DESIGNING RELIABLE CONTROLLERS FOR BIPEDAL ROBOTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Matthew P. Kelly

May 2016

DESIGNING RELIABLE CONTROLLERS FOR BIPEDAL ROBOTS

Matthew P. Kelly, Ph.D.

Cornell University 2016

This dissertation is made up of several chapters, each of which is a stand-alone document. Together, these chapters are representative of my research here at Cornell, focused on non-linear control of bipedal walking robots. Controllers for bipedal walking often rely on simple models. Here I present two controllers for simplified models, both of which are designed to be robust to bounded disturbances. The first is a foot-step planner for the inverted pendulum model of walking, and the second is a trajectory tracking controller for a double-pendulum model of walking. Next, I have a chapter that talks about some of the interesting features of simulators that include contact mechanics, like those used for walking robots. One topic that comes up repeatedly when studying bipedal locomotion control is trajectory optimization. I wrote a introduction and tutorial paper on the topic, which is included here as a chapter. I also developed my own trajectory optimization algorithm, DirCol5i, which is described in a second chapter on trajectory optimization. The final thrust of my research at Cornell was developing a walking controller for the Cornell Ranger walking robot. Here I describe my methodology for controller design. This process is divided into three parts: the first is to use simulation of the robot to develop a control architecture for the robot, which can be described using a small number of parameters. These parameters are then selected using optimization, first of the simulation of Ranger, and then using experiments on the physical robot. In discussing the results, we place special emphasis on the interplay between human intuition and computer optimization for controller design.

BIOGRAPHICAL SKETCH

Matthew Kelly grew up in Marbletown, New York, a small town in the foothills of the Catskill Mountains. After graduating from Rondout Valley high school in 2007, Matt went on to study engineering at Tufts University. He graduated from Tufts in 2011, with a bachelors of science in mechanical engineering and a minor in music (classical theory and trombone performance). While at Tufts, Matt completed an honors thesis project, where he developed a novel sagittal bone saw. Also in 2011, Matt was a visiting researcher at the Massachusetts Institute of Technology, where he worked on a non-linear controller for a filament-stretching extensional rheometer. After completing his bachelors degree, Matt enrolled in the graduate program at Cornell University in mechanical engineering. His research was focused on non-linear control for bipedal walking robots. Matt designed and implemented a robust controller for the Cornell Ranger, and wrote his own trajectory optimization library for Matlab. He completed a masters of science in 2014, and graduated with a doctorate in philosophy in 2016.

ACKNOWLEDGEMENTS

The successful completion of my dissertation would not have been possible without the help and support of a whole community of people. First, I would like the thank my advisor Andy Ruina for his advice, long talks about robots, and many difficult questions, all of which have made me a better researcher. I also am grateful to my committee members: Hadas Kress-Gazit, Brandon Hencey, and Hod Lipson for their guidance and support. I would also like to thank professors: Ashutosh Saxena, Alex Vladimirsky, Anil Rao, and Russ Tedrake, for spending time discussing robotics and optimization. Finally, I would like to thank Marcia Sawyer, for her guidance and support, and making all of the logistics of graduate school work out correctly.

Throughout my time at Cornell I spent much time working and talking with my fellow lab members, and they have helped with my research all along: Matt Sheen, Jason Cortell, Javad Hasaneini, Boris Kogan, Atif Chaurhry, Petr Zaytsev, and Anoop Grewal. I would also like thank several of the past lab members, all of whom were happy to help with my research: Pranav Bhounsule, Manoj Srinivasan, and Steve Collins.

I would also like to thank my friends and family, all of whom helped support me through the years of graduate school: my parents and grandparents, Mike, Chris, Sarah, Tim, Katie, Dave, Sam, Amy, Lauren, J.J., Lorraine, and the myriad of contra dancers in Ithaca.

Lastly, this research was supported by the National Science Foundation Graduate Research Fellowship Program (fellow ID: 2011116308), the National Robotics Initiative (grant number: 1317981), and three semesters of teaching assistantships from the Cornell Mechanical Engineering department.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This dissertation is divided into a few categories. My primary project has been to develop a reliable walking controller for Cornell Ranger (Chapters §2 and §3). Next, is a collection of smaller projects that are related to robust control of bipedal robots (Chapters §4, §6, and §5), but that were not directly used in the Ranger project. Finally, there are two chapters relating to trajectory optimization (§7 and §8), an important technique that is used for designing walking controllers.

The Ranger project has been a constant thread throughout my research, starting with simple walking models and culminating with a controller on the real robot that is able to automatically improve itself while walking. I developed my own simulation environment for Ranger (§2.16), and then used this to develop a control architecture (§2.5). Next, I used optimization of the simulated model of Ranger to determine the best set of parameters for the walking controller. This controller can be transfered directly from the simulation to the real robot. Finally, I used on-line optimization to continually improve the controller while Ranger is walking, accounting for differences between the simulation and the real robot. My preliminary work on the controller (§2) was presented in ICRA 2016 and the full research is described in Chapter §3, which will be submitted to a peer-reviewed journal.

While trying to come up with a control architecture for Ranger, I developed a variety of control algorithms for simple models of walking robots. I've included two of these here. The first project (published in ICRA 2015, §4) is a robust controller for a point-mass model of walking, that is able to prevent falls despite large disturbances. This controller is able to compute the best choice of foot-placement locations for a simple walking model. The next project (Chapter §5) was to develop a robust controller for the

swing leg of a walking robot, modeled as a double pendulum, which could reliably put the robot's foot in the correct place, as specified by the high-level controller.

Simulation of walking robots can be challenging, in part due to the intermittent nature of the contacts, as the feet strike and leave the ground. Chapter §6 covers some of the more interesting features found in these simulations. I developed my own time-stepping simulator for Ranger, which is specialized for handling rolling contact of smooth irregular shapes. It is described in Section §2.16.

Many of the smaller control projects that I worked on throughout my graduate research required trajectory optimization, which is a numerical method that computes an open-loop solution to the optimal control problem. For example, trajectory optimization can be used to find a desirable walking gait for a robot. Over the past few years I developed a general-purpose trajectory optimization software, which I've since made open-source (§1.8). I took all that I learned over the years of doing trajectory optimization and wrote a tutorial and review paper for some of the standard trajectory optimization methods and how they were implemented (§7). Finally, I developed my own specialized trajectory optimization algorithm, which is described in Chapter §8, and will be submitted to a peer-reviewed journal.

1.1 Automatic Controller Design on Ranger

The central research project for my dissertation is my walking controller design for the Cornell Ranger walking robot, presented in Chapter §2. There are two key ideas for this project. First, the walking controller for Ranger should be robust: the robot should avoid falling when subject to (reasonable) unmodeled disturbances. The second idea is that the controller design *process* should be as automatic as possible, using computer opti-

mization rather than human intuition. This chapter will be submitted to a peer reviewed journal, after slight modifications.

At a high level, we work to achieve 'automatic' controller design by solving two optimization problems, one using a simulation of the Ranger, and the second one using experiments with the real robot. In order to solve these optimization problems, we must first construct a control architecture which can be optimized. A perfectly general controller (for example using a large neural net or dense look-up table) would have too many free parameters for any reasonable optimization. Instead, we use human intuition to develop a control architecture that is fairly general, but with a low-dimensional parameterization.

Another important aspect of the controller design process is the objective function for the optimization problem. We search for a 'robust' walking controller, which is not a well-defined concept. As a proxy, we find a controller that can regulate some desired walking speed while perfectly rejecting a representative set of disturbances. In this case, we use sloping ground and pushes (in simulation) to represent the set of all distributes that the real robot might be subject to.

At the end of the project, we were able to design a fairly robust controller for Ranger, as measured by a variety of tests on the physical robot. Although we did use computer optimization to find the important parameters of the controller, we still required human design to construct the controller architecture and the objective function for the optimization. The design process was iterative: the result of an optimization would often highlight a failure in the controller architecture or objective function, which we could then use to construct an optimization problem that was more representative of the desired behavior.

1.2 Off-line controller design for reliable walking of Ranger

The preliminary research for the Ranger controller design process is covered in Chapter §3, and will be presented at the 2016 International Conference on Robotics and Automation under the title *"Off-line controller design for reliable walking of Ranger"*. When compared with the *"Automatic Controller Design on Ranger"* paper, this paper has a less sophisticated controller and does not include the online optimization part of the controller design. It also focuses more on the controller architecture, rather than the details of the controller design process.

1.3 Non-linear robust control for inverted-pendulum 2D walking

Many walking robots rely on simple controllers for planning their high-level balance, especially for foot-placement. I present a robust controller for an inverted pendulum model of walking in Chapter §4. It can prevent falls despite large (but bounded) errors in sensing, modeling, and actuation. This research was presented at the 2015 International Conference on Robotics and Automation under the title: *"Non-linear robust control for inverted-pendulum 2D walking"*.

1.4 Swing Leg Controller

The robust controller presented in Chapter §4 prescribes a sequence of foot step locations. The follow-up project, presented in Chapter §5, is a robust swing-leg controller for a double-pendulum model of walking, which is used to achieve a desired foot-placement location.

I used trajectory optimization to construct a library of optimal reference trajectories. At run time, the controller computes a reference trajectory by interpolation from this library, and then uses a phase-space trajectory tracking controller to achieve the desired motion. The tracking controller is constructed by solving the finite-horizon continuous-time linear quadratic regulator problem. I verify the performance of the controller by performing a reachability analysis of the closed-loop system.

1.5 Simulations with Sliding and Intermittent Contact

Designing robust controller for walking robots often involves making simulations of these systems. These simulations are interesting because the dynamics of the robot change whenever a foot makes or breaks contact with the ground. Early on in my graduate research I did several small projects to study these simulations with intermittent contact, which are collected and discussed in Chapter §6. One of the more interesting projects is looking at the difference between the assumption that a contact point does not slip and the assumption that there is an infinite coefficient of friction at the contact point. Understanding these nuances of simulation was critical when developing my time-stepping simulation of Ranger (§2.16).

1.6 An introduction to trajectory optimization: how to do your own direct collocation

During the course of my graduate research I have spent a huge amount of time using trajectory optimization to compute desired gaits for walking robots. I found that there

were very few resources for learning the basics of the subject. The few texts and papers that were available were generally too focused and assumed a high-level of domain knowledge. After spending years learning about this topic I decided to write this tutorial paper, targeted at first-year graduate students that are interested in learning about trajectory optimization. It starts by providing a broad overview of all methods that are used for trajectory optimization, and then provides in-depth implementation details for two direct collocation methods. The second part of the paper shows how to solve four example problems using these methods.

This paper is titled *"An introduction to trajectory optimization: how to do your own direct collocation"* and has been submitted to Siam Review for publication. The full text of the paper is reproduced here in Chapter §7.

I wrote an electronic supplement to accompany the paper. It includes a full-featured trajectory optimization library (see §1.8) that I wrote in Matlab. The supplement also includes code to solve each of the four example problems in the paper, as well as several additional examples. The source code for the entire supplement is well-documented, as it is intended to be read as a tutorial itself.

I have given two seminar talks on the topic of trajectory optimization, and both were well received by the audience. The first was at the Cornell Robotics in Society seminar (November 2015) and the second was the Cornell Scientific Computation and Numerics seminar (March 2016). I posted the slides for the presentation on my website, and have received several emails from graduate students at other institutions indicating that the slides were helpful.

1.7 DirCol5i: a method for computing minimum-snap trajectories for systems with non-trivial dynamics

I developed my own trajectory optimization method, DirCol5i. Initially, I used the algorithm for computing optimal trajectories of second-order dynamical systems written in implicit form, but then discovered that it was quite good at computing minimum-jerk and minimum-snap trajectories. The algorithm is presented in Chapter §8 and will be submitted to a peer-reviewed journal after slight modifications.

The key part of all trajectory optimization methods is *transcription*: the process by which an optimization over the space of functions is converted to an optimization over the space of real numbers. There are two important differences between DirCol5i and standard trajectory optimization problems. The first is that DirCol5i uses an implicit second-order form of the dynamics, a generalization of the explicit first-order form used by most methods. Second, DirCol5i allows the user to specify an objective function that includes higher derivatives of both the state and control, with minimal computational overhead. Standard trajectory optimization methods require that the objective function only depend on time, state, and the control (not their derivatives).

DirCol5i works by approximating the position trajectory $x(t)$ using a quintic (5^{th}-order) spline, represented in Hermite form. The derivatives of the position: velocity, acceleration, jerk, and snap, can all be computed by analytic differentiation of the position spline. The control is represented using a cubic Hermite spline, and control rate is obtained by analytic differentiation of that spline. The objective function is evaluated using Gauss-Lobatto quadrature, and the system dynamics are enforced at the Gauss-Lobatto collocation points.

I implemented DirCol5i as a general-purpose library in Matlab, including algorithms for error analysis and mesh refinement. I compared DirCol5i to standard methods on several different example problems and objective functions. In all cases DirCol5i computes an accurate solution that is consistent than those found with standard methods: trapezoid collocation, Chebyshev-Lobatto pseudospectral collocation, and GPOPS-II (a professionally-developed implementation of Radau orthogonal collocation). DirCol5i is a bit slower on problems with simple objective functions (minimum-torque), but is notably faster on minimum-snap objective functions. Additionally, DirCol5i solves minimum-snap problems natively, without the need for reformulating the problem as required by the standard solvers.

1.8 TrajOpt: A Trajectory Optimization Library for Matlab

While doing research with trajectory optimization, I developed my own general-purpose trajectory optimization library, which I'm calling TrajOpt for now. As far as I can tell, TrajOpt is the only open-source (and free) trajectory optimization library for Matlab. The project hosted on GitHub:
`github.com/MatthewPeterKelly/TrajOpt`

One reason that I wrote my own software is that commercial solvers, such as GPOPS-II typically have only one type of method. It is often useful to select a method-type based on the problem. For example, high-order orthogonal collocation is the best method for problems that have smooth solutions, but low-order direct collocation is superior for problems with non-smooth solutions, such as those with path constraints. TrajOpt includes four different methods, which can be called using the same standard interface. Another important reason for writing my own software was to learn how tra-

jectory optimization works. It turns out that solving a difficult trajectory optimization problem, such as for a walking robot, often requires a good understanding of trajectory optimization to debug.

When writing TrajOpt I was careful to make the code well-documented, and it is intended to be read as a tutorial about trajectory optimization. After making TrajOpt open-source I've received several emails from appreciative graduate students at other institutions who used my software to learn how trajectory optimization works.

TrajOpt contains four different trajectory methods: trapezoidal direct collocation, Hermite-Simpson direct collocation, Chebyshev Lobatto pseudospectral collocation, and (4^{th}-order) Runge–Kutta multiple shooting. I also provide a wrapper for calling GPOPS-II. Most methods include the ability to use analytic gradients and compute an error analysis for the solution mesh. The source code currently includes 11 examples, which show how to use TrajOpt on a range of problems, from a simple pendulum swing-up to a gait for a 5-link walking robot.

CHAPTER 2

AUTOMATIC CONTROLLER DESIGN ON RANGER

This chapter submitted to a peer reviewed journal after minor modifications.

Author List: Matthew P. Kelly, Matthew Sheen, Andy Ruina.

2.1 Abstract

In this paper we are interested in using automatic methods to design a walking controller for the Cornell Ranger walking robot. We use hierarchical control structure, with a high-level balance controller plans overall step characteristics and a lower-level joint controller coordinates the individual motors to achieve the needed limb movements. The balance controller is designed using two rounds of optimization. The first round is done using off-line optimization of the controller in simulation, which is then followed by an on-line optimization on the physical robot to fine-tune the controller parameters. We show that the resulting controller design process is able to produce reliable walking gaits on Ranger. Although much of the design process was automatic, we still used human intuition to design the controller architecture and low-level motor control. In addition to our final results, we present and discuss many of the intermediate controllers that we developed, showing how we arrived at the final control solution.

2.2 Introduction

In this paper, we develop and demonstrate automated techniques for designing a walking controller for Cornell Ranger, a bipedal walking robot shown in Figure 2.1. Here

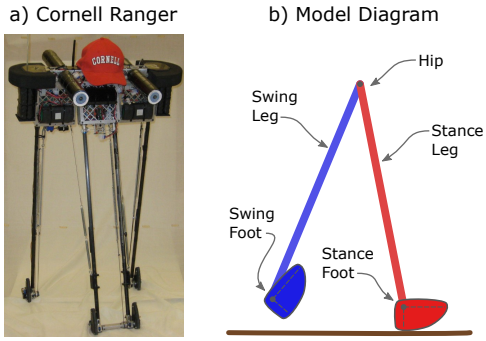

Figure 2.1: **Cornell Ranger walking robot:** A photo (a) and diagram (b) of Cornell Ranger, our experimental test platform for the controller.

we extend the controller design process presented in [57], by improving the controller architecture, and adding a final on-line optimization step.

To advance the science of robot control, we need algorithms that do not have hand-tuning on the robot as a key final step. Queries of robot builders reveal that such hand tuning of hardware is all too common. One major reason for this hand tuning is that simulators cannot capture all aspects of a real robot, particularly for legged locomotion with a dynamic gait. As a result, there is typically a loss of performance (if the robot walks at all) when a controller is transferred from a simulated robot to a real robot. For Ranger's previous controller, a graduate student intimately familiar with the robot spent many hours tuning parameters by hand.

There has been some work in designing locomotion controllers on hardware from scratch, but these have generally been for small statically stable robots [126], or robots that cannot fall down [128]. In both of these cases, open-loop control strategies are effective. On the other hand, Ranger can only walk using a dynamic gait, and any successful controller must have some degree of feed-back build into it. Additionally, Ranger can easily be damaged by a fall, and cannot stand up by itself.

Because our simulations cannot capture all aspects of the real robot, control devel-

opment would ideally be on the real robot. Yet, the robot is fragile and physical walking trials are prohibitively slow. Here we present the compromise solution: we start by designing a good control architecture in simulation, and then using off-line optimization over this simulation to get a reasonable guess for the parameters of that control architecture. Then we transfer this controller to the real robot, and use on-line optimization to fine-tune the controller, compensating for any small differences between the simulation and the real robot.

Walking is a complicated task, so the controllers for most walking robots rely on hierarchical control architectures [125, 79, 26, 90, 60, 121, 117, 9, 49, 62]. These simplify the design process by reducing the number of free parameters that describe the controller. The control architecture that we use here incorporates ideas from the previous controller for the Cornell Ranger [9], SimBiCon [125], and hybrid zero dynamics[121]. The high-level gait control is based on a finite state machine which regulates speed and maintains balance, while the lower-level joint control is simply a proportional-derivative tracking controller on each joint.

It is impractical to automatically design every feature for a walking controller — the search space is simply too large for modern techniques. Instead, we break down the problem. The control architecture is manually designed based on our own experience and discussions with experts, as well as the literature. The gains in the low-level joint controllers experimentally determined, using standard methods [80]. That leaves the parameters of the balance controller, 15 numbers in our implementation, to be automatically selected using optimization. Changing these numbers has drastic effects on the robot's behavior – whether it walks or falls, the speed and symmetry of steps, foot clearance, foot-ground impact, and more. Hence we believe (and will try to demonstrate) that these parameters contain enough richness for optimizing a robust controller for Ranger.

In this paper we present the a walking controller for Ranger, as well as the entire design process for getting that controller working on the robot hardware. One key feature of this paper is that we discuss many of the intermediate steps in the design process, rather than simply jumping to the final control solution. Although we make the controller design process as automatic as possible, there is a great deal to be learned by understanding the various non-automatic design choices that were made.

The final result is a controller for Ranger that was designed largely using optimization, first in simulation and then on the physical robot. The new controller much easier to launch (start walking) than the previous (hand-tuned) controller, and it is better at rejecting some kinds of disturbances.

2.3 Background – Locomotion Control

In this section we present a brief overview of some of the more popular techniques for controlling bipedal walking robots. Simple models of walking are at the root of many controllers, so we will mention three of the more common ones: linear inverted pendulum (LIP), spring-loaded inverted pendulum (SLIP), and inverted pendulum (IP). Afterwards, we will briefly discuss a few control techniques: zero-moment point (ZMP), capture point, hybrid zero-dynamics (HZD), SimBiCon, and a previous Ranger controller.

A good overview of the fundamentals of walking control is presented in [85], and an overview of some of the more formal techniques is given in [41].

2.3.1 Linear Inverted Pendulum Model of Walking

In the linear inverted pendulum (LIP) model of walking, the body of the robot is treated as a point mass that is constrained to move in a constant-height plane above the ground. Given a fixed support point and perfectly satisfied hip-height constraint, the motion of such a robot is identical to that of an inverted pendulum, linearized about the vertical orientation.

The LIP model is widely used in bipedal locomotion control algorithms, especially in zero-moment point controllers (§2.3.4) and capture point controllers (§2.3.5). The linear equations of the LIP model are mathematically convenient, and have analytic tricks that make it simpler to predict the motion of the robot at any given point. The mathematical simplicity makes LIP good for fast, online trajectory optimizations (e.g. model-predictive control). Controllers that are walking with controllers based on LIP models walk with a crouch-gait, which allows them to track a fixed hip-height, and avoid the kinematic singularity associated with a fully-extended leg. Additionally, controllers using the LIP model of walking tend to have large feet, giving them additional control authority over the motion of their center of mass.

2.3.2 Spring-Loaded Inverted Pendulum Model of Walking

The Spring-Loaded Inverted-Pendulum (SLIP) model of bipedal locomotion (walking and running) is another common point-mass model. The basic idea is that the entire robot is represented by a point mass at the hips, and each leg is replaced by a linear spring (and sometimes a damper as well). The balance controller works by computing a desired spring constant for the leg, as well as the desired heel-strike time and angle. Then, a low-level controller computes the joint torques to achieve the desired effective

spring constant in the leg.

One interesting feature of the SLIP model is that it inherently allows for dynamic gaits and robots with small feet. Although the SLIP model can be used for walking control, it is more often used for controlling running gaits.

Recently, the robot ATRIAS [51], was designed to be well-modeled by the SLIP model. Since then, the robot has been effectively controlled based on the SLIP model [92].

2.3.3 Inverted Pendulum Model of Walking

The Inverted Pendulum (IP) model treats the robot as a point-mass pendulum with a rigid "leg". The motion is not linearized and, as a result, the hip follows circular arcs rather than fixed-height lines (as in LIP). IP models are used to describe straight-legged walking. IP model walking is most energetically-effective of the point-mass models and similar to how people walk [105]. However, the non-linear equations, the kinematic singularity when the knee locks, and the harder impacts (more impulsive) when the locked leg swings forward and hits the ground make the IP model less popular.

2.3.4 Zero Moment Point Control

Many of the first bipedal robots walked using *Zero-Moment Point* (ZMP) controllers [117] [118], and the core principles are still used today [15, 49, 100]. The idea behind ZMP controllers is that walking is a perturbation of standing. Robots with ZMP controllers tend to have statically stable gaits, and rely on large feet and ankle torques for

control.

The zero-moment point is the point on the ground where the sum of all moments due to the contact forces on the foot is zero. Although it is not strictly necessary, most ZMP controllers also assume that the robot is well-modeled by the linear inverted pendulum model of walking §2.3.1. At the high-level, a ZMP controller will construct a desired trajectory for the ZMP to follow, which corresponds to a desirable motion for the robot. The low-level joint controllers then compute the joint torques to track the ZMP.

Robots that walk using ZMP controllers must have large feet, so that they can use ankle torques to move the ZMP. With good foot placement, these ankle torques can be minimized [53]. These robot walk with bent knees at all times, to prevent the control singularity associated with a fully extended leg (which cannot be made longer, and small changes in knee angle have no effect on the linearized kinematics). The simple form of ZMP breaks down when the target ZMP leaves the support polygon of the feet, thus preventing dynamic gaits. As such, ZMP controllers cannot work on robots that have points or curved rails for feet.

In recent years, several researchers have expanded on the classical ZMP controller to develop *Resolved Momentum Control* [54, 66, 18]. It has been demonstrated to be effective in both simulation [66], and on the HRP-2 [54] and the MIT Atlas humanoid robots [18, 61].

2.3.5 Capture Point Control

The capture point walking control algorithm, developed by [90], seeks to improve upon ZMP controllers by allowing dynamic walking gaits. This control strategy is also known

as divergent component of motion control [25].

Given the LIP model of walking (§2.3.1), there is a single capture point for each state [90], which is the location for the support point such that the point-mass will come to a stop directly above that point. In many cases, it is not possible to place the swing foot at the capture point. A simple extension is to compute the sequence of support points such that the robot comes to a stop above the N^{th} point.

This architecture is of practical use, in that, at any point, the controller knows a sequence of foot placement locations which will bring the robot to a complete stop. Walking is achieved by continually looking ahead N steps in the controller, and placing the feet in the correct places [60, 89]. At any instant during walking, the robot can control the instantaneous center of pressure within the support polygon to adjust the center of mass trajectory to be consistent with future foot placements.

Controllers based on capture point are widely used on bipedal robots [26, 25]

2.3.6 Hybrid Zero Dynamics Control

Hybrid zero dynamics (HZD) controllers [121, 124, 1] work in a fundamentally different way than controllers that we've discussed so far. Rather then control by forcing the robot to behave like a simple model, a HZD controller uses virtual constraints to force the robot to behave like a complicated system with one degree of freedom (DoF). The structure of those constraints are chosen such that the hybrid system associated with that single DoF satisfies some desired motion plan.

The key idea here is that the control laws are parameterized by the single remaining degree of freedom (typically related to the position of the center of mass of the

robot), rather than time, like a traditional feed-back control law. Although the original implementation of HZD used high-bandwidth feedback control, some more recent implementations relaxed the control a bit to achieve stable walking with a bit more compliance [103]. HZD controllers have been used effectively on modern robots [104], and in simulations [99].

An overview of the Hybrid Zero Dynamics [121] control architecture is provided below.

- The robot is modeled as a strict hybrid dynamical system, with a single continuous phase (single stance) and a single collision map (heel-strike).
- The robot is under-actuated, either from a lack of ankle torques or from small limits on ankle torques.
- The dynamics of the robot can be written in terms of a single monotonic phase variable. This corresponds to the under-actuated degree of freedom, and is typically related to the position of the center of mass of the robot with respect to the location of the stance foot.
- Trajectory optimization is used to compute a reference trajectory for the system, in terms of the phase variable.
- Feedback linearization is then used to compute a phase-varying trajectory tracking controller.
- With tight control loops, the resulting system dynamics are one-dimensional (in phase).
- The reference trajectory is constructed such that collision map for the (not one-dimensional) system is stable.

2.3.7 SimBiCon

The *Sim*ple *Bi*ped *Con*troller, SimBiCon [125], was developed for generating realistic motions for bipedal walking in computer animations. It is based on two key ideas. First, is that walking can be described using a simple global finite state machine, with four states, visited in series over two successive walking steps. The second idea is that the walking gait can be entirely synthesized by computing control laws that mimic virtual spring-mass-dampers connecting important points on the character to points in space.

More recently, SimBiCon has been generalized by [107], and also used in a variety of simulation-based studies including [36, 79]. An overview of the basic SimBiCon algorithm [125] is included below.

- The basic control structure is a finite state machine (FSM) with a total of four states. These states are traversed in series, with two states per step. The controller (and biped) are typically left-right symmetric, making the second pair of states a mirror of the first pair.
- There are two types of transitions in the FSM: the transition from the first to the second state is based on a timer, while the transition from the second to the third state is based on foot contact.
- Inside of each state of the FSM, there is a proportional-derivative (PD) control law computing the torque in each joint. These controllers rarely reach their set-points, and are thus always operating in transients, with relatively low tracking gains.
- Balance is achieved by adjusting some of the tracking controller set-points, again based on a simple PD control law.
- The PD control laws are usually with respect to absolute angle of the links, but are constructed such that the resulting torques are possible without applying a net

external torque to the system.

- This basic control law can be extended to 3D walking models, as well as models with more links.
- Stationary standing can be achieved by allowing stance ankle torques.

2.3.8 Previous Ranger Controller

The previous controller for the Cornell Ranger [9, 10] was designed for low-energy walking over flat ground. Like SimBiCon, it is based on finite state machines, but there is a key difference: on Ranger, each joint controller is running its own local finite state machine. The overall walking motion is coordinated by the shared global state of the robot, including events such as heel strike (the instant when the swing-foot collides with the ground on each step).

Inside of each state of the finite state machines (FSM), the controllers are running compliant control, loosely tracking trajectories computed by off-line trajectory optimization. The stabilization in this controller comes from two features. The first is a global feed-back, once per step, that adjusts some of the low-level trajectory set-points. In particular, if the robot is moving too fast, the robot takes a bigger step (thus dissipating energy). If the robot is moving too slowly, then it pushes off harder with the trailing feet, thus adding more energy to the gait. The second feed-back mechanism is the event-triggered transitions between states in the FSM, switching controllers when the feet strike the ground.

2.4 Cornell Ranger

Our test robot is the Cornell Ranger, which is described in detail in [9, 10]. Here we will present only a short overview.

Ranger, shown in Figure 2.1, is at the bottom of the bipedal robot food chain. It was designed for low-energy walking over flat terrain. It has four legs that are arranged into an inner and outer pair (almost like a person on crutches). This arrangement means that the walking control only needs to stabilize front-to-back motions, as lateral motions are passively stabilized. The feet are small and curved, which renders the robot under-actuated by one degree of freedom.

2.4.1 Hardware

Ranger has a some unusual characteristics. First is the lack of knees, which forces all changes in effective leg length to come from rotations of the feet. The feet on Ranger are curved with small radius, which means that Ranger cannot statically balance with its feet together. Additionally, the circular curve of ranger's feet is truncated close to the heel. This truncated shape allows for the feet to rapidly clear the ground during swing, but also further limits the effective foot size and the associated range over which the center of pressure can be moved along the feet.

Ranger has a variety of sensors for estimating its own state (no vision, LIDAR, etc.). The feet have sensors that measure the force between the heel of the foot and the ankle joint, which we threshold to determine if a foot is in contact with the ground or not. Each joint has absolute encoders for both the motor and the end-effector. Finally, there is an IMU (gyro and accelerometers) located on the outer legs that we use for estimating

the absolute orientation and angular rate of the robot.

2.4.2 Model

Our model for the Cornell Ranger, is largely based on our previous work [11, 9, 10]. We assume that the robot is a planar biped, with four rigid bodies (outer legs, inner legs, outer feet, inner feet) that are connected by three motors (outer ankles, hip, and inner ankles). We also use a full electro-mechanical model of the motors and gear boxes.

There are two notable differences between the model used here and our previous model of Ranger. The first is that here we assume that the drive cables connecting the ankle motors to the feet are rigid, where as the previous work [9] treated them as stiff springs. Second, we model the shape of the foot as a quintic spline (periodic, with 6 segments), rather than as a complete circle, as illustrated in Figure 2.2.

2.4.3 Motor Model

For our simulation we use the motor model developed by Bhounsule [9] to compute the torque produced by the motors (2.1) as well as the electrical power consumed (2.2).

$$T = K_m I - C_1 \omega - (C_0 + \mu K_m |I|) \cdot \mathrm{sgn}(\omega) \tag{2.1}$$

$$P = (IR + \mathrm{sgn}(\omega) V_c + K_m \omega) \cdot I \tag{2.2}$$

The notation here is taken from [9]. In summary: K_m is the effective motor constant (including the gearbox), C_0, C_1, and μ are loss parameters, I is the electrical current, V_c is the voltage drop across the brushes in the motor, and ω is the output shaft speed after the gearbox.

2.4.4 Sensing & Estimation

While Ranger is walking, we are continuously computing an estimate of the state of the robot: absolute angle and rate for both legs and feet, as well as which feet are in contact with the ground.

Ranger has a good inertial measurement unit (IMU), which contains a rate gyro and an accelerometer. The IMU contains its own computer that uses low-level sensor fusion to compute an accurate estimate for the absolute angle and rate for the outer leg pair. The hip joint and ankle joints contain absolute encoders, as well as low-level algorithms for estimating angular rate. We run the angle and rate sensor data through Butterworth filters before using it, along with the IMU angle, to compute the absolute orientation of the feet and legs.

Part of the walking controller requires an estimate of the distance between the ankle joints of the robot, and this is computed using the absolute angle estimates for the legs, as well as basic model geometry.

Finally, we need to be able to determine if the robot's feet are in contact with the ground. Each foot is actually made of a vaguely "C"-shaped piece of metal, with a gap in the heel [9]. When the robot puts weight on the foot, this gap shrinks, and that change is detected using an optical sensor. There is one optical sensor on each foot.

To determine whether a pair of feet is in contact with the ground (supporting the weight of the robot), we use a Butterworth filter to clean up the raw data from the optical strain sensors in the feet, then add the result together to compute a rough estimate of the force acting between the heel and the ankle joint. This signal is then thresholded to determine if the foot is in contact with the ground. The threshold is set using simple experiments, rocking the robot back and forth between the pairs of feet, while holding

the feet at a range of possible contact angles.

2.4.5 Simulation

The previous simulator for Ranger, described in [9], was designed to study open-loop trajectories with a prescribed sequence of contact configurations. For the research presented in this paper, we need to study the close-loop behavior of the robot for a variety of controllers, some of which will cause the robot to stumble and fall down — a behavior that was not able to be captured by previous simulations,.

To capture more complex contact sequences, we developed a "time-stepping" simulator for Ranger, which runs a contact-solver on each time-step. This simulation allows us to model the robot walking over any ground profile, using accurate collision shapes for the robot's feet. The simulation is implemented in Matlab, and then compiled to MEX for speed. Appendix §2.16 discusses the simulation in detail. The source code for the simulator is available at: `https://github.com/MatthewPeterKelly/RangerSimulation`

2.4.6 Control Considerations

During the course of a walking step, the motors of the robot must add energy to the system to compensate for frictional and collisional losses. Due to the small curved feet, Ranger cannot inject much energy by ankle torques through the step, except by push-off with the back foot at the end of each step. This extension is small (a few centimeters), but enough to propel the robot forward and to adjust walking speed. To get the maximum effect of this push-off it must be timed carefully to occur just before the collision on the

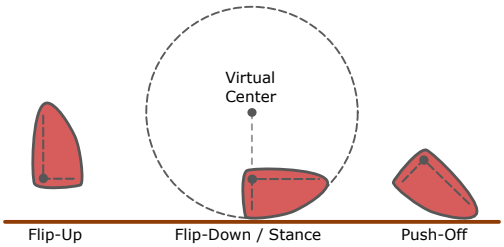

Figure 2.2: **Ranger Foot Diagram:** Ranger's feet are small, and their soles are sections of circular arcs. Here we show the three target configurations used by the controller. Flip-up is used for the swing foot, allowing the foot to clear the ground, since the robot has no knees. The flip-down/stance configuration is used by the stance foot for most of the step, providing a steady base for the robot. The final configuration, push-off is used to rapidly extend the foot, propelling the robot forward for the next step.

front foot.

Ranger does not have knees, and thus as soon as the push-off is complete, the foot needs to rotate up and out of the way so that it doesn't scuff as the swing leg moves forward. Then the foot needs to rotate back down just before heel-strike. Too early and the foot scuffs; too late and the foot strikes down on the back of the heel, which causes a trip, with the foot rotating back up. In each of these cases, the robot falls. See Figure 2.2 for details regarding foot orientation for push-off and foot-flip.

2.5 Controller Architecture

The control architecture for Ranger is divided into four levels, arranged highest-to-lowest:

- The *Balance Controller* runs once per step, at mid-stance, setting the five parameters that describe the gait controller.
- The *Gait Controller* is a finite-state machine, shown in Figure 2.3, which sets the target angles and rates in the joint controllers.

- The *Joint Controllers* are proportional-derivative controllers which compute the command torque for each joint.
- The *Motor Controllers* are proportional-integral controllers which compute the low-level PWM commands to achieve the desired torque in each joint.

We assume that both the robot and controller are symmetric. We will describe the controller for the case when the outer feet are on the ground. At the conclusion of the step, the whole controller is mirrored, with the inner legs becoming the new stance legs.

2.5.1 Motor Control

The motor controllers are at the bottom level of the controller. They run a simple proportional-integral control loop at 2 kHz on each of the three joint motors (outer ankle, inner ankle, and hip), tracking a desired joint torque. These motor controllers are coded at a low-level in the robot, and have not been changed since the robot was first built.

2.5.2 Joint Control

While the robot is walking, the joint controllers (outer ankle, inner ankle, and hip) are continuously running simple proportional-derivative (PD) controllers at 500 Hz. These controllers compute a command torque u, which is sent to the motor controllers. The reference angle q^*, rate $\dot{q}^*$, and torque u^* are all sent from the gait controller. The measured joint angle and rate are given by q and $\dot{q}$.

$$u = u^* + K_P\left(q^* - q\right) + K_D\left(\dot{q}^* - \dot{q}\right) \tag{2.3}$$

Conceptually, these PD controllers are performing simple behaviors for each joint, as described below.

Hip Scissor Track: During walking, the hip joint performs scissor-tracking to bring the swing leg through the each step. In this mode, the references for the hip joint are selected such that the absolute angle of the swing leg tracks an affine (linear + constant) function of the absolute angle of the stance leg.

Hip Drop Ankle: During the end of each step, it is desirable for the swing foot to descend directly towards the ground, to avoid a glancing collision and achieve the desired step length. To do this, we compute the references for the hip joint such that the ankle joint will follow a vertical trajectory (in absolute coordinates) towards the desired step location.

Ankle Stance Hold: During the majority of the step, the stance foot is supporting the weight of the robot. This is done by selecting a reference set for the ankle joint such that the absolute orientation of the stance foot is held constant. The target absolute angle is selected such that when both legs are vertical no ankle torques are required to support the robot.

Ankle Stance Push: At the end of each step, the stance feet rotate forward to propel the robot onto the next step. This is accomplished by tracking a target absolute angle for the stance foot, where that angle is selected by the balance controller to achieve the desired effect on the gait.

Ankle Swing Flip: The swing foot of the robot should always be flipped up and out of the way to keep it from scuffing on the ground during walking. This is achieved by tracking a constant relative ankle joint angle, such that the heel of the foot is above the ankle joint (near the hard stop of the joint).

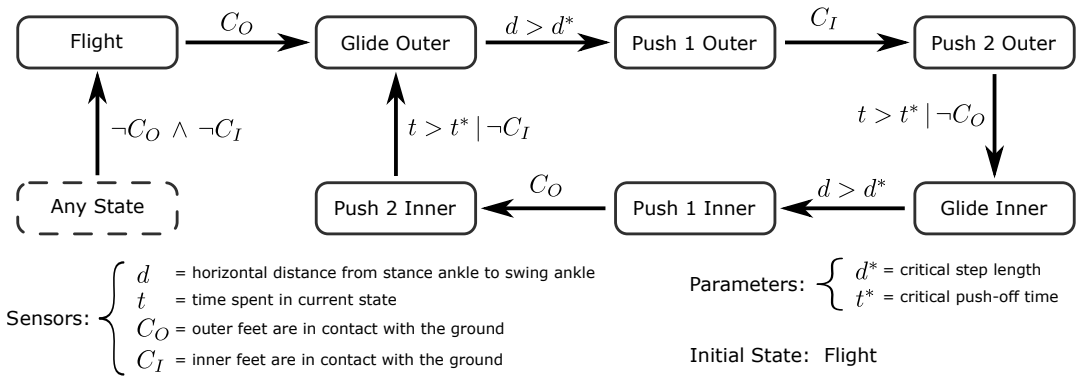

Figure 2.3: **Gait Controller:** The gait controller for Ranger is a finite state machine with the seven states shown above. The transitions are based on the the contact sensors for the feet, the distance between the swing and stance ankle joints, and a timer. The Glide and Push states are mirrored precisely between the inner and outer feet. The Flight mode typically is triggered by the user picking up the robot, and allows for the robot to be easily reset between walking bouts.

Safe Mode: Whenever the robot detects that a fall is imminent it enters safe mode. Upon entering safe mode the robot records the orientation of all joints on the robot. Each motor is then set to perform a weak compliant control about this joint orientation. This prevents the robot from doing something unexpected when it falls, thus making it easier for the user to catch (and reset) the robot. It also minimizes the risk of damage to the robot or harm to the user.

2.5.3 Gait Control

The gait controller coordinates the motion of the three joints on the robot, sending reference commands to the joint controllers at 500 Hz. The gait controller is a finite-state-machine (FSM), shown in Figure 2.3. When in each state of the FSM, the robot is setting a particular target behavior for each joint controller, which are implemented using proportional-derivative control laws. See §2.5.2 for descriptions of each joint behavior.

Flight Mode: Whenever the robot turns on or is picked up in the air, it enters flight mode. The purpose of this mode is to put the robot in a relatively passive state from which it can easily start walking. The hip motors are turned off, the inner feet are flipped up, and the outer feet are in stance hold, prepared to support the weight of the robot when it is placed on the ground.

Glide Outer: In this mode the outer feet are in stance hold, the inner feet are flipped-up, and the hip is performing scissor tracking, maintaining a desired absolute orientation of the inner legs as a function of the outer leg angle.

Push Outer (1 & 2): The outer feet are in push mode, tracking an absolute reference angle that pushes the outer toes into the ground, propelling the robot forward. The inner feet are entering stance mode, preparing to support the weight of the robot when they hit the ground. The hip is computing targets such that the inner feet hit the correct place on the ground.

Glide Inner: In this mode the inner feet are in stance hold, the outer feet are flipped-up, and the hip is performing scissor tracking, maintaining a desired absolute orientation of the outer legs as a function of the inner leg angle.

Push Inner (1 & 2): The inner feet are in push mode, tracking an absolute reference angle that pushes the inner toes into the ground, propelling the robot forward. The outer feet are entering stance mode, preparing to support the weight of the robot when they hit the ground. The hip is computing targets such that the outer feet hit the correct place on the ground.

Transitions: There are three different types of transitions used in this FSM, based on contact sensors, state estimates, and timers. If the robot is picked up in the air, then both contact sensors register false and the robot enters flight mode. During normal walking,

the contact sensors are primarily used to trigger the transition into the Push 2 mode. The robot spends very little time in the Push 2 mode, exiting when a timer reaches a threshold (set by the balance controller). The final transition occurs between Glide and Push 1 mode, and is triggered when the swing foot reaches the desired step length (while still in the air).

Parameters: The gait controller FSM has two parameters that are directly set by the balance controller. The first is the desired step length, and the second is the time-out on Push-2 mode. There are three additional control parameters, which affect the low-level controller behaviors. Two are used to compute the reference trajectory for the hip scissor tracking, and the final parameter sets the reference angle for the push-off.

2.5.4 Balance Control

The top-level of the control architecture is the balance control, which runs once per step at mid-stance. It changes the five parameters of the gait controller to regulate balance and walking speed. For example, if the robot is walking too slowly, it will increase the reference angle for the push off, adding more energy to the system.

There is a single input the the balance controller: the robot's speed at mid-stance. In this way, the balance controller is simply a function that maps the mid-stance speed of the robot at mid-stance to the set of five parameters that are passed to the gait controller. Here, we implement this function using a look-up table, storing the five parameter values for zero speed, the target speed, and the maximum speed. For intermediate speeds we use linear interpolation.

The look-up table for the balance controller has a total of 15 entries (5 parameters

for each of 3 speeds), which we compute using both off-line and on-line optimization, using methods discussed in §2.6, §2.7, and §2.8.

2.6 Controller Design

There are many aspects to the design of the walking controller presented in this paper. We claim that the controller is designed "using optimization", but we also acknowledge that there are many decisions that must be made by humans as well. Here we divide the control design process into different categories: some aspects are designed by human intuition, some by simple experiences, and some by optimization (both on-line and off-line).

We designed the architecture for the walking controller (§2.5) through insight and intuition, which we acquired through experiments, discussions at technical conferences, and the literature [9, 125, 121].

Within the controller architecture there are several constant parameters that are manually set. These include the orientation of the stance foot during swing (selected such that the ankle joint lies directly below the virtual center of the foot), and the relative angle of the ankle joint required during flip up (selected to be close to the joint limit).

The joint controllers (§2.5.2) in the ankles and hip all have proportional and derivative gains, which are selected by simple experiments on the hardware. These experiments are easily repeatable by any controls engineer using standard methods [80].

The remaining set of parameters, as described in §2.5.3 and §2.5.4, are selected entirely by computer optimization. These parameters include things like how the step length should be changed to stabilize walking, and the parameters of the swing-leg tra-

jectory. We use two rounds of optimization: first is the 'off-line' optimization (Section §2.7), in which we optimize the controller using a simulation of the robot. The controller from the off-line optimization is then directly transferred to the real robot (no intermediate hand-tuning), and it can begin walking. The final step in the controller design process is to run an 'on-line' optimization (Section §2.8) on the physical robot, which automatically performs final tuning of the controller, correcting for modeling errors in the simulation.

2.6.1 Designing the controller architecture

The controller architecture that we present in Section S2.5 was designed largely by intuition and experimentation. It has been changed several times over the course of this research, and will likely be further improved, as discussed in in the section on future work (§2.14).

The original version of the control architecture was developed entirely using the simulation of Ranger, and was based on ideas from three existing walking control architectures: SimBiCon [79], Hybrid Zero Dynamics [121], and Ranger's marathon controller [9]. Our general control architecture, made up of a single, simple finite state machine was borrowed from the SimBiCon controller. The swing-leg (hip joint) controller was inspired by the phase-variable tracking controllers in hybrid zero dynamics. Finally, many of the transition conditions, as well as the structure for Ranger-specific features (like push-off and foot-flip), were borrowed from Ranger's marathon controller.

Once we had a control architecture that worked in simulation, we transfered it to the robot to continue development. Most of the changes after this point were to improve usability: for example, making the robot enter a 'safe-mode' when a fall is detected,

or making it go into 'flight-mode' when it is picked up. We made two changes to the controller architecture since out original work [57]; we changed the event that triggers push-off, and allow push-off to continue after heel-strike.

The timing of push-off (and flip-down) is critical to making Ranger walk well. If the push-off occurs too early, then it picks the robot up, and the too-early flip-down can cause the swing foot to scuff the ground; if it occurs too late, then the robot won't have enough energy to complete the following step (and it can be damaged by landing on the exposed ankle joint). The original push-off trigger was based on the angle of the stance leg. The problem was that this condition did not depend on the position of the swing foot, which is more important for ensuring that the step is completed successfully. We changed the controller to trigger when the swing foot is in the correct location, which made the controller more robust to disturbances.

2.6.2 Simple (non-optimized) controller parameters

There are a variety of parameters in the walking controller which are designed through simple experiments, rather than as part of the optimization. These include things like joint-limits, thresholds for contact sensors, and tracking controller gains.

Foot-flip Target: During each step, the swing foot of the robot must 'flip-up' to avoid scuffing. The precise angle does not matter, so long as the foot clears the ground. In this case we make the target angle about 10° from the hard stop, providing adequate clearance while avoiding collisions with the hard stop due to controller over-shoot.

Contact Sensor Threshold: The threshold for the contact sensors is computed experimentally, since they rely on complicated un-modeled features in the feet. The threshold

was computed by locking (with a tight PD-controller) both feet in stance configuration and rocking the robot back-and-forth. We then measured the raw signal from the contact sensors. We selected a combination of Butterworth filter time-constant and threshold such that the resulting contact state was reliably triggered every time the foot hit the ground, without sending false positives.

Joint Rate Filters: There is a Butterworth filter running on each angle-rate sensor, with experimentally-determined cut-off frequency. Aside: the joint rates are computed using low-level pulse-counting and timing, rather than by direct differentiation of of the joint angles.

Joint Tracking Gains: At a low-level, each joint is running a PD controller. The hip controller always uses the same gains, while the ankle controller uses different gains, depending on whether the foot is in contact with the ground or not. These gains were all set using simple one-dimensional experiments and basic control theory. In each case, the joint was given a reference signal to track while in the expected configuration: The hip joint was tested while tracking a swing-leg trajectory in single-stance, the stance ankle controller was tested while supporting the weight of the robot, and the swing ankle controller was tested while in the air.

2.6.3 Optimized controller parameters

The remaining controller parameters were computed using optimization, as discussed in sections §2.7 and §2.8. The off-line optimization includes all parameters, while the on-line optimization includes only a sub-set, due to the limited number of experiments that can be run in the real world.

2.7 Off-line Optimization

The off-line optimization for the controller is searching the a set of 15 controller parameters that, at a high-level, stabilize the walking gait. These $15 = 5 \times 3$ parameters are the set of 5 free parameters in the gait controller, for each of 3 different speeds in the look-up table. The best set of parameters are those that regulate speed and prevent the robot from falling, despite the presence of disturbances. Each candidate set of parameters is evaluated by running a set of simulations, and then aggregating the performance of the controller across those simulations. We perform the optimization using CMA-ES, described in Section §2.7.2.

2.7.1 Objective Function for Off-line optimization

Our goal is to compute the controller parameters such that Ranger achieves a desired walking speed and can avoid falling. To quantify this, we design an objective function that has the simulation of Ranger walk several different trials in the presence of disturbances. A trial is considered to be successful if the simulated Ranger completes a prescribed number of steps without falling.

We consider two types of disturbances for the controller design process, which we believe are representative of the much larger set of disturbances facing the real robot. The first type of disturbance is a constant ground slope, either up-hill or down-hill. The second type of disturbance is a 'push' which is applied at some predefined time in in the trial. In all cases we introduce disturbances in symmetric pairs: if the candidate controller has to walk up-hill, then it also will have to walk down-hill, thus avoiding a bias in the control design process.

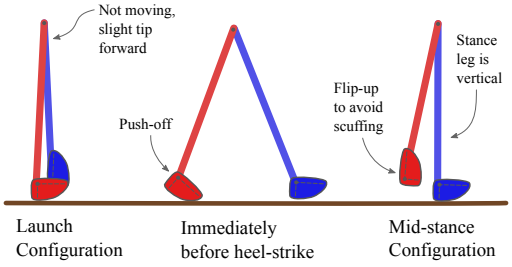

Figure 2.4: **Ranger Configurations** The launch configuration (left) shows the static configuration that we use to start Ranger walking, both in simulation and in reality. The middle configuration shows the robot immediately before heel-strike. The mid-stance configuration (right) is when the supporting leg vertical. This is the configuration that triggers an update from the balance controller.

We select the ground slope disturbance because it is particularly challenging for Ranger, and it is something that the robot will face in the real world, since few floors are actually flat. The push-disturbance is selected as a catch-all for general modeling, sensing, or actuation errors. In this case, we model the push by applying a constant force to the hip of the robot, for a selected duration.

Each of the simulations used when evaluating a candidate controller start from the same launch configuration, shown in Figure 2.4, where the robot is balanced at mid-stance with a slight tip forward. The first simulation is always under ideal conditions, followed by ground slope trials, and then simulations with pushes. We require that the controller successfully finish all simulations under ideal conditions and ground slopes before it is allowed to attempt the simulations with pushes.

The total score for the objective function is determined by summing the cost incurred during every step in the walking trials. Any steps that were not attempted due to a fall are given a speed of zero. All other steps are evaluated based on the average walking speed during that step. The cost per step is given below.

$$\text{Speed Cost per Step} = \left(1 - \frac{\text{measured speed}}{\text{target speed}}\right)^2 \tag{2.4}$$

2.7.2 Off-line Optimization Method: CMA-ES

Although the choice of optimization method is not central to our design philosophy, it happens that here we used a Covariance Matrix Adaptation Evolutionary Strategy (CMA-ES) [44, 45] because it deals well with our non-smooth objective function. We initialize the algorithm by first estimating bounds on the parameters. For example, the push-off target angle must be within the actuator limits, and the hip trajectory coefficients should be roughly consistent with bipedal walking (the swing leg must travel from back to front, etc.).

We also ran the off-line optimization using Particle Swarm Optimization (PSO), which is the same optimization technique that we used for the on-line optimization. It produced similar results to CMA-ES, although PSO took a bit longer to run.

2.8 On-line Optimization

Once we complete the off-line optimization, the control parameters can be directly transfered to Ranger is it is able to walk. The goal of the on-line optimization is to take this controller and make small changes to compensate for modeling differences between the simulation and the real robot.

2.8.1 Objective Function for On-line Optimization

The objective function of the on-line optimization has a simple goal: walk at a target speed without falling down. This objective function is similar to that used in the off-line simulation, but with one key difference: the on-line optimization only evaluates

the controller on a single walking bout, rather than running many separate simulations. The key idea here is that there is no need to inject disturbances into the real-world experiments, since there will be disturbances just from walking on the (slightly uneven) stone-tiled floor.

Each time that Ranger completes a step (mid-stance to mid-stance without falling), that step receives a score, computed below, where v_i is the average walking speed over that step, and v_T is the target walking speed.

$$\varepsilon_i = \left(\frac{v_i - v_T}{v_T} \right)^2 \tag{2.5}$$

$$\text{cost} = \frac{1}{N} \sum_{i=1}^{N} \varepsilon_i \tag{2.6}$$

Another difference between the off-line and on-line optimization is that the on-line optimization trials begin from steady-state walking. To do this, each trial begins with a transient period (usually 4-5 steps) that is ignored by the optimization, followed by the main trial (usually 10-15 steps). If all is going well, then the optimization will automatically load and test controllers back-to-back. Otherwise, the operator can start a trial from the launch configuration. In both cases, the controller will be at steady-state walking by the end of the transient period. Just like the off-line optimization, any steps that are not completed due to fall are given a speed of $v_i = 0$, thus creating a large penalty for falling.

2.8.2 On-line Optimization Method: PSO

For the on-line optimization of the controller, we use particle swarm optimization (PSO) [22, 16, 2]. PSO is a heuristic non-linear optimization algorithm, based on simple models of the foraging behavior of bird flocks. Our primary reason for selecting PSO is that

the update equations are simple, making it easy to implement on the small processor that is running Ranger. It also has the advantage that any new information for the objective is used immediately to guide the search, rather than waiting for a batch update once per generation.

PSO searches a space using a population (swarm) of particles, which move around according to set of stochastic update equations. The motion of any given particle is dependent on it's position, current search direction, as well as the its own best-ever position, and the global best position. The i^{th} particle has a position $\boldsymbol{x}^i$ and a "velocity" $\boldsymbol{v}^i$. The particle's best-ever position is given by $\boldsymbol{x}_L^i$ and the swarm's best-ever position is given by $\boldsymbol{x}_G$. The update equation for a particle is given below, where ω, α, and β are constant search parameters, and $R_\alpha \in U(0,1)$ and $R_\beta \in U(0,1)$ are random numbers drawn from a uniform distribution between zero and one.

$$\boldsymbol{v}_{\text{new}}^i = \omega \boldsymbol{v}^i + \alpha R_\alpha (\boldsymbol{x}_L^i - \boldsymbol{x}^i) + \beta R_\beta (\boldsymbol{x}_G - \boldsymbol{x}^i) \tag{2.7}$$

$$\boldsymbol{x}_{\text{new}}^i = \boldsymbol{x}^i + \boldsymbol{v}_{\text{new}}^i \tag{2.8}$$

The parameters ω, α, and β adjust the dynamics of the search [16]. Note that this Equation 2.8 can be thought of as doing an Euler integration step on the dynamics of a particle attached to two points with springs. Like all heuristic optimization algorithms, there is a trade-off between exploration (finding new local minima) and exploitation (converging towards existing local minima). Here, we choose parameters to favor exploitation, since we assume that the off-line optimization has gotten us close to an good solution.

We use a warm-start version of the algorithm, where we can specify an initial guess, which is used as the position of the first particle. This lets us start the on-line optimization using the best controller found in the off-line optimization.

2.8.3 Dimension Reduction

The off-line optimization had a 15-dimensional search space, computing each of the 5 parameters for each of the 3 speeds in the table for interpolation. This search space is simply too large for a reasonable search using online optimization. To reduce the search space, we observe that once the robot is walking, its behavior is largely governed by the set of parameters that are associated with the target walking speed. Thus, we only optimize this smaller set of 5 parameters for the online optimization, and use the other 10 parameters exactly as they came out of the offline optimization.

2.8.4 Dealing with Noise

Since this code is running on a real robot walking around in an environment with disturbances, we expect there to be some "noise" terms in the value of the objective function. There is some work on handling noisy objective functions with PSO [86], but the implementation requires significant computation, which is not practical on Ranger's small on-board computer. Instead, we experimented with two simple heuristics.

The first thing that we tried was to run each trial twice, and then report only the worse of the two objective function values. This is effective, since PSO is far more sensitive to false-positives (noise creating an artificially good score) than to false-negatives. The major down-side of this approach to noise reduction is that it doubled the length of time required for any optimization. Since the optimization runs are time-limited, this practically meant that we ran the optimization for half as many generations.

Our second, and perhaps better, method for reducing the effect of noise was to just increase the number of steps required in a trial. The key difference is that, unlike the

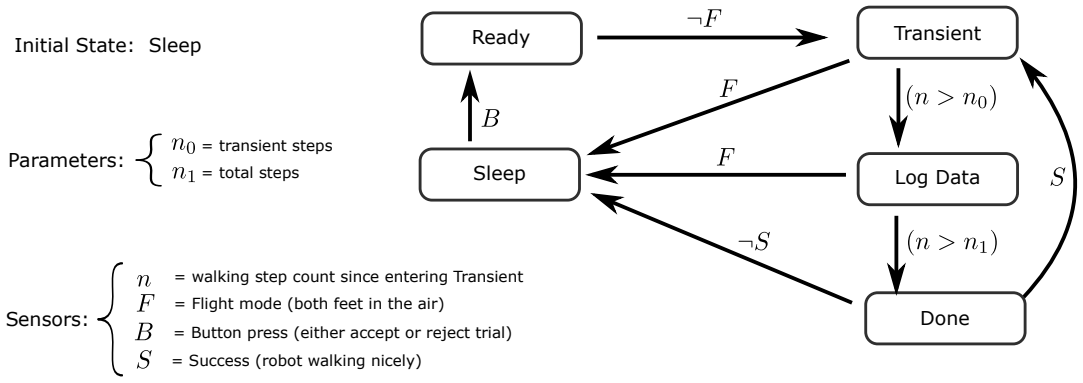

Figure 2.5: **Walking Experiment FSM:** The is the finite state machine that runs the on-line walking experiments. Each trial starts by giving the robot a few steps to reach steady state before it starts logging data. If the robot is walking well at the end of the trial then it will automatically send the objective function value to the optimization and start a new trial. If something goes wrong, then the robot will wait for input from the user. The user can then accept the trial (send objective function value to optimization), or reject the trial (re-run the trial). This is important because we are testing in busy atrium, where the trials are interrupted whenever the robot reaches a wall or a pedestrian.

repeated trial method, we did not need to repeat the set of transient steps at the start of each trial.

2.8.5 Logistics for Walking Experiments

Our testing space for walking Ranger is a large open atrium, which has two logistical problems: 1) There are people walking around, and 2) the atrium has walls. Some of the walking experiments get interrupted when a person walks in front of the robot or the robot reaches a wall. In these situations, the experimenter can pick up Ranger, causing it to go into *flight-mode*. In flight-mode, the robot disables the hip motor, and orients the feet in preparation for starting a new walking trial.

Whenever a walking trial is interrupted, either be entering flight-mode or safe-mode, the optimization waits for a button press from the user to either accept the previous data

(e.g. the controller caused the robot to fall), or re-run the trial (eg. the robot reached a wall). This is critical, because it allows the optimization to distinguish unplanned interruptions from falls caused by the controller.

A the start of each trial there is a transient in the walking gait, either to speed the robot up from launch configuration, or just from the change in controller. To minimize the effect of this transient we allow each controller to take several steps and reach steady-state before logging data for the optimization.

At the conclusion of a walking trial, there are two possible outcomes. If the robot was walking well (as determined by speed and step length), then the optimization automatically loads a new controller. This allows for continuous walking of the robot and more efficient experiments. On the other hand, if the robot finished the trial, but was walking badly (eg. very small steps or low speed) then the optimization will signal the experimented to pick up the robot and start the next trial from the default launch configuration. Figure 2.5 shows the finite-state-machine that we used for running the on-line optimization experiments.

2.9 Overview of Experiments

In this paper we have included a variety of our initial experiments, as well as the final experiments, in an effort to show the entire design process. As such, we've ended up with a long list of optimizations, control architectures, and experiments, which are summarized here.

2.9.1 Walking Controllers

- *Marathon Controller* was the exact same controller that was used for Ranger's 65 km marathon walk in 2011 [11, 9, 10]. It was developed by off-line controller optimization (very different than the methods presented here) followed by hand-tuning on the real robot. We use this controller as a benchmark for robustness testing.
- *ICRA Controller* was the controller that we presented at the 2016 International Conference on Robotics and Automation [57]. The ICRA controller is similar to the controller presented in this paper, but without the push-off delay (*ie.* $t^* = 0$ in Figure 2.3). The ICRA controller also uses a critical angle of the stance leg to trigger push-off, rather than a critical step length.
- *Controller 0* was an experimental controller that was designed using hand-tuning, to test the control architecture. Like the ICRA controller, it used a critical stance angle to transition from Glide to Push-off, rather than a critical step length.
- *Controller 1* was another experimental controller designed with hand-tuning. In this controller, we used the integral of the ankle motor current, rather than a timer, to control the transition from Push-off to Glide mode.
- *Controller 2* was the first controller with the exact architecture described in Section §2.5.3. It was designed using off-line optimization one, and used in robustness test one.
- *Controller 3* has the architecture described in Section §2.5.3, and was designed using Off-line optimization Two.
- *Controller 0*, 1*, 3*:* were used as the initial controller for on-line optimization one, two, and three respectively. The best controller produced by each of these

on-line optimizations is named Controller 0^*, 1^*, and 3^* respectively.

