Nonlinear Optimal Control: Reductions to Newton Optimization

Nathan Ratliff

May 6, 2014

Abstract

This document explores nonlinear optimal control from the perspective of generic nonlinear optimization. We discuss first why optimization is important to control and see one case, in particular, where optimizing future performance leads to a simple control algorithm that differs in a very interesting and intuitive way from an analogous PD control rule that we might have written down directly. Building from that example, we then develop some nonlinear optimal control algorithms and principles more concretely and show that the natural approach of iteratively solving second-order approximations to the problem parallels a more general methodology of Newton's method for unconstrained nonlinear optimization. In other words, classic algorithms such as DDP and iLQR may be understood simply as variants on Newton's method that exploit the underlying structure of the discrete time optimal control problem. This observation makes available a broad set of tools for solving these problem such as line search techniques, trust region methods, and nonlinear methods from constrained optimization. To this end we discuss the basic principles behind Newton's method for general nonlinear optimization and derive an inequality constrained variant of the Augmented Lagrangian algorithm and discuss their applicability to optimal control.

1 Why optimal control?

Consider a simple fixed-based system shown in Figure 1, and suppose we want to control the system's end-effector, defined by the forward kinematic function $\boldsymbol{x} = \phi(\boldsymbol{q})$, to some desired point $\boldsymbol{x}_d$. For simplicity, we'll assume that the origin of the Cartesian space coordinate system is at desired destination, so we say that we simply want to control the robot's end-effector to the origin $\boldsymbol{x}_d = 0$. How might we design such a controller?

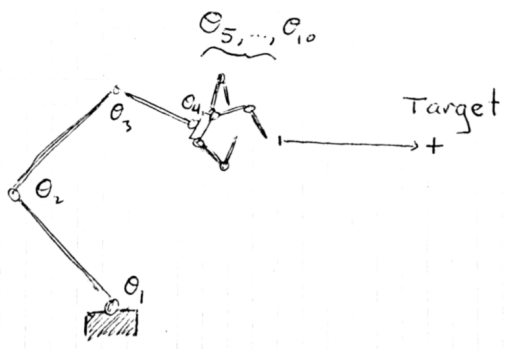

Figure 1: Simple fixed based manipulator. We want to control its fingertip (end-effector) to the indicated target.

1.1 A classical approach

Classically, we might set up a PD controller in the Cartesian space using the error $\boldsymbol{e} = \boldsymbol{x}_d - \boldsymbol{x}$ to implement a control rule of the form

$$\ddot{\boldsymbol{x}}_d = \boldsymbol{K}_p \boldsymbol{e} - \boldsymbol{K}_d \dot{\boldsymbol{x}}, \tag{1}$$

where $\boldsymbol{K}_p$ and $\boldsymbol{K}_d$ are typically diagonal control gain matrices that allow us to rescale individual dimensions as we see fit and trade off the damping term effectively with the proportional error term. Then there are a number of ways to turn that desired Cartesian acceleration into a torque command. We might use the equation $\ddot{\boldsymbol{x}} = \boldsymbol{J}_\phi \ddot{\boldsymbol{q}} + \dot{\boldsymbol{J}}_\phi \dot{\boldsymbol{q}}$, where $\boldsymbol{J}_\phi$ is the Jacobian of the kinematic function ϕ, along with some form of null space resolution, to first transform the acceleration into a desired configuration space acceleration $\ddot{\boldsymbol{q}}_d$ and then use inverse dynamics to calculate torque commands. Or we might use a more general QP approach to minimize a quadratic objective of the form

$$Q(\ddot{\boldsymbol{x}}, \boldsymbol{\tau}) = \frac{1}{2}\|\ddot{\boldsymbol{x}} - \ddot{\boldsymbol{x}}_d\|^2 + \frac{\alpha}{2}\|\boldsymbol{\tau}\|^2 \tag{2}$$

subject to constraints imposed by the fixed-base equations of motion (out of contact) $\boldsymbol{M}\ddot{\boldsymbol{q}} + \boldsymbol{h} = \boldsymbol{\tau}$ in addition to those kinematic equations $\ddot{\boldsymbol{x}} = \boldsymbol{J}_\phi \ddot{\boldsymbol{q}} + \dot{\boldsymbol{J}}_\phi \dot{\boldsymbol{q}}$.

This QP formulation gives us a lot of flexibility in the design of null space resolution and effort trade-offs, but even if we solve this equality constrained QP, it's behavior is still fundamentally defined by the behavior of the simple PD control differential equation given by Equation 1. Does that PD behavior accurately describe how we want the robot behave? It effectively simulates attaching a spring between the robot's end-effector and the desired point, placing the entire system in a viscous substance like honey, and letting it go. The spring exerts a force and pulls the end-effector toward the desired point, and the viscous solution exerts a *damping* force that tries to slow the end-effector

down proportionally to how fast it's going. Do these simple principles of control, built by analogy to physical spring-damper systems, really describe the nuances of the behavior we'd like to see?

1.2 One-step optimal control

Our answer to the above question is clearly "no." That's just an opinion, but there's increasing evidence that Optimal Control (especially Stochastic Optimal Control (see Todorov & Jordan (2002))) forms a good model of observed human behavior. This section won't go into detail about that work here, but instead more generally motivates nonlinear optimal control. We introduce a one-step lookahead optimal control approach to the above reaching example that derives an alternative rule for generating accelerations. The derived rule is similar to the PD rule, but differs in a very interesting and intuitive way. It explicitly accounts for the shape of the potential in how it defines its velocity damping which enables it to converge much more precisely to the desired point in practice. Importantly for our purposes, this example demonstrates the basic elements and tools that go into making nonlinear optimal control work.

Typically for problems like this we have some function in Cartesian space $\psi : \mathbb{R}^3 \to \mathbb{R}$ that defines progress toward the goal. It's large when we're far from our goal, it decreases as we approach the goal, and it achieves a minimum when we reach the goal. We find a number of different names for functions like this in the literature, such as goal potentials, terminal potentials, heuristic functions, value function approximations, cost-to-go approximators, etc. For our purposes, all of these names are essentially equivalent; we'll see below why some of the more obscure names make theoretical sense.

For instance, if $\boldsymbol{x}_d = 0$, as we're considering here, one such potential function may be $\psi(\boldsymbol{x}) = \|\boldsymbol{x}\|$. This function is simply the distance (the raw Euclidean distance and not the squared distance) to the desired position, in this case the origin.

Such a potential function is a very natural description of progress (we want to move in a straight line to the goal), but it's written in terms of $\boldsymbol{x}$ and doesn't directly tell us anything about $\ddot{\boldsymbol{x}}$ let along $\boldsymbol{\tau}$. How can we turn it into something at the very least that gives us a rule for Cartesian space accelerations $\ddot{\boldsymbol{x}}$ that we can use in place of Equation 1?

Consider the end-effector as though it were a point particle floating in space (with no gravitational or viscous (air) forces), and suppose we applied an acceleration $\ddot{\boldsymbol{x}}$ to that point particle. If the point particle is currently at position $\boldsymbol{x}_t$ and moving at velocity $\dot{\boldsymbol{x}}_t$, we know from basic mechanics that after a time interval of Δt, the particle will end up at a new position

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t + \Delta t \dot{\boldsymbol{x}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{x}}_t. \tag{3}$$

This equation gives us a glimpse into the future. It allows us to predict where we'd be one time step from now if we apply a particular acceleration

$\ddot{\boldsymbol{x}}$ for Δt seconds. Thus, plugging this equation into ψ as its input gives us a function over accelerations

$$\widetilde{\psi}(\ddot{\boldsymbol{x}}) = \psi(\boldsymbol{x}_t + \Delta t \dot{\boldsymbol{x}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{x}}) \tag{4}$$

which measures the relative performance of any possible acceleration. $\widetilde{\psi}$ tells us what value ψ would have at the place we'd end up after accelerating at a rate of $\ddot{\boldsymbol{x}}$ for Δt seconds had we started with position and velocity $\boldsymbol{x}_t$ and $\dot{\boldsymbol{x}}_t$. It's reasonable, therefore, to define the desired acceleration now using a rule of the form

$$\ddot{\boldsymbol{x}}_t = \underset{\ddot{\boldsymbol{x}}}{\operatorname{argmin}} \left(\widetilde{\psi}(\ddot{\boldsymbol{x}}) + \frac{\lambda}{2}\|\ddot{\boldsymbol{x}}\|^2 \right). \tag{5}$$

This rule says simply that we want to choose the acceleration that would get us to the best place (as measured by ψ), subject to that acceleration not being too large (hence the last term, which penalizes the size of accelerations—with large enough accelerations we can get anywhere, including the global minimum of ψ, in just one time step, and that's not realistic). This rule in Equation 5 is now something we can use in place of the above PD rule. And this new rule explicitly optimizes future performance over one time step.

