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

MATTHEW KELLY †

Abstract. This paper is an introductory tutorial for numerical trajectory optimization with a focus on direct collocation methods. These methods are relatively simple to understand and effectively solve a wide variety of trajectory optimization problems. Throughout the paper we illustrate each new set of concepts by working through a sequence of four example problems. We start by using trapezoidal collocation to solve a simple one-dimensional toy-problem and work up to using Hermite–Simpson collocation to compute the optimal gait for a bipedal walking robot. Along the way, we cover basic debugging strategies and guidelines for posing well-behaved optimization problems. The paper concludes with a short overview of other methods for trajectory optimization. We also provide an electronic supplement that contains well-documented Matlab code for all examples and methods presented in this paper. Our primary goal is to provide the reader with the resources necessary to understand and successfully implement their own direct collocation methods.

1. Introduction.

What is trajectory optimization? Let's start with an example: imagine a satellite moving between two planets. We would use the term *trajectory* to describe the path the the satellite takes between the two planets. Usually, this path would include both state (*e.g.* position and velocity) and control (*e.g.* thrust) as functions of time. The term *trajectory optimization* refers to a set of methods that are used to find the best choice of trajectory, typically by selecting the inputs to the system, known as *controls*, as functions of time.

1.1. Overview.

Why read this paper? Our contribution is to provide a tutorial that covers all of the basics required to understand and implement direct collocation methods, while still being accessible to broad audience. Where possible, we teach through examples, both in this paper and in the electronic supplement.

This tutorial starts with a brief introduction to the basics of trajectory optimization (§1), and then it moves on to solve a simple example problem using trapezoidal collocation (§2). The next sections cover the general implementation details for trapezoidal collocation (§3) and Hermite–Simpson collocation (§4), followed by a section about practical implementation details and debugging (§5). Next there are three example problems: cart-pole swing-up (§6), five-link bipedal walking (§7), and minimum-work block-move (§8). The paper concludes with an overview of related optimization topics and a summary of commonly used software packages (§9).

This paper comes with a two-part electronic supplement, which is described in detail in the appendix §A. The first part is a general purpose trajectory optimization library, written in Matlab, that implements both trapezoidal direct collocation, Hermite–Simpson direct collocation, direct multiple shooting (4^{th}-order Runge–Kutta), and global orthogonal collocation (Chebyshev Lobatto). The second part of the supplement is a set of all example problems from this paper implemented in Matlab and solved with the afore-mentioned trajectory optimization library. The code in the supplement is well-documented and designed to be read in a tutorial fashion.

1.2. Notation.

For reference, these are the main symbols we will use throughout the tutorial and 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
$w_k = w\big(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k\big)$	**integrand of objective function at knot point** k
$\boldsymbol{f}_k = \boldsymbol{f}\big(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k\big)$	**system dynamics at knot point** k
$\dot{q} = \frac{d}{dt}q \qquad \ddot{q} = \frac{d^2}{dt^2}q$	**first and second time-derivatives of** q

*This work was supported by the National Science Foundation

†Cornell University, Ithaca, NY. (mpk72@cornell.edu). Questions, comments, or corrections to this document may be directed to that email address.

FIG. 1. *Illustration of the boundary conditions for the simple block move example.*

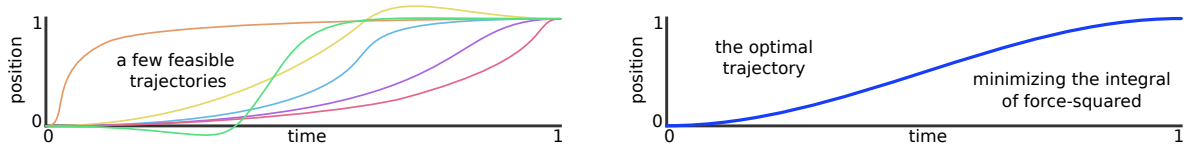

FIG. 2. *Comparison of feasible (left) and optimal (right) trajectories for the simple block move example.*

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.

1.3. A simple example. We will start by looking at a simple example: how to move a small block between two points, starting and finishing at rest, in a fixed amount of time. First, we will need to write down the *dynamics*, which describe how the system moves. In this case, we will model the block as a point-mass that travels in one dimension, and the *control* (input) to the system is simply the force applied to the block. Here we use x for position, ν for velocity, and u for control (force).

$$\dot{x} = \nu \qquad \dot{\nu} = u \qquad \textbf{system dynamics}$$

In this case, we would like the block to move one unit of distance in one unit of time, and it should be stationary at both start and finish. These requirements are illustrated in Figure 1 and are known as *boundary conditions*.

$$\begin{aligned} x(0) &= 0 \qquad & x(1) &= 1 \\ \nu(0) &= 0 \qquad & \nu(1) &= 0 \end{aligned} \qquad \textbf{boundary conditions}$$

A solution to a trajectory optimization problem is said to be *feasible* if it satisfies all of the problem requirements, known as *constraints*. In general, there are many types of constraints. For the simple block-moving problem we have only two types of constraints: the system dynamics and the boundary conditions. Figure 2 shows several feasible trajectories. The set of controls that produce feasible trajectories are known as *admissible* controls.

Trajectory optimization is concerned with finding the best of the feasible trajectories, which is known as the *optimal* trajectory, also shown in Figure 2. We use an *objective function* to mathematically describe what we mean by the 'best' trajectory. Later in this tutorial we will solve this block moving problem with two commonly used objective functions: minimal force squared (§2) and minimal absolute work (§8).

$$\min_{u(t),\,x(t),\,\nu(t)} \int_0^1 u^2(\tau)\,d\tau \qquad \textbf{minimum force-squared}$$

$$\min_{u(t),\,x(t),\,\nu(t)} \int_0^1 \left|u(\tau)\,\nu(\tau)\right| d\tau \qquad \textbf{minimum absolute work}$$

1.4. The trajectory optimization problem. There are many ways to formulate trajectory optimization problems [5, 45, 51]. Here we will restrict our focus to single-phase continuous-time trajectory optimization problems: ones where the system dynamics are continuous throughout the entire trajectory. A more general framework is described in [51] and briefly discussed in Section §9.9.

In general, an objective function can include two terms: a boundary objective $J(\cdot)$ and a path integral along the entire trajectory, with the integrand $w(\cdot)$. A problem with both terms is said to be in *Bolza form*. A problem with only the integral term is said to be in *Lagrange form*, and a problem with only a boundary term is said to be in *Mayer form*. [5] The examples in this paper are all in Lagrange form.

$$\min_{t_0,t_F,\boldsymbol{x}(t),\boldsymbol{u}(t)} \underbrace{J\big(t_0,t_F,\boldsymbol{x}(t_0),\boldsymbol{x}(t_F)\big)}_{\text{Mayer Term}} + \underbrace{\int_{t_0}^{t_F} w\big(\tau,\boldsymbol{x}(\tau),\boldsymbol{u}(\tau)\big)\,d\tau}_{\text{Lagrange Term}} \tag{1.1}$$

In optimization, we use the term *decision variable* to describe the variables that the optimization solver is adjusting to minimize the objective function. For the simple block moving problem the decision variables 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 (1.2-1.9). The first, and perhaps most important of these constraints is the system dynamics, which are typically non-linear and describe how the system changes in time.

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(t,\boldsymbol{x}(t),\boldsymbol{u}(t)\big) \qquad \textbf{system dynamics} \tag{1.2}$$

Next is the path constraint, which enforces 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.

$$\boldsymbol{h}\big(t,\boldsymbol{x}(t),\boldsymbol{u}(t)\big) \le \boldsymbol{0} \qquad \textbf{path constraint} \tag{1.3}$$

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.

$$\boldsymbol{g}\big(t_0,t_F,\boldsymbol{x}(t_0),\boldsymbol{x}(t_F)\big) \le \boldsymbol{0} \qquad \textbf{boundary constraint} \tag{1.4}$$

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 throughout the entire trajectory.

$$\boldsymbol{x}_{\text{low}} \le \boldsymbol{x}(t) \le \boldsymbol{x}_{\text{upp}} \qquad \textbf{path bound on state} \tag{1.5}$$

$$\boldsymbol{u}_{\text{low}} \le \boldsymbol{u}(t) \le \boldsymbol{u}_{\text{upp}} \qquad \textbf{path bound on control} \tag{1.6}$$

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, or that it reaches some goal region in state space.

$$t_{\text{low}} \le t_0 < t_F \le t_{\text{upp}} \qquad \textbf{bounds on initial and final time} \tag{1.7}$$

$$\boldsymbol{x}_{0,\text{low}} \le \boldsymbol{x}(t_0) \le \boldsymbol{x}_{0,\text{upp}} \qquad \textbf{bound on initial state} \tag{1.8}$$

$$\boldsymbol{x}_{F,\text{low}} \le \boldsymbol{x}(t_F) \le \boldsymbol{x}_{F,\text{upp}} \qquad \textbf{bound on final state} \tag{1.9}$$

1.5. Direct collocation method. Most methods for solving trajectory optimization problems can be classified as either *direct* or *indirect*. In this tutorial we will focus on direct methods, although we do provide a brief overview of indirect methods in Section §9.4. The key feature of a direct method is that is discretizes the trajectory optimization problem itself, typically converting the original trajectory optimization problem into a non-linear program (see §1.6). This conversion process is known as *transcription* and it is why some people refer to direct collocation methods as direct transcription methods.

In general, direct transcription methods are able to discretize a continuous trajectory optimization problem by approximating all of the continuous functions in the problem statement as polynomial splines. A *spline* is a function that is made up of a sequence of polynomials segments. Polynomials are used because they have two important properties: they can be represented by a small (finite) set of coefficients, and it is easy to compute integrals and derivatives of polynomials in terms of these coefficients.

Throughout this tutorial we will be studying two direct collocation methods in detail: trapezoidal collocation (§3) and Hermite–Simpson collocation (§4). We will also briefly cover a few other direct collocation techniques: direct single shooting (§9.5), direct multiple shooting (§9.6), and orthogonal collocation (§9.7).

1.6. Non-linear programming. Most direct collocation methods transcribe a continuous-time trajectory optimization problem into a *non-linear program*. A non-linear program is a special name given to a constrained parameter optimization problem that has non-linear terms in either its objective or constraint function. A typical formulation for a non-linear program is given below.

$$\min_{\boldsymbol{z}} J(\boldsymbol{z}) \qquad \text{subject to:} \tag{1.10}$$
$$\boldsymbol{f}(\boldsymbol{z}) = \boldsymbol{0}$$
$$\boldsymbol{g}(\boldsymbol{z}) \leq \boldsymbol{0}$$
$$\boldsymbol{z}_{\text{low}} \leq \boldsymbol{z} \leq \boldsymbol{z}_{\text{upp}}$$

In this tutorial we will not spend time examining the details of how to solve a non-linear program (see [34], [6], [11]), and instead will focus on the practical details of how to properly use a non-linear programming solver, such as those listed in Section §9.12.

In some cases, a direct collocation method might produce either a linear or quadratic program instead of a non-linear program. This happens when the constraints (including system dynamics) are linear and the objective function is linear (linear program) or quadratic (quadratic program). Both linear and quadratic programs are much easier to solve than non-linear programs, making them desirable for real-time applications, especially in robotics.

2. Block move example (minimum-force objective). In this section we continue with the simple example presented in the introduction: computing the optimal trajectory to move a block between two points.

2.1. Block move example: problem statement. We will model the block as a unit point mass that slides without friction in one dimension. The *state* of the block is its position x and velocity ν, and the *control* is the force u applied to the block.

$$\dot{x} = \nu \qquad\qquad \dot{\nu} = u \tag{2.1}$$

Next, we need to write the *boundary constraints* which describe the initial and final state of the block. Here we constrain the block to move from $x = 0$ at time $t = 0$ to $x = 1$ at time $t = 1$. Both the initial and final velocity are constrained to be zero.

$$\begin{aligned} x(0) &= 0 \qquad & x(1) &= 1 \\ \nu(0) &= 0 \qquad & \nu(1) &= 0 \end{aligned} \tag{2.2}$$

A trajectory that satisfies the system dynamics and the boundary conditions is said to be *feasible*, and the corresponding controls are said to be *admissible*. A trajectory is optimal if it minimizes an *objective function*. In general, we are interested in finding solution trajectories that are both feasible and optimal. Here we will use a common objective function: the integral of control effort squared. This cost function is desirable because it tends to produce smooth solution trajectories that are easily computed.

$$\min_{u(t),\, x(t),\, \nu(t)} \int_0^1 u^2(\tau)\, d\tau \tag{2.3}$$

2.2. Block move example: analytic solution. The solution to the simple block moving trajectory optimization problem (2.1-2.3) is given below, with a full derivation shown in Appendix B.

$$u^*(t) = 6 - 12t \qquad\qquad x^*(t) = 3t^2 - 2t^3 \tag{2.4}$$

The analytic solution is found using principles from calculus of variations. These methods convert the original optimization problem into a system of differential equations, which (in this special case) happen to have an analytic solution. It is worth noting that *indirect methods* for solving trajectory optimization work by using a similar principle: they analytically construct the necessary and sufficient conditions for optimality, and then solve then numerically. Indirect methods are briefly covered in Section 9.4.

2.3. Block move example: trapezoidal collocation. Now let's look at how to compute the optimal block-moving trajectory using trapezoidal collocation. We will need to convert the original continuous-time problem statement into a non-linear program. First, we need to discretize the trajectory, which gives us a finite set of decision variables. This is done by representing the continuous position $x(t)$ and velocity $v(t)$ by their values at specific points in time, known as *collocation points*.

$$t \to t_0 \dots t_k \dots t_N$$
$$x \to x_0 \dots x_k \dots x_N$$
$$\nu \to \nu_0 \dots \nu_k \dots \nu_N$$

Next, we need to convert the continuous system dynamics into a set of constraints that we can apply to the state and control at the collocation points. This is where the trapezoid quadrature (also known as the trapezoid rule) is used. The key idea is that the change in state between two collocation points is equal to the integral of the system dynamics. That integral is then approximated using trapezoidal quadrature, as shown below, where $h_k \equiv (t_{k+1} - t_k)$.

$$\dot{x} = \nu$$
$$\int_{t_k}^{t_{k+1}} \dot{x}\, dt = \int_{t_k}^{t_{k+1}} \nu\, dt$$
$$x_{k+1} - x_k \approx \tfrac{1}{2}(h_k)(\nu_{k+1} + \nu_k)$$

Simplifying and then applying this to the velocity equation as well, we arrive at a set of equations that allow us to approximate the dynamics between each pair of collocation points. These constraints are known as *collocation constraints*. These equations are enforced on every segment: $k = 0 \dots (N-1)$ of the trajectory.

$$x_{k+1} - x_k = \tfrac{1}{2}(h_k)\big(\nu_{k+1} + \nu_k\big) \tag{2.5}$$
$$\nu_{k+1} - \nu_k = \tfrac{1}{2}(h_k)\big(u_{k+1} + u_k\big) \tag{2.6}$$

The boundary conditions are straight-forward to handle: we simply apply them to the state at the initial and final collocation points.

$$\begin{aligned} x_0 &= 0 & x_N &= 1 \\ \nu_0 &= 0 & \nu_N &= 0 \end{aligned} \tag{2.7}$$

Finally, we approximate the objective function using trapezoid quadrature, converting it into a summation over the control effort at each collocation point:

$$\min_{u(t)} \int_{t_0}^{t_N} u^2(\tau)\, d\tau \quad \approx \quad \min_{u_0..u_N} \sum_{k=0}^{N-1} \tfrac{1}{2}(h_k)\big(u_k^2 + u_{k+1}^2\big) \tag{2.8}$$

2.4. Initialization. Most non-linear programming solvers require an initial guess. For easy problems, such as this one, a huge range of initial guesses will yield correct results from the non-linear programming solver. However, on difficult problems a poor initial guess can cause the solver to get "stuck" on a bad solution or fail to converge entirely. Section §5.1 provides a detailed overview of methods for constructing an initial guess.

For the block-moving example, we will simply assume that the position of the block (x) transitions linearly between the initial and final position. We then differentiate this initial position trajectory to compute the velocity (ν) and force (u) trajectories. Note that this choice of initial trajectory satisfies the system dynamics and position boundary condition, but it violates the velocity boundary condition.

$$x^{\text{init}}(t) = t \tag{2.9}$$
$$\nu^{\text{init}}(t) = \tfrac{d}{dt} x^{\text{init}}(t) = 1 \tag{2.10}$$
$$u^{\text{init}}(t) = \tfrac{d}{dt} \nu^{\text{init}}(t) = 0 \tag{2.11}$$

Once we have an initial trajectory, we can evaluate it at each collocation point to obtain values to pass to the non-linear programming solver.

$$x_k^{\text{init}} = t_k, \quad \nu_k^{\text{init}} = 1, \quad u_k^{\text{init}} = 0 \tag{2.12}$$

2.5. Block move example: non-linear program. We have used trapezoidal direct collocation to transcribe the continuous-time trajectory optimization problem into a non-linear program (constrained parameter optimization problem) (2.5)-(2.8). Now, we just need to solve it! Section §9.12 provides a brief overview of software packages that solve this type of optimization problem.

In general, after performing direct transcription, a trajectory optimization problem is converted into a non-linear programming problem. It turns out that, for this simple example, we actually get a quadratic program. This is because the constraints (2.5)-(2.7) are both linear, and the objective function (2.8) is quadratic. Solving a quadratic program is usually much easier than solving a non-linear program.

2.6. Block move example: interpolation. Let's assume that you've solved the non-linear program: you have a set of positions x_k, velocities, ν_k, and controls u_k that satisfy the dynamics and boundary constraints and that minimize the objective function. All that remains is to construct a spline (piece-wise polynomial function) that interpolates the solution trajectory between the collocation points. For trapezoidal collocation, it turns out that you use a linear spline for the control and a quadratic spline for the state. Section §3.4 provides a more detailed discussion and derivation of these interpolation splines.

3. Trapezoidal collocation method. Now that we've seen how to apply trapezoidal collocation to a simple example, we'll take a deeper look at using trapezoidal collocation to solve a generic trajectory optimization problem.

