CHAPTER 9

Dynamic Programming

In chapter 2, we spent some time thinking about the phase portrait of the simple pendulum, and concluded with a challenge: can we design a nonlinear controller to *reshape* the phase portrait, with a very modest amount of actuation, so that the upright fixed point becomes globally stable? With unbounded torque, feedback linearization solutions (e.g., invert gravity) can work well, but can also require an unecessarily large amount of control effort. The energy-based swing-up control solutions presented in section 3.6.2 are considerably more appealing, but required some cleverness and might not scale to more complicated systems. Here we investigate another approach to the problem, using computational optimal control to synthesize a feedback controller directly.

9.1 INTRODUCTION TO OPTIMAL CONTROL

In this chapter, we will introduce optimal control - a control design process using optimization theory. This approach is powerful for a number of reasons. First and foremost, it is very general - allowing us to specify the goal of control equally well for fully- or under-actuated, linear or nonlinear, deterministic or stochastic, and continuous or discrete systems. Second of all, this control formulation permits numerical solutions - we can harness the power of computational optimization theory to solve analytically intractable control problems. [13] is a fantastic reference on this material, and the source of a number of the examples worked out here.

The fundamental idea in optimal control is to formulate the goal of control as the *long-term* optimization of a scalar cost function. Optimal control has a long history in robotics. For instance, there has been a great deal of work on the minimum-time problem for pick-and-place robotic manipulators, and the linear quadratic regulator (LQR) and linear quadratic regulator with Gaussian noise (LQG) have become essential tools for any practicing controls engineer.

In this chapter, we will formulate the deterministic optimal feedback control problem, considering a system described by the equations:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t).$$

This structure certainly emcompasses the robots that are the focus of this book (and many other dynamical systems), but also doesn't take advantage of the structure of the manipulator equations (that topic will be discussed in chapter ?). In the general formulation, our goal is to design a feedback policy, π, where

$$\mathbf{u}(t) = \pi(\mathbf{x}, t).$$

We will define π^*, the *optimal* feedback policy, as the policy which minimizes the scalar cost function.

9.2 FINITE HORIZON PROBLEMS

Some control problems have a natural duration. For example, perhaps it is 1pm and your robot needs to get to the bank before it closes at 5pm. Or imagine your robot is working on an assembly room floor, and is allocated a fixed amount of time to complete each assembly (or pick-and-place) task. As we will see later in the chapter, other problems are more naturally described independent of the time interval, or over an infinite horizon, and we will see that the infinite horizon derivations follow naturally from the finite horizon derivations in this section.

In finite horizon problems, we define a finite time interval, $t \in [0, T]$, where T is called the horizon time. In general, T can be given, or left as a free-variable to be optimized. If we run the system from an initial condition, $\mathbf{x}_0$, executing policy π then the system will move through a trajectory $[\mathbf{x}(t), \mathbf{u}(t)]$. Using $\bar{\mathbf{x}}$ and $\bar{\mathbf{u}}$ to denote the entire continuous system trajectory over $[0, T]$, or

$$\bar{\mathbf{x}} = \{\mathbf{x}(0), ..., \mathbf{x}(T)\}, \quad \bar{\mathbf{u}} = \{\mathbf{u}(0), ..., \mathbf{u}(T)\},$$

then we can score the policy π by defining a cost function:

$$J^\pi(\mathbf{x}_0) = g(\bar{\mathbf{x}}, \bar{\mathbf{u}}), \quad \mathbf{x}(0) = \mathbf{x}_0, \mathbf{u}(t) = \pi(\mathbf{x}(t), t).$$

For general cost functions of this form, finding the optimal policy π for all $\mathbf{x}_0$ reduces to solving a (potentially very complex) nonlinear programming problem.

9.2.1 Additive Cost

It turns out that the optimization problem often becomes a lot more tractable (both analytically and computationally) if we are willing to design our cost functions with an additive form. In particular, let us restrict our costs to take the form:

$$J^\pi(\mathbf{x}_0) = h(\mathbf{x}(T)) + \int_0^T g(\mathbf{x}(t), \mathbf{u}(t), t)dt, \quad \mathbf{x}(0) = \mathbf{x}_0.$$

This form is actually quite general. Before we get into the mechanics of solving for the optimal policies, let's get a little intuition by considering some example problem formulations, and intuition about their solutions.

EXAMPLE 9.1 Minimum-time problems

Recall the simple pendulum...

EXAMPLE 9.2 Quadratic regulators

More examples here...