Unfortunately, this expression isn't directly something we can easily analyze (and analytically optimize) since the objective isn't our typical friendly quadratic function. However, as always, we can approximate it using a second-order Taylor expansion of ψ around our current position $\boldsymbol{x}_t$:

$$\begin{aligned} \widetilde{\psi}(\ddot{\boldsymbol{x}}) \approx \psi(\boldsymbol{x}_t) &+ \nabla\psi^T(\Delta t \dot{\boldsymbol{x}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{x}}) \\ &+ \frac{1}{2}\left(\Delta t \dot{\boldsymbol{x}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{x}}\right)^T \left[\ \nabla^2\psi\ \right] \left(\Delta t \dot{\boldsymbol{x}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{x}}\right) \\ &+ \frac{\lambda}{2}\|\ddot{\boldsymbol{x}}\|^2, \end{aligned} \tag{6}$$

where we've dropped the dependence of the gradient $\nabla\psi(\boldsymbol{x}_t)$ and Hessian $\nabla^2\psi(\boldsymbol{x}_t)$ on $\boldsymbol{x}_t$ to simplify the notation.

Now we can optimize the function analytically and solve for the update rule 5. Optimizing analytically amounts to setting the gradient (w.r.t. $\ddot{\boldsymbol{x}}$) of the right hand side of Equation 6 to zero and solving for $\ddot{\boldsymbol{x}}$:

$$\begin{aligned} &\nabla_{\ddot{\boldsymbol{x}}}\widetilde{\psi}(\ddot{\boldsymbol{x}}) \approx \left(\frac{1}{2}\Delta t^2\right)\nabla\psi + \left(\frac{1}{2}\Delta t^2\right)\left[\ \nabla^2\psi\ \right]\left(\Delta t \dot{\boldsymbol{x}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{x}}\right) + \lambda\ddot{\boldsymbol{x}} = 0 \\ \Rightarrow \quad &\nabla\psi + \Delta t \nabla^2\psi \dot{\boldsymbol{x}}_t + \underbrace{\left(\frac{\Delta t^2}{2}\nabla^2\psi + \frac{2\lambda}{\Delta t^2}\boldsymbol{I}\right)}_{\boldsymbol{A}(\Delta t, \lambda)} \ddot{\boldsymbol{x}} = 0. \end{aligned} \tag{7}$$

Note that the matrix $\boldsymbol{A}(\Delta t, \lambda)$ depends on the size of the time step Δt and the strength of the parameter λ which constrains the size of $\ddot{\boldsymbol{x}}$. In particular,

when Δt is small, $\boldsymbol{A}$ emphasizes primarily the second term $\boldsymbol{I}$, and additionally, as λ increases (i.e. we constrain the size of $\ddot{\boldsymbol{x}}$ more), $\boldsymbol{A}$ doubly emphasizes $\boldsymbol{I}$. Thus, for the most part we can think of $\boldsymbol{A}$ and its inverse as approximately proportional to $\boldsymbol{I}$. Effectively, it doesn't do much besides scale the result.

Solving for $\ddot{\boldsymbol{x}}$, therefore, gives

$$\begin{aligned} \ddot{\boldsymbol{x}}_t &= \boldsymbol{A}^{-1} \Big[\; -\nabla\psi - \Delta t \nabla^2 \psi \dot{\boldsymbol{x}}_t \; \Big] \\ &\approx \alpha \left(-\nabla\psi \; - \; \Delta t \nabla^2 \psi \dot{\boldsymbol{x}}_t \right), \end{aligned} \tag{8}$$

where α is some proportionality constant.

Referring back to the simple PD rule in Equation 1, this new expression bears a striking resemblance. If ψ is a squared distance to the goal, then the negative gradient is the error between the goal point and the current location, and the Hessian is just the identity, so this expression is practically the same. But here, we've derived this rule generically for any differentiable ψ. And using $\psi(\boldsymbol{x}) = \|\boldsymbol{x}\|$, as is our running example, produces a different but very interesting update rule.

The gradient and Hessian of $\psi(\boldsymbol{x}) = \|\boldsymbol{x}\|$ are

$$\begin{aligned} \nabla\psi(\boldsymbol{x}) &= \widehat{\boldsymbol{x}} \\ \nabla^2\psi(\boldsymbol{x}) &= \frac{1}{\|\boldsymbol{x}\|} \left(\boldsymbol{I} - \widehat{\boldsymbol{x}}\widehat{\boldsymbol{x}}^T \right). \end{aligned}$$

(See Appendix A for a brief derivation.) Here $\widehat{\boldsymbol{x}} = \boldsymbol{x}/\|\boldsymbol{x}\|$ is the normalized vector pointing in the direction of $\boldsymbol{x}$.

The *negative* gradient is just a unit vector pointing directly toward the desired destination (in this case the origin), and the Hessian is a projection matrix that projects orthogonally to that direction (see Appendix B) with an extra scaling factor of $1/\|\boldsymbol{x}\|$ that increases drastically as $\boldsymbol{x}$ approaches the origin.

Examining the optimization-based update rule in Equation 8 and using $\psi(\boldsymbol{x}) = \|\boldsymbol{x}\|$ (dropping the factor[1] $\boldsymbol{A}^{-1}$), we get a rule that chooses accelerations according to

$$\begin{aligned} \ddot{\boldsymbol{x}}_t \;&\propto\; -\nabla\psi - \Delta t \; \nabla^2\psi \; \dot{\boldsymbol{x}}_t \\ &=\; -\widehat{x} - \frac{\Delta t}{\|\boldsymbol{x}\|} \left(\boldsymbol{I} - \widehat{\boldsymbol{x}}\widehat{\boldsymbol{x}}^T \right) \dot{\boldsymbol{x}}_t. \end{aligned}$$

This rule tells us that, like the simple PD rule, we should accelerate in the direction of the origin. But unlike the PD rule, rather than adding a blanket damping term, this procedure suggests a more informed damping term that specifically only damps the components of velocities orthogonal to the direction

[1]Note that the behavior of $\boldsymbol{A}^{-1}$ is more subtle than we've depicted here near the origin since the factor $\frac{1}{\|x\|}$ approaches infinity. That nuance is actually less interesting to us here since in practice one would actually choose a smoother version of the potential function that retains differentiability at the origin. For instance, a practical candidate is the soft-max between $\|\boldsymbol{x}\|$ and its negative: $\psi(\boldsymbol{x}) = \log(e^{\|\boldsymbol{x}\|} + e^{-\|\boldsymbol{x}\|}) = \|x\| + \log(1 + e^{-2\|\boldsymbol{x}\|})$.

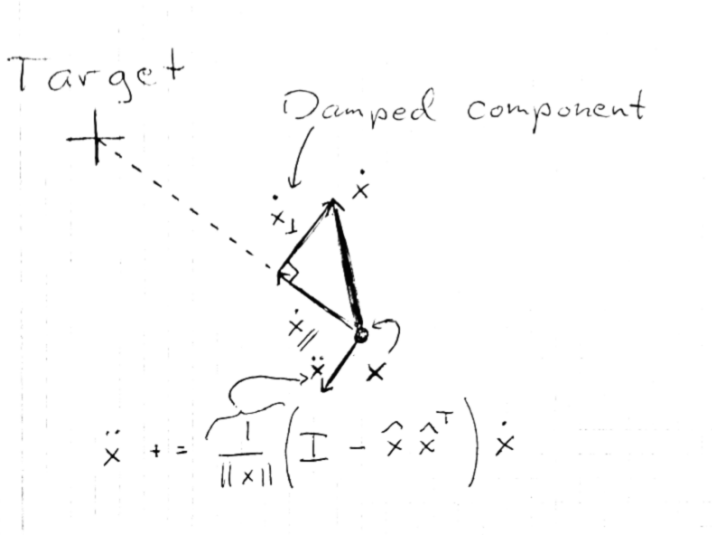

Figure 2: The optimal one-step lookahead rule discovers a policy that actively damps specifically the component of velocity orthogonal to the desired direction of travel.

we'd like the end-effector to be moving. Figure 2 depicts this damped direction pictorially. Importantly, that damping term is scaled by a factor $\frac{1}{\|\boldsymbol{x}\|}$ since the curvature of the potential becomes tighter as $\boldsymbol{x}$ approaches the origin. In terms of the update rule, this means that the term damping (erroneous) velocity components orthogonal to the desired direction of motion becomes more aggressive as the end-effector approaches the desired origin point in order to ensure that those components fully vanish.

This adaptive damping, which specifically act only on the components of the velocity that indicate movement in the wrong direction, is very intuitive, but it's something we probably wouldn't have hacked together on our own using finely tuned PD rules. It takes some insight. But leveraging optimality principles with forward prediction and Taylor approximations, deriving this rule is as straightforward as writing down a potential function ψ that intuitively measures progress toward the solution.