Trapezoidal collocation works by converting a continuous-time trajectory optimization problem into a non-linear program. This is done by using trapezoidal quadrature, also know as the trapezoid rule for integration, to convert each continuous aspect of the problem into a discrete approximation. In this section we will go through how this transformation is done for each aspect of a trajectory optimization problem.

3.1. Trapezoidal collocation: integrals. There are often integral expressions in trajectory optimization. Usually they are found in the objective function, but occasionally they are in the constraints as well. Our goal here is to approximate the continuous integral $\int w(\cdot)\,dt$ as a summation $\sum c_k w_k$. The key concept here is that the summation only requires the value of the integrand $w(t_k) = w_k$ at the collocation points t_k along the trajectory. This approximation is done by applying the trapezoid rule for integration between each collocation point, which yields the equation below, where $h_k = t_{k+1} - t_k$. [6]

$$\int_{t_0}^{t_F} w\big(\tau, \boldsymbol{x}(\tau), \boldsymbol{u}(\tau)\big)\; d\tau \quad \approx \quad \sum_{k=0}^{N-1} \tfrac{1}{2} h_k \cdot \big(w_k + w_{k+1}\big) \tag{3.1}$$

3.2. Trapezoidal collocation: system dynamics. One of the key features of a direct collocation method is that it represents the system dynamics as a set of constraints, known as *collocation constraints*. For trapezoidal collocation, the collocation constraints are constructed by writing the dynamics in integral form and then approximating that integral using trapezoidal quadrature [6].

$$\begin{gathered} \dot{\boldsymbol{x}} = \boldsymbol{f} \\ \int_{t_k}^{t_{k+1}} \dot{\boldsymbol{x}}\, dt = \int_{t_k}^{t_{k+1}} \boldsymbol{f}\, dt \\ \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \approx \tfrac{1}{2}\, h_k \cdot (\boldsymbol{f}_{k+1} + \boldsymbol{f}_k) \end{gathered}$$

This approximation is then applied between every pair of collocation points:

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = \tfrac{1}{2}\, h_k \cdot \big(\boldsymbol{f}_{k+1} + \boldsymbol{f}_k\big) \qquad k \in 0 \ldots (N-1) \tag{3.2}$$

Note that $\boldsymbol{x}_k$ is a decision variable in the non-linear program, while $\boldsymbol{f}_k = \boldsymbol{f}(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k)$ is the result of evaluating the system dynamics at each collocation point.

3.3. Trapezoidal collocation: constraints. In addition to the collocation constraints, which enforce the system dynamics, you might also have limits on the state and control, path constraints, and boundary constraints. These constraints are all handled by enforcing them at specific collocation points. For example, simple limits on state and control are approximated:

$$\boldsymbol{x} < \boldsymbol{0} \qquad \rightarrow \qquad \boldsymbol{x}_k < \boldsymbol{0} \quad \forall k \tag{3.3}$$

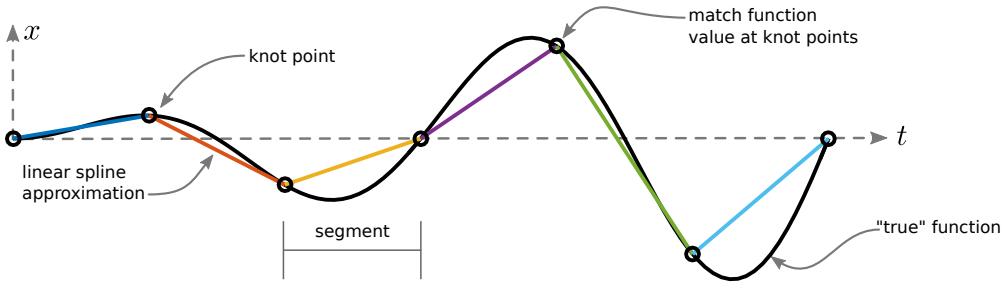

FIG. 3. *Function approximation using a linear spline.*

$$\boldsymbol{u} < \mathbf{0} \qquad \rightarrow \qquad \boldsymbol{u}_k < \mathbf{0} \quad \forall k \tag{3.4}$$

Path constraints are handled similarly:

$$\boldsymbol{g}(t, \boldsymbol{x}, \boldsymbol{u}) < \mathbf{0} \qquad \rightarrow \qquad \boldsymbol{g}(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k) < \mathbf{0} \quad \forall k \tag{3.5}$$

Boundary constraints are enforced at the first and last collocation points:

$$\boldsymbol{h}\big(t_0, \boldsymbol{x}(t_0), \boldsymbol{u}(t_0)\big) < \mathbf{0} \qquad \rightarrow \qquad \boldsymbol{h}\big(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0\big) < \mathbf{0} \tag{3.6}$$

Finally, there are two notes of caution with regard to constraints. First, trajectory optimization problems with path constraints tend to be much harder to solve than those without. The details are beyond the scope of this paper, but are well covered by Betts [6]. Second, in trapezoidal collocation the boundaries of the trajectory are always collocation points. There are some methods, such as those presented in Section §9.7, for which the trajectory boundaries are not collocation points. For these methods, special care must be taken when handling boundary constraints [3, 23].

3.4. Trapezoidal collocation: interpolation. Trapezoidal collocation works by approximating the control trajectory and the system dynamics as piece-wise linear functions, also known as a *linear splines*, shown in Figure 3. When constructing a spline, the term *knot point* is used to denote any point that joins two polynomial segments. For trapezoidal collocation, the knot points of the spline are coincident with the collocation points.

Let's start by constructing the control trajectory, which is a simple linear spline. We know both the time and control at each knot point, so it is a simple matter to derive the expression for $\boldsymbol{u}$ on the interval $t \in [t_k, t_{k+1}]$. To keep the math readable, let's define $\tau = t - t_k$ and $h_k = t_{k+1} - t_k$.

$$\boldsymbol{u}(t) \approx \boldsymbol{u}_k + \frac{\tau}{h_k}\left(\boldsymbol{u}_{k+1} - \boldsymbol{u}_k\right) \tag{3.7}$$

The state trajectory is represented by a *quadratic spline* — a piece-wise quadratic function. This might seem confusing, but it follows directly from the collocation equations (3.2). The trapezoidal collocation equations are exact when the system dynamics vary linearly between any two knot points, a fact that we use to approximate the dynamics over a single segment $t \in [t_k, t_{k+1}]$ as shown below.

$$\boldsymbol{f}(t) = \dot{\boldsymbol{x}}(t) \approx \boldsymbol{f}_k + \frac{\tau}{h_k}\left(\boldsymbol{f}_{k+1} - \boldsymbol{f}_k\right) \tag{3.8}$$

We are interested in $\boldsymbol{x}$ and not $\dot{\boldsymbol{x}}$, so we integrate both sides of the equation to get a quadratic expression for the state.

$$\boldsymbol{x}(t) = \int \dot{\boldsymbol{x}}(t)\, d\tau \approx \boldsymbol{c} + \boldsymbol{f}_k \tau + \frac{\tau^2}{2h_k}\left(\boldsymbol{f}_{k+1} - \boldsymbol{f}_k\right) \tag{3.9}$$

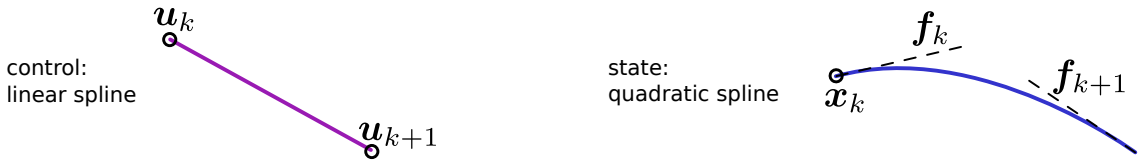

FIG. 4. *Illustration of the linear and quadratic spline segments that are used to approximate the control and state trajectories for trapezoidal collocation.*

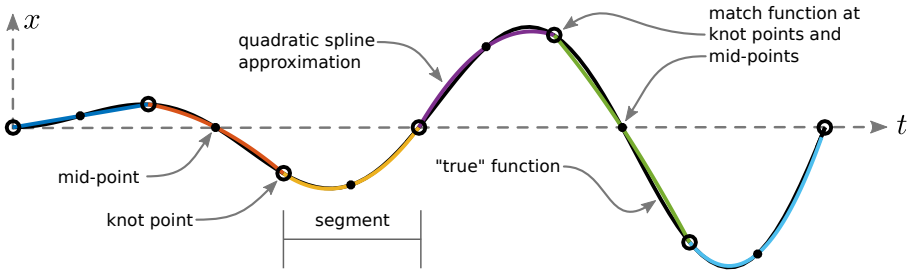

FIG. 5. *Function approximation using a quadratic spline. Notice that this approximation is far more accurate than the linear spline in Figure 3, for the same number of segments.*

We can solve for the constant of integration $\boldsymbol{c}$ by using the value of the state at the boundary $\tau = 0$ to get our final expression for the state.

$$\boldsymbol{x}(t) \approx \boldsymbol{x}_k + \boldsymbol{f}_k \tau + \frac{\tau^2}{2h_k}\left(\boldsymbol{f}_{k+1} - \boldsymbol{f}_k\right) \tag{3.10}$$

Figure 4 shows how a linear control segment and quadratic state segment are constructed. The spline equations (3.7) and (3.10) are specifically for trapezoidal collocation, since there is a one-to-one correspondence between the collocation equations and the interpolating spline. In general, if the control is a spline of order n, then the state is represented by a spline of order $n+1$ [6].

4. Hermite–Simpson collocation method. The Hermite–Simpson collocation is similar to trapezoidal collocation, but it provides a solution that is higher-order accurate. This is because trapezoidal collocation approximates the objective function and system dynamics as piece-wise linear functions, while Hermite–Simpson collocation approximates them as piece-wise quadratic functions, as shown in Figure 5. An additional benefit of the Hermite–Simpson collocation method is that the state trajectory is a cubic Hermite spline, which has a continuous first derivative.

4.1. Hermite–Simpson collocation: integrals. Integral expressions are common in trajectory optimization problems, especially in the objective function. The Hermite–Simpson collocation method approximates these integrals using Simpson quadrature. Simpson quadrature, also known as Simpson's rule for integration, works by approximating the integrand of the integral as a piece-wise quadratic function. This approximation is given below and derived in Appendix §C.

$$\int_{t_0}^{t_F} w(\tau)\, d\tau \quad \approx \quad \sum_{k=0}^{N-1} \frac{h_k}{6}\left(w_k + 4w_{k+\frac{1}{2}} + w_{k+1}\right)$$

4.2. Hermite–Simpson collocation: system dynamics. In any collocation method the *collocation constraints* are the set of constraints that are constructed to approximate the system dynamics. In the Hermite–Simpson collocation method we construct these constraints by rewriting the system dynamics in integral form: the change in state between any two knot points t_k should be equal to the integral of the

system dynamics $\boldsymbol{f}(\cdot)$ between those points.

$$\dot{\boldsymbol{x}} = \boldsymbol{f} \tag{4.1}$$

$$\int_{t_k}^{t_{k+1}} \dot{\boldsymbol{x}}\, dt = \int_{t_k}^{t_{k+1}} \boldsymbol{f}\, dt \tag{4.2}$$

The transcription from continuous dynamics to a set of collocation equations occurs when we approximate the continuous integral in (4.2) with Simpson quadrature and apply it between every pair of knot points.

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}_k = \tfrac{1}{6}\, h_k(\boldsymbol{f}_k + 4\boldsymbol{f}_{k+\frac{1}{2}} + \boldsymbol{f}_{k+1}) \tag{4.3}$$

For Hermite–Simpson collocation we actually need a second collocation equation, in addition to (4.3), to enforce the dynamics. This is because the dynamics at the mid-point of the segment $\boldsymbol{f}_{k+\frac{1}{2}}$ are a function of the state $\boldsymbol{x}_{k+\frac{1}{2}}$, which is not known *a priori*. We can compute the state at the mid-point by constructing an interpolant for the state trajectory (see Section §4.4) and then evaluating it at the mid-point of the interval.

$$\boldsymbol{x}_{k+\frac{1}{2}} = \frac{1}{2}\big(\boldsymbol{x}_k + \boldsymbol{x}_{k+1}\big) + \frac{h_k}{8}\big(\boldsymbol{f}_k - \boldsymbol{f}_{k+1}\big) \tag{4.4}$$

This second collocation equation (4.4) is special in that it can be computed explicitly in terms of the state at the knot points. Thus, it is possible to combine both equations (4.4) and (4.3) into a single complicated collocation constraint. When transcription of the system dynamics is performed using this single collocation constraint, the resulting formulation is said to be in *compressed form*. An alternative implementation is to create an additional decision variable for the state at the mid-point $\boldsymbol{x}_{k+\frac{1}{2}}$, and then use both (4.3) and (4.4) as constraint equations. When the collocation equations are formulated using this pair of constraints they are said to be in *separated form*.

There are a variety of trade-offs between the separated and compressed forms of Hermite–Simpson collocation, which are covered in detail in [6]. The general rule is that the separated form is better when working with a smaller number of segments, while the compressed form is better when the number of segments is large. Both constraint equations (4.3) and (4.4) can be found in Betts book [6].

4.3. Hermite–Simpson collocation: constraints. In addition to the collocation constraints, which enforce the system dynamics, you might also have limits on the state and control, path constraints, and boundary constraints. These constraints are all handled by enforcing them at specific collocation points. For example, simple limits on state and control are approximated:

$$\boldsymbol{x} < \mathbf{0} \qquad \rightarrow \qquad \begin{matrix} \boldsymbol{x}_k & < \mathbf{0} \\ \boldsymbol{x}_{k+\frac{1}{2}} & < \mathbf{0} \end{matrix} \tag{4.5}$$

$$\boldsymbol{u} < \mathbf{0} \qquad \rightarrow \qquad \begin{matrix} \boldsymbol{u}_k & < \mathbf{0} \\ \boldsymbol{u}_{k+\frac{1}{2}} & < \mathbf{0} \end{matrix} \tag{4.6}$$

Path constraints are handled similarly: they are applied at all collocation points, as shown below.

$$\boldsymbol{g}(t, \boldsymbol{x}, \boldsymbol{u}) < \mathbf{0} \qquad \rightarrow \qquad \begin{matrix} \boldsymbol{g}(t_k, \boldsymbol{x}_k, \boldsymbol{u}_k) & < \mathbf{0} \\ \boldsymbol{g}(t_{k+\frac{1}{2}}, \boldsymbol{x}_{k+\frac{1}{2}}, \boldsymbol{u}_{k+\frac{1}{2}}) & < \mathbf{0} \end{matrix} \tag{4.7}$$

Boundary constraints are enforced at the first and last knot points:

$$\boldsymbol{h}\big(t_0, \boldsymbol{x}(t_0), \boldsymbol{u}(t_0)\big) < \mathbf{0} \qquad \rightarrow \qquad \boldsymbol{h}\big(t_0, \boldsymbol{x}_0, \boldsymbol{u}_0\big) < \mathbf{0} \tag{4.8}$$

Just like in trapezoidal collocation, trajectory optimization problems with path constraints tend to be much harder to solve than those without [6]. Additionally, in Hermite–Simpson collocation the boundaries of the trajectory are always collocation points. There are some methods, such as those presented in Section §9.7, for which the trajectory boundaries are not collocation points. For these methods, special care must be taken when handling boundary constraints. [3, 23]

FIG. 6. *Illustration of the quadratic and cubic spline segments that are used to approximate the control and state trajectories for Hermite–Simpson collocation.*

4.4. Hermite–Simpson collocation: interpolation. After we've solved the non-linear program, we know the value of the state and control trajectories at each collocation point. The next step is to construct a continuous trajectory to interpolate the solution between the collocation points. Just like with trapezoidal collocation, we will use a polynomial interpolant that is derived from the collocation equations.

Hermite–Simpson collocation works by using Simpson quadrature to approximate each segment of the trajectory. As shown in Appendix §C, Simpson quadrature uses a quadratic segment, fitted through three uniformly spaced points, to approximate the integrand. In this case, we are approximating both the control and the system dynamics as quadratic over each segment of the trajectory.