Both controller 0 and controller 1 were designed using hand-tuning, to understand various trade-offs in the control architecture. Using that experience, as well as observations from on-line optimization one and two, we arrived at the control architecture presented in this paper, which is used for Controller 2 and Controller 3.

2.9.2 Off-line Optimization

- *Off-line Optimization One* was used to design Controller 2. It used shorter simulations, fewer simulations on sloping ground, and vertical push disturbances.
- *Off-line Optimization Two* was used to design Controller 3. It used longer simulations, more simulations on sloping ground, and horizontal push disturbances.

2.9.3 On-line Optimization

- *On-line Optimization One* was initialized with Controller 0, and produced Controller 0^*. It used short walking trials and a faster target walking speed.
- *On-line Optimization Two* was initialized with Controller 1, and produced Controller 1^*. It used long walking trials, which were each repeated, and a slower target walking speed.
- *On-line Optimization Three* was initialized with Controller 3, and produced Controller 3^*. It used long walking trials and a slower target walking speed.

2.9.4 Robustness Tests

- *Robustness Test One* compared the Marathon Controller, the ICRA Controller, and Controller 2. Each controller walked a long distance over a slightly uneven floor while subject to large modeling errors (added weights, removed hip spring).
- *Robustness Test Two* compared the Marathon Controller, Controller 3, and Controller 3^*. Each controller walked many short trials over uneven ground with masses added to the outer legs of the robot as a disturbances.

2.10 Results: Off-line Controller Optimization

We implemented the entire design process for the balance controller in Matlab, with most of the simulation code being compiled to MEX for faster run-time. We then ran two different sets of optimizations, using the off-line optimization architecture described in Section §2.7.

Off-line optimization one can be thought of as a pilot study; it used fewer disturbances and shorter simulations, and the goal was largely to learn about the optimization landscape. Off-line optimization two was the final set of optimizations, using more disturbances and longer simulations.

2.10.1 Off-line Optimization One

The objective function for off-line optimization one consisted of simulations of the robot walking under ideal circumstances, on sloping ground, and in the presences of pushes. The ground-slope trials were conducted on ground with a slope of ±0.01 radians, with

each trial consisting of 12 steps, starting from the static 'launch' configuration.

Each trial with a push started from steady-state walking, and the pushes were applied at either 1/3 or 2/3 of the way through a step, either straight up or straight down, for a total of four different disturbance trials. Each push was applied for 0.5 seconds, had a constant magnitude of 50 Newtons. Aside: After running the optimization we realized that a bug in the code was applying the force vertically rather than horizontally, as originally intended. Despite this, the push still provided a reasonable disturbance for controller testing.

In off-line optimization one, we were interested in learning how complicated the objective function is. For example, is there one clear best controller, or many different controllers that are of similar quality? To do this, we solved the same optimization problem many times, and compared the results. Since CMA-ES is stochastic, it will follow a different path to the solution on each new optimization run.

First, we solved the problem ten times, using a smaller population (20) and fewer generations (20). CMA-ES is a stochastic algorithm, so each run produces a different result. Next, we solved the same optimization problem once, using a large population (50) and running for many generations (100). We then studied the results, to see if the optimizations converged to the same solution. All optimizations were run on an Intel Quad-Core i5-3570K processor running at 3.40GHz.

In the batch run of the optimization (10 runs, 20 population, 20 generations), each run took about 45 minutes. We found that 8/10 converged to good solutions, 1/10 converged to a mediocre solution, and 1/10 got stuck in a bad local minimum. Of the good solutions, we found that there were several similar, but slightly different gaits.

The long optimization run (50 population, 100 generations) took just under 9 hours.

The population collapsed to a single local minimum near generation 65, and converged to a solution that was slightly better than all found by the batch-run of the optimization.

Controller 2 was the best point in the first of the ten smaller optimizations. Aside: we ran this first optimization, then did all of the testing, and later ran the remaining nine optimizations of the set as well as the long optimization to confirm that we had found a reasonable solution in that first optimization.

2.10.2 Off-line optimization Two

For the second set of off-line optimization experiments we did two large optimizations, running each for 40 generations with a population size of 80 in CMAES. We also modified one of the default parameters, making the size of the parent population 30 instead of the default of 40 = `floor(0.5*popSize)`.

Off-line optimization two uses the same basic structure as off-line optimization one, but with a few improvements. We increased the number of simulations walking on sloping ground (to avoid foot scuffing), and increased the length of all simulations (to improve steady-state walking). We fixed the push-disturbance force so that it was applied horizontally, rather than vertically (fixing a bug in the original version). Finally, we made the walking trials for the push-disturbances start from the launch configuration, rather than from steady state (making the robot better at initiating walking).

Each call to the objective function ran a set of simulations, each of which terminated when the robot fell down or reached 20 steps. In each case the robot started from its 'launch' configuration, at rest; a slight tip forward ensured that the robot would fall forward to start the first step. The first simulation was level ground walking without

disturbances, to make sure that the robot was able to walk well at the desired speed under ideal conditions. Next, the robot had to walk on sloping ground: $\pm 0.86^\circ$ and $\pm 1.15^\circ$. If the robot completed all trials so far (made it to 20 steps), then it would start the final set of simulations: pushes applied to the robot. All pushes were applied horizontally, and had a magnitude of 4N and a duration of 0.5s, and were applied at either 2.6s into the simulation or 2.8s into the simulation. Finally, each push was applied in the forward and reverse direction (on independent trials). Any steps that the robot failed to take, either due to falling or due to a incomplete first set of trials, was given a walking speed of zero, thus incurring a large cost.

We ran the optimization twice, to check consistency of the solution, on Intel Quad-Core i5-3570K processor running at 3.40GHz. Each optimization took about 12 hours, calling the objective function 3200 times in each case. Both runs found a large number of solutions that did not fall down, as shown in Figure 2.6. As time went on in each optimization, we see that the number of controllers that can avoid falling gets increasing large, and the change in the objective function levels out. Both optimizations find nearly the same solution, with similar objective function values.

Figure 2.7 shows the simulations that are performed in a single call to the objective function, shown for the best point found by each optimization. We see that all trials with level-ground walking asymptotically approach the target walking speed of 0.55m/s, and that there is a steady-state error for both uphill and downhill walking. Finally, we can also see the change in speed cause by both the forward and backward pushes, which are rejected within four steps.

Controller 3 was selected to be the best controller found out of all that were sampled in both of these optimization runs, and it's performance in simulation is shown on the left side of Figure 2.7.

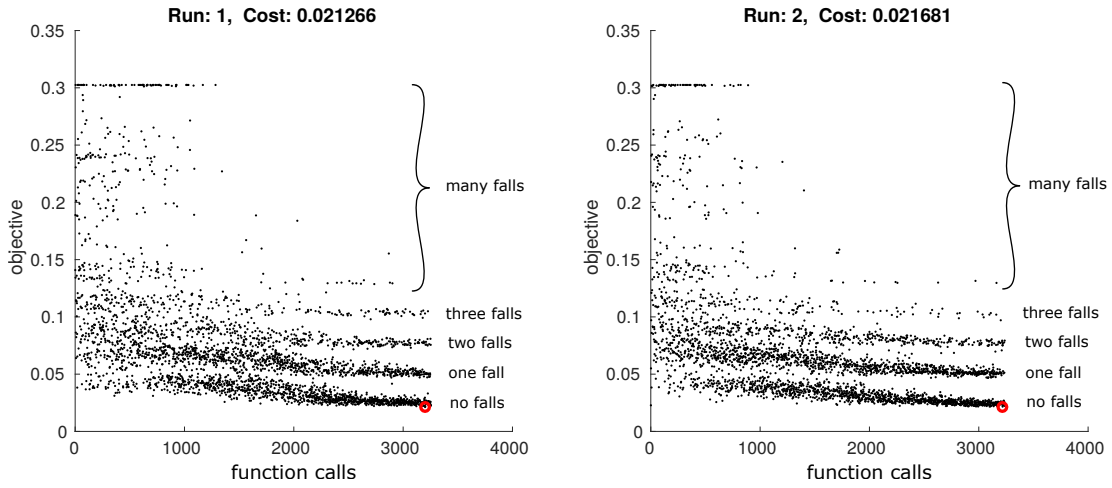

Figure 2.6: **Off-line Optimization Two: Objective Function** This figure shows the value of the objective function while during each run of off-line optimization two. We can see that the general structure of both plots is similar, and that the optimization does in fact find more good controllers as time goes on. The banding structure in both plots is a result of the objective function: when the robot falls down, it gets penalized for all incomplete steps. Since it usually falls during the first few steps of the trial, the result is a jump in the objective function equal to the spacing between bands. The red circle on each plot shows the best controller.

2.11 Results: On-line Controller Optimization Experiments

For on-line optimization, each evaluation of the objective function consists of the robot walking a specified number of steps, logging data, and then comparing its walking speed to a target value on each step. The goal of the optimization is to achieve steady-state walking while successfully rejecting real-world disturbances, such as small irregularities in the floor.

The optimization is performed on the robot, in real time, using particle swarm optimization (PSO), described in Section §2.8.2. We use a warm-start version of the algorithm, where the first controller to be tested is the controller that we obtained using the off-line optimization.

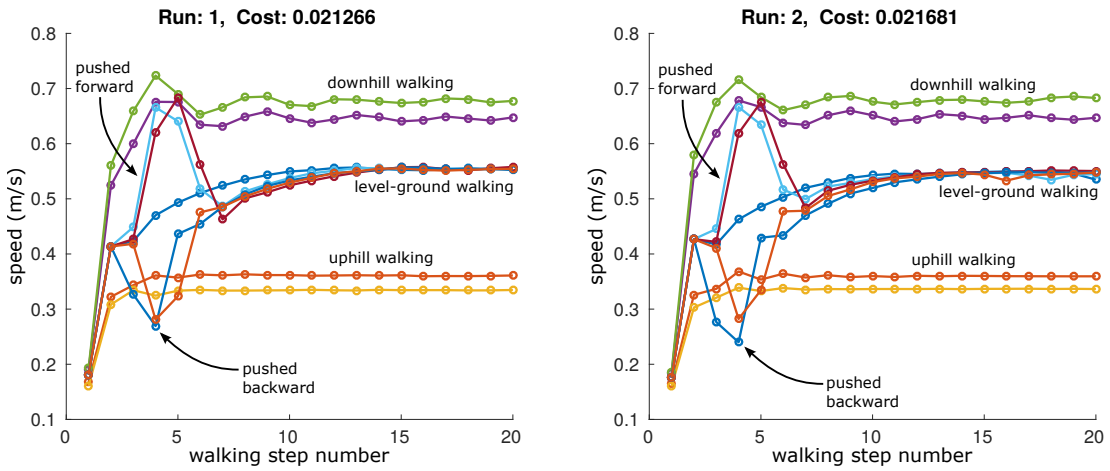

Figure 2.7: **Off-line Optimization Two: Simulations** This plot shows the nine simulations (walking speed vs. step number) that are computed during each evaluation of the objective function in off-line optimization two, shown here for the best controller found during each optimization run. In all cases, the controller is attempting to reach a target speed of 0.55 meters per second. Again, both plots are similar, suggesting that the controllers found by both optimizations are similar. The controller has a steady state error in speed for the two up-hill walking trials and the two down-hill walking trials. The controllers are able to recover from the two forward pushes and the two backward pushes in about four steps.

We performed three different on-line controller optimization experiments on Ranger. Each experiment took 1-2 hours, and all optimization calculations and controller swapping happened automatically on the robot's main processor. The high-level parameters of the optimizations are summarized in Table 2.1, and a general overview of the controller architecture for each optimization is provided in Section §2.9.

Table 2.1: **On-line Optimization:** objective function parameters. All optimizations use the same optimization parameters for PSO: $\omega = 0.7$, $\alpha = 1.8$, and $\beta = 1.5$; see Section §2.8.2 for details.

	Optimization 1	Optimization 2	Optimization 3
Target step speed	$0.6m/s$	$0.6m/s$	$0.55m/s$
Transient steps	2	4	5
Trial steps	8	12	12
Trials per controller	1	2	1
Number of particles	10	15	15
Generations run	12	7	30

2.11.1 On-line Optimization: Decision Variables

In each on-line optimization there were either 5 or 6 decision variables, depending on which version of the control architecture we were using. These correspond to the free parameters in the gait controller, and are briefly summarized below.

- *Scissor Offset* is the constant term in the swing-leg scissor trajectory. Used in all on-line optimizations.
- *Scissor Gain* is the linear term in the swing-leg scissor trajectory. Used in all on-line optimizations.
- *Ankle Push* is a normalized push-off magnitude, with larger values corresponding to larger push-off. Used in all on-line optimizations.
- *Critical Step Length* is the critical distance between the ankle joints of the robot, which triggers the transition into push-off mode. Used in on-line optimization two and three.
- *Double Stance Delay* is the duration of time that the robot continues push-off after the swing foot strikes the ground. Used in on-line optimization one and three.

- *Critical Stance Angle* is a the critical angle of the stance leg which is used to trigger the transition into push-off mode, only used in on-line optimization one.
- *Hip Step Angle* is the fixed target angle for the hip joint during push off, only used in on-line optimization one.
- *Push Integral* is the critical value for the integral of current into the stance ankle, measured since the start of push-off, which is used to trigger the transition from push to glide mode in on-line optimization two.

2.11.2 On-line Optimization One

On-line optimization one was conducted using Controller 0, an experimental control architecture, which used a critical angle of the stance leg to trigger push-off, rather than a critical step length like the later Controllers 1, 2, and 3.

During the early parts of the first optimization run the robot fell frequently, in some cases not even making a single successful step. One common cause of these falls was foot scuffing, where the swing foot collided with the ground in mid-swing due to the Push-Off mode being triggered too early. In some cases this caused an immediate fall, while in others it caused a sequence of progressively shorter steps which eventually led to a fall.

As the optimization progressed, the walking gait clearly became more regular and reliable. By the forth generation most new gait patters resulted in stable walking, although there was still noticeable variation in speed between gaits. By the end of the optimization most gaits were walking near the desired speed, with only a few outliers.

One way to study the progress of the on-line optimization is to look at how each of

the parameters changes throughout the optimization, as shown in Figure 2.8. We can see that all parameters converged to some value by the end of the optimization, and that this value was different (and better) than the initial guess (from the offline computer optimization).

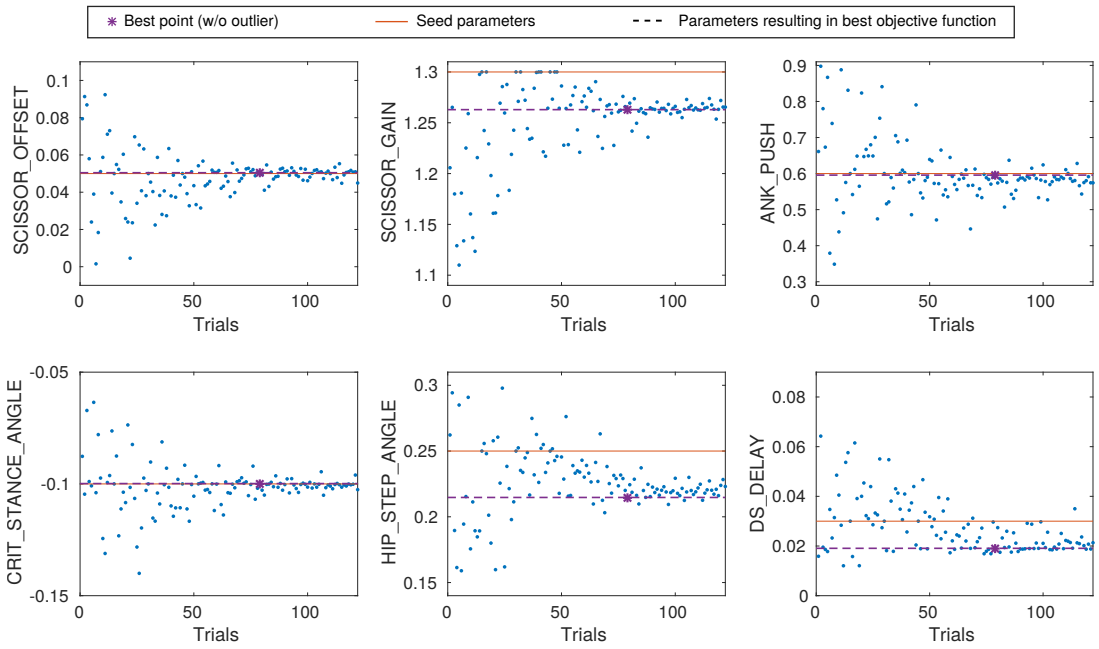

Figure 2.8: **Parameter Evolution, On-line Optimization One**. Each panel shows one of the six optimized parameters. The solid horizontal lines show the guess we seeded the first particle with. The dotted line shows the best-ever parameter values.

2.11.3 On-line Optimization Two

Controller 1, which was used in this optimization, was somewhat more tolerant to large changes in the gait parameters than the controller used in on-line optimization one. As a result, the robot rarely fell down during the optimization process, even at the beginning. As the optimization progressed the gaits did get a bit more reliable, with no fear of the robot falling down.

One problem that we observed was that the controllers tested by the optimization seemed to struggle to walk at the target speed. In on-line optimization one, we occasionally saw controllers that walked much too fast, but this rarely occurred in on-line optimization two, where most controllers seemed to be too slow.

We can look at the parameter convergence plots (Figure 2.9) for this optimization to get a better idea of what is going on. While we saw a clear convergence in on-line optimization one, it seems that the parameters in on-line optimization two did not converge to a clear value.

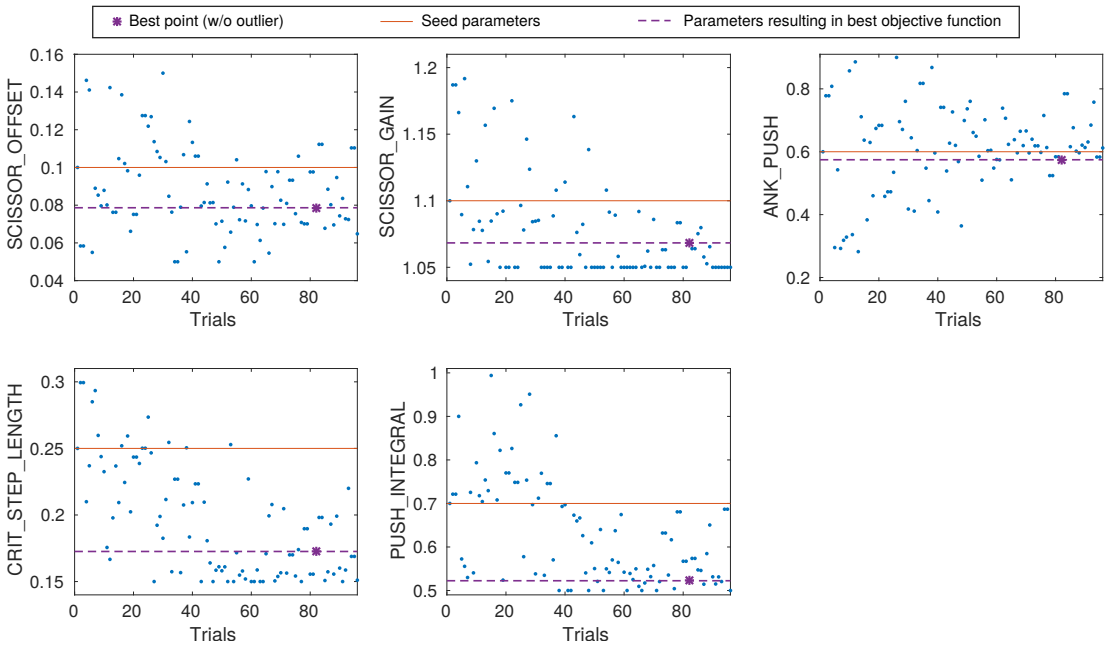

Figure 2.9: **Parameter Evolution, On-line Optimization Two.** Each panel shows one of the five optimized parameters. The solid horizontal lines show the guess we seeded the first particle with. The dotted line shows the best-ever parameter values.

One important difference between the first and second on-line optimization experiments is in the objective function. In on-line optimization two we used longer trials when computing the objective function, and also ran each trial twice. We passed whichever of the two (repeated) trials had the worse score to the optimization. The goal

here was too reduce the noise in the optimization, and also prevent 'false-positive' trials, which can cause major problems in particle swarm optimization.

To better quantify the repeatability of the objective function we looked at the difference between these repeated trials, and then compared this to the best-ever value of the objective function and the mean value of the objective function, shown in Figure 2.10. We can see that the difference between any two repeated trials is, for most trials, about an order of magnitude less than the mean value of the objective function.

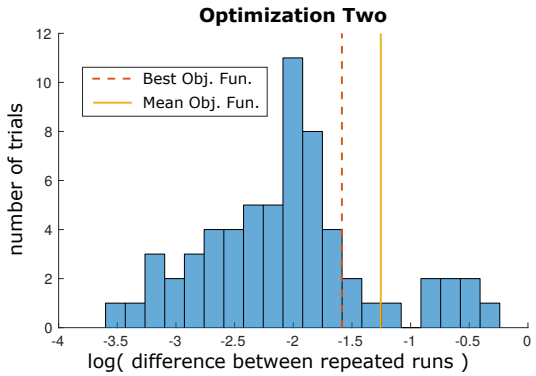

Figure 2.10: **Repeated Trials, On-line Optimization Two.** This is the distribution of the differences between the replicated trials in on-line optimization two. Also marked are the best and average objective function values to provide a scale for the changes. We see that the change between repeated calls to the objective function is smaller than the mean and best values of the objective function, suggesting that the noise problems from on-line optimization one were addressed by the longer trials in on-line optimization two.

2.11.4 On-line Optimization Three

The majority of the controllers that were tested by on-line optimization three were able to walk, although we observed a lot of variety in the walking gaits. Visually, it was very obvious when the robot switched controllers – the gait patterns, walking speeds, and foot impacts were different from trial to trial. The most common failure resulted from the robot twisting about its vertical axis, something that out planar simulator cannot

model. This behavior would start after heel-strike, with Ranger beginning to oscillate torsionally. In the best case, the robot changed its heading (up to $\approx$ 45 degrees) but stabilized. In failure cases, subsequent steps amplified the twisting motions until the robot fell.

In on-line optimization three, we did only one trial per controller but used the same longer trials as on-line optimization two. This seems to have been effective, as the controller parameters seemed to converge (Figure 2.11), and there were no obvious outliers.

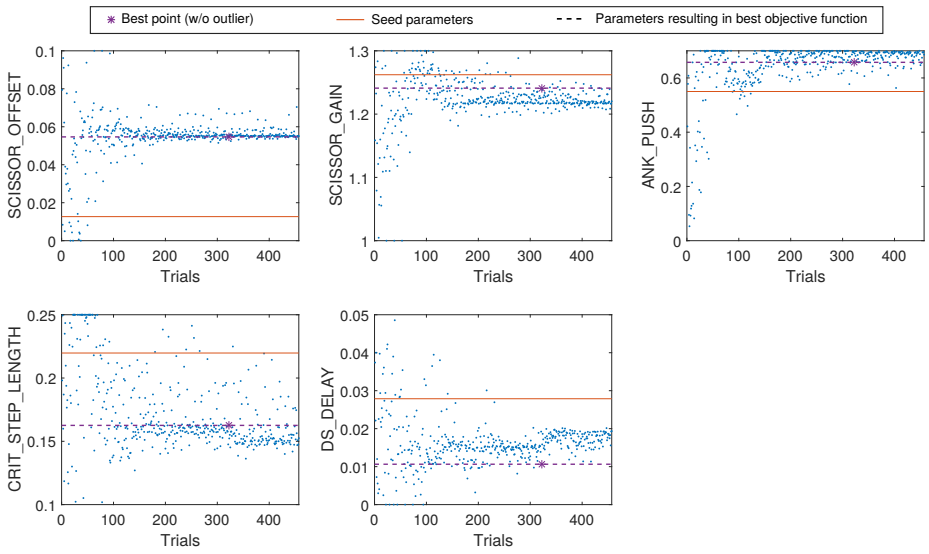

Figure 2.11: **Parameter Evolution for On-line Optimization Three.** Each panel shows one of the five optimized parameters. The solid horizontal lines show the guess we seeded the first particle with. The dotted line shows the best-ever parameter values.

2.11.5 On-line Optimization: Objective Function Comparison

This section will show and discuss how the objective function values changed over the course of the on-line optimization. Figure 2.12 shows the objective function values grouped by particle swarm optimization generation.

In on-line optimization one we can see that the objective functions start with a wide spread, which we expect due to the large number of falls early-on in the experiment. As the experiment progresses, the objective functions values start to collapse towards better values, and are well-clustered near the end of the optimization. Note that many of these trials get a worse score than if they had fallen on step one. This is because every failed step is given a speed of zero. Since the objective function is mean squared error of step speed, steps that are much faster than the target speed receive a very bad score. In on-line optimization one, there were many of these cases in which the robot took very fast, small steps before falling.

In on-line optimization two we can see that the objective function values span a much smaller range. There were fewer falls, and most of these falls were backwards – the steps were too slow, not too fast. Although the objective function values do seem to cluster towards the end of the optimization, they are converging to a much worse value for the objective function than in the first experiment. This indicates that the second optimization either A) failed to converge, B) converged to a bad local minimum, or C) that the new controller is unable to walk at the target speed. Another possible effect is that optimization one trials were fewer steps. Even though the objective is average speed, the average of fewer samples is more noisy and will result in a wider spread of objective function values.

In on-line optimization three, the objective functions remained largely clustered from the beginning. The controller was fairly stable even though we observed a wide variety of gaits. We see quite a bit of improvement in the first 10 generations but progress seems to stagnate afterwards even though the parameters continue to converge as seen in Figure 2.11. This suggests that a wide range of control parameters perform similarly.

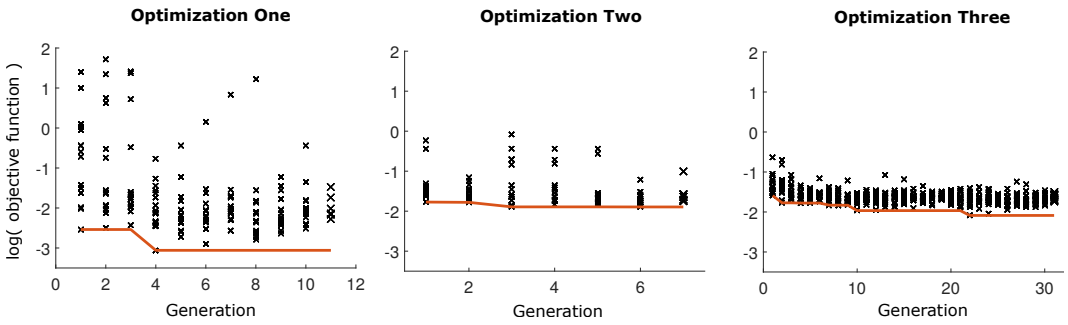

Figure 2.12: **Objective function value**, on-line optimization, grouped by generation. The solid line is the best-so-far value. The control architecture in optimization one was less stable, and so the objective function values are more scattered. Note that each optimization was run for a different number of generations.

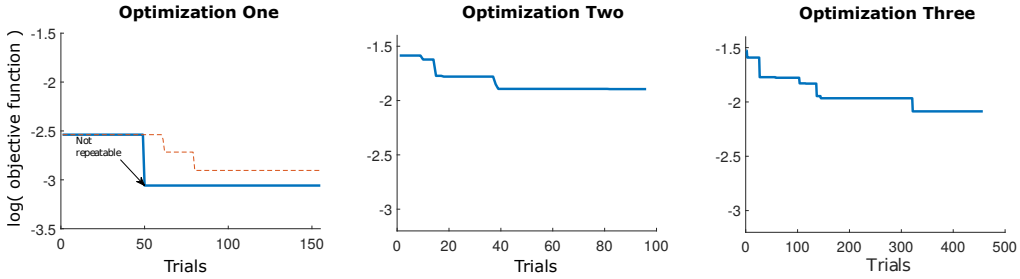

Figure 2.13: Best objective function seen so far, plotted against the trial number. Note that optimization 1 had a false best (unrepeatable). The dotted line shows the results without this outlier.

An alternative way to look at the progress of the optimization is examine just the best-ever objective function value, as shown in Figure 2.13. At a first glance, it looks like the optimization in the first experiment only found one improvement over the entire run, but this is clearly not true, since we observed continual improvement over the course of the optimization. We found that noise in the objective function caused a single point to get an artificially high score, thus skewing the results. With this outlier removed, the convergence data is as expected. As a side note, we ran the objective function trial again (after the experiment) for this one outlier point, and found that the objective function value was (as expected) not reproducible.

2.11.6 Open-Source Data Set

We ran an additional on-line optimization, purely for the purpose of generating a large set of walking data for other researchers to study. We logged the raw data from most of the critical sensors on the robot: all six channels of the IMU; all motor angles, rates, and currents; joint angles and rates; and the contact sensors on all four feet of the robot. This data set has been processed to be easily usable by others, and is available at: `http://ruina.tam.cornell.edu/research/topics/` ... (continued below)

... `locomotion_and_robotics/ranger/WalkingData/index.html`

2.12 Results: Robustness Experiments

The goal of this research is to automatically develop controllers that are better at rejecting disturbances. In the previous sections, we explained our process of synthesizing controllers. In this section, we compare the performance of several different controllers under disturbances. In each case, we apply a modeling error to the robot (*e.g.* add mass to a leg), and then have it walk over some ground profile that introduces small perturbations.

We performed two different robustness tests. Robustness Test One compared the Marathon Controller, the ICRA controller, and Controller 2, each walking a long distance. Robustness Test Two compared the Marathon Controller, Controller 3, and Controller 3*, each walking a many trials over a short distance. These controllers are described in detail in Section §2.9. In all cases, the disturbance was a known modeling error (intentionally introduced) combined with small random disturbances from the

ground.

2.12.1 Robustness Test One

This first robustness test extends the results presented in [57] to include an additional controller (Controller 2), which was produced by off-line optimization one. We tested each controller under four different conditions. In each case, the robot walked at least 180 meters and we counted the number of falls.

Tested conditions:

- *Baseline*: In the baseline experiment, we had each controller walk a distance on the test floor without intentional disturbances. The floor is made of stone tiles, and there are small (2mm) variations in ground height between each tile. These variations cause small perturbations to the walking gait. All other trials were conducted on the same floor, with additional disturbances.
- *Front Weights*: We added a 0.35 kg mass to the robot, a distance of 0.12 meters in front of the hip joint, on the outer legs as shown in Figure 2.14. This caused a continual disturbance torque while walking.
- *Rear Weights*: For this trial, we applied the same 0.35 kg mass to the robot, but this time a distance of 0.12 meters behind the hip joint.
- *Weak Spring*: Ranger has a spring connecting its inside and outside legs, as shown in Figure 2.14. To give an idea of relative stiffness, the hip motor can sustain a maximum hip joint angle of about 30° when pushing against the spring. We added

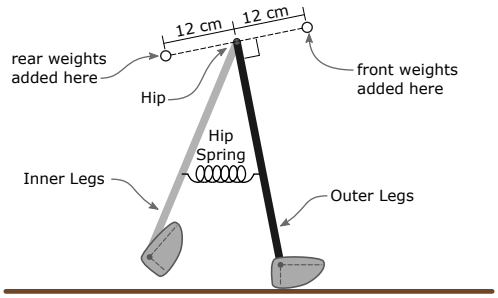

Figure 2.14: **Disturbance Diagram** In the robustness trials we applied three intentional disturbances to the robot. In the *Weak Spring* experiment, we added a loop of string in series with the hip spring, effectively removing it. In the *Rear Weights* experiment we added a small weight, as shown above to the rear of the robot. In the *Front Weights* experiment we added a small weight, as shown above to the front of the robot.

a loop of string in series with the spring, thus effectively removing it, except for very large hip angles.

In each experiment, we started by having the robot walk a minimum distance of about 180 meters. In the baseline experiment for the ICRA controller, we observed a single fall near the first 180 meters. We decided to extend the trial to about 750 meters to collect additional data to get a more reliable estimate of the fall rate. We also decided to have the new controller walk 350 meters in each trial, since no falls were observed in the first 180 meters.

The results of the experiment are shown in Figure 2.15. The primary goal of the new controller is to avoid falls while walking in environments with disturbances, as shown in Figures 2.15.

2.12.2 Robustness Test Two

We designed Robustness Test Two to have a more flexible. We wanted to introduce large enough range of disturbances to make every controller fall sometimes. We strapped a

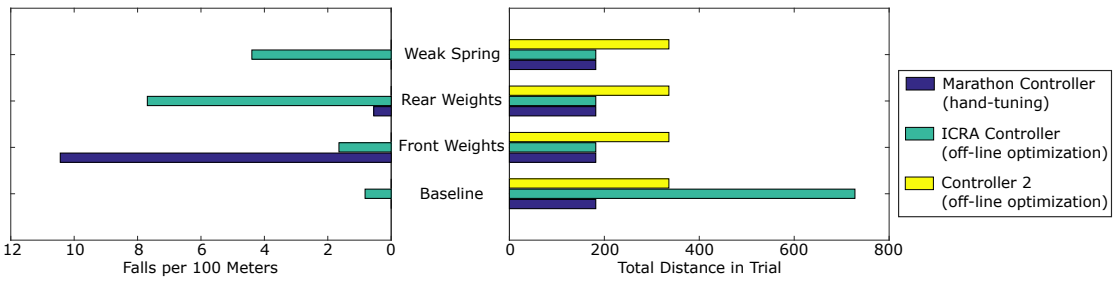

Figure 2.15: **Robustness Test One** Here we compare the Marathon Controller, the ICRA controller, and Controller 2 in their ability to avoid falls when subject to a variety of disturbances. Controller 2 is able to successfully avoid falling in all trials (left plot), a notable improvement over the two previous controllers. The right plot shows the number of steps in each trial. We extended some of the trials, particularly when falls were infrequent, to get more data on those controllers.

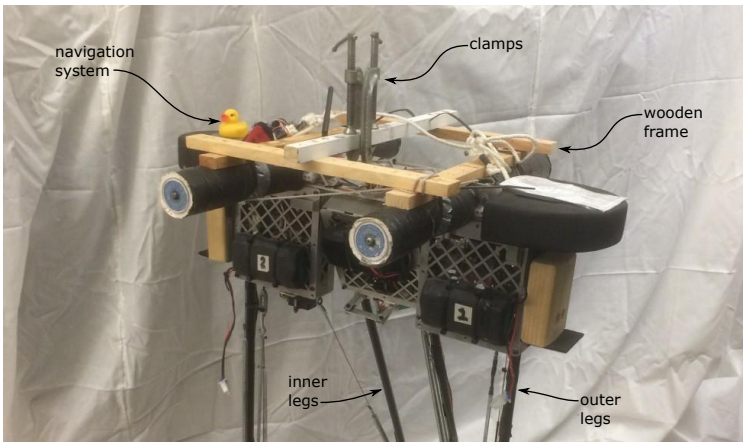

Figure 2.16: **Robustness Test Two: Disturbances** For robustness test two, we added a small wooden frame to Ranger. We attached a pair of clamps to this frame, allowing us to apply a variety of disturbances to the robot. They are shown here in a position 6 cm forward of the center of the robot, creating a net external torque due to gravity while walking.

light wooden frame on top of Ranger and attached steel clamps at various distances from the center of mass of the robot, as shown in Figure 2.16. This let us add adjustable disturbances to the robot's mass, rotational inertia, and external gravity torque.

The purpose of Robustness Test Two was to compare the controllers produced by Off-line Optimization Two (Controller 3) and and On-Line Optimization Three (Controller 3*) to the original (largely hand-tuned) Marathon controller for Ranger. The test

consisted of a set of baseline trials (no disturbance) followed by five different disturbances, summarized in Table 2.12.2. These disturbances were created by adding mass (in the form of clamps) to Rangers Outer legs (above the hip joint).

Table 2.2: **Robust Test Two: Disturbances** We added intentional disturbances to the robot, to study their effect on the performance of the controllers. The disturbances were created by adding a light wooden frame and two 'C'-clamps to the outer leg of the robot. The 'Mass' column describes how much mass was added to the robot by the frame and clamps. The 'Inertia' column gives the change in the moment of inertia, about the hip joint, of the outer legs. The 'Torque' column describes the net torque due to gravity acting on the clamps, about the hip joint, when the outer legs are vertical. Note that although the frame has a small mass and inertia, it holds the fall protectors rigid, which changes how the robot vibrates when it hits the ground.

Name	Added Mass (kg)	Added Inertia (kg-m^2)	Added Torque (N m)
Baseline	0	0	0
Frame Only	$4.5 \cdot 10^{-4}$	$3.9 \cdot 10^{-5}$	0
Mass Center	$6.6 \cdot 10^{-1}$	$2.2 \cdot 10^{-2}$	0
Mass Wide	$6.6 \cdot 10^{-1}$	$4.8 \cdot 10^{-2}$	0
Mass Front	$6.6 \cdot 10^{-1}$	$2.4 \cdot 10^{-2}$	-0.39
Mass Back	$6.6 \cdot 10^{-1}$	$2.4 \cdot 10^{-2}$	$+0.39$

For each controller and disturbance, we had the robot walk twenty trials, each eight meters in length. For each trial we assigned the robot a score between 0 and 1, where 0 corresponded to the robot falling down and 1 corresponded to perfect walking. This gave us a subjective score for each set of trials (the average of the scores), as well as an objective score (number of falls), both of which are reported in Figure 2.17. We find that all controllers walk reasonably well in the baseline trial, as well as the trial with only the frame attached to the robot. The controllers also seem to have more trouble when the masses are not placed symmetrically, thus creating a net gravity torque on the robot. There is no single best controller across all trials, but the original Marathon controller seems to out-perform Controller 3 and Controller 3* in many trials.

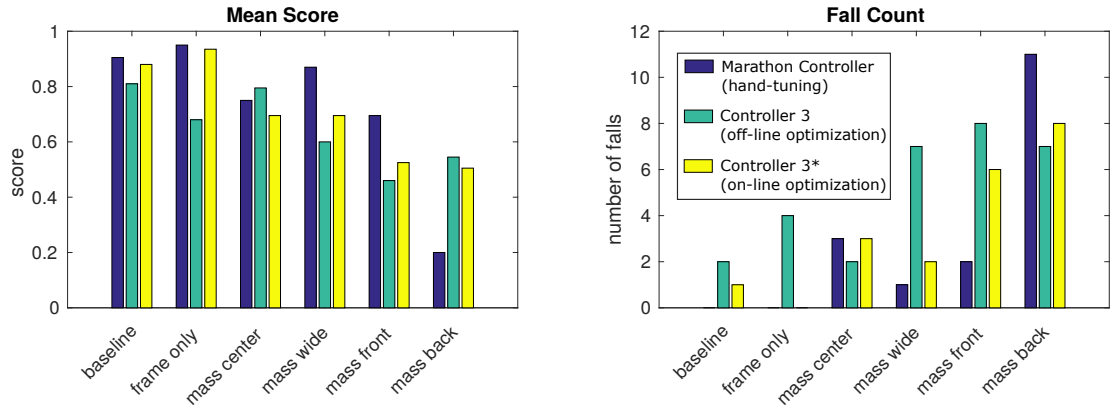

Figure 2.17: **Robust Test Two** Here we compare the previous 'marathon' controller [10], with our new controller, first after 'off-line' optimization (Controller 3) and then after 'on-line' optimization (Controller 3*). We compared these controllers on baseline walking as well as five disturbances, described in Table 2.12.2. Each controller walked twenty trials, each eight meters long. Each trial was given a score on the range 0-1, where 0 = robot fell down, 0.3-0.6 = robot had trouble walking, and 0.8-1.0 = robot walked well. The left chart 'mean score' shows the average across this subjective score for all twenty trials, while the right chart 'fall count' gives the objective number of trials during which the robot fell down (out of the total of twenty trials).

2.12.3 Gait Asymmetry

In all our new controllers for the robot, we have observed an asymmetry in the gait: the outer feet swing through the step faster, causing a larger step. The asymmetry varies in magnitude between different controllers, but seems to always be present. We have done many small experiments on the robot to isolate the source of the asymmetry. We have found many small effects, but no primary cause. The key idea behind testing is this: we have found the primary cause if we can reliably reproduce the opposite gait asymmetry. We have failed to do this. Below is a list of the basic experiments that we've done, and their results.

- **Controller bugs.** We had two people read through the code for any obvious bugs, such as asymmetric controller gains, set-points, or transitions. After that,

we tested each component of the controller in isolation, logging data and looking for asymmetries. We found that all components behaved as expected under controlled conditions.

- **State estimation and sensor error.** We looked for bias terms in the state estimator, with a particular focus on the absolute orientation of the robot, which uses the IMU. We tested and re-calibrated the IMU. No effect.
- **Mechanical asymmetry.** The physical robot is asymmetric (most notably, heavier outside pair of legs). This certainly causes some degree of asymmetry in gait since the controller treats both pairs of legs exactly the same. We did a series of dynamic leg and ankle trajectory tracking tests. The differences between the tracking errors for inner and outer legs were on the order of sensor error - not the primary cause of the problem.
- **Foot variance.** The rubber 'shoes' on the robot's feet are all slightly different. We experimented with changing the orientation constants to compensate for differences. This caused substantial changes to the gait, but not in a way that directly removed the asymmetry.
- **Foot push-off differences.** The transmission is different between the inner and outer legs. In particular, the compliance and damping are not quite identical. To test whether this could be a factor, we introduced an asymmetry in the ankle joints during push-off by adding a larger feed-forward term to one or the other pair of ankle joints. Again, we could change the gait, but not in a way that canceled out or inverted the asymmetry.
- **Asymmetry in the controller + model system.** We were able to replicate some of the asymmetry in simulation. For some control parameters (not necessarily the same as those on the robot), we found that the perfectly symmetric controller and model stabilized to a period-two gait in simulation.

- **Possible period-two gait.** We attempted to apply external torques to the real robot while walking to force it into the opposite period-two gait (inner legs swing too fast). We had no success – the robot always settled back into the original asymmetric gait.

We conclude there are probably many factors that combine to produce the gait asymmetry on Ranger. Our controller may have a stable period-two gait, but this cannot explain why the outer legs are always faster. It should be possible to induce the complementary period-two gait. It is possible that some small effects of the hardware asymmetry combine with the stable period-two gait of the controller to produce the observed behavior. A final possibility is that there could be an error in the control code which somehow got duplicated in both simulator and hardware.

2.13 Discussion

While working to automatically design a controller for Ranger, we gained many insights about the process of making a robot walk. We think these "morals" are our most important contribution since many of them extend beyond the specific robot we use. In the following sections, we'll discuss our process, the stumbling blocks, and some of our tricks for getting around them.

2.13.1 Big Picture Concepts

Although we present a method here for 'automatic controller design' there are still a large number of steps that are chosen manually, through human intuition and experimentation. For now, there is a fundamental reason for this: there is no (practical) way

to represent a general control architecture. Thus, the designers of any control system make choices about the architecture to reduce the scale of the problem. For many problems, this choice of controller is simply a linear gain matrix, as seen in all basic and intermediate control theory courses.

Simple linear controllers do not work for stabilizing walking robots, so robot designers have come up with a variety of control approaches (see Section §2.3. In each case, these controllers reduce the control problem to a manageable size by creating a specific controller architecture. In this paper, we constructed a specialized controller architecture, the parameters of which we found by optimization.

Although this paper is largely focused on the computer optimization aspect of this work, the truly difficult part is designing what comes before the computer optimization: modeling the robot, understanding where that model fails, choosing a 'good' control architecture, and selecting a reasonable choice of objective function. The state of the art seems to be humans spending long hours either: hand-tuning a controller, or refining an automatic process to do it for them. Both are highly iterative. In this section we seek to explore these 'meta' optimization problems, and show how they relate to robot control.

2.13.2 Thoughts on Modeling and Simulation

Simulation is an important tool for designing controllers for walking robots. In particular, it allows the control designer to understand how a controller might behave on the real robot, without needing to perform a real-world experiment. In this paper, we use simulation to compute a reasonable set of controller parameters, which can be transferred directly to the physical robot to produce a walking gait.

It is simple to make an arbitrary simulator, but quite difficult to make a simulator that is an accurate representation of real robot. There are many reasons for this, some of which are discussed here. Let's start by making a distinction between a *model* and a *simulation*. A model is a idealized mathematical description of the robot, while a simulation uses numerical techniques to predict the motion of that model. Thus we arrive at our first source of error: the numerical methods used to simulate some model, some of which are more or less accurate than others.

Given modern computers, it is generally easy to make a simulation that is sufficiently accurate at predicting the behavior of a model. In the case of most simulations, the bigger problem is arriving at a good model. The discrepancies between the real robot and a simulation (of a model) of the robot are typically caused by one of the following 'unknowns':

- *Stochastic Unknowns* are things like sensor noise and other small random perturbations, which are well-characterized in a probabilistic sense.
- *Known Unknowns* are features of the robot that the designers intentionally leave out of the model to keep the simulation manageable. A common example is that the links in robots are not actually rigid-bodies, but are modeled as such.
- *Unknown Unknowns* are features of the robot that the designer is unaware of, but that have a significant effect on the behavior.

Each of these unknowns must be addressed by the robot designer. *Stochastic Unknowns* are difficult to deal with in optimization, because it is expensive to compute enough *Monte-Carlo* simulations to get an accurate representation of the noise mode. One solution to stochastic unknowns is to have a good estimator running on the robot, which minimizes the effect of sensor noise and other small perturbations.

The general strategy for dealing with *Known Unknowns* is to identify which features have an effect, and design some simple way of representing that effect in the simulation. For example, Ranger has a complicated transmission that connects the ankle motors to the ankle joints. The transmission generally behaves like a rigid link, but under impulsive loading conditions it has non-zero compliance and damping that are difficult to model and simulate. The solution here was to modify the controller to prevent it from sending impulsive loading requests to the motors.

The best way to deal with *Unknown Unknowns* is to identify them using careful experiments, and then they become known unknowns, as was the case in Ranger's ankle joint transmission.

A good controller should be able to deal with all three types of unknowns automatically. Our general method was to construct a representative set of disturbances that captures the nature of the known and stochastic unknowns, and hope that it is reasonable at representing the unknown unknowns. For controller design on Ranger, we used walking on sloped ground and various pushes as our representative set of disturbances.

2.13.3 Thoughts on Controller Design

Since optimization on a simulator is much faster and much less damaging to the robot, we want to maximize the portion of the controller design which takes place on the simulator. In the limit, if we had a perfect simulator, we wouldn't need to do any refinement on the physical robot. No simulator is perfect, particularly for robots that are dynamic and involve intermittent contact. One candidate solution is to start with the best simulator, and then design a controller. Whenever we notice a feature that does not match reality, we update the simulation. For example, if the optimization finds a walking gait

that uses the wrong part of the foot, we add a check to say that the robot 'falls' (fails to complete the step) if the foot does not contact the ground on the sole.

Similarly, it is very important to understand the robot's failure modes. If all the failure modes are not captured by the simulator, then a controller designed on this simulator will probably not be robust against these sorts of failures. Some of the more common failures we observed on Ranger:

1. The robot doesn't have enough energy at mid-stance and falls backwards.
2. The robot scuffs its feet and takes quick shuffling steps until it falls.
3. The robot's feet hit an uneven surface exciting a twisting motion about the vertical axis until

Failures 1) and 2) are mostly captured by the simulator. Failure 3) is completely not modeled. The twisting happens when a pair of feet hit at different times (e.g. due to an uneven surface). Ranger's legs begin to flex, and the whole robot acts like a torsional spring. Our simulator is in 2D and treats all links as rigid bodies. Thus, our simulator cannot predict this failure.

Given free reign, our optimization will choose giant push-off parameters and step lengths. In 2D, this makes sense: the robot will be most robust if we put a lot of energy into the beginning of the step and dissipate it in the heel-strike collision at the end. We observe, however, that large push-off and collisions cause the twisting instability. Since this is a completely un-modeled effect, we choose to artificially limit push-off to compensate for dynamics our simulator does not understand.

2.13.4 Thoughts on controller architecture

Ideally, we would like an optimal control policy, a mapping of every state to an ideal action. This policy is infinite-dimensional and too computationally difficult to find. Instead, we assume a specific control architecture and optimize over a small set of parameters. This process is much like fitting a function to some arbitrary curve. A linear curve fit would be simple, but likely have large errors, while a high-order polynomial might be more accurate but difficult to compute. The only difference here is that we are not fitting some known curve, but instead trying to optimize the coefficients of the fitting curve to achieve some behavior in the controller. The quality of our controller is restricted by our choice of fitting function and the coefficients. This is where human intuition comes in, we select a (hopefully) good control architecture so that the optimization can select the best set of parameters for that function.

A weakness of our control design process is that is still relies on human intuition to select the control architecture, which can be limiting. For example, Figure 2.15, shows that Controller 2 is significantly more robust than the ICRA controller. They key difference between these controllers were two small changes in the control architecture. In both cases, optimization was used to compute the best choice of control parameters ($\approx$ 15 numbers), but not the sequencing of events (e.g. push-off, glide, etc.), transition triggers, structure of the low-level controllers, and numerous other details.

We arrived at final control architecture through human intuition and trial and error. In the process, we had many intermediate architectures: some couldn't walk (no control parameters could stabilize it), some limped along (had a pathological behavior not affected by the control parameters), and others were very sensitive to parameters (good walking, but very difficult to find the combination of control parameters that worked). In our final control architecture, we found that the robot could walk with a wide range

of control parameters. Also, by changing these parameters, we could change the way the robot walked (speed, cadence, foot impacts, etc.). These two characteristics seemed to indicate that we had arrived at a good architecture. Still, we do not think our control architecture can represent the best walking Ranger is capable of. In future work on Ranger, we think that iterating on the control architecture would have a much greater impact on robustness than improving our optimization process.

2.13.5 Thoughts on Off-line Optimization

When designing controllers using off-line optimization of a simulation, there are two places where the 'reality gap' becomes a problem. The first is that a human designer must create an precise objective function that describes some vague notion that they might have about what "*good walking*" means. The second source of problems is that the simulated model of the robot does not behave exactly like the real robot. A general feature of optimization is that it tends to exploit these gaps between the simulated world and the real world.

The rough solution that we have to this problem is a human-in-the-loop iteration between the optimization, testing in simulation, and testing in reality. The basic idea is that we run an optimization and then test the resulting controller in simulation. If we see some ridiculous behavior (like the robot going up-hill hopping on one foot) than we modify our problem statement (apply appropriate joint torque limits) to correct the behavior. Once the result looks good in simulation, we move it to the real robot and then see how it looks. If we identify problems, then we look at how we can change the model or objective function to be more realistic.

One example of this iterative process was an early objective function that only tested

disturbances from steady-state walking. The resulting controller performed well in simulation, but immediately fell down on the real robot. We realized that the optimization had only learned about steady-state walking, and he no ability to cope with disturbances at the onset of walking. We fixed this by making all trials start walking from the static launch configuration.

Another example was a controller that performed well in simulation but would always stumble and fall on the real robot. It turned out that the ground contact model in the simulator was too perfect, and the robot learned that it was ok to skim its foot just over the ground while walking. We fixed this by adding simulations over non-flat ground, in this case up-hill and down-hill slopes. This immediately ruled out foot-skimming behavior and the controllers transfered more reliably to the real robot.

The resulting controller is really a collaboration between human intuition and optimization. Rather than direct hand-tuning, we rely on experimental iteration to arrive at an objective function and model that is the best match for our robot and our goals. The resulting controller designed by this optimization is able to be transferred directly to the robot, and the robot walks.

2.13.6 Thoughts on on-line optimization

The on-line optimization was intended to make small improvements to the controller from the off-line optimization, rather than find a new solution entirely. As such, we selected optimization parameters for the Particle Swarm Optimization (PSO) so that it behaved more like a hill-climbing algorithm rather than a global-search algorithm. The general idea here was that the simulation should be a fairly accurate representation of the real robot, so the on-line optimization should just be compensating for relatively

small modeling errors. In all cases, the optimization found controllers better than the starting controller, although the improvements were not drastic.

One interesting challenge in the on-line optimization was finding a suitable number of steps in each trial. Ultimately the on-line optimization is limited by time, so more steps in each trial resulted in a few number of generations. On the other hand, if there were too few steps in the trial, then small noise terms in the objective function became dominant.

The noise in the objective function can be particularly problematic if it causes a controller to get an artificially good score, since PSO works by drawing particles towards the best-ever controllers. This happened in on-line optimization one, which had an outlier near the start which drew the optimization towards a controller that was actually not good, but had been scored well due to noise. Overall, the optimization still improved, likely because it was being pulled towards other best points besides just that one outlier. Also, the control architecture itself was less-stable so there was more room for improvement (steeper gradients).

In on-line optimization two, we increased the number of steps per trial, and did each trial twice. This time, we did not observe outliers. However, we did not make it through nearly as many generations and convergence was poor. By comparing the objective functions of repeated trials, we observed that the noise in the objective function was roughly an order of magnitude smaller than the values themselves.

In on-line optimization three, we kept the increased steps per trial but stopped doing each trial twice, and we didn't seem to have substantial outlier issues.

In general, with on-line optimization, objective function evaluations are very costly (time and robot wear). On the other hand, real-world factors make objective function

noise large, incentivizing doing long trials. It took us three trials to find an effective balance. In retrospect, we could (and should) have experimented with trial length before running running the full optimization, to find the minimum trial length which was acceptably repeatable.

Our cost function was perhaps too simplistic: the mean-squared error of step speed from a target value. For example, during many of the walking trials we saw controllers that looked to be excellent, nice smooth walking, with no sign of falling down. In post-processing we discovered that these controllers were not scored as highly as some more impulsive controllers, which did not walk as well in a qualitative sense: they were more impulsive, and looked less reliable. This indicates that our mathematical objective function was not a perfect match for the objective function that we had in our heads. The best explanation was that the "smooth" walking controllers had an asymmetric gait (fast step, slow step, fast step,...) , which was overly-penalized by the objective function, while the more highly-scored impulsive controllers had better speed regulation, but were less reliable on repeated trials.

2.14 Future Work

In this paper, we detail our controller design process for Ranger, using automatic methods (optimization) where possible. We can take control parameters directly from off-line optimization and make the robot walk and we can do an on-line optimization to improve these parameters a little bit. Below, we outline some of the future improvements we would like to make to the control architecture and both the on-line and off-line optimizations.

Control architecture:

- The current control architecture uses a single (global) finite state machine, which synchronizes the motions of the swing ankle, stance ankle, and hip joints. It seems that a architecture with concurrent finite state machines, like that used in the Marathon Controller [9], would allow for more graceful heel-strike collisions.
- The balance controller only updates the trajectories of the robot once per step. In the future we would like to make these updates continuous, allowing the robot to more quickly react to external disturbances such as a sudden push.
- The look-up table for the balance controller only uses a single input: the mid-stance speed of the robot. Ideally, the controller would use at least the state of the stance and swing legs, rather than this simple one-dimensional projection.

Off-line optimization:

- Ranger was initially built to be a low-energy walker. In this paper we focus on walking robustness. We could likely make energy part of the cost function without sacrificing too much in the way of robustness.
- We would like to design the controller to have a wide range of target walking speeds. We could optimize for several different speeds using our current methods and interpolate parameters to achieve intermediate target speeds.
- Human intuition (e.g. "that robot looks like it's going to fall down") is still a huge part of the optimization design process. We could embrace this and have human intuition automatically be a part of the process. For example, the optimization could show us several possible local optima it is exploring and let us eliminate ones that are problematic.

On-line optimization

- As discussed, particle swarm optimization is sensitive to good-scoring outliers. In the future we'd like to protect against this by doing validation trials for new best controllers. Another possibility would be to use a different optimization method that is not so sensitive to these kind of outliers.
- We would like to experiment with different kinds of cost functions. The simple step-speed objective fails for our purposes when a very shaky controller happens to walk at near the target speed. One solution is to have the user input a subjective rating after each trial. Another possibility would be to use acceleration data and try to minimize twisting and large banging/ringing in the system.
- Many features of 'good walking' are hard to measure automatically. It would be interesting to experiment with human-in-the loop optimization, where a human can provide some component of the score for the objective function.

2.15 Derivation of Equations of Motion

The equations of motion for each different contact configuration were derived using the Matlab symbolic toolbox. The full simulation then combines these equations using a contact solver. The simulator, along with derivation of the equations of motion, is available at:
`https://github.com/MatthewPeterKelly/RangerSimulation`

2.15.1 Newton-Euler Equations

The dynamics equations for Ranger start with writing out the floating-base dynamics. We model Ranger using four rigid bodies, connected by three joints, as shown in Figure

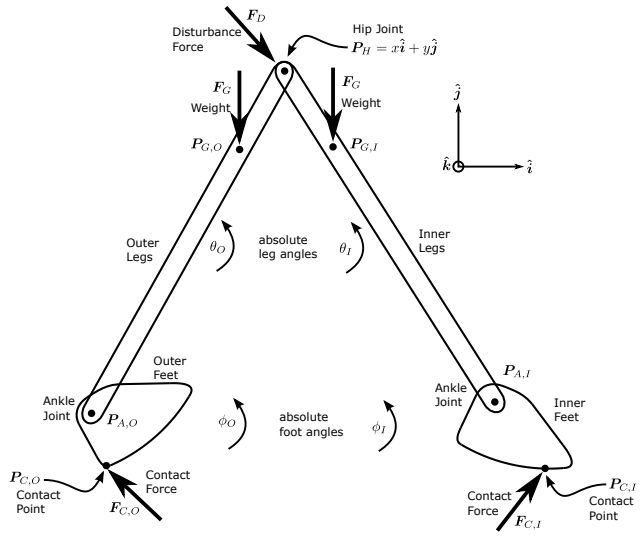

Figure 2.18: **Free Body Diagram for Ranger** The dynamics are derived using a floating base model of Ranger. There are external forces due to gravity, the contacts with the ground, and also a disturbance force applied at the hip.

2.18. There is a torque motor at each joint, and a known disturbance force can be applied to the hip joint. The mass of the feet is included in the leg mass m. The legs have a rotational inertia of I_L about their center of mass, and the feet have a rotational inertia of I_F about the ankle joints. We will assume that the contact solver provides the location of the contact point $\boldsymbol{P}_C$ on each foot, in the frame of the foot.

This model has six degrees of freedom: two for the position of the hip joint $\boldsymbol{P}_H = x\hat{\boldsymbol{i}} + y\hat{\boldsymbol{j}}$, and then one for the absolute orientation of each of the four rigid bodies: outer leg angle θ_O, inner leg angle θ_I, outer foot angle ϕ_O, and the inner foot angle ϕ_I. We will use the Newton–Euler equations to construct a system of six equations. The position vectors are written using $\boldsymbol{P}$ and all point from the origin to the points shown in Figure 2.18. The velocity and acceleration of these points, in the inertial frame, are written $\dot{\boldsymbol{P}}$ and $\ddot{\boldsymbol{P}}$ respectively.

