
% Outputs:
%   v               Vector of structures containing system data for each
%                   time step in the current flight phase.  See rb_slip_sim
%                   for structure explanation.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    tspan = [0 c.max_period];
    [t,y] = ode45(@flight_dynamics,tspan,vlo.y,flight_options);
    for j = 1:length(t)-1
        v(j,1).y  = y(j+1,:).';
        v(j,1).t  = t(j+1,:)+vlo.t;
        v(j,1).fp = [v(j).y(1)-c.d*sin(v(j).y(3))+c.etao*cos(beta);...
                     v(j).y(2)-c.d*cos(v(j).y(3))+c.etao*sin(beta)];
    end
end
```

```
function y = flight_dynamics(t,x)
%=========================================================================
% function y = flight_dynamics(t,x)
%
% This function calculates flight dynamics from the state.
%
% Inputs:
%   t   Time.
%
%   x   State vector at time t.
%
% Outputs:
%   y   Derivative of the state vector calculated with the equations of
%       motion.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    global c;
    y(1,:)  = x(4); %x
    y(2,:)  = x(5); %z
    y(3,:)  = x(6); %theta

    y(4,:)  = 0;    %xdot
    y(5,:)  = c.g;  %zdot
    y(6,:)  = 0;    %thetadot
end

function v = ground_phase(vtd,c,ground_options)
%=========================================================================
% function v = ground_phase(vtd,c,ground_options)
%
% This function runs ground phase simulations.
%
% Inputs:
%   vtd             Structure containing system data at the touchdown
%                   condition. See rb_slip_sim for structure explanation.
%
%   c               Structure model constants.  See rb_slip_sim for
%                   structure info.
%
%   ground_options  Vector of settings for ode45.  See odeset of more info.
%
% Outputs:
%   v               Vector of structures containing system data for each
%                   time step in the current ground phase.  See rb_slip_sim
%                   for structure explanation.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    vtd.y(1) = vtd.y(1)-vtd.fp(1);
    tspan   = [vtd.t vtd.t+c.max_period];
    [t,y] = ode45(@ground_dynamics, tspan, vtd.y, ground_options);
    for j=1:length(t)-1
        v(j,1).y    = y(j+1,:).';
        v(j,1).y(1) = y(j+1,1)+vtd.fp(1);
        v(j,1).t    = t(j+1);
        v(j,1).fp   = [vtd.fp(1);0];
    end
end
```

```
function dy = ground_dynamics(t,y)
%=========================================================================
% function dy = ground_dynamics(t,y)
%
% This function calculates ground phase dynamics from the state.
%
% Inputs:
%   t   Time.
%
%   y   State vector at time t.
%
% Outputs:
%   dy  Derivative of the state vector calculated with the equations of
%       motion.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    global c;
    dy = [y(4);...

          y(5);...

          y(6);...

          c.k/(c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(...
          3)))^2)^(1/2)-1)*(y(1)-c.d*sin(y(3)));...

          c.k/(c.m)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(...
          3)))^2)^(1/2)-1)*(y(2)-c.d*cos(y(3)))+c.g;...

          c.d*(c.k)/(c.I)*(c.etao/((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*...
          cos(y(3)))^2)^(1/2)-1)*(y(2)*sin(y(3))-y(1)*cos(y(3)))];
end

function [value,isterminal,direction] = touch_down_event(t,y)
%=========================================================================
% function [value,isterminal,direction] = touch_down_event(t,y)
%
% This function determines when the flight phase should end.
%
% Inputs:
%   t   Time.
%
%   y   State vector at time t.
%
% Outputs:
%   value       Vector of values.  When a value is 0, an event occurs.
%
%   isterminal  Vector of switches to tell if an event ends the simulation
%               or not.
%
%   direction   Vector of switches.  For direction = -1 the derivative of a
%               0 value must be nevative for the event to trigger.  For a
%               direction = 1 the derivative of a 0 value must be positive
%               for the event to trigger.  For a direction = 0 the event
%               can be triggered from any direction.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    global c beta_now tc_now i controlled
```