The general equation for quadratic interpolation is given in *Numerical Recipes in C* [49], and reproduced below for a curve $\boldsymbol{u}(t)$ that passes through three points: $(t_A, \boldsymbol{u}_A)$, $(t_B, \boldsymbol{u}_B)$, and $(t_C, \boldsymbol{u}_C)$.

$$\boldsymbol{u}(t) = \frac{(t-t_B)(t-t_C)}{(t_A-t_B)(t_A-t_C)}\boldsymbol{u}_A + \frac{(t-t_A)(t-t_C)}{(t_B-t_A)(t_B-t_C)}\boldsymbol{u}_B + \frac{(t-t_A)(t-t_B)}{(t_C-t_A)(t_C-t_B)}\boldsymbol{u}_C \tag{4.9}$$

For our specific case, we can simplify this equation quite a bit, since our points are uniformly spaced. Let's start by using points k, $k+\frac{1}{2}$, and $k+1$ in place of A, B, and C. Next, recall from previous sections that $h_k = t_{k+1} - t_k$, $t_{k+\frac{1}{2}} = \frac{1}{2}(t_k + t_{k+1})$, and $\tau = t - t_k$. After making these substitutions, and doing some algebra, we can arrive at the following simplified equation for interpolating the control trajectory.

$$\boldsymbol{u}(t) = \frac{2}{h_k^2}\big(\tau - \tfrac{h_k}{2}\big)\big(\tau - h_k\big)\boldsymbol{u}_k \;-\; \frac{4}{h_k^2}\big(\tau\big)\big(\tau - h_k\big)\boldsymbol{u}_{k+\frac{1}{2}} \;+\; \frac{2}{h_k^2}\big(\tau\big)\big(\tau - \tfrac{h_k}{2}\big)\boldsymbol{u}_{k+1} \tag{4.10}$$

Hermite–Simpson collocation also represents the system dynamics $\boldsymbol{f}(\cdot) = \dot{\boldsymbol{x}}$ using quadratic polynomials over each segment. As a result, the quadratic interpolation formula that we developed for the control trajectory can also be applied to the system dynamics.

$$\boldsymbol{f}(t) = \dot{\boldsymbol{x}} = \frac{2}{h_k^2}\big(\tau - \tfrac{h_k}{2}\big)\big(\tau - h_k\big)\boldsymbol{f}_k \;-\; \frac{4}{h_k^2}\big(\tau\big)\big(\tau - h_k\big)\boldsymbol{f}_{k+\frac{1}{2}} \;+\; \frac{2}{h_k^2}\big(\tau\big)\big(\tau - \tfrac{h_k}{2}\big)\boldsymbol{f}_{k+1} \tag{4.11}$$

Usually we are interested in obtaining an expression for the state trajectory $\boldsymbol{x}(t)$ rather than its derivative $\dot{\boldsymbol{x}}(t)$. To get the state trajectory, we simply integrate (4.11), after rearranging it to be in standard polynomial form.

$$\boldsymbol{x}(t) = \int \dot{\boldsymbol{x}}\, dt = \int \left[\boldsymbol{f}_k + \left(-3\boldsymbol{f}_k + 4\boldsymbol{f}_{k+\frac{1}{2}} - \boldsymbol{f}_{k+1} \right)\left(\frac{\tau}{h_k}\right) + \left(2\boldsymbol{f}_k - 4\boldsymbol{f}_{k+\frac{1}{2}} + 2\boldsymbol{f}_{k+1} \right)\left(\frac{\tau}{h_k}\right)^2 \right] dt \tag{4.12}$$

We can compute the integral using basic calculus, and then solve for the constant of integration using the boundary condition $\boldsymbol{x}(t_k) = \boldsymbol{x}_k$. The resulting expression is given below, which allows us to interpolate the state trajectory.

$$\boldsymbol{x}(t) = \boldsymbol{x}_k \;+\; \boldsymbol{f}_k\left(\frac{\tau}{h_k}\right) \;+\; \frac{1}{2}\left(-3\boldsymbol{f}_k + 4\boldsymbol{f}_{k+\frac{1}{2}} - \boldsymbol{f}_{k+1} \right)\left(\frac{\tau}{h_k}\right)^2 \;+\; \frac{1}{3}\left(2\boldsymbol{f}_k - 4\boldsymbol{f}_{k+\frac{1}{2}} + 2\boldsymbol{f}_{k+1} \right)\left(\frac{\tau}{h_k}\right)^3 \tag{4.13}$$

The interpolants for the state and control trajectories are illustrated in Figure 6.

5. Practical considerations. This section of the paper provides an overview of several important topics that are related to trajectory optimization in general, rather than to some specific method. We start with some practical suggestions about how to initialize trajectory optimization problems, followed by two sections that explain how to check the accuracy of a given solution. We conclude by looking at some common bugs that show up in trajectory optimization code and how to go about fixing them.

5.1. Initialization. Nearly all trajectory optimization techniques require a good initial guess to begin the optimization. In the best case, a good initialization will ensure that the solver rapidly arrives at the globally optimal solution. In the worst case, a bad initialization can cause the non-linear programming solver to fail to solve an otherwise correct optimization problem.

To understand these concepts, let's use an analogy: 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 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 a fundamental problem with non-linear programming solvers: they cannot always find a solution, and if they do find a solution, it is only guaranteed to be locally optimal.

The best initializations for trajectory optimization usually require some problem-specific knowledge, but there are a few general approaches that can be useful. In this way, initialization is more of an art than a science. One good practice is to try several different initialization strategies and check that they all converge to the same solution. See §5.4 for some debugging suggestions to help determine if a solution is converging correctly.

One of the simplest initialization techniques is to 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 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. Our favorite technique is to simplify the trajectory optimization problem until we can get a reasonable solution using a simple initialization technique. Then we use the solution of the simplified problem to initialize the original problem. If this doesn't work, then we 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, let's say that you want to find a minimum-work trajectory for a walking robot. This objective function is challenging optimize (see §8), and there are some difficult non-linear constraints: foot clearance, contact forces, and walking speed. Start by replacing the objective function with something simple: a minimum torque-squared objective (like the five-link biped example, §7). Next, remove most of the constraints and replace the non-linear dynamics with simple kinematics (joint acceleration = joint torque). Solve this problem, and then use the solution to initialize a slightly harder version of the problem where you've added back in some of the constraints. You can then repeat this process until you have a solution to your original trajectory optimization problem. This process is also a good way to find bugs in both your problem statement and code.

5.2. Mesh refinement. The direct transcription process approximates a trajectory using polynomial splines, which allows the trajectory optimization problem to be converted into a non-linear program. The collocation constraints in the resulting non-linear program are acting as implicit Runge–Kutta integration schemes [6]. Just like any integration scheme, there are numerical errors associated with the choice of time step and method order. Using short time steps (dense mesh) and a high-order method will result in an accurate approximation, but at a significant computational cost.

Mesh refinement is the process by which a trajectory optimization problem is solved on a sequence of different collocation meshes, also known as *collocation grids*. The mesh (grid) refers to the choice of discretization along the trajectory. Generally, the first mesh is coarse, with a small number of collocation points and (or) a lower-order collocation method. Subsequent meshes have more points and (or) higher-order collocation methods. This iterative strategy is implemented to obtain the most accurate solution with the least amount of computational effort: the solutions using the initial meshes are easy to solve but inaccurate,

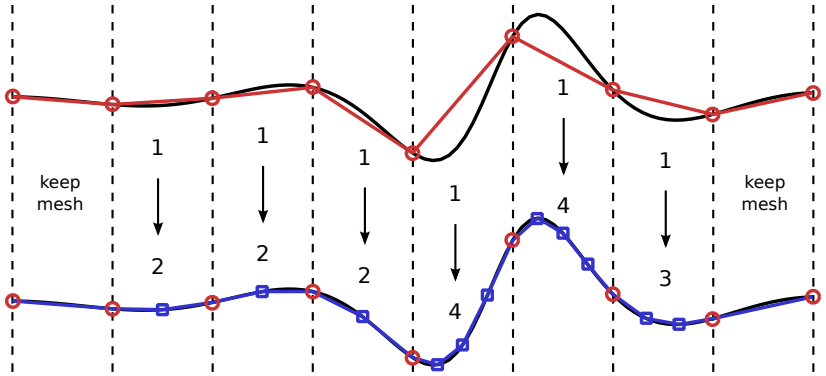

FIG. 7. *Illustration of mesh refinement by sub-dividing segments. The number of sub-segments is determined by the peak error in each segment.*

while the solutions on subsequent meshes are more costly to compute but more accurate.

Figure 7 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. The 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.

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 segment. Such algorithms are referred to as hp-adaptive meshing. The decision to sub-divide the mesh or to increase the polynomial order is made by examining the error profile within a single segment. If there is a spike in the error, then the segment is sub-divided, otherwise the polynomial order is increased, for example switching from trapezoidal to Hermite–Simpson collocation [16], [45], and [6].

5.3. Error analysis. There are two types of numerical errors that are present in the solution of a trajectory optimization problem: transcription errors and errors in the solution to the non-linear program. Here we will focus on the accuracy of the transcription process, quantifying how much error was introduced by the choice of discretization (both method and grid). We can then use these error estimates to compute a new discretization, as described in §5.2.

There are many possible error metrics for trajectory optimization [6]. 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 precisely satisfy the system dynamics:

$$\dot{\boldsymbol{x}}^*(t) = \boldsymbol{f}\big(t, \boldsymbol{x}^*(t), \boldsymbol{u}^*(t)\big)$$

From this, we can construct an expression for the error in the solution to the system dynamics along the candidate trajectory. It is important that the solution $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ is evaluated using method consistent interpolation [6].

$$\boldsymbol{\varepsilon}(t) = \dot{\boldsymbol{x}}(t) - \boldsymbol{f}\big(t, \boldsymbol{x}(t), \boldsymbol{u}(t)\big)$$

This error $\boldsymbol{\varepsilon}(t)$ will be zero at each collocation point and non-zero elsewhere. We can compute the integral of the error $\boldsymbol{\varepsilon}(t)$ numerically to determine how far the candidate solution (polynomial spline) may have deviated from the true solution along each dimension of the state. The following expression for the error is typically evaluated using Rhomberg quadrature [6].

$$\boldsymbol{\eta}_k = \int_{t_k}^{t_{k+1}} |\boldsymbol{\varepsilon}(\tau)|\, d\tau$$

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 (§5.2) so that your optimization converges to an optimal solution that satisfies the continuous dynamics. See [6] and [16] for additional details about how to compute error estimates and perform mesh refinement.

5.4. 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 [6] also provides a good list of debugging suggestions.

One particularly tricky type of bug occurs when there is a family of optimal solutions, rather than a single unique solution. This causes a failure to converge because the optimization is searching for a locally optimal solution, which it never finds because many solutions are equally good. The fix is to modify the problem statement so that there is a unique solution. One simple way to do this is to add a small regularization term to the cost function, such as the integral of control squared along the trajectory. This puts a shallow bowl in the objective function, forcing a unique solution. Trajectory optimization problems with non-unique solutions often have *singular arcs*, which occur when the optimal control is not uniquely defined by the objective function. A more formal treatment of singular arcs is provided in [5] and [6].

A trajectory optimization problem with a non-smooth *solution* (control) 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 (§8). There three basic ways to deal with a discontinuous solution (control). The first is to do mesh refinement (§5.2) so that there are many short segments near the discontinuity. The second is to slightly modify the problem, typically by introducing a smoothing term, such that the solution is numerically stiff but not discontinuous. This second approach was used in [55]. The third approach is to solve the problem using a multi-phase method (see §9.9), such that the control in each phase of the trajectory is continuous, and discontinuities occur between phases.

Another common cause of poor convergence in the non-linear programming solver occurs when then objective and constraint functions are not consistent (see §5.5). There are many sources of inconsistency that find their way into trajectory optimization problems: discontinuous functions (`abs()`, `min()`, `max()`...), random number generators, variable step (adaptive) integration, iterative root finding, and table interpolation. All of these will cause significant convergence problems if placed inside of a standard non-linear programming solver. Section §5.5 covers some methods for handling inconsistent functions.

If the non-linear programming solver returns saying that the problem is infeasible, there are two possible scenarios. The first is that your problem statement is actually impossible: you have contradictory constraints. In this 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? You can also debug this type of failure by removing constraints from the problem until it converges and then adding constraints back one at a time.

The second cause of an infeasible report from a non-linear programming solver is when a complicated optimization problem is initialized with a poor guess. In this 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 §5.1 to compute a better initialization.

It is challenging to determine 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 this is the globally optimal solution. Another such heuristic is to use different transcription methods and check that all methods all converge to the same solution.

5.5. 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 [50], IPOPT [10], and FMINCON [36], 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 [6]. This is essentially like saying that the function must have no logical branches, be deterministic, and have outputs that vary smoothly with the inputs.

For example, the `abs()` function is not consistent, because of the discontinuity in the derivative at the

origin. The functions `min()` and `max()` are also not consistent. Imaging a function with two widely spaced peaks. A small change in the shape of the function could cause the maximum value to jump from one peak $f(x_1)$ to a second peak $f(x_2)$. The problem here is in the gradients: when the peak moves, the gradient $\frac{\partial f}{\partial x_1}$ jumps to zero, and the gradient $\frac{\partial f}{\partial x_2}$ jumps from zero to some non-trivial value.

There is a neat trick that allows many inconsistent functions (such as `abs()`, `min()`, and `max()`) to be implemented consistently by introducing extra decision variables (known as *slack variables*) and constraints to your problem. An example is given in Section §8, showing how to correctly implement the `abs()` function. This topic is also covered by Betts [6]. An alternative way to handle such functions is to use smoothing, which is also demonstrated in the block-moving example in §8.

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 correct way to implement a root-finding method inside of an optimization is to use a fixed number of iterations. Likewise, and a variable-step integration method should be replaced with a fixed-step method [6].

There are many situations where evaluating the dynamics or constraint functions require a table look-up, for example computing the lift force generated by an airfoil. Linear interpolation of a table has a discontinuous derivative when switching between two different table entries. The fix is to switch to an interpolation scheme that has continuous derivatives. Continuous first derivatives are required by most solvers when computing gradients (first partial derivatives). Solvers that compute both gradients and Hessians (second partial derivatives) will require continuous second derivatives [6].

One final source of inconsistency is the use of a time-stepping simulators such as Bullet [14] or Box2d [12] to compute the system dynamics. The contact solvers in these simulators are inconsistent, which then leads to poor convergence in the non-linear program. The best way to address this source of inconsistency is to rewrite the system dynamics. If the sequence of contacts is known and the dynamics can be described as a simple hybrid system, then you can use multi-phase trajectory optimization to compute the solution (see §9.9). For more complex systems where the contact sequence is unknown, you can use through-contact trajectory optimization to compute the solution [39, 47] (see §9.10). If you need to use the time-stepping simulator, then you can use some of the methods developed by the computer graphics community [1,33,60,61].

6. Cart-pole swing-up example. The cart-pole system is commonly used as a teaching tool in both introductory controls and in trajectory optimization. The system comprises a cart that travels along a horizontal track and a pendulum that hangs freely from the cart. There is a motor that drives the cart forward and backward along the track. It is possible to move the cart in such a way that the pendulum, initially hanging below the cart at rest, is swung up to a point of inverted balance above the cart. In this section, we will use direct collocation to compute the minimum-force trajectory to perform this so-called 'swing-up' maneuver.

6.1. Cart-pole example: 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 derive by hand, although for more complicated systems it is generally a good idea to use a computer algebra package instead.

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 the length of the pole and acceleration due to gravity are ℓ and g, as shown in Figure 8. The dynamics ($\ddot{q}_1$ and $\ddot{q}_2$) for the cart-pole system are shown below.

$$\ddot{q}_1 = \frac{\ell\, m_2 \sin(q_2)\, \dot{q}_2^2 + u + m_2\, g\, \cos(q_2)\, \sin(q_2)}{m_1 + m_2\left(1 - \cos^2(q_2)\right)} \tag{6.1}$$

$$\ddot{q}_2 = -\frac{\ell\, m_2 \cos(q_2)\, \sin(q_2)\, \dot{q}_2^2 + u\, \cos(q_2) + (m_1 + m_2)\, g\, \sin(q_2)}{\ell\, m_1 + \ell m_2\left(1 - \cos^2(q_2)\right)} \tag{6.2}$$

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

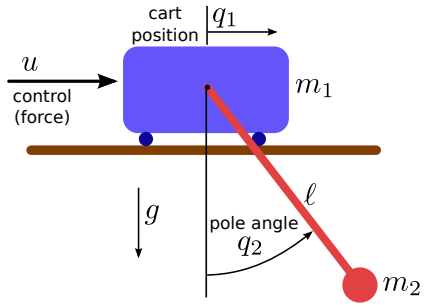

FIG. 8. *Physical model for the cart-pole example problem. The pendulum is free to rotate about its support point on the cart.*

the state. Note that $\ddot{q}_1$ and $\ddot{q}_2$ are defined in (6.1) and (6.2).

$$\boldsymbol{x} = \begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} \qquad \dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}, u) = \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix}$$

6.2. Cart-pole example: objective function. For this example we will use one of the more common objective functions in trajectory optimization: the integral of the actuator-effort (control) squared.

$$J = \int_0^T u^2(\tau)\, d\tau \tag{6.3}$$

This objective function (6.3) tends to produce smooth trajectories, which 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 benefit of smooth trajectories is that they tend to be easier to stabilize with conventional controllers when implemented on a real system.

6.3. Cart-pole example: boundary constraints. Many trajectory optimization problems include *boundary constraints*, which restrict the state of the system at the boundaries of the trajectory. 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 during its swing-up maneuver. The (constant) boundary constraints for this situation are given below.

$$\begin{aligned} q_1(t_0) &= 0 & q_1(t_F) &= d \\ q_2(t_0) &= 0 & q_2(t_F) &= \pi \\ \dot{q}_1(t_0) &= 0 & \dot{q}_1(t_F) &= 0 \\ \dot{q}_2(t_0) &= 0 & \dot{q}_2(t_F) &= 0 \end{aligned}$$

6.4. Cart-pole example: 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.

$$\begin{aligned} -d_{\max} \leq q_1(t) \leq d_{\max} \\ -u_{\max} \leq u(t) \leq u_{\max} \end{aligned}$$

6.5. Cart-pole example: trapezoidal collocation. We can collect all of the equations in this section and combine them with the trapezoidal collocation method from §3, to write down the cart-pole swing-up

problem as a non-linear program.

minimize:

$$J = \sum_{k=0}^{N-1} \tfrac{h_k}{2}\big(u_k^2 + u_{k+1}^2\big) \tag{6.4}$$ **objective function**

decision variables:

$$\boldsymbol{x}_0 ... \boldsymbol{x}_N \quad u_0 \dots u_N \tag{6.5}$$

subject to:

$$\tfrac{1}{2}h_k\big(\boldsymbol{f}_{k+1} + \boldsymbol{f}_k\big) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \qquad k \in 0 \dots (N-1) \tag{6.6}$$ **collocation constraints**

$$-d_{\max} \le q_1 \le d_{\max} \tag{6.7}$$ **path constraints**

$$-u_{\max} \le u \le u_{\max} \tag{6.8}$$ **path constraints**

$$\boldsymbol{x}_0 = \boldsymbol{0} \quad \boldsymbol{x}_N = [d, \pi, 0, 0]^T \tag{6.9}$$ **boundary constraints**

Note that $h_k = t_{k+1} - t_k$. Here, we will use 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.