Note that if we modify the potential function ψ to measure progress to the goal in terms of both position $\boldsymbol{x}$ and velocity $\dot{\boldsymbol{x}}$, we can specify that the end-effector reach the desired point (the origin) with, for instance, zero velocity:

$$\psi(\boldsymbol{x}, \dot{\boldsymbol{x}}) = \|\boldsymbol{x}\| + \gamma \|\dot{\boldsymbol{x}}\|, \tag{9}$$

where $\gamma > 0$ is some positive trade-off constant. Using the velocity update equation $\dot{\boldsymbol{x}}_{t+1} = \dot{\boldsymbol{x}}_t + \Delta t \ddot{\boldsymbol{x}}$, we can perform a similar transformation of this second term as we did above for the first term. Following this procedure leads

to a similar acceleration rule with a new damping term that now regulates the overall velocity of the end-effector:

$$\ddot{\boldsymbol{x}} \approx \alpha \left(-\widehat{\boldsymbol{x}} - \Delta t \left(\boldsymbol{I} - \widehat{x}\widehat{x}^T \right) \dot{\boldsymbol{x}} - \frac{2\gamma}{\Delta t} \widehat{\dot{\boldsymbol{x}}}. \right).$$

2 Foreseeing the future

The above example demonstrates many of the elements of modern approaches to nonlinear optimal control. In general, we define a function ψ over the space of positions and velocities that measures progress toward some task or goal. We then predict forward into the future to see where the robot would be after taking one or more actions and evaluate that forward prediction under ψ. That composition gives us a nonlinear function over those hypothesized actions which predicts the value of taking each action. And finally, we use a second-order Taylor expansion (perhaps with some approximations to the Hessian) to turn that nonlinear function into a nice quadratic function that we can analyze (in particular, optimize) more easily. This section formalizes these ideas in a more general setting enabling us to look even further into the future.

Suppose we have some abstract dynamics function of the form $\boldsymbol{s}_{t+1} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{u}_t)$ where $\boldsymbol{s}_t$ is an abstract state of the system at time t (which may include both position $\boldsymbol{q}_t$ and velocity $\dot{\boldsymbol{q}}_t$ components), and $\boldsymbol{u}_t$ is an abstract action applied at that time (e.g. it may be the torques applied to the robot's joints). The discrete-time dynamics function $\boldsymbol{f}$ assumes that action $\boldsymbol{u}_t$ is applied uniformly for the entire time period Δt from time t to time $t+1$, and it tells us how the state transforms under the action during that time period. This abstract function $\boldsymbol{f}$ is analogous to the simple Cartesian space kinematic equations we saw in the example above:

$$\boldsymbol{s}_{t+1} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{u}_t) \quad \rightarrow \quad \begin{bmatrix} \boldsymbol{x}_{t+1} \\ \dot{\boldsymbol{x}}_{t+1} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_t + \Delta t \dot{\boldsymbol{x}}_t + \frac{1}{2}\Delta t^2 \ddot{\boldsymbol{x}}_t \\ \dot{\boldsymbol{x}}_t + \Delta t \ddot{\boldsymbol{x}}_t \end{bmatrix}, \tag{10}$$

with $\boldsymbol{s}_t = [\boldsymbol{x}_t; \dot{\boldsymbol{x}}_t]$ and $\boldsymbol{u}_t = \ddot{\boldsymbol{x}}_t$, but in this section we make no assumptions about $\boldsymbol{f}$'s specific structure, besides assuming that it is second-order differentiable in both inputs.

The dynamics function $\boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{u}_t)$ enables us to see into the future. Starting from some state $\boldsymbol{s}_1$, if we posit that the robot take T actions $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_T$, we can calculate precisely where the robot will be at time $T+1$ by recursively applying the dynamics function

$$\begin{aligned} \boldsymbol{s}_2 &= \boldsymbol{f}(\boldsymbol{s}_1, \boldsymbol{u}_1) \\ \boldsymbol{s}_3 &= \boldsymbol{f}(\boldsymbol{s}_2, \boldsymbol{u}_2) \\ &\vdots \\ \boldsymbol{s}_{T+1} &= \boldsymbol{f}(\boldsymbol{s}_T, \boldsymbol{u}_T). \end{aligned}$$

We typically care about two types of performance measure when examining future states $\boldsymbol{s}_t$ that result from actions $\boldsymbol{u}_t$. The first type of cost function measures how costly it is that we find ourselves in state $\boldsymbol{s}_t$ taking action $\boldsymbol{u}_t$. For instance, if a robot finds itself close to a dangerous obstacle such as the edge of a 100m cliff, and especially if it takes an action that brings it even closer to the edge, then that's much more costly than finding itself out dead center in the middle of a parking lot not moving at all. That last example might not be very good for achieving its goal (unless coincidentally that's precisely where it wanted it be), but this state cost term doesn't care about goals, it cares only very locally about how bad it is for the robot to find itself in its current state taking its current action. We model this *state-action cost* using a term $c_t(\boldsymbol{s}_t, \boldsymbol{u}_t)$. Generally, these terms can encode information such as proximity to joint limits or obstacles in the environment, secondary tasks such as holding objects upright, or even biases for contacting surfaces in the environment to promote stability. The second type of cost function measures progress toward some goal as defined by a *terminal cost* $\psi(\boldsymbol{s}_{T+1})$. This goal is an abstract notion of what we would like the robot to accomplish (eventually). For instance, the above reaching task measured progress toward some distinct goal state. Other terminal costs may measure how far the center of mass of the robot is at time $T+1$ from some desired location, or it may measure simply proximity to losing balance is all we want is the robot to stay upright.

Using both types of terms, we can write the problem of optimizing actions over future performance as

$$\begin{aligned} &\min_{\boldsymbol{s}_{1:T+1}, \boldsymbol{u}_{1:T}} \sum_{t=1}^{T} c_t(\boldsymbol{s}_t, \boldsymbol{u}_t) + \psi(\boldsymbol{s}_{T+1}) \\ &\text{s.t.} \quad \boldsymbol{s}_{t+1} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{u}_t) \quad \text{for each } t = 1, \ldots, T \\ &\text{[More equality and inequality constraints].} \end{aligned} \tag{11}$$

As indicated in Equation 11, we can additionally add more problem specific equality and inequality constraints. For instance, rather than measuring proximity to objects in the environment as costs, we could encode the surfaces in the environment using the zero sets of functions $\boldsymbol{g}(\boldsymbol{s}_t)$ that are positive outside obstacles and negative when the robot is penetrating an obstacle. In that case, the requirement that the robot should not be penetrating any obstacle may be specified abstractly as the inequality constraints $\boldsymbol{g}(\boldsymbol{s}_t) \geq 0$ for all $t = 1, \ldots, T+1$.

Most of this document assumes that such inequality constraints are modeled instead in terms of intermediate cost terms, but Section 4.2 reviews an effective nonlinear constrained optimization algorithm that constructs increasingly accurate unconstrained objectives by converting these extra constraints into iteratively tuned penalties. The algorithm, in effect, automates the process of reducing the problem to unconstrained optimization, taking that potentially finicky step out of the hands of the controls engineer.

In the next section we'll see the role quadratic approximations can play in classical approaches to this nonlinear optimal control problem, and then later

in Section 5 we'll see that we can understand these approaches simply as applications of Newton's method that exploit structure in the Hessian for efficient computation of Newton steps. Before we get there, though, Section 4 covers some general unconstrained and constrained methods for nonlinear optimization and particularly discusses the importance of building and solving local quadratic approximations in nonlinear optimization.

3 Quadratic approximations in nonlinear optimal control

As we saw in the example from Section 1.2, writing down the objective function usually isn't enough. We also need to optimize it. One way to do that, of course, is to perform some form of gradient descent, but the convergence of such algorithms can be slow due to chattering down long narrow troughs of the objective landscape. So instead, to both optimization and analytic gain, we first took a second-order Taylor expansion of the objective to find its best quadratic approximation. That step is precisely what we do in this more general T-step nonlinear control setting.

One of the primary questions that arises in this context of second-order Taylor expansion is, where? Around what points do we create the approximation? This section explores the second-order Taylor expansion in the context of nonlinear optimal control and this question in particular.

Here we consider only objective functions and dynamics, which is the classical setting. We'll assume there are no hard equality constraints (except for the dynamics function) and that any inequality constraints are encoded as cost terms (e.g. barrier or penalty functions) that approach infinity at the boundary (for instance, there might be a joint limit potential whose cost increases to infinity as the robot approaches a joint limit). Section 4.2 introduces some techniques from optimization for relaxing these assumptions.

3.1 The question of where

Where should we calculate a second-order Taylor expansions of our objective terms? The second-order expansion of the tth term $c_t(\boldsymbol{s}_t, \boldsymbol{u}_t)$ requires that we choose nominal points $\{(\boldsymbol{s}_t, \boldsymbol{u}_t)\}_{t=1}^{T}$ around which to create the expansion in advance. But the reason we're taking this second-order expansion is to answer the question of what the best $\boldsymbol{s}_t$ and $\boldsymbol{u}_t$ would be. This chicken-and-egg problem is a general problem in nonlinear optimization, and Section 4.1 explores it in more detail. But within the current optimal control context, we'll describe an intuitive iterative algorithm that, as we'll see below (modulo some technical details), is the one that's generally used in practice.