```
  % Detect touchdown
    value(1) = (y(2)-c.d*cos(y(3))+c.etao*sin(beta_now));
  % Detect fall
    value(2) = (y(2)-c.d*cos(y(3)));
  % Make sure a fall is not missed
    if value(2)>0
        value(2) = sin(1e4*t);
    end
  % x as measured in the ground phase
    x = -c.etao*cos(beta_now);
    if controlled
      % If the system is controlled, the flight phase can end any time the
      % foot is on the ground and the leg is entering compression.
        dervy   = (x-c.d*sin(y(3)))*(y(4)-c.d*cos(y(3))*y(6))+...
                  (y(2)-c.d*cos(y(3)))*(y(5)+c.d*sin(y(3))*y(6));
    else
      % If there is a fixed angle reset policy, the leg touches down only
      % when the foot is on the ground and has a vertical velocity
      % downward.
        dervy   = -y(5)-c.d*sin(y(3))*y(6);
    end
  % If the system is controlled and there is more than one point in time
  % the foot could touch down at the angle specified, the control system
  % deciedes when the leg should touch down.
    if dervy <=0 & (~controlled|(t>=.8*tc_now & t<=1.2*tc_now))
        isterminal = [1,1];   % stop the integration
    else
        isterminal = [0,1];
    end
    direction  = [0,0];
end

function [value,isterminal,direction] = lift_off_event(t,y)
%=========================================================================
% [value,isterminal,direction] = lift_off_event(t,y)
%
% This function determines when the ground phase should end.
%
% Inputs:
%   t   Time.
%
%   y   State vector at time t.
%
% Outputs:
%   value       Vector of values.  When a value is 0, an event occurs.
%
%   isterminal  Vector of switches to tell if an event ends the simulation
%               or not.
%
%   direction   Vector of switches.  See touch_down_event for details.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    global c;
  % Spring length during the ground phase
    eta     = sqrt((y(1)-c.d*sin(y(3)))^2+(y(2)-c.d*cos(y(3)))^2);
  % Detect lift off
    value(1,1) = (c.etao-eta);
  % Detect fall
    value(2,1) = (y(2)-c.d*cos(y(3)));
```

```
  % Make sure a fall is not missed
    if value(2)>0
        value(2) = sin(1e4*t);
    end
    isterminal = [1;1];   % stop the integration
    direction = [-1;0];
end
```

C.2 Front End Codes

The simulation code is very general and therefore requires an enormous amount of input. It also outputs almost every piece of data from the simulation. In order to organize the inputs and outputs to and from the rb_slip_sim front end codes were developed as an interface.

C.2.1 auto_rb_fixed_point_find1

The function auto_rb_fixed_point_find1 finds rigid body fixed points in a specified gait family using a similar point mass fixed point as a starting point. It should be started in the middle of the gait family and run in either direction for best results.

```
function auto_rb_fixed_point_find1
%=========================================================================
% auto_rb_fixed_point_find1
%
% This function works with rb_slip_sim to find rigid body fixed points
% using similar point mass fixed points as initial guesses.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    clear all
    global c sp delta xlo
    format long
  % Model Parameters
    c.k     = 20;       %spring constant
    c.etao  = 0.015; %nominal leg length
    c.d     = 0.004; %distance of leg attachment point from COM 0.004
    c.m     = 0.0025;%mass
    c.I     = 1.86e-7;      %moment of inertia Iyy 1.86e-7
    c.g     = 9.81;      %gravitational acceleration
    c.yfix  = [0;0;0;0;0;0];
    c.q     = [0; 0; 0; 0; 0; 0];
    c.max_period = 0.5;

  % Simulation Parameters
    sp.control       = 0;
    sp.max_step_size = 1e-4;
    sp.beta          = 1.2;
    sp.beta_const    = 1;
    sp.end_criterion = @default_end_criterion;
```

```
  % Load point mass fixed points.
    load(['FixedPoints/fproachparamb' num2str(sp.beta*100)])

    for n = 51:101
        delta = -fvtot(n,2);
                    % [zo ;vo ; thetao; thetadoto]
        thetao  = 0;
        if n==51
               % [vo ; thetao; thetadoto]
            x0 = [fvtot(n,1);thetao; 0]
        else
            x0 = [fvtot(n,1);fvtot_rb(n+1,4:5)']
        end
        sp.skip_flight_1 = 1;
        sp.max_steps     = 2;
        options = optimset('MaxFunEvals',10e7,'TolFun',10e-15);
        [xp,fval] = fsolve(@fsolve_io,x0,options)
        F = fsolve_io(xp)
        sp.skip_flight_1 = 0;
        sp.max_steps     = 1;
        xro = [xlo(1:2);xlo(4:5)]
        [xrf,fval] = fsolve(@fsolve_io,xro,options)
               %[zo; vo; delta; thetao; thetadoto]
        xlop = [xrf(1:2);-delta;xrf(3:4)];
        fvtot_rb(n,:) = xlop';
        save(['FixedPoints/fproachparamb' num2str(beta*100)],'fvtot_rb',...
              '-append')
    end
end