6.6. Cart-pole example: Hermite–Simpson collocation. We can also use Hermite-Simpson collocation (§4) to construct a non-linear program for the cart-pole swing-up problem. This is similar to the trapezoidal collocation, but it uses a quadratic (rather than linear) spline to approximate the dynamics and control. Here we will use the separated form of the Hermite–Simpson method, which requires including collocation points for the state and control at the mid-point of each segment $t_{k+\frac{1}{2}}$ (see §4.2).

minimize:

$$J = \sum_{k=0}^{N-1} \tfrac{h_k}{6}\big(u_k^2 + 4u_{k+\frac{1}{2}}^2 + u_{k+1}^2\big) \tag{6.10}$$ **objective function**

decision variables:

$$\boldsymbol{x}_0,\, \boldsymbol{x}_{0+\frac{1}{2}} \dots \boldsymbol{x}_N \qquad u_0,\, u_{0+\frac{1}{2}} \dots u_N$$

subject to:

$$\boldsymbol{x}_{k+\frac{1}{2}} = \tfrac{1}{2}\big(\boldsymbol{x}_k + \boldsymbol{x}_{k+1}\big) + \tfrac{h_k}{8}\big(\boldsymbol{f}_k - \boldsymbol{f}_{k+1}\big) \qquad k \in 0 \dots (N-1) \tag{6.11}$$ **interpolation constraints**

$$\tfrac{h_k}{6}\big(\boldsymbol{f}_k + 4\boldsymbol{f}_{k+\frac{1}{2}} + \boldsymbol{f}_{k+1}\big) = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k \qquad k \in 0 \dots (N-1) \tag{6.12}$$ **collocation constraints**

$$-d_{\max} \le q_1 \le d_{\max} \tag{6.13}$$ **path constraints**

$$-u_{\max} \le u \le u_{\max} \tag{6.14}$$ **path constraints**

$$\boldsymbol{x}_0 = \boldsymbol{0} \quad \boldsymbol{x}_N = [d, \pi, 0, 0]^T \tag{6.15}$$ **boundary constraints**

6.7. Cart-pole example: initialization. The cart-pole swing-up problem is a boundary value problem: we are given the initial and final state, and our task is to compute an optimal trajectory 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 simple guess works well for this problem, despite its failure satisfy the system dynamics.

$$\boldsymbol{x}_{\text{guess}}(t) = \frac{t}{T}\begin{bmatrix} d \\ \pi \\ 0 \\ 0 \end{bmatrix} \qquad u_{\text{guess}}(t) = 0 \tag{6.16}$$

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 (6.16) at each knot point t_k (and the mid-point $t_{k+\frac{1}{2}}$ for Hermite-Simpson collocation).

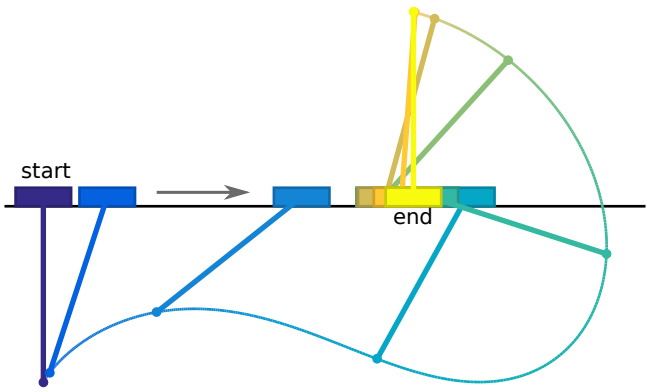

FIG. 9. *Illustration of the optimal trajectory for the cart-pole swing-up example. The frames are uniformly spaced in time, moving from blue (dark) to yellow (light) as the trajectory progresses.*

6.8. Cart-pole example: results. Here 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 §E.1. We computed the solution in Matlab, on a regular desktop computer[1], using the code provided in the electronic supplement (§A). The non-linear program was solved by FMINCON in 5.91 seconds (71 iterations) using default convergence settings.

Figure 9 shows a stop-action animation of the swing-up maneuver, with uniformly spaced frames. The same solution is shown in Figure 10 as plots of state and control versus time. Finally, Figure 11 shows the error estimates along the trajectory.

Notice that the error metrics 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 approximating the system dynamics. A more sophisticated 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: we can make small changes to the initial guess or the direct transcription method and get the same basic answer out. If we can change some of the problem parameters it can make things more difficult. For example, if we increase the duration T will causes the optimal solution to include several swings back-and-forth before the final swing-up. As a result, the optimization problem has many local minima, one for each (incorrect) number of swings back and forth. Another way to make the optimization more challenging is to reduce the actuator limits $u_{\max}$. If these limits are made small enough, then the optimal solution will no longer be smooth. To solve it, we would need to re-mesh the discretization (time) grid to place additional points near the discontinuities in the force trajectory. An alternative way to address the discontinuity in the control would be to rewrite the problem as a multi-phase problem, but this is beyond the scope of this paper.

[1]processor: 3.4GHz quad-core Intel i5-3570K

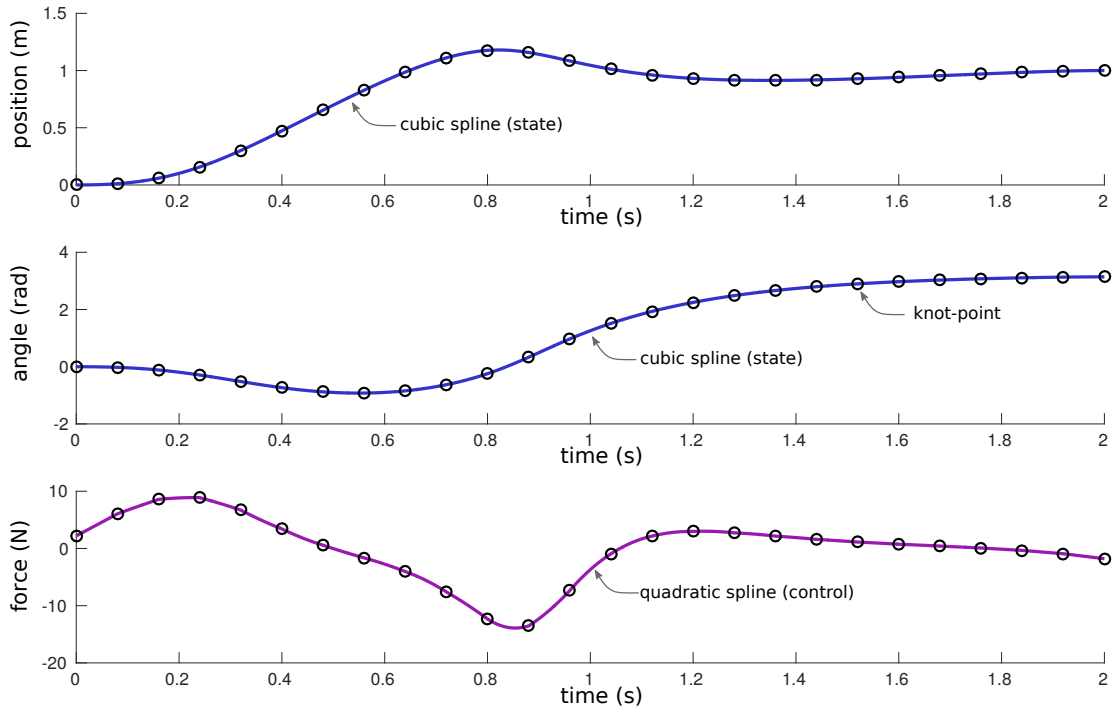

FIG. 10. *Plots showing the optimal trajectory for the cart-pole swing-up example.*

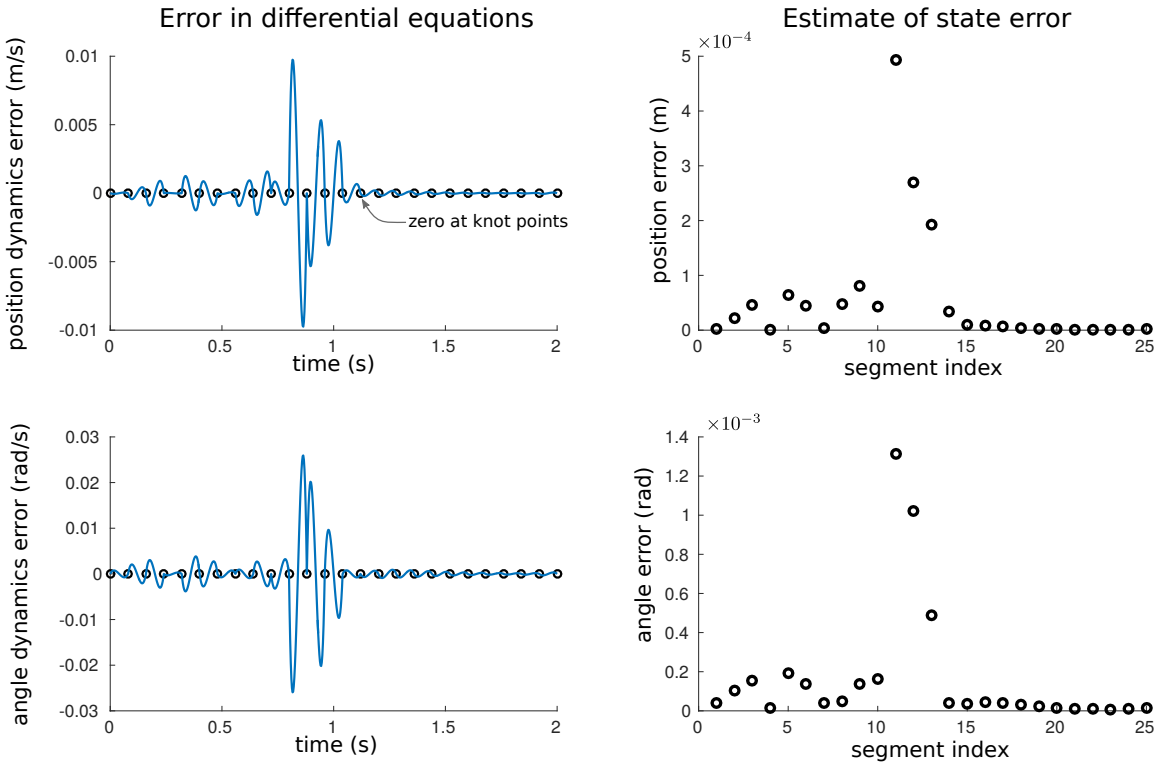

FIG. 11. *Plots showing the error in the system dynamics along the optimal trajectory for the cart-pole swing-up example. The plots on the left show the error in the differential equations, while the plots on the right show the integral of that error over each segment.*

7. Five-link biped example. In this section we will use trajectory optimization to find a periodic walking gait for a five-link (planar) biped walking model. This model is commonly used when studying bipedal walking robots [25, 43, 48, 54, 66, 67]. For this example, we will use the model developed by [66], with parameters that are selected to match the walking robot RABBIT [13] and given in Appendix §E.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.

7.1. Five-link biped: model. Figure 12 shows the five-link biped model as it takes a step. This model consists of a torso connected to two legs, each of which has an upper and lower link. The *stance* leg is supporting the weight of the robot, while the *swing* leg is free to move above 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 frictionless 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 F.

7.2. Five-link biped: system dynamics. During single stance, the five-link biped model has five degrees of freedom: the absolute angles of both lower legs (q_1 and q_5), both upper legs (q_2 and q_4), and the torso (q_3), as shown in Figure 12. 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 F.

7.3. Five-link biped: 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.

$$J = \int_0^T \left(\sum_{i=1}^{5} u_i^2(\tau) \right) \, d\tau \tag{7.1}$$

There are many other cost functions that we could have 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 [8, 59]. It turns out that CoT is a difficult cost function to optimize over, because the solutions tend to be discontinuous. The simple example in Section §8 shows a few ways to deal with such discontinuities.

7.4. Five-link biped: constraints. A variety of constraints are required to produce a sensible walking gait. The constraints presented here are similar those used in [66].

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, as defined in Appendix §F.

$$\boldsymbol{x}_0 = \boldsymbol{f}_H(\boldsymbol{x}_F) \tag{7.2}$$

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

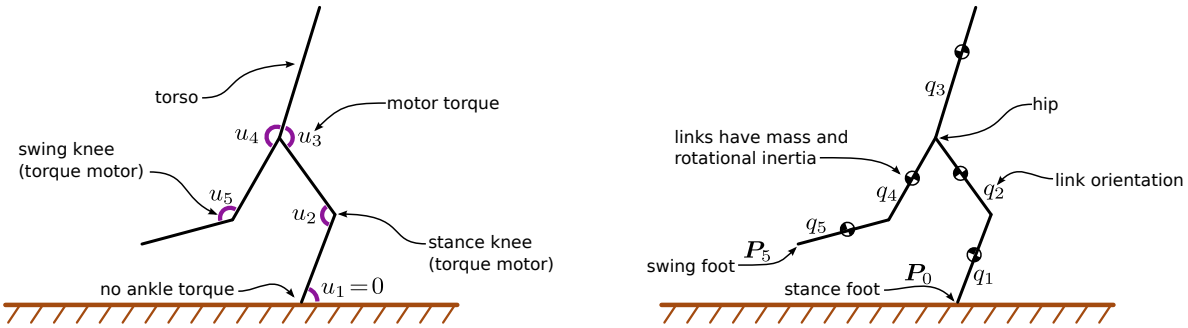

FIG. 12. *Illustration of the five-link biped model. We assume that the biped is a planar kinematic chain, with each joint is connected to its parent by an ideal revolute joint and torque source. The biped is under-actuated, because the stance ankle has no motor.*

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. Note that we use the [] notation to show a two element column vector, where the top element is the horizontal component and the bottom element is the vertical component.

$$\boldsymbol{P}_5(T) = \begin{bmatrix} D \\ 0 \end{bmatrix} \qquad \text{(Note: } \boldsymbol{P}_0(t) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ by definition)} \tag{7.3}$$

We have added an additional constraint on the biped robot to make the problem more interesting: that the stance ankle torque is identically zero throughout the trajectory. This constraint is essentially like saying 'the robot has small feet', and is widely used in the Hybrid Zero Dynamics technique for controlling walking robots [66].

When we derived the heel-strike collision equations (see Appendix §F), 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, $\hat{\boldsymbol{n}} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$, because the ground is flat and level.

$$0 < \dot{\boldsymbol{P}}_5(0) \cdot \hat{\boldsymbol{n}} \qquad 0 > \dot{\boldsymbol{P}}_5(T) \cdot \hat{\boldsymbol{n}} \tag{7.4}$$

Next we have a constraint to keep the swing foot above the ground at all times, shown below. Interestingly, the optimal solution for the minimum torque-squared walking gait keeps the foot above the ground (at least for our chosen set of parameters) so this constraint is unnecessary.

$$0 < \boldsymbol{P}_5(t) \cdot \hat{\boldsymbol{n}} \qquad \forall t \in (0, T) \tag{7.5}$$

In some cases, it might be desirable to achieve some ground clearance for the swing foot, or to work with some non-flat ground profile. 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. In both cases, it is critical that the constraint is consistent with the boundary conditions and that that the implementation is smooth, to avoid over-constraining the problem. Both methods are shown below, where $\hat{\boldsymbol{i}} = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$ and $\hat{\boldsymbol{j}} = \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]$.

$$y(t) < \boldsymbol{P}_5(t) \cdot \hat{\boldsymbol{n}} \qquad \forall t \in (0, T) \qquad \textbf{foot clearance (time-based)} \tag{7.6}$$

$$y\big(\boldsymbol{P}_5(t) \cdot \hat{\boldsymbol{i}}\big) < \boldsymbol{P}_5(t) \cdot \hat{\boldsymbol{j}} \qquad \forall t \in (0, T) \qquad \textbf{foot clearance (state-based)} \tag{7.7}$$

Finally, it is worth noting one mistake that is common in these optimizations: redundant constraints. Notice, for example, that for step length we only put a constraint on the final position of the foot (7.3). The initial position is fully constrained given (7.3) and the periodic step map constraint (7.2). 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.

7.5. Five-link biped: initialization. When we solve the trajectory optimization problem, we need to provide an initial guess for the trajectory. In this case, we created this guess by constructing an initial and final state, and then using linear interpolation to obtain intermediate states. We constructed the final state by selecting joint angles that formed a reasonable walking pose. We then computed the initial joint angles by applying the step map (see F.9) to the final state.

$$\boldsymbol{q}(0)_{\text{guess}} = \begin{bmatrix} -0.3 \\ 0.7 \\ 0.0 \\ -0.5 \\ -0.6 \end{bmatrix} \qquad \boldsymbol{q}(T)_{\text{guess}} = \begin{bmatrix} -0.6 \\ -0.5 \\ 0.0 \\ 0.7 \\ -0.3 \end{bmatrix} \tag{7.8}$$

$$\boldsymbol{q}_{\text{guess}}(t) = \boldsymbol{q}_{\text{guess}}(0) + \frac{t}{T}\big(\boldsymbol{q}_{\text{guess}}(T) - \boldsymbol{q}_{\text{guess}}(0)\big) \tag{7.9}$$

We initialized the joint rates by differentiating the joint angle guess.

$$\dot{\boldsymbol{q}}_{\text{guess}}(t) = \frac{d}{dt}\bigg(\boldsymbol{q}_{\text{guess}}(t)\bigg) = \frac{1}{T}\big(\boldsymbol{q}_{\text{guess}}(T) - \boldsymbol{q}_{\text{guess}}(0)\big) \tag{7.10}$$

Finally, we initialize the joint torques to be constant at zero.

$$\boldsymbol{u}_{\text{guess}}(t) = \boldsymbol{0} \tag{7.11}$$

Note that this initial guess does not satisfy the system dynamics (or most of the other constraints), but it does provide something that is close to the desired walking motion. This is the key feature of an initial guess - that it starts the optimization close enough to the desired behavior so that the optimization will find the 'correct' solution.

7.6. Five-link biped: results. We solved this example problem in Matlab, using FMINCON's [36] interior-point algorithm as the non-linear programming solver. The physical parameters that we used are given in Appendix E.2, and the optimization was computed on a regular desktop computer[2]. We chose to use analytic gradients (Appendix F) for the entire problem, although similar results are obtained for numerical gradients.

All source code for solving this trajectory optimization problem, including derivation of the equations of motions, is given in the electronic supplement (see Appendix §A).

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 compute 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 13 as a stop-action animation with uniformly spaced frames. The same trajectory is also shown in Figure 14, with each joint angle and torque given as a continuous function of time. Finally, Figure 15 shows the error estimates computed along the trajectory.