First, we will compute the linear momentum balance for the entire system. This will

give us two equations, obtained by dotting the result with the $\hat{\boldsymbol{i}}$ and $\hat{\boldsymbol{j}}$ unit vectors.

$$\boldsymbol{F}_G \;+\; \boldsymbol{F}_G \;+\; \boldsymbol{F}_D \;+\; \boldsymbol{F}_{C,O} \;+\; \boldsymbol{F}_{C,I} \quad = \quad m\ddot{\boldsymbol{P}}_{G,O} \;+\; m\ddot{\boldsymbol{P}}_{G,I} \tag{2.9}$$

Next, we use angular momentum balance of the outer leg and foot about the hip joint, where u_H is the hip motor torque acting on the inner legs. The final scalar equation is obtained by dotting the following equation with the $\hat{\boldsymbol{k}}$ unit vector, which we will do for all following equations as well.

$$\begin{aligned}(\boldsymbol{P}_{G,O} - \boldsymbol{P}_H) \times \boldsymbol{F}_G \;+\; (\boldsymbol{P}_{C,O} - \boldsymbol{P}_H) \times \boldsymbol{F}_{C,O} \;+\; (-u_H\hat{\boldsymbol{k}}) \\ = \quad (\boldsymbol{P}_{G,O} - \boldsymbol{P}_H) \times (m\ddot{\boldsymbol{P}}_{G,O}) \;+\; (I_L\ddot{\theta}_O\hat{\boldsymbol{k}}) \;+\; (I_F\ddot{\phi}_O\hat{\boldsymbol{k}})\end{aligned} \tag{2.10}$$

Then angular momentum balance for the outer foot about the outer ankle joint, where u_O is the torque acting on the leg from the outer foot.

$$(\boldsymbol{P}_{C,O} - \boldsymbol{P}_{A,O}) \times \boldsymbol{F}_{C,O} \;+\; (-u_O\hat{\boldsymbol{k}}) \quad = \quad (I_F\ddot{\phi}_O\hat{\boldsymbol{k}}) \tag{2.11}$$

Next, we use angular momentum balance of the inner leg and foot about the hip joint.

$$\begin{aligned}(\boldsymbol{P}_{G,I} - \boldsymbol{P}_H) \times \boldsymbol{F}_G \;+\; (\boldsymbol{P}_{C,I} - \boldsymbol{P}_H) \times \boldsymbol{F}_{C,I} \;+\; (u_H\hat{\boldsymbol{k}}) \\ = \quad (\boldsymbol{P}_{G,I} - \boldsymbol{P}_H) \times (m\ddot{\boldsymbol{P}}_{G,I}) \;+\; (I_L\ddot{\theta}_I\hat{\boldsymbol{k}}) \;+\; (I_F\ddot{\phi}_I\hat{\boldsymbol{k}})\end{aligned} \tag{2.12}$$

Then angular momentum balance for the inner foot about the inner ankle joint, where u_I is the torque acting on the leg from the inner foot.

$$(\boldsymbol{P}_{C,I} - \boldsymbol{P}_{A,I}) \times \boldsymbol{F}_{C,I} \;+\; (-u_I\hat{\boldsymbol{k}}) \quad = \quad (I_F\ddot{\phi}_I\hat{\boldsymbol{k}}) \tag{2.13}$$

Now that we have the dynamics equations for Ranger (2.9-2.13), we are almost ready to solve them. We have six equations for six accelerations ($\ddot{x}$, $\ddot{y}$, $\ddot{\theta_O}$, $\ddot{\theta}_I$, $\ddot{\phi_O}$, $\ddot{\phi}_I$), but still need another four equations in order to solve for the contact forces.

The contact solver will determine which of Ranger's feet are on the ground before calling the dynamics function. If the outer feet are on the ground, then the contact solver

will prescribe the desired acceleration for the contact point: $\ddot{\boldsymbol{P}}_{C,O}$ is given, which allows the dynamics to compute the contact forces $\boldsymbol{F}_{C,O}$. If the outer feet are in the air, then the contact forces are zero. The same holds true of the inner feet. If they are on the ground, then the contact solver prescribes $\ddot{\boldsymbol{P}}_{C,I}$, making it possible to solve for the contact forces $\boldsymbol{F}_{C,I}$. Otherwise, the feet are in the air and the contact forces are zero.

2.16 Simulation

This simulator for Ranger is a time-stepping simulation. This means that the simulation works by marching forward using small constant-size time steps. On each time step, a contact solver runs to determine what constraints need to be applied to satisfy the hybrid dynamics of the system. Then a constraint solves runs to compute the contact forces (or in this case, accelerations) that are required to satisfy these constraints. Finally, the state is propagated using the a symplectic euler step. The source code for this simulator is available at:
`https://github.com/MatthewPeterKelly/RangerSimulation`

In the rest of this section, we will carefully go through a single time step in the simulation, showing what happens in the simulator.

2.16.1 Contact Solver

We assume that there are four possible contact configurations.

- *double stance:* both feet are in contact with the ground
- *flight:* both feet are not contact with the ground

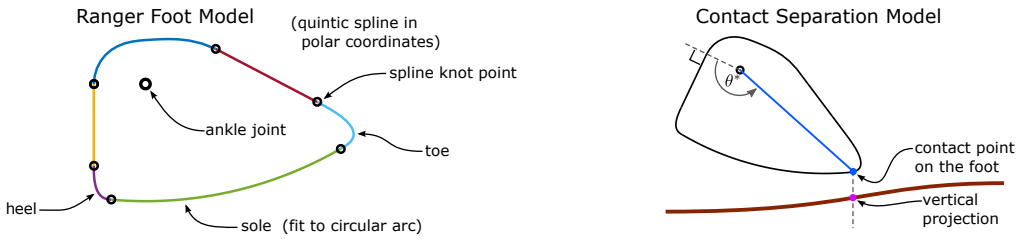

Figure 2.19: **Foot Contact Model** Ranger's foot is modeled using a six-part quintic spline in polar coordinates about the ankle joint. The contact point on the foot is the point that is closest to the ground, as measured by a vertical projection. This contact point is computed using robust non-linear optimization to find θ^*.

- *outer stance:* only the outer feet are on the ground
- *inner stance:* only the inner feet are on the ground

The first step in the contact solver is to compute the point on each foot that is closest to the ground. The feet are modeled using quintic splines, and the ground is an arbitrary analytic function. We use a non-linear (smooth) optimization to compute the exact point on the foot that is closest to the ground, as measured by vertical projection. This point is then passed to the dynamics function.

We choose to use a complementarity constraint for the ground contact model: The contact point can either be on the ground or in the air. If the contact point is in the air, then the contact forces are zero. Otherwise, the contact forces can be non-zero, and are computed by the dynamics function.

If the contact is active, then we assume that the foot is rolling. Let's define $\boldsymbol{x}$, $\boldsymbol{v}$, and $\boldsymbol{a}$ to be the relative position, velocity, and acceleration of the instantaneous contact point on the foot, with respect to the instantaneous contact point on the ground. We can choose the contact force on the foot to achieve a desired acceleration $\boldsymbol{a}$. Ideally, we would like $\boldsymbol{x} = \boldsymbol{0}$ and $\boldsymbol{v} = \boldsymbol{0}$ at the end of the time step while the contact is active.

Let's assume that the contact is active throughout the time step, and that we are integrating using the symplectic Euler method. We can then look at the the relative position and velocity as a discrete-time linear system, where h is the time step.

$$\boldsymbol{v}_{k+1} = \boldsymbol{v}_k + h\boldsymbol{a}_k$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{v}_{k+1} = \boldsymbol{x}_k + h\boldsymbol{v}_k + h^2\boldsymbol{a}_k$$

We cannot drive both $\boldsymbol{x} \rightarrow \mathbf{0}$ and $\boldsymbol{v} \rightarrow \mathbf{0}$ in a single time step (this follows from basic linear system theory). Instead, we will choose a linear control law to compute $\boldsymbol{a}$ such that the both the position and velocity error asymptotically approach zero. This control law has two parameters: ω_n and ξ, which control the characteristic frequency and damping ratio of the resulting behavior, respectively. We've found that $\omega_n = 0.6/(2h)$ and $\xi = 2.0$ work well for this simulation.

$$\boldsymbol{a}_k = -\omega_n^2\boldsymbol{x}_k - 2\omega_n\xi\boldsymbol{v}_k \tag{2.14}$$

This contact solver is a bit different from those used in commercial simulators, such as Box2d [13] or Bullet[17]. Those simulators require that the collision shapes be composed of primitives (circles and polygons), where as our collision shapes are analytic functions. Analytic collision shapes allow for a better approximation of smooth rolling contact in our simulator, at substantial computational cost. One thing to note is that our simulator requires that the collision shapes are locally convex: in other words, there must be precisely one point where the foot and the ground can meet at any instant.

2.16.2 Constraint Solver

At this point in the process we know the contact point on each foot, which may or may not be in contact with the ground. We also have a way to compute a desired acceleration

for that point, should the foot be in contact. Now we just need to find out which feet are actually in contact. We start by assuming that the contact configuration at the start of the step is correct. If we determine that it is not valid, then we try a new contact mode. Typically this process is done using a linear complementarity solver, but here a simple guess and check is fine, since there are only four options and we have a good initial guess.

How do we check if the constraint is correct? If the contact is active, then we pass the desired acceleration of the contact point (2.14) to the dynamics engine, which in turns computes the next state, and the contact forces that were required to achieve the desired acceleration of the contact point. Suppose that $\hat{\boldsymbol{n}}$ is the normal vector for the ground at the contact point, and $\boldsymbol{f}$ is the contact force vector. Then the contact is valid if $\hat{\boldsymbol{n}} \cdot \boldsymbol{f} > 0$. If the contact is inactive, then we simple require that the contact point at the end of the time step must not be in penetration with the ground.

Thus far, we actually have an incomplete set of contact modes. It turns out that there are rare situations when our "*no-slip*" rolling assumption has no physical solution [73]. In these special cases you can have no-slip or positive contact forces, but not both. Here we choose to allow negative contact forces, rather than sliding. In practice this situation rarely (never) comes up, but the computer will throw a warning if it does occur.

CHAPTER 3

OFF-LINE CONTROLLER DESIGN FOR RELIABLE WALKING OF RANGER

Published in the proceedings of the 2016 International Conference on Robotics and Automation [57].

Author List: Matthew Kelly, Matthew Sheen, Andy Ruina.

3.1 ABSTRACT

We present a method for designing a walking controller for the walking robot Cornell Ranger. Our goal is a controller that can be designed using model-based optimization, and then transferred directly to the robot without the need for after-the-fact hand-tuning. The structure of the controller is hierarchical, with a high-level balance controller that plans step-to-step motions, and a lower-level joint controller that coordinates the individual joint motors to achieve the desired limb motions. The balance controller is designed through optimization, with the explicit goals of *a)* achieving a desired walking speed while *b)* minimizing energy use and *c)* avoiding falls due to disturbances. We demonstrate this walking controller on the Cornell Ranger, and find that the resulting gait is comparable to that of a previous (hand-tuned) controller, with regard to energy use, speed regulation, and fall prevention.

3.2 INTRODUCTION

Here we present a new control architecture for the Cornell Ranger, a bipedal walking robot shown in Figure 3.1. While previous controllers for this robot required extensive hand tuning, the controller presented here is designed using off-line optimization and is meant to transfer directly to the robot without modification.

To advance the science of robot control, we desire algorithms that do not have hand-tuning on the robot as a key final step. Queries of robot builders reveal that such hand tuning of hardware is all too common. To avoid controller tuning on mechanical hardware, we turn to off-line model-based optimization for controller design. This process requires an accurate simulation for the robot, as well as a mechanism for making the controller robust to simulation and modeling errors.

Walking is complicated, so the controllers for most walking robots rely on hierarchical control architectures [125, 79, 26, 90, 60, 121, 117, 9, 49, 62]. These simplify the design process in part by reducing the number of free parameters that describe the controller. The control architecture here incorporates some ideas from the previous controller for the Cornell Ranger [9], SimBiCon [125], and hybrid zero dynamics[121]. The high-level gait control is based on a finite state machine which regulates speed and maintains balance, while the lower-level joint control is simply a proportional-derivative tracking controller on each joint.

It is impractical to automatically design every feature for a walking controller — the state and control space is simply too large for modern techniques to compute a true optimal policy with useful resolution. Instead, we break down the problem. The control architecture *is* manually designed based on our own experience and discussions with experts, as well as the literature. The gains in the low-level joint controllers are

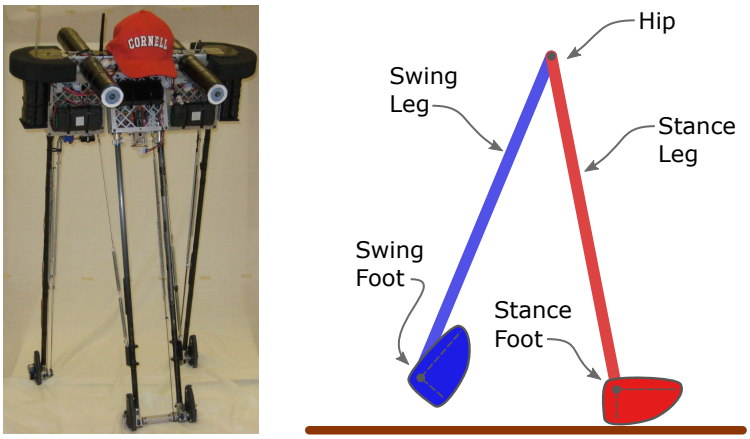

Figure 3.1: **Cornell Ranger walking robot:** A photo and a diagram of Cornell Ranger, our experimental test platform for the controller.

experimentally determined, using standard methods [80]. That leaves the parameters of the balance controller, 15 numbers in our implementation, to be automatically selected using optimization.

3.3 CORNELL RANGER

Our test robot is the Cornell Ranger, which is described in detail in [9, 10]. Here we will present only a short overview.

Ranger, shown in Figure 3.1, is at the bottom of the bipedal robot food chain. It was designed only for low-energy walking over flat terrain. It has four legs that are arranged into an inner and outer pair. This arrangement means that the walking control only needs to stabilize front-to-back motions: lateral motions are passively stabilized. The robot is under-actuated by one degree of freedom: there is no motor that can directly control the angle of the stance leg.

3.3.1 Hardware

Although simulation is useful for controller design and basic testing, the only way to know if a controller really works is to test it on a real robot. For this purpose we use the Cornell Ranger [9, 10].

Ranger lacks of knees, which forces all changes in effective leg length to come from rotations of the feet. The feet on Ranger are curved with small radius, which means that Ranger cannot statically balance with its feet together. Additionally, the circular curve of ranger's feet is truncated close to the heel. This truncated shape allows for the feet to rapidly clear the ground at the start of swing, but also further limits the effective control authority of the ankle motors.

Ranger was designed for energy effectiveness, which we measure using the total cost of transport (CoT). As described in [112] and [113], CoT is the ratio of total energy consumed to the weight multiplied by distance traveled. The total energy is measured at the batteries, and thus includes the power consumption by the motors, sensors, on-board computers, and communication. The best experimental controller on Ranger had a CoT of 0.19, but Ranger's (more reliable) marathon controller [9] had a CoT of 0.28.

Ranger has a variety of sensors. Foot sensors measure the force between the heel of the foot and the ankle joint, which we then threshold to determine if a foot is in contact with the ground or not. Each joint has absolute angle encoders for both the motor and the end-effector. Finally, there is an IMU (gyro and accelerometers) located on the outer legs that we use for estimating the absolute orientation and angular rate of the outer legs.

3.3.2 Model

Our model for the Cornell Ranger is largely based on our previous work [11, 9, 10]. We assume that the robot is a planar biped, with four rigid bodies (outer legs, inner legs, outer feet, inner feet) which are connected by three motors (outer ankles, hip, and inner ankles). We also use a full bench-tested electro-mechanical model of the motors and gear boxes.

There are two notable differences between the model used here and our previous model of Ranger. The first is that here we simplify by assuming that the drive cables connecting the ankle motors to the feet are rigid, where as the previous work [9] treated them as stiff springs. This was done in part because the time-stepping simulation had did not perform well using the stiff spring cable model. Second, we model the shape of the foot as a quintic spline (periodic, with 6 segments), rather than as a complete circle, as illustrated in Figure 3.2. This allows use to simulate interactions between the heel and the ground.

3.3.3 Simulation

The previous simulator for Ranger [9] was designed to study open-loop trajectories with a prescribed sequence of contact configurations. For the research presented in this paper, we need to study the close-loop behavior of the robot for a variety of controllers, some of which will cause the robot to stumble and fall down — a behavior that was not able to be simulated by previous simulations,.

To capture more complex contact sequences, we developed a "time-stepping" simulator for Ranger, which runs a contact-solver on each time-step. This simulation allows

us to model the robot walking over any ground profile, using accurate collision shapes for the robot's feet. The simulation is implemented in Matlab, and then compiled to MEX for speed.

3.3.4 Control Considerations

While walking, the motors of the robot must add energy to the system to compensate for frictional and collisional losses. Due to the small curved feet, Ranger cannot inject much energy by ankle torques through the step, except by push-off with the back foot at the end of each step. This extension is small (a few centimeters), but enough to propel the robot forward and to adjust walking speed. To get the maximum effect of this push-off it must be timed carefully to occur just before the collision on the front foot [63, 105, 47].

Ranger does not have knees, and thus as soon as the push-off is complete, the foot needs to rotate up and out of the way so that it doesn't scuff as the swing leg moves forward. Then the foot needs to rotate back down just before heel-strike. Too early and the foot scuffs; too late and the foot strikes down on the back of the heel, which causes a trip, with the foot rotating back up. In each of these cases, the robot falls. See Figure 3.2 for details regarding foot orientation for push-off and foot-flip.

3.4 CONTROLLER ARCHITECTURE DETAILS

The control architecture here is divided into four levels, arranged highest-to-lowest:

- The *Balance Controller* runs once per step, at mid-stance, setting the five parameters that describe the gait controller, to achieve balance.

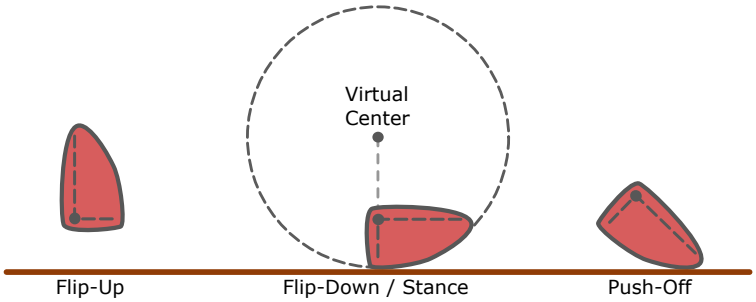

Figure 3.2: **Ranger Foot Diagram:** Ranger's feet are small, and their soles are sections of circular arcs. Here we show the three target configurations used by the controller. Flip-up is used for the swing foot, allowing the foot to clear the ground, since the robot has no knees. The flip-down/stance configuration is used by the stance foot for most of the step, providing a steady base for the robot. The final configuration, push-off is used to rapidly extend the foot, propelling the robot forward for the next step.

- The *Gait Controller* is a finite-state machine, shown in Figure 3.3, which sets the target angles and rates for the joint controllers.
- The *Joint Controllers* are proportional-derivative controllers which compute the command torque for each joint.
- The *Motor Controllers* are proportional-integral controllers which compute a low-level PWM commands to achieve a commanded torque in each joint.

We assume that both the robot and controller are left-right symmetric. We will describe the controller for the case when the outer feet are on the ground. At the conclusion of the step, the whole controller is mirrored, with the inner legs becoming the new stance legs.

3.4.1 Motor Control

The motor controllers are at the bottom level of the controller. They run a simple proportional-integral control loop at 2 kHz on each of the three joint motors (outer an-

kle, inner ankle, and hip), tracking a desired joint torque. These motor controllers are coded at a low-level in the robot, and we did not change these.

3.4.2 Joint Control

While the robot is walking, the joint controllers (outer ankle, inner ankle, and hip) are continuously running simple proportional-derivative controllers at 500 Hz. Each controller computes a command torque u, which is sent to corresponding motor controller. The reference angle q^*, rate $\dot{q}^*$, and torque u^* are all sent from the gait controller. The measured joint angle and rate are given by q and $\dot{q}$.

$$u = u^* + K_P\left(q^* - q\right) + K_D\left(\dot{q}^* - \dot{q}\right) \tag{3.1}$$

3.4.3 Gait Control

The gait controller coordinates the motion of the three joints on the robot, sending reference commands to the joint controllers at 500 Hz. There are two parts to the gait controller. The first is a finite-state-machine (FSM), which is shown in Figure 3.3. During a single walking step, this FSM can be in either the *glide* mode, or the *push* mode. In glide mode, the robot is smoothly moving forward, with the gait controller pulling the swing leg through the step. In push mode, the robot extends the rear foot, propelling the robot forward, while simultaneously rotating the swing foot down in preparation for heel-strike.

Glide Mode: The reference commands sent to the swing foot joint controller are simple: the angle is constant, selected such that the foot will not scuff the ground, and the rate and torque references are zero.The reference angle and rate for the stance foot

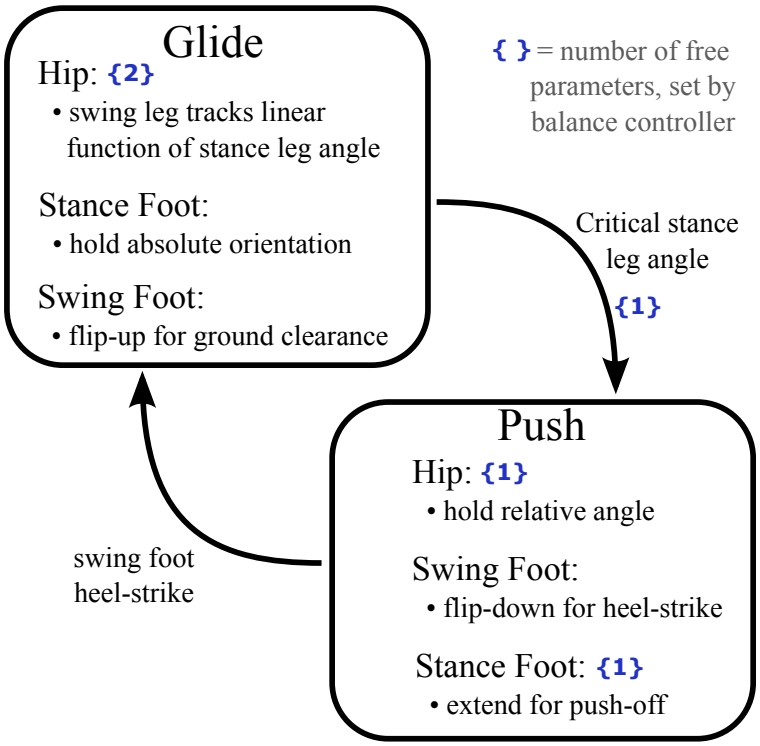

Figure 3.3: **Gait Controller:** A finite-state-machine that sets the targets for the joint controllers. It has two states: Glide (or swing) and Push (or step-to-step transition). There are a total of {5} parameters, which are set once per step by the balance controller. At each heel-strike transition, the old swing leg becomes new stance leg, and vice versa.

are selected to keep the absolute orientation of the foot constant, while the robot rotates over it. The hip joint in glide is more complicated: the joint angle and rate are set based on a linear function of the stance leg angle, and the reference torque is computed to compensate for the torques on the joint due to gravity and the hip spring (which connects the two legs). This phase-based tracking is inspired by [121].

Push Mode: Both ankle joints are set to maintain a constant absolute orientation of the foot, with the stance foot pushing-off the ground and the swing foot flipping-down for heel-strike, as shown in Figure 3.2. During push, the hip joint holds a constant angle, with the help of feed-forward torque compensation.

Transitions: There are two state transitions in the finite-state-machine. The transition into glide mode is triggered when the contact sensors on the swing foot detect heel-strike. The transition into push mode is triggered as the stance leg passes a critical absolute angle.

Parameters: The gait controller has five free parameters that are set by the balance controller. These parameters are illustrated in Figure 3.3 and are: 1,2) constant and linear coefficient for the hip joint reference trajectory in glide mode; 3) constant hip angle reference during push mode; 4) critical angle of the stance leg for transition from glide mode to push mode; and 5) absolute angle reference for the stance foot during push mode.

3.4.4 Balance Control

The top-level of the control architecture is the balance control, which runs once per step at mid-stance. It changes the five parameters of the gait controller to regulate balance and walking speed. For example, if the robot is walking too slowly, it will increase the reference angle for the push off, adding more energy to the system.

There is a single input the the balance controller: the robot's speed at mid-stance. Thus, the balance controller is simply a function that maps the mid-stance speed of the robot at mid-stance to the set of five parameters that are passed to the gait controller. Here, we implement this function using a look-up table, storing the five parameter values for zero speed, the target speed, and the maximum expected speed. For intermediate speeds we use linear interpolation.

The look-up table for the balance controller has a total of 15 entries (5 parameters

at each of 3 speeds), which we compute using off-line optimization, using methods discussed in §3.5.

3.5 CONTROLLER DESIGN

We claim that the controller is designed "using optimization", but we also acknowledge that there are many decisions that are made by humans as well.

We designed the architecture for the walking controller (§3.4) through insight and intuition, which we acquired through experiments, discussions at technical conferences, and the literature [9, 125, 121].

There are several constant parameters in the controller, which are also manually set. These include the orientation of the stance foot during glide mode (selected such that the ankle joint lies directly below the virtual center of the foot), and the relative angle of the ankle joint required during flip up (selected to be close to the joint limit).

The joint controllers (§3.4.2) in the ankles and hip all have proportional and derivative gains, which are selected by simple experiments on the hardware. These experiments are easily repeatable by any controls engineer using standard methods [80].

The final set of parameters (§3.4.3, §3.4.4), are selected entirely by computer optimization (§3.5.1, §3.5.2). These parameters are copied directly from the output of the optimization to the robot. There is no final hand-tuning step.

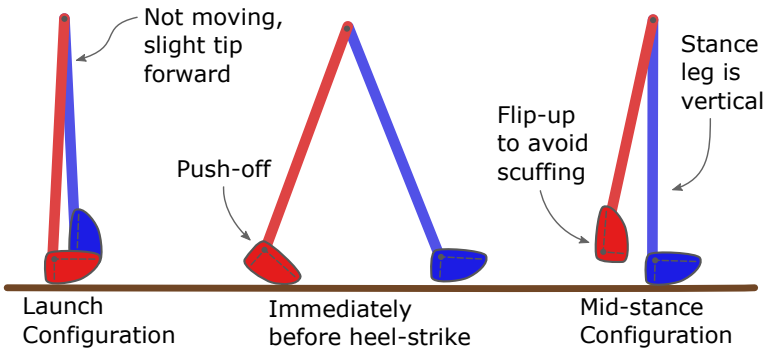

Figure 3.4: **Ranger Configurations** The launch configuration (left) shows the static configuration that we use to start Ranger walking, both in simulation and in reality. The middle configuration shows the robot immediately before heel-strike. The final configuration (left) is mid-stance, with the supporting leg vertical. This is the configuration that triggers an update from the balance controller.

3.5.1 Objective Function for Optimization

The balance controller (§3.4.4) is parameterized by 15 numbers, the 3×5 look-up table entries. These are computed by off-line optimization. The objective function in this optimization is chosen to reject disturbances, while minimizing speed and cost of transport.

The objective function evaluates a candidate controller by running several simulations. Each of these simulations starts from the same launch configuration, shown in Figure 3.4, and then the robot walks over several different ground profiles. The first ground profile is flat and level ground. The remaining ground profiles serve as disturbance tests and are either constant slopes (uphill or downhill) or rolling hills (sine curves).

A candidate controller receives a reward for each successful step that it takes, where the reward is related to both the speed and energy used to complete the step. The reward is maximized by a controller that walks at the desired speed using little energy. Each trial (ground profile) has a fixed number of steps. If the robot falls during a trial, then it

receives a reward of zero for that and all future steps in the trial.

The optimization then finds the controller that maximizes the sum of rewards over all trials. The structure of the objective function is such that fall avoidance is more important than speed regulation or energy effectiveness.

3.5.2 Optimization Method

Here we used a Covariance Matrix Adaptation Evolutionary Strategy (CMAES) [44, 45] to optimize the balance controller, because it deals well with our non-smooth objective function. We initialize the algorithm by first estimating bounds on the parameters. For example, the push-off target angle must be within the actuator limits, and the hip trajectory coefficients should be roughly consistent with bipedal walking (the swing leg must travel from back to front, etc.).

3.6 RESULTS

3.6.1 Off-line Controller Design (Optimization)

We implemented the entire design process for the balance controller in Matlab, with most of the simulation code being compiled to MEX for faster run-time. The code takes about 10 minutes to compute the optimal control parameters running on a laptop (Intel Quad-Core i7 CPU Q720, 1.60 GHz), where each walking step of the robot takes about 0.034 seconds to compute, for a total of about 18,000 steps per optimization.

Here we designed a controller to achieve a single walking speed, although we could

repeat the process to compute walking gaits for a whole set of target speeds.

3.6.2 Walking: Simulation vs. Experiment

Our controller design process relies on accurate simulation of the robot; in this section we compare experimental data collected during walking to what we expected based on the simulation.

We collected data on the robot over two trials. In each case the robot walked about 80 meters over a stone-tiled floor. The surface of each tile was flat, but there was a change in height of roughly 2 millimeters between the edge of any two tiles. Over a large scale there is no measurable slope to the floor.

The model and simulation match well on power consumption: the simulation predicts 22.2 Watts, and the two trials used 22.1 Watts and 22.9 Watts respectively. The simulation does not do as good of a job at predicting speed, with the simulation walking at 0.66 meters per second, and real robot walking at about 0.58 meters per second. This difference in speed makes the cost of transport a bit higher on the real robot: 0.49 instead of 0.42 in simulation. These results are summarized in Table 3.1.

We also compared the angles of the robot's legs and feet, as functions of time, to those predicted by the simulation. We selected six consecutive walking steps at random,

Table 3.1: Comparison of simulation and experimental data.

	Simulation	Trial 1	Trial 2
Duration	119 s	162 s	159 s
Distance	79 m	91 m	91 m
Average Power	22.2 W	22.1 W	22.9 W
Total CoT	0.42	0.48	0.49
Average Speed	0.66 m/s	0.57 m/s	0.58 m/s

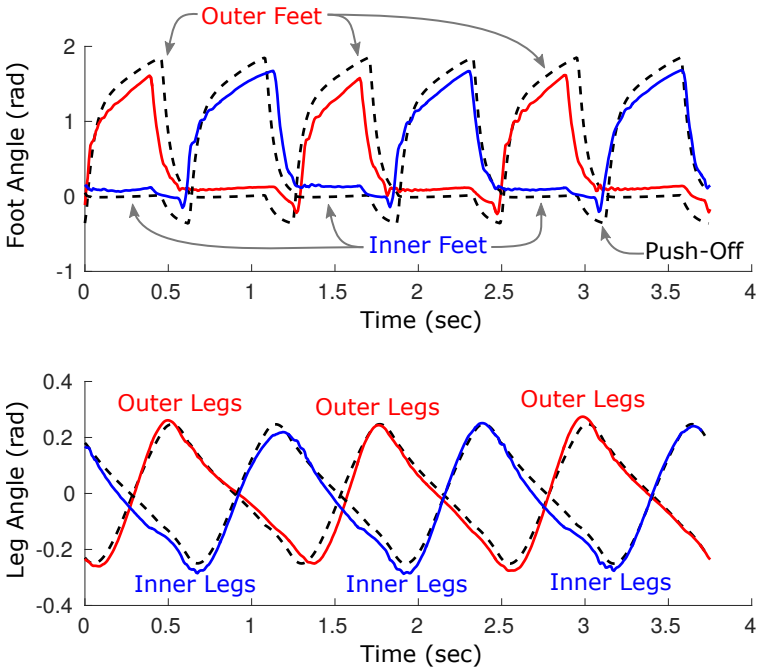

Figure 3.5: **Walking: Simulation vs. Experiment** Six steps of periodic walking. The dashed lines show the simulation and the solid lines show experimental data. All angles are absolute, and measured from vertical. Most angles match well, but there is a noticeable difference between simulation and experiment during push-off.

during steady state walking, for both the simulation and the experimental data. We found that the leg angles are a close match throughout the gait. The ankle angles are close for much of the step, but there are significant deviations during push-off, as shown in Figure 3.5.

3.6.3 Robustness Experiments

We evaluated the robustness of the "new" controller by comparing its performance to the previous "old" controller, which was used for Ranger's marathon walk in 2011 [9]. Each controller was first evaluated walking without disturbances to establish a base-line. Then we subjected the robot to disturbances and measured each controller's ability to

regulate walking speed and prevent falls. The disturbances are illustrated in Figure 3.6, and the results are given in Figure 3.7.

Baseline: We established a baseline performance for each controller by having them walk on a flat stone-tiled floor. The only disturbances were the slight perturbations caused by the height variations ($\approx$ 2 mm) between stone tiles. The old controller walked 183 meters with no sign of falling, while the new controller walked 732 meters, falling 6 times.

Trial 3: Our first disturbance was walking on a more challenging floor, which had sagged over the decades. The resulting ground profile was smooth, but with slight rolling hills: the peaks were 4 meters apart, and the peak-to-trough height was about 2 centimeters. The old controller walked 429 meters, falling 3 times, while the new controller walked 501 meters, falling 6 times.

Trial 4a: Next, we brought the robot back to the original stone-tiled floor and removed the "hip spring" which connects the inner and outer legs, inducing a substantial modeling error. Both controllers walked a distance of 183 meters. The old controller did not fall and the new controller fell 6 times.

Trial 4b: We returned the hip spring to the robot, and then added a small mass (0.35 kg) to the outer legs. The weight was positioned 0.12 meters in front of the hip joint, as shown in Figure 3.6.The old controller was unstable: it continually sped up until is stumbled and fell, a total of 19 times in 183 meters. The new controller walked well with the added weight on the front. It increased it's speed and only fell 3 times in 183 meters.

Trial 4c: For our final disturbance, we moved the mass (0.35 kg) to the back of the robot, still on the outer legs, as shown in Figure 3.6. In this case the old controller

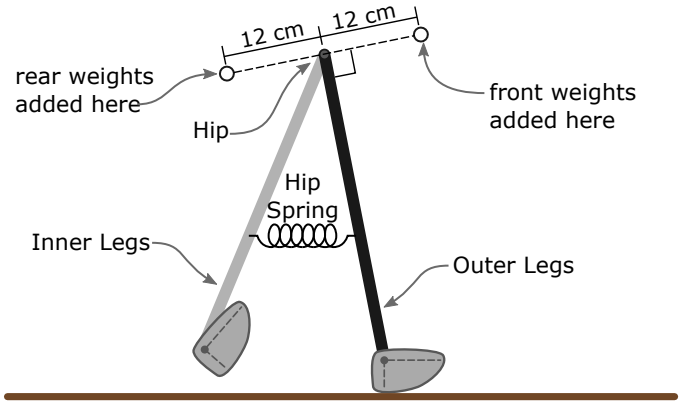

Figure 3.6: **Disturbance Diagram** In Trial 4a, we removed the hip spring (k = 7.6 Nm/rad), which coupled the angles of the inner and outer legs. In Trial 4b we added a mass of 0.35 kg in front of the outer legs, in the location shown. In trial 4c, we added the same 0.35 kg mass behind the outer legs.

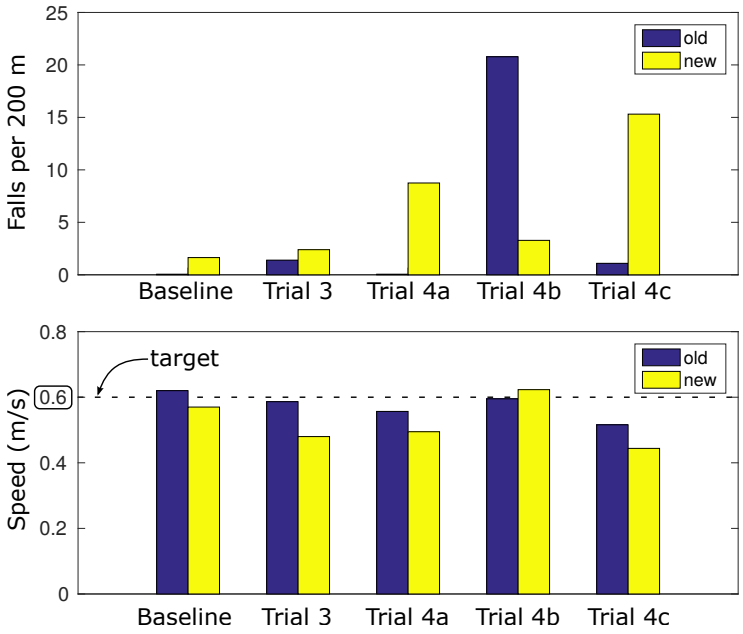

Figure 3.7: **Robustness Test** Each of the trials compares the old (2011 Marathon) controller with the new controller presented in this paper. The baseline trial was conducted on a flat stone-tiled floor. Trial 3 was walking on smooth ground with rolling hills: 4 meter peak to peak, 0.02 meters peak to trough. Trials 4a-4c were all conducted on the flat stone-tiled floor to test robustness to modeling errors: missing hip spring (4a), weights added to front (4b), and weights added to back (4c).

performed well, only falling once in 183 meters. The new controller did not fare as well, falling 14 times in 183 meters.

3.7 DISCUSSION

In this paper we presented a new controller for the Cornell Ranger, a bipedal walking robot. The controller architecture is based on variety of ideas taken from previous walking controllers, including [125, 121, 9]. Although the low-level joint controllers are all hand-tuned proportional-derivative tracking controllers, the parameters of the high level balance controller are designed entirely using off-line optimization. In the results section of this paper, we seek to answer two questions, detailed below.

1) ***How well does the simulation match experimental results?*** We found that the simulation is a good match for the experimental data on power use and leg angles. The ankle angles during push-off do not match well. This discrepancy is likely due, in part, to our assumption that the drive cables for the ankle joint are inextensible. As a result, the effect of push-off on the real robot is reduced, which might explain the reduced walking speed seen in the experiments.

2) ***How does this new controller compare to the previous (hand-tuned) controller?*** In general, the previous (hand-tuned) controller out-performs the new controller, although not by a huge margin. This assessment is based on the fact that it does a better job of regulating speed in the presence of disturbances, and has a lower fall rate in all but one trial.

Summary: We were able to design a controller using off-line optimization, without after-the-fact hand-tuning, which walked reasonably well under a variety of disturbances. Despite this, the previous hand-tuned controller out-performed the new controller on most tests. Although the new controller was designed in large part by computer optimization, the architecture of both the controller and optimization were manually selected. It is likely that these manual choices limited the ultimate performance of the new

controller.

3.8 FUTURE WORK

The work presented here is preliminary: it shows that we can set up a model-based optimization that can design a controller in simulation that we can directly use on a real robot. Although the initial results are passable, there is still much left to do.

In the current implementation, the balance controller only updates the trajectories of the robot once per step. In the future we would like to make these updates continuous, allowing the robot to more quickly react to external disturbances such as a sudden push.

An additional area for improvement is the look-up table for the balance controller, which now uses only a single input: the mid-stance speed of the robot. Ideally, this policy would use at least the state of the stance and swing legs, rather than this simple one-dimensional projection.

Our model of Ranger is reasonably accurate, but there are still some discrepancies between model and reality, especially during push-off. In the future we plan on moving the last few iterations of the optimization process to the robot, using experimental data rather the simulation for automatic fine-tuning of the controller.

Finally, we plan on designing the controller to be capable of walking at several speeds, rather than just the one speed that it is capable of now.

CHAPTER 4

NON-LINEAR ROBUST CONTROL FOR INVERTED-PENDULUM 2D WALKING

Published in the proceedings of the 2015 International Conference on Robotics and Automation [56].

Author List: Matthew Kelly, Andy Ruina.

4.1 ABSTRACT

We present an approach to high-level control for bipedal walking exemplified with a 2D point-mass inextensible-legs inverted-pendulum model. Balance control authority here is only from step position and trailing-leg push-off, both of which are bounded to reflect actuator limits. The controller is defined implicitly as the solution of an optimization problem. The optimization robustly avoids falling for given bounded disturbances and errors and, given that, minimizes the number of steps to reach a given target speed. The optimization can be computed in advance and stored for interpolated real-time use online. The general form of the resulting optimized controller suggests a few simple principles for regulating walking speed: 1) The robot should take bigger steps when speeding up and should also take bigger steps when slowing down 2) push-off is useful for regulating small changes in speed, but it is fully saturated or inactive for larger changes in speed. While the numerically optimized model is simple, the approach should be applicable to, and we plan to use it for, control of bipedal robots in 3D with many degrees of freedom.

4.2 INTRODUCTION

In the long run we would like to explain, and robotically reproduce, the efficiency, speed and versatility of human bipedal locomotion in various terrains. Here we work towards this goal by designing a controller for walking, focusing on the ability to avoid falling in level walking, considering various disturbances. Bipedal locomotion violates many common assumptions in basic classical control: it is nonlinear, nonholonomic [97], has discontinuities, and changes governing equations during the motion. Hence there is no generally-accepted classical-controls approach to stabilizing walking. Here we pursue a hierarchical controller strategy for walking control.

Most balance controllers use some sort of hierarchical structure, factoring the control problem for walking into two parts: a high-level controller concerned with overall robot position and balance, and a low-level controller that deals with the many internal degrees of freedom. The high-level controller may specify foot placement, leg length, and ground reaction forces and torques to achieve balance; while the low level controller determines individual motor commands to realize these sub-goals.

A few examples of robots that use a hierarchical approach include Atlas [62], Asimo [49], MABEL [103], and the Cornell Ranger [9], all of which use a balance controller based on one or another simple low-dimensional model. For instance, the capture point controller [90, 60, 89, 26] uses the linear inverted pendulum (LIP) model to plan future foot step locations and a trajectory for the center of mass. These high-level commands are then realized by giving commands to individual motors based on inverse kinematics and dynamics of a full high-dimensional model of the robot.

Although these simple models are primarily used to make the locomotion balance problem tractable, they also seem to be genuinely good models for balance control.

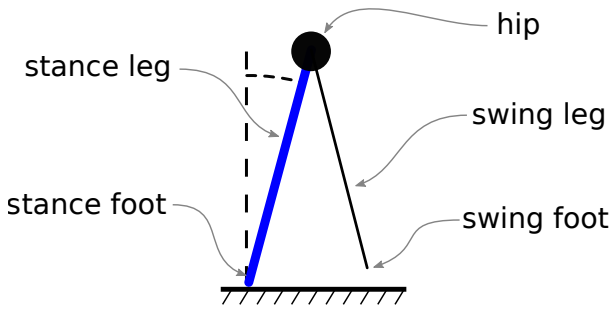

Figure 4.1: **Point-mass walking model.** All mass is concentrated at a point at the hip. The legs are mass-less. There are two controls: The relative angle of the legs at heel-strike, and the size of the push-off impulse by the trailing leg just before heel-strike. Uncertainties in sensing, model, and actuation, as well as actuator limits on push-off impulse and step length, are discussed in the text.

Because of their utility, however motivated, most successful simple models are based on a point mass at or near the hip, light or mass-less legs, and actuation to control the time and location of foot falls and the ground reaction force. The models tend (sensibly, we think) to neglect the effects on balance of upper body motions and the details of leg swing between steps.

Within the class of point-mass models just mentioned, there are still choices. One common choice is to use the 'linear inverted-pendulum' (LIP) model, in which the height of the point mass is held constant (e.g. with 'capture point' and 'zero moment point' control). Here, however, we use an 'inverted-pendulum' (IP) model which has constant leg length instead of constant hip height. The IP model has not been as frequently used for walking balance control as the LIP model, in part because of the mathematical simplifications of the LIP (e.g. [55]).

We use the IP model for control because it uses energy more effectively than the LIP model. More precisely, Srinivasan and Ruina [105, 106] used numerical trajectory optimization to find positive-work minimizing gaits on a general point-mass walker and found that the energy-minimal walking gait used constant-length legs: the IP model of

walking. In this model the stance leg length remains constant, and the push-off at the end of the stance phase is (at least approximately) impulsive. This is a powered version of the so-called 'Simplest Walker' [31]. The energy effectiveness of impulsive push-off and straight leg walking is also discussed in Kuo [63] for a similar model. Because of the IP's good energetics, it is the basis of our walking robots. In the design of the controller here, however, we do not explicitly consider energy, focusing instead on robustness to disturbances and quick return to a nominal gait.

Zaytsev [127] has introduced a simple walking controller for this IP model, based on finding simple control laws that maximize the distance from failure boundaries. Here, we extend Zaytsev's work by optimizing an arbitrary-form controller that is robust to disturbances and also stable. Additionally, the design process here is fast enough that it can be repeated to stabilize a range of speeds thus generating a full feedback policy for a range of walking-speed goals. Despite a difference of methods with Zaytsev, there are common results (discussed below). This may indicate that we are extracting features that are necessary aspects of any robust walking balance controller that is reasonably constrained by actuator and sensor limits.

4.3 WALKING MODEL

Our model consists of a point-mass hip on mass-less in-extensible legs (Fig. 4.1). The model has two control inputs at each step: the angle of the stance leg (ϕ) at heel-strike, and the push-off impulse (p) that occurs immediately before heel-strike.

A step comprises four distinct phases, starting from a mid-stance (more or less a Poincaré Section) where the stance leg is vertical and rotating clockwise . These phases are illustrated in Fig. 4.2: a swing-down to the step angle (ϕ); followed by an impul-

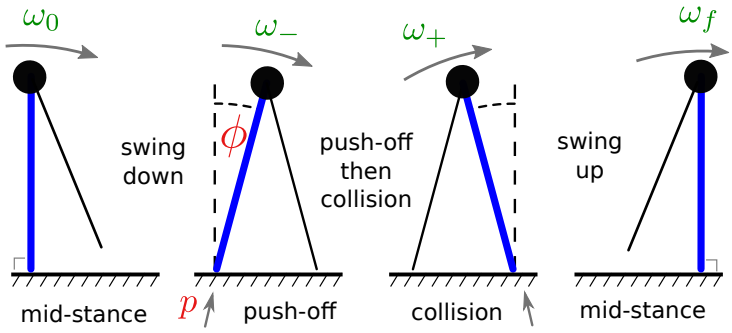

Figure 4.2: **One walking step.** A step starts when the stance leg passes clockwise through the vertical orientation. The hip then falls to the right. Just before the swing leg hits the ground there is push-off from the ground on the stance leg. Immediately after push-off, the swing leg hits the ground and becomes the stance leg. The former stance leg leaves the ground and the new stance leg swings up to the start of the next step. In the leftmost and rightmost pictures, the angle of the swing leg is arbitrary (because it is fully controlled).

sive push-off (p); then heel-strike and leg-switch; and finally a swing-up to mid-stance. Through-out the paper we measure walking speed in the various phases of the gait by the dimensionless (divide by $\sqrt{g/\ell}$) angular velocity ω of the stance leg.

4.3.1 Actuation

The step length is determined by the step angle: step length $= 2\ell \sin\phi$. We neglect swing-leg inertia: its motion does not affect the motion of the stance leg. The step angle is bounded (constrained) to mimic the joint limits of a real robot.

The impulsive push-off is meant to be a proxy for the energy injected into the walking motion by the extension of the trailing leg (ankle extension). The bounds on the push-off are a proxy for the maximum power available for leg extension.

4.3.2 Equations of Motion

The model is a simple inverted pendulum, but with impulses and resets at each step. The parameters m, g, and ℓ represent the robot's mass, gravitational acceleration, and leg length respectively. The control variables ϕ and p represent the step angle and push-off impulse respectively.

The step starts with the stance leg vertical and rotating with speed (ω_k) towards heel-strike. The angular rotation rate immediately before push-off can be obtained via conservation of energy:

$$\omega^- = \sqrt{(\omega_k)^2 + \frac{2g}{\ell}(1 - \cos\phi)}. \tag{4.1}$$

Next, a push-off impulse is applied to the point-mass along the stance leg, changing the hip's velocity vector. After that, the swing leg becomes the new stance leg as it collides with the ground, exerting another impulse on the point-mass hip. The composition of these two collisions, governed by angular momentum balance about the new stance foot, yields the rotational speed of the new stance leg (ω^+),

$$\omega^+ = (\omega^-)\left(\cos^2\phi - \sin^2\phi\right) + \frac{2p}{ml}\cos\phi\sin\phi. \tag{4.2}$$

After the two collisions, the stance leg swings up to the next mid-stance, again ruled by conservation of energy,

$$\omega_{k+1} = \sqrt{(\omega^+)^2 - \frac{2g}{\ell}(1 - \cos\phi)}. \tag{4.3}$$

4.3.3 Feasibility conditions

A walking step is considered successful (i.e. the robot did not fail) if two conditions are met: first, the stance leg is always in compression (this is also a no-flight condition), and

second, the speed after the heel-strike must be sufficient to continue on to the next step without falling over backwards. These conditions are expressed with the four inequality constraints:

$$\omega^{-} < \sqrt{\frac{g}{\ell}\cos\phi} \tag{4.4}$$

$$0 < (2m\ell)(\omega^{-})\cos\phi\sin\phi - p\left(\cos^2\phi - \sin^2\phi\right)2\ell \tag{4.5}$$

$$\sqrt{\frac{2g}{\ell}(1-\cos\phi)} < \omega^{+} < \sqrt{\frac{g}{\ell}\cos\phi}. \tag{4.6}$$

The first inequality 4.4 is a restriction on the speed before the push-off impulse. The second inequality 4.5 is a restriction on the collision impulse - the collision cannot pull the walker towards the ground. The final inequalities 4.6 are restrictions on the lower and upper bounds on the speed after collision, preventing both falling over backwards and flight.

4.3.4 Disturbance Model

In this paper we claim that the controller is robust to errors in modeling 4.7, actuation (4.8)(4.9), and sensing 4.10. Each type of error is modeled as a perturbation:

$$\ell := \ell + \delta_\ell, \qquad |\delta_\ell| \le \Delta_\ell \tag{4.7}$$

$$p := p + \delta_p, \qquad |\delta_p| \le \Delta_p \tag{4.8}$$

$$\phi := \phi + \delta_\phi, \qquad |\delta_\phi| \le \Delta_\phi \tag{4.9}$$

$$\omega_k := \omega_k + \delta_\omega, \qquad |\delta_\omega| \le \Delta_\omega \tag{4.10}$$

We represent the vector of disturbances:

$$\boldsymbol{\delta} = [\delta_\ell, \delta_p, \delta_\phi, \delta_\omega], \quad \boldsymbol{\delta} \in \mathcal{D} \tag{4.11}$$

The set of disturbances $\mathcal{D}$ may be thought of as a four-dimensional hyper-rectangle. We define a set of maximal disturbances $\mathcal{D}_{\max}$ that correspond to the 2^4 corners[1]of this

hyper-rectangle.

$$\mathcal{D}_{\max} = \{[\pm\Delta_\ell, \pm\Delta_p, \pm\Delta_\phi, \pm\Delta_\omega]\} \tag{4.12}$$

We use the full set of disturbances ($\mathcal{D}$) for doing controller verification and testing, and the smaller sub-set of maximal disturbances ($\mathcal{D}_{\max}$) for controller design and optimization.

4.4 CONTROLLER DESIGN

Our controller is a function that maps the estimated mid-stance[2] speed ($\hat{\omega}_k$) to a desired push-off impulse (p) and step angle (ϕ). When implemented on a real robot, the push-off impulse is related to how much energy the extension of the trailing leg adds to the robot before the next step. The step angle corresponds to a target foot-placement location on the ground.

$$\{p, \phi\} = K(\hat{\omega}_k) \tag{4.13}$$

The controller K aims to stabilize the walking gait to a user-specified target speed (ω^*). Here, we outline the design process for a single given target speed. The method is then repeated to find controllers for a variety of speeds (see results section).

We discretize the range of possible input mid-stance speeds $\Omega_I = [0, \omega_{\max}]$. The output of the controller at each of these grid-points is computed by solving an optimization problem: p and ϕ are the controls that stabilize the mid-stance speed ($\omega_\infty \to \omega^*$) in the fewest number of steps while preventing falls despite all possible bounded disturbances. At run-time the controller is evaluated via linear interpolation over this grid.

[1]For example, $[+\Delta_\ell, +\Delta_p, -\Delta_\phi, +\Delta_\omega] \in \mathcal{D}_{\max}$

[2]Mid-stance is the point during a step when the stance leg is vertical.

We express the design requirements (stability and robustness) as constraints in the optimization problem that defines the controller. Thus, if the optimization returns a feasible solution, then the controller will satisfy the design requirements precisely for every grid point in the controller.

4.4.1 Asymptotic Stability

We want the controller to bring the robot towards the desired walking speed (ω^*), starting from any mid-stance speed ω_k drawn from the set of allowed initial speeds Ω_I. This goal can be expressed as the need for reduction at each step k of a discrete Lyapunov function (V), with V defined as the mid-stance speed-error squared:

$$V(\omega_k) = (\omega^* - \omega_k)^2 \tag{4.14}$$

The controller is asymptotically stable if the Lyapunov function decreases at each successive step:

$$V(\omega_{k+1}) < V(\omega_k) \quad \forall \omega_k \in \Omega_I \tag{4.15}$$

This condition 4.15 is imposed as a constraint in the controller design, thus any controller will be asymptotically stable, *in the absence of disturbances*.

4.4.2 Robust Stability

Asymptotic stability in the absence of disturbances is good, but we would also like to show that the controller is still stable given any disturbance $\delta \in \mathcal{D}$. In this case, it is not possible to show convergence to a point, but we can (and do) show convergence to some finite set Ω_G that contains the target mid-stance speed.

For this aspect of the controller, we separate the controller design and verification. For the controller design we find the controls that minimize the Lyapunov function 4.14 over the finite the set of maximal disturbances $\mathcal{D}_{\max}$. The set $\mathcal{D}_{\max}$ is a proxy for the disturbance that precisely maximizes the Lyapunov function. Note that ω_i is the next mid-stance speed, subject to disturbance $\boldsymbol{\delta}_i$. The controls (p, ϕ) and initial state (ω_k) are held constant over each step in the sum:

$$f(p, \phi) = \sum_{\boldsymbol{\delta}_i \in \mathcal{D}_{\max}} (\omega^* - \omega_i)^2 \tag{4.16}$$

Once the controller has been designed, we do stability verification by running an additional optimization to find the precise disturbance that maximizes the Lyapunov function at the next step $\forall \omega_k \in \Omega_I$. The result of this optimization is the size of the goal set Ω_G that satisfies the following equations.

$$V(\omega_{k+1}) < V(\omega_k) \qquad \forall \omega \in \Omega_I - \Omega_g \quad \forall \boldsymbol{\delta} \in \mathcal{D} \tag{4.17}$$

$$V(\omega_{k+1}) \leq V(\omega_k) \qquad \forall \omega_{k+1} \in \Omega_G \quad \forall \omega_k \in \delta\Omega_G \quad \forall \boldsymbol{\delta} \in \mathcal{D} \tag{4.18}$$

The first of these equations 4.17 shows that the controller is asymptotically stable to Ω_G, even in the presence of disturbances. The second equation 4.18 shows that once the walking speed is inside the boundary of the goal set $(\delta\Omega_G)$, that there is no disturbance that can push it out.

4.4.3 Fall Prevention

In addition to reaching the target walking speed, we would like that any execution of the controller avoid falling. This is accomplished by adding the constraints (4.1)-(4.3) to the optimization problem, and requiring that they hold for $\forall \boldsymbol{\delta} \in \mathcal{D}_{\max}$.

4.4.4 Implementation

The constraints for the optimization problem require solving total of 17 simulated walking steps: one nominal step (without perturbation), and 2^4 perturbed steps (one for each disturbance in $\mathcal{D}_{\max}$). All of these simulated steps start from the same initial state and use the same control.

For each of these simulated walking steps, the intermediate speeds ω^-, ω^+ and next step speed ω_{k+1} are passed as decision variables in the optimization. This allows the dynamics (4.1)-(4.3) and no-falling constraints (4.4)-(4.6) to be expressed as simple non-linear functions of the decision variables. The asymptotic stability condition 4.15 and objective function 4.16 are both quadratic in the decision variables. Posing each optimization problem in this way makes it easy to compute gradients and solve quickly using standard non-linear constrained optimization packages.

4.5 RESULTS

In this section we present an optimal controller, as designed using the framework presented in this paper.

4.5.1 Design Parameters

All parameters in this paper (Table 4.1) have values that roughly match the Cornell Ranger [10].

4.5.2 Optimization

We used Matlab's [70] FMINCON optimization software to solve all optimization problems presented here.

The optimization problem that defines the controller is evaluated for each point on a grid. The results here were computed using a grid of 50 points. Similar results are obtained for grids with fewer points, although there is a slight degradation in performance due to interpolation errors at run-time.

The results presented here took approximately 62 seconds[3] to generate in Matlab. Three controllers were generated (including verification), with 50 grid-points each, giving a average time of 0.4 seconds per grid-point.

4.5.3 Optimized Push-Off Controller

The optimized push-off controller is shown in Fig. 4.3. It has a relatively simple form: big push-off to increase speed, and no push-off to slow down. For fast walking, the

[3]Processor: Intel Core i5-3570K @ 3.40GHz x 4

Table 4.1: Parameters

Symbol	Name	Value
m	mass	9.91 kg
g	gravity	9.81 m/s^2
ℓ	leg length	0.96 m
$p_{\max}$	max push-off impulse	12.2 kgm/s
$\phi_{\max}$	max stance angle	30° = 0.52 rad
$\omega_{\max}$	max mid-stance speed	2.56 rad/s
Δ_ℓ	leg length error bound	± 0.05 ℓ
Δ_p	push-off error bound	± 0.05 $p_{\max}$
Δ_ϕ	step length error bound	± 0.05 $\phi_{\max}$
Δ_ω	mid-stance speed error bound	± 0.05 $\omega_{\max}$

push-off is saturated at the maximum value for much of the domain.

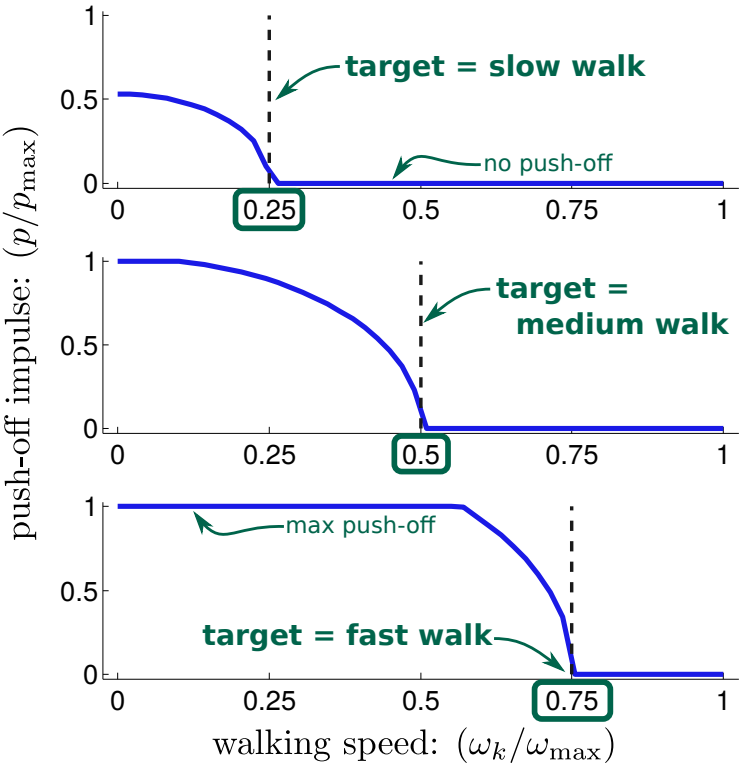

Figure 4.3: **Optimized Push-Off Controller.** This figure shows the optimal push-off controller for three different walking speeds. The general trend is simple: too fast → no push-off, too slow → big push-off. Notice that the actuation is saturated for much of the domain for the *fast* controller.

4.5.4 Optimized Step-Length Controller

The optimized step-length controller is shown in Fig. 4.4. The general trend is that you should take small steps when you are near the target speed, and bigger steps otherwise. In the absence of push-off ($p = 0$), taking bigger steps increases the energy lost due to collision, and will slow the walking gait. If the push-off is non-zero, then it is scaled by a term that increases with the step angle. In effect, taking bigger steps allows the push-off impulse to be more effective. This explains the general trend: if a large push-off is used,

then that term dominates the collision losses, and the walking gait speeds up; if a small push-off is used, then the collision losses dominate and the walker slows.

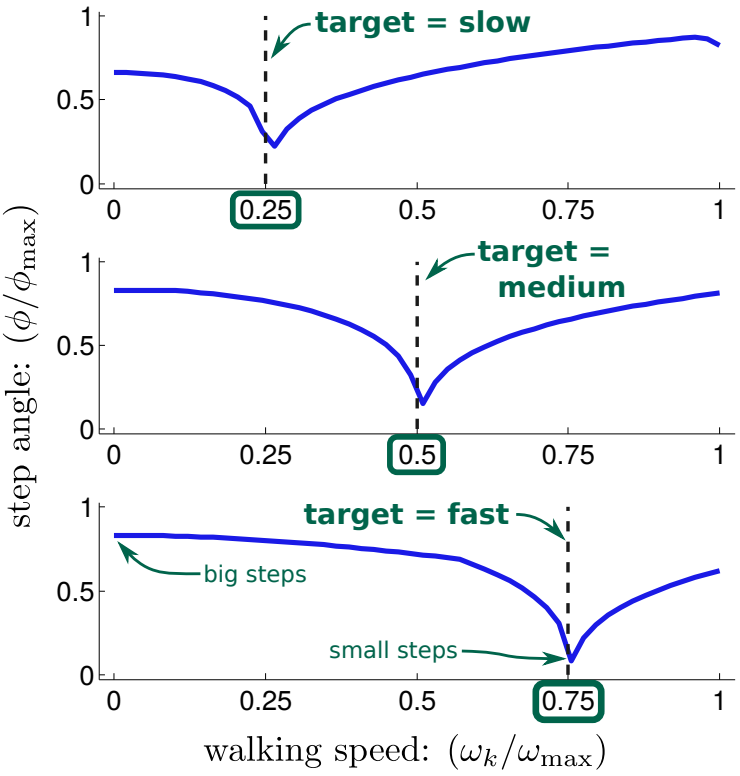

Figure 4.4: **Optimized Step-Length Controller.** This figure shows the optimal step-length controller, for three different walking speeds. The general strategy is to use small steps for the nominal walking speed, and take larger steps otherwise. The slight corner in the slow- and medium-speed walking controllers for high speeds is cause by the no flight constraint 4.4.