function F = fsolve_io(x)
%=========================================================================
% F = fsolve_io(x)
%
% This function interfaces between the simulation and the fsolve routine.
%
% Inputs:
%   x   The states that are being varied by fsolve.
%
% Outputs:
%   F   The difference between the beginning state and the end state.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    global c sp delta xlo
    if length(x)==3
        %[vo ; thetao; thetadoto]
        zstart = -0.015*sin(sp.beta)+0.004*cos(x(2));
        yo     = [0;zstart;x(2);x(1)*cos(delta);-x(1)*sin(delta);x(3)];
    elseif length(x)==4
        yo    = [0;x(1);x(3);x(2)*cos(delta);-x(2)*sin(delta);x(4)];
    else
        yo    = [0;x(1);x(4);x(2)*cos(x(3));-x(2)*sin(x(3));x(5)];
    end
    [v,vlo,vtd,return_status] = rb_slip_sim(yo,sp,c);
    vtyp = 0.3;
    if length(x)==3
        xlo  = [vlo(2,1).y(2); sqrt(vlo(2,1).y(4)^2+vlo(2,1).y(5)^2);...
                atan(-
vlo(2,1).y(5)/vlo(2,1).y(4));vlo(2,1).y(3);vlo(2,1).y(6)];
```

```
        F    = [zstart-vtd(2,1).y(2);...
                x(1)-sqrt(vtd(2,1).y(4)^2+vtd(2,1).y(5)^2);...
                atan(-vtd(2,1).y(5)/vtd(2,1).y(4))-delta;...
                x(2)-vtd(2,1).y(3); x(3)-vtd(2,1).y(6)]...
                .*[1/c.etao;1/vtyp;1;1;c.etao/vtyp];
        post = [[zstart;x(1);delta;x(2:3)],F]
    elseif length(x)==4
        if i<2
            F    = [x(1)-vtd(1,1).y(2);...
                    x(2)-sqrt(vtd(1,1).y(4)^2+vtd(1,1).y(5)^2);...
                    atan(-vtd(1,1).y(5)/vtd(1,1).y(4))+delta;...
                    x(3)-vtd(1,1).y(3); x(4)-vtd(1,1).y(6)]...
                    .*[1/c.etao;1/vtyp;1;1;c.etao/vtyp];
        else
            F    = [x(1)-vlo(2,1).y(2);...
                    x(2)-sqrt(vlo(2,1).y(4)^2+vlo(2,1).y(5)^2);...
                    atan(-vlo(2,1).y(5)/vlo(2,1).y(4))+delta;...
                    x(3)-vlo(2,1).y(3); x(4)-vlo(2,1).y(6)]...
                    .*[1/c.etao;1/vtyp;1;1;c.etao/vtyp];
        end
        post = [[x(1:2);delta;x(3:4)],F]
    else
        vend = vlo(2,1);
        F    = [x(1)-vlo(2,1).y(2);...
                x(2)-sqrt(vlo(2,1).y(4)^2+vlo(2,1).y(5)^2);...
                atan(-vlo(2,1).y(5)/vlo(2,1).y(4))-x(3);...
                x(4)-vlo(2,1).y(3); x(5)-vlo(2,1).y(6)]...
                .*[1/c.etao;1/vtyp;1;1;c.etao/vtyp];
         post = [x,F]
    end
end
```

C.2.2 rb_perterbation_return1

The function rb_perterbation_return1 is a front end for rb_slip_sim which allows the user to observe system behavior at a specified rigid body fixed point or perturbed from it in a specified fashion. A similar code was used to observe the point mass system in the same way.