9.3 DYNAMIC PROGRAMMING IN DISCRETE TIME

Now we will begin to develop the machinery for finding and/or verifying an optimal policy. This machinery can be developed more easily in discrete-time, so let's temporarily consider optimal control problems of the form:

$$\mathbf{x}[n+1] = \mathbf{f}(\mathbf{x}[n], \mathbf{u}[n], n) \tag{9.1}$$

$$J^\pi(\mathbf{x}_0) = h(\mathbf{x}[N]) + \sum_{n=0}^{N-1} g(\mathbf{x}[n], \mathbf{u}[n], n), \quad \mathbf{x}[0] = \mathbf{x}_0, \mathbf{u}[n] = \pi(\mathbf{x}[n], n). \tag{9.2}$$

Thanks to the additive form of this long-term cost, we can now start systematically solving for an optimal policy. Let's consider a subproblem, in which we start from $\mathbf{x}_m$ and time m and then continue on until N. Let us define the "cost-to-go" given that we are following policy π as $J^\pi(\mathbf{x}_m, m)$; it is given by

$$J^\pi(\mathbf{x}_m, m) = h(\mathbf{x}[N]) + \sum_{n=m}^{N-1} g(\mathbf{x}[n], \mathbf{u}[n]), \quad \mathbf{x}[m] = \mathbf{x}_m, \mathbf{u}[n] = \pi(\mathbf{x}, n).$$

Our original $J^\pi(\mathbf{x}_0)$ is simply the cost-to-go $J^\pi(\mathbf{x}_0, 0)$. For this reason, J^π is known as the *cost-to-go function* or the *value function*. The cost-to-go function can be written recursively:

$$\begin{aligned} J^\pi(\mathbf{x}, N) =& h(\mathbf{x}) \\ J^\pi(\mathbf{x}, N-1) =& g(\mathbf{x}[N-1], \mathbf{u}[N-1], N-1) + h(\mathbf{x}[N]) \\ &\vdots \\ J^\pi(\mathbf{x}, n) =& g(\mathbf{x}, \mathbf{u}, n) + J^\pi(\mathbf{x}[n+1], n+1) \end{aligned}$$

Even more important, the *optimal* cost-to-go function, J^* (simply defined as J^π for π^*), has almost the same recursive structure:

$$\begin{aligned} J^*(\mathbf{x}, N) =& h(\mathbf{x}) \\ J^*(\mathbf{x}, N-1) =& \min_{\mathbf{u}} \left[g(\mathbf{x}[N-1], \mathbf{u}, N-1) + h(\mathbf{x}[N]) \right] \\ &\vdots \\ J^*(\mathbf{x}, n) =& \min_{u} \left[g(\mathbf{x}, \mathbf{u}, n) + J^*(\mathbf{x}[n+1], n+1) \right]. \end{aligned}$$

Furthermore, the optimal policy is given by:

$$\pi^*(\mathbf{x}, n) = \operatorname*{argmin}_{\mathbf{u}} \left[g(\mathbf{x}, \mathbf{u}, n) + J^*(\mathbf{x}[n+1], n+1) \right].$$

This recursion, which starts at the terminal time and works backwards towards time 0, is known as *dynamic programming*. It is based on the key observation that for systems with a scalar additive cost, the cost-to-go function represents everything that the system needs to know about the future. The optimal action is simply the action which minimizes the sum of the one-step cost and the future optimal cost-to-go.

9.3.1 Discrete-State, Discrete-Action

For systems with continuous states and continuous actions, dynamic programming is a mathematical recipe for deriving the optimal policy and cost-to-go. For systems with a

Algorithm 1: The Dynamic Programming Algorithm.

```
Input: states $S$, actions $A$, horizon time $N$, dynamics $s' = f(s,a,n)$, instantaneous
       cost $g(s,a,n)$, and terminal cost $h(s)$
Output: optimal policy $\pi^*(s,n)$ and optimal cost-to-go $J^*(s,n)$

foreach state $s \in S$ do
    $J^*(s,N) \leftarrow h(s)$ ;
end
for $n \leftarrow N-1$ to $0$ do
    foreach state $s$ do
        $J^*(s,n) \leftarrow \min_{a \in A} \left[ g(s,a,n) + J^*(f(s,a,n), n+1) \right]$ ;
        $\pi^*(s,n) \leftarrow \text{argmin}_{a \in A} \left[ g(s,a,n) + J^*(f(s,a,n), n+1) \right]$ ;
    end
end
```

finite, discrete set of states and a finite, discrete set of actions, dynamic programming also represents a very efficient *algorithm* which can compute optimal policies (see Algorithm 1). This algorithm uses the states $\{s_1, s_2, ...\} \in S$, where S is the finite state space, and actions as $\{a_1, a_2, ...\} \in A$, where A is the finite action space.