[2]processor: 3.4GHz quad-core Intel i5-3570K

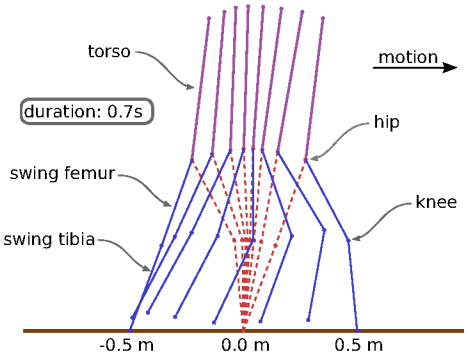

FIG. 13. *Illustration of the optimal trajectory for the five-link biped example. The poses are uniformly spaced in time and the biped is moving from left to right.*

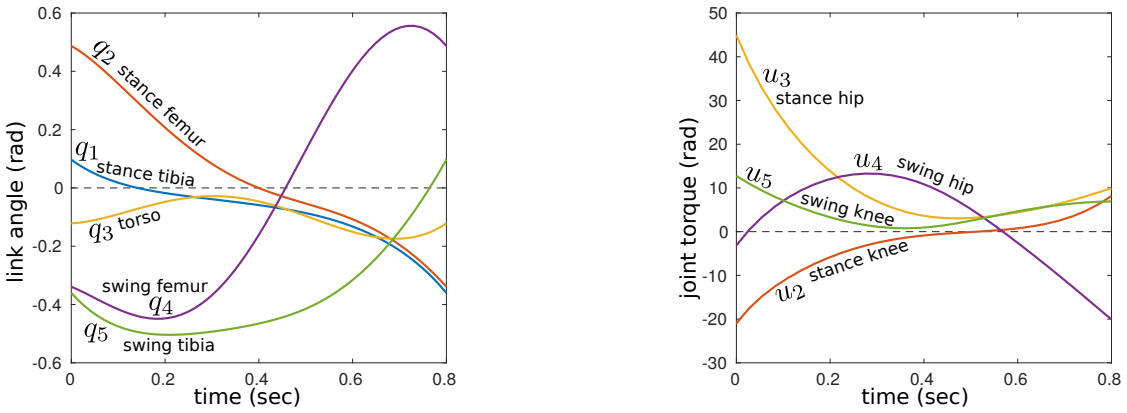

FIG. 14. *Plots showing the optimal trajectory for the five-link biped example. Notice that the curves are smooth, partially due to the integral of torque-squared cost function. The torque curve for the stance ankle $u_1 = 0$ is not shown, because it is zero by definition.*

8. Block move example (minimum-work). In this section, we will revisit the simple block-moving example from §2, 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 type of solution, which consists of alternating periods of maximum and zero control effort, is known as a *bang-bang* solution. Bang-bang solutions are difficult to handle with standard direct collocation because the discretization method (based on polynomial splines) cannot accurately approximate the discontinuity in the solution. In this section, we will study a few commonly used techniques for dealing with such discontinuities in the solution to a trajectory optimization problem.

8.1. Block move example: problem statement. Our goal here is to move a block one unit along a one-dimensional friction-less surface, in a one unit of time, along a trajectory that minimizes the integral of the absolute work done by the control force u. The objective function is given below, where the position and velocity of the block are given by x and ν respectively.

$$\min_{u(t),\, x_1(t),\, \nu(t)} \int_0^1 |u(\tau)\, \nu(\tau)|\, d\tau \tag{8.1}$$

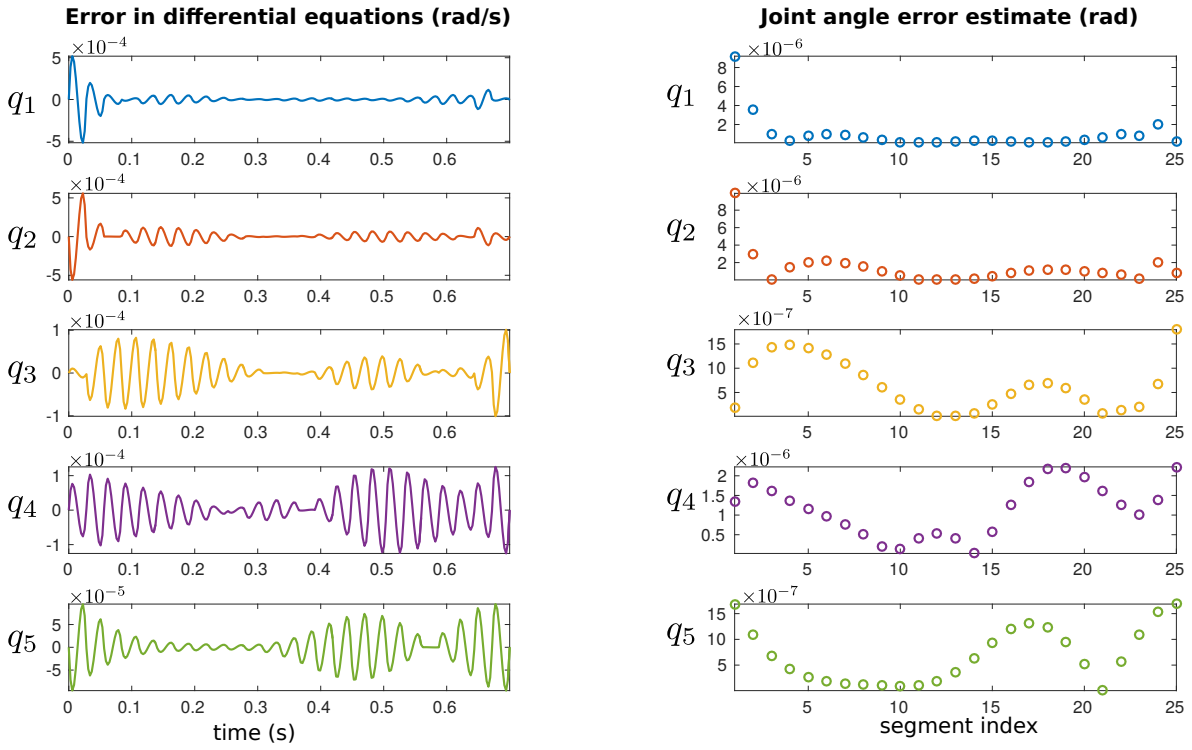

FIG. 15. *Plots showing the error in the system dynamics along the optimal trajectory for the five-link biped example. These error estimates are computed using the techniques described in §5.3.*

We will assume that the block has unit mass and slides without friction, so we can write it's dynamics as shown below.

$$\dot{x} = \nu \qquad \dot{\nu} = u \tag{8.2}$$

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.

$$\begin{aligned} x(0) &= 0 & x(1) &= 1 \\ \nu(0) &= 0 & \nu(1) &= 0 \end{aligned} \tag{8.3}$$

Finally, we will assume that the force moving the block is bounded:

$$-u_{\max} \leq u(t) \leq u_{\max} \tag{8.4}$$

8.2. Block move example: analytic solution. The analytic solution to this problem can be constructed using a slightly modified version of the method shown in Appendix B, but constraints on the control and the non-linear objective function in this problem makes the resulting formulation somewhat complicated. Instead, we will use simple intuition to make a guess at the form of the analytic solution. We find that the numerical results converge to this analytic solution, which suggests (but does not prove) that it is the correct solution.

We start by observing that in the case where $u_{\max} \to \infty$ there is a feasible solution with zero cost: the control is a delta function at the boundaries (positive at the beginning, negative at the end) and zero otherwise. We can then extend this solution to non-zero values of $u_{\max}$ by using a bang-bang control law: maximum force, then zero force, then minimum force. This leaves two unknowns in the control trajectory: the two switching times, which can be solved for using the boundary values for the problem. The resulting

controller is given below.

$$u^*(t) = \begin{cases} u_{\max} & t < t^* \\ 0 & \text{otherwise} \\ -u_{\max} & (1-t^*) < t \end{cases} \qquad \text{where} \qquad t^* = \frac{1}{2}\left(1 - \sqrt{1 - \frac{4}{u_{\max}}}\right) \tag{8.5}$$

The most important aspect of this solution to notice is that the control $u(t)$ is discontinuous. This means that that the linear and quadratic spline control approximations used by the trapezoidal and Hermite-Simpson collocation methods *cannot* perfectly represent this solution, although they can get arbitrarily close with enough mesh refinement. One way to obtain a more precise solution would be pose this problem as a multi-phase trajectory optimization problem [45]. These methods are briefly discussed in Section §9.9, and amount to solving the problem as a sequence of three coupled trajectories, allowing the discontinuity to occur precisely at the switching points between trajectories.

Another interesting point is that if $u_{\max} < 4$ then there is no feasible solution for the trajectory: the switching is imaginary. Finally, if there is no force limit $u_{\max} \to \infty$ then the solution is impulsive: not just discontinuous, but a delta function.

8.3. Block move example: discontinuities. There are two types of discontinuities present in this example problem. The first is obvious: the `abs()` in the objective function (8.1). The second discontinuity is found in the solution (8.5) 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 §8.7.

The discontinuity in the solution is a bit harder to detect and address. We can detect the discontinuity by observing that the optimization is slow to converge, and by visually inspecting the resulting trajectories. If you're stuck using single-phase direct collocation, like the methods presented in this paper, then the best way to handle the discontinuity is to smooth the problem (if possible) and then to use mesh refinement to make a dense collocation grid near the discontinuity. If you have access to a multi-phase solver (see §9.9) then you can break the trajectory into multiple segments, and force the discontinuity to occur between the segments.

8.4. Block move example: initialization. We will compute an initial guess for position by linear interpolation between the initial position $x(0) = 0$ and final position $x(1) = 1$. We then set the velocity guess to be the derivative of position, and the force (acceleration) to be the derivative of velocity. There are many other schemes that could be used, we choose this one because it is simple and effective. Once we have an initial trajectory, we can evaluate it at each collocation point to obtain values to pass to the non-linear programming solver.

$$x^{\text{init}}(t) = t \tag{8.6}$$

$$\nu^{\text{init}}(t) = \tfrac{d}{dt}x^{\text{init}}(t) = 1 \tag{8.7}$$

$$u^{\text{init}}(t) = \tfrac{d}{dt}\nu^{\text{init}}(t) = 0 \tag{8.8}$$

8.5. Block move example: slack variables. The most 'correct' way to rewrite the objective function (8.1) is using slack variables: this moves the discontinuity from the objective function to a constraint. The slack variable approach here is taken from [6]. The benefit of rewriting the trajectory optimization problem using slack variables to represent the absolute value function is that it is mathematically identical to the original optimization problem. That being said, there are a few downsides to this method. The first is that the solution will still be discontinuous, and direct collocation cannot precisely represent it (although it can get arbitrarily close). Second, 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.

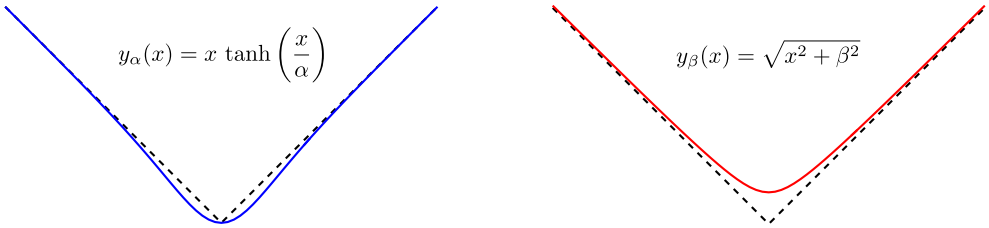

FIG. 16. *Comparison of two smooth approximations for the absolute value function: hyperbolic tangent smoothing (left) and Square-root smoothing (right).*

The key idea behind the slack variable approach is that you can push the discontinuity from the objective function to a constraint, where the non-linear programming solver can properly handle it. We start by introducing two slack variables (s_1 and s_2), and rewriting the objective function. Note that the slack variables here are to be treated as decision variables for the purposes of transcription.

$$\min_{u(t),\, x(t),\, v(t)} \int_0^1 \left|u(t)\, v(t)\right| d\tau \qquad \rightarrow \qquad \min_{\substack{u(t),\, x(t),\, v(t) \\ s_1(t),\, s_2(t)}} \int_0^1 \big(s_1(\tau) + s_2(\tau)\big)\, d\tau \tag{8.9}$$

Next, we introduce a few constraints. The first require that the slack variables be positive:

$$0 \leq s_1(t) \qquad 0 \leq s_2(t) \tag{8.10}$$

Finally, we require that the difference between the slack variables is equal to the term inside of the `abs()` function (8.1).

$$s_1(t) - s_2(t) = u(t)\, v(t) \tag{8.11}$$

This set of constraints (8.10) and (8.11) means 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 that $s_1(t) = s_2(t) = 0$ 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 caused by the spline approximation in the transcription method, as shown in Section §8.7.

8.6. Block move example: 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 smooth approximations for `abs()`, both of which are given below and plotted in Figure 16.

$$y_\alpha(x) = x \tanh\left(\frac{x}{\alpha}\right) \quad \approx |x| \tag{8.12}$$

$$y_\beta(x) = \sqrt{x^2 + \beta^2} \quad \approx |x| \tag{8.13}$$

The smooth approximation to `abs()` using the hyperbolic tangent function (8.12), also known as exponential smoothing, is always less than $|x|$, while the approximation using the square-root function (8.13) is always greater than $|x|$. 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 with a more accurate solution.

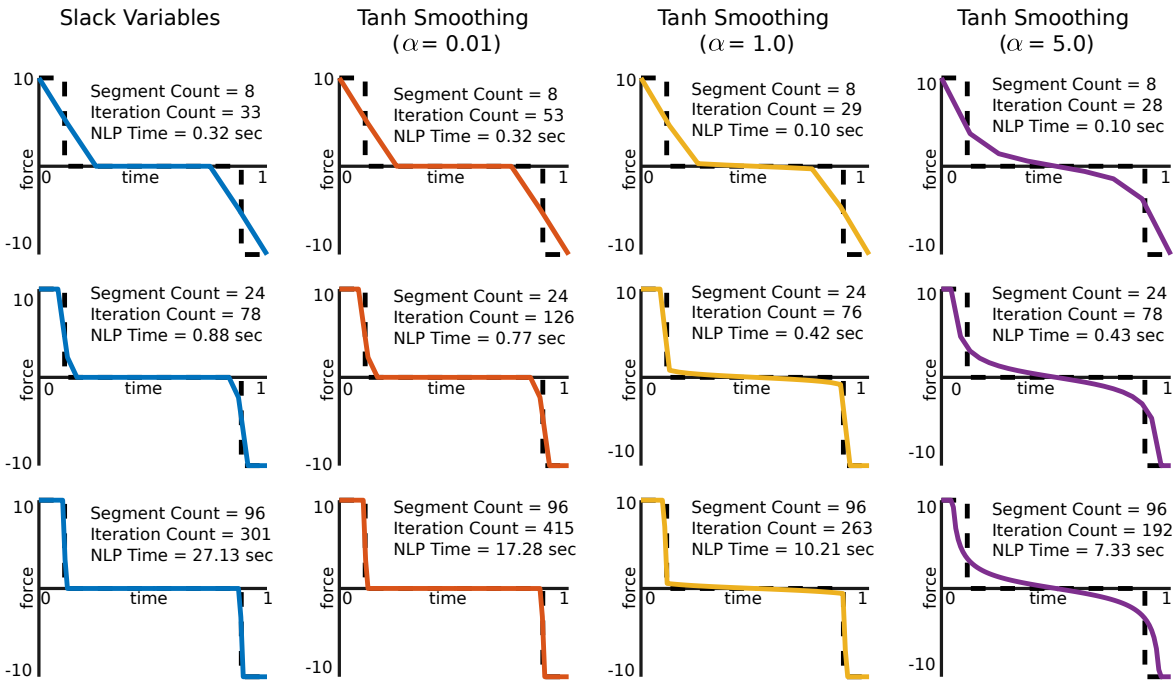

FIG. 17. *Plots showing the solution to the minimal-work block-moving example, computed using various methods and parameters. In each case, the analytic solution is given by a dashed black line, and the solid colored line gives the numerical solution using direct collocation. The left column shows the 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 accuracy is degraded. Note that the smoothed version of the problem results in a more smooth solution.*

One important thing to note is that smoothing fundamentally changes the optimization 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 [55] and [9].

8.7. Block move example: results. We solved this more complicated version of the block moving problem using the trapezoidal collocation method, and we used FMINCON's [36] interior-point solver to solve the non-linear program. Although this optimization problem appears simple, it is actually difficult to numerically solve without careful mesh refinement (or re-posing the problem using multi-phase trajectory optimization, see §9.9). 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 17 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 §A).

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 this block-pushing problem looks like it should be easier. The difficulty, as best we can tell, comes from the discontinuity in the solution.

The solution obtained using slack variables (left column) converges to the analytic 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 smaller number of decision variables). As the smoothing parameter is increased ($\alpha = 1.0$ and $\alpha = 5.0$), the solution is obtained faster, at a loss of accuracy.

FIG. 18. *Comparison of an open-loop solution (optimal trajectory) with a closed-loop solution (optimal policy). 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**.

9. Background. The topics in this section are selected to provide the reader with a broad understanding of some of the concepts that are related to direct collocation. We start with a few topics about optimization in general and then move on to other methods for solving trajectory optimization problems. We conclude with a method comparison and a list of optimization software.

9.1. Trajectory optimization vs parameter optimization. Trajectory optimization is concerned with minimizing a *functional* $J\big(\boldsymbol{f}(t)\big)$, where $\boldsymbol{f}(t)$ is an arbitrary vector function. In contrast, parameter optimization is concerned with minimizing some function $J(\boldsymbol{x})$, where $\boldsymbol{x}$ is a vector of real numbers. This makes trajectory optimization more challenging than parameter optimization, because the space of functions is much larger than the space of real numbers.

9.2. 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 an optimal *policy*. Unlike an optimal trajectory, an optimal policy provides the optimal control for *every* point in the state space. Another name for the optimal policy is the *closed-loop* solution to the optimal control problem. An optimal trajectory starting from any point in the state space can be recovered from a closed-loop solution by a simple simulation. Figure 18 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 (computing an optimal policy) 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 [42]. 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 — the so-called 'curse of dimensionality' [41].