The best second-order approximation of the problem is the one formed around the optimal trajectory $(\boldsymbol{s}_1, \boldsymbol{u}_1^*), \ldots, (\boldsymbol{s}_T, \boldsymbol{u}_T^*)$. So heuristically, one may presume that the better we make nominal trajectory, the better that second-order approximation will be, thereby leading to better nominal trajectories

through optimization. That cyclic logic suggests the following iterative algorithmic template

1. Guess at some initial trajectory. This guess may come from previous policies optimized earlier that we're currently executing and simultaneously re-optimizing. Or we may be optimizing for the first time, in which case, we can only really guess heuristically, or simply guess that we apply no action at all. There's no real science to this initialization phase, and different heuristics work better for different problems. Typically this phase is swept under the rug under the moniker of systems engineering, and depending on the problem complexity, it could range from difficult to trivial.

2. Using the current state-action trajectory, take a second-order Taylor expansion of the problem.

3. Optimize the second-order Taylor expansion to find a better optimal state-action trajectory. Go back to stage 2 and repeat until convergence.

3.2 How do we form and optimize the quadratic approximation?

So far we've argued that quadratic approximations are our end goal, but why should we believe that we can solve these quadratic approximations efficiently? In most interesting robots there are (roughly) anywhere between 7 and 30 action variables $\boldsymbol{u}_t$ at each time step (the torques applied to the joints, for instance), approximately double that number of state-variables since there are both configurations and velocities to represent, and there can be upward to maybe 100 time steps for effective discrete approximations. That, as a baseline, is 2100 to 9000 variables, and maybe even more. The Hessian of the system naïvely, then, is a huge matrix on the order of 5000 by 5000 dimensional. Solving such a dense system to optimize the quadratic can be done, but it's rather slow.

Fortunately, we can leverage the structure of the problem to construct the second-order approximation backward from the final time step while simultaneously solving it. As we'll see, exploiting this structure significantly reduces the computational complexity solving the system.

The full dynamically constrained problem (without extra constraints) takes the form[2]

$$\begin{aligned} &\min_{\boldsymbol{u}_{1:T}} \sum_{t=1}^{T} c_t(\boldsymbol{s}_t, \boldsymbol{u}_t) + \psi(\boldsymbol{s}_{T+1}) \\ &\text{s.t.} \quad \boldsymbol{s}_{t+1} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{u}_t) \quad \text{for all } t. \end{aligned}$$

We'll first describe the optimal solution in terms of generic argmin operations, and then discuss the nice properties that result when we can compute those argmins analytically, in particular when we use quadratic approximations.

[2]The states $\boldsymbol{s}_t$ are fully determined by the action sequence, so we denote the optimization to be over actions only.

Consider first the final potential $\psi(\boldsymbol{s}_{T+1})$. That final state is linked to the second-to-last state and action through the dynamics function $\boldsymbol{s}_{T+1} = \boldsymbol{f}(\boldsymbol{s}_T, \boldsymbol{u}_T)$. Substituting that dynamics function into the potential gives $\psi(\boldsymbol{f}(\boldsymbol{s}_T, \boldsymbol{u}_T))$, which is a function that looks one step into the future to tell us how costly it will be to end up where taking action $\boldsymbol{u}_T$ from state $\boldsymbol{s}_T$ lands. There's also a cost explicitly on taking that action from that state $c_T(\boldsymbol{s}_T, \boldsymbol{u}_T)$. That's the only other place in the objective where $\boldsymbol{u}_T$ appears, though, so to optimize over $\boldsymbol{u}_T$, we only need to account for ψ and c_T. In doing so, we define

$$V_T(\boldsymbol{s}_T) = \min_{\boldsymbol{u}_T} \left\{ c_T(\boldsymbol{s}_T, \boldsymbol{u}_T) + \psi(\boldsymbol{f}(\boldsymbol{s}_T, \boldsymbol{u}_T)) \right\}$$

as the summary of the optimization process after optimizing out $\boldsymbol{u}_T$. Using this function, the objective reduces slightly to

$$\begin{aligned} \min_{\boldsymbol{u}_{1:(T-1)}} & \sum_{t=1}^{T-1} c_t(\boldsymbol{s}_t, \boldsymbol{u}_t) + V(\boldsymbol{s}_T) \\ \text{s.t.} \quad & \boldsymbol{s}_{t+1} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{u}_t) \quad \text{for all } t. \end{aligned}$$

Now there are only $T-1$ intermediate cost terms along with a new terminal cost term $V(\boldsymbol{s}_T)$, but other than that this objective retains the same structure as the original objective. Thus, we can recursively continue this line of reasoning to incrementally collapse each of the intermediate cost terms one-by-one into new functions $V_t(\boldsymbol{s}_t)$:

$$V_t(\boldsymbol{s}_t) = \min_{\boldsymbol{u}_t} \underbrace{\left\{ c_t(\boldsymbol{s}_t, \boldsymbol{u}_t) + V_{t+t}\left(f(\boldsymbol{h}_t, \boldsymbol{u}_t)\right) \right\}}_{Q_t(\boldsymbol{s}_t, \boldsymbol{u}_t)}. \tag{12}$$

Note that if we define $V_{T+1}(\boldsymbol{s}_{T+1}) = \psi(\boldsymbol{s}_{T+1})$, this equation holds recursively for all time $t = 1, \ldots, T$. It's common, especially in machine learning, to refer to the unminimized intermediate cost plus value function as the *Q-function* $Q_T(\boldsymbol{s}_t, \boldsymbol{u}_t)$ as indicated in the equation.

Equation 12 is important enough in Optimal Control to have its own name: The Bellman Equation Bertsekas (2000). It characterizes the optimal solution through these *cost-to-go* functions $V_t(\boldsymbol{s}_t)$. Each cost-to-go function represents the cumulative cost that an optimal policy would accrue running that optimal policy until time $T+1$ starting from state $\boldsymbol{s}_t$ at time t. Under this observation the role of the terminal potential function ψ becomes clear when it's used as a computational proxy for a process that in reality continues beyond the artificial time horizon $T+1$. It acts as an estimate of the optimal cost-to-go function approximating the cumulative cost of taking optimal actions from then on out to the end of time.

However, we have left the computational or analytical process by which we solve the recursive inner loop minimizations in Equation 12 unspecified. In general, for nonlinear systems and nonlinear objective terms, those minimizations are hard. By itself, the Bellman equation only tells us something about

the structure of the problem. But suppose every time we compute one of the Q-functions $Q_t(\boldsymbol{s}_t, \boldsymbol{u}_t)$, we subsequently approximated it using a second-order Taylor expansion, denoted $\widetilde{Q}_t(\boldsymbol{s}_t, \boldsymbol{u}_t)$, around the particular state-action pair $(\boldsymbol{s}_t, \boldsymbol{u}_t)$ from the current nominal trajectory. (Since we need to optimize $\widetilde{Q}$, we may have to compute the best fit *positive definite* quadratic approximation rather than a strict Taylor expansion, depending on the definiteness of the Hessian. See Section 4.1 for a brief description of this issue in the context of Newton's method.)

Now that we have a quadratic function $\widetilde{Q}_t(\boldsymbol{s}_t, \boldsymbol{u}_t)$ at our disposal, we can solve for the argmin over $\boldsymbol{u}_t$ as a function of the state $\boldsymbol{s}_t$. Rather than reviewing the tedious calculation that derives the result we simply denote it as

$$\boldsymbol{u}_t^*(\boldsymbol{s}_t) = \underset{\boldsymbol{u}_t}{\operatorname{argmin}}\, \widetilde{Q}_t(\boldsymbol{s}_t, \boldsymbol{u}_t). \tag{13}$$

When $\widetilde{Q}$ is quadratic in both arguments, this function $\boldsymbol{u}_t^*(\boldsymbol{s}_t)$ is simply an affine transformation of $\boldsymbol{s}_t$. Thus, plugging it back into the quadratic tells us that the cost-to-go function is also quadratic

$$V_t(\boldsymbol{s}_t) = \widetilde{Q}(\boldsymbol{s}_t, \boldsymbol{u}_t^*(\boldsymbol{s}_t)), \tag{14}$$

which means that information over future variables entirely collapses down into a single compact quadratic function over just the state. This algorithm for recursively backing up second-order information along the trajectory using second-order Taylor approximations of the Q-functions, in conjunction with the iterative recalculations of the nominal trajectory around which these approximations are made (see Section 3.1), is known as Differential Dynamic Programming Todorov (2006); Stengel (1994); Bertsekas (2000).

Importantly, since each analytical solution $\boldsymbol{u}_t^*(\boldsymbol{s}_t)$ is an affine function of the state, it's a valid approximation for the optimal action to take from *any* state $\boldsymbol{s}_t$ anywhere within the region of validity of the tth Taylor approximation. In other words, it acts as an effective *feedback control law* within that region of validity providing robustness to noise and system perturbation.