4.5.5 Stability and Robustness

Given the optimal controller shown in Figs. 4.3 and 4.4, it is possible to compute the closed-loop dynamics of the system, mapping the mid-stance speed from one step to the next. Figure 4.5 shows this so-called one-step map for an intermediate speed walking gait. The horizontal axis shows the mid-stance speed at step k, and the vertical axis shows the mid-stance speed the next step $(k+1)$. Any given point on the horizontal axis

maps to a set of points on the vertical axis, corresponding to the set of reachable speeds given any allowed disturbance.

The purpose of this figure is to visualize the stability of the controller. The diagonal dashed lines show the points where the Lyapunov function 4.14 is unchanged from one step to the next. The entire horizontal axis is the set of initial speeds Ω_I, and there is a thin vertical shaded region that shows the goal set Ω_G. For initial speeds that are outside of the goal set, the Lyapunov function is always decreasing from one step to the next. For initial speeds inside the goal set, the Lyapunov function (error squared) might increase, but the next step speed will never leave the goal set.

In the absence of disturbances the controller rejects nearly all speed error in a single step, as shown by the nearly horizontal line labeled *no disturbance*. In the presence of disturbances the controller is still quite stable, reaching the goal set in two or three steps even with the worst possible disturbances.

4.5.6 Simulation Test

As one final check, we simulated the closed loop system for a total of 10^6 steps, where the robot was subject to random disturbances, uniformly drawn from $\mathcal{D}$[4]. We ran 10^3 simulations, each consisting of 10^3 steps and starting from a randomly chosen initial condition uniformly drawn from Ω_I. In all cases the mid-stance speed stabilized to the target speed within a few steps, and the mid-stance speed remained within the goal set (Ω_G) on all subsequent steps. That is, despite the disturbances applied at each step, the controller was able to prevent falls in all cases.

[4]The perturbation on leg length was randomly drawn once at the beginning of each of the 10^3 simulations, and then held constant for all 10^3 steps in each simulation. This was done to more closely mimic a robot that had a modeling error. The other disturbances were randomly drawn on every single step.

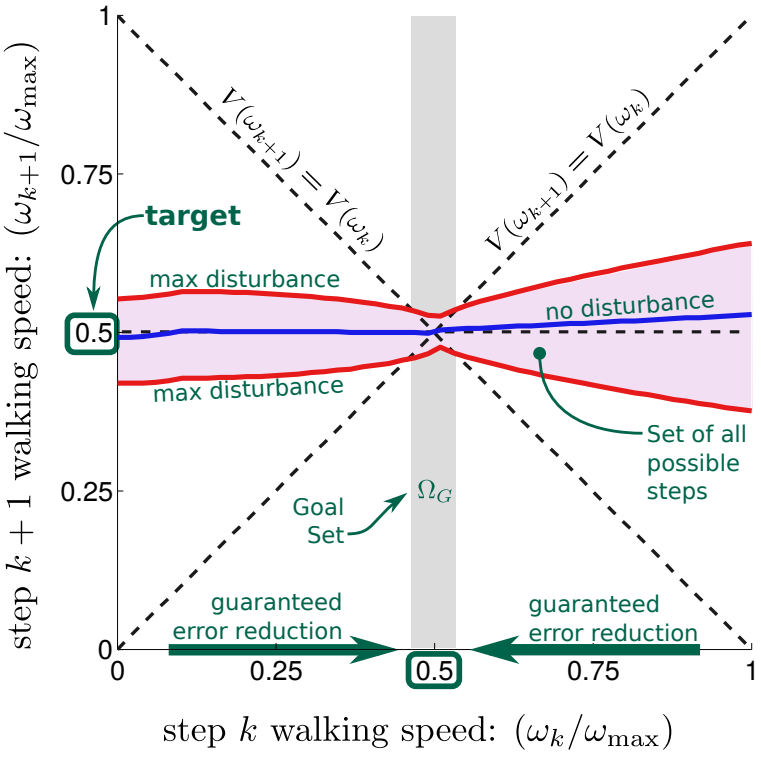

Figure 4.5: **One-step speed map.** This figure shows the closed-loop dynamics for the medium-speed ($\omega^* = 0.5\,\omega_{\max}$) walking controller. The horizontal axis gives the mid-stance speed for step k, and the vertical axis gives the mid-stance speed for step $k+1$ that is achieved by the robust controller. The horizontal shaded region show all possible steps that occur. The *no disturbance* line shows the behavior of the controller in the absence of disturbances. The boundary of the shaded region, marked with *max disturbance*, shows maximum possible speed error at the next step due to a disturbance. The dashed lines show the points where the speed error is unchanged from one step to the next. Notice that for most of the domain of the controller, there is a large reduction in error from one step to the next, despite disturbances. For example, if $\omega_k = 0.75\,\omega_{\max}$, then $\omega_{k+1} \in [0.4\,\omega_{\max}, 0.6\,\omega_{\max}]$. If there were no disturbances, then ω_{k+1} would be $0.51\,\omega_{\max}$. The vertical shaded region shows the goal set Ω_G that satisfies (4.17) and (4.18).

4.6 DISCUSSION

We have presented a controller for a simple model of walking that is maximally robust, by our measures. This controller can regulate a desired walking speed while preventing falls due to reasonable errors in the model, sensors, and actuation. The controller avoids falls for all reasonable values of disturbed and desired walking speeds. That is, assuming we consider only forwards walking, the basin of attraction of this controller is maximal with the given noise.

The approach here was inspired by robust control: the controller should be robust (never fail) for any bounded disturbance, and the walking speed should converge to some goal set. The controller was designed using non-linear optimization, where these robustness and stability requirements were enforced as constraints on the optimization. Although the resulting controller is correct by construction, we demonstrated the performance of the controller using massive simulation.

We designed a controller that could robustly stabilize slow, moderate, and fast walking gaits for the inverted pendulum model of walking, using parameters based on our robot, the Cornell Ranger. An interesting and perhaps general result is this: To increase speed, the controller takes large steps and uses large push-off. To maintain speed, it takes small steps and uses intermediate values of push-off. To decrease speed, the controller takes large steps with no push-off. This general trend is observed across all walking speeds. Although only implemented here on a simple 2D model, we believe the approach will be useful for a more complex robot in 3D.

4.7 FUTURE WORK

We plan to develop this controller for use on a 2D robot, Cornell Ranger, and then later, for a 3-D robot with many degrees of freedom.

One limitation of our model is that we restricted it to forward motion without a flight phase. These restrictions make calculations easier, but are overly conservative with respect to the real limits on falling; a small flight phase, or a step backwards does not necessarily lead to a fall. With a more general model that allowed some flight and back-stepping, we could make a robust controller with even larger allowed bounds on the various disturbances.

The extension from a 2-D robot to a 3-D robot will raise the mid stance state from 1 number to 2 (or 3 if heading is to be stabilized). The actuation will go from 2 controls to 3 (push off, step length and steering angle). However, the addition of more internal degrees of freedom does not change the form of this high-level controller. Rather the output of a controller of the type developed here, will serve as input to the micro-management of the various joints so as to achieve the desired push off and foot placement.

CHAPTER 5

ROBUST SWING-LEG CONTROLLER

This chapter was originally submitted as a final report for Cornell MAE 6950 – Formal Methods for Robotics. Author List: Matthew Kelly

5.1 Abstract

The control systems for modern bipedal walking robots do not use energy effectively. In order for such robots to be useful in the future, they will need to reduce their energy consumption. Here I propose an energy-effective policy for controlling the swing leg of a bipedal robot during the swing phase of walking.

5.2 Introduction

There are three predominant methods for control of modern bipedal robots: capture point [90], zero moment point [117] [55], and hybrid zero dynamics [121]. Each of these controllers is in some way hierarchical: there is a high-level planner for whole-body motion and foot placement, and a low level planner that coordinats individual joint motions. Typically such controllers use trajectory tracking on the full state of the robot, making heavy use of ankle motors.

I suggest a modification to this scheme, which allows the upper body of the biped to follow a nearly ballistic trajectory, while tracking control is only applied to the swing leg. Such a controller does not rely on ankle motors, which are unnecessary during

the swing phase of walking. Instead, it uses a trajectory library to compute an energy-optimal swing leg trajectory, that is updated as a functional of the upper body motion.

5.3 Problem Statement

> *"Design a feedback controller for leg swing that 1) minimizes energy use and 2) achieves the desired swing-leg state at the conclusion of the swing. Additionally, provide some guarantees about the maximum state error at the end of the step."*

5.3.1 Assumptions

Here I list assumptions that are required to make the problem statement well-defined. See 5.1 for a diagram of the walking model.

- Double pendulum model of walking (see Section 5.4).
- Stance leg angle is bounded and monotonic: $\theta_1 \in [-\phi, \phi], \quad \omega_1 > 0$.
- Swing leg angle, swing leg rate, and hip torque are bounded.
- A walking step starts with $\theta_1 = -\phi$ and ends when $\theta_1 = \phi$.
- The target swing leg state (θ_2, ω_2) is given for the end of the step.
- Motor power use is proportional to torque-squared (τ^2).

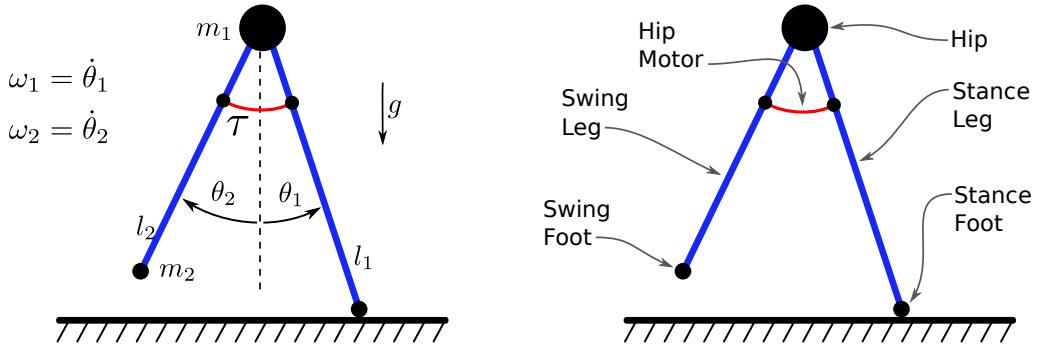

Figure 5.1: The double pendulum model of walking.

5.4 Model

Consider the double pendulum model of walking, as shown in 5.1. This model treats the upper body as a point mass centered on the hip. The legs are massless and inextensible, with a point-mass centered on each foot. There is a single actuator, which I will call the hip motor. It is modeled as a torque source (with saturation) connecting the swing leg to the stance leg.

Throughout the controller design process, I will switch between three versions of the double pendulum model of walking. The first model is a mathematical expression of the model described in the previous paragraph. The second is identical to the first, but with the dynamics expressed in terms of phase rather than time. The final model is the limiting case of the second model, where the swing foot mass is negligible ($m_2/m_1 \rightarrow 0$). The equations of motion for each model were derived using the Matlab Symbolic Toolbox.

5.4.1 Full Model

$$\dot{\mathbf{x}} = \mathbf{f}^*(\mathbf{x}, \tau, \mathbf{p}) \qquad \text{dynamics} \tag{5.1}$$

$$\mathbf{x} = [\theta_1, \theta_2, \omega_1, \omega_2]^T \qquad \text{state vector}$$

$$\tau \in [\tau_{\min}, \tau_{\max}] \qquad \text{input saturation} \tag{5.2}$$

$$\mathbf{p} = [m_1, m_2, g, l_1, l_2] \qquad \text{parameter vector}$$

5.4.2 Phase Model

Although system dynamics typically express a change in state with respect to time, it is also possible to express the change in state with respect to any monotonically increasing quantity, known as a phase variable. In this case, I use the stance leg angle as a phase variable, as shown in (5.3).

$$\mathbf{z}' = \mathbf{f}(\varphi, \mathbf{z}, \tau, \mathbf{p}) \qquad \text{phase-based dynamics} \tag{5.3}$$

$$\varphi = \theta_1 \qquad \text{phase variable: stance leg angle}$$

$$\mathbf{z} = [\theta_2, \omega_1, \omega_2]^T \qquad \text{phase-based state vector}$$

5.4.3 Reduced (Swing-Leg) Model

In most bipeds, the mass of the feet and lower legs are relatively small when compared to the mass of the upper body. By taking this limit ($m_2/m_1 \to 0$), it is possible to decouple the motion of the swing and stance leg. In particular, the stance leg dynamics reduce to

those of an inverted pendulum, and the swing leg dynamics are shown in (5.4).

$$\bar{\mathbf{z}}' = \bar{\mathbf{f}}(\varphi, \bar{\mathbf{z}}, \tau, \bar{\mathbf{p}}) \qquad \text{swing-leg dynamics} \tag{5.4}$$

$$\bar{\mathbf{z}} = [\theta_2, \omega_2]^T \qquad \text{reduced (swing-leg) state vector} \tag{5.5}$$

$$\bar{\mathbf{p}} = [m_1, m_2, g, l_1, l_2, e] \qquad \text{parameter vector}$$

Notice that there is a new element in the parameter vector: e, the *stance leg energy*. It turns out that in the limit as $m_2/m_1 \to 0$ the stance leg energy is a conserved quantity: $e' = 0$.

$$e = \frac{1}{2} m_1 l_2^2 \omega_1^2 + m_1 g l_1 \cos(\theta_1) \tag{5.6}$$

5.5 Controller Design

The overall controller design method is based on creating, and then stabilizing, a library of trajectories for the swing leg. The end result is a non-linear full-state feedback controller that stabilizes the system to an enrgy-optimal trajectory (5.16).

5.5.1 Trajectory Optimization

At the core of this controller is a library of energy-optimal trajectories for the reduced swing-leg model (5.4) of the system. These trajectories are generated using the trajectory optimization software GPOPS II [84], which is a Matlab-based transcription[1]method.

The trajectory library is a one-dimensional family of swing leg trajectories, parameterized by the stance leg energy (e). The independent variable in each of these trajec-

[1]Transcription is the process by which a trajectory optimization problem is converted into a sparse, non-linear, constrained optimization problem, which in this case is solved by SNOPT[91].

tories is the stance leg angle ($\theta_1 = \varphi$), which is treated as a phase variable (instead of time). The stance leg energy is bounded: $e_{\min} < e < e_{\max}$. If $e < e_{\min}$, then the biped will be unable to complete a forward step. If $e > e_{\max}$ then the biped will enter a flight phase of motion.

Consider a single trajectory optimization, for some given value for e. The target final state for the system is given in the problem description. Although the initial state need not be specified, I constrain it to be the nominal state at the start of the swing, to keep the trajectory library consistent.

The trajectory optimization method finds the optimal trajectory between the initial and target final state. In this case, the optimal trajectory is one which minimizes the energy use, given by the following equation.

$$\text{cost} = \int_{t(\varphi_0)}^{t(\varphi_f)} \tau(\varphi)^2 dt = \int_{\varphi_0}^{\varphi_f} \frac{\tau(\varphi)^2}{\omega_1(\varphi)} d\varphi \tag{5.7}$$

$$\omega_1(\varphi) = \frac{\delta\varphi}{\delta t(\varphi)} = \frac{1}{l_1}\sqrt{\frac{2}{m_1}}\sqrt{e - m_1 g l_1 \cos(\varphi)} \tag{5.8}$$

Each optimal trajectory is returned as a set of ordered points: $\{\varphi_0^*, \ldots \varphi_N^*, \bar{\mathbf{z}}_0^*, \ldots \bar{\mathbf{z}}_N^*, \tau_0^*, \ldots \tau_N^*\}$, and then non-linear least squares to fit a polynomial to the trajectory. The trajectory can then be evaluated using Barycentric Interpolation [5] as a function of phase: $\{\bar{\mathbf{z}}^*(\varphi), \tau^*(\varphi)\}$. Once a set of trajectories has been generated, it is possible to use the same interpolation method along the stance leg energy dimension to evaluate the library of trajectories: $\{\bar{\mathbf{z}}^*(\varphi, e), \tau^*(\varphi, e)\}$.

5.5.2 Phase-Varying LQR

Given a single value for the stance leg energy, the trajectory library gives the optimal swing leg trajectory and hip torque as a function of φ. The next step in the design

process is to use phase-varying linear quadratic regulator (LQR) to stabilize this optimal reference trajectory.

Aside: Note that the LQR controller is designed using the reduced model (5.4) of the system. In this model, there is no coupling between the actuation (hip torque) and phase (stance leg angle), which is important for stability of the LQR controller. When implemented on the full system, the trajectory library stabilizes this coupling term by adjusting the reference trajectory. Other researchers have accomplished this decoupling by a method known as transverse linearization [102].

In order to solve the standard phase-varying LQR problem [109], I express the dynamics in phase varying linear form, where $\{\bar{\mathbf{z}}_0(\varphi), \tau_0(\varphi)\}$ is an optimal reference trajectory.

$$\hat{\mathbf{z}}(\varphi)' \approx \mathbf{A}(\varphi) \cdot \hat{\mathbf{z}}(\varphi) + \mathbf{B}(\varphi) \cdot \hat{\tau}(\varphi) \tag{5.9}$$

$$\hat{\mathbf{z}}(\varphi) = \bar{\mathbf{z}}(\varphi) - \bar{\mathbf{z}}_0(\varphi) \qquad \hat{\tau} = \tau(\varphi) - \tau_0(\varphi) \tag{5.10}$$

$$\mathbf{A}(\varphi) = \left.\frac{\delta \bar{\mathbf{f}}}{\delta \bar{\mathbf{z}}}\right|_{\bar{\mathbf{z}}_0(\varphi),\tau_0(\varphi)} \qquad \mathbf{B}(\varphi) = \left.\frac{\delta \bar{\mathbf{f}}}{\delta \tau}\right|_{\bar{\mathbf{z}}_0(\varphi),\tau_0(\varphi)} \tag{5.11}$$

The LQR controller stabilizes the the system by following trajectories that minimize a quadratic cost function: (5.12). The behavior of the controller can be adjusted using the three cost matricies: $\mathbf{F}$, $\mathbf{Q}$, and R.

$$J = \hat{\mathbf{z}}(\varphi_f)^T \mathbf{F} \hat{\mathbf{z}}(\varphi_f) + \int_{\varphi_0}^{\varphi_f} \left(\hat{\mathbf{z}}^T(\varphi) \mathbf{Q} \hat{\mathbf{z}}(\varphi) + \hat{\tau}^T(\varphi) R \hat{\tau}(\varphi) \right) d\varphi \tag{5.12}$$

The optimal control is given by:

$$\hat{\tau}(\varphi) = -R^{-1} \mathbf{B}^T(\varphi) \mathbf{S}(\varphi) \hat{\mathbf{z}}(\varphi) = -\mathbf{K}(\varphi) \hat{\mathbf{z}}(\varphi) \tag{5.13}$$

where $\mathbf{S}(\varphi)$ is the solution to (5.14), which in this case was solved using 4^{th}-order Runge-Kutta integration.

$$-\dot{\mathbf{S}}(\varphi) = \mathbf{Q} - \mathbf{S}(\varphi)\mathbf{B}(\varphi)\mathbf{R}^{-1}\mathbf{B}^T(\varphi)\mathbf{S}(\varphi) + \mathbf{S}(\varphi)\mathbf{A}(\varphi) + \mathbf{A}^T(\varphi)\mathbf{S}(\varphi), \qquad \mathbf{S}(\varphi_f) = \mathbf{F} \tag{5.14}$$

The result is a state feedback law along the optimal reference trajectory:

$$\tau(\varphi, \bar{\mathbf{z}}) = \tau_0(\varphi) - \mathbf{K}(\varphi) \cdot (\bar{\mathbf{z}} - \bar{\mathbf{z}}_0(\varphi)) \tag{5.15}$$

Each element in the gain matrix $\mathbf{K}(\varphi)$ is expressed as a polynomial in phase, again using non-linear least squares. This phase-varying gain matrix is stored along with each optimal reference trajectory in the trajectory library. Now, the trajectory library returns the nominal torque $\tau^*(\varphi, e)$, state $\bar{\mathbf{z}}^*(\varphi, e)$, and gain matrix $\mathbf{K}^*(\varphi, e)$ for an arbitrary phase (φ) and stance leg energy (e) using interpolation. The result is a full state feedback control law (5.16) for the full system (5.1). Recall that there is a direct mapping from $\mathbf{x}$ to $\{\varphi, e, \bar{\mathbf{z}}\}$, as discussed in Section 5.4.

$$\tau(\mathbf{x}) = \tau^*(\varphi, e) - \mathbf{K}^*(\varphi, e) \cdot (\bar{\mathbf{z}} - \bar{\mathbf{z}}^*(\varphi, e)) \tag{5.16}$$

5.5.3 Function Approximation and Interpolation

Function approximation and interpolation are critical to the success of a controller that relies on a trajectory library. There are many possible function approximation and interpolation schemes; I have chosen to use Chebyshev Polynomials and Barycentric Interpolation [5], which have a variety of nice numerical properties if the underlying functions are smooth, which is a good assumption for this library.

Barycentric Interpolation of a Chebyshev Polynomial is a fast, accurate, and numerically stable way to evaluate polynomials of an arbitrarily high order. It works by computing a weighted average of the function's value at the a special set of points, called Chebyshev Nodes, spread over the domain of the function. The Chebyshev Nodes are spaced more closely near the boundries of the domain, and have a variety of special properties with regard to function approximation[5].

5.6 Controller Analysis

5.6.1 Stability

In the general case, for a linear system with arbitrarily powerful actuators, a LQR controller will be globally stable. This is not the case for this system, which is a phase-varying linearization of a non-linear system with bounds on the actuator. It is not immediately obvious what stability means for a non-linear system, particularily one like this that has no fixed point. For this purpose I will adapt a metric that is used in computing the stability of non-linear periodic systems, known as the contraction ratio. It is a ratio of how some initial perturbation maps to some final perturbation, taken at specific points in time or phase (often called Poincaré Sections). For the purpose of this paper, I define r as shown in (5.17), where the subscript i indicates the initial phase, and subscript f indicates the final phase, in this case the end of the step ($\varphi_f = \phi$).

$$r_{i \to f} = \frac{\| \bar{\mathbf{z}}(\varphi_i) - \bar{\mathbf{z}}^*(\varphi_i) \|^2}{\| \bar{\mathbf{z}}(\varphi_f) - \bar{\mathbf{z}}^*(\varphi_f) \|^2} \tag{5.17}$$

I compute the contraction ratio over a grid of possible perturbations in swing leg state, and then repeat this procedure for several values of φ_i. The result shows which perturbations the controller can reject, along the entire length of the trajectory. If the contraction ratio is zero, the the perturbation is completely rejected, and if it is one, then the perturbation's magnitude is unchanged. Contraction ratios larger than one indicate that the controller is not stable for that initial perturbation.

5.6.2 Reachability

After checking the stability of the controller, I perform is a reachability analysis, using a front-propagation method. The initial front is an ellipse, centered on the nominal initial condition, and whos axis are scaled to match the maximum expected deviation in initial swing-leg state. The controller is stable for all points on the ellipse, as defined in Section 5.6.1. That initial ellipse is the propagated through the dynamics until the end of the step. The results produced are very similar to those obtained using alternate methods, such as the Level-Set Toolbox [75].

In addition to doing a reachability analysis on a single trajectory from the trajectory library, I did a similar analysis for the full-state feedback controller (5.16), using the fully coupled system dynamics (5.1). In this case, the initial front was an ellipse and the swing leg state, for each of several stance leg energies, all starting from the initial stance leg angle.

5.7 Results

I used the methods described in this paper to design a swing-leg controller for a simple model of walking. It is capable of stabilizing a large set of initial states to the desired target state. Each entry in the trajectory library (optimal trajectory and stabilizing controller) is generated in about one second using a regular laptop. The remainder of this section is a collection of figures, where the detailed discussion of each figure can be found in its caption. In all cases, we use the non-linear dynamics for evaluating the performance of the controller.

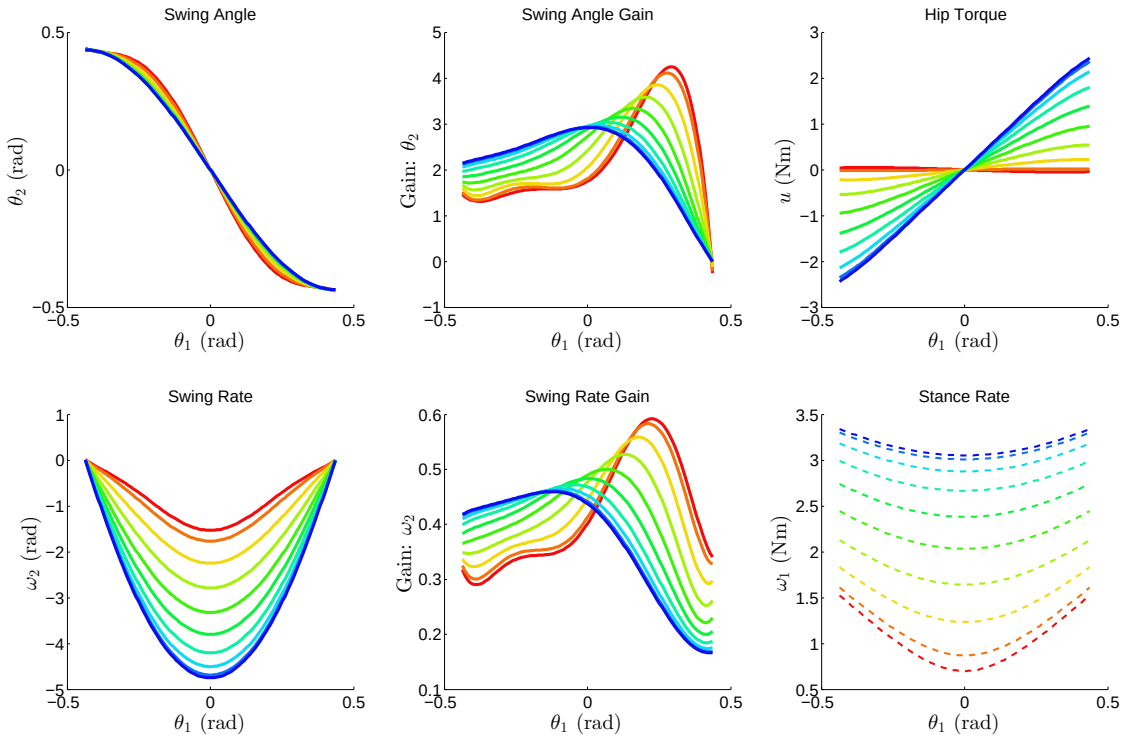

Figure 5.2: Trajectory library. Each color shows the optimal trajectory for a different stance leg energy. Notice that the shape of the curves varies smoothly with the stance leg energy parameter, which is important for the polynomial interpolation.

- 5.2: Trajectory library
- 5.3: Stability of controller for large peturbations
- 5.4: Reachable set for a single trajectory
- 5.5: Reachable set for full model

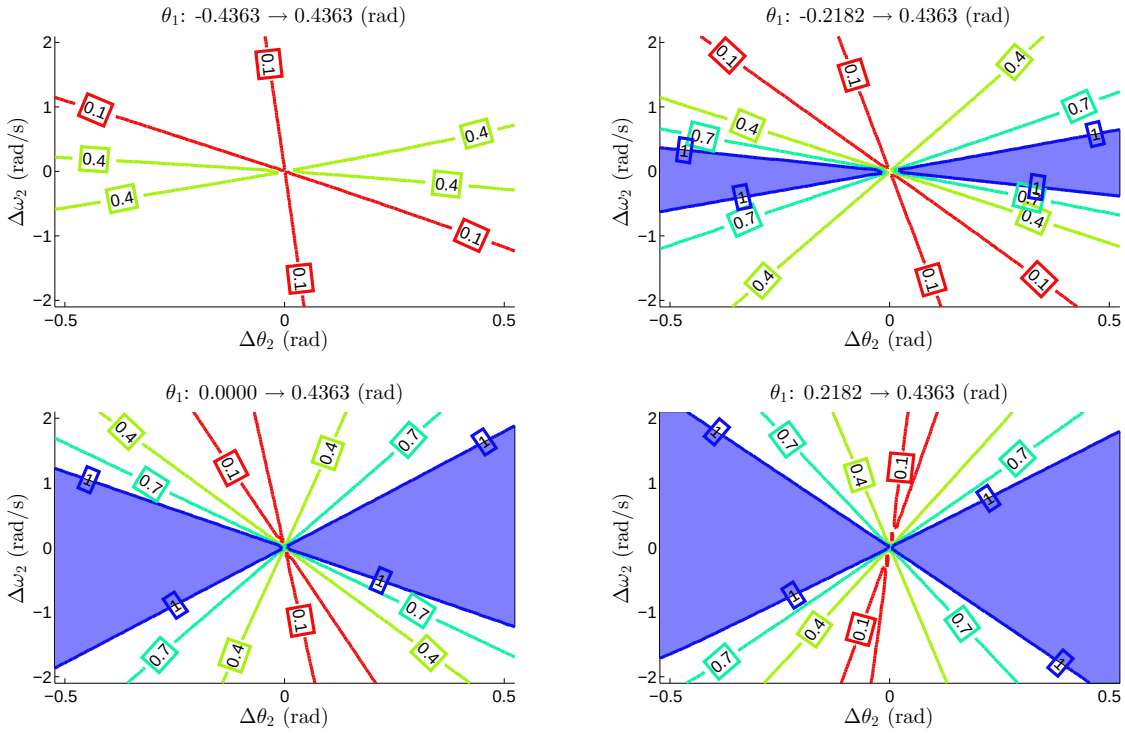

Figure 5.3: This plot shows how well the controller can stabilize different initial conditions. The state space if four dimensional, so the visualization is a bit tricky here. First, we only show the results for a single energy leg energy level: $e = e_{\min} + 0.25 \cdot (e_{\max} - e_{\min})$. Next, we show four different level sets, for different angle of the stance leg (-0.43, -2.2, 0.0, 2.2). Each of the four resulting plots shows the contraction ratio for each swing leg angle and rate. If the contraction ratio for a point is less than one, then that point can be stabilized by the controller. The blue regions here show the sets of points that are not able to be stabilized. Notice all points in the upper left plot can be stabilized, while only about half in the lower right plot can be stabilized. This is related to the amount of time that the controller has to stabilize the system. In the upper left plot it has the entire step, while in the lower right plot it only has a small fraction of a step.

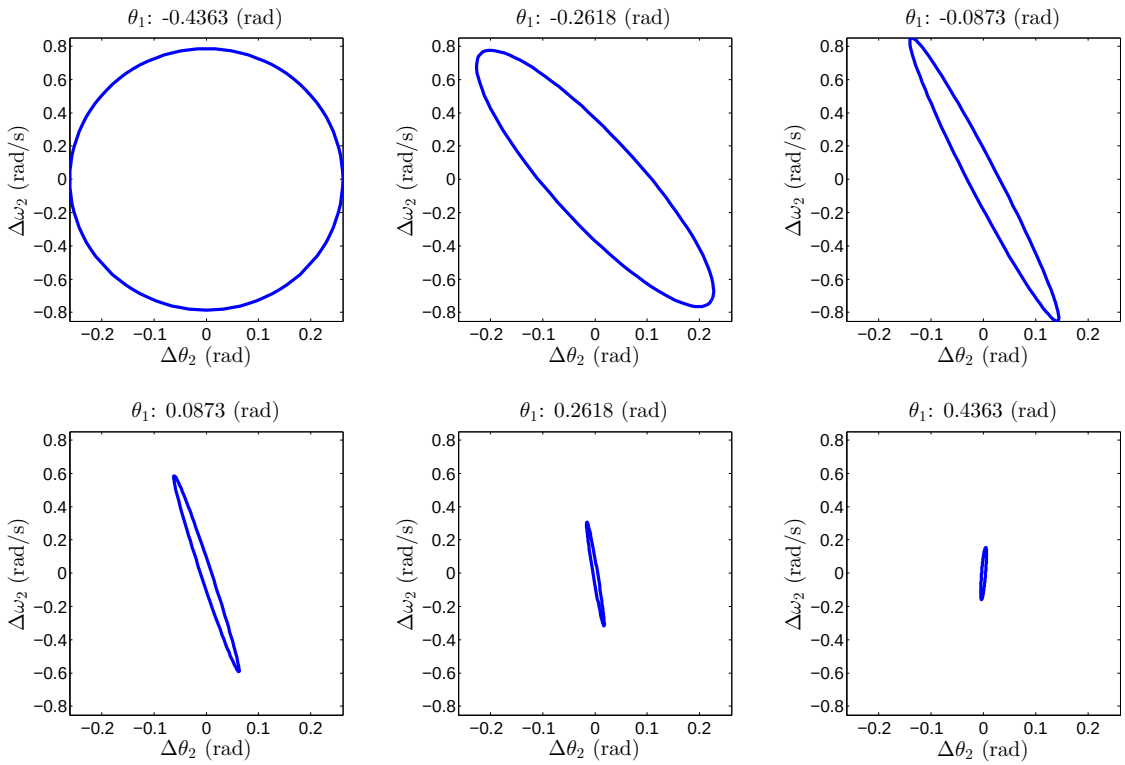

Figure 5.4: The reachable set for an ellipse of initial states. Both axis show perturbations from the nominal trajectory, and each successive plot shows the set contracting over the course of the step. Note that the final ellipse is thin the the $\Delta\theta_2$ direction and longer in the $\Delta\omega_2$ direction; this is because the LQR controller gains are set to have a large cost for error in the final swing leg angle, and a small cost for errors in the swing leg rate. If the LQR cost on actuation effort goes down relative to the cost on state errors, then the ellipse for the end of the step can be made arbitrarily small. Stance leg energy: $e = e_{\text{min}} + 0.25 \cdot (e_{\text{max}} - e_{\text{min}})$. This figure is computed using the non-linear dynamics model.

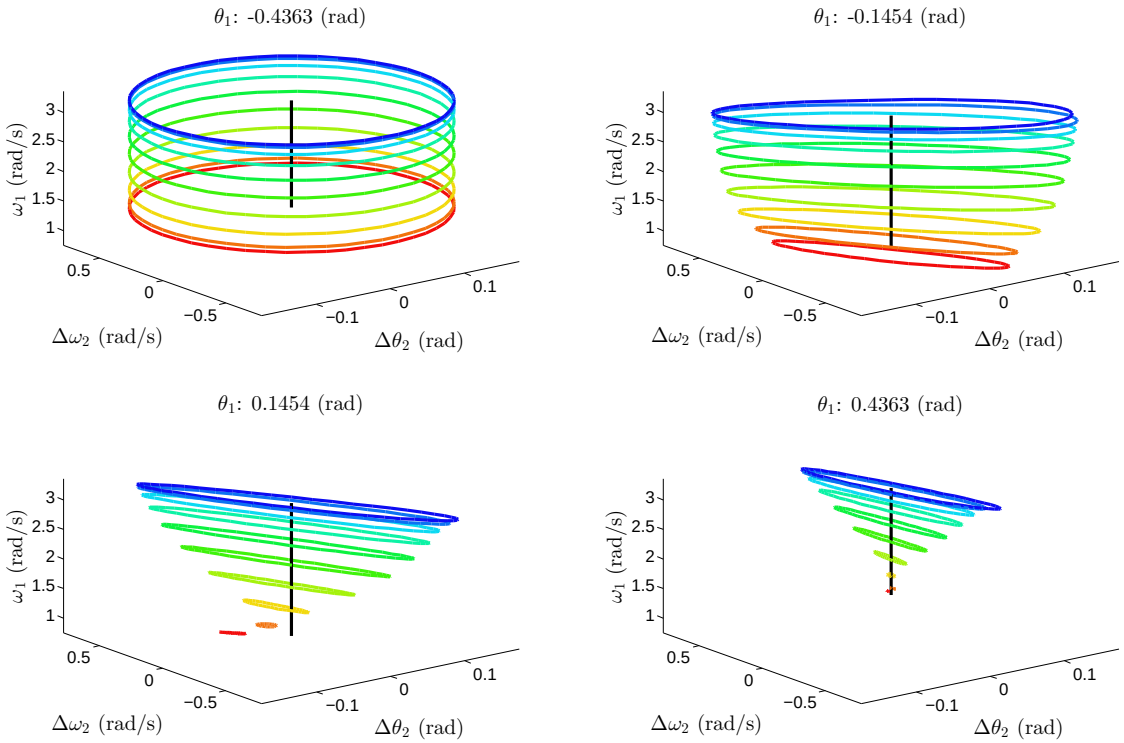

Figure 5.5: This plot shows reachable set for the system, starting from a few specific energies, for the same set of swing leg perturbations. Each energy is shown as a different color, where red corresponds to low energy, and blue corresponds to high energy. This reachability calculation uses the full model (double pendulum, with coupling). The coupling terms can be seen in the plot, as each ring is slightly distorted out of its initial plane. The controller is able to stabilize all initial conditions inside of this set. Notice that the controller cannot stabilize the points corresponding to faster walking as well as slow, because the bounded torque can only accelerate the swing leg so quickly. The ellipse has a preferred direction, as a result of the LQR gains. The shape of the final ellipses is strongly related to the choice of LQR cost matricies. The controller in this figure has large penalties on actuation and swing leg angle error, but a small penalty on the swing leg rate.

5.8 Conclusion and Future Work

In this paper I have presented an energy effective swing-leg controller, and demonstrated that it is able to stabilize a large region of state space around a normal walking gait. Although this controller is designed for a single target step length, it is possible to extend the work here to include an arbitrary target step length, as an additional dimension in the trajectory library.

The analysis in this paper assumes that the system is purely deterministic, which is not true in the 'real' system. The next step in analysis would be to generate a noise model for the system, and then show how the controller is able to reject stochastic disturbances.

The model for motor power consumption that is used here is very simple. The work can easily be extended to use a more accurate model of the power consumed by the motor.

CHAPTER 6

SIMULATIONS WITH SLIDING AND INTERMITTENT CONTACT

This chapter was written in preparation for my thesis proposal exam in 2014.

Author List: Matthew Kelly.

6.1 Abstract

Systems that have sliding and intermittent contact are said to have hybrid dynamics. These systems have continuous modes (sliding, rotating, flight...) that are connected with discrete transitions (collision, lift-off...). Simulation of hybrid systems can be tricky, and this report will cover a few subtitles that are encountered when simulating hybrid systems that are associated with the contact mechanics of rigid bodies.

6.2 Simulation Architecture

When simulating mechanical hybrid systems, there are two fundamentally different architectures for the simulator. We will call the first *event-based* simulation and the second *impulse-based* simulation. Event-based simulation is the traditional way to simulate a hybrid system, and it directly uses the finite state machine structure of the hybrid system. Impulse-based simulation is an approximation that is used for systems where the full finite state machine is too complex to explicitly write out. The simulations in this report exclusively use event-based simulation, since their finite state machines are simple and I'm interested in high-accuracy results.

6.2.1 Event-Based Simulation

In an event-based simulation a full finite state machine must be written out for the system dynamics. In each continuous state a set of differential equations governs the continuous dynamics of the system. The integration algorithm is typically a variable-step integrator that uses root finding to determine the precise instant in time when the state invariants or transition guards switch. This results in an accurate simulation, since both the continuous and discrete dynamics can be calculated to near machine precision.

The major down-side of this type of simulation is that for some mechanical systems, the full finite state machine is too complicated to write down. For example, suppose that you are trying to simulate a system of 10 dice being thrown. Each die can contact the ground at either one, two, or four points[1], and then every die and touch a few other dice as well, in arbitrary locations.

6.2.2 Impulse-Based Simulation

Impulse-based simulation was developed to deal with systems with arbitrarily complex contact mechanics, such as the dice example from the previous section. It is used in most general dynamics engines, such as Bullet and Open Dynamics Engine (ODE). Some people also call impulse-based simulation *time-stepping* simulation.

The key idea in impulse-based simulation is that each rigid body[2] is handled independently. The dynamics are written out such that all forces are expressed as impulses, given some fixed time-step. At every time step, the impulsive contact forces between

[1] four contact points causes difficulties even for a single die in isolation...

[2] Some algorithms have special algorithms for dealing with linkages, such as a "Featherstone's Algorithm" which is actually a collection of algorithms, published by Roy Featherstone in his book *Rigid Body Dynamics Algorithms* [27].

objects are solved as part of a large optimization program. This optimization problems solves for the unique solution to these three constraints, applied at every contact pair that is detected[3]. These constraints form a linear complementarity problem (LCP).

$d_n > 0$	Contact seperation
$J_t \leq \mu J_n$	Contact force in friction cone
$d_n J_t = 0$	Contact force when touching

6.3 Bouncing Ball

One of the simplest hybrid systems is a bouncing ball. There is one continuous phase of motion: (flight through the air)

$$\dot{x} = v \tag{6.1}$$

$$\dot{v} = -g \tag{6.2}$$

and one discrete transition: (collision with the ground)

$$v^+ = -k \cdot v^- \tag{6.3}$$

Simulation of this system will show that the ball simply bounces up and down, attaining less height on each successive bounce (assuming a realistic value of $0 < k < 1$).

6.3.1 Zeno's Paradox

It seems that simulating such a simple system would be trivial, but it turns out that you will quickly discover a problem. The problem is that the simple model of a bouncing

[3]There is a whole field of research dedicated to determining which points of which objects are in contact, but it is outside the scope of this report.

ball will experience an infinite number of bounces in a finite amount of time. This behavior is known as Zeno's paradox, and it will typically break the simulator, if not handled properly.

Let's look at why this happens. The analytic solution for bounce time, for the ball bouncing equations given above is:

$$T_i = 2\frac{v_i}{g} = 2\frac{(k^i v_0)}{g} \tag{6.4}$$

Since this is a geometric series, it can readily be shown that the time required for an infinite number of bounces is given by Equation 6.5. Assuming that the coefficient of restitution (k) is strictly less than one, an infinite number of bounces will occur in a finite time. As a computer attempts to run a simulation up to and through this critical time, it will necessarily run into an error (such as the integration time step going to machine precision, or the event detection missing the collision, or just missing a bounce entirely).

$$\sum_{i=0}^{\infty} T_i = \frac{2v_0}{g}\sum_{i=0}^{\infty} k^i = \left(\frac{2v_0}{g}\right)\left(\frac{k}{1-k}\right) \tag{6.5}$$

This particular issue of bouncing contact is common in simulations, and most simulation engines treat it as a special case. One common way to do this is to prescribe a minimum escape velocity; if the separation velocity is smaller than this threshold, then the simulator will assume the the objects stay in contact.

6.3.2 When event detection fails

A bouncing ball on flat ground is not very exciting, but things get more interesting when it is bouncing around on hilly terrain. Since the dynamics are still fairly simple, this is an excellent place to use a variable-step integration algorithm with event detection (as opposed to an impulse-based simulator).

Event detection works by evaluating an event vector at every grid point, and checking the sign of the result. If there is a sign change, then an event occurs and the simulator calls a root finding routine. Interestingly, if the dynamics are simple, but the event function is complicated, this scheme will miss events. This is because the variable-step integrator (Matlab's ode45 in the case of Figure 6.3.2) is trying to maximize step size while meeting an accuracy constraint. Since the dynamics are simple, it takes huge time steps, which can miss rapid changes in the event function.

In the case of the ball bouncing over hilly terrain, the height of the ball with respect to the ground (the event function) is changing much more rapidly than the absolute height of the ball (the state dynamics). This allows the ball to tunnel through a steep hill, as shown in Figure 6.3.2. One way to solve this problem is to put a limit on the maximum step size of the integrator, such that it is unlikely to miss a collision.

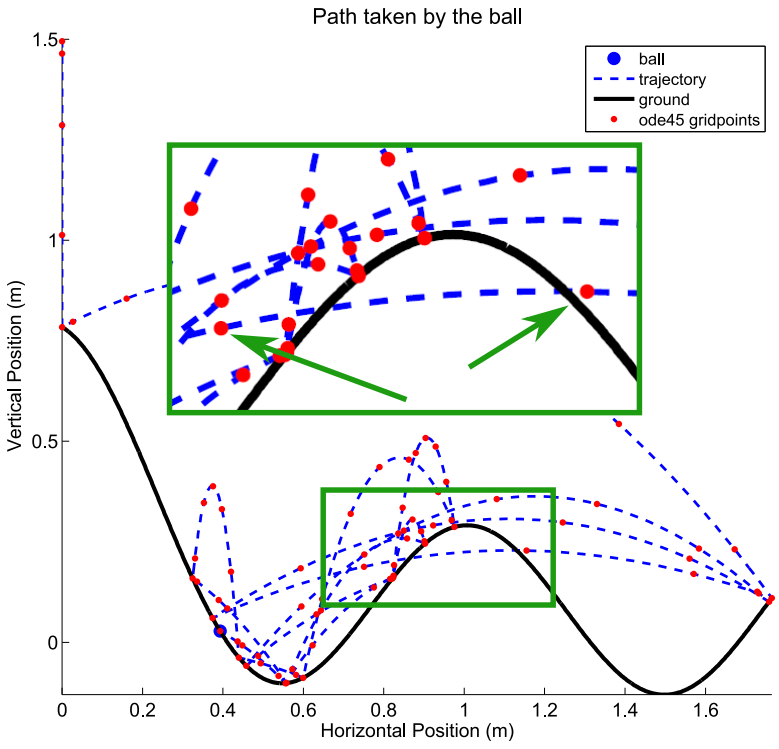

Figure 6.1: A bouncing ball over hilly terrain. This figure illustrates a special case where matlab's built in event detection fails due to a large step-size in ode45.

6.4 Sliding is not optional

In many simple simulations, it is tempting to assume a 'no-slip' boundary condition. Under this (bad) assumption, the contact point can either be pinned or free, and the transitions between modes are based on the normal force and separation distance. A simulation using the no-slip boundary condition will either use negative contact forces or allow the colliding object to penetrate, depending on the type of error. The solution to this problem is to assume some sort of contact model that allows for sliding.

6.4.1 The Toppling Pencil

In the 1980's, Tad McGeer published a paper [73] which showed that a simulation of a pencil toppling from rest produce non-physical results if you make this 'no-slip' assumption. His simulations showed the pencil slowly falling over, and then at some critical angle, the pencil would begin to accelerate through the floor. Usually one would assume that this was a bug in the code, but in this case the code was correct, and the problem was due to the no-slip assumption.

As the pencil topples from rest, there is some critical angle where the normal force goes to zero. In general, the tangential force on the contact is non-zero at this point, and in reality the pencil would slide in response to this force. Since McGeer neglected sliding, his simulator set the contact mode to free. Instead of the contact point remaining above the ground, it instead began to accelerate through the ground. Why?

Let's consider the instantaneous acceleration of the contact point when the pin-constraint is removed. Before the constraint is removed, the contact point is stationary, but has a non-zero external force applied to it. This implies that there are some inertial

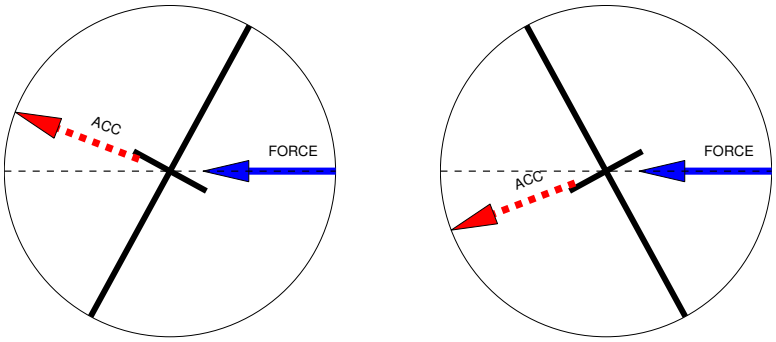

Figure 6.2: Acceleration of the contact point of a thin rod. The solid blue arrow shows an applied force, and the red dashed arrow shows the resulting acceleration.

forces that are attempting to accelerate this point, and that these forces precisely balance the external force. When the constraint is removed, so are these external forces, and thus the contact point accelerates due to these inertial forces.

One way to understand which direction the contact point accelerates is by understanding the mass matrix of the contact point. This matrix describes how a point on a rigid body moves in response to a force applied at that point. In general, the direction of acceleration is not aligned with the direction of the force. Figure 6.2 shows a visualization of the mass matrix, where the length of the dark line shows the effective mass in that direction. The left side of the figure shows the case observed in the toppling pencil example: The change in contact forces is directed to the right, and the pencil is tipped to the right. The resulting inertial effects act to swing the contact point into the ground.

6.4.2 No-Slip ≠ Infinite Friction

One reason that people give for making the 'no-slip' assumption is that it is "the limiting case as the coefficient of friction goes to infinity". This is completely false, as is demonstrated by the following example.

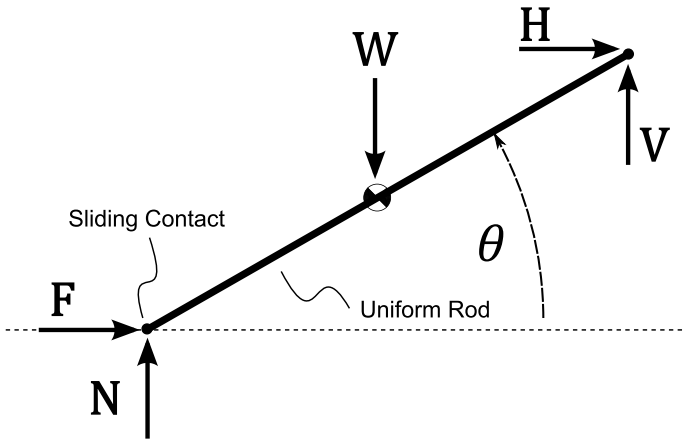

Figure 6.3: A stick that is being dragged along the ground with constant velocity.

Consider a simple example of sliding: a stick being dragged with constant velocity as shown in Figure 6.3. Assume that the contact forces are governed by Coulomb friction with coefficient μ. Since the velocity of the stick is constant, we can treat this as a statics problem. It can be shown that horizontal force (H) required to maintain constant speed is given by Equation 6.6

$$H(\theta) = \frac{W\mu\cos(\theta)}{2\cos(\theta) + \mu\sin(\theta)} \tag{6.6}$$

Taking the limit as $\mu \to \infty$ shows that the horizontal force remains finite. It can also be shown that the normal force (N) at the contact goes to zero as $\mu \to \infty$. These trends are shown numerically in Figure 6.4

$$\lim_{\mu\to\infty} H(\theta) = \frac{W}{\tan(\theta)} \tag{6.7}$$

6.4.3 Contact Finite State Machine

The 'correct' way to deal with contacts is to allow for three different contact modes at each contact: free, sliding, and pinned. Each of these contact modes can be expressed

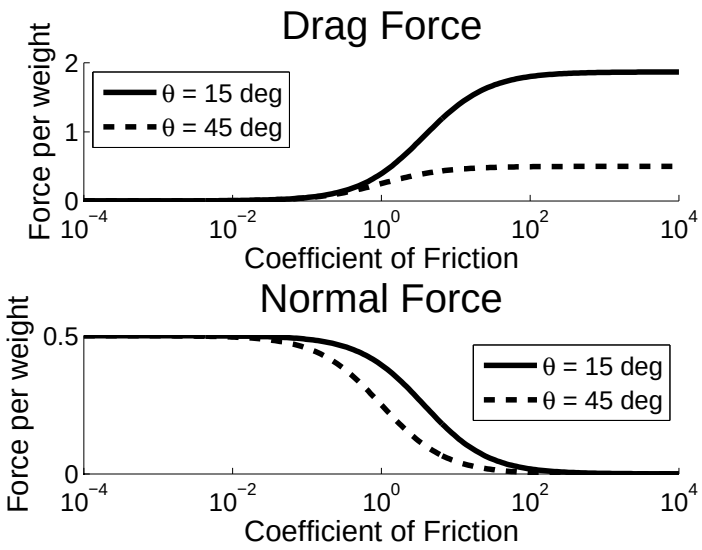

Figure 6.4: Contact force limits as a function of μ for a stick dragging with constant velocity.

as a constraint at that contact point:

- **free -** The normal force is zero
- **sliding -** The tangential force is such that the contact point travels along the surface
- **pinned -** The tangential force cannot do positive work on the system.

In addition to these constraints, there are also a set of transition conditions that link each of these continuous modes. Figure 6.5 shows an example of a finite state machine constructed from these states and transitions. Notice that each state has different equations of motion. This is fine for a single contact, but things get very complicated if you have multiple contacts. Consider robot model with two contact points - now the finite state machine has nine states instead of three.

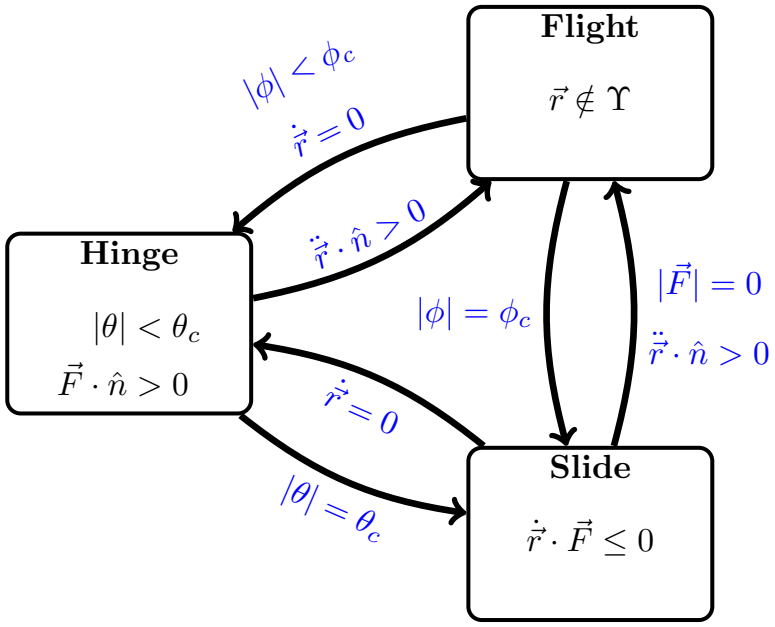

Figure 6.5: Finite State Machine for generalized rigid body contact

6.5 Toppling Stick

This first part of this section is an overview of how to create a finite-state-machine-based simulator for a two-dimensional rigid stick. The remainder of the section discusses some simulation results that investigate the conditions necessary for slipping as the pencil topples from rest.

6.5.1 Peinlevé Paradox

Peinlevé's Paradox occurs when a rigid body is sliding along a surface while tipped backwards. Under these conditions it is possible for a jamming motion to occur where the contact forces cause the body to rotate upwards, which in turn increases the contact force. Under these special conditions the contact forces and accelerations to become infinite in a finite amount of time. This phenomena is well studied and is thoroughly

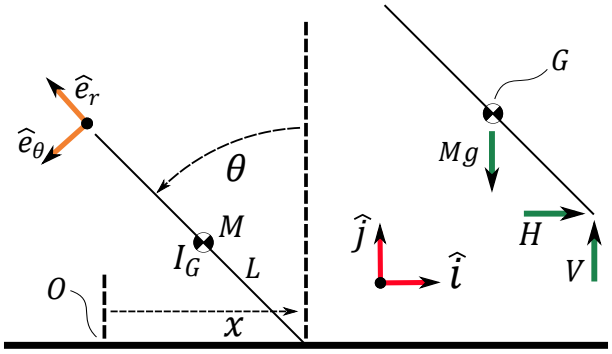

Figure 6.6: Free body diagram for both phases of motion.

discussed in references: [81] [39] [67]. It is this effect that allows dashed lines to be easily and rapidly drawn on a chalk board. I mention it here as an interesting side note, as remainder of the section are not related to this paradox.

6.5.2 Model for Continuous Dynamics

This section discusses the general model that is shared across all contact modes. The hinge and slide modes will be discussed in the following sections. The stick is modeled as a rigid body with:

M = mass

I_G = moment of inertia about center of mass (CoM)

L = distance between contact point and CoM

Forces acting on the stick:

Mg = body force due to gravity

H = horizontal contact force

V = vertical contact force

The stick has two degrees of freedom (neglecting flight phase):

θ = angle of the stick

x = horizontal position of contact point

The equations of motion are derived using two frames of reference - an inertial frame and a body frame:

$\hat{i}$ = positive horizontal direction

$\hat{j}$ = positive vertical direction

$\hat{k} \equiv \hat{i} \times \hat{j}$

$\hat{e}_r$ = direction along stick, pointing away from contact

$\hat{e}_\theta$ = direction transverse to $\hat{e}_r$

Define rotating reference frame and derivatives:

$$\begin{cases} \hat{e}_r &= -\sin(\theta)\hat{i} + \cos(\theta)\hat{j} \\ \hat{e}_\theta &= -\cos(\theta)\hat{i} - \sin(\theta)\hat{j} \end{cases}$$

$$\begin{cases} \dot{\hat{e}}_r &= \dot{\theta}\hat{e}_\theta \\ \dot{\hat{e}}_\theta &= -\dot{\theta}\hat{e}_r \end{cases}$$

$$\begin{cases} \ddot{\hat{e}}_r &= \ddot{\theta}\hat{e}_\theta - \dot{\theta}^2\hat{e}_r \\ \ddot{\hat{e}}_\theta &= -\ddot{\theta}\hat{e}_r - \dot{\theta}^2\hat{e}_\theta \end{cases}$$

6.5.3 Hinge Phase Derivation

In the Hinge phase of motion the contact point is fixed at the origin. The system is constrained to rotate about this point, thus giving the system one degree of freedom. The following constraints are used to determine when this phase is no longer valid. They correspond to transitions to falling, flight, and sliding respectively.

$$-\pi/2 < \theta < \pi/2 \quad \text{stick above ground}$$

$$V \geq 0 \quad \text{positive normal contact force}$$

$$-\mu V \leq H \leq \mu V \quad \text{coulomb friction}$$

Define position vector from the contact point to the CoM:

$$\vec{r} = L\hat{e}_r$$

$$\dot{\vec{r}} = L\dot{\hat{e}}_r$$

$$\ddot{\vec{r}} = L\ddot{\hat{e}}_r$$

Conservation of angular momentum (rate) about the contact point:

$$\sum M_{/o} = \dot{\vec{H}}_{/o}$$

$$\vec{r} \times (-Mg\hat{j}) = I_G\ddot{\theta}\hat{k} + \vec{r} \times (M\ddot{\vec{r}})$$

Conservation of linear momentum (rate):

$$\sum F = \dot{\vec{L}}$$

$$H\hat{i} + V\hat{j} - Mg\hat{j} = M\ddot{\vec{r}}$$

I used Matlab's symbolic toolbox to directly solve these equations for the angular acceleration and contact forces. These symbolic expressions, along with expressions for the kinematics and energy, were then automatically written to functions to be used by the simulation.

6.5.4 Slide Phase Derivation

In the Slide phase of motion, the contact points is constrained to move along horizontal ($\hat{i}$) axis and the contact forces obey coulomb friction. The following constraints are monitored during simulation. A transition is triggered when they are no longer satisfied.

$$\begin{aligned} -\pi/2 < \theta < \pi/2 \quad & \text{stick above the ground} \\ V \geq 0 \quad & \text{positive normal contact force} \\ |\dot{x}| \leq 0 \quad & \text{direction cannot change} \\ |H| = \mu V \quad & \text{Coulomb friction} \end{aligned}$$

Define position vector from the origin to the CoM:

$$\vec{r} = x\hat{i} + L\hat{e}_r$$

$$\dot{\vec{r}} = \dot{x}\hat{i} + L\dot{\hat{e}}_r$$

$$\ddot{\vec{r}} = \ddot{x}\hat{i} + L\ddot{\hat{e}}_r$$

Conservation of angular momentum (rate) about the origin:

$$\sum M_{/o} = \dot{\vec{H}}_{/o}$$

$$\vec{r} \times (-Mg\hat{j}) + (x\hat{i}) \times (V\hat{j}) = I_G \ddot{\theta}\hat{k} + \vec{r} \times (M\ddot{\vec{r}})$$

Conservation of linear momentum (rate):

$$\sum F = \dot{\vec{L}}$$

$$H\hat{i} + V\hat{j} - Mg\hat{j} = M\ddot{\vec{r}}$$

I used Matlab's symbolic toolbox to directly solve these equations for the angular acceleration and contact forces. These symbolic expressions, along with expressions for the kinematics and energy, were then automatically written to functions to be used by the simulation.

6.5.5 Simulation

I wrote a simulation in matlab that was based on a finite state machine (FSM) architecture. Each continuous phase of motion was integrated using ODE45 in Matlab, with built-in event detection to check that the constraints were satisfied. At the start of the simulation, it determines which phase of motion the system is in, and then integrates the corresponding equations of motion until a constraint is violated. At this point, the system runs the FSM to determine which phase of motion to transition to. Flight phase was included for completeness, but is never used in these experiments.