9.3. Continuous-time and discrete-time systems. Trajectory optimization is generally concerned with finding optimal trajectories for a *dynamical system*. The dynamics describe how the state of a system changes in response to some input or decision, typically referred to as a control.

There are many different types of dynamical systems. In this tutorial we have focused 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.

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}) \qquad \textbf{continuous-time system} \tag{9.1}$$

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 [35]. Trajectory optimization for these systems is generally easier than on fully continuous systems. Discrete-time systems are often constructed to approximate continuous time systems.

$$\boldsymbol{x}_{k+1} = \boldsymbol{f}_k(\boldsymbol{x}_k, \boldsymbol{u}_k) \qquad \textbf{discrete-time system} \tag{9.2}$$

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 (optimal control from every point in the state space) require the dynamical system to be in this discrete form. A common example would be a traffic network, where there is a discrete set of states (cities), and a discrete set of controls (roads out of each city). 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.

9.4. Indirect methods. Both the trapezoidal and Hermite–Simpson collocation methods presented in this tutorial are *direct* methods, which discretize the trajectory optimization problem, converting it into a non-linear program. There are another set of methods for solving trajectory optimization problems, known as an *indirect* methods. Indirect methods analytically construct the necessary and sufficient conditions for optimality. Then they discretize these conditions and solve them numerically. A common way to categorize these two methods is that a direct method discretizes and then optimizes, while an indirect method optimizes and then discretizes.

Let's consider a simple scalar optimization problem to illustrate how an indirect method works: minimizing $y = f(t)$. Basic calculus tells us that the minimum value $y^* = f(t^*)$ will occur when the derivative is zero $y'(t^*) = 0$. Additionally, we need to check that the curvature is positive: $y''(t^*) > 0$, ensuring that we have a local minimum, rather than a local maximum (or saddle point). If both of those conditions hold, then we know that $y^* = f(t^*)$ is indeed a local minimum. An indirect optimization works along the same principle, but the conditions are a bit more difficult to construct and solve. 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: $y(t_0) > y(t_1) > \ldots > y(t^*)$ [6].

The major benefit of an indirect method, when compared to a direct method, is that an indirect method will generally be more accurate and have a more reliable error estimate. Both of these benefits come from the analytic expressions for the necessary and sufficient conditions that the user derives while constructing the indirect problem.

There are several difficulties associated with indirect methods when compared to direct methods. For example, 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 [5]. 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 [6]. 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 [5].

9.5. Direct single shooting. Like direct collocation, the *direct single 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 an open-loop parameterization of the control along the trajectory, as well as 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. [5]

9.6. Direct multiple shooting. A common extension of the direct single shooting method is *direct 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 [5].

When compared to collocation methods, shooting methods tend to create small dense non-linear programs, which have fewer decision variables that are more coupled. 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 [5]. 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 poor convergence in some cases [5, 6].

9.7. Orthogonal collocation. Orthogonal collocation is similar to direct collocation, but it generally uses higher-order polynomials. The collocation points for these methods are located at the roots an orthogonal polynomial, typically either Chebyshev or Legendre [15]. Increasing the accuracy of a solution is typically achieved by increasing either the number of 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 [51], *if* the underlying function is sufficiently smooth [58]. In cases where the entire trajectory is approximated using a single high-order polynomial, the resulting method is called *pseudospectral* collocation or *global collocation* [51].

One of the key implementation details about orthogonal collocation is that the trajectory is represented using *Barycentric Interpolation* [4], 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 §D for further details about how to work with orthogonal polynomials.

9.8. Differential dynamic programming. One final method 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 uses 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 [30, 38], and a good overview was provided by [40].

9.9. Multi-phase methods. There are many trajectory optimization problems that have a sequence of continuous motion phases separated by discrete jumps. One common example is the trajectory of a multi-stage rocket, which has continuous motion punctuated by discrete changes when each stage separates. Another example is the gait of a walking robot, which has a discontinuity as each foot strikes the ground. Solving a multi-phase problem is sort of like solving multiple single-phase problems in parallel. The key difference is that the boundary constraints between any two phases can be connected, thus coupling the trajectory segments. Multi-phase methods are covered in detail in [45, 63].

9.10. Through-contact methods. Through-contact methods are specialized for computing optimal trajectories for hybrid dynamical systems that describe contact mechanics: imagine the gait of a walking robot, or two objects colliding and then falling to the ground. Most physics simulators use a complementarity constraint to model contact between two rigid objects: a contact force is allowed if and only if the two objects are in contact. The key idea in through-contact optimization is to treat the contact forces as decision variables in the optimization, and then apply a complementarity constraint at each grid point: the contact force must be zero unless the objects are in contact. These methods are covered in detail in [47], [46], and [39].

9.11. Which method is best?. In short, there is no best method for trajectory optimization. There are many trade-offs between the different methods, and a good understanding of these trade-offs will help determine which method is best for a specific application. A good high-level comparison of methods can also be found in [5] and [51]. Here I will provide a brief overview of some of these trade-offs.

In general, indirect methods tend to produce more accurate solutions than direct methods, at the cost of being more difficult to construct and solve. This is because indirect methods explicitly compute the necessary and sufficient conditions for optimality of the original problem, while a direct method precisely solves a discrete approximation of the original problem. One common approach to obtain accurate solutions is to first compute an approximation of the solution using a direct method, and then use this to initialize an

TABLE 1
Trajectory Optimization Software

Name	License	Interface	Method
GPOPS-II [45]	commercial	Matlab	direct orthogonal collocation
PSOPT [2]	open source	C++	direct collocation
SOS [7]	commercial	GUI	direct collocation (methods from [6])
DIRCOL [63]	free license	C	direct collocation
DIDO [52]	commercial	Matlab	indirect orthogonal (pseudospectral) collocation

TABLE 2
Non-Linear Programming Solvers

Name	License	Interface
FMINCON [36]	Matlab (commercial)	Matlab
SNOPT [50]	commercial	C++
IPOPT [64]	open source	C++

indirect method. As a side note: both shooting and collocation (transcription) methods can be applied to either a direct or indirect formulation of a trajectory optimization problem [6].

Shooting methods are best for applications where the dynamics must be computed accurately, but the control trajectory is simple. For example, computing the trajectory of a spacecraft, where you occasionally fire the thrusters to change course, but are otherwise are following a ballistic trajectory. Multiple shooting methods are generally preferred over single shooting, except in cases where the control is very simple or the initial guess is very good.

Collocation (transcription) methods are best for applications where the dynamics and control must be computed to a similar accuracy, and the structure of the control trajectory is not known *a priori*. For example, computing the torque to send to the joints of a robot as it performs some motion.

Both shooting and collocation methods can be either low- or high-order. High-order collocation methods are given a special name: orthogonal collocation. Trapezoidal collocation would be considered a low-order method, while Hermite–Simpson collocation would usually be considered a medium-order method. The trade-off between using a method with more low-order segments or few high-order segments is complicated [16]. The general approach is to use a relatively lower-order method to obtain an initial solution to the trajectory, and then perform an error analysis [6, 16]. The result will indicate whether it is better to re-mesh the trajectory using additional lower-order segments, or replacing lower-order segments with higher-order segments.

In situations where you need to compute trajectory for a hybrid system, there are two choices: multi-phase optimization (§9.9) and through-contact optimization (§9.10). Multi-phase optimization is preferable for most situations: the optimizations are easier to compute and tend to be more accurate. Through-contact optimization is preferable when the the discontinuities are due to contact mechanics and the sequence of continuous motion phases is unknown.

9.12. Trajectory optimization software. There are a variety of software programs that solve trajectory optimization problems, some of which are given in Table 1. Each of these solvers performs some transcription method and then hands the problem off to a non-linear programming solver. Table 2 shows a few popular software packages for solving non-linear programming problems. The electronic supplement, described in Appendix §A, 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.

10. Summary. The goal of this tutorial is to give the reader an understanding of the concepts required to implement their own direct collocation methods. We focus primarily on trapezoidal and Hermite–Simpson collocation, and we briefly touch on a variety of other methods. We include practical suggestions, debugging techniques, and a complete set of equations and derivations. Throughout the tutorial we convey concepts

through a sequence of four example problems, and the electronic supplement shows how to solve each example using Matlab.

Appendix A. 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 part is a general purpose trajectory optimization library, written in Matlab, that solves the trajectory optimization problems 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 and 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.

A.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/OptimTraj`

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 and easily 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 and both direct collocation methods provide the user an error estimate along the solution trajectory.

A.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 trajectory optimization code from Appendix §A.1. 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 figures from 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`.

Appendix B. Analytic solution to block-move example. In this section we show how to find the analytic solution to the block-moving problem from Section 2. The method presented here is based on calculus of variations, and is described in detail in the textbook by Bryson and Ho [11]. Here we show two slightly different solution methods. The first solution, in Section B.1, treats the problem as a true optimal control problem, where the state and control are separate and the dynamics are handled with multiplier functions. The second solution, in Section B.2, simplifies the problem by first substituting the dynamics into the cost function.

B.1. Full solution. We would like to minimize the cost functional $J()$, given below, where u is the control force applied to the block.

$$J(t, \boldsymbol{z}, u) = \int_0^1 u^2(\tau)\, d\tau \tag{B.1}$$

The system dynamics $\boldsymbol{f}()$ are given below, where x is position, ν is velocity, and $\boldsymbol{z} = [x, v]^T$ is the state vector.

$$\dot{\boldsymbol{z}} = \begin{bmatrix} \dot{x} \\ \dot{\nu} \end{bmatrix} = \boldsymbol{f}(\boldsymbol{z}, u) = \begin{bmatrix} v \\ u \end{bmatrix} \tag{B.2}$$

We will also apply the following boundary conditions, where subscripts are used to denote evaluation at the boundary points on the trajectory.

$$\boldsymbol{z}_0 = \boldsymbol{z}(t)|_{t=0} = \begin{bmatrix} x_0 \\ \nu_0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \boldsymbol{z}_1 = \boldsymbol{z}(t)|_{t=1} = \begin{bmatrix} x_1 \\ \nu_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{B.3}$$

We need to satisfy the dynamics to ensure a feasible solution. This is done by modifying the cost functional to include the system dynamics and a vector of multiplier functions $\boldsymbol{\lambda} = [\lambda_x, \lambda_\nu]^T$. Notice that when the dynamics are satisfied $\boldsymbol{f} - \dot{\boldsymbol{z}} = 0$ and thus $\bar{J} = J$ regardless of what the multiplier functions are.

$$\bar{J} = \int_0^1 \left(u^2(\tau) + \boldsymbol{\lambda}^T(\boldsymbol{f} - \dot{\boldsymbol{z}})\right)\, d\tau \tag{B.4}$$

Now we can use integration by parts to re-write the modified cost function [11]. Here again we use the subscript notation to indicate evaluation at the boundary condition (*e.g* $\boldsymbol{\lambda}_0 = \boldsymbol{\lambda}(t)|_{t=0}$).

$$\bar{J} = \boldsymbol{\lambda}_0^T \boldsymbol{z}_0 - \boldsymbol{\lambda}_1^T \boldsymbol{z}_1 + \int_0^1 \left(u^2(\tau) + \boldsymbol{\lambda}^T \boldsymbol{f}\right) + \left(\dot{\boldsymbol{\lambda}}^T \boldsymbol{z}\right)\, d\tau \tag{B.5}$$

At this point, it is useful to define two quantities that will be useful throughout the rest of the derivation. The first is the *Lagrangian* $\mathcal{L}$, which is the term inside the integral of the original cost function J. The second term is the *Hamiltonian* $\mathcal{H}$, which is the sum of the Lagrangian and product of the multiplier functions with the system dynamics [11].

$$\mathcal{L} = u^2 \tag{B.6}$$

$$\mathcal{H} = \mathcal{L} + \boldsymbol{\lambda}^T \boldsymbol{f} \quad = \quad u^2 + \lambda_x \nu + \lambda_\nu u \tag{B.7}$$

Consider a simple optimization problem: finding the minimum of a scalar function. The minimum will occur when the first derivative is zero and the second derivative is positive. A similar principle can be used for trajectories, although we use the term *variation* instead of derivative. An optimal trajectory must have a first variation equal to zero, and a second variation that is non-negative. Here we will focus on the necessary condition: that the first variation is zero.

Let's suppose that the optimal trajectory is given by $\boldsymbol{z}^*$ and u^*. A trajectory that is sub-optimal can now be written as a sum of the optimal trajectory and a small perturbation from that trajectory, as shown below where ε is a small parameter and $\delta\boldsymbol{z}$ and δu are small (arbitrary) variations in the state and control.

$$\delta\boldsymbol{z} = \boldsymbol{z}^* + \varepsilon\, \delta\boldsymbol{z} \qquad\qquad u = u^* + \varepsilon\, \delta u \tag{B.8}$$

The first variation of the cost function is its partial derivative with respect to this small parameter ε.

$$\delta\bar{J} \equiv \frac{\partial}{\partial\varepsilon}\bar{J}\bigg|_{\varepsilon=0} \tag{B.9}$$

Using the chain rule, we can now write out an expression for the first variation of the cost function [11].

$$\delta\bar{J} = \boldsymbol{\lambda}_0^T \left.\frac{\partial \boldsymbol{z}_0}{\partial\varepsilon}\right|_{\varepsilon=0} - \boldsymbol{\lambda}_1^T \left.\frac{\partial \boldsymbol{z}_1}{\partial\varepsilon}\right|_{\varepsilon=0} + \int_0^1 \left[\left.\frac{\partial\mathcal{H}}{\partial\varepsilon}\right|_{\varepsilon=0} + \dot{\boldsymbol{\lambda}}^T \left.\frac{\partial \boldsymbol{z}}{\partial\varepsilon}\right|_{\varepsilon=0}\right] d\tau \tag{B.10}$$

$$\delta\bar{J} = \boldsymbol{\lambda}_0^T \delta\boldsymbol{z}_0 - \boldsymbol{\lambda}_1^T \delta\boldsymbol{z}_1 + \int_0^1 \left[\left(\frac{\partial\mathcal{H}}{\partial\boldsymbol{z}} + \dot{\boldsymbol{\lambda}}^T\right)\delta z + \frac{\partial\mathcal{H}}{\partial\boldsymbol{u}}\delta\boldsymbol{u}\right] d\tau \tag{B.11}$$

The first variation of the cost function $\delta\bar{J}$ (B.11) must be zero along the optimal trajectory. The variations in state at the initial and final points on the trajectory are zero, since the boundary conditions are fixed ($\delta\boldsymbol{z}_0 = 0$, $\delta\boldsymbol{z}_1 = 0$). Thus the first two terms in (B.11) are both zero. The variations in state $\delta\boldsymbol{z}$ and in control $\delta\boldsymbol{u}$ along the trajectory are arbitrary, thus each of their coefficients must be zero in order for the integral term to be zero.

$$\frac{\delta\mathcal{H}}{\delta\boldsymbol{z}} + \dot{\boldsymbol{\lambda}}^T = 0 \tag{B.12}$$

$$\frac{\delta\mathcal{H}}{\delta\boldsymbol{u}} = 0 \tag{B.13}$$

These two equations (B.12) and (B.13) form the necessary conditions for optimality: a solution that satisfies them will be at a stationary point. To be rigorous, we would also need to show that the second variation is

non-negative, which implies that that solution is at a minimum (as opposed to a maximum or saddle point). This calculation is beyond the scope of this paper, but is covered in [11].