```
function rb_perterbation_return1
%==========================================================================
% rb_perterbation_return
%
% This function is a front end for rb_slip_sim which allows for general
% observation of system behavior with control and without.
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    clear all
    global c
    beta = 1.2;
    load(['FixedPoints/fproachparamb' num2str(beta*100)])
    format long
    %[v delta theta thetadot]
    n      = 51
```

```
    pstate = 3
                % [zo;vo;deltao;theato,thetadoto]
    fp          = fvtot_rb(n,:);
    pt          = 0.3;
    so          = fp;
    so(pstate)  = fp(pstate)+pt
  % Convert from polar velocity to rectangular
    yo          = [0;so(1);so(4);so(2)*cos(so(3));-so(2)*sin(so(3));so(5)];

  % Model Parameters
    c.k     = 20;       %spring constant
    c.etao  = 0.015; %nominal leg length
    c.d     = 0.004; %distance of leg attachment point from COM 0.004
    c.m     = 0.0025;%mass
    c.I     = 1.86e-7;      %moment of inertia Iyy 1.86e-7
    c.g     = 9.81;      %gravitational acceleration
    c.yfix  = [0;fp(1);fp(4);fp(2)*cos(fp(3));-fp(2)*sin(fp(3));fp(5)];
    c.q     = [0; 4; 8; 1; 3; 7];
    c.max_period = 0.5;

  % Simulation Parameters
    sp.control       = 1;
    sp.max_steps     = 2;
    sp.max_step_size = 1e-4;
    sp.beta          = [1.2, 1.2, 1.2];
    sp.beta_const    = 1;
    sp.end_criterion = @default_end_criterion;
    sp.skip_flight_1 = 0;

  % Simulation
    [v,vlo,vtd,return_status] = rb_slip_sim(yo,sp,c);
  % Animation
    Elipse2d(v,c);
  % Plot
    Y   = [v.y];
    Ylo = [vlo.y];
    Ytd = [vtd.y];
    figure
    subplot(5,1,1)
    plot([vlo.t],Ylo(2,:),'b.')
    set(gca,'YDir','reverse');
    ylabel('z')
    subplot(5,1,2)
    plot([vlo.t],Ylo(3,:),'b.')
    ylabel('\theta')
    subplot(5,1,3)
    plot([vlo.t],Ylo(4,:),'b.')
    ylabel('xdot')
    subplot(5,1,4)
    plot([vlo.t],Ylo(5,:),'b.')
    set(gca,'YDir','reverse');
    ylabel('zdot')
    subplot(5,1,5)
    plot([vlo.t],Ylo(6,:),'b.')
    xlabel('time')
    ylabel('\thetadot')
end
```

C.2.3 rb_eig_fam1

The function rb_eig_fam1 finds the eigenvalues of the Poincaré map linearized about fixed points of a specified gait family for the rigid body SLIP. A similar code was used to find eigenvalues for the point mass SLIP.

```
function rb_eig_fam1(book_beta,start_over,timed)
%=========================================================================
% rb_eig_fam1(book_beta,start_over,timed)
%
% This function calculates the eigenvalues of the Poincare map linearized
% about the fixed points of a gait family and records them in a .mat file.
%
% Inputs:
%   beta_book   An integer between 110 and 130 which is 100 times the value
%               of beta associated with the gait family to be tested.
%
%   start_over  Start from the last saved set of eigenvalues (0) or start
%               from the begining (1)
%
%   timed       Stop calculations before computer lab opens in the morning
%               (1) or continue calculations regardless of the time (0).
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    global gait_family
    %load fixed points of beta from file
    load(['FixedPoints/fproachparamb' num2str(book_beta)])
    beta = book_beta/100;
    gait_family = beta;
    if start_over
        clear eigs_for_col4_q_4_5_1_3_3
    end
    format long
    %[v delta theta thetadot]
    global pt
    pt = 10^-7;
    min_mult = 1e-2;
    if exist('eigs_for_col4_q_4_5_1_3_3')
       size_of_eigs = size(eigs_for_col4_q_4_5_1_3_3);
       starti = size_of_eigs(1)+1;
    else
        i = 1;
        fp = fvtot_rb(i,:)
        eigs_for_col4_q_4_5_1_3_3(i,:) = hocd_eig(fp,pt,min_mult,i);
        starti = i+1;
    end
    for i = starti:length(fvtot_rb)
        fp = fvtot_rb(i,:);
        eigs_out         = hocd_eig(fp,pt,min_mult,i);
        for j =1:5
            [Y,I] = min(abs(eigs_out(:)-eigs_for_col4_q_4_5_1_3_3(i-1,j)));
            eigs_for_col4_q_4_5_1_3_3(i,j) = eigs_out(I(1));
            eigs_out(I(1)) = inf;
        end
        save(['FixedPoints/fproachparamb' num2str(book_beta)],...
```