6.5.6 Experiment - Pencil Toppling from Rest

The goal of the experiment is to better understand what happens when the contact forces on an object go to zero. In this case, every experiment starts with a stick toppling from rest. The coefficient of friction at the contact and moment of inertia of the stick a both adjusted over a wide range of values. For every trial the slip distance and critical angle before slipping are recorded.

The parameters in the problem were scaled to be dimensionless: $\{M = 1, g = 1, L = 1\}$. Additionally, the moment of inertia (I_G) is scaled such that $I_G = 0$ corresponds to a point mass at the CoM and $I_G = 1$ corresponds to half the mass at each end of a stick with length = $2L$. As a point of reference, $I_G = 1/3$ corresponds to a slender rod.

In order for the stick to fall over in simulation, it must be given an initial perturbation. To keep results consistent, the initial energy of the system is held constant across all trials. This prescribes a fixed relationship between the initial angle and rate:

$$E(\theta, \omega) \equiv MgL\cos(\theta) + \frac{1}{2}(ML^2 + I_G)\omega^2 \tag{6.8}$$

$$E(\theta_0, \omega_0) \equiv E(0, 0) = MgL \tag{6.9}$$

Solving for initial angular rate yields:

$$\omega_0(\theta_0) = \sqrt{\frac{2MgL(1 - \cos(\theta_0))}{ML^2 + I_G}} \tag{6.10}$$

It turns out that the results of the experiment are sensitive to these initial conditions. For example, if you assume that the system has a small initial angle but no initial velocity, then you will get slightly different results.

6.5.7 Toppling Point Mass with Infinite Friction

With infinite friction at the contact, the stick will start to slip when the vertical component of the contact forces goes to zero. Assuming the initial conditions from Equation 6.10, the vertical reaction force is given by:

$$\begin{aligned} V(\theta) =& \frac{Mg(I_G - 2mL^2\cos(\theta) + 3ML^2\cos^2(\theta))}{I_G + ML^2}\Bigg|_{I_G \to 0} \\ =& Mg\ cos(\theta)\ (3\cos(\theta) - 2) \end{aligned}$$

Solving for the non-trivial root of this equation yields the critical value for θ at which slipping occurs:

$$\theta_c = \arccos(2/3) \approx 48.189685^\circ \tag{6.11}$$

A little bit of simple algebra yields a nice expression for the angular velocity at this critical angle:

$$\omega_c = \sqrt{\frac{2g}{3L}} \tag{6.12}$$

While slipping with infinite friction the vertical component of the contact force is zero by definition. The horizontal component is given by:

$$H = 2I_G\left(\frac{g - L\omega^2\cos(\theta)}{L^2\sin(2\theta)}\right)\Bigg|_{I_G \to 0} = 0 \tag{6.13}$$

Since the only non-zero force acting on the stick (in this special case) is its own weight, the center of mass (CoM) will follow a parabolic trajectory. The initial position and velocity of the CoM are given by:

$$\vec{r}_c = \langle -L\sin(\theta_c), L\cos(\theta_c)\rangle \tag{6.14}$$

$$\dot{\vec{r}}_c = \langle -L\omega_c\cos(\theta_c), -L\omega_c\sin(\theta_c)\rangle \tag{6.15}$$

The final distance (d^*) reached by the CoM at the time of impact (t^*) can be found by solving the following system:

$$0 = \Big(L\cos(\theta_c)\Big) + \Big(-L\omega_c \sin(\theta_c)\Big)t^* + \frac{1}{2}\Big(-g\Big)(t^*)^2 \tag{6.16}$$

$$d^* = \Big(-L\sin(\theta_c)\Big) + \Big(-L\omega_c \cos(\theta_c)\Big)t^* \tag{6.17}$$

Since the CoM is falling/sliding in the negative direction, we need to add L to get the distance that the contact point slipped:

$$d_{slip} = d^* + L \tag{6.18}$$

After doing quite a bit of algebra and a letting $g = 1$ and $L = 1$ it can be shown that the slip distance is:

$$d_{slip} = \left(1 - \frac{4}{27}\sqrt{23} - \frac{5}{27}\sqrt{5}\right) \approx -0.124580 \tag{6.19}$$

6.5.8 Results - Pencil Toppling from Rest - General Case

For intermediate values of μ there is a regime in which the stick slips in *both* directions. In these cases, the stick first slides backwards, then rotates about a fixed point, and then slides forwards until ultimately striking the ground.

Sticks with low I_G slide for *all* values of μ. This includes the limit as the coefficient of friction goes to infinity.

Sticks with large I_G have a critical value of μ for which they do not slide.

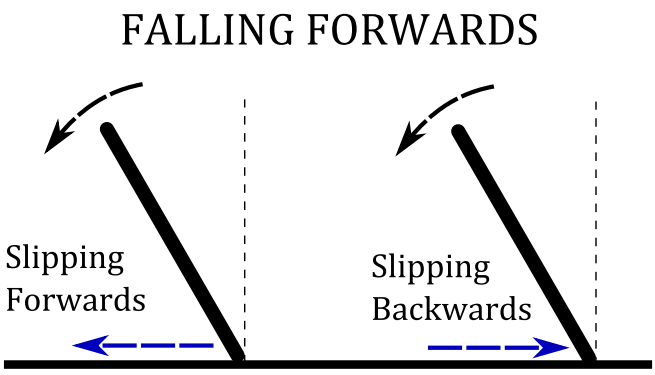

Figure 6.7: Shows sign conventions for slipping forwards and backwards

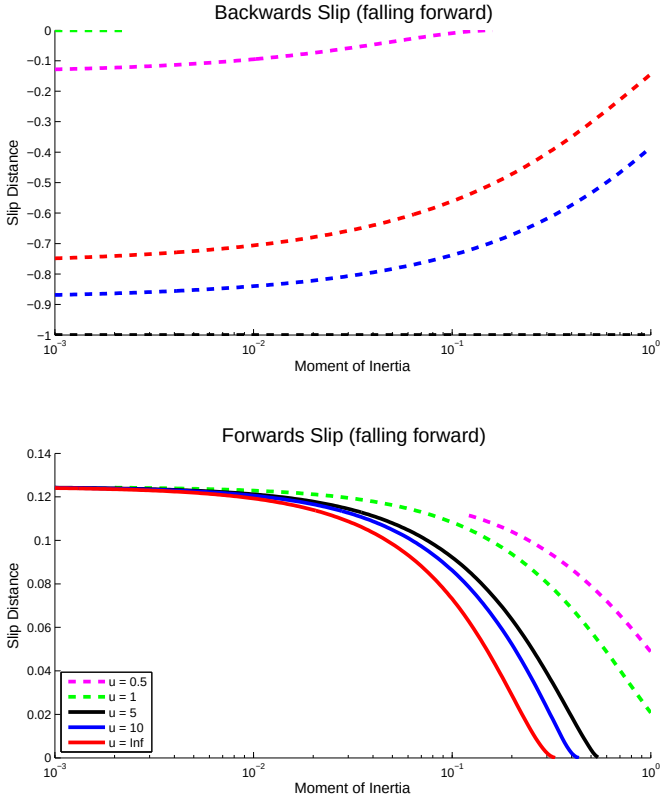

Figure 6.8: Positive and negative slip as a function of coefficient of friction and moment of inertia. Notice that the stick slides backwards with low friction and forwards for high friction. The slip distance is smaller for large sticks with a large moment of inertia.

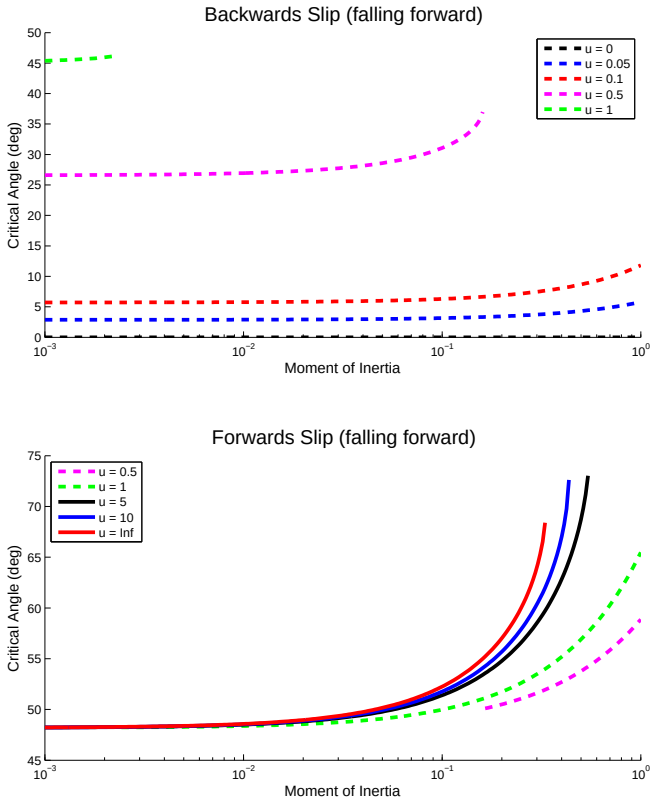

Figure 6.9: Critical angle before slipping as a function of coefficient of friction and moment of inertia. Notice that as the moment of inertia becomes small (like a point mass) the critical slip angle asymptotically approaches that of a ball bearing rolling off of a sphere.

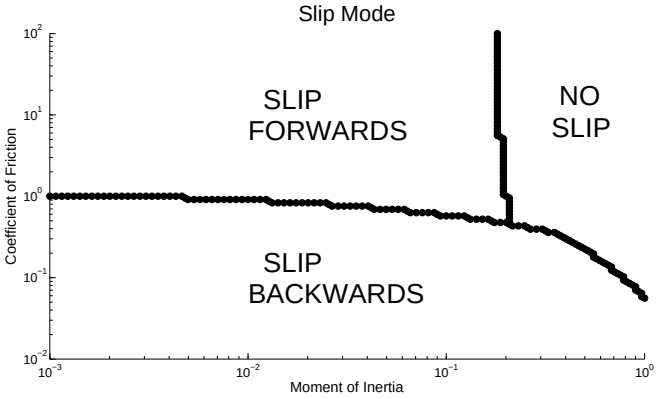

Figure 6.10: Shows which direction the stick will slip first. Notice that there is a small region where no sliding occurs, but that most of the region is dominated by either forwards or backwards sliding. Notice that for all reasonable values of friction ($\mu < 1$) the stick slips backwards first.

CHAPTER 7

AN INTRODUCTION TO TRAJECTORY OPTIMIZATION: HOW TO DO YOUR OWN DIRECT COLLOCATION

This chapter has been submitted to SIAM Review for publication as a tutorial and review paper.

Author List: Matthew Kelly

7.1 Abstract

This is a tutorial on numerical trajectory optimization, with a focus on direct collocation methods. These methods are often the go-to method for trajectory optimization, as they are both simple and effective. Additionally, these methods serve as a good starting point towards understanding more sophisticated trajectory optimization topics. After an introduction to trajectory optimization in general, we describe two common direct collocation methods in detail: trapezoidal and Hermite-Simpson. We then use these methods to set up and solve four example problems: 1) minimal-force trajectory to move a block between two points; 2) minimal-torque swing-up trajectory for a cart-pole; 3) minimal-torque periodic walking gait for a five-link model of bipedal walking; and 4) a pathological problem: minimal-work trajectory to move a block between two points.

7.2 Introduction

What is trajectory optimization? A trajectory is a path, for example, position vs time. More generally, the path may also include various other functions of time besides posi-

tion, such as velocities and the values of any controls. Trajectory optimization a set of methods that are used to find the controls, as a function of time, that result in the best path. The meaning of the word "best" varies from problem to problem, and is often dependent on constraints.

7.2.1 Why this paper?

There are many books and papers about trajectory optimization. The single best resource that I've found is *Practical Methods for Optimal Control and Estimation using Nonlinear Programming* by Betts [7], particularly the first 4 chapters. As opposed to Betts' book, this paper is narrower, aiming just at 'how to do it', and is intended to be a more gentle introduction. My goal is to give a good overview, as well as some fairly explicit recipes for doing trajectory optimization.

This tutorial is roughly divided into two parts. The first part focuses on giving a good introduction to the field, and giving all of the practical details required to implement both trapezoidal and Hermite-Simpson direct collocation methods. The second part of the tutorial is a collection of four example problems, which cover all of the practical details required to solve each problem.

Finally, this paper comes with an electronic supplement, which is described in detail in the appendix §7.12. Like the tutorial, there are two parts to the supplement. The first is a general purpose trajectory optimization library, written in Matlab, that implements both direct collocation methods from this tutorial, as well as direct multiple shooting (4^{th}-order Runge–Kutta) and global orthogonal collocation (Chebyshev Lobatto). The second part of the supplement is a collection of example problems, including all examples in this paper, implemented in Matlab and solved with the afore-mentioned trajectory

optimization library.

7.2.2 A simple example

Let's get started by looking at a simple example. Consider the problem of how to move a small block between two points on a straight line, starting and finishing at rest, in a fixed amount of time. A solution (trajectory) is said to be *feasible* if it satisfies all of the requirements (constraints) on the problem. For this simple problem, there are an infinite number of feasible solutions — ways to move the block between these two points in the allotted time.

Trajectory optimization is concerned with finding the best of the feasible trajectories, which is known as the *optimal* trajectory. We use an *objective function* to mathematically describe what we mean by "best" trajectory. Later in this tutorial we will solve this block moving problem with two commonly used objective functions: minimal force (§7.7) and minimal work (§7.10).

There are a huge number of ways to compute an optimal trajectory, and this tutorial will only focus on one small set of methods, known as *direct collocation*. One feature of direct collocation is that it computes the *open-loop* solution to the problem. In the case of our simple example, the open-loop solution would be the force to be applied to the block, as a function of time, that will move it between the initial and final points. In contrast, the *closed-loop* solution would give the force as a function of the state (position and velocity) of the block. These concepts are covered further in §7.3.3.

Another key feature of direct collocation is that the solution is obtained by discretizing the trajectory optimization problem (using a polynomial spline), and then optimizing

the discretized problem using non-linear constrained parameter optimization (non-linear programming). This process is known as *direct transcription*. The trade-offs between direct collocation and other methods are discussed in more detail in the following section (§7.3).

7.3 Background

In this section we will go into a more detailed discussion of the types of problems that are solved by direct collocation, and how it fits into the broader field of optimal control. Additionally, we will briefly cover some of the other techniques that are used to solve trajectory optimization problems, including indirect methods, direct multiple shooting,and orthogonal collocation.

7.3.1 Notation

For reference, these are the main symbols we will use throughout the tutorial, as will be described in detail later.

t_k	**time at knot point** k
N	**number of trajectory (spline) segments**
$h_k = t_{k+1} - t_k$	**duration of spline segment** k
$\boldsymbol{x}_k = \boldsymbol{x}(t_k)$	**state at knot point** k
$\boldsymbol{u}_k = \boldsymbol{u}(t_k)$	**control at knot point** k
$w_k = w(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k)$	**integrand of objective function at knot point** k
$\boldsymbol{f}_k = \boldsymbol{f}(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k)$	**system dynamics at knot point** k
$\dot{q} = \frac{dq}{dt} \qquad \ddot{q} = \frac{d^2q}{dt^2}$	**first and second time-derivatives of** q

In some cases we will use the subscript $k + \frac{1}{2}$ to indicate the mid-point of spline segment k. For example, $\boldsymbol{u}_k$ gives the control at the beginning of segment k, and $\boldsymbol{u}_{k+\frac{1}{2}}$ gives the control at the mid-point of segment k.

7.3.2 Trajectory optimization vs parameter optimization

Parameter optimization is concerned with minimizing some function $J(\boldsymbol{x})$, where $\boldsymbol{x}$ is a vector of real numbers. In contrast, trajectory optimization is concerned with minimizing a *functional* $J(\boldsymbol{f}(t))$, where $\boldsymbol{f}(t)$ is an arbitrary vector function. This makes trajectory optimization a more challenging problem, because the space of functions is much larger than the space of real numbers. *Direct Methods*, including direct collocation, work by converting a trajectory optimization problem into a related parameter

Figure 7.1: **Open- vs. close-loop control:** An *open-loop* solution (left) to an optimal control problem is a sequence of controls $\boldsymbol{u}(t)$ that move the system from a single starting point **A** to the destination point **B**. In contrast, the *closed-loop* solution gives the controls $\boldsymbol{u}(\boldsymbol{x})$ that can move the system from any point in the state space to the destination point **B**.

optimization problem. In practice, there are many ways to make this conversion, some of which will be discussed later in this tutorial.

7.3.3 Open-Loop vs. Closed-Loop Solutions

Trajectory optimization is a collection of techniques that are used to find *open-loop* solution to an optimal control problem. In other words, the solution to a trajectory optimization problem is a sequence of controls $\boldsymbol{u}^*(t)$, given as a function of time, that move a system from a single initial state to some final state. This sequence of controls, combined with the initial state, can then be used to define a single trajectory that the system takes through state space.

There is another set of techniques, known as *dynamic programming*, which find the *closed-loop* solution to an optimal control problem. These methods find the best control $\boldsymbol{u}^*(\boldsymbol{x})$ to take starting from *every* point in the state space. An optimal trajectory starting from any point in the state space can be recovered from a closed-loop solution by a simple simulation. Figure 7.1 illustrates the difference between an open-loop and a closed-loop solution.

In general, trajectory optimization is most useful for systems that are high-dimensional, have a large state space, or need to be very accurate. The resulting solution is open-loop, so it must be combined with a stabilizing controller when applied to a real system. One major short-coming of trajectory optimization is that it will sometimes fail to converge, or converge to a *locally optimal* solution, failing to find the *globally optimal* solution.

Dynamic programming tends to be most useful on lower-dimensional systems with small but complex state spaces, although some variants have been applied to high-dimensional problems [79]. There are two advantages to dynamic programming over trajectory optimization. The first is that dynamic programming gives the optimal control for *every* point in state space, and can thus be applied directly to a real system. The second, and perhaps more important advantage is that it will (at least in the basic formulations) always find the globally optimal solution. The downside of dynamic programming is that computing the optimal solution for every point in the state space is very expensive, scaling exponentially with the dimension of the problem.

7.3.4 Continuous-Time and Discrete-Time Systems

Trajectory optimization is generally concerned with finding optimal trajectories for a *dynamical system*. At any instant, the system is described by its state *state* $\boldsymbol{x}$. The state changes in time according the *dynamics* $\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u})$, where t is the current time, and $\boldsymbol{u}$ is the *control*, or input to the system.

There are many different types of dynamical systems. In this tutorial we will focus on *continuous-time* dynamical systems, which have continuous time, state, and control. This type of system is common in robotics and the aerospace industry, for example

planning the trajectory that a spacecraft would take between two planets.

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}) \qquad \textbf{continuous-time system} \tag{7.1}$$

Another common system is a *discrete-time* dynamical system, which has discrete time-steps, but continuous state and control. This type of system is commonly used in model predictive control, for example in building climate control systems [69]. Trajectory optimization for these systems is generally easier than on fully continuous systems.

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k) \qquad \textbf{discrete-time system} \tag{7.2}$$

A final type of dynamical system is a *directed graph*, where there is a finite set of states (nodes on the graph) and controls (transitions, actions, edges on the graph). Most algorithms for computing an optimal policy require the dynamical system to be in this form. A common example would be a traffic network, where the states are cities, and the controls are the different roads that could be taken between the cities. Sometimes continuous-time problems are abstracted into this form so that they can make use of sophisticated graph search algorithms to approximate the optimal policy.

7.3.5 The Trajectory Optimization Problem

There are many ways to formulate trajectory optimization problems [93, 6, 84]. Here we will restrict our focus to single-phase (no jumps between one governing equation and another) continuous-time trajectory optimization problems. A more general framework, which includes multi-phase problems can be found in [93]. Multi-phase problems typically arise from hybrid systems, which multiple continuous phases of motion, punctuated by discrete jumps.

Our objective function includes two terms: a boundary objective $J(\cdot)$ and a path integral along the entire trajectory, with the integrand $w(\cdot)$. Thus, our problem is said to

be in *Bolza form*. A problem with only the integral term is said to be in *Lagrange form*, while a problem with only a boundary term is said to be in *Mayer form*. [6]

$$\min_{t_0,t_F,\boldsymbol{x}(t),\boldsymbol{u}(t)} \underbrace{J(t_0,t_F,\boldsymbol{x}(t_0),\boldsymbol{x}(t_F))}_{\text{Mayer Term}} + \underbrace{\int_{t_0}^{t_F} w(\tau,\boldsymbol{x}(\tau),\boldsymbol{u}(\tau))\,d\tau}_{\text{Lagrange Term}} \tag{7.3}$$

The decision variables in the optimization are the initial and final time (t_0, t_F), as well as the state and control trajectories, $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ respectively.

The optimization is subject to a variety of limits and constraints, detailed in the following equations (7.4-7.11). The first, and perhaps most important of these constraints are the system dynamics, which are typically non-linear and describe how the system changes in time.

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(t,\boldsymbol{x}(t),\boldsymbol{u}(t)) \qquad \textbf{system dynamics} \tag{7.4}$$

Next are the path constraints, which enforce restrictions along the trajectory. A path constraint could be used, for example, to keep the foot of a walking robot above the ground during a step.

$$\boldsymbol{h}(t,\boldsymbol{x}(t),\boldsymbol{u}(t)) \leq \boldsymbol{0} \qquad \textbf{path constraints} \tag{7.5}$$

Another important type of constraint is a non-linear boundary constraint, which puts restrictions on the initial and final state of the system. Such a constraint would be used, for example, to ensure that the gait of a walking robot is periodic.

$$\boldsymbol{g}(t_0,t_F,\boldsymbol{x}(t_0),\boldsymbol{x}(t_F)) \leq \boldsymbol{0} \qquad \textbf{boundary constraints} \tag{7.6}$$

Often there are constant limits on the state or control. For example, a robot arm might have limits on the angle, angular rate, and torque that could be applied at each joint, during the entire trajectory.

$$\boldsymbol{x}_{\text{low}} \leq \boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upp}} \qquad \textbf{path bound on state} \tag{7.7}$$

$$\boldsymbol{u}_{\text{low}} \leq \boldsymbol{u}(t) \leq \boldsymbol{u}_{\text{upp}} \qquad \textbf{path bound on control} \tag{7.8}$$

Finally, it is often important to include specific limits on the initial and final time and state. These might be used to ensure that the solution to a path planning problem reaches the goal within some desired time window.

$$t_{\text{low}} \leq t_0 < t_F \leq t_{\text{upp}} \qquad \textbf{bounds on initial and final time} \tag{7.9}$$

$$\boldsymbol{x}_{0,\text{low}} \leq \boldsymbol{x}(t_0) \leq \boldsymbol{x}_{0,\text{upp}} \qquad \textbf{bound on initial state} \tag{7.10}$$

$$\boldsymbol{x}_{F,\text{low}} \leq \boldsymbol{x}(t_F) \leq \boldsymbol{x}_{F,\text{upp}} \qquad \textbf{bound on final state} \tag{7.11}$$

This problem is difficult to solve, because the decision variables include *functions*, rather than just real numbers. Additionally, the objective and constraint functions include both integrals and derivatives, which inherently operate on functions. In the following section we will cover some methods to address these concerns, by converting the trajectory optimization problem into a non-linear program (non-linear constrained parameter optimization).

7.3.6 Indirect Methods

Traditionally, trajectory optimization problems were solved by *indirect methods*, which are based on calculus of variations and Pontryagin's maximum principle [115]. An indirect method works by constructing the necessary and sufficient conditions for optimality, which are then solved numerically. This is sort of like minimizing $y = f(t)$ by solving the equation $y'(t^*) = 0$ subject to the constraint $y''(t^*) > 0$. In contrast, a *direct* method will minimize $y(t)$ by constructing a sequence of guesses such that each subsequent guess is an improvement on the previous: $y(t_0) > y(t_1) > \ldots > y(t^*)$.

The major benefit of an indirect method, when compared to a direct method, is that the indirect method tends to be more accurate. Additionally, the error estimates for an

indirect method tend to be more reliable, since they can be constructed from the adjoint equations.

There are several difficulties associated with indirect methods when compared to direct methods. The region of convergence tends to be smaller for indirect methods than direct methods, which means that an indirect method will require a better initialization [6]. Furthermore, the initialization of an indirect method is complicated by the need to initialize the adjoint variables, which are not used in a direct method [7]. Finally, in order to obtain an accurate solution for an indirect method, it is typically necessary to construct the necessary and sufficient conditions analytically, which can be challenging [6].

7.3.7 Direct Collocation Method

In this tutorial we will focus on direct collocation methods for solving trajectory optimization problems. Direct collocation methods are desirable because they are easy to construct and initialize (when compared to indirect methods), and are easier to use on problems with path constraints (when compared to shooting methods, covered in §7.3.9) [50, 6, 7]. Aside: the name *direct collocation* is sometimes used interchangeably with the name *direct transcription* [6].

The key feature of a direct method is that it converts the trajectory optimization problem into a non-linear program. The decision variables in a trajectory optimization problem are continuous functions, which are infinite-dimensional, and thus difficult to optimize over. A non-linear program (7.12), by contrast, is a non-linear constrained

parameter optimization, over a finite set of real numbers.

$$\min_{\boldsymbol{z}} J(\boldsymbol{z}) \qquad \text{subject to:} \tag{7.12}$$

$$\boldsymbol{f}(\boldsymbol{z}) = \mathbf{0}$$

$$\boldsymbol{g}(\boldsymbol{z}) \leq \mathbf{0}$$

$$\boldsymbol{z}_{\text{low}} \leq \boldsymbol{z} \leq \boldsymbol{z}_{\text{upp}}$$

There are a variety of methods for solving non-linear programs, which are covered in [68, 7, 6]. In this tutorial, we will not cover methods for solving non-linear programs, apart from how they interact with the direct collocation method. There are several good software programs that implement non-linear programming solvers, including [37, 91, 119, 12, 70].

Direct collocation methods work by approximating the solution to the trajectory optimization problem as piece-wise polynomial functions (polynomial splines). This approximation is then used to construct *collocation constraints*, which enforce the system dynamics along the trajectory. These collocation constraints are applied at specific points (times) along the trajectory, know as *collocation points*. In other words, the polynomial spline must satisfy the system dynamics exactly at each collocation point. Additionally, each type of polynomial spline defines a quadrature rule, which can then be used to approximate any integral expressions in the problem. [7, 50, 46]

7.3.8 Orthogonal Collocation

Orthogonal collocation is similar to direct collocation, but it generally uses high-order polynomials with collocation points at the roots of an orthogonal polynomial, typically either Chebyshev or Legendre [19]. Increasing the accuracy of a solution is typically

achieved by increasing either the number or trajectory segments or the order of the polynomial in each segment.

One important reason to use high-order orthogonal polynomials for function approximation is that they achieve *spectral* convergence. This means that the convergence rate is exponential in the order of the polynomial [93], *if* the underlying function is sufficiently smooth [111]. If the entire trajectory is approximated using a single high-order polynomial, then the resulting method is called a *pseudospectral* collocation (also called *global collocation*) [93].

One of the key implementation details about orthogonal collocation is that the trajectory is represented using *Barycentric Interpolation* [5], rather than directly from the definition of the orthogonal polynomial. Barycentric interpolation provides a numerically efficient and stable method for interpolation, differentiation, and quadrature, all of which can be computed by knowing the trajectory's value at the collocation points. See Appendix §7.14 for further details about how to work with orthogonal polynomials.

7.3.9 Direct Shooting Methods

Like direct transcription, the *direct shooting method* (also known as *single shooting*) solves a trajectory optimization problem by transforming it into a non-linear program. The key difference is that a direct shooting method approximates the trajectory using a simulation. The decision variables in the non-linear program are some (open-loop) parameterization of the control along the trajectory, and the initial state. Direct shooting is well suited to applications where the control is simple and there are few path constraints, such as space flight.[6]

A common extension of the direct shooting method is *multiple shooting* (also called *parallel shooting*). Rather than represent the entire trajectory as a single simulation, the trajectory is divided up into segments, and each segment is represented by a simulation. Multiple shooting tends to be much more robust than single shooting, and thus is used on more challenging trajectory optimization problems. [6]

When compared to collocation methods, shooting methods tend to create smaller non-linear programs (fewer decision variables in the constrained parameter optimization). One difficulty with direct shooting methods is that it is difficult to implement path constraints, since the intermediate state variables are not decision variables in the non-linear program [6]. Another difficulty with shooting methods, particularly with direct shooting, is that the relationship between the decision variables and constraints is often highly nonlinear, which can cause convergence problems in some cases [6, 7].

7.3.10 Differential Dynamic Programming

One final method to briefly mention is *Differential Dynamic Programming*. It is similar to direct shooting, in that it simulates the system forward in time, and then optimizes based on the result of that simulation. The difference is in how the optimization is carried out. While direct shooting is optimized by a general-purpose non-linear programming solver, the differential dynamic programming algorithm optimizes the trajectory by propagating the optimal control backward along the candidate trajectory. In other words, it exploits the time-dependent nature of the trajectory. It was described in [72, 52], and a good overview provided by [78].

7.3.11 Trajectory Optimization Software

There are a variety of software programs that solve trajectory optimization problems. Each of these solvers performs some transcription method and then hands the problem off to a non-linear programming solver.

- GPOPS-II [84] is a trajectory optimization library developed for Matlab
- PSOPT [3] is an open-source trajectory optimization library
- SOS [8] is a professional trajectory optimization software, developed by John T. Betts, implementing algorithms in [7].
- DIRCOL [116] is an open-source trajectory optimization library

Direct transcription methods solve a trajectory optimization problem by converting it into a non-linear programming. Here I have included a list of a few popular software packages for solving non-linear programming problems.

- *FMINCON* [70] is part of the Matlab optimization toolbox.
- *SNOPT* [37] is produced by TomLab optimization, and is widely regarded as one of the best sparse non-linear programming solvers.
- *IPOPT* [119] is an open-source non-linear programming solver.

The electronic supplement, described in Appendix §7.12, also includes a Matlab library for trajectory optimization. It was written to go along with this tutorial, and it implements trapezoidal and Hermite-Simpson collocation, as well as all four examples problems.

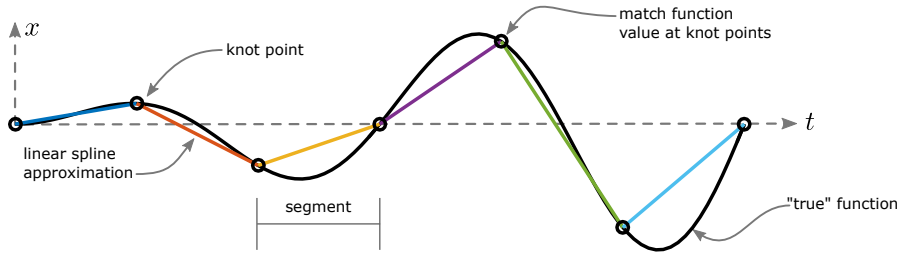

Figure 7.2: **Linear Spline:** The trapezoidal collocation method is derived by assuming that the dynamics and control functions are approximated by linear splines, like the one shown here. Note that the spline and the function that it is approximating match exactly at the knot points.

7.4 Trapezoidal Collocation Method

The trapezoidal method for direct collocation works by approximating both the control and the dynamics along the trajectory as linear splines. Practically, this means that the continuous state and control trajectories $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ can be represented by their values at the knot points $t_0 \dots t_N$ of the spline, where N is the number of linear segments. A *knot point* is a point on a spline that connects two continuous segments, in this case two *linear* segments as shown in Figure 7.2.

7.4.1 Quadrature

There are often integral expressions in trajectory optimization. Usually they are in the objective function, but occasionally they are found in the constraints as well. Just like the dynamics and control, we approximate the integrand of the objective function as a linear spline through the knot points. The integral of a linear spline is computed using the trapezoid rule, as shown below. [7]

$$\int_{t_0}^{t_F} w(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau))\, d\tau \quad \approx \quad \sum_{k=0}^{N-1} \frac{h_k}{2}(w_k + w_{k+1}) \tag{7.13}$$

7.4.2 Collocation Constraints

In a collocation method, the *collocation constraints* are used to approximate the system dynamics, and are equivalent to implicit Runge–Kutta integration schemes. The collocation constraints are applied at *collocation points*. In the case of trapezoidal collocation, the collocation points happen to be identical to the knot points of the linear spline, although this is not true of all direct collocation methods. [7]

The trapezoidal collocation equations (below) are very similar to the quadrature rule (7.13). It basically says that the integral of the linear spline approximation of the dynamics along the trajectory must move the system between the successive knot points of the trajectory. This constraint is then applied for each segment of the linear spline. [7]

$$\frac{h_k}{2}(\boldsymbol{f}_{k+1} + \boldsymbol{f}_k) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \qquad k \in 0 \ldots (N-1) \tag{7.14}$$

Note that $\boldsymbol{x}_k$ is a decision variable in the non-linear program, while $\boldsymbol{f}_k$ is the result of evaluating the system dynamics.

7.4.3 Interpolating the Solution

The quadrature and collocation equations were derived by assuming that the system dynamics and control were both linear splines. Thus, we will use these same splines when evaluating the solution between the collocation points. One interesting aspect of this formulation is that the state is a quadratic spline, because $\dot{\boldsymbol{x}} = \boldsymbol{f}(\cdot)$ and $\boldsymbol{x} = \int \dot{\boldsymbol{x}}\, dt$, and $\boldsymbol{f}(\cdot)$ is a linear spline. In general, if the control is a spline of order n, then the state is represented by a spline of order $n + 1$ [7].

We start by looking at segment k, such that the query time t, is inside the given

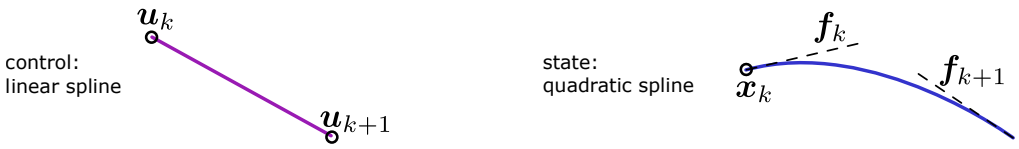

Figure 7.3: **Solution Interpolation: trapezoidal Collocation** When using trapezoidal collocation, we are implicitly assuming that the dynamics and control are linear splines, which are then explicitly used to evaluate the solution. Since the state is the integral of the system dynamics, it is represented by a quadratic spline (the integral of a linear spline).

segment.

$$t_k \leq t \leq t_{k+1}$$

We will define $\delta_k = t - t_k$ to be the difference between the query time and the time at the lower knot point. Note that $0 \leq \delta_k < h_k$ if the interval is chosen correctly. The control spline is then given by simple linear interpolation.

$$\boldsymbol{\beta} = \frac{1}{h}(\boldsymbol{u}_{k+1} - \boldsymbol{u}_k)$$

$$\boldsymbol{u}(t) = \boldsymbol{u}_k + \delta_k \boldsymbol{\beta}$$

The state spline is very similar, and it is constructed by assuming that the dynamics are approximated by the same type of linear spline as the control. We can obtain the state by integrating the dynamics. Similarily, we obtain the (quadratic) state spline equations by integrating the (linear) dynamics spline. Alternatively, you can think of each segment of the state trajectory like a parabola defined by the value at one side, and the slope at both. [7]

$$\boldsymbol{\gamma} = \frac{1}{2h}(\boldsymbol{f}_{k+1} - \boldsymbol{f}_k)$$

$$\hat{\boldsymbol{x}}(t) = \boldsymbol{x}_k + \delta_k \boldsymbol{f}_k + \delta_k^2 \boldsymbol{\gamma}_k$$

Figure 7.3 shows how these splines are constructed, and Appendix §7.13 shows how to derive these equations.

7.4.4 Putting it all Together

The first step in solving any trajectory optimization problem is to clearly pose it. First, write down the objective function, the system dynamics, and any constraints. Next, find a good non-linear programming solver, such as FMINCON [70], SNOPT [37], or IPOPT [119]. The non-linear programming solver will then demand that you define an objective function, as well as a function for any non-linear constraints. There will also be a way to include simple linear or constant constraints. Our goal here is to take the trajectory optimization problem (infinite dimensional constrained optimization) and convert it into a non-linear program (finite dimensional constrained optimization).

Let's start with the objective function. The integral terms in the objective function are given by (7.13). The boundary objective terms can be directly evaluated, since the initial and final state (and time) are both included as decision variables.

The system dynamics (7.4) are treated as an equality constraint, using (7.14). Any path constraints are applied to the the state at all collocation-points, while the boundary constraints are only applied to the initial and final points. Constant limits on the state and control can should be applied to all collocation-points.

Finally, once you've managed to pose the trajectory optimization problem as a non-linear program, you can solve it! Once that is done, you then use linear interpolation to compute a continuous approximation of the control, and quadratic interpolation to compute a continuous approximation of the state.

This process is written out for the block-move and cart-pole swing up examples, in §7.7.4 and §7.8.5 respectively.

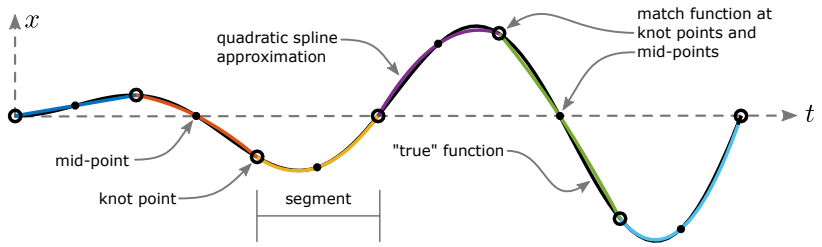

Figure 7.4: **Quadratic Spline:** This figure shows a quadratic spline, which is used to approximate the dynamics and control functions in the Hermite-Simpson collocation method. Notice that this approximation is far more accurate than the linear spline in Figure 7.2, for the same number of segments.

7.5 Hermite-Simpson Collocation Method

The Hermite-Simpson method for direct collocation works by approximating both the control $\boldsymbol{u}(t)$ and the dynamics $\boldsymbol{f}(\cdot)$ along the trajectory as piece-wise quadratic splines. As a result, the state is approximated by a cubic spline. One advantage of Hermite-Simpson Collocation, over trapezoidal collocation, is that the state has a continuous first derivative across the knot points when the collocation constraints are perfectly satisfied. Additionally, Hermite-Simpson collocation is 4^{th}-order accurate, while trapezoidal collocation is only 2^{nd}-order accurate [7].

In the Hermite-Simpson method, both end-points and the mid-point of each spline segment are collocation points, as shown in Figure 7.4. Note that the quadratic spline used here is different than the quadratic spline used to represent the state in trapezoidal collocation. The spline used here is defined by its value at both end-points and the mid-point, while the quadratic spline for interpolation the state in trapezoidal collocation is defined by its value at one end-point and the *slope* at both end-points.

7.5.1 Quadrature

Just like in trapezoidal collocation, we approximate definite integrals in the objective and constraint functions using quadrature. The integrand, like the dynamics and control, is approximated by a quadratic spline. The integral of a quadratic spline is given by Simpson's rule, below.

$$\int_{t_0}^{t_F} w(\tau)\, d\tau \quad \approx \quad \sum_{k=0}^{N-1} \frac{h_k}{6}(w_k + 4w_{k+\frac{1}{2}} + w_{k+1})$$

7.5.2 Collocation Constraints

The Hermite-Simpson method is a bit more complicated than the trapezoid rule when it comes to enforcing the system dynamics. Just like the trapezoid rule, we are going to use the quadrature rule to ensure the quadratic spline approximation of the dynamics moves the system between knot points. In order to do this, we actually need two constraints.

The first constraint ensures that the mid-point state is consistent with the cubic spline approximation of the state. This constraint is derived from the definition of a cubic Hermite spline, hence the term *Hermite* in the method name.

$$\boldsymbol{x}_{k+\frac{1}{2}} = \frac{1}{2}(\boldsymbol{x}_k + \boldsymbol{x}_{k+1}) + \frac{h_k}{8}(\boldsymbol{f}_k - \boldsymbol{f}_{k+1}) \tag{7.15}$$

The second constraint is actually the collocation constraint, ensuring that the system dynamics are satisfied. This equation is based on Simpson quadrature, hence the term *Simpson* in the method name.

$$\frac{h_k}{6}(\boldsymbol{f}_k + 4\boldsymbol{f}_{k+\frac{1}{2}} + \boldsymbol{f}_{k+1}) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \tag{7.16}$$

Both (7.15) and (7.16) are from Bett's book [7], and a derivation of them is given in the Appendix §7.13.6 of this tutorial.

7.5.3 Compressed vs. Separated Form

There are two common implementations of the Hermite-Simpson method. The *compressed* method only includes the state of the system at the knot points $(\boldsymbol{x}_0 \dots \boldsymbol{x}_N)$ as decision variables. The state at each mid-point $(\boldsymbol{x}_{0+\frac{1}{2}} \dots \boldsymbol{x}_{(N-1)+\frac{1}{2}})$ is computed explicitly using (7.15) before computing the dynamics $\boldsymbol{f}_{k+\frac{1}{2}}$ at each mid-point [7].

The *separated* method includes the state at both end-points and mid-points as decision variables, and (7.15) is given as an constraint to the NLP solver.

The textbook by Betts [7] gives a detailed comparison of the two methods. The trade-offs are complicated and problem dependent, but as a general rule, the separated form is better when the number of spline segments (N) is small, and the compressed form is better when the number of spline segments is large.

7.5.4 Interpolating the Solution

After the non-linear programming solver has given us a solution, we may wish to evaluate that solution at points along the trajectory other than the collocation points. We can do this by *interpolating the solution*, using interpolation rules constructed from the same assumptions that we used to derive the quadrature and collocation equations: the system dynamics and control were both quadratic splines.

We start by selecting the segment k that contains the query time t.

$$t_k \leq t \leq t_{k+1}$$

We will define $\delta_k = t - t_k$ to be the difference between the query time and the time at the lower knot point. Note that $0 \leq \delta_k < h_k$ if the interval is chosen correctly. The control

Figure 7.5: **Solution Interpolation: Hermite-Simpson Collocation** is derived by assuming that both the dynamics and control are quadratic splines. The state is represented by a cubic spline, defined by its slope(dashed lines) at both end-points and the mid-point, as well as the value (small circles) at the lower knot point [7].

spline is then given by quadratic interpolation, where the coefficients are determined by the control values at both end-points of the segment, as well as the mid-point. [7]

$$\boldsymbol{\beta}_1 = \frac{-1}{h_k}(3\boldsymbol{u}_k - 4\boldsymbol{u}_{k+\frac{1}{2}} + \boldsymbol{u}_{k+1})$$
$$\boldsymbol{\beta}_2 = \frac{2}{h_k^2}(\boldsymbol{u}_k - 2\boldsymbol{u}_{k+\frac{1}{2}} + \boldsymbol{u}_{k+1})$$
$$\boldsymbol{u}(t) = \boldsymbol{u}_k + \delta_k\,\boldsymbol{\beta}_1 + \delta_k^2\,\boldsymbol{\beta}_2$$

The state spline is very similar. It is constructed by assuming that the dynamics are approximated by the same type of quadratic spline as the control. You can think of each segment of the state spline being defined by the slope (dynamics) at both end-points and the mid-point, and by the state at the lower knot point. [7]

$$\gamma_2 = \frac{-1}{2h_k}(3\boldsymbol{f}_k - 4\boldsymbol{f}_{k+\frac{1}{2}} + \boldsymbol{f}_{k+1})$$
$$\gamma_3 = \frac{2}{3h_k^2}(\boldsymbol{f}_k - 2\boldsymbol{f}_{k+\frac{1}{2}} + \boldsymbol{f}_{k+1})$$
$$\boldsymbol{x}(t) = \boldsymbol{x}_k + \delta_k\,\boldsymbol{f}_k + \delta_k^2\,\gamma_2 + \delta_k^3\,\gamma_3$$

Figure 7.5 shows a graphical representation of the state and control splines, and Appendix 7.13 gives derivation of the interpolation equations.

7.6 Practical Considerations

7.6.1 Initialization

Direct collocation methods work by computing a sequence of candidate solution trajectories, each of which is better than the previous. Such an algorithm must start from somewhere, known as the initial guess, which is provided by the user. This initial guess can be very important, depending on the problem.

Imagine that the optimization is trying to get to the top of a hill. If the landscape is simple, with only one hill, then it doesn't matter where the optimization starts: it can just go uphill until it finds the solution. What happens if there are two different hills, and one is higher? Then, there will be some starting points where going uphill will only get you to the shorter of the two hills. In this case, the optimization will know that it got to the top of the hill, but it won't know that there is an even higher hill somewhere else.

Just like in the simple hill-climbing analogy, the choice of initial guess can affect which local minimum the optimization eventually converges to. The presence of constraints makes it even worse: there might be some starting points from which the optimization cannot even find a feasible solution. This is just a fundamental problem with non-linear programming solvers: they cannot always find a solution, and if they do find a solution, then it is only guaranteed to be locally optimal.

The best initializations for trajectory optimization problems usually require some knowledge that is specific to that problem, but there are a few general approaches that might be useful. In this way, initialization is more of an art than a science. A generally good practice is to try several different initialization strategies, and see if they all converge to the same solution (if they converge at all). See §7.6.4 for some debugging

concepts that can be used to help determine if a solution is converging correctly.

One of the simplest initialization techniques is to just assume that the trajectory is a straight line in state space between the initial and final states. This approach is easy to implement, and will often work well, especially for simple boundary value problems.

If you have a rough idea of what the behavior should look like, then you can just put that in as the initial guess. For example, if you want a robot to do a back-flip, sketch out the robot at a few points throughout the back-flip, figure out the points in state-space for each configuration, and then use linear interpolation between those points.

For complicated problems, a more principled approach might be required. My favorite technique is to simplify the trajectory optimization problem until I can get a reasonable solution using a simple initialization technique. Then I use the solution of the simplified problem to initialize the original problem. If this doesn't work, then I simply construct a series of trajectory optimization problems, each of which is slightly closer to the desired problem, and which uses the previous solution as the initial guess.

For example, lets say that you want to find a minimum-work trajectory for a walking robot. This cost function is difficult (see §7.10), and there are some difficult non-linear constraints (foot clearance, contact forces, walking speed). Start by replacing the cost function with minimum torque-squared (like the five-link biped example, §7.10), removing most of the constraints, and replacing the non-linear dynamics with simple kinematics (joint acceleration = joint torque). Solve this problem, and then use the solution to solve the same problem, but with the real system dynamics. Then add back in all of the constraints, and use the previous solution to initialize the new problem. Add back in the constraints, and then the original objective function. This process also is a good way to detect and isolate bugs in your code.

7.6.2 Mesh Refinement

The direct transcription process approximates a trajectory using polynomial splines, which allows the trajectory optimization to be converted into a non-linear program. One side effect of this is that the non-linear program is solving the system dynamics along the trajectory; the collocation constraints are acting as implicit Runge–Kutta integration schemes [7].

The important detail here is that the direct collocation process is solving the system dynamics on some fixed *mesh* (or *grid*) defined by the collocation points. This grid can limit the accuracy of the solution. Using a fine mesh (many collocation points) can make the solution more accurate, but at significant computational cost.

Mesh refinement is a process by which the trajectory optimization is initially solved iteratively: first on a coarse mesh, and then adding more collocation points to the mesh on each subsequent iteration. Since it is expensive to add new points everywhere, mesh refinement methods typically determine where to place new collocation points such that they will have the most impact on improving the accuracy of the solution.

New points are typically added to the mesh first computing the discretization error in each segment (see §7.6.3), and then sub-dividing the worst segments into one or more new segments. Figure 7.6 shows a simple example of how the mesh for a linear spline might be refined to produce a more accurate representation by adding a small number of points. Notice that segments with a small error are left unchanged, while segments with more error are sub-divided into 2, 3, or 4 sub-segments for the next iteration. Betts discusses algorithms for mesh refinement in his text book [7].

In more sophisticated mesh-refinement methods, the accuracy of a given segment might be improved by sub-dividing it or by increasing the polynomial order inside the

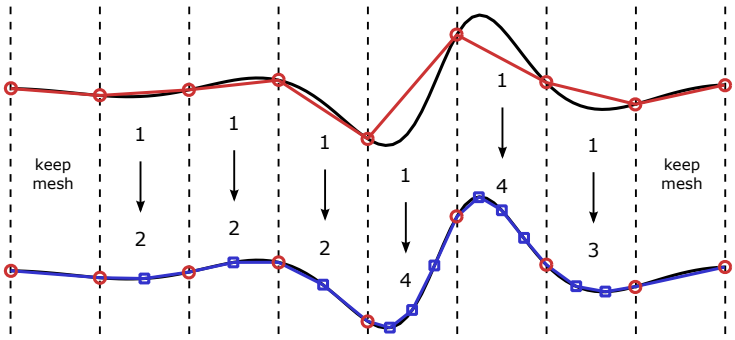

Figure 7.6: **Mesh Refinement:** This figure shows an example of how the mesh of a linear spline could be refined to increase the accuracy of the function approximation. Each segment of the spline is sub-divided into a integer number of sub-segments, where the number of sub-segments is related to how much error there was in the original approximation.

segment. Such algorithms are referred to as hp-adaptive meshing. The rough idea is to sub-divide a segment if there is a big spike in the error profile (within the segment), but to increase the polynomial order otherwise. Details of the algorithm can be found in [20], which are implemented in the GPOPS-II software [84].

7.6.3 Error Analysis

It is important to quantify the accuracy of a candidate solution to a trajectory optimization, to check if it meets the required tolerances. This is done by estimating the error along the trajectory. If the tolerances are not met, then this error estimate can be used to re-mesh the problem and solve again, as discussed in §7.6.2.

There are many possible error metrics for trajectory optimization [7]. Here we will construct an error estimate based on how well the candidate trajectory satisfies the system dynamics *between* the collocation points. The logic here is that if the system dynamics are accurately satisfied between the collocation points, then the polynomial spline is an accurate representation of the system, which would then imply that the non-linear

program is an accurate representation of the original trajectory optimization problem.

We do not know the true solution $\boldsymbol{x}^*(t),\ \boldsymbol{u}^*(t)$ of the trajectory optimization problem, but we do know that it must satisfy the system dynamics:

$$\dot{\boldsymbol{x}}^*(t) = \boldsymbol{f}(t, \boldsymbol{x}^*(t), \boldsymbol{u}^*(t))$$

From this, we can construct an expression for the error in the solution to the differential equations, along the candidate trajectory [7].

$$\varepsilon(t) = \dot{\boldsymbol{x}}(t) - \boldsymbol{f}(t, \boldsymbol{x}(t), \boldsymbol{u}(t))$$

This error $\varepsilon(t)$ will be zero at each collocation point, and non-zero elsewhere. We can compute the integral of this function numerically to determine how far the candidate solution (polynomial spline) may have deviated from the true solution along each dimension of the state. [7]

$$\boldsymbol{\eta}_k = \int_{t_k}^{t_{k+1}} |\varepsilon(\tau)|\, d\tau$$

Once you have the error in each state over each segment of the trajectory, you can use this to determine how to re-mesh the trajectory (§7.6.2) so that your solution converges to an optimal solution that satisfies the continuous dynamics. Betts [7] provides additional details about this process, and how it can be used to automate mesh refinement.

7.6.4 Debugging your Code

There are many ways that trajectory optimization can go wrong. In this section, we discuss some common bugs that find their way into code, and a few techniques for locating and fixing them. Betts [7] also provides a good list of debugging suggestions.

One common problem is that the non-linear program fails to converge: it just runs until it hits a resource limit. One cause of this is that your problem has a family of

non-unique solutions. Basically, the objective function looks like a plateau in some high-dimensional space. This causes a problem because there is no single best solution. A simple fix is to add a small regularization term to your cost function, such as the integral of control squared along the trajectory. This puts a shallow bowl in the objective function, modifying the problem such that it has a unique solution.

A trajectory optimization problem with a non-smooth *solution* might cause the non-linear program to converge very slowly. This occurs in our final example: finding the minimal work trajectory to move a block between two points (§7.10). There two basic ways to deal with a discontinuous solution. The first is to do mesh refinement (§7.6.2), so that there are many short segments near the discontinuity. The second way to handle a discontinuous solution is to slightly modify the problem, typically by introducing a smoothing term, such that the solution is stiff, but not discontinuous. This second approach was used in [105].

The non-linear program expects both the objective and constraint functions to be *consistent*, which means that they perform the same exact sequence of arithmetic operations on ever call [7]. For practical purposes you can think of a consistent function as one that is both smooth and deterministic. Any functions that fail to meet this criteria will show up as noise to the non-linear programming solver, which will cause slow convergence. See §7.6.5 for more details.

If the non-linear programming solver returns saying that the problem is infeasible, then there are two likely scenarios. The first is that your problem is actually impossible, for example, you might have contradictory constraints. In these cases, you can often figure out some clues by looking at final point in the non-linear programming solution (the best of the infeasible trajectories). What constraints are active? Is the trajectory right on top of your initial guess? Is it running into an actuator limit? The basic debugging

procedure is to double check all of your constraints, and then check to see if removing a constraint allows for a feasible solution.

The second likely cause of an infeasible report from a non-linear programming solver is when a trajectory optimization problem is complicated and initialized with a poor guess. In these cases, the optimization gets stuck in a 'bad' local minima, that has no feasible solution. The best fix in this case it to use the methods discussed in §7.6.1 to compute a better initialization.

Another tricky thing with trajectory optimization is determining if a candidate solution is at a global or a local minimum. In both cases the non-linear programming solver will report success. In general, there is no rigorous way to determine if you have the globally optimal solution, but there are many effective heuristics. One such heuristic is to run the optimization from a wide variety of initial guesses. If most of the guesses converge to the same solution, and it is better than all others found, there is a good chance that it is the globally optimal solution. Another such heuristic is to use different transcription methods, and check that they all converge to the same solution.

7.6.5 Consistent Functions

Direct transcription solves a trajectory optimization problem by converting it to a non-linear program. Most non-linear programming solvers, such as SNOPT [91], IPOPT [12], and FMINCON [70], require that the user-defined objective and constraint functions be *consistent*. A function is consistent if it performs the exact same sequence of arithmetic operations on each call [7]. This is essentially like saying that the output of functions must smoothly vary with the inputs, and that the function must be deterministic.

For example, the `abs()` function is not consistent, because it contains an `if()` statement, which changes the sequence of operations: when $t > 0$ it simple returns t, but when $t \leq 0$ it negates t before returning. The functions `min` and `max` are also not consistent functions, although the reasoning is a bit more complicated.

There is a neat trick that allows many inconsistent functions (such as `abs()`, `max()`) to be implemented by introducing extra decision variables (*slack variables*) and constraints to your problem. An example is given in §7.10, to correctly treat the `abs()` function in the objective function of the block moving example. Bett's text book [7] also covers the topic in more detail. An alternative way to handle such functions is to use smoothing, which is also demonstrated in the block-moving example in §7.10.

Another place where inconsistency shows up is when a function has an internal iteration loop, such as in root finding, or in a variable-step integration method. The solution for the root-finding method is to just use a fixed number of iterations, and the variable-step integration should be replaced with a fixed-step method. [7]

One final source of inconsistency, which is particularly common in direct shooting methods, is the use of a time-stepping simulation to compute the system dynamics. The contact solvers in such simulators are inconsistent, which then leads to problems in the non-linear program. One solution to this problem is to use through-contact trajectory optimization [76, 87].

7.7 Block Move Example: Minimal Torque

Let's consider a simple example problem, where we would like to move a block between two points on a plane in a fixed amount of time, as shown in Figure 7.7. There are an

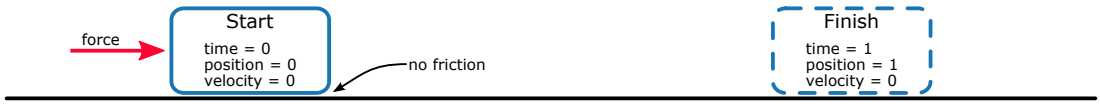

Figure 7.7: **Block Move Example:** Find the optimal trajectory to move a block between two points on a friction-less plane, starting and finishing at rest, in a fixed amount of time.

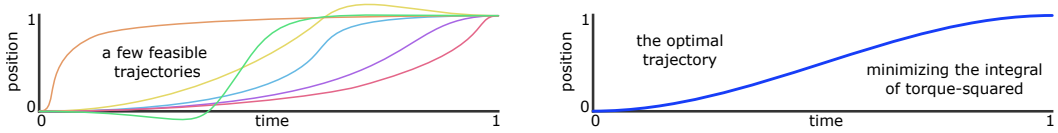

Figure 7.8: **A Simple Example:** Move a block between two points by applying a force, such that both the initial and final speed are zero. All of the trajectories shown on the left are *feasible*: they satisfy the problem constraints. In trajectory optimization we typically introduce an objective function, either to force a unique solution or to obtain some desired behavior. The plot on the right shows an *optimal* trajectory, obtained by minimizing the integral of torque-squared along the trajectory.

infinite number of trajectories that satisfy this requirement, so we will add an *objective function*, which describes the 'best' trajectory between these points. Figure 7.8 shows a few of the possible trajectories between these two points, as well as the optimal solution, which, in this case, is minimizing the integral of force-squared along the trajectory.

7.7.1 Problem Statement

We would like to move a block along a one-dimensional friction-less surface, in a finite time, along a trajectory that minimizes the integral of the absolute work done by the control force u, where the position and velocity of the block are given by x and v respectively.

$$\min_{u(t),\,x(t),\,v(t)} \int_0^1 u^2(\tau)\,d\tau \tag{7.17}$$

We will assume that the block has unit mass and slides without friction, so we can write it's dynamics:

$$\dot{x} = v \qquad \dot{v} = u \tag{7.18}$$

Next, the block must start at the origin, and move one unit of distance in one unit of time. Note that the block must be stationary at both start and finish.

$$\begin{aligned} x(0) &= 0 & x(1) &= 1 \\ v(0) &= 0 & v(1) &= 0 \end{aligned} \tag{7.19}$$

7.7.2 Analytic Solution

The solution to the simple block moving trajectory optimization problem (7.17-7.19) is:

$$u^*(t) = 6 - 12t \qquad x^*(t) = 3t^2 - 2t^3 \tag{7.20}$$

It turns out that this simple problem has an analytic solution, but, in general, trajectory optimization problems must be solved *numerically* — they have no analytic solution.

7.7.3 Initialization

There are many ways to initialize this problem. In this tutorial we simply assume that the position of the block (x) transitions linearly between the initial and final position. Then, we select the velocity (v) and force (u) profiles that are consistent with that initial guess for position.

$$x^{\text{init}}(t) = t \qquad v^{\text{init}}(t) = 1 \qquad u^{\text{init}}(t) = 0 \tag{7.21}$$

7.7.4 Solution via Trapezoid Collocation

We can collect all of the equations that describe the block moving problem (7.17-7.19), and combine them with the trapezoidal collocation method from §7.4, to write down the block moving trajectory optimization problem as a constrained parameter optimization problem (non-linear program).

minimize:

$$J = \sum_{k=0}^{N-1} \tfrac{h}{2}(u_k^2 + u_{k+1}^2) \qquad \textbf{objective function}$$

decision variables:

$$x_0 \ldots x_N, \quad v_0 \ldots v_N, \quad u_0 \ldots u_N$$

subject to:

$$\tfrac{h}{2}(v_{k+1} + v_k) = x_{k+1} - x_k \qquad k \in 0 \ldots (N-1) \qquad \textbf{collocation constraints}$$

$$\tfrac{h}{2}(u_{k+1} + u_k) = v_{k+1} - v_k \qquad k \in 0 \ldots (N-1) \qquad \textbf{collocation constraints}$$

$$x_0 = 0 \quad x_N = 1 \qquad \textbf{boundary constraints}$$

$$v_0 = 0 \quad v_N = 0 \qquad \textbf{boundary constraints}$$

initial guess:

$$x_k^{\text{init}} = \tfrac{k}{N}, \quad v_k^{\text{init}} = 1, \quad u_k^{\text{init}} = 0$$

Note that the duration of each segment of the trajectory is given by $h = 1/N$, where N is the number of segments used in the transcription. In general, direct collocation methods require solving a non-linear program. This problem is special; the constraints are all linear, and the cost function is quadratic, making it a quadratic program. Quadratic programs (in general) are much easier to solve than non-linear programs.

7.8 Cart-Pole Example Problem

The second example in this tutorial is the cart-pole swing-up problem. The system has a cart that travels on a horizontal track, and a simple pendulum freely hangs from the cart. A motor in the cart can be used to apply a force to accelerate the cart along the track. The goal is to compute the force profile, as a function of time, that moves the cart back and forth on the track in such a way that the pendulum, initially hanging at rest, is swung up to be balanced (inverted) above the cart.

In this section, we will turn this physical problem statement into a clear trajectory optimization problem, and then show how to convert that trajectory optimization problem into a non-linear program using direct collocation.