3.3 Do we have to calculate the full Taylor expansion?

The Differential Dynamic Programming iterative quadratic approximation technique is only one way to calculate a quadratic approximation of the nonlinear problem around the current nominal trajectory. That approximation requires accounting for second-order information about the dynamics, and that can be slow to compute, especially when using finite-differencing Todorov et al. (2012). More recently a slightly simpler approximation has become popular Todorov (2006); Li & Todorov (2004); Todorov & Li (2005) that theoretically retains similar convergence and convergence rate guarantees and empirically tends to even find solutions faster in some situations (which comes from a combination of a cheaper approximation and an empirical observation that it can converge in fewer iterations).

This approximation, known as the Linear Quadratic approximation, makes linear approximations of the dynamics function $\boldsymbol{s}_{t+1} = \boldsymbol{f}(\boldsymbol{s}_t, \boldsymbol{u}_t)$ around the nominal trajectory and quadratic approximations of intermediate cost functions and the terminal cost. Using these constituent approximations, the resulting dynamics are linear and the costs are quadratic giving

$$\begin{aligned}
&\boldsymbol{s}_{t+1} = \boldsymbol{A}_t \boldsymbol{s}_t + \boldsymbol{B}_t \boldsymbol{u}_t \\
&c_t(\boldsymbol{s}_t, \boldsymbol{u}_t) \approx \frac{1}{2} \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{u}_t \end{bmatrix}^T \boldsymbol{C}_t \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{u}_t \end{bmatrix} - \boldsymbol{b}_t^T \begin{bmatrix} \boldsymbol{s}_t \\ \boldsymbol{u}_t \end{bmatrix} + c_t, \\
&\psi(\boldsymbol{s}_{T+1}) \approx \frac{1}{2} \boldsymbol{s}_{T+1}^T \boldsymbol{G} \boldsymbol{s}_{T+1} - \boldsymbol{d}^T \boldsymbol{s}_{T+1} + l,
\end{aligned}$$

for each time step t, where the parameters $\boldsymbol{A}_t, \boldsymbol{B}_t, \boldsymbol{C}_t, \boldsymbol{b}_t, c_t, \boldsymbol{G}, \boldsymbol{d}, l$ all come from the respective Taylor expansions.

Since the approximate terminal potential is quadratic, substituting the linear dynamics equation in as $\boldsymbol{s}_{T+1}$ automatically gives a quadratic function for the last time step's Q-function $Q(\boldsymbol{s}_T, \boldsymbol{u}_T)$. As discussed above, that quadratic Q-function collapses to a quadratic function $V_T(\boldsymbol{s}_T)$ over only $\boldsymbol{s}_T$ when we optimize over the action variable $\boldsymbol{u}_T$. Thus, we can apply that argument recursively from there to show that for all time the cost-to-go functions $V_t(\boldsymbol{s}_t)$ of this Linear Quadratic approximation are all, themselves, quadratic. No further approximation is necessary, and every minimization step is just a linear algebraic manipulation as we back up optimal cost-to-go information from the last time step to the first. This algorithm, since each optimal feedback control policy $\boldsymbol{u}_t^*(\boldsymbol{s}_t)$ is classically known as the Linear Quadratic Regulator (LQR) of the Linear Quadratic system, is known as the Iterative Linear Quadratic Regulator (iLQR) Li & Todorov (2004).

The next section explores an algorithm for nonlinear unconstrained optimization algorithm called Newton's method, which has a very similar flavor to the above iterative quadratic approximation schemes for unconstrained nonlinear optimal control. Indeed, Section 5 formalizes this connection and shows that both DDP and iLQR may be viewed as simply efficiently solved implementations of Newtons method applied to the unconstrained objective with particularly defined Hessian approximations. Anticipating that connection, Section 4.2 discusses a very efficient form of general constrained nonlinear optimization algorithm that iteratively constructs unconstrained approximations and solves them using Newton's method. Section 5 additionally discusses applications of this constrained algorithm to handle more generally constrained optimal control problems.

4 A segue into more general nonlinear optimization

We'll see in Section 5 that the above iterative algorithms for nonlinear optimal control are simply applications of traditional methods for nonlinear opti-

mization to the optimal control problem. This section explores some specific high-performance algorithms for both unconstrained and constrained nonlinear optimization.

4.1 Newton's method

We can analytically optimize quadratic functions with symmetric positive definite Hessians. Suppose $Q(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x} + c$ where $\boldsymbol{x}, \boldsymbol{b} \in \mathbb{R}^n$, $\boldsymbol{A} \in \mathbb{R}^{n^2}$, and $c \in \mathbb{R}$. Then if $\boldsymbol{A}$ is symmetric positive definite, then the global optimum is where the gradient vanishes

$$\begin{aligned} \nabla Q(\boldsymbol{x}^*) &= \boldsymbol{A}\boldsymbol{x}^* - \boldsymbol{b} = 0 \\ \Rightarrow \quad \boldsymbol{x}^* &= \boldsymbol{A}^{-1}\boldsymbol{b}. \end{aligned}$$

General smooth functions are more difficult to optimize. Around any point $\boldsymbol{x}_0$ we can always find the best fit quadratic approximation using the second-order Taylor expansion. And we can solve that approximation analytically so long as its Hessian is symmetric positive definite. But that latter criterion doesn't always hold—since the function need not be convex, there may be local regions where the function is fully negative definite (curved downward everywhere) or other areas where the function is positive definite (curved upward) in some dimensions but negative definite (curved downward) in other dimensions (a saddle point). More generally, the local curvature, even if it's positive definite (curved up in all directions), may not accurately reflect the shape of the function closer to the minimum. Constructing a quadratic approximation based only on the Hessian may be misleading. Thus, an optimization algorithm that leverages quadratic approximations would have to iteratively calculate the best (or a good) *symmetric positive definite* quadratic approximation in a way that doesn't overfit to the local shape, and only then solve it, and iterate from the new point.

And indeed, this is the crux of Newton's method and its variants: find a good, trustworthy, symmetric positive definite quadratic approximation to the function around where we currently are, solve it, and iterate. Different variants of this algorithm are defined by how we correct the quadratic approximation to ensure that it's trustworthy and symmetric positive definite, and how we search in the direction suggestion by the approximation's minimizer. Techniques for ensuring that the approximation is both trustworthy and symmetric positive definite are arguably the most interesting of these issues, so we explore them in slightly more detail and refer to Nocedal & Wright (2006) for further discussion around these and other issues.

Suppose our objective function is $f : \mathbb{R}^n \to \mathbb{R}$ and our current (ith) iterate is $\boldsymbol{x}_i \in \mathbb{R}^n$. Then we denote this best fit trustworthy positive definite quadratic approximation by

$$\begin{aligned} f(\boldsymbol{x}) &\approx \widetilde{f}_{\boldsymbol{x}_i}(\boldsymbol{x}) \qquad (15) \\ &= f(\boldsymbol{x}_i) + \nabla f(\boldsymbol{x}_i)^T(\boldsymbol{x} - \boldsymbol{x}_i) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}_i)^T\boldsymbol{H}_i(\boldsymbol{x} - \boldsymbol{x}_i), \end{aligned}$$

where $\boldsymbol{H}_i$ approximates the Hessian $\nabla^2 f(\boldsymbol{x}_i)$ in a way that ensures that it's positive definite (Hessians are always symmetric for the functions we're considering). To calculate $\boldsymbol{H}_i$, we may start immediately with an approximation to the Hessian or we may start from the exact Hessian. Denote that (potentially approximate) Hessian by $\widetilde{\nabla}^2 f(\boldsymbol{x}_i)$. If the Hessian has some negative Eigenvalues, the approximation may add some positive constant to all Eigenvalues to ensure their positivity, or it may first truncate those negative Eigenvalues to 0 before doing that. We'll denote this *fixed* Hessian by $\widetilde{\nabla}^2_+ f(\boldsymbol{x}_i)$. Finally, to ensure that the quadratic approximation is trustworthy, we may add a term $\frac{\lambda}{2}\|\boldsymbol{x} - \boldsymbol{x}_i\|^2$ that pulls it back to the previous iterate $\boldsymbol{x}_i$. This term has the effect of adding $\lambda \boldsymbol{I}$ to the Hessian of the quadratic. Thus, with this notation, our trustworthy positive definite Hessian takes the form $\boldsymbol{H}_i = \widetilde{\nabla}^2_+ f(\boldsymbol{x}_i) + \lambda \boldsymbol{I}$.

The bottom line is that there are a number of techniques for calculating variants of $\boldsymbol{H}_i$, but the main feature of this matrix is that it starts from an approximation to the Hessian of the objective, and then is modified to ensure that we aren't overfitting to the local second-order shape of the function should that curvature not accurately reflect the shape closer to the minimum. Those modifications stem from both ensuring that the matrix is positive definite and that it's trustworthy by effectively constraining how far the minimizer travels from the current iterate.