```
            'eigs_for_col4_q_4_5_1_3_3','-append')
        keyboard
        time = clock;
        if(timed & ((time(4)>=6 & time(5)>0) & (time(4)<=20 & time(5)>0)))
            exit
        end
    end
    exit
end


function eigs = hocd_eig(fp,pt,min_mult,j)
%=========================================================================
% eigs = hocd_eig(fp,pt,min_mult,j)
%
% This function calculates eigenvalues of the Poincare map linearized about
% a particular fixed point.
%
% Inputs:
%   fp        The fixed point in [z,v,delta,theta,thetadot] form.
%
%   pt        The amount of perturbation to use for the difference formula.
%
%   min_mult  The minimum scaleing to use on the perturbation.
%
%   j         The index of the fixed point.
%
% Outputs:
%   eigs      The eigenvalues of the linearized Poincare map.
%
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    controlled = 1;
    for i = 1:5
        if abs(fp(i)) < min_mult
            pts     = pt*min_mult;
        else
            pts     = pt*abs(fp(i));
        end
        so      = fp;
        so(i)   = fp(i)-2*pts;
        fprintf(['Calculation for fp %d, perterbing state %d, minus 2pt'...
                 '\n'],j,i)
        sfmm    = Vertical_Plane_Fixed_Point_Test(so,fp,controlled);
        so(i)   = fp(i)-pts;
        fprintf(['Calculation for fp %d, perterbing state %d, minus pt'...
                 '\n'],j,i)
        sfm     = Vertical_Plane_Fixed_Point_Test(so,fp,controlled);
        so(i)   = fp(i)+pts;
        fprintf(['Calculation for fp %d, perterbing state %d, plus pt'...
                 '\n'],j,i)
        sfp     = Vertical_Plane_Fixed_Point_Test(so,fp,controlled);
        so(i)   = fp(i)+2*pts;
        fprintf(['Calculation for fp %d, perterbing state %d, plus 2pt'...
                 '\n'],j,i)
        sfpp    = Vertical_Plane_Fixed_Point_Test(so,fp,controlled);
        A(:,i)  = (sfmm-8*sfm+8*sfp-sfpp)./(12*pts);
    end
    eigs = eig(A)'
```

```
end


function sf = Vertical_Plane_Fixed_Point_Test(so,fp,control)
%=========================================================================
% sf = Vertical_Plane_Fixed_Point_Test(so,fp,control)
%
% This function starts a one stride simulation with the given initial
% condtions and retuns the results in polar velocity form.
%
% Inputs:
%   so        The initial conditions in [z,v,delta,theta,thetadot] form.
%
%   fp        The fixed point in [z,v,delta,theta,thetadot] form.
%
%   control   Control on/off (1/0)
%
% Outputs:
%   sf        The final state in [z,v,delta,theta,thetadot] form.
%
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    global c gait_family
  % Model Parameters
    c.k     = 20;       %spring constant
    c.etao  = 0.015; %nominal leg length
    c.d     = 0.004; %distance of leg attachment point from COM 0.004
    c.m     = 0.0025;%mass
    c.I     = 1.86e-7;      %moment of inertia Iyy 1.86e-7
    c.g     = 9.81;      %gravitational acceleration
    c.yfix  = [0;fp(1);fp(4);fp(2)*cos(fp(3));-fp(2)*sin(fp(3));fp(5)];
    c.q     = [0; 4; 8; 1; 3; 7];
    c.max_period = 0.5;

  % Simulation Parameters
    sp.control       = control;
    sp.max_steps     = 1;
    sp.max_step_size = 1e-4;
    sp.beta          = gait_family;
    sp.beta_const    = 1;
    sp.end_criterion = @default_end_criterion;
    sp.skip_flight_1 = 0;

    yo = [0;so(1);so(4);so(2)*cos(so(3));-so(2)*sin(so(3));so(5)];
    [v,vlo,vtd,return_status] = rb_slip_sim(yo,sp,c);
    j = length(vlo);
    sf(:,1) = [vlo(j).y(2);sqrt(vlo(j).y(4)^2+vlo(j).y(5)^2);...
                   atan(-vlo(j).y(5)/vlo(j).y(4));vlo(j).y(3);vlo(j).y(6)];
end
```

C.2.4 rb_auto_perterbation_return1

The function rb_auto_perterbation_return1 systematically tests to see if, for a given rigid body fixed point, the control scheme can return the system to the fixed point if the velocity angle is perturbed

by an angle between $-\frac{31\pi}{32}$ and π. A similar code was used to test the perturbation returnability for the point mass case.