7.8.1 System Dynamics

The cart-pole is a second-order dynamical system, and its equations of motion can be derived using methods found in any undergraduate dynamics text book. The dynamics of this system are simple enough to compute by hand, although for more complicated systems it is generally a good idea to make use of a computer algebra package to derive the dynamics.

The dynamics for the cart-pole system are shown below. The position of the cart is given by q_1, the angle of the pole is given by q_2, and the control force is given by u. The mass of the cart and pole are given by m_1 and m_2 respectively, and then length of the pole and acceleration due to gravity are ℓ and g, as shown in Figure 7.9.

$$\ddot{q}_1 = \frac{\ell\, m_2 \sin(q_2)\, \dot{q}_2^2 + u + m_2\, g\, \cos(q_2)\, \sin(q_2)}{m_1 + m_2\,(1 - \cos^2(q_2))} \tag{7.22}$$

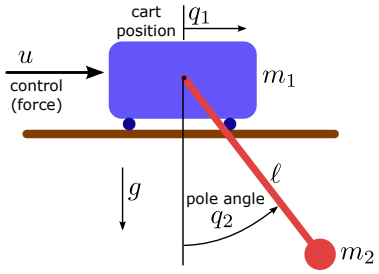

Figure 7.9: **Cart-Pole:** This figure shows an the set-up for the cart-pole system.

$$\ddot{q}_2 = -\frac{\ell\, m_2 \cos(q_2)\, \sin(q_2)\, \dot{q}_2^2 + u\, \cos(q_2) + (m_1 + m_2)\, g\, \sin(q_2)}{\ell\, m_1 + \ell m_2\, (1 - \cos^2(q_2))} \tag{7.23}$$

All standard trajectory optimization methods require that the dynamics of the system be in first-order form. This is accomplished by including both the minimal coordinates (q_1 and q_2) and their derivatives in the state. Note that $\ddot{q}_1$ and $\ddot{q}_2$ are defined in (7.22,7.23).

$$\boldsymbol{x} = \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} \qquad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, u) = \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix}$$

7.8.2 Objective Function

For this example we will use one of the more common objective functions in trajectory optimization: the integral of the control squared.

$$J = \int_0^T u^2(\tau)\, d\tau \tag{7.24}$$

This objective function (7.24) tends to produce smooth trajectories. Smooth trajectories are desirable for two key reasons. The first is that most transcription methods assume

that the solution to the trajectory optimization problem is well-approximated by a polynomial spline, thus a problem with a solution that is smooth will be solved more quickly and accurately than a problem with a non-smooth solution. The second reason that smooth trajectories are nice is that they tend to be easier to stabilize with conventional controllers, when implemented on a real system.

The torque-squared objective function (7.24) only includes terms involving the control (u), but in general the objective function can include both the time t and the state $\boldsymbol{x}$ of the system as well. Furthermore, this objective function only has a path integral term, but in general the objective function could have a path integral term as well as a term for the boundaries of the trajectory. Notice also that we have prescribed both the initial and final times $t_0 = 0$ and $t_F = T$ for this problem, although in general these can also be decision variables.

7.8.3 Boundary Constraints

Many trajectory optimization problems include *boundary constraints*, which restrict the state of the system at the boundaries of the trajectory — either the initial or final point (or both). Here we will restrict the full state of the cart-pole system at both the initial and final points on the trajectory. Let's suppose that we want the cart to start in the center of the rails and translate a distance d for its swing-up maneuver. The (constant) boundary constraints for this situation are given below.

$$
\begin{aligned}
q_1(t_0) &= 0 & q_1(t_F) &= d \\
q_2(t_0) &= 0 & q_2(t_F) &= \pi \\
\dot{q}_1(t_0) &= 0 & \dot{q}_1(t_F) &= 0 \\
\dot{q}_2(t_0) &= 0 & \dot{q}_2(t_F) &= 0
\end{aligned}
$$

7.8.4 State and Control Bounds

The cart-pole swing-up problem has a few simple constraints. First, let's look at the state. The cart rides on a track which has a finite length, so we need to include a simple constraint the limits the horizontal range of the cart. Additionally, we will restrict the motor force to some maximal force in each direction.

$$-d_{\max} \leq q_1(t) \leq d_{\max}$$

$$-u_{\max} \leq u(t) \leq u_{\max}$$

7.8.5 Trapezoidal Collocation - Cart-Pole Example

We can collect all of the equations in this section, and combine them with the trapezoidal collocation method from §7.4, to write down the cart-pole swing-up problem as a non-linear program.

minimize:

$$J = \sum_{k=0}^{N-1} \frac{h_k}{2}(u_k^2 + u_{k+1}^2) \qquad \textbf{objective function} \tag{7.25}$$

decision variables:

$$\boldsymbol{x}_0 ... \boldsymbol{x}_N \quad u_0 \ldots u_N \tag{7.26}$$

subject to:

$$\frac{h_k}{2}(\boldsymbol{f}_{k+1} + \boldsymbol{f}_k) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \qquad k \in 0 \ldots (N-1) \qquad \textbf{collocation constraints} \tag{7.27}$$

$$-d_{\max} \leq q_1 \leq d_{\max} \qquad \textbf{path constraints} \tag{7.28}$$

$$-u_{\max} \leq u \leq u_{\max} \qquad \textbf{path constraints} \tag{7.29}$$

$$\boldsymbol{x}_0 = \mathbf{0} \quad \boldsymbol{x}_N = [d, \pi, 0, 0]^T \qquad \textbf{boundary constraints} \tag{7.30}$$

Note that $h_k = t_{k+1} - t_k$. Here, we will us a uniform grid, so $t_k = k\frac{T}{N}$, where N is the number of segments used in the transcription. In general, you could solve this problem on an arbitrary grid; in other words, each h_k could be different.

7.8.6 Hermite-Simpson Transcription Method

We can also use Hermite-Simpson collocation (§7.5) to construct a non-linear program for the cart-pole swing-up problem. This is similar to the trapezoidal collocation, but using a second-order (rather than first-order) spline to approximate the dynamics and control. This requires including collocation points for the state and control at the mid-

point of each segment. Here we are using the *separated* form of the method (see §7.5.3).

minimize:

$$J = \sum_{k=0}^{N-1} \frac{h_k}{6}(u_k^2 + 4u_{k+\frac{1}{2}}^2 + u_{k+1}^2) \qquad \textbf{objective function} \tag{7.31}$$

decision variables:

$$\boldsymbol{x}_0,\ \boldsymbol{x}_{0+\frac{1}{2}}\ \ldots\ \boldsymbol{x}_N \qquad u_0,\ u_{0+\frac{1}{2}}\ \ldots\ u_N$$

subject to:

$$\boldsymbol{x}_{k+\frac{1}{2}} = \tfrac{1}{2}(\boldsymbol{x}_k + \boldsymbol{x}_{k+1}) + \tfrac{h_k}{8}(\boldsymbol{f}_k - \boldsymbol{f}_{k+1}) \qquad k \in 0 \ldots (N-1) \qquad \textbf{interpolation constraints} \tag{7.32}$$

$$\tfrac{h_k}{6}(\boldsymbol{f}_k + 4\boldsymbol{f}_{k+\frac{1}{2}} + \boldsymbol{f}_{k+1}) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \qquad k \in 0 \ldots (N-1) \qquad \textbf{collocation constraints} \tag{7.33}$$

$$-d_{\max} \leq q_1 \leq d_{\max} \qquad \textbf{path constraints} \tag{7.34}$$

$$-u_{\max} \leq u \leq u_{\max} \qquad \textbf{path constraints} \tag{7.35}$$

$$\boldsymbol{x}_0 = \mathbf{0} \quad \boldsymbol{x}_N = [d, \pi, 0, 0]^T \qquad \textbf{boundary constraints} \tag{7.36}$$

7.8.7 Constructing an initial guess

The cart-pole swing-up problem is a boundary value problem. We known the initial and final state, and simply need to compute a (optimal) trajectory to move the system between those two points. An obvious (and simple) initial guess is that the system linearly moves between the initial and final state with zero control effort. This is not

physically possible, but the motion is similar to the desired motion.

$$\boldsymbol{x}_{\text{guess}}(t) = \frac{t}{T}\begin{bmatrix} d \\ \pi \\ 0 \\ 0 \end{bmatrix} \qquad u_{\text{guess}}(t) = 0 \tag{7.37}$$

Additionally, we will start with a uniform grid, such that $t_k = k\frac{T}{N}$. The initial guess for each decision variable in the non-linear program is then computed by evaluating (7.37) at each knot point t_k (and the mid-point $t_{k+\frac{1}{2}}$ for Hermite-Simpson collocation).

7.8.8 Results

In this section we show the optimal swing-up trajectory for the cart-pole system, computed using Hermite-Simpson collocation with 25 trajectory segments. The set of parameters that we use are given in Appendix §7.15.1. We computed the solution in Matlab, on a regular desktop computer[1], using the code provided in the electronic supplement (§7.12). The non-linear program was solved by FMINCON in 5.91 seconds (71 iterations) using default convergence settings.

Figure 7.10 shows a stop-action animation of the swing-up maneuver, with uniformly spaced frames. The same solution is shown in Figure 7.11 as plots of state and control versus time. Finally, Figure 7.12, shows the error estimates along the trajectory.

The error in both the differential equations and the state increase noticeably near the middle of the trajectory. At this point, the system is changing rapidly as the pole swings-up, and the uniform grid has difficulty capturing the solution. A more sophisticated

[1] processor: 3.4GHz quad-core Intel i5-3570K

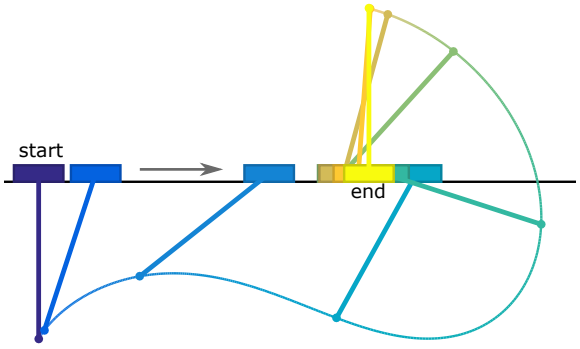

Figure 7.10: **Cart-Pole Swing-Up: Draw** This figure shows a stop-action animation of the cart-pole swing-up. Blue (dark) to yellow (light) as the trajectory progresses.

method would compute a new grid, such that the trajectory segments were shorter near this point where the system is rapidly changing.

We selected parameters for this problem such that it is well behaved. For example, we can make small changes to the initial guess or the direct transcription method, and we get the same basic answer out. There are a few interesting behaviors for other choices of parameters. For example, setting $d = 0$, so that the cart ends at the same point on the track where it started, gives the problem two equally good solutions: one where the cart moves left, and one where it moves right. If d is small, but non-zero, then this gives a strong local minimum to the problem: a bad initialization could easily produce a solution which is locally optimal, but not the true global solution.

Another way to make this problem harder is to increase the duration T, so that the optimal solution includes several swings back-and-forth before the ultimate swing-up. Reducing the actuator limits $u_{\max}$ sufficiently will also make the problem harder, since the solution will no longer be smooth. This will then require careful mesh-refinement for an accurate answer. These changes are easy to make in the code provided in the electronic supplement.

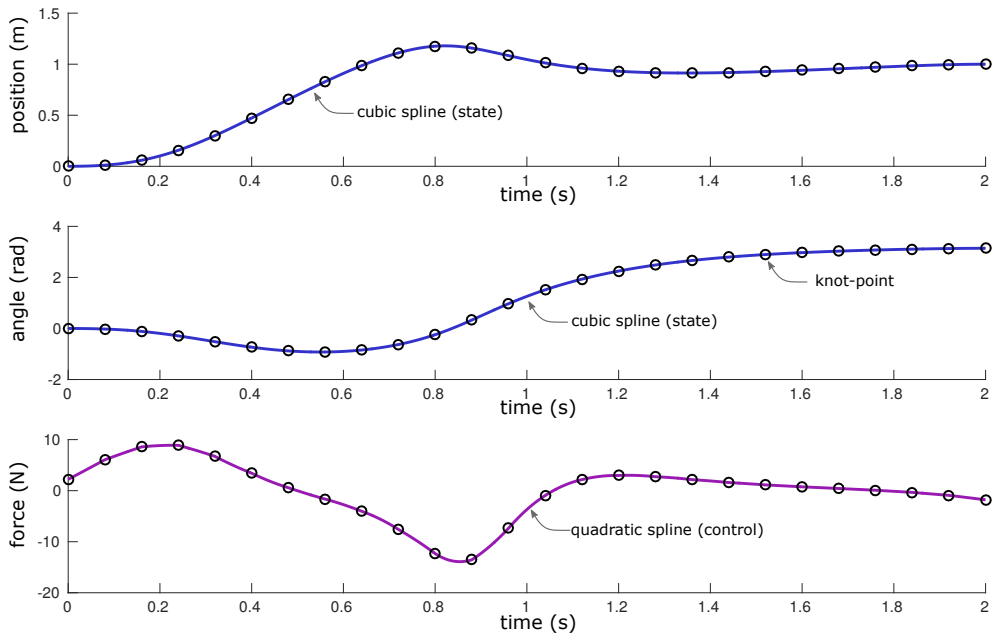

Figure 7.11: **Cart-Pole Swing-Up: Plot** This figure shows the solution to the cart-pole swing-up, using a 25 segment Hermite-Simpson transcription over a uniform grid. The top two plots show the horizontal position and the angle of the pole, while the bottom plot shows the control force.

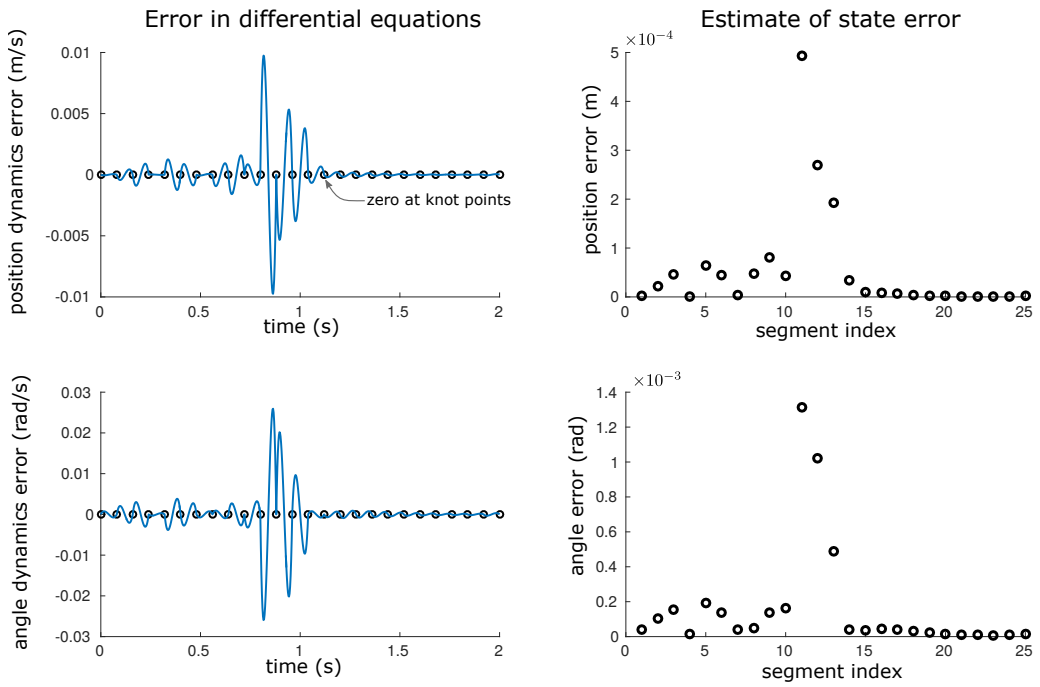

Figure 7.12: **Cart-Pole Swing-Up: Error** This figure shows the estimates in the error for the cart-pole solution. The left two figures show the continuous error in the solution of the differential equations (dynamics), while the right shows the estimate for the error in the state over each segment of the solution spline. These error estimates are calculated using the methods described in §7.6.3. Notice that the error is largest near the middle of the trajectory, where the angular rate and applied force are both large. If re-meshing the problem, this would be an excellent place to subdivide the mesh (see §7.6.2).

7.9 Five-Link Biped

In this section we will use trajectory optimization to find a periodic walking gait for the five-link (planar) biped walking model. This model is commonly used when studying bipedal walking robots [124, 88, 82, 40, 98, 121]. For this example, we will use the model developed by [121], with parameters that are selected to match for the walking robot RABBIT [14] and given in Appendix §7.15.2.

We will assume that the robot is left-right symmetric, so we can search for a periodic walking gait using a single step (as opposed to a stride, which would consist of two steps). A periodic walking gait means that joint trajectories (torques, angles, and rates), are the same on each successive step. We will be optimizing the walking gait such that it minimizes the integral of torque-squared along the trajectory. We chose this objective function because it tends to produce smooth solution trajectories. Optimization based on more realistic energy models, such as minimal work trajectories, tend to be more difficult to work with, as illustrated in §7.10.

7.9.1 Five-Link Biped Model

Figure 7.13 shows a cartoon of the five-link biped model as it takes a step. The model consists of two legs, each of which has an upper (femur) and lower (tibia) link, as well as a torso. The *stance* leg is supporting the weight of the robot, while the *swing* leg is not touching the ground. Each link is modeled as a rigid body, with both mass and rotational inertia. Links are connected to each-other with ideal torque motors across friction-less revolute joints, with the exception of the ankle joint, which is passive. We have included the derivation of the equations of motion for this model in Appendix 7.16.

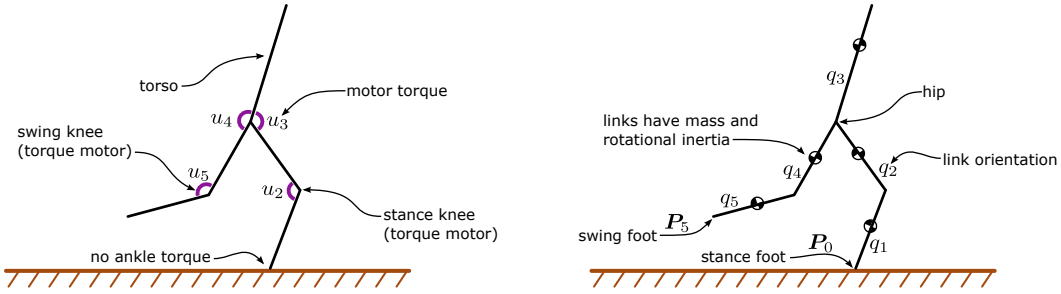

Figure 7.13: **Five Link Biped Model:** Assumptions: planar rigid body dynamics, revolute joints. The dynamics are modeled as a kinematic chain, with each joint is connected to its parent by an ideal revolute joint and torque source. The robot is under-actuated, because the stance ankle has no motor.

7.9.2 System Dynamics

During single stance, the five-link biped model has 5 degrees of freedom, shown by q_i in Figure 7.13. We will collect these configuration variables into single vector $\boldsymbol{q}$. Because the model has second order dynamics, we must also keep track of the derivative of the configuration: $\dot{\boldsymbol{q}}$. Thus, we can write the state and the dynamics as shown below, where $\ddot{\boldsymbol{q}}$ is calculated from the system dynamics.

$$\boldsymbol{x} = \begin{bmatrix} \boldsymbol{q} \\ \dot{\boldsymbol{q}} \end{bmatrix} \qquad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u}) = \begin{bmatrix} \dot{\boldsymbol{q}} \\ \ddot{\boldsymbol{q}} \end{bmatrix}$$

Unlike the cart-pole, the dynamics function $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})$ cannot easily be written in closed form. We have shown one method for deriving and evaluating the system dynamics in Appendix 7.16.

7.9.3 Objective Function

Just like in the cart-pole example, we will use the integral of torque-squared cost function. This cost function tends to produce smooth, well-behaved solutions. This is desired for a few reasons. First, a smooth solution means that a piece-wise polynomial spline

will do a good job of approximating the solution, thus the non-linear program will converge well. The second reason is that a smooth solution is easier to control on a real robotic system. Finally, minimizing the torque-squared tends to keep the solution away from large torques, which are sometimes undesirable on real robotic systems.

$$J = \int_0^T \left(\sum_{i=1}^{5} u_i^2(\tau) \right) d\tau \tag{7.38}$$

There are many other cost functions that could be used. One common one is *cost of transport* (CoT), the ratio of energy used over the trajectory to the horizontal distance moved by the robot [112, 9]. It turns out that CoT is actually a difficult cost function to optimize over, because the *solutions* to such problems tend to be discontinuous. This concept is covered in detail in §7.10.

7.9.4 Constraints

A variety of constraints are required to produce a sensible walking gait. The constraints presented here are similar those used in [121].

First, we will require that the walking gait is *periodic*. That is, the initial state must be identical to the final state after it is mapped through heel-strike. *Heel-strike* is the event that occurs when the swing foot strikes the ground at the end of each step, becoming the new stance foot.

For a single step, let's define $\boldsymbol{x}_0$ to be the initial state, and $\boldsymbol{x}_F$ to be the final state on the trajectory, immediately *before* heel-strike. Then we can express the periodic walking constraint as shown below, where $\boldsymbol{F}_H(\cdot)$ is the heel-strike map, defined in Appendix §7.16.

$$\boldsymbol{x}_0 = \boldsymbol{f}_H(\boldsymbol{x}_F) \tag{7.39}$$

Next, we would like the biped to walk at some desired speed. There are many ways to do this, but what we have chosen here is to prescribe the duration of a single step (T), and then put a equality constraint on step length (D). Additionally, we assume that the robot is walking on flat ground. This constraint can then be written as shown below, where $\boldsymbol{P}_5(T)$ is the position of the swing foot at the end of the step, and $\boldsymbol{P}_0(t)$ is the position of the stance foot throughout the step.

$$\boldsymbol{P}_5(T) = \begin{bmatrix} D \\ 0 \end{bmatrix} \qquad \text{(Note: } \boldsymbol{P}_0(t) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ by definition)} \tag{7.40}$$

In our lab, we are interested in walking robots with small feet, so we have added an additional constraint on the biped robot: that the stance ankle torque is identically zero throughout the trajectory. As a side note: it is a trivial matter to remove this constraint, and it does not change the optimization problem too much. On the other hand, the presence (or absence) of ankle torques has a large effect on the type of control strategy that can be used to stabilize such a walking gait [121].

When we derived the heel-strike collision equations (see Appendix §7.16), we assumed that the trailing foot left the ground at the instant the leading foot collided with the ground. We can ensure that this is true by introducing a constraint that the vertical component of the swing foot velocity at the beginning of the trajectory must be positive (foot lifting off the ground), and that it must be negative at the end of the trajectory (foot moving towards the ground). These constraints can be expressed as inequality constraints on the initial and final state, where $\hat{\boldsymbol{n}}$ is the normal vector of the ground. In our case, $\hat{\boldsymbol{n}} = \hat{\boldsymbol{j}} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$, because the ground is flat and level.

$$0 < \dot{\boldsymbol{P}}_5(0) \cdot \hat{\boldsymbol{n}} \qquad\qquad 0 > \dot{\boldsymbol{P}}_5(T) \cdot \hat{\boldsymbol{n}} \tag{7.41}$$

We could have also included a constraint to keep the swing foot above the ground at all times, but we observed that this was always true, so the additional constraint was

unnecessary. If we were to select a different cost function, say cost of transport, then we may need to add such a constraint, and it would be written:

$$0 < \boldsymbol{P}_5(t) \cdot \hat{\boldsymbol{n}} \qquad \forall t \in (0, T) \tag{7.42}$$

In some cases, it might be desirable to achieve some clearance for the swing foot. There are a few ways to do this. The easiest is to require that the swing foot remain above some continuous function $y(t)$ of time. A slightly more complicated version is to prescribe some continuous function $y(x)$ that the swing foot must remain above, such as a simple quadratic or cubic polynomial (7.44), where $\hat{\boldsymbol{i}} = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$. In both cases, it is critical that the constraint is consistent with the boundary conditions, and that that the implementation is smooth and does not over-constrain the problem.

$$y(t) < \boldsymbol{P}_5(t) \cdot \hat{\boldsymbol{n}} \qquad \forall t \in (0, T) \qquad \textbf{foot clearance (time-based)} \tag{7.43}$$

$$y(\boldsymbol{P}_5(t) \cdot \hat{\boldsymbol{i}}) < \boldsymbol{P}_5(t) \cdot \hat{\boldsymbol{j}} \qquad \forall t \in (0, T) \qquad \textbf{foot clearance (state-based)} \tag{7.44}$$

Finally, it is worth noting one common source of error: redundant constraints. Notice, for example, that for step length we only have a constraint on the final position of the foot (7.40). The initial position is fully constrained given (7.40) and the periodic step map constraint (7.39). If we were to add a constraint on the initial position of the foot, it would only serve to cause numerical problems in the non-linear program.

7.9.5 Initialization

To initialize the problem, we constructed a final set of joint angles that put the robot in a pose that was close to a reasonable walking configuration. Then, we applied the angle portion of the step map (see §7.89), to construct a consistent set of angles for the initial state. We obtained all intermediate configurations by linear interpolation between these

two configurations.

$$\boldsymbol{q}(0)_{\text{guess}} = \begin{bmatrix} -0.3 \\ 0.7 \\ 0.0 \\ -0.5 \\ -0.6 \end{bmatrix} \qquad \boldsymbol{q}(T)_{\text{guess}} = \begin{bmatrix} -0.6 \\ -0.5 \\ 0.0 \\ 0.7 \\ -0.3 \end{bmatrix} \tag{7.45}$$

We initialized the joint rates such that they were consistent with the angle joint configuration interpolation.

$$\dot{\boldsymbol{q}}(t)_{\text{guess}} = \frac{1}{T}(\boldsymbol{q}(T)_{\text{guess}} - \boldsymbol{q}(0)_{\text{guess}}) \tag{7.46}$$

Finally, we initialize the joint torques to be constant at zero.

$$\boldsymbol{u}(t)_{\text{guess}} = \mathbf{0} \tag{7.47}$$

This initial guess is clearly far from satisfying the system dynamics and constraints. One of the easiest ways to refine this guess is to solve a 'rough' version of the problem. In this case, we first solve the problem on a coarse mesh (5 segments in the trajectory), and use loose convergence tolerances in the non-linear programming (NLP) solver. We use this 'coarse-grid' solution as the initial guess for the second iteration, using a fine mesh (25 segments) with tight convergence tolerances in the NLP solver.

7.9.6 Results

We solved this problem in Matlab, using FMINCON's [70] interior-point algorithm as the non-linear programming solver. The physical parameters that we used are given in Appendix 7.15.2, and the optimization programs were run on a regular desktop computer[2] . We chose to use analytic gradients (Appendix 7.16) for the entire problem, although similar results are obtained for numerical gradients.

[2]processor: 3.4GHz quad-core Intel i5-3570K

All source code for solving this trajectory optimization problem, including derivation of the equations of motions, is given in the electronic supplement (see Appendix §7.12.

We solved the problem on two meshes, using Hermite-Simpson collocation in both cases. The initial mesh had 5 segments, and a low convergence tolerance (in FMINCON, `'TolFun' = 1e-3`). For the second (final) mesh, we used a mesh with 25 segments, and increased the convergence tolerance in FMINCON to `'TolFun' = 1e-6`. Both meshes had segments of uniform duration. This process could be repeated further, to achieve increasingly accurate solutions.

The solution on the initial (5-segment) mesh took 0.96 seconds to compute, and 29 iterations in FMINCON's interior-point method. The solution on the final (25-segment) mesh took 21.3 seconds to computer, and 56 iterations in the NLP solver.

As an aside, if we solve the problem using FMINCON's build-in numerical derivatives, rather than analytic derivatives, we get the same solution as before, but it takes longer: 4.30 seconds and 29 iterations on the coarse mesh, and 79.8 seconds and 62 iterations on the fine mesh. Also, for this problem, it turns out that solving on two different meshes is not critical; we could directly solve the problem on the fine (25 segment) mesh, and obtain similar results.

The solution for a single periodic walking step is shown in Figure 7.14 as a stop-action animation with uniformly spaced frames. The same trajectory is also shown in Figure 7.15, with each joint angle and torque given as a continuous function of time. Finally, Figure 7.16 shows the error estimates computed along the trajectory.

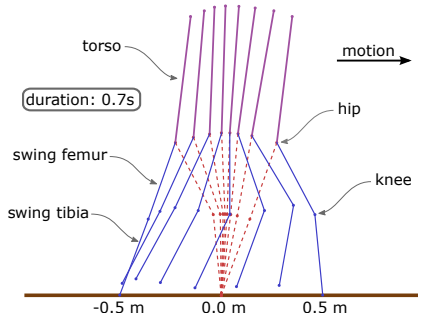

Figure 7.14: **Five-Link Biped: Stop-Action Animation** of the minimal-torque walking gait for the five-link biped model of walking.

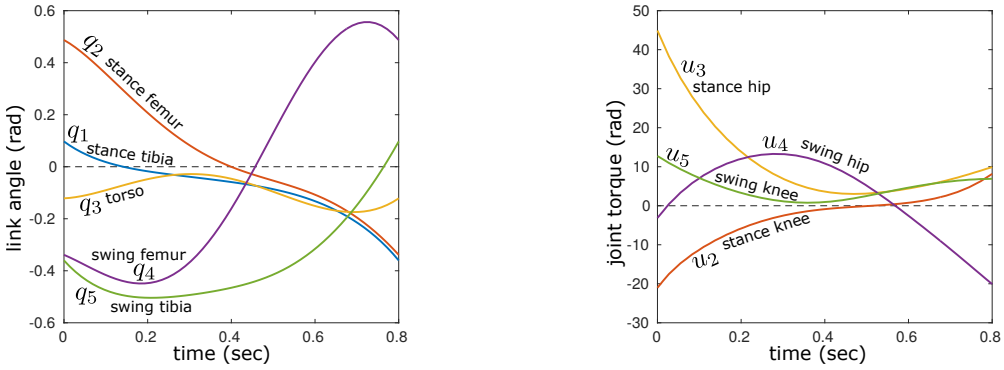

Figure 7.15: **Five-Link Biped:** This figure shows the solution to the 5-link biped trajectory optimization. Notice that the curves are smooth, partially due to the integral of torque-squared cost function.

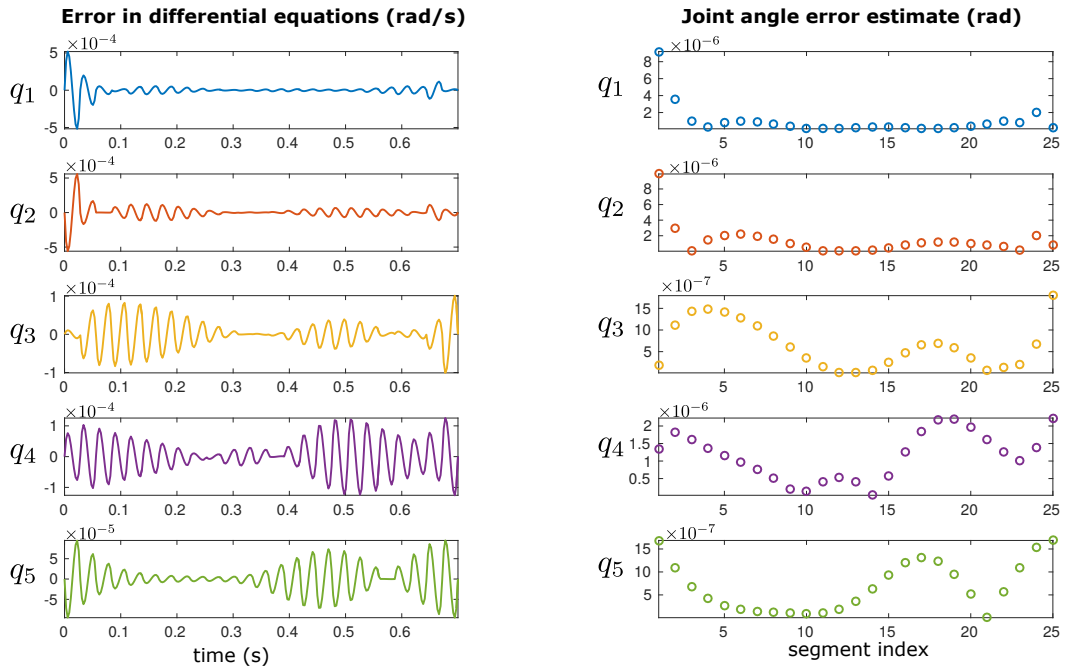

Figure 7.16: **Five-Link Biped: Error** This figure shows the estimates in the error for the five-link biped solution. The left two figures show the continuous error in the solution of the differential equations (dynamics), while the right shows the estimate for the error in the state over each segment of the solution spline. These error estimates are computed using the techniques described in §7.6.3.

7.10 Block Move Example: Minimal Work

In this section, we will revisit the simple block-moving example from §7.7, but with a more challenging objective function. All other details of the problem remain unchanged: The block must move between two points that are one unit of distance apart in one unit of time, starting and finishing at rest. The new objective function is to minimize the integral of the absolute value of the work done by the force acting on the block. It turns out that there is a simple analytic solution to this problem: Apply maximum force to get the block up to speed, then let the block coast, then apply maximum negative force to bring it to a stop at the target point. This solution is a so-called *bang-bang* control, and it is discontinuous at two points along the trajectory (switching from maximum force to zero force, and from zero force to maximum negative force). It is difficult to model a discontinuous solution with a polynomial spline, unless you know ahead of time where the discontinuities occur. In this section, we will study a few commonly used techniques for dealing with such discontinuities in the solution to a trajectory optimization problem.

7.10.1 Problem Statement

We would like to move a block along a one-dimensional friction-less surface, in a finite time, along a trajectory that minimizes the integral of the absolute work done by the control force u, where the position and velocity of the block are given by x and v respectively.

$$\min_{u(t),\, x_1(t),\, v(t)} \int_0^1 |u(\tau)\, v(\tau)|\, d\tau \tag{7.48}$$

We will assume that the block has unit mass and slides without friction, so we can write it's dynamics:

$$\dot{x} = v \qquad\qquad \dot{v} = u \tag{7.49}$$

Next, the block must start at the origin, and move one unit of distance in one unit of time. Note that the block must be stationary at both start and finish.

$$\begin{aligned} x(0) &= 0 & x(1) &= 1 \\ v(0) &= 0 & v(1) &= 0 \end{aligned} \tag{7.50}$$

Finally, we will assume that the force moving the block is bounded:

$$-u_{\max} \leq u(t) \leq u_{\max} \tag{7.51}$$

7.10.2 Analytic Solution

The analytic solution to this problem is given by:

$$u^*(t) = \begin{cases} u_{\max} & t < t^* \\ 0 & \text{otherwise} \\ -u_{\max} & (1 - t^*) < t \end{cases} \qquad \text{where} \qquad t^* = \frac{1}{2}\left(1 - \sqrt{1 - \frac{4}{u_{\max}}}\right) \tag{7.52}$$

The most important aspect of this solution to notice is that the control $u(t)$ is discontinuous. This means that that the linear and cubic spline control approximations used by the trapezoid and Hermite-Simpson collocation methods *cannot* perfectly represent this solution, although they can get arbitrarily close with enough mesh refinement.

Another interesting point is that there if $u_{\max} < 4$ then there is no feasible solution for the trajectory (because the duration is fixed). Finally, if there is no force limit ($u_{\max} \to \infty$) then the solution is impulsive (not just continuous, but a delta function).

7.10.3 Discontinuities

There are two types of discontinuities present in this problem. The first is obvious: the `abs()` function in the objective function (7.48). The second discontinuity is found in the solution (7.52) itself.

There are two ways to handle the discontinuity in the objective function, both of which we will cover here. The first is to re-write the `abs()` using slack variables, thus pushing the discontinuity to a constraint, which are easily handled by the non-linear programming solver. The second is to replace the `abs()` with a smooth approximation. Both methods work, although they have different implications for the convergence time and solution accuracy, as will be demonstrated in §7.10.7.

The discontinuity in the solution is a bit more subtle. Let's assume that we are stuck using the direct collocation methods from discussed in this paper, and that we do not know about the discontinuity in the solution ahead of time. One effect of a non-smooth solution is that the non-linear program will be slow to converge. Once we get a solution it will be clear that the is a discontinuity in the force trajectory (by looking at the plots). If you've done a good job of error analysis and re-meshing, then you should be able to automatically add points near the discontinuity, which will make the solution more accurate on the next iteration. It will never be perfect, but it will get fairly close.

7.10.4 Initialization

There are many ways to initialize this problem. In this tutorial we simply assume that the position of the block (x) transitions linearly between the initial and final position.

Then, we select the velocity and force profiles that are consistent with that initial guess.

$$x^{\text{init}}(t) = t \qquad v^{\text{init}}(t) = 1 \qquad u^{\text{init}}(t) = 0 \tag{7.53}$$

7.10.5 Slack Variables

The most "correct" way to rewrite the objective function (7.48) is using slack variables; it is mathematically identical to the original problem. The slack variable approach here is taken from [7]. There are a few downsides to using slack variables. The first is that the solution will still be discontinuous, and direct collocation cannot precisely represent it (although it can get arbitrarily close). Additionally, the addition of slack variables will greatly increase the size of the non-linear program: two additional controls and three additional constraints at every collocation point, for each `abs()`. Finally, the slack variables are implemented using a path constraint, which tends to cause the non-linear program to converge more slowly.

The key idea behind the slack variable approach is that you can push the discontinuity in the problem to a constraint, where the non-linear programming solver can properly handle it. We start by introducing two slack variables (s_1, s_2), and rewriting the objective function. Note that the slack variables here are to be treated as control variables for the purposes of transcription.

$$\min_{\substack{u(t),\, x(t),\, v(t) \\ s_1(t),\, s_2(t)}} \int_0^1 \left(s_1(\tau) + s_2(\tau)\right) d\tau \tag{7.54}$$

Next, we introduce a few constraints. The first require that the slack variables be positive:

$$0 \leq s_1(t) \qquad 0 \leq s_2(t) \tag{7.55}$$

Finally, we require that the difference between the slack variables is equal to the term

inside of the `abs()` function (7.48). Note that this is an example of a path constraint.

$$s_1(t) - s_2(t) = u(t)\, v(t) \tag{7.56}$$

This set of constraints (7.55,7.56) mean that $s_1(t)$ represents the positive part of the argument to the `abs()` function, while $s_2(t)$ represents the magnitude of the negative part.

The system dynamics, boundary constraints, and force limits remain unchanged. This modified version of the problem is now acceptable to pass into a non-linear programming solver. There are many possible ways to initialize the slack variables, but we've found the $s_1(t) = s_2(t) = 0$ is a good place to start.

The resulting non-linear program does not solve quickly, but the solver will eventually find a solution. The result will be the best possible trajectory, given the limitations of the spline approximation in the transcription method.

7.10.6 Smoothing

Although the slack variable method for representing `abs()` is exact, the resulting non-linear program can be complicated to construct and slow to solve. An alternative approach is to replace the `abs()` function with a smooth approximation. This method is simple to implement and solve, but at a loss of accuracy. Here we will discuss two possible smoothing solutions, both of which asymptotically approach $|x|$ as $x \to \infty$. Both of these approximations can be made arbitrarily accurate, with the problem becoming more difficult to solve with increasing accuracy.

Figure 7.17 shows two methods for smoothing `abs()`, one of which uses `sqrt()` and the other uses `tanh()`. The smooth approximation to `abs()` using the hyperbolic

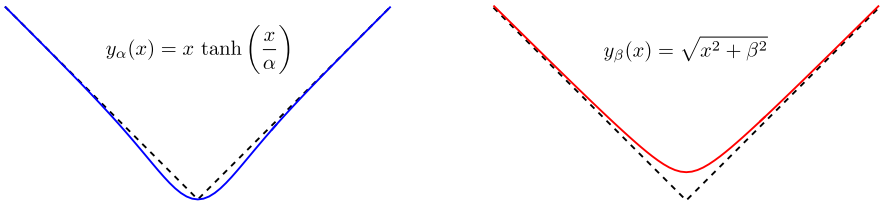

Figure 7.17: **Smoothing Comparison:** Two smooth approximations to the absolute value function, which have different properties. Both are non-negative, but the square-root smoothing is always greater than `abs(x)`, while the hyperbolic tangent smoothing is always less than `abs(x)`. Both approximations match the asymtotic behavior of `abs(x)`, while only the hyperbolic tangent matches its value at the origin.

tangent function, also known as exponential smoothing, is always less than $|x|$, while the approximation using the square-root function is always greater than $|x|$. Both are given below.

$$y_\alpha(x) = x \tanh\left(\frac{x}{\alpha}\right) \quad \approx |x| \tag{7.57}$$

$$y_\beta(x) = \sqrt{x^2 + \beta^2} \quad \approx |x| \tag{7.58}$$

The smoothing parameters α and β can be used to adjust the amount of smoothing on the problem, with the smooth versions of the functions approaching $|x|$ as $\alpha \to 0$ and $\beta \to 0$. The size of these smoothing parameters and choice of smoothing method are both problem dependent. In general, smaller values for the smoothing parameters make the non-linear program increasingly difficult to solve, but more representative of the original optimization problem.

One important thing to note is that smoothing fundamentally changes the problem, and not necessarily in an obvious way. For this reason, it is important to do convergence tests, solving the problem with successively smaller and smaller values for the smoothing parameter to ensure the correct solution is obtained. An example of this can be found in both [105] and [10].

7.10.7 Results

We solved this problem using the trapezoidal collocation method, and the FMINCON's [70] interior-point solver to solve the non-linear program. Although this problem appears simple, it is actually difficult to numerically solve without careful mesh refinement. To illustrate some trade-offs, we have solved the problem on three different meshes, using both slack variables and smoothing to handle the `abs()` function in the objective. Figure 7.7 shows the solution for each of these different set-ups, and compares each to the analytic solution. All solutions were obtained using the same solver settings and initialization, and the source code is included in the electronic supplement (Appendix §7.12).

One interesting thing to notice is that all of these solutions require a large number of iterations to solve the non-linear program, when compared to both the cart-pole swing-up problem and the five-link biped problem. This might seem odd, since the block-pushing problem looks like it should be easier. The difficulty, as best we can tell, comes from the discontinuity in the solution, which makes the resulting non-linear program challenging to solve.

The solution using slack variables (left column) converges to the correct solution, although it takes some time and a very fine mesh. The solution using light smoothing ($\alpha = 0.01$) is quite close to the solution obtained with slack variables, although the smooth version of the problem take more iterations (because the problem is stiff), and less time (because of the increase problem size for the slack variables). As the smoothing parameter is increased ($\alpha = 1.0$ and $\alpha = 5.0$), the solution is obtained faster, at a loss of accuracy.

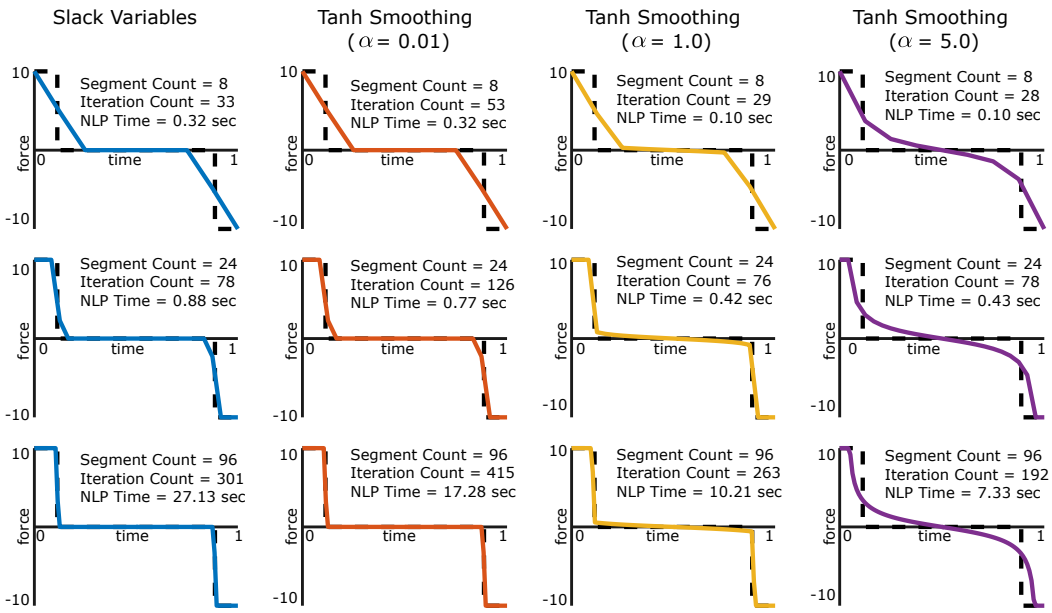

Figure 7.18: **Block Move Solution:** This figure shows a different solutions to the block moving problem. In each case, the analytic solution is given by a dashed black line, and the solid colored line gives the numerical solution using direct collocation. The left column shows the solution when the `abs()` in the objective function is handled with slack variables. The remaining columns show the result obtained using `tanh()` smoothing, for light smoothing ($\alpha = 0.01$), medium smoothing ($\alpha = 1.0$), and heavy smoothing ($\alpha = 5.0$). Notice that the solution obtained using slack variables and light smoothing are similar to each other, with the smoothing taking more iterations but less time. The problem solves even faster with medium and heavy smoothing, although the accuracy of the solution is degraded.

7.11 Summary

If we have been successful in this tutorial, you should be able to pose and solve a trajectory optimization problem on a computer, using your own direct collocation code. We've covered the trapezoidal collocation and the Hermite-Simpson collocation methods in detail. These methods are good general-purpose techniques for solving a wide range of problems, and they are actually used in practice, for example in aerospace [35, 6] (plane routes, and satellite orbits) and robotics [121, 122, 109].

We have taken care to include all equations that will be required to implement these direct collocation methods. In addition to the quadrature and collocation equations, this also includes formulas to interpolate polynomial splines, and estimate the error in a candidate solution.

This tutorial covers four example problems, showing how direct collocation methods can be applied to a specific application. The simple block moving example is a basic introduction to the field, and the cart-pole swing-up problem is a good problem that is moderately difficult but well behaved. The five-link biped problem shows a more challenging problem that can be applied to a real robot. The final example is a pathological case, a seemingly simple block-moving problem, which demonstrates how to work with problems that have discontinuities in both the problem statement and in the solution.

This paper comes with an electronic supplement, described in Appendix §7.12, which contains Matlab code to solve all examples presented here. Additionally, the electronic supplement includes a general-purpose trajectory optimization library, written by the author, that implements both trapezoidal and Hermite-Simpson collocation. It also includes a direct multiple shooting method (4^{th}-order Runge–Kutta), and a global orthogonal collocation method (Chebyshev Lobatto).

The techniques covered in this tutorial are just the beginning. We briefly mentioned some other techniques, such as direct multiple shooting and orthogonal collocation, both of which are also widely used in practice. The review paper by Betts [6] provides a good overview of more traditional methods for trajectory optimization, including direct and indirect shooting, as well as direct and indirect collocation. The survey paper by Rao [93] is also good, and includes more details about orthogonal collocation and pseudospectral methods.

The single best resource that we've found for learning about the fundamentals of trajectory optimization is the textbook by Betts [7]. It covers all aspects of direct collocation that have been discussed here, as well as many other related topics.

If you're interested in learning about high-order methods (orthogonal collocation), then I suggest reading about function approximation with orthogonal polynomials first. The textbook [111] and paper [5] by Trefethen are excellent resources for this. Then start reading about orthogonal collocation - the overview by Garg *et al* [33] is a good place to start, followed by the paper by Darby *et al* on adaptive meshing for othogonal collocation [20].

7.12 Overview of Electronic Supplementary Material

This tutorial has an electronic supplement that accompanies it. The supplement was written to go with this tutorial, and contains two parts. The first is a general purpose trajectory optimization library, written in Matlab, that solves the types of trajectory optimization of the type presented here. The second part of the supplement is a set of code that solves each of the example problems in this tutorial. There are a few other Matlab scripts, which can be used to derive some of the equations in the text, as well as

to generate some of the simple figures.

All of the source code in the electronic supplement is well documented, with the intention of making it easy to read and understand. Each directory in the supplement contains a `README` file that gives a summary of the contents.

7.12.1 Trajectory Optimization Code

This supplement includes a general-purpose Matlab library for solving trajectory optimization problems, written by the author. The source code is well-documented, such that it can be read as a direct supplement to this tutorial. This code is still under development, and the most up-to-date version is publicly available on GitHub:
`https://github.com/MatthewPeterKelly/TrajOpt`

The trajectory optimization code allows the user to choose from four different methods: trapezoidal direct collocation, Hermite-Simpson, direct collocation, 4^{th}-order Runge–Kutta direct multiple shooting, and Chebyshev orthogonal collocation (global lobatto method). The user can switch between methods by changing a single field in the options struct. Although this implementation does not perform automatic mesh refinement, it does allow the user to specify a mesh refinement schedule.

The solution is returned to the user at each grid-point along the trajectory. In addition, a function handle is provided to compute method-consistent interpolation for each component of the solution. Both direct collocation methods will also give the user an error estimate along the solution trajectory.

The trajectory optimization code includes its own set of example problems, which include some of the problems from this tutorial, along with a few others.

7.12.2 Example Problems

The electronic supplement includes a solution (in Matlab) to each of the four examples in this tutorial. Each example is in its own directory, and calls the same trajectory optimization code. Some example problems are implemented with many files, but the entry-point script always has the prefix `MAIN`. In some cases an additional script, with the prefix `RESULTS` is included, which is used to generate a figure in the tutorial.

Both the cart-pole and five-link biped examples make use of the Matlab symbolic toolbox to generate their equations of motion. These automatically generated files have the prefix `autoGen_`, and are created by a script with the prefix `Derive`.

7.13 Spline Derivations

All direct collocation methods are constructed by assuming that the system dynamics and control are well-approximated by polynomial splines. In this section, we will show one technique for deriving and representing these polynomial splines. We will consider a single segment of each type of spline, and then construct a system of linear equations from the boundary conditions prescribed by each method, given by [7]. Solving this system will generate the interpolation equations that we provide in the sections on trapezoidal and Hermite-Simpson collocation.

7.13.1 Notation

$u(t),\ x(t)$	**control and state spline equations, within a single segment**
$\dot{x}(t) = f(t)$	**time derivative of state spline equation (dynamics)**
$u_{\text{low}},\ x_{\text{low}},\ f_{\text{low}}$	**control, state, and dynamics value at lower boundary**
$u_{\text{mid}},\ x_{\text{mid}},\ f_{\text{mid}}$	**control, state, and dynamics value at mid-point**
$u_{\text{upp}},\ x_{\text{upp}},\ f_{\text{upp}}$	**control, state, and dynamics value at upper boundary**
c_i	**spline coefficients within a segment**
h	**duration of the spline segment**

Throughout this section we will assume that the time variable has been shifted, such that the time at the beginning of the segment is $t_{\text{low}} = 0$, the time at the mid-point of the segment is $t_{\text{mid}} = h/2$, and the time at the end of the segment is $t_{\text{upp}} = h$. The equations constructed throughout this section are valid on the domain $t \in [0, h]$.

7.13.2 Trapezoidal Collocation: Control

The control spline for trapezoidal collocation is piece-wise linear, defined by the value of the spline at both end-points[7].

$$u(t) = c_0 + c_1 t \tag{7.59}$$

The boundary conditions can be written as a system of linear equations, which can be symbolically solved for the coefficients c_0 and c_1.

$$\begin{bmatrix} u(0) \\ u(h) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & h \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \end{bmatrix} = \begin{bmatrix} u_{\text{low}} \\ u_{\text{upp}} \end{bmatrix} \tag{7.60}$$

7.13.3 Trapezoidal Collocation: State

The state spline for trapezoidal collocation is piece-wise quadratic. The coefficients are constructed by choosing the slope at each end-point, and the value at the first end-point. While it may seem odd, this choice of boundary conditions is not random - it follows directly from the assumption that the system dynamics (the derivative of the state spline) are piece-wise linear. [7]

$$x(t) = c_0 + c_1 t + c_2 t^2 \qquad \dot{x}(t) = f(t) = c_1 + 2c_2 t \tag{7.61}$$

These boundary conditions can be written as a system of linear equations, which can be symbolically solved for the coefficients c_0, c_1, and c_2.

$$\begin{bmatrix} x(0) \\ f(0) \\ f(h) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 2h \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} x_{\text{low}} \\ f_{\text{low}} \\ f_{\text{upp}} \end{bmatrix} \tag{7.62}$$

7.13.4 Hermite-Simpson Collocation: Control

The state spline for Hermite-Simpson collocation is piece-wise quadratic, defined by the value of the spline at both end-points and the mid-point [7]. Notice that the construction of this spline is different than the quadratic spline used for the state in trapezoidal collocation.

$$u(t) = c_0 + c_1 t + c_2 t^2 \tag{7.63}$$

These boundary conditions can be written as a system of linear equations, which can be symbolically solved for the coefficients c_0, c_1, and c_2.

$$\begin{bmatrix} u(0) \\ u(\frac{h}{2}) \\ u(h) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & \frac{h}{2} & \frac{h^2}{4} \\ 1 & h & h^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} u_{\text{low}} \\ u_{\text{mid}} \\ u_{\text{upp}} \end{bmatrix} \tag{7.64}$$

7.13.5 Hermite-Simpson: State

The state spline for the Hermite-Simpson method is piece-wise cubic.The coefficients are constructed using the the slope at each end-point and the mid-point, as well as the value at the first end-point. This interpolation scheme is selected such that it is consistent with a piece-wise quadratic approximation of the dynamics at the collocation points.

$$x(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 \qquad \dot{x}(t) = f(t) = c_1 + 2c_2 t + 3c_3 t^2 \tag{7.65}$$

These boundary conditions can be written as a system of linear equations, which can be symbolically solved for the coefficients c_0, c_1, c_2, and c_3.

$$\begin{bmatrix} x(0) \\ f(0) \\ f(\frac{h}{2}) \\ f(h) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & h & \frac{3}{4}h^2 \\ 0 & 1 & 2h & 3h^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} x_{\text{low}} \\ f_{\text{low}} \\ f_{\text{mid}} \\ f_{\text{upp}} \end{bmatrix} \tag{7.66}$$

7.13.6 Hermite-Simpson: Collocation Equations

In this section we will cover one way to derive the Hermite-Simpson collocation equations (7.15) and (7.16). We will start by assuming that the state trajectory is approximated by a cubic polynomial on the segment of interest.

$$x(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 \qquad \dot{x}(t) = f(t) = c_1 + 2c_2 t + 3c_3 t^2 \tag{7.67}$$

Next, we will solve for the coefficients in that polynomial by assuming that the state and dynamics are given at the endpoints of the interval.

$$\begin{bmatrix} x(0) \\ f(0) \\ x(h) \\ f(h) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & h & h^2 & h^3 \\ 0 & 1 & 2h & 3h^2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} x_{\text{low}} \\ f_{\text{low}} \\ x_{\text{upp}} \\ f_{\text{upp}} \end{bmatrix} \tag{7.68}$$

We can solve this system of linear equations (7.68) for the coefficients c_0, c_1, c_2, and c_3, to give an expression for $x(t)$ and $\dot{x}(t)$ in terms of time and the known boundary conditions. Then we will evaluate these expressions at the mid-point of the interval.

$$x_{\text{mid}} = x(\tfrac{h}{2}) = \frac{1}{2}(x_{\text{low}} + x_{\text{upp}}) + \frac{h}{8}(f_{\text{low}} - f_{\text{upp}}) \tag{7.69}$$

$$f_{\text{mid}} = f(\tfrac{h}{2}) = \frac{1}{4h}(6x_{\text{upp}} - 6x_{\text{low}}) + \frac{1}{4}(f_{\text{low}} + f_{\text{upp}}) \tag{7.70}$$

We're done! Equation (7.69) is an exact match for the Hermite interpolant constraint (7.15) , and a bit of simple algebra will show that (7.70) is the same as the Simpson quadrature constraint (7.16), both from §7.5.

7.14 Orthogonal Polynomials

Orthogonal polynomials provide a fast an accurate means of function approximation. In the same way that direct collocation methods are derived by assuming that the system dynamics and control are polynomial splines, orthogonal collocation methods are derived by assuming that the dynamics and control are orthogonal polynomial splines.

The basic idea behind function approximation with orthogonal polynomials is that any function can be represented by an infinite sum of basis functions. The Fourier Series is one well known example, where you can represent an arbitrary function by an infinite sum of sine and cosine functions. It turns out that if the function of interest is smooth, as is often the case in trajectory optimization, then orthogonal polynomials make an excellent choice of basis function. There are many papers that cover the detailed mathematics of orthogonal polynomials [38, 83, 5, 65, 111, 43] and their use in trajectory optimization[46, 114, 24, 48, 23, 96, 4, 110, 32, 33, 19, 30], but here we will focus on the practical implementation details.

Let's start by assuming that we have some function $f(t)$ that we would like to approximate over the interval $[-1, 1]$. We can do this using *barycentric interpolation*: representing the function's value at any point on the interval by a convex combination of its value at several carefully chosen *interpolation (grid) points*. We will write these points as t_i and the value of the function at these points as f_i. The set of points t_i can then be used to compute a set of interpolation weights v_i, and quadrature weights w_i, and a differentiation matrix D. If the points t_i are carefully chosen to be the roots of an orthogonal polynomial, and the function $f(t)$ is smooth, then the resulting interpolation, integration, and differentiation schemes tend to be both accurate and easy to compute. Other distributions of points do not given nice results. For example, choosing t_i to be uniformly spaced over the interval will result in numerically unstable schemes [5].

Orthogonal collocation techniques for trajectory optimization make extensive use of these properties of orthogonal polynomials. In particular, the differentiation matrix can be used to construct a set of collocation constraints to enforce the dynamics of a system, the quadrature weights can be used to accurately approximate an integral cost function or constraints, and the barycentric interpolation is used to evaluate the solution.

For the rest of this section we will assume that the function of interest has been mapped to the interval $t \in [-1, 1]$. If the function is initially defined on the interval $\tau \in [\tau_A, \tau_B]$, this mapping can be achieved by:

$$t = 2\frac{\tau - \tau_A}{\tau_B - \tau_A} - 1 \tag{7.71}$$

7.14.1 Computing Polynomial Roots

An orthogonal polynomial approximation can be defined by the value of the function $f(t)$ at the roots t_i of that orthogonal polynomial . There are many different orthogonal

polynomials to choose from, each of which has slightly different properties. The ChebFun [21] library for Matlab provides subroutines for computing the interpolation points t_i, interpolation weights v_i, and quadrature weights w_i for most common orthogonal polynomials.

The Chebyshev orthogonal polynomials are one popular choice, in part because their roots are easy to compute. The Chebyshev-Lobatto points, also called the Chebyshev points of the second kind, are given by [111].

$$t_i = \cos(i\pi/n), \qquad 0 \leq i \leq n \tag{7.72}$$

The Legendre orthogonal polynomials are also commonly used. Unlike the Chebyshev polynomials, the roots of the Legendre polynomials have no closed-form solution, and must be numerically computed. The methods for computing these points are given by [38, 43], although various sub-routines can be found with a quick internet search. ChebFun [21] has a particularly good implementation for Matlab.

There are three commonly used sets of Legendre points. The *Legendre-Gauss* points are given by the roots of the $P_n(t)$, the n^{th}-degree Legendre polynomial. The *Legendre-Gauss-Radau* points are given by the roots of $P_n(t) + P_{n-1}(t)$. Finally, the *Legendre-Gauss-Lobatto* points are given by the roots of $\dot{P}_{n-1}(t)$ along with the boundary points -1 and 1 [33].

The important distinction between these three sets of points are whether or not the end-points of the interval are included in a given set of points. Orthogonal collocation schemes can be constructed from any of these sets of points, although they will have different properties [33]. Here we have outlined these points for the Legendre polynomials, but the naming convention (Gauss, Radau, and Lobatto) apply to any orthogonal polynomial. Figure 7.19 shows an illustration of the Gauss, Radau, and Lobatto points

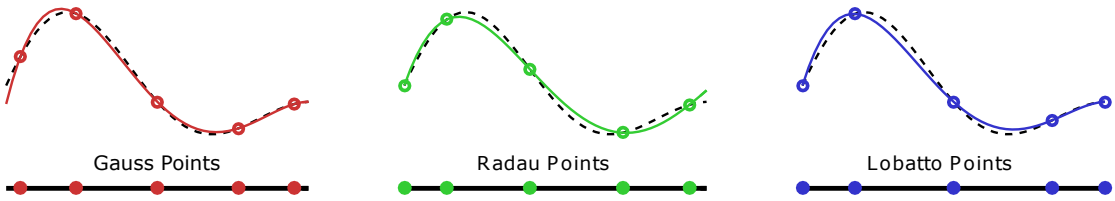

Figure 7.19: **Types of Collocation Points:** There are three types of collocation methods, classified as either Gauss, Radau, or Lobatto methods, based on whether one, both, or neither end-points are collocation points. In this figure we have shown the Gauss, Radau, and Lobatto points for the 4^{th}-order Legendre orthogonal polynomials. The dashed line in each figure is the same, and the solid lines show the barycentric interpolant that is defined by that set of collocation points. Notice that the interpolant behaves differently for each set of points.

for the Legendre orthogonal polynomials.