Finally, solving this best fit symmetric positive definite approximation gives an update rule of the form

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \alpha_t \boldsymbol{H}_i^{-1} \nabla f(\boldsymbol{x}_i), \tag{16}$$

where $\alpha_t > 0$ is a step size chosen by any number of techniques (e.g. fixed sequence, line search, etc.). It's beyond the scope of this document to dig deeper into the details of line searches and Hessian modifications, but see Nocedal & Wright (2006) for a thorough discussion of some of the subtleties around Newton's method and unconstrained optimization.

4.2 Augmented Lagrangian

Now suppose we have constraint functions as well. Mathematically, our abstract optimization now becomes

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &\text{s.t. } \boldsymbol{g}(\boldsymbol{x}) \geq 0 \\ &\quad\ \ \boldsymbol{h}(\boldsymbol{x}) = 0, \end{aligned} \tag{17}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function as before, $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^k$ defines a set of inequality constraints, and $\boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^l$ defines a set of l equality constraints.

In this section, we present an equality and inequality constrained Augmented Lagrangian algorithm for solving the general constrained nonlinear optimization problem given in Equation 17.

The Lagrangian of the above problem is

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\gamma}) = f(\boldsymbol{x}) - \boldsymbol{\lambda}^T \boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{\gamma}^T \boldsymbol{h}(\boldsymbol{x}). \tag{18}$$

It's beyond the scope of this document to fully review the first-order optimality conditions of this constrained optimization problem (see Nocedal & Wright (2006) for a full development), but Appendix C briefly reviews the role of Lagrange Multipliers. For our purposes, the most important of these conditions are that 1. all of the constraints must be satisfied, and 2. at the minimizer,

$$\nabla_{\boldsymbol{x}} \mathcal{L} = \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) - \sum_{i=1}^{k} \lambda_i \nabla_{\boldsymbol{x}} g_i(\boldsymbol{x}) - \sum_{j=1}^{l} \gamma_j \nabla_{\boldsymbol{x}} h_j(\boldsymbol{x}) = 0 \tag{19}$$

$$\text{with} \quad \lambda_i \geq 0 \quad \text{for all } i = 1, \ldots, k.$$

$\boldsymbol{\lambda}$ and $\boldsymbol{\gamma}$ are vectors of Lagrange Multipliers corresponding to inequality and equality constraints, respectively. Note that the positivity constraints are imposed only on $\boldsymbol{\lambda}$ because they're associated with inequality constraints (see Appendix C).

A natural approach to optimizing f while simultaneously satisfying the constraints is to place penalties on constraint violations and then to optimize instead the sum of f plus the constraint penalties. Specifically, for equality constraints $\boldsymbol{h}(\boldsymbol{x}) = 0$, any deviation of $\boldsymbol{h}$ from zero is a constraint violation, so we can measure the size of these violations using simply a squared norm penalty $\|\boldsymbol{h}(\boldsymbol{x})\|^2$. And since only negative deviations from $\boldsymbol{g}(\boldsymbol{x}) = 0$ are violations of the inequality constraints, the inequality penalty can simply be a one-sided version of the squared penalty

$$\|\boldsymbol{g}(\boldsymbol{x})\|_-^2 = \sum_{i=1}^{k} s(g_i(\boldsymbol{x}))\ g_i(\boldsymbol{x})^2, \tag{20}$$

where $s(\alpha)$ is 1 when α is negative and zero otherwise. Figure 3 shows a graph of this one-sided penalty.

This basic penalty approach would optimize a surrogate objective of the form

$$\psi(\boldsymbol{x}) = f(\boldsymbol{x}) \ + \ \frac{\mu}{2} \|\boldsymbol{g}(\boldsymbol{x})\|_-^2 \ + \ \frac{\eta}{2} \|\boldsymbol{h}(\boldsymbol{x})\|^2, \tag{21}$$

where μ and η are positive penalty scalars. Unfortunately, for any finite setting of μ and η the constraints will never be fully satisfied because the penalties are always fighting with the objective (the constraints are only fully satisfied when the penalties are exactly zero). The only way to fully satisfy the constraints is to incrementally increase μ and η until they're strong enough that the pull of the function f is negligible. In practice, these large scalars induce poor conditioning and numerical instability, so with this naïve approach we will always have potentially non-negligible constraint violations.

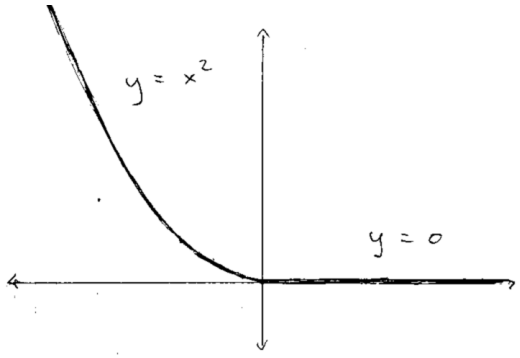

Figure 3: One-dimensional rendering of a one-sided quadratic penalty.

The main problem with this penalty method is that the zero point of the penalty function is precisely where the constraint surface is. We can never make that penalty go entirely to zero because it's battling with the objective function, so solutions to the penalized problem never fully satisfy the constraints. The Augmented Lagrangian algorithm addresses this issue by modifying where that zero point lies in reaction to how the objective is pulling the optimizer into constraint violations.

Intuitively, if $\boldsymbol{x}^*$ is the minimizer of the penalized objective given in Equation 21 and $\boldsymbol{x}^*$ currently violates the jth equality constraint ($h_i(\boldsymbol{x}^*) \leq 0$), we need to modify the penalty so that it pulls harder on negative values of h_i. Thus, we move the zero point of the penalty to be a little bit positive by modifying it to be $(v_j - h_j(\boldsymbol{x}))^2$ where $v_j > 0$. This modified penalty function is minimized at the positive value v_j, so it automatically pulls a little harder on h_j (without increasing the scalar penalty parameter η at all) to coax it toward the constraint boundary. Importantly, we can come up with a very effective rule for updating the penalty offsets by analyzing the first-order optimality conditions of penalty function and comparing them to the optimality conditions of the original problem.

The (offset) penalty method optimizes a surrogate objective of the following form called the Augmented Lagrangian:

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\gamma}) = f(\boldsymbol{x}) + \frac{\mu}{2}\|\frac{\boldsymbol{\lambda}}{\mu} - \boldsymbol{g}(\boldsymbol{x})\|_{-}^2 \; + \frac{\eta}{2}\|\frac{\boldsymbol{\gamma}}{\eta} - \boldsymbol{h}(\boldsymbol{x})\|^2. \tag{22}$$

Here we've written it as a penalized objective to emphasize the connection to penalty methods, but as its name suggests, it's actually closely related to the Lagrangian function of the original problem. Multiplying out the penalty functions and removing the constant terms reveals that it's equivalently constructed by simply adding the naïve zero-centered penalties to the original Lagrangian

function (and hence it's an *augmented* Lagrangian):

$$\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\gamma}) = f(\boldsymbol{x}) - \boldsymbol{\lambda}^T \boldsymbol{g}(\boldsymbol{x}) - \boldsymbol{\gamma}^T \boldsymbol{h}(\boldsymbol{x}) + \left(\frac{\mu}{2} \|\boldsymbol{g}(\boldsymbol{x})\|_-^2 + \frac{\eta}{2} \|\boldsymbol{h}(\boldsymbol{x})\|^2 \right). \quad (23)$$

Since Equation 22 is an unconstrained objective, the first-order optimality conditions are simply a statement that it's gradient is zero at the optimizer. Thus, once we optimize this surrogate function $\mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\gamma})$ over $\boldsymbol{x}$ using, for instance, a Newton method, the following condition should hold

$$\nabla_x \mathcal{L}(\boldsymbol{x}, \boldsymbol{\lambda}, \boldsymbol{\gamma}) = \nabla f(\boldsymbol{x}) - \underbrace{(\boldsymbol{\lambda} - \mu \boldsymbol{g}(\boldsymbol{x}))}_{\boldsymbol{\lambda}\leftarrow} \nabla \boldsymbol{g}(\boldsymbol{x}) - \underbrace{(\boldsymbol{\gamma} - \eta \boldsymbol{h}(\boldsymbol{x}))}_{\boldsymbol{\gamma}\leftarrow} \nabla \boldsymbol{h}(\boldsymbol{x}).$$

As we've indicated in the equation, referring back to the first-order optimality conditions of the original problem in Equation 19 we see that they're exactly the same, except with the substitutions

$$\begin{aligned} \boldsymbol{\lambda} &\leftarrow \boldsymbol{\lambda} - \mu \boldsymbol{g}(\boldsymbol{x}) \\ \boldsymbol{\gamma} &\leftarrow \boldsymbol{\gamma} - \eta \boldsymbol{h}(\boldsymbol{x}). \end{aligned} \quad (24)$$

Thus, if we use these equations as update rules for the Lagrange Multipliers, after just a single iteration, the first-order optimality condition given by Equation 19 will be satisfied. (Note that we need to modify the update rule for $\boldsymbol{\lambda}$ to ensure that the components are never negative. We can do this using $\lambda_i = \max\{0, \lambda_i - \mu g_i(\boldsymbol{x})\}$.) There are more optimality conditions, so we're not done, but this is a very powerful observation and it forms the foundation of the Augmented Lagrangian algorithm.