```
function rb_auto_perterbation_return1(n,start_over,timed)
%=========================================================================
% rb_auto_perterbation_return1(n,start_over,timed)
%
% Test the control system to see if it can return the system from various
% large energy conservative perturbations.
%
% Inputs:
%   n           The index number of the fixed point in a spacific gait
%               family to be tested.
%
%   start_over  Start from the last saved perturbation (0) or start
%               from the begining (1)
%
%   timed       Stop calculations before computer lab opens in the morning
%               (1) or continue calculations regardless of the time (0).
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    beta = 1.2
    format long
    %[v delta theta thetadot]
    pstate = 3
    success = 1;
    try
        load(['FixedPoints/pert_ret_beta_',num2str(beta*100),'_n_',...
              num2str(n)])
    catch
        success = 0;
    end
    if (start_over==1 || ~success)
        load(['FixedPoints/fproachparamb' num2str(beta*100)])
        %[zo;vo;deltao]
        fp     = fvtot_rb(n,:);
        starti = 1;
    elseif (start_over == 0)
        starti = size(return_for_col2G_q_4_5_1_3_3,1)+1;
    else
        starti = start_over
    end
    pt     = -31*pi/32:pi/32:pi;
    if start_over > 1
        lengthpt = start_over;
    else
        lengthpt = length(pt)
    end
    for i = starti:lengthpt
        fprintf('Going from perterbation of %9.8g.\n',pt(i));
        so     = fp;
        so(pstate)  = fp(pstate)+pt(i)
        return_for_col2G_q_4_5_1_3_3(i,3) = pt(i);
        [return_for_col2G_q_4_5_1_3_3(i,1),...
         return_for_col2G_q_4_5_1_3_3(i,2)] =...
                          Vertical_Plane_Fixed_Point_Test(so,fp,1,timed);
        save(['FixedPoints/pert_ret_beta_',num2str(beta*100),...
```

```
            '_n_' num2str(n)],'return_for_col2G_q_4_5_1_3_3', 'fp',...
            '-append')
        time = clock;
        if (timed & ((time(4)>=6 & time(5)>0) & (time(4)<=20 & time(5)>0)))
           exit
        end
    end
    if ((start_over <= 1) & (timed == 1))
        exit
    end
    %Centered difference for the rest of the fixed points.
end

function [return_status,i]...
                    = Vertical_Plane_Fixed_Point_Test(so,fp,control,timed)
%=========================================================================
% [return_status,i] = Vertical_Plane_Fixed_Point_Test(so,fp,control,timed)
%
% This function starts simulations which determine if the system returned
% to a fixed point, did not return to a fixed point, or just fell.
%
% Inputs:
%   so        The initial conditions in [z,v,delta,theta,thetadot] form.
%
%   fp        The fixed point in [z,v,delta,theta,thetadot] form.
%
%   control   Control on/off (1/0)
%
%   timed     Stop calculations before computer lab opens in the morning
%             (1) or continue calculations regardless of the time (0).
%
% Outputs:
%   return_status  How the simulaiton ended. (returned, unretruned, fallen)
%
%   i              The number of steps the simulation took to do what ever
%                  it did.
%
% Cary R. Maunder, Oregon State University, 2006
%=========================================================================
    global c
  % Model Parameters
    c.k     = 20;       %spring constant
    c.etao  = 0.015; %nominal leg length
    c.d     = 0.004; %distance of leg attachment point from COM 0.004
    c.m     = 0.0025;%mass
    c.I     = 1.86e-7;      %moment of inertia Iyy 1.86e-7
    c.g     = 9.81;      %gravitational acceleration
    c.yfix  = [0;fp(1);fp(4);fp(2)*cos(fp(3));-fp(2)*sin(fp(3));fp(5)];
    c.q     = [0; 4; 8; 1; 3; 7];
    c.max_period = 0.5;

  % Simulation Parameters
    sp.control       = 1;
    sp.max_steps     = 100;
    sp.max_step_size = 1e-4;
    sp.beta          = 1.2;
    sp.beta_const    = 1;
    sp.end_criterion = @perturbation_end_criterion;
    sp.skip_flight_1 = 0;
```