The next step is to solve for the multiplier functions, which we do by rearranging (B.12) to give us of differential equations as shown below.

$$-\dot{\boldsymbol{\lambda}}^T = \frac{\delta \mathcal{H}}{\delta \boldsymbol{z}} \tag{B.14}$$

$$\dot{\boldsymbol{\lambda}} = -\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}}\right)^T - \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{z}}\right)^T \boldsymbol{\lambda} \tag{B.15}$$

We can now evaluate (B.15) for our specific problem.

$$\begin{bmatrix} \dot{\lambda}_x \\ \dot{\lambda}_\nu \end{bmatrix} = -\begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \lambda_x \\ \lambda_\nu \end{bmatrix} \tag{B.16}$$

This system of equations (B.16) is linear, and thus a solution is easily obtained, where c_0 and c_1 are constants of integration and time is given by t.

$$\lambda_x = c_0 \tag{B.17}$$

$$\lambda_\nu = c_1 - c_0 t \tag{B.18}$$

Now that we know the multiplier functions, we can go back and solve for the control functions using (B.13).

$$0 = \frac{\partial \mathcal{H}}{\partial \boldsymbol{u}} \tag{B.19}$$

$$0 = \frac{\partial}{\partial \boldsymbol{u}}\left(u^2 + \lambda_x \nu + \lambda_\nu u\right) \tag{B.20}$$

$$0 = 2u + 0 + (c_1 - c_0 t) \tag{B.21}$$

$$u = \tfrac{1}{2}(c_0 t - c_1) \tag{B.22}$$

We can use the system dynamics to obtain expressions for the position and velocity as functions of time, as shown below.

$$\nu = \int u(\tau)\, d\tau = \tfrac{1}{4} c_0 t^2 - \tfrac{1}{2} c_1 t + c_2 \tag{B.23}$$

$$x = \int \nu(\tau)\, d\tau = \tfrac{1}{12} c_0 t^3 - \tfrac{1}{4} c_1 t^2 + c_2 t + c_3 \tag{B.24}$$

Next, we need to solve for the unknown constants of integration c_i. We can do this by constructing a linear system from the boundary conditions.

$$\begin{bmatrix} x(0) \\ v(0) \\ x(1) \\ v(1) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ \frac{1}{12} & \frac{-1}{4} & 1 & 1 \\ \frac{1}{4} & \frac{-1}{2} & 1 & 0 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \tag{B.25}$$

Solving the linear system and substituting in the coefficients yields the solution below, which is valid for the domain of the problem $t \in [0, 1]$.

$$x(t) = -2t^3 + 3t^2 \tag{B.26}$$

$$\nu(t) = -6t^2 + 6t \tag{B.27}$$

$$u(t) = -12t + 6 \tag{B.28}$$

B.2. Short solution. For this problem, a shorter solution can be obtained since the control u is simply the second derivative of the position x. As a result, our cost function can be written:

$$J = \int_0^1 u^2(\tau)\, d\tau = \int_0^1 \ddot{x}^2(\tau)\, d\tau \tag{B.29}$$

In this case, we get the Lagrangian:

$$\mathcal{L}(t, x, \dot{x}, \ddot{x}) = \mathcal{L}(\ddot{x}) = \ddot{x}^2 \tag{B.30}$$

For a fully rigorous solution, one would need to show that the first variation of the objective function is zero, and the second variation is non-negative. Here we will focus on the first variation, which is the necessary condition for the optimal solution x^*. The following equation is constructed using integration by parts:

$$\frac{\partial \mathcal{L}}{\partial x^*} - \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{x}^*} - \frac{d^2}{dt^2}\frac{\partial \mathcal{L}}{\partial \ddot{x}^*} = 0 \tag{B.31}$$

The first two terms are zero, since $\mathcal{L}$ depends only on $\ddot{x}$. The final term can be evaluated and simplified to arrive at the following ordinary differential equation.

$$(0) - (0) - \frac{d^2}{dt^2}(2\ddot{x}^*) = 0 \tag{B.32}$$

$$\frac{d^4}{dt^4}x^* = 0 \tag{B.33}$$

The solution to this equation is a cubic polynomial with four unknown coefficients, identical to that found in (B.24). We solve these coefficients using the boundary conditions (B.3) to arrive at the solution:

$$x(t) = -2t^3 + 3t^2 \tag{B.34}$$

Appendix C. Derivation of simpson quadrature. Simpson quadrature is used to compute an approximation to the definite integral of a function by evaluating it at the boundaries and mid-point of the domain. It is precise when this function (the integrand) is quadratic, and we will use this fact to derive the rule. Let's start with a quadratic curve $\nu(t)$, given below.

$$\nu(t) = A + Bt + Ct^2 \tag{C.1}$$

Now suppose that we wish to compute a quantity x by integrating the function $\nu(t)$.

$$x = \int_0^h \nu(t)\, dt \tag{C.2}$$

$$x = \int_0^h A + Bt + Ct^2\, dt \tag{C.3}$$

$$x = At + \tfrac{1}{2}Bt^2 + \tfrac{1}{3}Ct^3 \,\Big|_0^h \tag{C.4}$$

$$x = Ah + \tfrac{1}{2}Bh^2 + \tfrac{1}{3}Ch^3 \tag{C.5}$$

We can use the value of ν at three points to uniquely determine the value of the coefficients A, B, and C. We will choose these points to be at the boundaries and mid-point of the interval:

$$\nu(0) = \nu_L \qquad \nu(\tfrac{h}{2}) = \nu_M \qquad \nu(h) = \nu_U \tag{C.6}$$

Doing a bit of algebra will show that the coefficients are given by:

$$A = \nu_L \tag{C.7}$$

$$Bh = -3\nu_L + 4\nu_M - \nu_U \tag{C.8}$$

$$Ch^2 = 2\nu_L - 4\nu_M + 2\nu_U \tag{C.9}$$

Finally, we can plug these coefficients into (C.5) and then simplify to arrive at Simpson's rule for quadrature:

$$x = \tfrac{h}{6}\big(\nu_L + 4\nu_M + \nu_U\big) \tag{C.10}$$

Appendix D. Orthogonal polynomials. All direct collocation methods are based on using polynomial splines to approximate continuous functions. The trapezoidal and Hermite–Simpson methods that we covered in this paper both use relatively low-order polynomial splines. Orthogonal collocation methods are similar, but use high-order splines instead. Working with these high-order polynomials requires some special attention to ensure that your implementations are numerically stable.

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. A rough approximation of the function can be made by including a small number of terms in the sum, while a more accurate approximation can be made by including more terms. 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. The number of terms in the infinite series is related to the *order* of the polynomial: a higher-order polynomial approximation will be more accurate. There are many papers that cover the detailed mathematics of orthogonal polynomials [4, 24, 27, 32, 44, 58] and their use in trajectory optimization [3, 15, 18, 19, 21–23, 28, 29, 53, 57, 62]. Here we will focus on the practical implementation details and on gaining a qualitative understanding of how orthogonal collocation works.

For the rest of this section, let's assume 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, quadrature weights w_i, and a differentiation matrix D. If the points t_i are 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 t_i do not given nice results. For example, choosing t_i to be uniformly spaced over the interval will result in numerically unstable schemes [4].

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 constraint, and barycentric interpolation is used to evaluate the solution trajectory.

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:

$$t = 2\frac{\tau - \tau_A}{\tau_B - \tau_A} - 1 \tag{D.1}$$

D.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 [17] 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 [58] and shown below.

$$t_i = \cos\left(\frac{i\pi}{n}\right), \qquad 0 \le i \le n \tag{D.2}$$

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

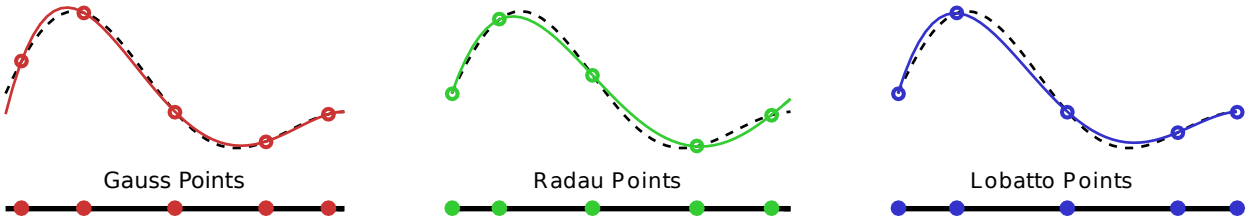

FIG. 19. *Illustration showing the three sets of points that are associated with each orthogonal polynomial. In this figure we have shown the Gauss, Radau, and Lobatto points for the 4^{th}-order Legendre orthogonal polynomials. The dashed line in each figure is the same, and the solid lines show the barycentric interpolant that is defined by that set of collocation points. Notice that the interpolant behaves differently for each set of points.*

methods for computing these points are given by [24, 27], although various sub-routines can be found with a quick internet search. ChebFun [17] 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 [23].

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 [23]. Here we have outlined these points for the Legendre polynomials, but the naming convention (Gauss, Radau, and Lobatto) applies to any orthogonal polynomial. Figure 19 shows an illustration of the Gauss, Radau, and Lobatto 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 [6]. A high-order Lobatto method based on Chebyshev orthogonal polynomials is described in [19].

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 [21, 26].

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 [45] uses a high-order Radau method, based on Legendre orthogonal polynomials.

These three types of methods are discussed in more detail in [22, 23], and are illustrated in Figure 19. Garg *et al.* [23] 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.

D.2. Barycentric lagrange interpolation. The best way to store and evaluate high-order orthogonal polynomials is using barycentric Lagrange interpolation. This works by expressing the value of the function at any point $f(t)$ using a weighted combination of the function's value ($f_i = f(t_i)$) at the roots of the orthogonal polynomial (t_i). The equation for barycentric interpolation is given below, with further details in [4]. 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.

$$f(t) = \frac{\sum_{i=0}^{n} \frac{v_i}{t - t_i} f_i}{\sum_{i=0}^{n} \frac{v_i}{t - t_i}} \tag{D.3}$$

Thus far, we know all parameters in (D.3), except for the interpolation weights v_i. These weights are calculated below, using the equation given by [4].

$$v_i = \frac{1}{\prod_{j \neq i}(t_i - t_j)}, \qquad i = 0, \ldots, n \tag{D.4}$$

Interestingly, the barycentric interpolation formula (D.3) will still interpolate the data at points f_i if the weights v_i are chosen arbitrarily. The choice of weights given by (D.4) is special in that it defines the unique *polynomial* interpolant, where other any other choice of weights will result in interpolation by some rational function [4]. Notice that these weights can be scaled by an arbitrary constant, and still produce the correct interpolation in (D.3), as well as the correct differentiation matrix (D.6). For example, ChebFun [17] 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.

D.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 (D.5).

$$\dot{\boldsymbol{f}} = \mathcal{D}\boldsymbol{f} \tag{D.5}$$

Each element of the differentiation matrix $\mathcal{D}$ can be computed as shown below, using a formula from [4].

$$\mathcal{D}_{ij} = \begin{cases} \dfrac{v_j/v_i}{t_i - t_j} & i \neq j \\ -\displaystyle\sum_{i \neq j} \mathcal{D}_{ij} & i = j \end{cases} \tag{D.6}$$

We can use the same interpolation weights v_i for interpolation of this derivative — we just replace the f_i terms in (D.3) with $\dot{f}_i$ to get the equation below.

$$\dot{f}(t) = \frac{\displaystyle\sum_{i=0}^{n} \frac{v_i}{t - t_i} \dot{f}_i}{\displaystyle\sum_{i=0}^{n} \frac{v_i}{t - t_i}} \tag{D.7}$$

D.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).

$$\int_{-1}^{1} f(\tau)\, d\tau \approx \sum_{i=0}^{n} w_i \cdot f_i \tag{D.8}$$

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 and the Legendre-Gauss-Lobatto weights can be computed as shown below.

$$w_i = W \frac{v_i^2}{(1 - t_i^2)} \qquad \textbf{Legendre-Gauss} \tag{D.9}$$

$$w_i = W v_i^2 \qquad \textbf{Legendre-Gauss-Lobatto} \tag{D.10}$$

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 $f_i = 1$, as shown below.

$$\int_{-1}^{1} d\tau = 2 = \sum_{i=0}^{n} w_i \cdot (1) \tag{D.11}$$

More details on the calculation of quadrature rules can be found in [20, 31, 58, 65].

Appendix E. Parameters for example problems. In this section we provide tables for the parameter values that we used when generating the results for the both the cart-pole swing-up example problem and the five-link biped example problem.

E.1. Cart-pole swing-up parameters. For the cart-pole swing-up example problem we chose parameters for our model to match something like you might see in a cart-pole in a controls lab demonstration. These parameters are given in Table 3.

E.2. Five-link biped parameters. For the five-linke biped walking gait example we chose parameters for our model to match the walking robot RABBIT [13, 66] which are reproduced here in Table 4. We also selected a trajectory duration of $T = 0.7s$ and a step length of $D = 0.5m$.

Appendix F. Biped dynamics. In this section we will cover some of the more detailed calculations for the five-link biped model of walking, including 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 [37].

F.1. Kinematics. Let's start by defining the position vectors that point from the origin $\boldsymbol{P}_0$ to each joint of the robot $\boldsymbol{P}_i$ and the center of mass of each link $\boldsymbol{G}_i$, as shown in Figure 21. Each of these position vectors is dependent on the configuration of the robot: $\boldsymbol{P}_i = \boldsymbol{P}_i(\boldsymbol{q})$ and $\boldsymbol{G}_i = \boldsymbol{G}_i(\boldsymbol{q})$, where $\boldsymbol{q} = [q_1\ q_2\ q_3\ q_4\ q_5]^T$ is a column vector of absolute link orientations. We will define $\boldsymbol{P}_0 = \boldsymbol{0}$.

There are many ways to compute the position vectors. Here we work from the root joint $\boldsymbol{P}_0$ outward along the kinematic chain, defining each successive position $\boldsymbol{P}_i$ in terms of a previously defined position vector $\boldsymbol{P}_{i-1}$ and a relative vector in the link frame.

Once the position vectors are defined, we compute velocity and acceleration vectors using the chain rule. The velociies are given below, where $\dot{\boldsymbol{q}} = [\dot{q}_1\ \dot{q}_2\ \dot{q}_3\ \dot{q}_4\ \dot{q}_5]^T$ is the vector of absolute angular rates.

$$\dot{\boldsymbol{P}}_i = \left(\frac{\partial \boldsymbol{P}_i}{\partial \boldsymbol{q}}\right)\dot{\boldsymbol{q}} \qquad \dot{\boldsymbol{G}}_i = \left(\frac{\partial \boldsymbol{G}_i}{\partial \boldsymbol{q}}\right)\dot{\boldsymbol{q}} \tag{F.1}$$

TABLE 3
Physical parameters for the cart-pole example.

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 4
Physical parameters for the five link biped model (RABBIT) [13]

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

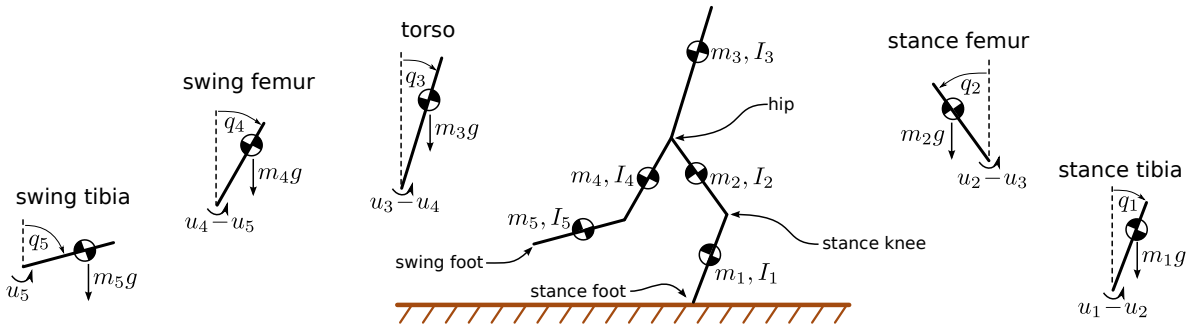

FIG. 20. *Dynamics model for the five-link biped model, shown here in single stance. We assume that the dynamics are planar (2D) and modeled as a kinematic chain, with each link assigned a number: 1 = stance tibia, 2 = stance femur, 3 = torso, 4 = swing femur, and 5 = swing tibia. Each joint is connected to its parent by an ideal revolute joint and torque source. Joint torques are given by u_i, link masses and inertias by m_i and I_i, and gravity is g. The absolute orientation of each link is given by q_i.*

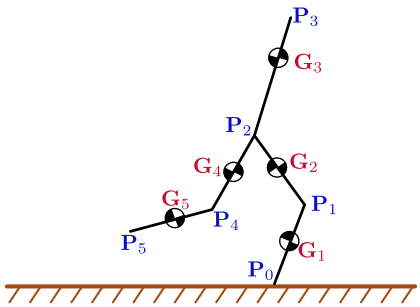

FIG. 21. *Kinematics for the five-link biped model. The illustration shows both joints $\boldsymbol{P}_i$ and the center of mass of each link $\boldsymbol{G}_i$.*

The calculation for the acceleration vectors 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 $\ddot{\boldsymbol{q}} = [\ddot{q}_1\ \ddot{q}_2\ \ddot{q}_3\ \ddot{q}_4\ \ddot{q}_5]^T$.

$$\ddot{\boldsymbol{P}}_i = \left(\frac{\partial \dot{\boldsymbol{P}}_i}{\partial \boldsymbol{z}}\right)\dot{\boldsymbol{z}} \qquad \ddot{\boldsymbol{G}}_i = \left(\frac{\partial \dot{\boldsymbol{G}}_i}{\partial \boldsymbol{z}}\right)\dot{\boldsymbol{z}} \tag{F.2}$$

Both of these calculations (F.1) and (F.2) can be implemented in Matlab with the following commands, where all variables are defined to be column vectors.

```
>> dP = Jacobian(P,q)*dq;
>> dG = Jacobian(G,q)*dq;
>> ddP = Jacobian(dP,[q; dq])*[dq; ddq];
>> ddG = Jacobian(dG,[q; dq])*[dq; ddq];
```

F.2. Single-stance dynamics. In trajectory optimization it is best to use a minimal coordinate formulation of the dynamics: one where there is one equation for each degree of freedom. For this example we will use the absolute angle of each link in the robot for the minimal coordinates, and compute their accelerations (the equations of motion) using the Newton-Euler equations. Although it is possible to derive these equations by hand, we suggest that you use a computer algebra package for the derivation, such as the Matlab Symbolic Toolbox [37] or the Python Symbolic Library [56].

The goal of the dynamics calculations are to arrive at a set of equations define the link accelerations $\ddot{\boldsymbol{q}}$ in terms of the link angles $\boldsymbol{q}$, rates $\dot{\boldsymbol{q}}$, and torques $\boldsymbol{u} = [u_1\ u_2\ u_3\ u_4\ u_5]^T$. Here we will use computer algebra to generate a linear system of equations, which we will then solve numerically at run time for the accelerations

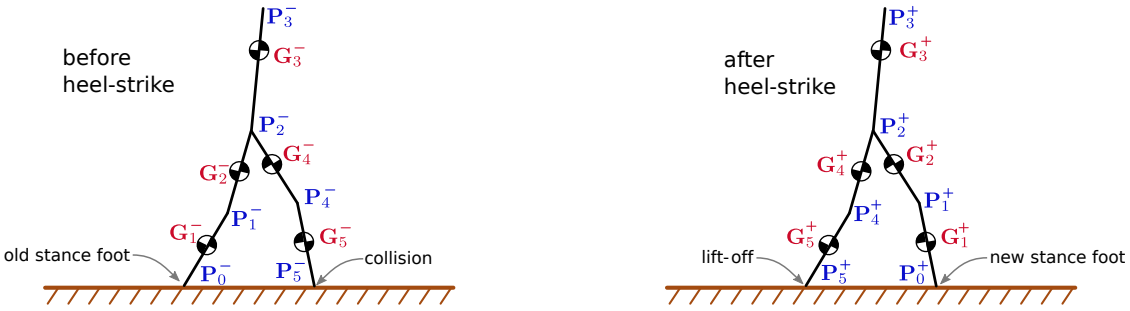

FIG. 22. *Illustration of the kinematics of the five-link biped model both before $^-$ and after $^+$ heel-strike. Note that the points on the robot are re-labeled during the collision, reflecting the left-right symmetry of the robot.*

$\ddot{\boldsymbol{q}}$. It turns out that this approach is significantly faster (both run time and derivation time) than solving for the joint accelerations explicitly.

$$\boldsymbol{\mathcal{M}}(\boldsymbol{q}) \cdot \ddot{\boldsymbol{q}} = \boldsymbol{\mathcal{F}}(\boldsymbol{q}, \dot{\boldsymbol{q}}, \boldsymbol{u}) \tag{F.3}$$

For our five-link biped, there are five linearly independent equations required to construct (F.3), 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. Here we will 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 $\boldsymbol{P}_0$.

$$u_1 + \hat{\boldsymbol{k}} \cdot \sum_{i=1}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_0) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=1}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_0) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \Big) \tag{F.4}$$

The next equation is obtained by simply moving one joint out along the robot, computing the angular momentum balance about the stance knee $\boldsymbol{P}_1$.

$$u_2 + \hat{\boldsymbol{k}} \cdot \sum_{i=2}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_1) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=2}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_1) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \Big) \tag{F.5}$$

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 $\boldsymbol{P}_2$.

$$u_3 + \hat{\boldsymbol{k}} \cdot \sum_{i=3}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_2) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=3}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_2) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \Big) \tag{F.6}$$

$$u_4 + \hat{\boldsymbol{k}} \cdot \sum_{i=4}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_2) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=4}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_2) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \Big) \tag{F.7}$$

$$u_5 + \hat{\boldsymbol{k}} \cdot \sum_{i=5}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_4) \times (-m_i\, g\, \hat{\boldsymbol{j}}) \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=5}^{5} \Big((\boldsymbol{G}_i - \boldsymbol{P}_4) \times (m_i\, \ddot{\boldsymbol{G}}_i) + \ddot{q}_i\, I_i\, \hat{\boldsymbol{k}} \Big) \tag{F.8}$$

F.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: 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 assume 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, but does not affect the configuration (angles). The second part of the map

swaps the swing and stance legs. The leg swap is done to enforce a symmetry in the solution: we want the step taken by the left leg to be identical to the right, and for both to be periodic.