Collocation methods whose collocation points include both endpoints of a segment are called *Lobatto* methods. Two popular Lobatto methods are the trapezoidal collocation and Hermite-Simpson collocation methods covered in this paper. [7] A high-order Lobatto method based on Chebysehv orthogonal polynomials is described in [23].

A *Gauss* method is one where the neither endpoint of the segment is a collocation point. A common low-order example would be the implicit mid-point method. A high-order Gauss method based on Legendre orthogonal polynomials is described in [42, 30].

Finally, a *Radau* method is one where a single endpoint of each segment is a collocation point, such as the backward Euler Method. The trajectory optimization software GPOPS [84], uses a high-order Radau method, based on Legendre orthogonal polynomials. These three types of methods are discussed in more detail in [32, 33], and are illustrated in 7.19.

Garg *et al* [33] suggest that high-order Lobatto collocation schemes should be avoided in trajectory optimization, due to poor numerical properties, and that schemes based on Radau and Gauss points should be preferred.

7.14.2 Barycentric Lagrange Interpolation

The best way to store and evaluate high-order orthogonal polynomials is using barycentric Lagrange interpolation. This is basically expressing the value of the function at any point with a weighted combination of the function's value (f_i) at the roots of the orthogonal polynomial (t_i). The equation for this barycentric interpolation is given below, with further details in [5]. Note that when this expression is not valid when evaluated at the interpolation points $t = t_i$. This provides no problem, since the value of the function at these points is already known to be f_i.

$$f(t) = \frac{\sum_{i=0}^{n} \frac{v_i}{t - t_i} f_i}{\sum_{i=0}^{n} \frac{v_i}{t - t_i}} \tag{7.73}$$

Thus far, we know all parameters in (7.73), except for the interpolation weights v_i. These weights are calculated below, using the equation given by [5].

$$v_i = \frac{1}{\prod_{j \neq i}(t_i - t_j)}, \qquad i = 0, \ldots, n \tag{7.74}$$

Interestingly, the barycentric interpolation formula (7.73) will still interpolate the data at points f_i if the weights v_i are chosen arbitrarily. The choice of weights given by (7.74) is special in that it defines a *polynomial* interpolant, where other any other choice of weights will result in interpolation by some rational function [5].

Notice that these weights can be scaled by an arbitrary constant, and still produce the correct interpolation in (7.73), as well as the correct differentiation matrix (7.76). For example, ChebFun [21] normalizes the barycentric weights such that the magnitude of the largest weight is 1.

In an orthogonal collocation method, barycentric interpolation would be used to evaluate the solution. It is not used when constructing the non-linear program; the

decision variables of the non-linear program are the values of the state and control at each collocation point t_i.

7.14.3 Differentiation Matrix

Another useful property of orthogonal polynomials is that they are easy to differentiate. Let's define a column vector $\boldsymbol{f} = [f_0,\ f_1,\ ...\ f_n]^T$, which contains the value of $f()$ at each interpolation point t_i. It turns out that we can find some matrix $\mathcal{D}$ that can be used to compute the derivative of $f()$ at each interpolation point (7.75). Additionally, we can use the same interpolation weights v_i for interpolation of this derivative.

$$\dot{\boldsymbol{f}} = \mathcal{D}\boldsymbol{f} \tag{7.75}$$

Each element of the differentiation matrix $\mathcal{D}$ can be computed as shown below, using a formula from [5].

$$\mathcal{D}_{ij} = \begin{cases} \dfrac{v_j / v_i}{t_i - t_j} & i \neq j \\ -\displaystyle\sum_{i \neq j} \mathcal{D}_{ij} & i = j \end{cases} \tag{7.76}$$

7.14.4 Quadrature

Each type of orthogonal polynomial has a corresponding quadrature rule to compute its definite integral. In orthogonal collocation, these quadrature rules are used to evaluate integral constraints and objective functions. The quadrature rule is computed as shown below, and is a linear combination of the function value at each interpolation point (t_i).

$$\int_{-1}^{1} f(\tau)\, d\tau \approx \sum_{i=0}^{n} w_i \cdot f_i \tag{7.77}$$

Typically these quadrature weights (w_i) are computed at the same time as the interpolation points (t_i) and weights (v_i). Alternatively, the quadrature weights can be determined directly from the interpolation points and weights, although the equations are specific to each type of orthogonal polynomial. For example, the Legendre-Gauss quadrature weights can be computed by (7.78), and the Legendre-Gauss-Lobatto weights can be computed by (7.79).

$$w_i = W \frac{v_i^2}{(1 - t_i^2)} \qquad \textbf{Legendre-Gauss} \tag{7.78}$$

$$w_i = W v_i^2 \qquad \textbf{Legendre-Gauss-Lobatto} \tag{7.79}$$

In both cases the scaling constant W should be selected such that $\sum w_i = 2$. This scaling can be derived by computing the integral of unity:

$$\int_{-1}^{1} d\tau = 2 = \sum_{i=0}^{n} w_i \tag{7.80}$$

More details on the calculation of quadrature rules can be found in [111, 29, 59, 120].

7.15 Parameters for Example Problems

This appendix gives parameter values that were used when computing the results for the both the cart-pole swing-up example problem and the five-link biped example problem.

7.15.1 Cart-Pole Swing-Up Parameters

We choose parameters, shown in Table 7.1, for our model to match something like you might see in a cart-pole in a controls lab demonstration.

7.15.2 Five-Link Biped Parameters

We choose parameters for our model to match the robot RABBIT[121, 14] which are reproduced here in Table 7.2. We also selected a trajectory duration of $T = 0.7s$ and a step length of $D = 0.5m$.

Table 7.1: Physical Parameters: Cart-Pole

Symbol	Value	Name
m_1	1.0 kg	mass of cart
m_2	0.3 kg	mass of pole
ℓ	0.5 m	pole length
g	9.81 m/s^2	gravity acceleration
$u_{\max}$	20 N	maximum actuator force
$d_{\max}$	2.0 m	extents of the rail that cart travels on
d	1.0 m	distance traveled during swing-up
T	2.0 s	duration of swing-up

Table 7.2: Physical Parameters: Five Link Biped model (RABBIT) [14]

Symbol	Value	Name
m_1, m_5	3.2 kg	mass of tibia (lower leg)
m_2, m_4	6.8 kg	mass of femur (upper leg)
m_3	20 kg	mass of torso
I_1, I_5	0.93 kg-m^2	rotational inertia of tibia, about its center of mass
I_2, I_4	1.08 kg-m^2	rotational inertia of femur, about its center of mass
I_3	2.22 kg-m^2	rotational inertia of torso, about its center of mass
ℓ_1, ℓ_5	0.4 m	length of tibia
ℓ_2, ℓ_4	0.4 m	length of femur
ℓ_3	0.625 m	length of torso
d_1, d_5	0.128 m	distance from tibia center of mass to knee
d_2, d_4	0.163 m	distance from femur center of mass to hip
d_3	0.2 m	distance from torso center of mass to hip

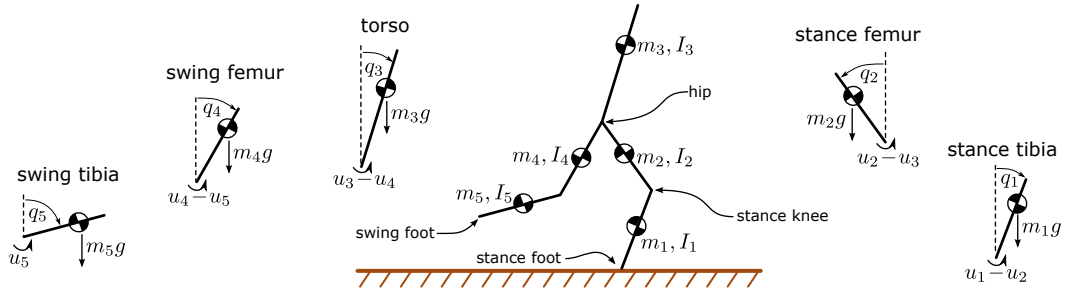

Figure 7.20: **Five Link Biped Model:** Shown here in single stance. We assume that the dynamics are planar (2D) and modeled as a kinematic chain, with each link assigned a number: 1 = stance tibia, 2 = stance femur, 3 = torso, 4 = swing femur, and 5 = swing tibia. Each joint is connected to its parent by an ideal revolute joint and torque source. Joint torques are given by u_i, link masses and interias by m_i and I_i, and gravity is g. The absolute orientation of each link is given by q_i.

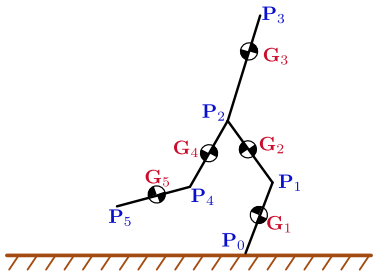

Figure 7.21: **Five Link Biped Single-Stance Kinematics:** This figure shows various positions of interest ($\boldsymbol{P}_i$) and the center of mass of each link ($\boldsymbol{G}_i$) during single stance.

7.16 Biped Dynamics

In this section, we will cover some of the more detailed calculations for the five-link biped model of walking, including the kinematics, single stance dynamics, heel-strike dynamics, and gradients. We will assume that the reader has a solid understanding of the dynamics of rigid body mechanisms, as well as experience deriving equations of motion using a symbolic algebra computer package, such as the Matlab Symbolic Toolbox [71].

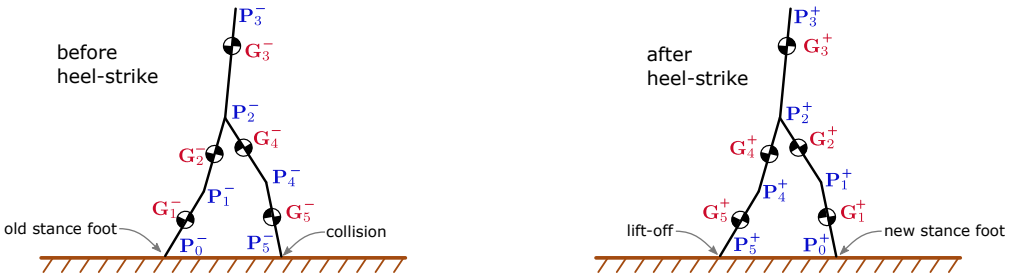

Figure 7.22: **Five Link Biped Heel-Strike Kinematics:** This figure shows various positions of interest ($\boldsymbol{P}_i$) and the center of mass of each link ($\boldsymbol{G}_i$), both before ($^-$) and after ($^+$) heel-strike. Note that the points on the robot are re-labeled during the collision, reflecting the left-right symmetry of the robot.

7.16.1 Kinematics

Let's start by defining the position vectors $\boldsymbol{P}_i$ and $\boldsymbol{G}_i$ that point from the origin ($\boldsymbol{P}_0$) to various points of interest and the center of mass of each link, as shown in Figure 7.21. All of these vectors should be functions of the configuration of the robot: $\boldsymbol{P}_i = \boldsymbol{P}_i(\boldsymbol{q})$, where $\boldsymbol{q} = [q_1 \; q_2 \; q_3 \; q_4 \; q_5]^T$ is a column vector of absolute joint orientations. We will define $\boldsymbol{P}_0 = \boldsymbol{0}$.

Next, we will need to compute the velocity $\dot{\boldsymbol{G}}_i$ and accelerations $\ddot{\boldsymbol{G}}_i$ of center of mass of each link. The chain rule can be applied here. Define a vector of absolute link angular rates: $\dot{\boldsymbol{q}} = [\dot{q}_1 \; \dot{q}_2 \; \dot{q}_3 \; \dot{q}_4 \; \dot{q}_5]^T$, and then the velocity of the center of mass of each link is given by:

$$\dot{\boldsymbol{G}}_i = \left(\frac{\delta \boldsymbol{G}_i}{\delta \boldsymbol{q}} \right) \dot{\boldsymbol{q}} \tag{7.81}$$

The calculation for the acceleration vectors ($\ddot{\boldsymbol{G}}_i$) is carried out in a similar fashion, although we need to include the joint rates in the list of partial derivatives. We can do this by defining: $\boldsymbol{z} = [\boldsymbol{q} \; \dot{\boldsymbol{q}}]^T$ and $\dot{\boldsymbol{z}} = [\dot{\boldsymbol{q}} \; \ddot{\boldsymbol{q}}]^T$, where $\ddot{\boldsymbol{q}} = [\ddot{q}_1 \; \ddot{q}_2 \; \ddot{q}_3 \; \ddot{q}_4 \; \ddot{q}_5]^T$.

$$\ddot{\boldsymbol{G}}_i = \left(\frac{\delta \dot{\boldsymbol{G}}_i}{\delta \boldsymbol{z}} \right) \dot{\boldsymbol{z}} \tag{7.82}$$

Both of these derivative calculations can be implemented in Matlab with the following

commands:

```
>> dG = jacobian(G,q)*dq;
>> ddG = jacobian(dG,[q; dq])*[dq; ddq];
```

7.16.2 Single-Stance Dynamics

There are many methods to compute the minimal coordinate dynamics for kinematic chain like this five-link biped robot. Here we will show one method using the Newton-Euler equations. Although it is possible to derive such equations by hand, we present them here in a form that is convenient for use with a computer algebra package, such as the Matlab Symbolic Toolbox[71].

The goal of the dynamics calculations are to arrive at a set of equations that relate the joint accelerations $\ddot{\boldsymbol{q}}$, rates $\dot{\boldsymbol{q}}$, angles $\boldsymbol{q}$, and torques $\boldsymbol{u} = [u_1\ u_2\ u_3\ u_4\ u_5]^T$. The second-order dynamics $\ddot{\boldsymbol{q}}$ of the system can then be solved at run-time by numerically solving for the angular acceleration of each joint. Even though it is technically possible to symbolically solve for these accelerations, the resulting expressions take much longer to evaluate when compared to numerically solving the linear system.

$$\boldsymbol{\mathcal{M}}(\boldsymbol{q}) \cdot \ddot{\boldsymbol{q}} = \boldsymbol{\mathcal{F}}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{u}) \tag{7.83}$$

For our five-link biped, there are five linearly independent equations required to construct (7.83), one for each degree of freedom. One way to construct such a system is to write out the equations for angular momentum balance about each successive joint in the robot. Starting at the root joint (stance foot) and working out. We start with angular momentum balance of the entire robot about the stance foot joint (below). Note that the left side of the equation is a sum over all external torques applied to the system about

point $\boldsymbol{P}_0$, the stance foot. The right side of the equation gives the time rate of change in the angular momentum of the system about $\boldsymbol{P}_0$.

$$u_1 + \hat{\boldsymbol{k}} \cdot \sum_{i=1}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_0) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=1}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_0) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \right) \quad (7.84)$$

The next equation is obtained by simply moving one joint out along the robot, computing the angular momentum balance about the stance knee.

$$u_2 + \hat{\boldsymbol{k}} \cdot \sum_{i=2}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_1) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=2}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_1) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \right) \quad (7.85)$$

The remaining three equations are given below, following a similar pattern. Notice that the pattern slightly breaks down at the hip joint, because link 3 and link 4 are both connected to the hip joint $\boldsymbol{P}_2$.

$$u_3 + \hat{\boldsymbol{k}} \cdot \sum_{i=3}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_2) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=3}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_2) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \right) \quad (7.86)$$

$$u_4 + \hat{\boldsymbol{k}} \cdot \sum_{i=4}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_2) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=4}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_2) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \right) \quad (7.87)$$

$$u_5 + \hat{\boldsymbol{k}} \cdot \sum_{i=5}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_4) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=5}^{5} \left((\boldsymbol{G}_i - \boldsymbol{P}_4) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \right) \quad (7.88)$$

7.16.3 Heel-Strike Dynamics

For our biped walking model, we will assume that the biped transitions directly from single stance on one foot to single stance on the other. In other words, as soon as the leading foot strikes the ground, the trailing foot leaves the ground. This transition is known as a heel-strike map. We will also make the assumptions that this transition occurs instantaneously, and that the robot is symmetric.

There are two parts to the heel-strike map. The first is an impulsive collision, which changes the joint velocities throughout the robot. The second part is to switch the swing

and stance legs, so that we can exploit the symmetry of the problem, allowing us to optimize over a single step, rather than a pair of steps.

Figure 7.22 shows the biped model immediately before and after the heel-strike map. Notice that the old swing foot $\boldsymbol{P}_0^-$, has become the new stance foot $\boldsymbol{P}_5^+$ after the map. Similar re-naming has been applied throughout the robot. Here is the expression for the mapping between the old and new joint angles:

$$\boldsymbol{q}^+ = \begin{bmatrix} 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 \end{bmatrix} \boldsymbol{q}^- \tag{7.89}$$

The joint angle rates, however, are a bit more tricky. Like the joint accelerations in the single-stance dynamics, we will derive a linear equation that describes the map, and then solve it numerically at run time.

$$\boldsymbol{\mathcal{M}}_H(\boldsymbol{q}^-) \cdot \dot{\boldsymbol{q}}^+ = \boldsymbol{\mathcal{F}}_H(\boldsymbol{q}^-, \dot{\boldsymbol{q}}^-) \tag{7.90}$$

One way to derive this system of equations is to observe that the system must conserve angular momentum about the collision point, as well as all joints in the robot. The five equations defining the system are given below. Notice that the left side of each equation is the angular momentum of the entire system before heel-strike, about the swing foot (which is about to become the new stance foot). The right side of each equation is the angular momentum of the entire system after heel-strike about the stance foot (which was previously the swing foot). Figure 7.22 shows the naming conventions used throughout these equations.

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{5} \left((\boldsymbol{G}_i^- - \boldsymbol{P}_5^-) \times (m_i \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=1}^{5} \left((\boldsymbol{G}_i^+ - \boldsymbol{P}_0^+) \times (m_i \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \right) \tag{7.91}$$

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{4} \left((\boldsymbol{G}_i^- - \boldsymbol{P}_4^-) \times (m_i \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=2}^{5} \left((\boldsymbol{G}_i^+ - \boldsymbol{P}_1^+) \times (m_i \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \right) \tag{7.92}$$

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{3} \left((\boldsymbol{G}_i^- - \boldsymbol{P}_2^-) \times (m_i \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=3}^{5} \left((\boldsymbol{G}_i^+ - \boldsymbol{P}_2^+) \times (m_i \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \right) \quad (7.93)$$

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{2} \left((\boldsymbol{G}_i^- - \boldsymbol{P}_2^-) \times (m_i \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=4}^{5} \left((\boldsymbol{G}_i^+ - \boldsymbol{P}_2^+) \times (m_i \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \right) \quad (7.94)$$

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{1} \left((\boldsymbol{G}_i^- - \boldsymbol{P}_1^-) \times (m_i \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \right) = \hat{\boldsymbol{k}} \cdot \sum_{i=5}^{5} \left((\boldsymbol{G}_i^+ - \boldsymbol{P}_4^+) \times (m_i \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \right) \quad (7.95)$$

Although these equations (7.91-7.95) look complicated, if you write them out in terms of individual states and parameters, you will find that they are linear in the angular rate after heel-strike ($\dot{\boldsymbol{q}}^+$). It is this property that makes it easy to numerically compute $\dot{\boldsymbol{q}}^+$.

We can combine (7.89) and (7.90) into a single heel-strike map equation, where $\boldsymbol{x}^-$ is the state of the system before heel-strike and $\boldsymbol{x}^+$ is the state after heel-strike.

$$\boldsymbol{x}^- = \begin{bmatrix} \boldsymbol{q}^- \\ \dot{\boldsymbol{q}}^- \end{bmatrix} \qquad \boldsymbol{x}^+ = \begin{bmatrix} \boldsymbol{q}^+ \\ \dot{\boldsymbol{q}}^+ \end{bmatrix} \quad (7.96)$$

$$\boldsymbol{x}^+ = \boldsymbol{f}_H(\boldsymbol{x}^-) \quad (7.97)$$

7.16.4 Gradients

For trajectory optimization, it is generally a good idea to use analytic gradients where possible. This means that at some point we will need to calculate the expressions:

$$\frac{\delta \ddot{\boldsymbol{q}}}{\delta \boldsymbol{q}} \qquad \frac{\delta \ddot{\boldsymbol{q}}}{\delta \dot{\boldsymbol{q}}} \qquad \frac{\delta \ddot{\boldsymbol{q}}}{\delta \boldsymbol{u}} \qquad \frac{\delta \dot{\boldsymbol{q}}^+}{\delta \boldsymbol{q}^-} \qquad \frac{\delta \dot{\boldsymbol{q}}^+}{\delta \dot{\boldsymbol{q}}^-} \quad (7.98)$$

Unfortunately, we can't just use the `jacobian` command in the symbolic software, because we plan to calculate $\ddot{\boldsymbol{q}}$ and $\dot{\boldsymbol{q}}^+$ by numerically solving a linear system at run time. The solution is to use the symbolic software to compute the gradients of $\boldsymbol{\mathcal{M}}$, $\boldsymbol{\mathcal{F}}$, $\boldsymbol{\mathcal{M}}_H$,

and $\mathcal{F}_H$ and then derive an expression for the gradient of $\ddot{\boldsymbol{q}}$ and $\dot{\boldsymbol{q}}^+$ in terms of these known matrices. The derivation is straight-forward:

$$\boldsymbol{\mathcal{M}}^{-1}\boldsymbol{\mathcal{M}} = \boldsymbol{\mathcal{I}} \tag{7.99}$$

$$\frac{\delta}{\delta q_i}\left(\boldsymbol{\mathcal{M}}^{-1}\boldsymbol{\mathcal{M}}\right) = \mathbf{0} \tag{7.100}$$

$$\frac{\delta}{\delta q_i}\left(\boldsymbol{\mathcal{M}}^{-1}\right)\boldsymbol{\mathcal{M}} + \boldsymbol{\mathcal{M}}^{-1}\frac{\delta}{\delta q_i}(\boldsymbol{\mathcal{M}}) = \mathbf{0} \tag{7.101}$$

$$\frac{\delta \boldsymbol{\mathcal{M}}^{-1}}{\delta q_i} = -\boldsymbol{\mathcal{M}}^{-1}\frac{\delta \boldsymbol{\mathcal{M}}}{\delta q_i}\boldsymbol{\mathcal{M}}^{-1} \tag{7.102}$$

We will now apply (7.102) to compute gradient of the link accelerations ($\ddot{\boldsymbol{q}}$) with respect to a single link angle (q_i). This process can then be repeated for the remaining joint angles, rates ($\dot{q}_i$), and torques (u_i). The same process can also be used for the heel-stike equations.

$$\frac{\delta \ddot{\boldsymbol{q}}}{\delta q_i} = \frac{\delta}{\delta q_i}\left(\boldsymbol{\mathcal{M}}^{-1}\boldsymbol{\mathcal{F}}\right) \tag{7.103}$$

$$\frac{\delta \ddot{\boldsymbol{q}}}{\delta q_i} = \left(-\boldsymbol{\mathcal{M}}^{-1}\frac{\delta \boldsymbol{\mathcal{M}}}{\delta q_i}\boldsymbol{\mathcal{M}}^{-1}\right)\boldsymbol{\mathcal{F}} + \boldsymbol{\mathcal{M}}^{-1}\left(\frac{\delta \boldsymbol{\mathcal{F}}}{\delta q_i}\right) \tag{7.104}$$

$$\frac{\delta \ddot{\boldsymbol{q}}}{\delta q_i} = \boldsymbol{\mathcal{M}}^{-1}\left(-\frac{\delta \boldsymbol{\mathcal{M}}}{\delta q_i}\ddot{\boldsymbol{q}} + \frac{\delta \boldsymbol{\mathcal{F}}}{\delta q_i}\right) \tag{7.105}$$

CHAPTER 8

DIRCOL5I: A METHOD FOR COMPUTING MINIMUM-SNAP TRAJECTORIES FOR SYSTEMS WITH NON-TRIVIAL DYNAMICS

This chapter will be submitted to a peer reviewed journal, after minor modifications.

Author List: Matthew Kelly.

8.1 Abstract

Minimum-snap and minimum-jerk trajectories are widely used in robotics, particularly in quad-rotor helicopters and robot arms. The traditional algorithms for computing these trajectories do not consider the system dynamics. Rather, they compute a kinematic trajectory, and assume that the system can follow it. There are some dynamical systems where this methodology will fail, generally due to some type of under-actuation in the system. In this paper we present DirCol5i, a direct collocation method that is specialized for solving trajectory optimization problems with non-trivial dynamics and objective functions that include derivatives of the state and control. Although these problems can be solved using standard direct collocation methods, the resulting formulation is difficult to implement. DirCol5i has two significant advantages over standard trajectory optimization techniques, for this class of problem. The first is that is make setting up the problem simple, and the second is that it tends to be more numerically robust. An additional feature of DirCol5i is that the system dynamics can be represented implicitly in second-order form, another feature that is not common in standard trajectory optimization methods.

8.2 Introduction

Trajectory optimization is widely used in robotics, typically as a way to compute a desirable motion for the robotic system. For example, a walking robot might use trajectory optimization to determine how to move throughout a step [121], or to plan out its motion over several steps [18]. Trajectory optimization is also used to plan flight paths for quadrotors [74, 77, 94] and robot arms [34, 101].

There are three parts to any trajectory optimization problem: 1) the system dynamics, 2) an objective function, and 3) a set of constraints. The system dynamics describe how the system changes with time, the objective function encodes the desired features of the motion, and the constraints can be used to apply limitations to the motion.

For some robotic systems, particularly under-actuated robots [108], the trajectory optimization problem must include the full non-linear dynamics model for the robot. If the full dynamics are included in the problem, then it is difficult to have higher-derivatives (*eg.* jerk and snap) in the objective function, as discussed in §8.6. To avoid these problems, many algorithms solve only for a kinematics trajectory, neglecting the dynamics, or approximating the dynamics by carefully chosen kinematic constraints. By treating only the kinematics of a system, it is simple to find minimal-jerk and minimal-snap trajectories. This technique is commonly used for quadrotor and robot arm trajectory generation.

In this paper we will focus on computing optimal trajectories for systems where the system dynamics cannot be neglected, but where we wish to use objective functions that include higher-derivatives of the state (and control). Standard trajectory optimization methods are inadequate for solving this type of problem, since it would require constructing a complicated set of chain integrators. The non-linear program that re-

sults from this formulation tends to be difficult to implement and initialize, an has poor numerical properties.

DirCol5i, the method presented in this paper, uses 5^{th}-order splines to represent the position trajectory for the system. The higher-order derivatives are computed by analytic differentiation of this spline, which avoids many of the numerical difficulties encountered when using standard methods to solve these problems. One of the side-effects of this process is that the system dynamics can be expressed implicitly, in either first-order or second-order form.

We compare DirCol5i to several standard trajectory optimization algorithms, on three different problems with a variety of objective functions. We find that DirCol5i is able to obtain accurate solutions to all problems. On problems with simple objective functions (*eg.* minimal-force), DirCol5i is a bit slower than traditional methods. On the other hand, DirCol5i is significantly better than traditional methods when solving problems that involve higher-derivatives. It is easier to use and computes the solution more quickly.

8.3 Background

DirCol5i, the method presented in this paper, is a **dir**ect **col**location method for **i**mplicit second-order systems. It uses 5^{th}-order splines to represent the position trajectory and 3^{th}-order splines for the control. Although DirCol5i is specialized for problems that include higher-derivatives in their cost function, it is a general purpose optimization algorithm, and can be applied to any smooth trajectory optimization problem. In this section we provide a brief overview of trajectory optimization, showing how DirCol5i relates to other methods.

8.3.1 Trajectory Optimization

Trajectory optimization describes a set of numerical methods that compute open-loop solutions to the optimal control problem. The solution to a trajectory optimization problem is a sequence of controls $\boldsymbol{u}(t)$ that move a dynamical system from an initial point $\boldsymbol{x}(t_0)$ to a final point $\boldsymbol{x}(t_F)$, satisfying the system dynamics as well as other constraints, while minimizing some objective function. See §8.4.1 for more details.

There are a wide range of methods for trajectory optimization [6, 7, 93], some of which are described here for reference. DirCol5i, The method presented in this paper is a medium-order direct collocation method, and is most closely related to Hermite-Simpson direct collocation [7] and Hermite-Legendre-Gauss-Lobatto direct transcription [123].

The essence of any trajectory optimization method is the means by which it *transcribes* the trajectory optimization problem (optimizing over vector functions) to a nonlinear program (optimizing over real numbers). The method presented here transcribes the problem by representing the position trajectory $\boldsymbol{x}(t)$ using a quintic (5^{th}-order) Hermite spline, and the control trajectory $\boldsymbol{u}(t)$ as a cubic (3^{th}-order) Hermite spline. As a result, the state trajectory is C^2 continuous, and the control trajectory is C^1 continuous.

The key difference between this method and standard direct collocation methods, is that we obtain the velocity $\dot{\boldsymbol{x}}(t)$, acceleration $\ddot{\boldsymbol{x}}(t)$, and higher-order derivatives of position by direct differentiation of the position spline. Standard methods can only provide a single derivative, and thus the user must modify the state space to write any higher-derivatives in first-order form. See §8.6 for more details. An additional difference is DirCol5i uses the second-order implicit form of the system dynamics, while most trajectory optimization methods require explicit first-order form.

8.3.2 Indirect vs. Direct Methods

Traditionally, trajectory optimization problems were solved by *indirect methods*, which are based on calculus of variations and Pontryagin's maximum principle [115]. An indirect method works by constructing the necessary and sufficient conditions for optimality, which are then solved numerically. By comparison, a *direct method* works by discretizing the optimization problem itself, so that it can then be solved as a non-linear program. [6].

Indirect methods tend to be more accurate than direct methods, and the error estimates themselves are more accurate for indirect methods. Direct methods are far easier implement than indirect methods, since the user does not need to construct and initialize the adjoint equations [7]. As a result, direct methods tend to be used more commonly, and indirect methods are reserved for problems where accuracy is important, like path-planning for space flight.

8.3.3 Shooting vs. Collocation

All trajectory optimization methods perform *transcription*, which converts an optimization over the space of vector functions to an optimization over the space of real numbers. There are two key ways to do this: 1) simulation and 2) function approximation.

Shooting methods are fundamentally based on simulation, and use explicit integration schemes to satisfy the system dynamics. *Single shooting* methods approximate the entire trajectory as a single simulation, and tend to have numerical problems. *Multiple shooting* methods approximate the entire trajectory as a series of short simulations, and are more reliable. The accuracy of a solution can be improved by increasing the num-

ber of integration sub-steps, or increasing the order of the explicit integration method [7, 50].

Collocation methods, which are also known as *simultaneous methods*, are fundamentally based on function approximation, typically using polynomial splines. They typically use implicit integration schemes to satisfy the dynamics. *Global collocation*, also called *pseudospectral collocation* is a limiting case where a single (very) high-order (orthogonal) polynomial is used to represent the entire trajectory. *Orthogonal collocation* is an intermediate case, where medium-order polynomials are used for each segment of the spline. Accurate solutions are obtained by modifying the order of each segment (independently) and also the location and number of knot points between segments of the spline. Finally, *direct collocation* methods use low- or medium-order splines, where the order is fixed. Convergence is obtained by increasing the number of segments in the spline to more accurately fit the solution [93, 7, 46]. DirCol5i is a (relatively high-order) direct collocation method.

8.3.4 Transcription and Re-meshing

All trajectory optimization methods discretize the trajectory over a grid (also know as a mesh). If a small number of grid-points are used, then the solution may be inaccurate, but a large number of grid-points result in an expensive computation. Most algorithms now use a technique known as re-meshing, described in [7] and [20]. The basic idea is to solve the problem on a sparse initial grid, then compute the discretization error along the solution. Then the problem is solved again, only adding grid points (thus sub-dividing the trajectory) where the error estimates are not tolerable. This process is repeated until the error estimates are within allowable bounds. DirCol5i includes a basic re-meshing

algorithm to obtain accurate solutions.

8.3.5 Kinematic Trajectory Optimization

There are some applications it is sufficient to solve a purely kinematic trajectory optimization problem, such as for quad-rotor helicopters or fully-actuated robot arms. In some cases, kinematic constraints (*eg. maximum acceleration limits*) are constructed as a proxy for the actual system dynamics. There are a specialized set of trajectory optimization methods for computing minimum-snap trajectories for purely kinematic system, such as [74]. Carefully constructed constraints and objective functions can make the problem solvable as a linear or quadratic program, which is desirable for real-time planning and control. These methods cannot be used for more complicated systems, such as the cart-pole example problem discussed in Section §8.9.

8.4 DirCol5i: Method

DirCol5i is a **dir**ect **col**location method for **i**mplicit second-order systems, that uses 5^{th}-order splines to represent the position trajectory and 3^{th}-order splines for the control. Integral cost functions and constraints are computed using Gauss-Lobatto quadrature, and path constraints and dynamics are satisfied at the knot points of the spline, as well as the intermediate collocation points.

8.4.1 The Trajectory Optimization Problem

There are many ways to formulate trajectory optimization problems [93, 6, 84]. Here we will restrict our focus to smooth single-phase continuous-time trajectory optimization problems. There are two major differences in the problem formulation presented here, when compared to standard problem formulations. The first is that we allow higher-derivatives in the objective function, and the second is that we have a more general form of the dynamics: second-order implicit, rather than first-order explicit.

$$\begin{aligned} &\min_{t_0, t_F, \boldsymbol{x}(t), \boldsymbol{u}(t)} J\Big(t_0,\, t_F,\, \boldsymbol{x}(t_0),\, \dot{\boldsymbol{x}}(t_0),\, \boldsymbol{x}(t_F),\, \dot{\boldsymbol{x}}(t_F)\Big) \\ &+ \int_{t_0}^{t_F} w\Big(\tau,\, \boldsymbol{x}(\tau),\, \dot{\boldsymbol{x}}(\tau),\, \ddot{\boldsymbol{x}}(\tau),\, \dddot{\boldsymbol{x}}(\tau),\, \ddddot{\boldsymbol{x}}(\tau),\, \boldsymbol{u}(\tau),\, \dot{\boldsymbol{u}}(\tau)\Big)\, d\tau \end{aligned} \tag{8.1}$$

The decision variables in the optimization are the initial and final time (t_0, t_F), as well as the position and control trajectories, $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ respectively.

The optimization is subject to a variety of limits and constraints, detailed in the following equations (8.2-8.12). The first, and perhaps most important of these constraints are the system dynamics, which are typically non-linear and describe how the system changes in time. There is one important thing to note here: we allow for *implicit* dynamics. This allows a more general class of dynamics problems to be solved.

$$\mathbf{0} = \boldsymbol{f}(t,\, \boldsymbol{x}(t),\, \dot{\boldsymbol{x}}(t),\, \ddot{\boldsymbol{x}}(t),\, \boldsymbol{u}(t)) \qquad \textbf{system dynamics} \tag{8.2}$$

Next are the path constraints, which enforce restrictions along the trajectory. A path constraint could be used, for example, to keep the foot of a walking robot above the ground during a step.

$$\boldsymbol{h}(t,\, \boldsymbol{x}(t),\, \dot{\boldsymbol{x}}(t),\, \boldsymbol{u}(t)) \leq \mathbf{0} \qquad \textbf{path constraints} \tag{8.3}$$

Another important type of constraint is a non-linear boundary constraint, which puts restrictions on the initial and final state of the system. Such a constraint would be used,

for example, to ensure that the gait of a walking robot is periodic.

$$\boldsymbol{g}(t_0,\, t_F,\, \boldsymbol{x}(t_0),\, \dot{\boldsymbol{x}}(t_0),\, \boldsymbol{x}(t_F),\, \dot{\boldsymbol{x}}(t_F)) \leq \boldsymbol{0} \qquad \textbf{boundary constraints} \qquad (8.4)$$

Often there are constant limits on the state or control. For example, a robot arm might have limits on the angle, angular rate, and torque that could be applied at each joint, during the entire trajectory.

$$\boldsymbol{x}_{\text{low}} \leq \boldsymbol{x}(t) \leq \boldsymbol{x}_{\text{upp}} \qquad \textbf{path bound on position} \qquad (8.5)$$

$$\dot{\boldsymbol{x}}_{\text{low}} \leq \dot{\boldsymbol{x}}(t) \leq \dot{\boldsymbol{x}}_{\text{upp}} \qquad \textbf{path bound on velocity} \qquad (8.6)$$

$$\boldsymbol{u}_{\text{low}} \leq \boldsymbol{u}(t) \leq \boldsymbol{u}_{\text{upp}} \qquad \textbf{path bound on control} \qquad (8.7)$$

Finally, it is often important to include specific limits on the initial and final time and state. These might be used to ensure that the solution to a path planning problem reaches the goal within some desired time window.

$$t_{\text{low}} \leq t_0 < t_F \leq t_{\text{upp}} \qquad \textbf{bounds on initial and final time} \qquad (8.8)$$

$$\boldsymbol{x}_{0,\text{low}} \leq \boldsymbol{x}(t_0) \leq \boldsymbol{x}_{0,\text{upp}} \qquad \textbf{bound on initial position} \qquad (8.9)$$

$$\boldsymbol{x}_{F,\text{low}} \leq \boldsymbol{x}(t_F) \leq \boldsymbol{x}_{F,\text{upp}} \qquad \textbf{bound on final position} \qquad (8.10)$$

$$\dot{\boldsymbol{x}}_{0,\text{low}} \leq \dot{\boldsymbol{x}}(t_0) \leq \dot{\boldsymbol{x}}_{0,\text{upp}} \qquad \textbf{bound on initial velocity} \qquad (8.11)$$

$$\dot{\boldsymbol{x}}_{F,\text{low}} \leq \dot{\boldsymbol{x}}(t_F) \leq \dot{\boldsymbol{x}}_{F,\text{upp}} \qquad \textbf{bound on final velocity} \qquad (8.12)$$

8.4.2 Direct Transcription

The decision variables to a trajectory optimization problem include the vector functions that describe the position $\boldsymbol{x}(t)$ and control $\boldsymbol{u}(t)$ trajectories. Direct transcription is the process by which these vector functions are approximated using a vector of real numbers $\boldsymbol{z}$, which can then be optimized using a non-linear program.

DirCol5i uses a direct transcription that approximates these vector functions using *Hermite splines*; quintic for state, cubic for control. A spline is a function that is made up of a sequence of polynomial segments, connected at *knot points*. A cubic Hermite spline, used here to represent $\boldsymbol{u}(t)$ is defined entirely by its value $\boldsymbol{u}_k$ and derivative $\dot{\boldsymbol{u}}_k$ at the knot points t_k, where $k \in 0 \ldots N$ and N is the number of trajectory segments. This means that a control spline, by definition, is C^1 continuous.

$$\boldsymbol{u}(t) \qquad \rightarrow \qquad \{\boldsymbol{u}_k,\, \dot{\boldsymbol{u}}_k\} \quad k \in 0 \ldots N \tag{8.13}$$

A quintic Hermite spline, used here to represent the position $\boldsymbol{x}(t)$ trajectory, is defined by its value $\boldsymbol{x}_k$, slope $\dot{\boldsymbol{x}}_k$ and curvature $\ddot{\boldsymbol{x}}_k$ at each knot point t_k. Thus, the state spline is C^2 continuous.

$$\boldsymbol{x}(t) \qquad \rightarrow \qquad \{\boldsymbol{x}_k,\, \dot{\boldsymbol{x}}_k,\, \ddot{\boldsymbol{x}}_k\} \quad k \in 0 \ldots N \tag{8.14}$$

The objective function, constraints, and dynamics must also be discretized. To do this, we will introduce two intermediate points for each segment, carefully chosen to be the intermediate points required for Gauss–Lobatto quadrature, the topic of the following section. These points, combined with the knot points in each segment are known as collocation points.

Here we will choose the decision variables to be the value for the position, velocity, and acceleration at each knot point, as well as the control and the control rate. This set of variables is sufficient to fully define the position and control splines. We will need to compute derivatives of the position and control trajectories for use in the objective function as well as the dynamics and constraints. These derivatives are computed by analytic differentiation of the position and control spline. Section §8.12 provides additional details about the splines.

The dynamics are enforced by simply requiring that (8.2) holds true at each of

the collocation points. This type of constraint is identical to applying Gauss–Lobatto quadrature to the system dynamics over each segment of the trajectory. Similarly, the integrand of the objective function is evaluated at each of the collocation points, and the integral is computed numerically using Gauss-Lobatto quadrature. Finally, the path constraints are enforced at all collocation points.

8.4.3 Gauss–Lobatto Quadrature

We use Gauss–Lobatto quadrature to compute the integral term in the cost function (8.1). The method presented here is based on Gauss–Lobatto quadrature with four points, described in [48]. This method is exact for 5^{th}-order polynomials, and the error terms are 6^{th}-order. The quadrature rule is reproduced below for the interval $\tau \in [-1, 1]$.

$$\int_{-1}^{1} \boldsymbol{f}(\tau)\, d\tau \;\approx\; \frac{1}{6}\boldsymbol{f}(-1) + \frac{5}{6}\boldsymbol{f}\left(\tfrac{-1}{5}\sqrt{5}\right) + \frac{5}{6}\boldsymbol{f}\left(\tfrac{1}{5}\sqrt{5}\right) + \frac{1}{6}\boldsymbol{f}(1) \tag{8.15}$$

We can apply this rule to an arbitrary domain $t \in [a, b]$ by applying the transformation below. The integral term in the objective function is computed using this quadrature method for each segment of the trajectory.

$$\tau = 2\,\frac{t-a}{b-a} - 1 \qquad\qquad t = \frac{a+b}{2} + \tau\,\frac{b-a}{2} \tag{8.16}$$

8.4.4 Collocation Equations

The decision variables in the optimization are the initial and final times (t_0, t_N), as well as the position spline (defined by $\{\boldsymbol{x}_k,\ \dot{\boldsymbol{x}}_k,\ \ddot{\boldsymbol{x}}_k\}$) and the control spline (defined by$\{\boldsymbol{u}_k,\ \dot{\boldsymbol{u}}_k\}$). These decision variables define some arbitrary trajectory (both state and control) for the system. The *collocation equations* are a set of constraints that ensure that this trajectory

satisfies the system dynamics.

$$\mathbf{0} = \boldsymbol{f}(t,\, \boldsymbol{x}(t),\, \dot{\boldsymbol{x}}(t),\, \ddot{\boldsymbol{x}}(t),\, \boldsymbol{u}(t)) \tag{8.17}$$

We enforce the system dynamics $\boldsymbol{f}(\cdot)$ at a set of special points along the trajectory, known as *collocation points*. It turns out that the choice of collocation points corresponds to an implicit quadrature method. In this case, we choose the same Gauss–Lobatto points that we used for computing the objective function integral (8.15), shown below on the domain $\tau \in [-1, 1]$.

$$\left\{-1,\ \tfrac{-1}{5}\sqrt{5},\ \tfrac{1}{5}\sqrt{5},\ 1\right\} \tag{8.18}$$

We need to enforce the dynamics at these collocation points for each segment of the trajectory, $t \in [t_k, t_{k+1}]$. This means we need value for $\boldsymbol{x}$, $\dot{\boldsymbol{x}}$, $\ddot{\boldsymbol{x}}$, and $\boldsymbol{u}$ at each point. The decision variables directly give us these values at the boundary points, but we need to use interpolation to obtain the values at the two intermediate points on each interval.

The choice of interpolation method here is important, in that it must match the method that we used initially to transcribe the continuous decision variables ($\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$) into a set of real numbers. In other words, we need to compute the values for position, velocity, acceleration, and control by evaluating the position spline (and its derivatives) and the control spline.

8.4.5 Trajectory Mesh

The trajectory mesh, also known as a grid, describes the relative location of the knot points on the position and control splines. Recall that both the initial and final times (t_0, t_N) of the trajectory are decision variables in the optimization. There are many

intermediate knot points t_k, where $k \in 1 \ldots (N-1)$. These knot points are typically computed by scaling a fixed grid (or mesh), such that the first and last point match the initial and final time. This is standard practice in trajectory optimization [7], as numerical problems often arise if all knot point locations are made decision variables.

It turns out that the location and spacing of the knot points is critical to obtaining an accurate solution. Although it is possible for a user to specify a good mesh manually, it is common to automatically construct a good mesh, using procedures like those described in the following section. This is typically done iteratively, first starting with a simple coarse mesh, and then adding collocation points after each iteration, based on error estimates, as discussed in Section §8.5.

8.5 Error Analysis and Convergence

Thus far we have covered how to pose a trajectory optimization problem as a non-linear program using DirCol5i. Once we have a solution to the non-linear program, we can then construct an estimate of how well DirCol5i captures the underlying dynamics of the system *between the collocation points*. Here we will us a slightly modified version of the approach described by Betts in [7].

We start by defining a term $\varepsilon(t)$ that gives the defect in the system dynamics along the trajectory. Notice that this error function will be identically zero at the collocation points, assuming that the non-linear program has converged successfully.

$$\varepsilon(t) = \boldsymbol{f}\big(t, \boldsymbol{x}(t), \dot{\boldsymbol{x}}(t), \ddot{\boldsymbol{x}}(t), \boldsymbol{u}(t)\big) \tag{8.19}$$

Next, we will compute an estimate of the absolute error within each segment k. The

definite integral here is computed using Romberg quadrature [95, 64].

$$\boldsymbol{\eta}_k = \int_{t_k}^{t_{k+1}} |\boldsymbol{\varepsilon}(t)|\, dt \tag{8.20}$$

This error estimate is still a vector quantity within each segment, but we will need a scalar for the mesh analysis, which in this case is obtained by selecting error corresponding to the dimension with the maximum error in a given segment. In cases where the problem is not well scaled, it might be necessary to apply a weight to the estimate along each dimension before computing the maximum [7]. In the following equation, k is the segment index and i is the dimension index.

$$\epsilon_k = \max_i \boldsymbol{\eta}_{k,i} \tag{8.21}$$

At this point, we would like to know how to sub-divide each segment of the trajectory. We would like to add just enough new segments to achieve the desired error tolerance. Here we use a highly simplified version of the approach by Betts [7]. The idea is to use the method order to predict how the error will change by sub-dividing any given segment. We know that the quadrature method corresponding the to collocation points is 6^{th}-order ($p = 6$), so we can estimate the number of sub-segments required to achieve a desired tolerance ϵ^* We introduce a safety factor of $\kappa \approx \frac{1}{10}$ as suggested by Betts [7].

$$N_k \approx \left(\frac{\kappa\, \epsilon^*}{\epsilon_k}\right)^{\left(\frac{-1}{p}\right)} \tag{8.22}$$

A larger value of κ will have fewer sub-divisions per iteration, but more iterations, while a smaller value of κ will have more sub-divisions per iteration, but fewer iterations. As a practical matter, N_k is rounded up to the nearest integer in the set $N_k \in \{1, 2, ...N_{\max}\}$, where $N_{\max}$ is typically about five [7].

8.6 Solving with Traditional Methods

In this paper we present a new algorithm (DirCol5i) for solving trajectory optimization problems that have non-trivial dynamics and higher derivatives in their cost functions. Such problems can be solved with traditional algorithms, although it requires a major reformulation of the problem statement. Traditional solvers require the dynamics to be in first order form, and the objective and constraint functions can only operate on the state and control (not their derivatives). In this section we present a method for transforming the problem statement for this paper (an objective function including higher-derivatives) to one that can be used with a standard trajectory optimization technique.

Here we show the formulations for minimum-snap, minimum-jerk, and minimum-input-rate problems separately, but they can easily be combined when the objective function requires it. When using these transformations, it is important to only include decision variables that actually appear in the problem. For example, the minimum-snap (4^{th}-derivative of position) problem formulation should not be used for problems that only includes derivatives up to jerk (3^{rd}-derivative of position). This means that each change to the objective function will (likely) require a change in the problem formulation. DirCol5i does not require these transformations, which is a significant advantage for problems with higher-derivatives in the objective function.

8.6.1 Minimum Snap

Let's start by consider a simplified version of a trajectory optimization problem that includes forth-derivatives of the position, as shown below.

$$J = \min \int w(\tau,\, \boldsymbol{x},\, \dot{\boldsymbol{x}},\, \ddot{\boldsymbol{x}},\, \dddot{\boldsymbol{x}},\, \ddddot{\boldsymbol{x}},\, \boldsymbol{u})\, d\tau \tag{8.23}$$

$$\text{subject to:} \qquad \ddot{\boldsymbol{x}} = \boldsymbol{f}(t,\, \boldsymbol{x},\, \dot{\boldsymbol{x}},\, \boldsymbol{u}) \tag{8.24}$$

Most traditional trajectory optimization problems require that the problem be in standard form:

$$J = \min \int \bar{w}(\tau,\, \bar{\boldsymbol{x}},\, \bar{\boldsymbol{u}})\, d\tau \tag{8.25}$$

$$\text{subject to:} \qquad \dot{\bar{\boldsymbol{x}}} = \bar{\boldsymbol{f}}(\tau,\, \bar{\boldsymbol{x}},\, \bar{\boldsymbol{u}}) \tag{8.26}$$

We can make this substitution using a *chain integrator*, a standard trick that is used for converting a high-order differential equation into a system of first-order differential equations. We do this by creating a new set of variables:

$$\begin{aligned} \boldsymbol{x}_0 &:= \boldsymbol{x} & \boldsymbol{x}_2 &:= \ddot{\boldsymbol{x}} & \boldsymbol{u}_1 &:= \boldsymbol{u} \\ \boldsymbol{x}_1 &:= \dot{\boldsymbol{x}} & \boldsymbol{x}_3 &:= \dddot{\boldsymbol{x}} & \boldsymbol{u}_x &:= \ddddot{\boldsymbol{x}} \end{aligned}$$

Next we need to satisfy the dynamics. We do this by creating an additional state variable for the velocity. Notice that we now have two variables representing velocity: $\boldsymbol{x}_d$ and $\boldsymbol{x}_1$.

$$\boldsymbol{x}_d := \dot{\boldsymbol{x}}$$

We can now collect all of these new variables into column vectors state $\bar{\boldsymbol{x}}$ and control $\bar{\boldsymbol{u}}$.

$$\bar{\boldsymbol{x}} = [\boldsymbol{x}_0,\, \boldsymbol{x}_1,\, \boldsymbol{x}_d,\, \boldsymbol{x}_2,\, \boldsymbol{x}_3]$$

$$\bar{\boldsymbol{u}} = [\boldsymbol{u}_1,\, \boldsymbol{u}_x]$$

In our new form we have two states the track the velocity: $\boldsymbol{x}_1$ is used to satisfy the dynamics of the chain integrator, and $\boldsymbol{x}_d$ is used to satisfy the system dynamics. We need to resulting solution to be consistent, so we add a path constraint that requires both versions of the velocity to be identical:

$$\boldsymbol{x}_d = \boldsymbol{x}_1$$

Now we can write out the combined (first-order) dynamics for the system: $\dot{\bar{\boldsymbol{x}}} = \bar{\boldsymbol{f}}(\tau, \bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})$, which are given below.

$$\begin{bmatrix} \dot{\boldsymbol{x}}_0 \\ \dot{\boldsymbol{x}}_1 \\ \dot{\boldsymbol{x}}_2 \\ \dot{\boldsymbol{x}}_3 \\ \dot{\boldsymbol{x}}_d \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{x}_3 \\ \boldsymbol{u}_x \\ \boldsymbol{f}(t, \boldsymbol{x}_0, \boldsymbol{x}_d, \boldsymbol{u}_1) \end{bmatrix}$$

We can use these new variables to arrive at an expression for $\bar{w}(\cdot)$, shown below.

$$J = \min \int \bar{w}(\tau, \bar{\boldsymbol{x}}, \bar{\boldsymbol{u}})\, d\tau = \min \int w(\tau, \boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{u}_x, \boldsymbol{u}_1)\, d\tau \quad (8.27)$$

8.6.2 Minimum Jerk

It is also common to solve for minimum-jerk trajectories. The formulation is similar to that used for minimum-snap, just with one fewer chain integrator:

$$\boldsymbol{x}_0 := \boldsymbol{x} \qquad \boldsymbol{x}_2 := \ddot{\boldsymbol{x}} \qquad \boldsymbol{u}_1 := \boldsymbol{u}$$
$$\boldsymbol{x}_1 := \dot{\boldsymbol{x}} \qquad\qquad \boldsymbol{u}_x := \dddot{\boldsymbol{x}}$$

The modified state and control vectors now become:

$$\bar{\boldsymbol{x}} := [\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_d, \boldsymbol{x}_2]$$
$$\bar{\boldsymbol{u}} := [\boldsymbol{u}_1, \boldsymbol{u}_x]$$

The first-order dynamics are:

$$\begin{bmatrix} \dot{\boldsymbol{x}}_0 \\ \dot{\boldsymbol{x}}_1 \\ \dot{\boldsymbol{x}}_2 \\ \dot{\boldsymbol{x}}_d \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \boldsymbol{u}_x \\ \boldsymbol{f}(t, \boldsymbol{x}_0, \boldsymbol{x}_d, \boldsymbol{u}_1) \end{bmatrix}$$

8.6.3 Minimum Control Rate

Let's suppose that we want to compute a trajectory where the objective function includes a term on the control rate. This can be done using a standard substitution, where the original control becomes a new state variable, and the new control variable is the rate of change in the original control. Consider the simplified objective function below:

$$J = \min \int w(\tau,\, \boldsymbol{x},\, \dot{\boldsymbol{x}},\, \boldsymbol{u},\, \dot{\boldsymbol{u}})\, d\tau \tag{8.28}$$

$$\text{subject to:} \qquad \ddot{\boldsymbol{x}} = \boldsymbol{f}(t,\, \boldsymbol{x},\, \dot{\boldsymbol{x}},\, \boldsymbol{u}) \tag{8.29}$$

Most traditional trajectory optimization problems require that the problem be in standard form:

$$J = \min \int \bar{w}(\tau,\, \bar{\boldsymbol{x}},\, \bar{\boldsymbol{u}})\, d\tau \tag{8.30}$$

$$\text{subject to:} \qquad \dot{\bar{\boldsymbol{x}}} = \bar{\boldsymbol{f}}(\tau,\, \bar{\boldsymbol{x}},\, \bar{\boldsymbol{u}}) \tag{8.31}$$

Just like with the minimum-snap formulation, we can use a chain integrator to represent the minimum-control-rate problem in standard form.

$$\boldsymbol{x}_0 := \boldsymbol{x} \qquad \boldsymbol{x}_u := \boldsymbol{u}$$

$$\boldsymbol{x}_1 := \dot{\boldsymbol{x}} \qquad \bar{\boldsymbol{u}} := \dot{\boldsymbol{u}}$$

We can now collect all of these new variables into column vectors state $\bar{\boldsymbol{x}}$ and control $\bar{\boldsymbol{u}}$.

$$\bar{\boldsymbol{x}} := [\boldsymbol{x}_0,\, \boldsymbol{x}_1,\, \boldsymbol{x}_u]$$

Now we can write out the combined dynamics for the system: $\dot{\bar{\boldsymbol{x}}} = \bar{\boldsymbol{f}}(\tau,\, \bar{\boldsymbol{x}},\, \bar{\boldsymbol{u}})$, which are given below.

$$\begin{bmatrix} \dot{\boldsymbol{x}}_0 \\ \dot{\boldsymbol{x}}_1 \\ \dot{\boldsymbol{x}}_u \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{f}(t, \boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_u) \\ \bar{\boldsymbol{u}} \end{bmatrix}$$

We can use these new variables to arrive at an expression for $\bar{w}(\cdot)$, shown below.

$$J \quad = \quad \min \int \bar{w}(\tau,\, \bar{\boldsymbol{x}},\, \bar{\boldsymbol{u}})\, d\tau \quad = \quad \min \int w(\tau,\, \boldsymbol{x}_0,\, \boldsymbol{x}_1,\, \boldsymbol{x}_u,\, \bar{\boldsymbol{u}})\, d\tau \tag{8.32}$$

8.6.4 Scaling

One problem that occurs when using a chain integrator is that of scaling. In particular, consider a problem where the optimal position trajectory is $x(t) = \sin(c\,t)$. We can then easily arrive at expressions for the derivatives of x:

$$\dot{x}(t) = c\cos(c\,t)$$

$$\ddot{x}(t) = -c^2\sin(c\,t)$$

$$\dddot{x}(t) = -c^3\cos(c\,t)$$

$$\ddddot{x}(t) = c^4\sin(c\,t)$$

We can easily see that the amplitude of the position trajectory is c^0, while the amplitude of the snap (4^{th}-derivative of position) is c^4. This presents a problem if c is not close to 1. Let's consider a case where $c = 10$, which would imply that the snap trajectory has an amplitude that is 10^4 times larger than the position trajectory. Since the integrator chain dynamics are enforced numerically, the constraint solver can run into numerical problems when the problem is improperly scaled.

8.7 Quadrotor Example

For our first example we will study the minimal-force trajectory to perform a lateral transfer maneuver for a simple quadrotor helicopter model. This problem can be solved using traditional methods with no modification. Here we will compare the solution obtained using DirCol5i (implemented in Matlab) to that obtained using GPOPS-II, a professionally-developed trajectory optimization library for Matlab [84].

8.7.1 Quadrotor Problem Statement

The quadrotor is modeled as a planar rigid body with three degrees of freedom: lateral and vertical translation, as well as roll angle. The propellers are modeled as two force actuators on opposite sides of the quadrotor. Each applies a force perpendicular to the width of the quad-rotor. Finally, we assume that there is no friction and that the quad-rotor is symmetric.

The equations of motion are given below, where x, y, and θ describe the state of the quadrotor: horizontal position, vertical position, and roll angle. The mass distribution is approximated by two point masses $m = 0.2$kg, one positioned at each edge of the frame, separated by a distance $d = 0.3$m. The force provided by each propeller is given by u_1 and u_2, and the acceleration is given by $g = 9.81\text{m/s}^2$.

$$\begin{aligned}
\ddot{x} &= \frac{-1}{m}\sin(\theta)(u_1 + u_2) \\
\ddot{y} &= \frac{1}{m}\cos(\theta)(u_1 + u_2) - 2g \\
\ddot{\theta} &= \frac{1}{2md}(u_2 - u_1)
\end{aligned}$$

The starting state for the quadrotor is a stationary hover at the origin. The final state is a stationary hover some prescribed distance away, at the same height, as shown below.

$$\begin{aligned}
t_0 = 0 \quad x(t_0) = 0 \quad y(t_0) = 0 \quad \theta(t_0) = 0 \\
t_F = 1 \quad x(t_F) = 1 \quad y(t_F) = 0 \quad \theta(t_0) = 0
\end{aligned} \tag{8.33}$$

The objective is to compute the trajectory between these points (in a fixed amount of time) that minimizes the integral of the actuator force squared along the trajectory.

$$J = \int_{t_0}^{t_F} (u_1^2 + u_2^2)\, dt \tag{8.34}$$

This is a relatively simple trajectory optimization, and here we use it as a basic demonstration that DirCol5i obtains the same solution as standard methods.

8.7.2 Quadrotor Results

We used DirCol5i to compute the optimal lateral transfer maneuver for the planar quadrotor model, minimizing the integral of the thrust force squared. The resulting trajectory is shown in Figure 8.1. We then solved the same problem using GPOPS-II [84], and find that both algorithms arrived at the same solution.

GPOPS-II reported that it's trajectory had a maximum mesh error of 1.9×10^{-10}, and the non-linear programming solver (IPOPT [12]) successfully converged. DirCol5i reported an maximum mesh error of 6.8×10^{-5}, and non-linear programming solver (FMINCON [70]) also converged successfully. The difference between the objective function value computed by each algorithm was 4.9×10^{-6}, the maximum difference between the position trajectories was 5.2×10^{-5}.

GPOPS-II computed the optimal trajectory in 2.02 seconds, to a mesh tolerance of 10^{-10}. DirCol5i computed the solution using two iterations, automatically computing a new mesh for the second iteration. The first (coarse-grid) solution took 7.15 seconds, while the second (fine-grid) solution took 13.6 seconds. Both GPOPS-II and DirCol5i are implemented in Matlab, although the code-base for GPOPS-II is highly optimized, particularly with regard to gradient calculations. The optimal trajectory is shown in Figure 8.1.