Note that if $h_j(\boldsymbol{x})$ is negative, this update will try to increase the zero point $\frac{\gamma}{\eta}$ of the jth penalty. And vice versa, if $h_j(\boldsymbol{x})$ is positive, it'll try to decrease that zero point. Thus, the update always attempts to modify the penalty in a way that pulls the current optimizer toward the equality constraint. This process only converges when $\boldsymbol{h}(\boldsymbol{x}) = 0$, which means that all equality constraints are satisfied. Likewise, for the inequality constraints we have a similar scenario. However, in this case, when $g_i(\boldsymbol{x})$ is already positive (i.e. the constraint is satisfied) the update will decrease that component down to 0 and then the process will stop (for that constraint).

Finally, as motivated by the derivation of the update rule in Equation 24, the variables $\boldsymbol{\lambda}$ and $\boldsymbol{\gamma}$ converge to good estimates of the original problem's Lagrange Multipliers.

5 Nonlinear optimal control algorithms are applications of optimization methods

The algorithms of Section 4 are generic. Interestingly, though, we see now that the algorithms DDP and iLQR described in Section 3 are more concisely described simply as applications of Newton's method to the optimal control problem. The specifics of how each variant constructs its approximate subproblem

are simply different ways of calculating the quadratic approximation. Moreover, practical implementations of both DDP and iLQR need to leverage tricks from the generic Newton's method toolbox to correct negative definite Hessians (i.e. to find the best fit *positive definite* approximation), ensure trustworthiness, and choose step sizes appropriately. These tricks modify the approximation and weaken our original analogy that these quadratic subproblems were just quadratic (best) approximations to the original control problem. The better description is simply that these algorithms are applications of Newton's method that exploit structure in the Hessian to efficiently solve the Newton step linear system at each iteration.

This observation suggests additionally that constrained optimal control problems can be solved using the constrained cousin of Newton's method: the Augmented Lagrangian algorithm. Indeed, empirically the algorithm seems to be very effective Toussaint (2014), although many recent explorations of constrained optimal control in the literature have used a competing solution technique called Sequential Quadratic Programming Posa et al. (2014). It is not currently clear which algorithm is superior (or whether either algorithm is uniformly better than the other in all cases) for this class of structured optimization problem.

A Derivatives of distance potential

Consider the function $\psi(\boldsymbol{x}) = \|\boldsymbol{x}\|$. We can calculate the gradient and Hessian of the function (away from the origin where it's undefined) by writing it as $\psi(\boldsymbol{x}) = (\boldsymbol{x}^T\boldsymbol{x})^{\frac{1}{2}}$. For these calculations, it's easy to forget the specifics of transposes and what not in the rules for calculating derivatives of multivariate functions, so it's often easiest to simply calculate the individual partial derivatives of the expression in terms of generic indices and then look to see if we can recognize their compact vector and matrix forms.

First the gradient:

$$\begin{aligned} \frac{\partial}{\partial x_i}(\boldsymbol{x}^T\boldsymbol{x})^{\frac{1}{2}} &= \frac{1}{2}\left(\boldsymbol{x}^T\boldsymbol{x}\right)^{-\frac{1}{2}} 2x_i &= \frac{x_i}{\|\boldsymbol{x}\|} \\ \Rightarrow \nabla\|\boldsymbol{x}\| &= \widehat{\boldsymbol{x}}, \end{aligned}$$

where $\widehat{\boldsymbol{x}} = \boldsymbol{x}/\|\boldsymbol{x}\|$.

Then the Hessian is:

$$\begin{aligned} \frac{\partial^2}{\partial x_i \partial x_j}(\boldsymbol{x}^T\boldsymbol{x})^{\frac{1}{2}} &= \frac{\partial}{\partial x_j}\left(\frac{\partial}{\partial x_i}(\boldsymbol{x}^T\boldsymbol{x})^{\frac{1}{2}}\right) &= \frac{\partial}{\partial x_j}\frac{x_i}{\|\boldsymbol{x}\|} \\ = \frac{\|\boldsymbol{x}\|\frac{\partial x_i}{\partial x_j} - x_i\frac{\partial}{\partial x_j}\|\boldsymbol{x}\|}{\|\boldsymbol{x}\|^2} &= \frac{1}{\|\boldsymbol{x}\|}\left(\frac{\partial x_i}{\partial x_j} - \frac{x_i}{\|\boldsymbol{x}\|}\frac{x_j}{\|\boldsymbol{x}\|}\right) \\ \Rightarrow \quad \nabla^2\|\boldsymbol{x}\| &= \frac{1}{\|\boldsymbol{x}\|}\left(\boldsymbol{I} - \widehat{\boldsymbol{x}}\widehat{\boldsymbol{x}}^T\right). \end{aligned}$$

B Projecting orthogonally to a given direction.

For any unit vector $\widehat{\boldsymbol{x}}$, the matrix $\boldsymbol{P}_\perp = \boldsymbol{I} - \widehat{\boldsymbol{x}}\widehat{\boldsymbol{x}}^T$ projects a vector orthogonally to the direction indicated by $\widehat{\boldsymbol{x}}$ as depicted in Figure 4. Why is this?

First, let's examine what multiplication by $\boldsymbol{P}_\perp$ entails. Suppose we multiply a vector $\boldsymbol{v}$ by $\boldsymbol{P}_\perp$. We get

$$\left(\boldsymbol{I} - \widehat{\boldsymbol{x}}\widehat{\boldsymbol{x}}^T\right)\boldsymbol{v} = \boldsymbol{v} - \left(\widehat{\boldsymbol{x}}\widehat{\boldsymbol{x}}^T\right)\boldsymbol{v} = \boldsymbol{v} - \left(\widehat{\boldsymbol{x}}^T\boldsymbol{v}\right)\widehat{\boldsymbol{x}}. \tag{25}$$

The inner product in parentheses in that final expression is $\widehat{\boldsymbol{x}}^T\boldsymbol{v} = \|\boldsymbol{v}\|\cos\theta$, where θ is the angle between the two vectors $\widehat{\boldsymbol{x}}$ and $\boldsymbol{v}$. As indicated in Figure 4, this quantity is the length of the projection of $\boldsymbol{v}$ onto $\widehat{\boldsymbol{x}}$. Thus, multiplying that length by the unit vector $\widehat{\boldsymbol{x}}$, itself, gives the full vector which is the projection of $\boldsymbol{v}$ onto $\widehat{\boldsymbol{x}}$. One may also interpret this projection as simply the component of $\boldsymbol{v}$ in the direction $\widehat{\boldsymbol{x}}$.

Therefore, if $(\widehat{\boldsymbol{x}}^T\boldsymbol{v})\widehat{\boldsymbol{x}}$ is the component of $\boldsymbol{v}$ along the direction $\widehat{\boldsymbol{x}}$, subtracting that component from $\boldsymbol{v}$ gives the component of $\boldsymbol{v}$ orthogonal to the direction $\widehat{\boldsymbol{x}}$. In other words, multiplying a vector $\boldsymbol{v}$ by the matrix $\boldsymbol{P}_\perp$ gives the component of $\boldsymbol{v}$ orthogonal to $\widehat{\boldsymbol{x}}$. Or, viewed as a projection, the matrix projects the vector $\boldsymbol{v}$ orthogonally to $\widehat{\boldsymbol{x}}$.

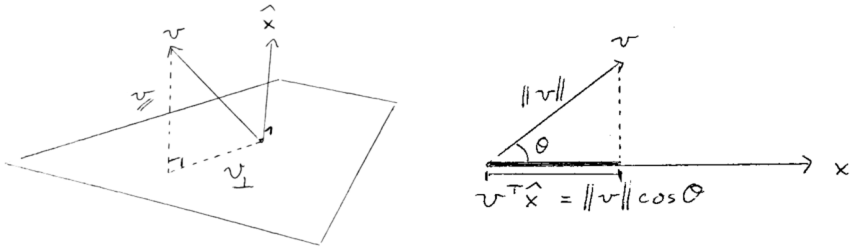

Figure 4: **Left:** Depiction of a projection orthogonal to a vector $\widehat{\boldsymbol{x}}$. $\boldsymbol{v}_{/\!/}$ is the component of $\boldsymbol{v}$ parallel to $\widehat{\boldsymbol{x}}$, and $\boldsymbol{v}_\perp$ is the component orthogonal to it. $\boldsymbol{v}_\perp$ lives in an entire space of orthogonal vectors, depicted here as a plane orthogonal to $\widehat{\boldsymbol{x}}$. **Right:** The inner product has the relation $\boldsymbol{v}^T\boldsymbol{x} = \|\boldsymbol{v}\|\|\boldsymbol{x}\|\cos\theta$, where θ is the angle between the two vectors. Thus $\boldsymbol{v}^T\widehat{x}$ is the length of the projection of $\boldsymbol{v}$ onto $\boldsymbol{x}$ (where $\widehat{\boldsymbol{x}} = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}$).