Let's experiment with dynamic programming in a discrete-state, discrete-action, discrete-time problem to further improve our intuition.

EXAMPLE 9.3 Grid World

Robot living in a grid (finite state) world. Wants to get to the goal location. Possibly has to negotiate cells with obstacles. Actions are to move up, down, left, right, or do nothing. [84].

STILL NEED TO FILL IN THIS EXAMPLE WITH PLOTS, ETC.

Dynamic programming and value iteration. Pseudo-code? (or matlab code). Convergence results. Scalability.

9.3.2 Continuous-State, Discrete-Action

Barycentric interpolation. Examples on pendulum, acrobot, cart-pole, ...

9.3.3 Continuous-State, Continous-Actions

For some cost functions, we can solve for $\min_u$ analytically, and build that into our algorithm.

9.4 INFINITE HORIZON PROBLEMS

9.5 VALUE ITERATION

9.6 VALUE ITERATION W/ FUNCTION APPROXIMATION

Point-based VI

Insert diagram of the brick with control force here.

FIGURE 9.1 The brick on ice - a mechanical double integrator.

9.6.1 Special case: Barycentric interpolation

see also stochastic optimal control.

9.7 DETAILED EXAMPLE: THE DOUBLE INTEGRATOR

Let's apply our dynamic progrmming algorithms to get some intuition about optimal control of the simplest possible second-order dynamical system:

$$\ddot{q} = u.$$

This system is most commonly referred to as the double integrator. If you would like a mechanical analog of the system (I always do), then you can think about this as a unit mass brick moving along the x-axis on ice (no friction), with a control input which provides a horizontal force, u. The task is to design a control system, $u = \pi(\mathbf{x}, t)$, $\mathbf{x} = [q, \dot{q}]^T$ to regulate this brick to $\mathbf{x} = [0, 0]^T$.

9.7.1 Pole placement

There are many ways to investigate and control this simple system. The state space representation of this system is

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u.$$

If we consider simple linear feedback policies of the form

$$u = -\mathbf{K}\mathbf{x} = -\begin{bmatrix} k_1 & k_2 \end{bmatrix} \mathbf{x} = -k_1 q - k_2 \dot{q},$$

then we have

$$\dot{\mathbf{x}} = (\mathbf{A} + \mathbf{B}\mathbf{K})\mathbf{x} = \begin{bmatrix} 0 & 1 \\ -k_1 & -k_2 \end{bmatrix} \mathbf{x}.$$

The closed-loop eigenvalues, λ_1 and λ_2, and eigenvectors, v_1 and v_2 of this system are

$$\lambda_{1,2} = \frac{-k_2 \pm \sqrt{k_2^2 - 4k_1}}{2}, \quad v_1 = \begin{bmatrix} 1 \\ \lambda_1 \end{bmatrix}, \quad v_2 = \begin{bmatrix} 1 \\ \lambda_2 \end{bmatrix}.$$

The system is exponentially stable when both eigenvalues are strictly negative, e.g., $0 < \sqrt{k_2^2 - 4k_1} < k_2$. This implies that $k_1 > 0$. The eigenvalues are real when $k_2^2 \geq 4k_1$ (when $k_2^2 = 4k_1$, the system is critically damped, when $k_2^2 < 4k_1$ it is under-damped, and when $k_2^2 > 4k_1$ it is over-damped). As control designers, if we are to design k_1 and k_2, we could choose k_1 arbitrarily high, then choose $k_2 = 2\sqrt{k_1}$ to obtain an arbitrarily fast, critically damped, convergence.

An alternative way to say this same thing is to take the Laplace transform of the closed-loop system, $\ddot{q} = -k_1 q - k_2 \dot{q} + u'$, with the transfer function

$$H(s) = \frac{1}{s^2 + k_2 s + k_1} = \frac{1}{(s + \lambda_1)(s + \lambda_2)}.$$

By the same argument as above, we can easily select $k_1 = \frac{k_2^2}{4}$ to keep the poles on the real-axis, $H(s) = \frac{1}{(s+\frac{k_2}{2})^2}$, and drive those poles arbitrarily far into the left-half plane by choosing a large k_2. Or, with a more traditional root-locus analysis, we could amplify the gains k_1 and k_2 by the same linear multiplier k and see that for low k the system is underdamped (complex poles), and increasing k causes the poles to meet at the real axis and move off in opposite directions.