Figure 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, and can be computed using the following equation.

$$\boldsymbol{q}^+ = \begin{bmatrix} 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 \end{bmatrix} \boldsymbol{q}^- \tag{F.9}$$

Next we derive a linear system that relates the angular rates before and after the collision. Like the single-stance dynamics, we will solve this system numerically at run time.

$$\boldsymbol{\mathcal{M}}_H(\boldsymbol{q}^-) \cdot \dot{\boldsymbol{q}}^+ = \boldsymbol{\mathcal{F}}_H(\boldsymbol{q}^-, \dot{\boldsymbol{q}}^-) \tag{F.10}$$

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, taken 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, taken about the stance foot (which was previously the swing foot). Figure 22 shows the naming conventions used throughout these equations. Note that the structure of these equations is somewhat similar to those used for the single stance dynamics.

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{5} \Big((\boldsymbol{G}_i^- - \boldsymbol{P}_5^-) \times (m_i \, \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=1}^{5} \Big((\boldsymbol{G}_i^+ - \boldsymbol{P}_0^+) \times (m_i \, \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \Big) \tag{F.11}$$

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{4} \Big((\boldsymbol{G}_i^- - \boldsymbol{P}_4^-) \times (m_i \, \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=2}^{5} \Big((\boldsymbol{G}_i^+ - \boldsymbol{P}_1^+) \times (m_i \, \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \Big) \tag{F.12}$$

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{3} \Big((\boldsymbol{G}_i^- - \boldsymbol{P}_2^-) \times (m_i \, \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=3}^{5} \Big((\boldsymbol{G}_i^+ - \boldsymbol{P}_2^+) \times (m_i \, \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \Big) \tag{F.13}$$

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{2} \Big((\boldsymbol{G}_i^- - \boldsymbol{P}_2^-) \times (m_i \, \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=4}^{5} \Big((\boldsymbol{G}_i^+ - \boldsymbol{P}_2^+) \times (m_i \, \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \Big) \tag{F.14}$$

$$\hat{\boldsymbol{k}} \cdot \sum_{i=1}^{1} \Big((\boldsymbol{G}_i^- - \boldsymbol{P}_1^-) \times (m_i \, \dot{\boldsymbol{G}}_i^-) + \dot{q}_i^- \, I_i \, \hat{\boldsymbol{k}} \Big) = \hat{\boldsymbol{k}} \cdot \sum_{i=5}^{5} \Big((\boldsymbol{G}_i^+ - \boldsymbol{P}_4^+) \times (m_i \, \dot{\boldsymbol{G}}_i^+) + \dot{q}_i^+ \, I_i \, \hat{\boldsymbol{k}} \Big) \tag{F.15}$$

Our final step is to combine (F.9) and (F.10) into the heel-strike map equation, shown below, where $\boldsymbol{x}^-$ is the state of the system before heel-strike and $\boldsymbol{x}^+$ is the state after heel-strike.

$$\boldsymbol{x}^- = \begin{bmatrix} \boldsymbol{q}^- \\ \dot{\boldsymbol{q}}^- \end{bmatrix} \qquad \boldsymbol{x}^+ = \begin{bmatrix} \boldsymbol{q}^+ \\ \dot{\boldsymbol{q}}^+ \end{bmatrix} \tag{F.16}$$

$$\boldsymbol{x}^+ = \boldsymbol{f}_H(\boldsymbol{x}^-) \tag{F.17}$$

F.4. Gradients. For trajectory optimization, it is generally a good idea to use analytic gradients where possible. This means that we will need to calculate the following expressions:

$$\frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{q}} \qquad \frac{\partial \ddot{\boldsymbol{q}}}{\partial \dot{\boldsymbol{q}}} \qquad \frac{\partial \ddot{\boldsymbol{q}}}{\partial \boldsymbol{u}} \qquad \frac{\partial \dot{\boldsymbol{q}}^+}{\partial \boldsymbol{q}^-} \qquad \frac{\partial \dot{\boldsymbol{q}}^+}{\partial \dot{\boldsymbol{q}}^-} \tag{F.18}$$

Unfortunately, we can't 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 $\boldsymbol{\mathcal{F}}_H$ and then derive an expression for the gradient of $\ddot{\boldsymbol{q}}$ and $\dot{\boldsymbol{q}}^+$ in terms of these known matrices. We start by deriving the gradient of the matrix inverse operator.

$$\boldsymbol{\mathcal{M}}^{-1}\boldsymbol{\mathcal{M}} = \boldsymbol{\mathcal{I}} \tag{F.19}$$

$$\frac{\partial}{\partial q_i}\left(\boldsymbol{\mathcal{M}}^{-1}\boldsymbol{\mathcal{M}}\right) = \boldsymbol{0} \tag{F.20}$$

$$\frac{\partial}{\partial q_i}\left(\boldsymbol{\mathcal{M}}^{-1}\right)\boldsymbol{\mathcal{M}} + \boldsymbol{\mathcal{M}}^{-1}\frac{\partial}{\partial q_i}\left(\boldsymbol{\mathcal{M}}\right) = \boldsymbol{0} \tag{F.21}$$

$$\frac{\partial \boldsymbol{\mathcal{M}}^{-1}}{\partial q_i} = -\boldsymbol{\mathcal{M}}^{-1}\frac{\partial \boldsymbol{\mathcal{M}}}{\partial q_i}\boldsymbol{\mathcal{M}}^{-1} \tag{F.22}$$

We will now apply (F.22) 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 partial derivatives with respect to the remaining joint angles, rates $\dot{q}_i$, and torques u_i. These same calculations (F.25) can be applied to the heel-strike calculations.

$$\frac{\partial \ddot{\boldsymbol{q}}}{\partial q_i} = \frac{\partial}{\partial q_i}\left(\boldsymbol{\mathcal{M}}^{-1}\boldsymbol{\mathcal{F}}\right) \tag{F.23}$$

$$\frac{\partial \ddot{\boldsymbol{q}}}{\partial q_i} = \left(-\boldsymbol{\mathcal{M}}^{-1}\frac{\partial \boldsymbol{\mathcal{M}}}{\partial q_i}\boldsymbol{\mathcal{M}}^{-1}\right)\boldsymbol{\mathcal{F}} \quad + \quad \boldsymbol{\mathcal{M}}^{-1}\left(\frac{\partial \boldsymbol{\mathcal{F}}}{\partial q_i}\right) \tag{F.24}$$

$$\frac{\partial \ddot{\boldsymbol{q}}}{\partial q_i} = \boldsymbol{\mathcal{M}}^{-1}\left(-\frac{\partial \boldsymbol{\mathcal{M}}}{\partial q_i}\ddot{\boldsymbol{q}} + \frac{\partial \boldsymbol{\mathcal{F}}}{\partial q_i}\right) \tag{F.25}$$

REFERENCES

[1] S. Agrawal, S. Shen, and M. V. D. Panne, *Diverse motion variations for physics-based character animation*, Proceedings of the 12th ACM SIGGRAPH/Eurographics Symposium on Computer Animation - SCA '13, (2013), pp. 37–44.

[2] V. M. Becerra, *PSOPT Optimal Control Solver User Manual*, 2011.

[3] D. a. Benson, G. T. Huntington, T. P. Thorvaldsen, and A. V. Rao, *Direct Trajectory Optimization and Costate Estimation via an Orthogonal Collocation Method*, Journal of Guidance, Control, and Dynamics, 29 (2006), pp. 1435–1440.

[4] J.-P. Berrut and L. N. Trefethen, *Barycentric Lagrange Interpolation*, SIAM Review, 46 (2004), pp. 501–517.

[5] J. T. Betts, *A Survey of Numerical Methods for Trajectory Optimization*, Journal of Guidance, Control, and Dynamics, (1998), pp. 1–56.

[6] ———, *Practical Methods for Optimal Control and Estimation Using Nonlinear Programming*, Siam, Philadelphia, PA, 2010.

[7] ———, *SOS: Sparse Optimization Suite - User's Guide*, 2013.

[8] 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 (2014), pp. 1305–1321.

[9] 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).

[10] 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 (2009), pp. 575–582.

[11] A. E. Bryson and Y.-C. Ho, *Applied Optimal Control*, Taylor & Francis, 1975.

[12] E. Catto, *Box2D User Manual*, 2013.

[13] B. C. Chevallereau, G. Abba, Y. Aoustin, F. Plestan, E. R. Westervelt, C. Canudas-de wit, and J. W. Grizzle, *RABBITA Testbed for Advanced Control Theory*, IEEE Control Systems Mag., 23 (2003), pp. 57–79.

[14] E. Coumans, *Bullet Physics SDK Manual*, 2015.

[15] C. L. Darby, D. Garg, and A. V. Rao, *Costate Estimation using Multiple-Interval Pseudospectral Methods*, Journal of Spacecraft and Rockets, 48 (2011), pp. 856–866.

[16] C. L. Darby, W. W. Hagar, and A. V. Rao, *An hp-adaptive pseudospectral method for solving optimal control problems*, Optimal Control Applications and Methods, 32 (2011), pp. 476–502.

[17] T. A. Driscoll, N. Hale, and L. N. Trefethen, *Chebfun Guide*, Pafnuty Publications, Oxford, 1 ed., 2014.

[18] G. Elnagar, M. A. Kazemi, and M. Razzaghi, *The Pseudospectral Legendre Method for Discretizing Optimal Control Problems*, IEEE, 40 (1995), pp. 1793–1796.

[19] G. N. Elnagar and M. a. Kazemi, *Pseudospectral Chebyshev optimal control of constrained nonlinear dynamical systems*, Computational Optimization and Applications, 217 (1998), pp. 195–217.

[20] B. Fornberg, *A practical guide to pseudospectral methods*, Cambridge University Press, 1996.

[21] C. C. Francolin, D. A. Benson, W. W. Hager, and A. V. Rao, *Costate Estimation in Optimal Control Using Integral Gaussian Quadrature Orthogonal Collocation Methods*, Optimal Control Applications and Methods, (2014).

[22] D. Garg, M. Patterson, and W. Hager, *An Overview of Three Pseudospectral Methods for the Numerical Solution of Optimal Control Problems*, Advances in the . . . , (2009), pp. 1–17.

[23] D. Garg, M. Patterson, W. W. Hager, A. V. Rao, D. a. Benson, and G. T. Huntington, *A unified framework for the numerical solution of optimal control problems using pseudospectral methods*, Automatica, 46 (2010), pp. 1843–1851.

[24] G. H. Golub and J. H. Welsch, *Calculation of Gauss quadrature rules*, Mathematics of Computation, 23 (1968), pp. 221–221.

[25] J. W. Grizzle, J. Hurst, B. Morris, H. W. Park, and K. Sreenath, *MABEL, a new robotic bipedal walker and runner*, Proceedings of the American Control Conference, (2009), pp. 2030–2036.

[26] W. W. Hager and A. V. Rao, *Gauss Pseudospectral Method for Solving Infinite-Horizon Optimal Control Problems*, (2010), pp. 1–9.

[27] N. Hale and A. Townsend, *Fast and Accurate Computation of Gauss–Legendre and Gauss–Jacobi Quadrature Nodes and Weights*, SIAM Journal on Scientific Computing, 35 (2013), pp. A652–A674.

[28] 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 (1987), pp. 338–342.

[29] 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 (1996), pp. 522–529.

[30] D. H. Jacobson and D. Q. Mayne, *Differential Dynamic Programming*, Elsevier, 1970.

[31] G. Klein and J.-P. Berrut, *Linear barycentric rational quadrature*, BIT Numerical Mathematics, 52 (2012), pp. 407–424.

[32] D. P. Laurie, *Computation of Gauss-type quadrature formulas*, Journal of Computational and Applied Mathematics, 127 (2001), pp. 201–217.

[33] L. Liu, M. V. D. Panne, and K. Yin, *Guided Learning of Control Graphs for Physics-Based Characters*, ACM Transactions on Graphics, 35 (2016), pp. 1–14.

[34] D. G. Luenberger and Y. Ye, *Linear and Nonlinear Programming*, Springer, third edit ed., 2008.

[35] 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 (2012), pp. 796–803.

[36] Mathworks, *Matlab Optimization Toolbox*, 2014.

[37] ———, *Matlab Symbolic Toolbox*, 2014.

[38] D. Mayne, *A Second-order Gradient Method for Determining Optimal Trajectories of Non-linear Discrete-time Systems*, International Journal of Control, 3 (1966), pp. 85–95.

[39] I. Mordatch, E. Todorov, and Z. Popović, *Discovery of complex behaviors through contact-invariant optimization*, ACM Transactions on Graphics, 31 (2012), pp. 1–8.
[40] 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 (1984), pp. 395–414.
[41] A. Ng, *Stanford CS 229 Lecture Notes*, in Machine Learning, 2012, ch. XIII - Rei, pp. 1–15.
[42] X. B. NPeng, G. Berseth, and M. van de Panne, *Dynamic Terrain Traversal Skills Using Reinforcement Learning*, in SIGGRAPH, 2015.
[43] H. W. Park, K. Sreenath, A. 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, (2012), pp. 45–50.
[44] S. V. Parter, *On the Legendre-Gauss-Lobatto Points and Weights*, Journal of Scientific Computing, 14 (1999), pp. 347–355.
[45] M. A. Patterson and A. 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 (2013), pp. 1–41.
[46] M. Posa, S. Kuindersma, and R. Tedrake, *Optimization and stabilization of trajectories for constrained dynamical systems*, Proceedings - IEEE International Conference on Robotics and Automation, 2016-June (2016), pp. 1366–1373.
[47] M. Posa and R. Tedrake, *Direct Trajectory Optimization of Rigid Body Dynamical Systems Through Contact*, Algorithmic Foundations of Robotics X, (2013), pp. 527–542.
[48] J. Pratt, *Virtual Model Control: An Intuitive Approach for Bipedal Locomotion*, The International Journal of Robotics Research, 20 (2001), pp. 129–143.
[49] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Cambridge University Press, second edi ed.
[50] L.-s. N. Programming, P. E. Gill, W. Murray, and M. A. Saunders, *User ' s Guide for SNOPT Version 7 : Software for*, (2006), pp. 1–116.
[51] A. Rao, *A survey of numerical methods for optimal control*, Advances in the Astronautical Sciences, 135 (2009), pp. 497–528.
[52] I. M. Ross, *DIDO*, 2001.
[53] I. M. Ross and F. Fahroo, *Legendre pseudospectral approximations of optimal control problems*, New Trends in Nonlinear Dynamics and Control and their Applications, 295 (2003), pp. 327–342.
[54] C. O. Saglam and K. Byl, *Robust Policies via Meshing for Metastable Rough Terrain Walking*.
[55] M. Srinivasan and A. Ruina, *Computer optimization of a minimal biped model discovers walking and running.*, Nature, 439 (2006), pp. 72–5.
[56] SymPy Development Team, *SymPy: Python library for symbolic mathematics*, 2016.
[57] L. N. Trefethen, *A rational spectral collocation method with adaptively transformed chebyshev grid points* , 28 (2006), pp. 1798–1811.
[58] ———, *Approximation Theory and Approximation Practice*, SIAM, 2013.
[59] V. a. Tucker, *Energetic cost of locomotion in animals.*, Comparative biochemistry and physiology, 34 (1970), pp. 841–846.
[60] C. D. Twigg and D. L. James, *Many-worlds browsing for control of multibody dynamics*, ACM Transactions on Graphics, 26 (2007), p. 14.
[61] ———, *Backward steps in rigid body simulation*, ACM Transactions on Graphics (TOG), 27 (2008), p. 1.
[62] J. Vlassenbroeck and R. V. Dooren, *A Chebyshev technique for solving nonlinear optimal control problems*, Automatic Control, IEEE ..., 33 (1988).
[63] 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, (1999).
[64] A. Wächter and L. T. Biegler, *On the implementation of primal-dual interior point filter line search algorithm for large-scale nonlinear programming*, vol. 106, 2006.
[65] H. Wang and S. Xiang, *On the Convergence Rate of Legendre Approximation*, Mathematics of Computation, 81 (2011), pp. 861–877.
[66] E. R. Westervelt, J. W. Grizzle, and D. E. Koditschek, *Hybrid zero dynamics of planar biped walkers*, IEEE Transactions on Automatic Control, 48 (2003), pp. 42–56.
[67] 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 (2009), pp. 277–290.