C A brief review of Lagrange multipliers

General constrained nonlinear optimization problems take the form

$$\begin{aligned} &\min_{\boldsymbol{x}} f(\boldsymbol{x}) \\ &\text{s.t. } \boldsymbol{g}(\boldsymbol{x}) \geq 0 \\ &\quad\ \boldsymbol{h}(\boldsymbol{x}) = 0, \end{aligned} \tag{26}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is the objective, $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^k$ is a collection of k inequality constraint functions wrapped up into a single vector-valued function, and $\boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^l$ is a collection of l equality constraint functions wrapped up into a single vector-valued function. We assume all functions are at least second-order differentiable.

This section covers some intuition behind Lagrange multipliers. This isn't a complete development, but it should help in understanding the above material. See Nocedal & Wright (2006) for a more complete discussion.

First consider only equality constraints. Note that for each equality constraint $h_j(\boldsymbol{x})$, the equation $h_j(\boldsymbol{x}) = 0$ defines a surface of dimension $n-1$ on which the optimal solution must lie. Figure 5 depicts this surface pictorially for a 3-dimensional function. Importantly, at an optimal point of the optimization problem in Equation 26 must be a point for which the gradient of the objective is orthogonal to the surface. If it isn't orthogonal to the surface, then there is some component of the negative gradient tangent to the surface, which means that we can move a small amount ϵ in that direction along the surface and decrease the objective value slightly. If we're at a local optimum, that would be a contradiction, so that can't happen. The gradient of the objective is orthogonal to the surface.

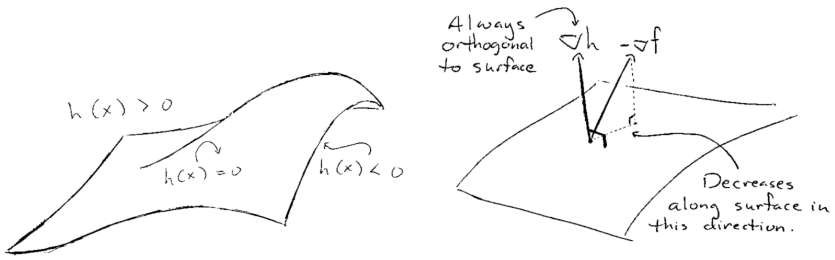

Figure 5: **Left:** Representation of a constraint surface as the zero set of a function $h(\boldsymbol{x})$. On one side of the surface $h(\boldsymbol{x})$ is strictly positive, and on the other side of the surface $h(\boldsymbol{x})$ is strictly negative. Exactly on the surface it's zero. **Right:** Example of why the objective gradient ∇f must be orthogonal to the constraint surface at the optimizer. Otherwise, there is a distinct direction parallel to the surface along the function decreases.

Note that the gradient of the constraint function itself $\nabla h_j(\boldsymbol{x})$ is already orthogonal to the function's zero set. If it weren't, again there would be a component of the gradient tangent to the zero-set's surface which means the constraint function's value would change slightly as we move in the direction of that component. But the zero set is a set for which the function value never changes (it's always zero), so it must be that that doesn't happen. The only conclusion is that the gradient $\nabla h_j(\boldsymbol{x})$ must be orthogonal to this zero set $h_j(\boldsymbol{x}) = 0$.

Generalizing this idea slightly, we can argue similarly that all rows of the Jacobian of the vector valued constraint function $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ must be orthogonal to the combined zero set $\boldsymbol{h}(\boldsymbol{x}) = 0$ since each row of that Jacobian is the transpose of the gradient of one of the constituent h_j functions:

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} = \begin{bmatrix} \nabla h_1(\boldsymbol{x})^T \\ \nabla h_2(\boldsymbol{x})^T \\ \vdots \\ \nabla h_l(\boldsymbol{x})^T \end{bmatrix}. \tag{27}$$

Since the zero set must be a set for which any tangent vector is orthogonal to all gradients $\nabla h_j(\boldsymbol{x})$, that tangent space must be the null space of $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$. In other words, at a given point $\boldsymbol{x}$, the null space of the Jacobian $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$ is the space tangent to the constraint surface, and the row space of the Jacobian $\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}$, which is orthogonal to the null space, is the space of all vectors orthogonal to the constraint surface.

Understanding this combined geometric and analytic view of the constraint surface, the statement that the gradient of the objective function $\nabla f(\boldsymbol{x})$ evaluated at the local optimizers $\boldsymbol{x}^*$ must be orthogonal to the equality constraint

surface can be formalized as

$$\nabla f(\boldsymbol{x}^*) \in \mathbf{span}\left[\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}^T (\boldsymbol{x}^*)\right], \tag{28}$$

where $\mathbf{span}(\cdot)$ denotes the linear span of the columns of the given matrix. Another way of stating this span requirement is to say that there exist some coefficients $\gamma_j \in \mathbb{R}$ for $j = 1, \ldots, l$ for which

$$\nabla f = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}^T \boldsymbol{\gamma} \tag{29}$$

at $\boldsymbol{x}^*$, where $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_l)^T$. These coefficients $\boldsymbol{\gamma}$ are what we've been calling the Lagrange Multipliers.

The inequality constraints have a similar argument. But in this case, each of the constraints $g_i(\boldsymbol{x})$ may be either "on" or "off". If the objective function is pushing the optimizer into one of the inequality constraints (i.e. f would like the $g_i(\boldsymbol{x})$ to be negative, but $g_i(\boldsymbol{x})$ won't let it), then the best we can do is push g_i down to 0. In that case, $\boldsymbol{x}^*$ is going to lie exactly on the constraint surface, and we can effectively treat that constraint as though it were an equality constraint. On the other hand, if f would like $\boldsymbol{x}$ to be in a region of the space where g_i is already positive, then the constraint doesn't actually affect the problem. We could throw it away without changing the solution. It's as though the constraint never existed.

Therefore, if we only had inequality constraints, the following conditions succinctly describe these criteria

$$\begin{aligned} \nabla f &= \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{x}}^T \boldsymbol{\lambda} \\ \boldsymbol{\lambda} &\geq 0, \end{aligned} \tag{30}$$

where $\boldsymbol{\lambda} \in \mathbb{R}^k_+$ is again a vector of Lagrange multipliers.

Combining these arguments, in the general case where we have both equality and inequality constraints, at the optimizer the gradient of f must lie in the span of gradients of the equality constraints and the gradients of the *active* inequality constraints. Combining all of these criteria into a single list of equations gives the Karush-Kuhn-Tucker (KKT) conditions, which form the first-order optimality conditions for the problem: at the optimizer $\boldsymbol{x}^*$, it must be that

$$\begin{aligned} &\nabla f(\boldsymbol{x}^*) - \sum_{i=1}^{k} \lambda_i \nabla g_i(\boldsymbol{x}) - \sum_{j=1}^{l} \gamma_j \nabla h_j(\boldsymbol{x}) = 0 \\ &\boldsymbol{g}(\boldsymbol{x}^*) \geq 0 \quad \text{and} \quad \boldsymbol{h}(\boldsymbol{x}^*) = 0 \\ &\boldsymbol{\lambda} \geq 0 \\ &\lambda_i g_i(\boldsymbol{x}^*) = 0 \ \text{ for all } \ i = 1, \ldots, k. \end{aligned}$$

That last equation is known as the *complementarity condition*. It states mathematically that if λ_i is strictly positive, then the ith inequality constraint must be active: $g_i(\boldsymbol{x}^*) = 0$. On the other hand, if the ith constraint is inactive $g_i(\boldsymbol{x}^*) > 0$, then the corresponding Lagrange multiplier must be zero.

References

Bertsekas, D. *Dynamic Programming and Optimal Control*. Athena Scientific, Belmont, MA, 2nd edition, 2000.

Li, W. and Todorov, E. Iterative linear-quadratic regulator design for nonlinear biological movement systems. In *In proceedings of the 1st International Conference on Informatics in Control, Automation and Robotics*, volume 1, pp. 222–229, 2004.

Nocedal, Jorge and Wright, Stephen. *Numerical Optimization*. Springer, 2006.

Posa, Michael, Cantu, Cecilia, and Tedrake, Russ. A direct method for trajectory optimization of rigid bodies through contact. *International Journal of Robotics Research*, 33(1):69–81, January 2014.

Stengel, R. *Optimal Control and Estimation*. Dover, New York, 1994.

Todorov, E. Optimal control theory. *In Bayesian Brain: Probabilistic Approaches to Neural Coding*, pp. 269–298, 2006.

Todorov, E and Jordan, M. Optimal feedback control as a theory of motor coordination. *Nature Neuroscience*, 5(11):1226–1235, 2002.

Todorov, E. and Li, W. A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems. In *In proceedings of the American Control Conference*, volume 1, pp. 300–306, 2005.

Todorov, E, Erez, T, and Tassa, Y. Mujoco: A physics engine for model-based control. In *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

Toussaint, Marc. Personal communication, 2014.