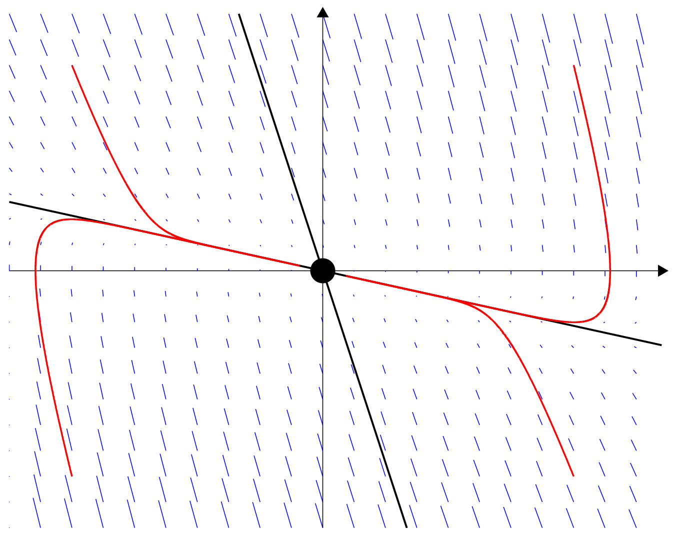

FIGURE 9.2 Phase plot of a double integrator with linear feedback, $k_1 = 1, k_2 = 4$.

9.7.2 The optimal control approach

Simple tuning of linear feedback gains can cause an arbitrarily fast convergence to the goal, but it may have many undesirable qualities. Perhaps the most obvious problem is that arbitrarily large feedback gains require arbitrarily large forces to implement. In some cases, we may have hard limits on the amount of force/torque that our actuators can produce. In other cases, since setting the gains to infinite is not a reasonable solution, we may like a more principled approach to balancing the cost of exerting large forces with the rate at which the system gets to the goal. In the remainder of this chapter we will consider to alternative formulations of optimal control problems for the double integrator to handle these two cases.

Thanks to the simplicity of the dynamics of this plant, each of the examples worked out here, will be computed analytically in the next chapters.

9.7.3 The minimum-time problem

Let's consider the problem of trying to get to the goal as fast as possible in the face of hard limits on the amount of force we can produce:

$$\ddot{q} = u, \quad u \in [-1, 1].$$

Informal derivation.

Before we do the real derivation, let's use our intuition to think through what the optimal control solution for this problem might look like.

We believe that the policy which optimizes this control problem is bang-bang; the control system should acclerate maximally towards the goal until a critical point at which it should hit the brakes in order to come perfectly to rest at the origin. Here we'll prove that this hypothesis is indeed correct.

First, we must define the bang-bang policy. We can do this by trying to figure out the manifold of states which, when the brakes are fully applied, bring the system to rest precisely at the origin. Integrating the equations, we have

$$\dot{q}(t) = ut + \dot{q}(0)$$
$$q(t) = \frac{1}{2}ut^2 + \dot{q}(0)t + q(0).$$

If we require that $q(t) = \dot{q}(t) = 0$, then these equations define a manifold of initial conditions $(q(0), \dot{q}(0))$ which come to rest at the origin:

$$\dot{q}(0) = -ut, \quad q(0) = \frac{1}{2}ut^2.$$

Cancelling t and using the "braking" policy on this manifold, $u = \text{sgn}(q)$, we have

$$\dot{q} = -\,\text{sgn}(q)\sqrt{2\,\text{sgn}(q)q}.$$

For all $\dot{q}$ greater than this switching surface, we apply $u = -1$, and less than, we apply $u = 1$. This policy is cartooned in Figure 9.3. Trajectories of the system executing this policy are also included - the fundamental characteristic is that the system is accelerated as quickly as possible toward the switching surface, then rides the switching surfance in to the origin.

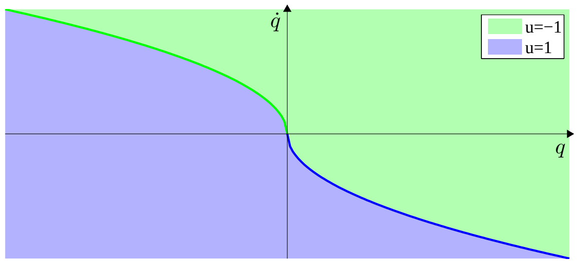

FIGURE 9.3 Candidate optimal (bang-bang) policy for the minimum-time double integrator problem.

Performance.

compare phase plots with linear control.