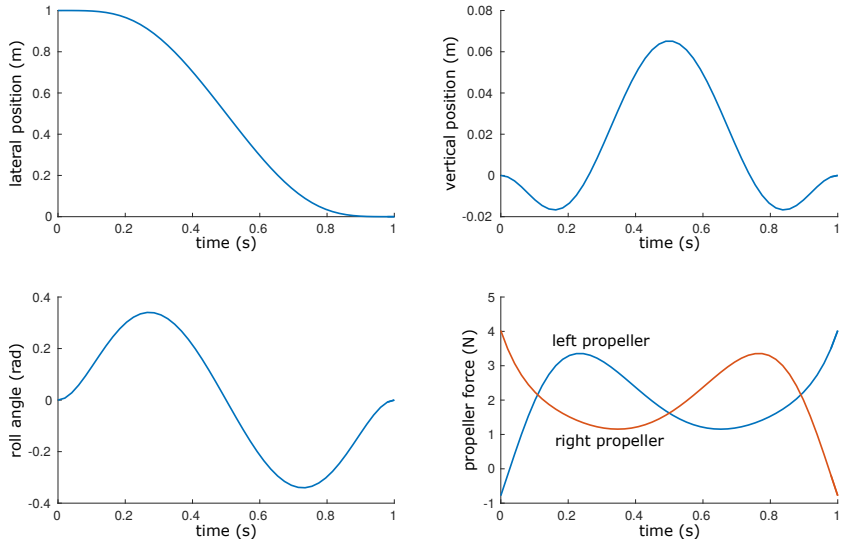

Figure 8.1: **Quad-rotor lateral transfer, minimum force** Minimal-torque trajectory for a lateral shift maneuver for a planar model of a quad-rotor helicopter.

8.8 Pendulum Swing-up Example

Our next problem is a swing-up for a simple pendulum, both with and without actuator limits. We compare the performance of DirCol5i to three other trajectory optimization algorithms: GPOPS-II (Radau orthogonal collocation), trapezoid direct collocation, and Chebyshev-Lobatto global (pseudospectral) collocation. Each of these methods is then used to compute optimal swing-up trajectories for a simple pendulum, using five different objective functions. Finally, each trajectory is computed both with and without actuator limits.

8.8.1 Solvers

- *DirCol5i* is the solver presented in this paper. It is a medium-order direct collocation method. For the purposes of this experiment we compute the solution on a single uniform mesh with 12 segments.

- *GPOPS-II* is a professionally-developed trajectory optimization software for Matlab [84]. It implements a multiple-segment Radau orthogonal collocation. For the purposes of this experiment we use a single uniform mesh with 10 segments (4 collocation points per segment).
- *Trapezoid* direct collocation is a standard low-order method for trajectory optimization [7]. Here we compute the solution on a single uniform mesh with 50 collocation points.
- *Chebyshev*-Lobatto collocation is a high-order global (pseudospectral) trajectory optimization method [23]. Here we use a single 24^{th}-order Chebyshev polynomial to represent the trajectory.

We use the implementations of the trapezoid and Chebyshev collocation methods found in TrajOpt, an open-source trajectory optimization library for Matlab [58].

8.8.2 Pendulum Swing-Up Problem Statement

Compute the optimal swing-up trajectory for a simple pendulum, given a torque motor with actuator limits and viscous friction at the hinge. Let x describe the angle of the pendulum, with $x = 0$ corresponding to the minimum-energy state, and u be the torque acting on the pendulum from a motor. The dynamics for the simple pendulum (with damping) are given below, where $k = 3.0$ and $b = 0.2$ are physical parameters.

$$\ddot{x} = -b\dot{x} - k\sin(x) + u \tag{8.35}$$

The swing-up problem can be expressed by the following boundary values:

$$\begin{aligned} t_0 = 0 \quad x(t_0) = 0 \quad \dot{x}(t_0) = 0 \\ t_F = 5 \quad x(t_F) = \pi \quad \dot{x}(t_F) = 0 \end{aligned} \tag{8.36}$$

We will test each solver on five different objective functions, listed below.

- Minimum-torque: $J = \int u^2 \, dt$
- Minimum-torque-rate: $J = \int \dot{u}^2 \, dt$
- Minimum-acceleration: $J = \int \ddot{x}^2 \, dt$
- Minimum-jerk: $J = \int \dddot{x}^2 \, dt$
- Minimum-snap: $J = \int \ddddot{x}^2 \, dt$

For each of these objective functions, we will compute the optimal swing-up trajectory for two situations: first without actuator limits ($u_{\max} = \infty$), and then with actuator limits ($u_{\max} = 2$). The choice of actuator limit ($u_{\max} = 2$) is chosen such that any feasible swing-up trajectory must reverse direction at some-point in the swing. This makes the optimization problem harder to solve, thus providing a better comparison metric.

8.8.3 Pendulum Swing-Up Results

We computed the optimal swing-up trajectory for a simple pendulum, for five different objective functions, with and without actuator saturation, using five different objective functions. The results of all 40 optimizations are summarized in Tables 8.1 and 8.2.

For each trajectory we provide the value of the objective function and also the amount of time required to compute the solution. Unless otherwise specified, methods that have the same objective function value arrived at the same solution. In cases where the optimization failed to find a solution the value of the objective function is omitted, and the cause of the failure is listed in place of the computation time.

DirCol5i is the only method that was able to successfully compute a feasible trajectory for all problems. It found the optimal solution on all but one of the problems: minimum-snap, without actuator limits, where it found a feasible but slightly sub-optimal solution. DirCol5i took more computation time than the other algorithms for most, but not all problems.

GPOPS-II computed the correct solution for 7/10 problems. When it converged, it was about ten times faster than the other methods. On the three problems where it failed, the interpolant diverged wildly between collocation points.

The Trapezoid direct collocation method found the correct solution to 8/10 problems, got stuck on an infeasible point in one problem, and found a sub-optimal solution on another.

The Chebyshev global collocation method performed well on the five problems without actuator limits. It found the correct solution more quickly than DirCol5i or Trapezoid. It had more trouble on the set of five problems with actuator limits. Although it arrived at nearly the correct solution for all problems, in most cases the interpolant exhibited a numerical artifact known as 'ringing'. It occurs because a polynomial cannot properly approximate a solution with a discontinuity, in this case cause by actuator saturation. This is a well-known problem with global (pseudospectral) collocation methods [93].

8.9 Cart-Pole Swing-Up Example

One of the standard problems in trajectory optimization is the *cart-pole swing-up* problem. In this problem there is a cart that can travel along a horizontal rail, powered by a

motor. A pendulum, or pole, is pinned to the cart, such that it can freely rotate. The goal is to compute a force profile to apply to the cart that will cause the pole to swing-up to

Table 8.1: Pendulum Swing-up: no actuator limits. Four different solvers, five different objective functions. DirCol5i finds the correct solution for all but minimum-snap objective functions. DirCol5i is generally the slowest solver. GPOPS-II is the fastest solver, but fails on two objective functions. Trapezoid and Chebyshev solvers both get the correct solution to all problems, with a moderate computation time.

	$J = \int u^2\, dt$ $\lvert u(t)\rvert < \infty$	$J = \int \dot{u}^2\, dt$ $\lvert u(t)\rvert < \infty$	$J = \int \ddot{x}^2\, dt$ $\lvert u(t)\rvert < \infty$	$J = \int \dddot{x}^2\, dt$ $\lvert u(t)\rvert < \infty$	$J = \int \ddddot{x}^2\, dt$ $\lvert u(t)\rvert < \infty$
`DirCol5i`	$J = 9.17$ 4.66 sec	$J = 2.30$ 1.31 sec	$J = 0.948$ 4.58 sec	$J = 0.379$ 3.67 sec	$J = 0.0377$ 1.74 sec
`GPOPS-II`	$J = 9.17$ 0.33 sec	$J = 2.30$ 0.13 sec	$J = --$ diverged	$J = --$ diverged	$J = 0.0000$ 0.23 sec
`Trapezoid`	$J = 9.21$ 2.90 sec	$J = 2.30$ 1.23 sec	$J = 0.949$ 2.64 sec	$J = 0.380$ 2.61 sec	$J = 0.0000$ 3.36 sec
`Chebyshev`	$J = 9.17$ 0.95 sec	$J = 2.30$ 1.19 sec	$J = 0.948$ 2.13 sec	$J = 0.379$ 1.57 sec	$J = 0.0000$ 1.96 sec

Table 8.2: Pendulum Swing-up, with actuator limits. Four different solvers, five different objective functions. DirCol5i is the only solver to compute the correct solution to all problems, and it does so in a moderate amount of time. GPOPS-II quickly computes the solution to most problems, but fails on the minimum-snap trajectory. Trapezoid direct collocation solves three problems correctly, get stuck on an infeasible solution on one, and finds a sub-optimal solution on another. Chebyshev global collocation gets nearly the right solution to most problems, but exhibits 'ringing', a numerical artifact caused by a discontinuous solution (due to torque-limits).

	$J = \int u^2\, dt$ $\lvert u(t)\rvert < u_{\max}$	$J = \int \dot{u}^2\, dt$ $\lvert u(t)\rvert < u_{\max}$	$J = \int \ddot{x}^2\, dt$ $\lvert u(t)\rvert < u_{\max}$	$J = \int \dddot{x}^2\, dt$ $\lvert u(t)\rvert < u_{\max}$	$J = \int \ddddot{x}^2\, dt$ $\lvert u(t)\rvert < u_{\max}$
`DirCol5i`	$J = 9.24$ 5.68 sec	$J = 12.9$ 2.22 sec	$J = 8.29$ 5.46 sec	$J = 27.4$ 4.69 sec	$J = 129$ 3.28 sec
`GPOPS-II`	$J = 9.22$ 0.17 sec	$J = 12.9$ 0.19 sec	$J = 8.06$ 0.26 sec	$J = 27.2$ 0.37 sec	$J = --$ diverged
`Trapezoid`	$J = 9.26$ 2.98 sec	$J = --$ infeasible	$J = 8.31$ 6.34 sec	$J = 43.1$ 16.3 sec	$J = 1220$ 18.1 sec
`Chebyshev`	$J = 9.22$ ringing	$J = 12.8$ 5.51 sec	$J = 8.28$ ringing	$J = 27.3$ ringing	$J = 205$ ringing

be balanced above the cart, starting from rest hanging below the cart.

Here we will solve this problem for four different objective functions, using DirCol5i, as well as trapezoid direct collocation [7] (implemented using TrajOpt[58]), and GPOPS-II[84].

8.9.1 Cart-Pole Problem Statement

The cart-pole system is modeled as two point masses: one for the cart and one at the tip of the pole (pendulum). We neglect friction, and assume that the rails are perfectly horizontal. There is a single actuator, modeled as a force that is applied to the cart, parallel to the rails.

In the starting state the system is stationary, with the pendulum hanging directly below the cart. In the final state, the cart has moved some prescribed distance along the rails, and the pendulum is balanced above the cart. The objective is to find the force profile that achieves this transition in some fixed time, while minimizing the integral of some objective function along the trajectory.

The equations of motion for the cart-pole system are given below, where x gives the horizontal position of the cart and θ gives the angle of the pole, where $\theta = 0$ describes the configuration where the pole is in the minimum potential energy state. The motor force is given by u, gravity acceleration by $g = 9.81\text{m/s}^2$, and the length of the pole by $\ell = 0.5\text{m}$. The mass of the cart is given by $m_1 = 2.0\text{kg}$ and the point-mass at the tip of the pole is given by $m_2 = 0.5\text{kg}$.

$$0 = u - m_2\left(-\ell\dot{\theta}^2\sin(\theta) + \ddot{x} + \ell\ddot{\theta}\cos(\theta)\right) - m_1\ddot{x} \tag{8.37}$$

$$0 = -\ell m_2\left(\ell\ddot{\theta} + \ddot{x}\cos(\theta) + g\sin(\theta)\right) \tag{8.38}$$

The swing-up problem is defined by the boundary conditions, given below:

$$\begin{aligned} t_0 = 0 \quad x(t_0) = 0 \quad \dot{x}(t_0) = 0 \quad \theta(t_0) = 0 \quad \dot{\theta}(t_0) = 0 \\ t_F = 2 \quad x(t_F) = 1 \quad \dot{x}(t_F) = 0 \quad \theta(t_F) = \pi \quad \dot{\theta}(t_F) = 0 \end{aligned} \tag{8.39}$$

We solve this problem for four different objective functions, which are listed below.

- Minimum-torque: $J = \int u^2 \, dt$
- Minimum-torque-rate: $J = \int \dot{u}^2 \, dt$
- Minimum-jerk: $J = 10^{-3} \int \|\dddot{\boldsymbol{p}}\|^2 \, dt$
- Minimum-snap: $J = 10^{-6} \int \|\ddddot{\boldsymbol{p}}\|^2 \, dt$

Note that the minimum-jerk and minimum-snap trajectories are minimizing the jerk and snap of the position of the tip of the pole $\boldsymbol{p}$, rather than the jerk and snap of the minimal coordinates. These expressions are given below.

$$\begin{aligned} \|\dddot{\boldsymbol{p}}\|^2 = &\left(-\ell\dot{\theta}^3\cos(\theta) - 3\ell\dot{\theta}\ddot{\theta}\sin(\theta) + \dddot{x} + \ell\dddot{\theta}\cos(\theta)\right)^2 \\ &+ \ell^2\left(-\dot{\theta}^3\sin(\theta) + 3\dot{\theta}\ddot{\theta}\cos(\theta) + \dddot{\theta}\sin(\theta)\right)^2 \end{aligned} \tag{8.40}$$

$$\begin{aligned} \|\ddddot{\boldsymbol{p}}\|^2 = &\left(-3\ell\ddot{\theta}^2\sin(\theta) - 6\ell\ddot{\theta}\dot{\theta}^2\cos(\theta) + \ell\dot{\theta}^4\sin(\theta) - 4\dddot{\theta}\ell\dot{\theta}\sin(\theta) + \ddddot{x} + \ddddot{\theta}\ell\cos(\theta)\right)^2 \\ &+ \ell^2\left(3\ddot{\theta}^2\cos(\theta) - 6\ddot{\theta}\dot{\theta}^2\sin(\theta) - \dot{\theta}^4\cos(\theta) + 4\dddot{\theta}\dot{\theta}\cos(\theta) + \ddddot{\theta}\sin(\theta)\right)^2 \end{aligned} \tag{8.41}$$

8.9.2 Cart-Pole Swing-Up Results

The optimal solution for each objective function, as computed by DirCol5i, is shown in Figure 8.2. Additionally, Figure 8.3 shows the path that the tip of the pendulum traces during the swing-up maneuver, for each objective function. The optimization

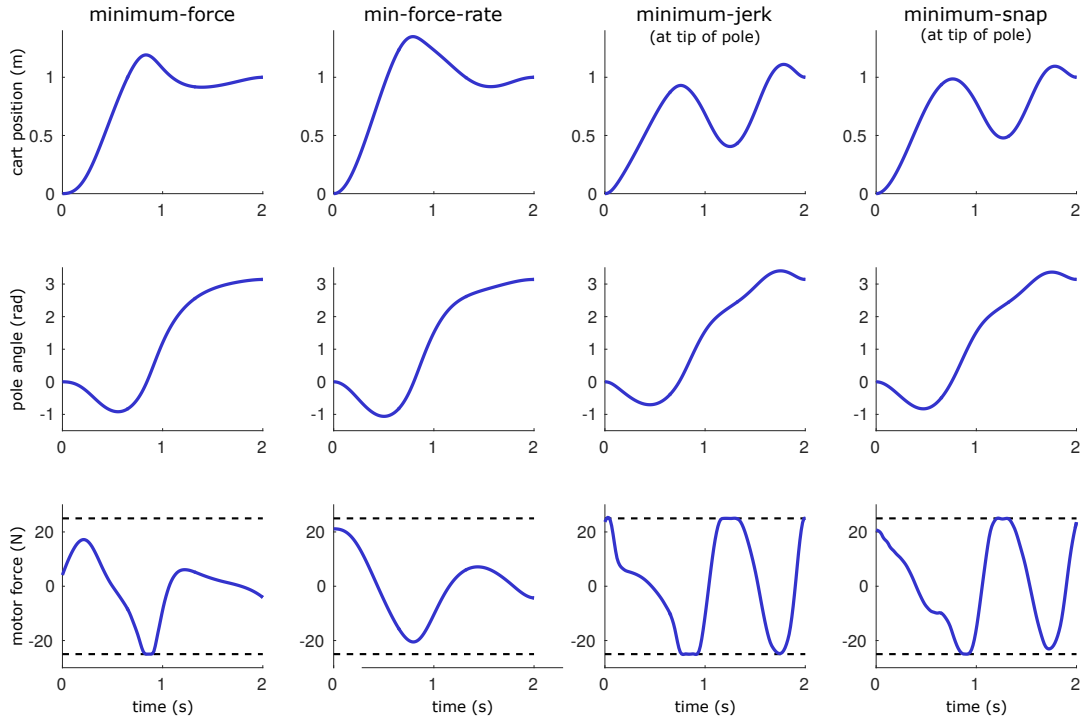

Figure 8.2: **Cart-Pole Swing-Up:** Cart position, pole angle, and motor force, as functions of time, obtained using DirCol5i. The dashed lines in the motor force plots show the actuator limits.

finds two different swing-up strategies: one for the minimum-force and minimum-force-rate trajectories and the other for the minimum-jerk and minimum-snap trajectories. All solutions except for the minimum-torque-rate saturate the actuator

We solved the cart-pole swing-up problem for each of the four objective functions, using DirCol5i, trapezoid direct collocation and GPOPS-II. For the minimum-torque and minimum-torque-rate problems, all solvers arrived at the same solution, within the expected numerical tolerances. DirCol5i and GPOPS-II found the same (optimal) solution for the minimum-jerk problem, and trapezoid direct collocation found a similar, but sub-optimal, solution. On the minimum-snap problem only DirCol5i computed the optimal solution. GPOPS-II arrived at nearly the same solution, but failed to properly converge due to a numerical artifact at the beginning of the trajectory. Trapezoid direct collocation successfully converged, but to a far sub-optimal solution. DirCol5i was

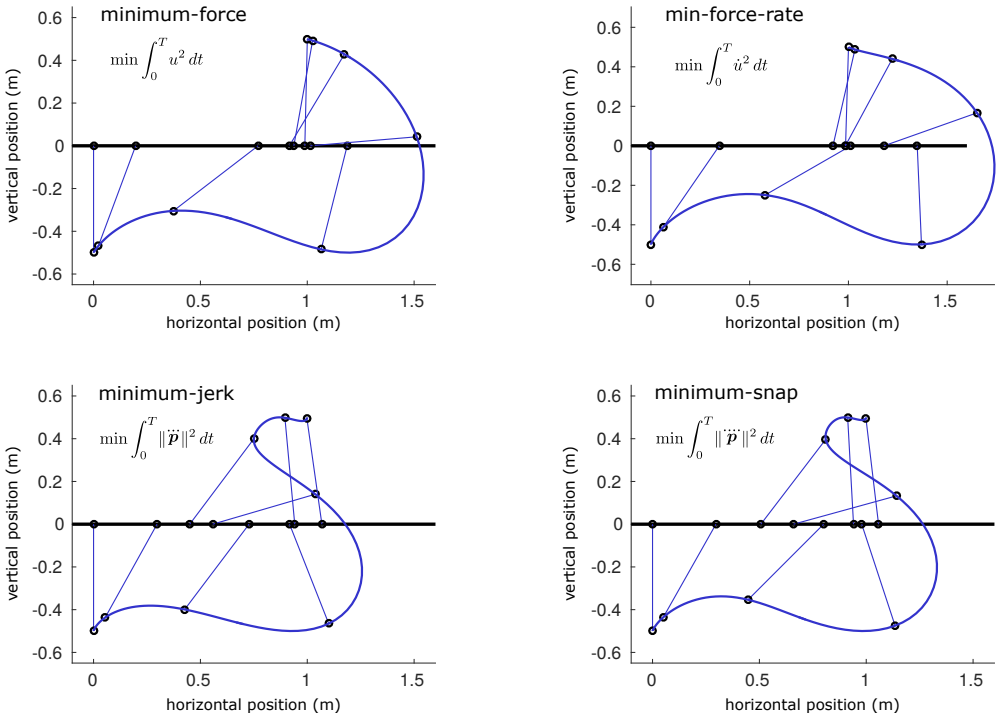

Figure 8.3: **Cart-Pole Swing-Up:** Trace made by pendulum bob along each trajectory. Straight lines show the pendulum at uniformly spaced time intervals along the swing-up trajectory. The minimum-force and minimum-force-rate solutions use a similar strategy, which is different from the strategy used by the minimum-jerk and minimum-snap solutions. Note that point $\boldsymbol{p}$ in the minimum-jerk and minimum-snap objective functions is the tip of the pole.

slower than the traditional methods on the minimum-torque and minimum-torque-rate problems, while it was faster on the minimum-jerk and minimum-snap problems. The results are summarized in Table 8.3.

8.10 Discussion

In this paper we presented DirCol5i, a trajectory optimization technique for solving minimal-snap trajectories for systems with non-trivial dynamics. There are two key differences between DirCol5i and other trajectory optimization methods. The first is that

DirCol5i uses an implicit form of the dynamics. This allows for DirCol5i to be used on a wider ranger of problems. The second difference between DirCol5i and other methods is that DirCol5i computes derivatives of the trajectory (for use in the objective function, dynamics, and constraints) by analytic differentiation of the spline that is approximating the position. In addition to making implicit dynamics functions possible, this makes it easy to work with higher-order derivatives in the objective function. The cost for these two features is a reduction in the sparsity in the non-linear program, making DirCol5i a bit slower than traditional methods on standard problems, where these features are not required.

We compared DirCol5i to traditional trajectory optimization methods for solving three different example problems, each for a variety of cost functions. Here we will summarize the results of this comparison.

- *Accuracy:* In most cases DirCol5i computes the same solution as the traditional

Table 8.3: Cart-Pole Swing-Up. All solvers arrived at the same solution for the minimum-force and minimum-force-rate objective functions. The Trapezoid method found a sub-optimal solution for both the minimum-jerk and minimum-snap objective functions. GPOPS-II and DirCol5i find the same (optimal) solution for all four objective functions. The solve times for DirCol5i tend to go down as the objective function is operating on higher derivatives, while the solve-times increase for the Trapezoiod direct collocation and GPOPS-II. *GPOPS-II failed to successfully converge on the minimum-snap objective function, although it is due to a small numerical artifact on each end of the trajectory. It is otherwise similar to the solution obtained by DirCol5i.

	$J = 10^{-2} \int u^2 \, dt$	$J = 10^{-3} \int \dot{u}^2 \, dt$	$J = 10^{-3} \int \|\ddot{\boldsymbol{p}}\|^2 \, dt$	$J = 10^{-6} \int \|\dddot{\boldsymbol{p}}\|^2 \, dt$
`DirCol5i`	$J = 2.220$ 218.8 sec	$J = 4.610$ 53.94 sec	$J = 21.39$ 96.21 sec	$J = 3.854$ 22.47 sec
`GPOPS-II`	$J = 2.220$ 0.64 sec	$J = 4.610$ 1.220 sec	$J = 21.36$ 139.0 sec	$J = 3.768^*$ 346.7^* sec
`Trapezoid`	$J = 2.234$ 23.88 sec	$J = 4.639$ 13.18 sec	$J = 21.74$ 663.7 sec	$J = 10.22$ 4929 sec

solvers, with trajectories matching within expected bounds. It only got stuck in a local minimum for one example: minimum-snap pendulum swing-up with unbounded actuation. Additionally, the re-meshing algorithm in DirCol5i does a good job at quantifying the transcription accuracy, and it matches the error estimates predicted by GPOPS-II, a commercially produced software.

- *Reliability:* One place where DirCol5i stands out, at least for this set of example problems, is reliability. It is the only solver that was able to compute a feasible trajectory for every single example problem. This was particularly apparent for the minimum-jerk and minimum-snap problems, where other solvers often failed to converge, or got stuck in bad local minima.

- *Speed:* In general, DirCol5i was slower than traditional methods, particularly for minimum-torque and minimum-torque-rate problems. That being said, it was still the fastest method for some of the minimum-jerk and minimum-snap problems.

- *Implementation:* It was simple to switch objective functions in DirCol5i, since the higher derivatives are always available. On the other hand, solving these problems with traditional solvers required a complete rewrite of the problem formulation for each different objective function. This also meant that, when using the traditional methods, it was difficult to determine whether numerical effects that were due to the objective function rather than the changes to the structure of the problem.

- *Scaling:* In many problems, the higher derivatives of the position trajectory are badly scaled. The scaling of such problems seems to have a larger negative effect on traditional methods, implemented with a chain integrator, than in DirCol5i.

- *Problem Size:* Implementing a minimum-jerk or minimum-snap trajectory optimization using standard methods (GPOPS-II, trapezoid collocation) requires adding a large number of decision variables to the problem without improving

the sparsity of the Jacobian. The resulting non-linear program is thus more difficult to solve.

In looking over the detailed results from these three example problems, there are a few questions of interest, which we answer below.

- *Why did traditional solvers have such a hard time at the minimum-jerk and minimum-snap problems?* DirCol5i is designed specifically for handling minimum-jerk and minimum-snap problems, while traditional solvers are not. In DirCol5i, changing the objective function from minimum-torque to minimum-snap does not fundamentally change the problem size or structure. On the other hand, traditional methods must introduce a large number of intermediate variables to make the change from minimum-torque to minimum-snap. This makes the problem larger and more difficult to solve.
- *Why was DirCol5i so slow on minimum-torque and minimum-torque-rate problems?* Trajectory optimization methods generally strike to construct a sparse non-linear program (NLP). On standard problems, for example minimum-torque, the NLP produced by DirCol5i is less sparse than that produced by traditional methods, such as GPOPS-II or trapezoid collocation. As a result, the DirCol5i version of the problem is harder to solve. That being said, there is no added complexity for DirCol5i when switching to a minimum-jerk or minimum-snap problem, unlike traditional solvers.
- *Why was GPOPS-II much faster on some problems?* The primary reason that GPOPS-II is faster has to do with code optimization, since GPOPS-II is a professionally produced software. In particular, GPOPS-II is careful about how it constructs the sparse non-linear program, and also about how it computes the

finite-differences for the gradients of the problem. Additionally, on minimum-torque and torque-rate problems, both GPOPS-II and trapezoid collocation have the advantage of producing non-linear programs that are more sparse than those produced by DirCol5i.

- *Why does the Chebyshev method work so well on the unbounded pendulum swing-up, and so poorly on the torque-limited swing-up?* The Chebyshev-Lobatto method implemented here is a global pseudospectral method. These methods work well for problems that have a smooth solution. In the torque-limited case, the solution is not smooth, as part of it lies on a constraint (the torque limit). A single polynomial cannot properly represent this 'kink' in the torque solution, and it causes numerical artifacts.

8.10.1 Summary

In this paper we present DirCol5i, a direct collocation (trajectory optimization) method. It is specialized for solving problems that have 1) non-trivial dynamics and 2) objective functions that include higher derivatives of state (such as acceleration, jerk, or snap).

We compared DirCol5i to traditional methods: trapezoidal direct collocation, Chebyshev-Lobatto pseudospectral collocation, and Radau Orthogonal Collocation (GPOPS-II). These methods were then tested on three example problems: planar quad rotor lateral transfer, pendulum swing-up, and cart-pole swing-up. In each case we tested several objective functions, including minimum: torque, torque-rate, jerk (derivative of position), and snap (derivative of jerk).

We found that DirCol5i was able to compute the correct solution to all problems, confirmed by reasonable convergence tests and agreement with solutions obtained by

other methods. We are able to compute solutions to a specified tolerance using automatic mesh refinement.

When solving standard trajectory optimization problems, those without higher-derivatives in the objective function, DirCol5i computes the correct answer, but it takes longer than traditional methods. This is primarily due to a reduction in sparsity of the non-linear program created by DirCol5i when compared to traditional methods.

When solving trajectory optimization problems that include higher-derivatives and have non-trivial dynamics, DirCol5i is superior to traditional methods in three ways.

1. DirCol5i is able to solve problem with minimum-jerk and minimum-snap directly. This is a huge advantage over traditional methods, which require a complicated reformulation of the problem.
2. DirCol5i was the only method that was able to compute a feasible solution for all problems. The traditional solvers all ran into numerical issues on at least one of the problems, typically minimum-jerk or minimum-snap.
3. Although DirCol5i was slower than traditional methods for standard problems, it was of comparable speed or faster on problems with minimum-jerk or minimum-snap in the objective.

8.11 Future Work

The initial work on DirCol5i, presented here, suggests that it is a useful tool for solving minimum-jerk and minimum-snap trajectory optimization problems for systems with non-trivial dynamics. In the future, we would like to study the performance of DirCol5i on a wider range of test problems. Additionally, the analysis presented here is largely

experimental: comparing DirCol5i to standard methods. It would be good to supplement these experiments with a more rigorous mathematical analysis.

Here we present DirCol5i as a fixed-order direct collocation method. These techniques could be extended to higher- and lower-order methods, which might be useful one some problems. For example, a problem with a smooth solution, like the quadrotor helicopter example, might benefit from fewer higher-order segments, while a problem with a discontinuous solution, like the torque-limited pendulum swing-up example, would benefit from a lower-order method.

DirCol5i represents the solution trajectory as Hermite Splines, which are represented by value and derivatives at the knot points. There are many other representations for splines, and it is possible that alternative representations might provide improved numerical properties. For example, Orthogonal collocation methods exclusively represent splines using Barycentric interpolation, storing the value of the spline at the roots of its associated orthogonal polynomial.

Finally, the software implementation of DirCol5i could be improved. As of now, it can only solve single-phase trajectory optimization problems, but DirCol5i (as an algorithm) can easily be applied to multi-phase problems. The code could be optimized further as well, for example only computing higher derivatives when they are actually required by the user.

8.12 Spline Derivations

At the core of DirCol5i is a set of splines, representing position, velocity, acceleration, and control. Here, we represent the position as a quintic Hermite spline, and the control

as a cubic Hermite spline. Velocity and acceleration are then computed by differentiating each segment of the position spline analytically. To keep the derivations readable, we will assume that the domain of interest has been mapped to the domain $\tau \in [-1, 1]$.

8.12.1 Domain Mapping

Although the spline derivations here are constructed for a specific domain $\tau \in [-1, 1]$, we can apply a simple transform to apply these equations to an arbitrary domain $t \in [a, b]$, where a and b would correspond to two successive knot points in the spline.

$$t = \frac{a+b}{2} + \tau \frac{b-a}{2} \tag{8.42}$$

$$\tau = 2\frac{t-a}{b-a} - 1 \tag{8.43}$$

The derivative expressions must also be scaled, with the scaling computed by the chain rule. Note that $\boldsymbol{x}$ is the true state, while $\bar{\boldsymbol{x}}$ is the state after domain mapping.

$$\frac{d^i}{dt^i}\boldsymbol{x} = \left(\frac{d\tau}{dt}\right)^i \frac{d^i}{d\tau^i}\bar{\boldsymbol{x}} = \left(\frac{2}{b-a}\right)^i \frac{d^i}{d\tau^i}\bar{\boldsymbol{x}} \tag{8.44}$$

$$\frac{d^i}{d\tau^i}\bar{\boldsymbol{x}} = \left(\frac{dt}{d\tau}\right)^i \frac{d^i}{dt^i}\boldsymbol{x} = \left(\frac{b-a}{2}\right)^i \frac{d^i}{dt^i}\boldsymbol{x} \tag{8.45}$$

8.12.2 Cubic Hermite Spline (Control)

The control trajectory is represented using a cubic Hermite spline, defined by its value and slope at each knot point[28]. In other words, the decision variables corresponding to the control trajectory are the control and the control rate at each knot point of the spline. For this section, we will assume that we are considering a single segment, where $\tau \in [-1, 1]$, and the boundary conditions are given by $\bar{\boldsymbol{u}}_L$, $\dot{\bar{\boldsymbol{u}}}_L$ for the lower side of the

segment and $\bar{\boldsymbol{u}}_U$, $\dot{\bar{\boldsymbol{u}}}_U$ for the upper side. Notice that $\bar{\boldsymbol{u}}$ is just a simple cubic polynomial, and $\dot{\bar{\boldsymbol{u}}}$ is just the derivative of that polynomial.

$$\begin{bmatrix} \bar{\boldsymbol{u}} \\ \dot{\bar{\boldsymbol{u}}} \end{bmatrix} = \begin{bmatrix} C_0 & C_1 & C_2 & C_3 \\ C_1 & 2C_2 & 3C_3 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \tau \\ \tau^2 \\ \tau^3 \end{bmatrix} \tag{8.46}$$

Now we need to compute the values for the coefficients $C_0 \ldots C_3$ that satisfy the boundary conditions, which can be expressed as a system of four linear equations.

$$\begin{bmatrix} 1 & -1 & 1 & -1 \\ 0 & 1 & -2 & 3 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{u}}_L \\ \dot{\bar{\boldsymbol{u}}}_L \\ \bar{\boldsymbol{u}}_U \\ \dot{\bar{\boldsymbol{u}}}_U \end{bmatrix} \tag{8.47}$$

This system can easily be solved analytically, providing a means for the coefficients as a function of the boundary conditions (*ie.* decision variables). These coefficients, combined with (8.46), are used for interpolation of the solution, as well as calculation of the control at the collocation points during optimization.

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 2 & 1 & 2 & -1 \\ -3 & -1 & 3 & -1 \\ 0 & -1 & 0 & 1 \\ 1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} \bar{\boldsymbol{u}}_L \\ \dot{\bar{\boldsymbol{u}}}_L \\ \bar{\boldsymbol{u}}_U \\ \dot{\bar{\boldsymbol{u}}}_U \end{bmatrix} \tag{8.48}$$

8.12.3 Quintic Hermite Spline (State)

The position trajectory is represented using a quintic Hermite spline, which is defined by its value, slope, and curvature at each knot point [28]. Thus, the decision variables in DirCol5i correspond to the position, velocity, and acceleration at each knot point. Given these values, we are interested in computing intermediate value for position, velocity, and acceleration. For this section, we will assume that we are considering a single segment, where $\tau \in [-1, 1]$, and the boundary conditions are given by $\bar{\boldsymbol{x}}_L$, $\dot{\bar{\boldsymbol{x}}}_L$, $\ddot{\bar{\boldsymbol{x}}}_L$ for the lower side of the segment and $\bar{\boldsymbol{x}}_U$, $\dot{\bar{\boldsymbol{x}}}_U$, $\ddot{\bar{\boldsymbol{x}}}_U$ for the upper side. Notice that the top set of equations is that of a simple quintic polynomial, and each successive row is obtained by differentiation of the preceding row.

$$\begin{bmatrix} \bar{\boldsymbol{x}} \\ \dot{\bar{\boldsymbol{x}}} \\ \ddot{\bar{\boldsymbol{x}}} \\ \dddot{\bar{\boldsymbol{x}}} \\ \ddddot{\bar{\boldsymbol{x}}} \end{bmatrix} = \begin{bmatrix} C_0 & C_1 & C_2 & C_3 & C_4 & C_5 \\ C_1 & 2C_2 & 3C_3 & 4C_4 & 5C_5 & 0 \\ 2C_2 & 6C_3 & 12C_4 & 20C_5 & 0 & 0 \\ 6C_3 & 24C_4 & 60C_5 & 0 & 0 & 0 \\ 24C_4 & 120C_5 & 0 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \tau \\ \tau^2 \\ \tau^3 \\ \tau^4 \\ \tau^5 \end{bmatrix} \tag{8.49}$$

Now we need to compute the values for the coefficients $C_0 \dots C_5$ that satisfy the boundary conditions prescribed by the position, velocity, and acceleration at each knot point.

The result is a system of six equations, which are linear in the coefficients.

$$\begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 \\ 0 & 1 & -2 & 3 & -4 & 5 \\ 0 & 0 & 2 & -6 & 12 & -20 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 2 & 6 & 12 & 20 \end{bmatrix} \cdot \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{bmatrix} = \begin{bmatrix} \bar{\boldsymbol{x}}_L \\ \bar{\dot{\boldsymbol{x}}}_L \\ \bar{\ddot{\boldsymbol{x}}}_L \\ \bar{\boldsymbol{x}}_U \\ \bar{\dot{\boldsymbol{x}}}_U \\ \bar{\ddot{\boldsymbol{x}}}_U \end{bmatrix} \tag{8.50}$$

Now we can solve for the coefficients analytically, the solution given below. We can now use these coefficients in (8.49) to interpolate the solution of the trajectory optimization, as well as compute the position and its derivatives at each knot point.

$$\begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \\ C_5 \end{bmatrix} = \frac{1}{16} \begin{bmatrix} 8 & 5 & 1 & 8 & -5 & 1 \\ -15 & -7 & -1 & 15 & -7 & 1 \\ 0 & -6 & -2 & 0 & 6 & -2 \\ 10 & 10 & 2 & -10 & 10 & -2 \\ 0 & 1 & 1 & 0 & -1 & 1 \\ -3 & -3 & -1 & 3 & -3 & 1 \end{bmatrix} \cdot \begin{bmatrix} \bar{\boldsymbol{x}}_L \\ \bar{\dot{\boldsymbol{x}}}_L \\ \bar{\ddot{\boldsymbol{x}}}_L \\ \bar{\boldsymbol{x}}_U \\ \bar{\dot{\boldsymbol{x}}}_U \\ \bar{\ddot{\boldsymbol{x}}}_U \end{bmatrix} \tag{8.51}$$

BIBLIOGRAPHY

[1] Aaron D. Ames, Kevin Galloway, Koushil Sreenath, and Jessy W. Grizzle. Rapidly Exponentially Stabilizing Control Lyapunov Functions and Hybrid Zero Dynamics. *IEEE Transactions on Automatic Control*, 59(4):876–891, apr 2014.

[2] Alec Banks, Jonathan Vincent, and Chukwudi Anyakoha. A review of particle swarm optimization. Part I: background and development. *Natural Computing*, 6(4):467–484, 2007.

[3] Victor M Becerra. PSOPT Optimal Control Solver User Manual, 2011.

[4] David a. Benson, Geoffrey T. Huntington, Tom P. Thorvaldsen, and Anil V. Rao. Direct Trajectory Optimization and Costate Estimation via an Orthogonal Collocation Method. *Journal of Guidance, Control, and Dynamics*, 29(6):1435–1440, 2006.

[5] Jean-Paul Berrut and Lloyd N. Trefethen. Barycentric Lagrange Interpolation. *SIAM Review*, 46(3):501–517, 2004.

[6] John T Betts. A Survey of Numerical Methods for Trajectory Optimization. *Journal of Guidance, Control, and Dynamics*, pages 1–56, 1998.

[7] John T. Betts. *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*. Siam, Philadelphia, PA, 2010.

[8] John T. Betts. SOS: Sparse Optimization Suite - User's Guide, 2013.

[9] P. a. Bhounsule, J. Cortell, a. Grewal, B. Hendriksen, J. G. D. Karssen, C. Paul, and a. Ruina. Low-bandwidth reflex-based control for lower power walking: 65 km on a single battery charge. *The International Journal of Robotics Research*, 33(10):1305–1321, jun 2014.

[10] P. a. Bhounsule, J. Cortell, A. Grewal, B. Hendriksen, J. G. D. Karssen, C. Paul, and A. Ruina. MULTIMEDIA EXTENSION # 1 International Journal of Robotics Research Low-bandwidth reflex-based control for lower power walking : 65 km on a single battery charge. *International Journal of Robotics Research*, 2014.

[11] Pranav a Bhounsule. *A controller design framework for bipedal robots: trajectory optimization and event-based stabilization*. PhD thesis, Cornell Univerisy, 2012.

[12] L. T. Biegler and V. M. Zavala. Large-scale nonlinear programming using IPOPT: An integrating framework for enterprise-wide dynamic optimization. *Computers and Chemical Engineering*, 33:575–582, 2009.

[13] Erin Catto. Box2D User Manual, 2013.

[14] By Christine Chevallereau, Gabriel Abba, Yannick Aoustin, Franck Plestan, E R Westervelt, Carlos Canudas-de wit, and J W Grizzle. RABBITA Testbed for Advanced Control Theory. *IEEE Control Systems Mag.*, 23:57–79, 2003.

[15] Christine Chevallereau, Dalila Djoudi, and Jessy W. Grizzle. Stable bipedal walking with foot rotation through direct regulation of the zero moment point. *IEEE Transactions on Robotics*, 24(2):390–401, 2008.

[16] M. Clerc and J. Kennedy. The particle swarm - explosion, stability, and convergence in a\nmultidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[17] Erwin Coumans. Bullet Physics SDK Manual, 2015.

[18] Hongkai Dai, Andres Valenzuela, and Russ Tedrake. Whole-body Motion Planning with Simple Dynamics and Full Kinematics. *International Conference on Humanoid Robots*, (JANUARY 2014):295–302, 2014.

[19] Christopher L. Darby, Divya Garg, and Anil V. Rao. Costate Estimation using Multiple-Interval Pseudospectral Methods. *Journal of Spacecraft and Rockets*, 48(5):856–866, sep 2011.

[20] Christopher L. Darby, WIlliam W. Hagar, and Anil V. Rao. An hp-adaptive pseudospectral method for solving optimal control problems. *Optimal Control Applications and Methods*, 32:476–502, 2011.

[21] Tobin A Driscoll, Nicholas Hale, and Lloyd N. Trefethen. *Chebfun Guide*. Pafnuty Publications, Oxford, 1 edition, 2014.

[22] R. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. *MHS'95. Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, 1995.

[23] G N Elnagar and M a Kazemi. Pseudospectral Chebyshev optimal control of constrained nonlinear dynamical systems. *Computational Optimization and Applications*, 217:195–217, 1998.

[24] Gamal Elnagar, Mohammad A. Kazemi, and Mohsen Razzaghi. The Pseudospectral Legendre Method for Discretizing Optimal Control Problems. *IEEE*, 40(October):1793–1796, 1995.

[25] Johannes Englsberger, Twan Koolen, Sylvain Bertrand, Jerry E Pratt, Christian Ott, and Alin Albu-schaffer. Trajectory generation for continuous leg forces during double support and heel-to-toe shift based on divergent component of motion. In *International Conference on Intelligent Robots and Systems*, Chicago, 2014.

[26] Johannes Englsberger, Christian Ott, Maximo A Roa, Alin Albu-sch, and Gerhard Hirzinger. Bipedal walking control based on Capture Point dynamics. In *International Conference on Intelligent Robots and Systems*, pages 4420–4427, San Francisco, 2011.

[27] Roy Featherstone. *Rigid Body Dynamics Algorithms*. 2008.

[28] David L Finn. MA 323 Geometric Modelling Course Notes : Day 09 Quintic Hermite Interpolation. Technical report, Rose-Hulman, 2004.

[29] Bengt Fornberg. *A practical guide to pseudospectral methods*. Cambridge University Press, 1996.

[30] Camila C Francolin, David A Benson, William W Hager, and Anil V Rao. Costate Estimation in Optimal Control Using Integral Gaussian Quadrature Orthogonal Collocation Methods. *Optimal Control Applications and Methods*, 2014.

[31] M Garcia, a Chatterjee, a Ruina, and M Coleman. The simplest walking model: stability, complexity, and scaling. *Journal of biomechanical engineering*, 120(2):281–8, apr 1998.

[32] D Garg, Ma Patterson, and Ww Hager. An Overview of Three Pseudospectral Methods for the Numerical Solution of Optimal Control Problems. *Advances in the . . .* , pages 1–17, 2009.

[33] Divya Garg, Michael Patterson, William W. Hager, Anil V. Rao, David a. Benson, and Geoffrey T. Huntington. A unified framework for the numerical solution of optimal control problems using pseudospectral methods. *Automatica*, 46(11):1843–1851, 2010.

[34] Alessandro Gasparetto and Vanni Zanotto. A technique for time-jerk optimal planning of robot trajectories. *Robotics and Computer-Integrated Manufacturing*, 24(3):415–426, 2008.

[35] Brian R Geiger, Joseph F Horn, Anthony M Delullo, and Lyle N Long. Optimal Path Planning of UAVs Using Direct Collocation with Nonlinear Programming. In *AIAA J. Guidance*, pages 1–13, 2006.

[36] Thomas Geijtenbeek, Michiel van de Panne, and a. Frank van der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics*, 32(6):1–11, nov 2013.

[37] Philip E Gill, Walter Murray, Michael A Saunders, E Gilit, and Michael A Saunderst. SNOPT : An SQP Algorithm for Large-Scale Constrained Optimization *. *SIAM Review*, 47(1):99–131, 2005.

[38] Gene H. Golub and John H. Welsch. Calculation of Gauss quadrature rules. *Mathematics of Computation*, 23(106):221–221, 1968.

[39] S. S. Grigoryan. The solution to the Painleve paradox for dry friction. *Doklady Physics*, 46(7):499–503, jul 2001.

[40] J. W. Grizzle, Jonathan Hurst, Benjamin Morris, Hae Won Park, and Koushil Sreenath. MABEL, a new robotic bipedal walker and runner. *Proceedings of the American Control Conference*, pages 2030–2036, 2009.

[41] Jessy W. Grizzle, Christine Chevallereau, Ryan W. Sinnet, and Aaron D. Ames. Models, feedback control, and open problems of 3D bipedal robotic walking. *Automatica*, 50(8):1955–1988, aug 2014.

[42] William W Hager and Anil V Rao. Gauss Pseudospectral Method for Solving Infinite-Horizon Optimal Control Problems. (August):1–9, 2010.

[43] Nicholas Hale and Alex Townsend. Fast and Accurate Computation of Gauss–Legendre and Gauss–Jacobi Quadrature Nodes and Weights. *SIAM Journal on Scientific Computing*, 35(2):A652–A674, 2013.

[44] N Hansen and a Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–95, jan 2001.

[45] Nikolaus Hansen. An analysis of mutative sigma-self-adaptation on linear fitness functions. *Evolutionary computation*, 14(3):255–75, jan 2006.

[46] C R Hargraves, C R Hargraves, S W Paris, S W Paris, C R Margraves, and S W Paris. Direct Trajectory Optimization Using Nonlinear Programming and Collocation. *AIAA J. Guidance*, 10(4):338–342, 1987.

[47] S. Javad Hasaneini, Chris J. B. Macnab, John E. a. Bertram, and Henry Leung. Optimal relative timing of stance push-off and swing leg retraction. *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3616–3623, nov 2013.

[48] A L Herman and B A Conway. Direct optimization using collocation based on high-order Gauss-Lobatto quadrature rules. {*AIAA*} *Journal of Guidance, Control, and Dynamics*, 19(3):522–529, 1996.

[49] Public Relations Division Honda. Asimo Technical Report. Technical Report September, 2007.

[50] David G. Hull. n of Optimal Control Problems into Parameter Optimization Problems. *1Journal of Guidance, Control, and Dynamics*, 20(1), 1997.

[51] Jonathan W Hurst. ATRIAS: Agile, Efficient, and Dynamic Bipedal Locomotion, 2016.

[52] David H. Jacobson and David Q. Mayne. *Differential Dynamic Programming*. Elsevier, 1970.

[53] Satoshi Kagami, Tomonobu Kitagawa, Koichi Nishiwaki, Tomomichi Sugihara, Masayuki Inaba, and Hirochika Inoue. A fast dynamically equilibrated walking trajectory generation method of humanoid robot. *Autonomous Robots*, 12(1):71–82, 2002.

[54] S Kajita, F Kanehiro, K Kaneko, K Fujiwara, K Harada, K Yokoi, and H Hirukawa. Resolved momentum control: humanoid motion planning based on the linear and angular momentum. *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, 2:1644–1650 vol.2, 2003.

[55] Shuuji Kajita and Kazuo Tan. Study of Dynamic Biped Locomotion on Rugged Terrain - Derivation and Application of the Linear Inverted Pendulum Mode. In *International Conference on Robotics and Automation*, number April, pages 1405–1411, 1991.

[56] Matthew Kelly and Andy Ruina. Non-linear robust control for inverted-pendulum 2D walking. In *International Conference on Robotics and Automation*, pages 4353–4358, Seattle, WA, 2015.

[57] Matthew Kelly, Matthew Sheen, and Andy Ruina. Off-line controller design for

reliable walking of Ranger. In *International Conference on Robotics and Automation*, Stockholm, Sweden, 2016.

[58] Matthew P. Kelly. TrajOpt - Trajectory Optimization Library for Matlab, 2015.

[59] Georges Klein and Jean-Paul Berrut. Linear barycentric rational quadrature. *BIT Numerical Mathematics*, 52(2):407–424, 2012.

[60] T. Koolen, T. de Boer, J. Rebula, A. Goswami, and J. Pratt. Capturability-based analysis and control of legged locomotion, Part 1: Theory and application to three simple gait models. *The International Journal of Robotics Research*, 31(9):1094–1113, jul 2012.

[61] Scott Kuindersma, Robin Deits, Maurice Fallon Andr, Hongkai Dai, Frank Permenter, Koolen Pat, and Marion Russ. Optimization-based Locomotion Planning , Estimation , and Control Design for the Atlas Humanoid Robot. *Autonomous Robots (accepted pending minor revision)*, 40(3):1–27, 2015.

[62] Scott Kuindersma, Frank Permenter, and Russ Tedrake. An Efficiently Solvable Quadratic Program for Stabilizing Dynamic Locomotion. In *International Conference on Robotics and Automation*, 2014.

[63] Arthur D. Kuo. Energetics of Actively Powered Locomotion Using the Simplest Walking Model. *Journal of Biomechanical Engineering*, 124(1):113, 2002.

[64] Jim Lambers. Romberg Integration. Technical report, University of Southern Mississippi, 2009.

[65] Dirk P. Laurie. Computation of Gauss-type quadrature formulas. *Journal of Computational and Applied Mathematics*, 127(1-2):201–217, 2001.

[66] Sung Hee Lee and Ambarish Goswami. A momentum-based balance controller for humanoid robots on non-level and non-stationary ground. *Autonomous Robots*, 33(4):399–414, 2012.

[67] R.I. Leine, B. Brogliato, and H. Nijmeijer. Periodic motion and bifurcations induced by the Painlevé paradox. *European Journal of Mechanics - A/Solids*, 21(5):869–896, jan 2002.

[68] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming*. Springer, third edit edition, 2008.

[69] Y Ma, F Borrelli, B Hencey, B Coffey, S Bengea, and P Haves. Model Predictive Control for the Operation of Building Cooling Systems. *IEEE Transactions on Control Systems Technology*, 20(3):796–803, 2012.

[70] Mathworks. Matlab Optimization Toolbox, 2014.

[71] Mathworks. Matlab Symbolic Toolbox, 2014.

[72] David Mayne. A Second-order Gradient Method for Determining Optimal Trajectories of Non-linear Discrete-time Systems. *International Journal of Control*, 3(1):85–95, 1966.

[73] Tad McGeer. Wobbling, toppling, and forces of contact. *American Journal of Physics*, 57(12):1089, 1989.

[74] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 2520–2525, 2011.

[75] Ian M Mitchell and Jeremy A Templeton. A Toolbox of Hamilton-Jacobi Solvers for Analysis of Nondeterministic Continuous and Hybrid Systems. *Hybrid Systems Computation and Control*, 3413:480–494, 2005.

[76] Igor Mordatch, Emanuel Todorov, and Zoran Popović. Discovery of complex behaviors through contact-invariant optimization. *ACM Transactions on Graphics*, 31(4):1–8, jul 2012.

[77] Mark W. Mueller, Markus Hehn, and Raffaello D'Andrea. A computationally efficient motion primitive for quadrocopter trajectory generation. 2015.

[78] D. M. Murray and S. J. Yakowitz. Differential dynamic programming and Newton's method for discrete optimal control problems. *Journal of Optimization Theory and Applications*, 43(3):395–414, 1984.

[79] Xue Bin NPeng, Glen Berseth, and Michiel van de Panne. Dynamic Terrain Traversal Skills Using Reinforcement Learning. In *SIGGRAPH*, 2015.

[80] Katsuhiko Ogata. *Modern Control Engineering*. Prentice Hall, 5th edition, 2010.

[81] Yizhar Or and Elon Rimon. Investigation of Painlev e ' s Paradox and Dynamic Jamming During Mechanism Sliding Motion. pages 1–20.

[82] Hae Won Park, Koushil Sreenath, Alireza Ramezani, and J. W. Grizzle. Switching control design for accommodating large step-down disturbances in bipedal robot walking. *Proceedings - IEEE International Conference on Robotics and Automation*, pages 45–50, 2012.

[83] Seymour V Parter. On the Legendre-Gauss-Lobatto Points and Weights. *Journal of Scientific Computing*, 14(4):347–355, 1999.

[84] Michael A Patterson and Anil V Rao. GPOPS II : A MATLAB Software for Solving Multiple-Phase Optimal Control Problems Using hp Adaptive Gaussian Quadrature Collocation Methods and Spa and rse Nonlinear Programming. 39(3):1–41, 2013.

[85] Pierre-Brice, Russ Tedrake, Scott Kuindersma, and Pierre-Brice Wieber. Modeling and Control of Legged Robots. In *Handbook of Robotics*, pages 1–53. Springer, 2 edition, 2016.

[86] G S Piperagkas, G Georgoulas, K E Parsopoulos, C D Stylios, and A C Likas. Integrating Particle Swarm Optimization with Reinforcement Learning in Noisy Problems. *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference*, pages 65–72, 2012.

[87] Michael Posa and Russ Tedrake. Direct Trajectory Optimization of Rigid Body Dynamical Systems Through Contact. *Algorithmic Foundations of Robotics X*, pages 527–542, 2013.

[88] J. Pratt. Virtual Model Control: An Intuitive Approach for Bipedal Locomotion. *The International Journal of Robotics Research*, 20(2):129–143, feb 2001.

[89] J. Pratt, T. Koolen, T. de Boer, J. Rebula, S. Cotton, J. Carff, M. Johnson, and P. Neuhaus. Capturability-based analysis and control of legged locomotion, Part 2: Application to M2V2, a lower-body humanoid. *The International Journal of Robotics Research*, 31(10):1117–1133, aug 2012.

[90] Jerry Pratt, John Carff, Sergey Drakunov, and Ambarish Goswami. Capture Point: A Step toward Humanoid Push Recovery. *2006 6th IEEE-RAS International Conference on Humanoid Robots*, pages 200–207, dec 2006.

[91] Large-scale Nonlinear Programming, Philip E Gill, Walter Murray, and Michael A Saunders. User ' s Guide for SNOPT Version 7 : Software for. pages 1–116, 2006.

[92] Alireza Ramezani, Jonathan W. Hurst, Kaveh Akbari Hamed, and J. W. Grizzle. Performance Analysis and Feedback Control of ATRIAS, A Three-Dimensional Bipedal Robot. *Journal of Dynamic Systems, Measurement, and Control*, 136(2):021012, 2013.

[93] Av Rao. A survey of numerical methods for optimal control. *Advances in the Astronautical Sciences*, 135:497–528, 2009.

[94] Charles Richter, Adam Bry, and Nicholas Roy. Polynomial Trajectory Planning for Aggressive Quadrotor Flight in Dense Indoor Environments. *Isrr*, (Isrr):1–16, 2013.

[95] Method O F Romberg. On the Method of Romberg Quadrature. *J. SIAM Numer. Anal.*, 2(2):250–259, 1965.

[96] I Michael Ross and Fariba Fahroo. Legendre pseudospectral approximations of optimal control problems. *New Trends in Nonlinear Dynamics and Control and their Applications*, 295(January):327–342, 2003.

[97] Andy Ruina. Nonholonomic stability aspects of piecewise holonomic systems. *Reports on Mathematical Physics*, 42(1-2):91–100, 1998.

[98] Cenk Oguz Saglam and Katie Byl. Robust Policies via Meshing for Metastable Rough Terrain Walking.

[99] Cenk Oguz Saglam and Katie Byl. Meshing Hybrid Zero Dynamics for Rough Terrain Walking. In *International Conference on Robotics and Automation*, number 1, pages 5718–5725, Seattle, WA, 2015.

[100] Yoshiaki Sakagami, Ryujin Watanabe, Chiaki Aoyama, Shinichi Matsunaga, Nobuo Higaki, and Kikuo Fujimura. The intelligent ASIMO: system overview and integration. *IEEE/RSJ International Conference on Intelligent Robots and System*, 3(October):2478–2483, 2002.

[101] S. F. P. Saramago and V. Steffen Jr. Optimization of the Trajectory Plannung of Robot Manipulators Taking Into Account the Dynamics of the System. *Mechanism and Machine Theory*, 33(7):883–894, 1998.

[102] A Shiriaev, J.W. Perram, and C. Canudas-de Wit. Constructive Tool for Orbital Stabilization of Underactuated Nonlinear Systems: Virtual Constraints Approach. *Automatic Control2*, 50(8):1164–1176, 5.

[103] K. Sreenath, H.-W. Park, I. Poulakakis, and J. W. Grizzle. A Compliant Hybrid Zero Dynamics Controller for Stable, Efficient and Fast Bipedal Walking on MABEL. *The International Journal of Robotics Research*, 30(9):1170–1193, sep 2011.

[104] Koushil Sreenath, Hae Won Park, Ioannis Poulakakis, and Jessy W. Grizzle. Embedding active force control within the compliant hybrid zero dynamics to achieve stable, fast running on MABEL. *The International Journal of Robotics Research*, 32(3):324–45, 2013.

[105] Manoj Srinivasan and Andy Ruina. Computer optimization of a minimal biped model discovers walking and running. *Nature*, 439(7072):72–5, jan 2006.

[106] Manoj Srinivasan and Andy Ruina. Idealized walking and running gaits minimize work. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 463(2086):2429–2446, oct 2007.

[107] Stelian Coros, Philippe Beaudoin, and Michiel van de Panne. Generalized biped walking control. 29(212):130:1–130:9, 2010.

[108] Russ Tedrake. Underactuated Robotics: Learning, Planning, and Control for Efficient and Agile Machines. Technical report, 2009.

[109] Russ Tedrake, Ian R Manchester, Mark Tobenkin, and John W Roberts. LQR-Trees : Feedback Motion Planning. In *Robotics: Science and Systems*, pages 1–28, 2010.

[110] Lloyd N Trefethen. A rational spectral collocation method with adaptively transformed chebyshev grid points . 28(5):1798–1811, 2006.

[111] Lloyd N Trefethen. *Approximation Theory and Approximation Practice*. SIAM, 2013.

[112] V a Tucker. Energetic cost of locomotion in animals. *Comparative biochemistry and physiology*, 34(4):841–846, 1970.

[113] V a Tucker. The energetic cost of moving about. *American scientist*, 63(4):413–9, 1975.

[114] Jacques Vlassenbroeck and R Van Dooren. A Chebyshev technique for solving nonlinear optimal control problems. *Automatic Control, IEEE . . .* , 33(4), 1988.

[115] O von Stryk, R. Bulirsch, and R. Bulirsh. Direct and indirect methods for trajectory optimization. *Annals of Operations Researach*, 37(1):357–373, 1992.

[116] O von Stryk, O von Stryk, O von Stryk, O von Stryk, and O von Stryk. User's guide for DIRCOL: A direct collocation method for the numerical solution of optimal control problems. *Lehrstuhl für Höhere Mathematik und Numerische*, (November), 1999.

[117] MIOMIR VUKOBRATOVIĆ and BRANISLAV BOROVAC. Zero-Moment Point - Thirty five years of its life. *International Journal of Humanoid Robotics*, 1(1):157–173, 2004.

[118] Miomir Vukobratović and Davor Juricic. Contribution to the Synthesis of Biped Gait. *Ieee Transactions on Bio-Medical Engineering*, (1):2–7, 1969.

[119] Andreas Wächter and Lorenz T. Biegler. *On the implementation of primal-dual interior point filter line search algorithm for large-scale nonlinear programming*, volume 106. 2006.

[120] Haiyong Wang and Shuhuang Xiang. On the Convergence Rate of Legendre Approximation. *Mathematics of Computation*, 81(278):861–877, 2011.

[121] E. R. Westervelt, J. W. Grizzle, and D. E. Koditschek. Hybrid zero dynamics of planar biped walkers. *IEEE Transactions on Automatic Control*, 48(1):42–56, 2003.

[122] N M Wilkey. OPTIMAL CONTROL OF A SATELLITE-ROBOT SYSTEM USING DIRECT COLLOCATION WITH NON-LINEAR PROGRAMMING. *Acta Astronautica*, 36(3):149–162, 1995.

[123] Paul Williams. Hermite-Legendre-Gauss-Lobatto Direct Transcription in Trajectory Optimization. *Journal of Guidance, Control, and Dynamics*, 32(4):1392–1395, 2009.

[124] T. Yang, E. R. Westervelt, a. Serrani, and J. P. Schmiedeler. A framework for the control of stable aperiodic walking inunderactuated planar bipeds. *Autonomous Robots*, 27(3):277–290, jul 2009.

[125] Kangkang Yin, Kevin Loken, and Michiel van de Panne. SIMBICON : Simple Biped Locomotion Control. In *SIGGRAPH*, 2007.

[126] Jason Yosinski, Jeff Clune, Diana Hidalgo, Sarah Nguyen, J.C. Zagal, and H. Lipson. Evolving Robot Gaits in Hardware: the HyperNEAT Generative Encoding Vs. Parameter Optimization. *Proceedings of the European Conference on Artificial Life (ECAL)*, pages 1–8, 2011.

[127] Petr Zaytsev, S Javad Hasaneini, and Andy Ruina. Two steps is enough : no need to plan far ahead for walking balance. In *International Conference on Robotics and Automation*, pages 6295–6300, 2015.

[128] V Zykov, J.C. Bongard, and H Lipson. Evolving dynamic gaits on a physical robot. *Proceedings of Genetic and . . .* , 2004.