```
    yo = [0;so(1);so(4);so(2)*cos(so(3));-so(2)*sin(so(3));so(5)];
    [v,vlo,vtd,return_status] = rb_slip_sim(yo,sp,c);
    i = length(vtd);
end
```

C.3 Simulation Ending Criterion

For different applications the simulation needed to end for different reasons. For this reason, functions were made to determine when the simulation should end based on the needs of the specific application. Handles to these functions were passed to the simulation and it called them at the end of each stride.

C.3.1 default_end_criterion

The function default_end_criterion was used when the basic end criterion were needed. The simulation was ended if the system fell or if the system had reached the maximum number of steps if was supposed to take.

```
function [k_i_u,r_status] = default_end_criterion(v,vlo,vtd,sp,c,r_status)
%=========================================================================
% [k_i_u,r_status] = default_end_criterion(v,vlo,vtd,r_status)
%
% This function determines if the simulation should stop.
%
% Inputs:
%   v         Vector of structures containing system data for a each
%             time step.  See rb_slip_sim for structure explanation.
%
%   vlo       Vector of structures containing system data at each
%             touchdown condition.  See rb_slip_sim for structure
%             explanation.
%
%   vtd       Vector of structures containing system data at each lift
%             off condition.  See rb_slip_sim for structure explanation.
%
%   sp        Simulation parameter structure.  See rb_slip_sim for
%             structure info.
%
%   c         Structure model constants.  See rb_slip_sim for structure
%             info.
%
%   r_status  Tells function if the system has fallen.
%
% Outputs:
%   k_i_u     Tells the simulation to keep it up (1) or stop (0)
%
%      r_status  How the simulaiton ended. (returned, unretruned, fallen)
```

```
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    global returned unreturned fallen
    k_i_u = 1;
    if r_status == fallen || length(vtd) >= sp.max_steps
        k_i_u = 0;
    end
end
```

C.3.2 perturbation_end_criterion

The function perturbation_end_criterion was used as the end function when the simulation needed to end if the system returned to within 1% of the fixed point from a perturbation.

```
function [k_i_u,r_status] = ...
                        perturbation_end_criterion(v,vlo,vtd,sp,c,r_status)
%==========================================================================
% [k_i_u,r_status] = perturbation_end_criterion(v,vlo,vtd,sp,c,r_status)
%
% This function determines if the simulation should stop.
%
% Inputs:
%   v         Vector of structures containing system data for a each
%             time step.  See rb_slip_sim for structure explanation.
%
%   vlo       Vector of structures containing system data at each
%             touchdown condition.  See rb_slip_sim for structure
%             explanation.
%
%   vtd       Vector of structures containing system data at each lift
%             off condition.  See rb_slip_sim for structure explanation.
%
%   sp        Simulation parameter structure.  See rb_slip_sim for
%             structure info.
%
%   c         Structure model constants.  See rb_slip_sim for structure
%             info.
%
%   r_status  Tells function if the system has fallen.
%
% Outputs:
%   k_i_u     Tells the simulation to keep it up (1) or stop (0)
%
%      r_status  How the simulaiton ended. (returned, unretruned, fallen)
%
% Cary R. Maunder, Oregon State University, 2006
%==========================================================================
    global returned unreturned fallen nv
    k_i_u = 1;
    if r_status == fallen || length(vtd) >= sp.max_steps
        k_i_u = 0;
    end
    if r_status ~= fallen
```

```
        if exist('nv')
            start = size(nv,1)+1;
        else
            start = 1;
        end
        for j=start:length(v)
            nv(j,:) = [v(j).y(2),sqrt(v(j).y(4)^2+v(j).y(5)^2),...
                atan(-v(j).y(5)/v(j).y(4)),v(j).y(3),v(j).y(6)];
        end
        r_status = returned;
        k_i_u = 0;
        fp = [c.yfix(2),sqrt(c.yfix(4)^2+c.yfix(5)^2),...
                atan(-c.yfix(5)/c.yfix(4)),c.yfix(3),c.yfix(6)]
        for j=1:5
            nvmag(j) = max(nv(:,j))-min(nv(:,j));
            enddif(j) = abs(nv(length(v),j)-fp(j));
            if enddif(j)>0.01*nvmag(j)
                r_status = unreturned;
                k_i_u = 1;
            end
        end
        exc = nv(length(v),:)
        enddif
    end
end
```