CHAPTER 12

Trajectory Optimization

So far, we have discussed a number of ways to solve optimal control problems via state space search (e.g., Dijkstra's and Dynamic Programming/Value Iteration). These methods have the drawback that they force you to discretize, typically both the state and action spaces. The resulting policy is optimal for the discretized system, but is only an approximation to the optimal solution for the continuous system. Today we're going to introduce a different type of algorithm which searches in the space of control policies, instead of in state space. These methods typically have the advantage that they scale (much) better to higher dimensional systems, and do not require any discretizations of state or action spaces. The disadvantage of this class of algorithms is that we must sacrifice guarantees of global optimality - they only guarantee local optimality, and can be subject to local minima.

12.1 THE POLICY SPACE

The fundamental idea is policy search methods is that, instead of discretizing an searching directly for the optimal policy, we define a class of policies by describing some parameterized control law. Then we search direcly for a setting of these parameters that (locally) optimize the cost function.

Notationally, we collect all of the parameters into a single vector, α, and write the controller (in general form) as $\pi_\alpha(\mathbf{x}, t)$. Here are some examples:

- linear feedback law:

$$\mathbf{u} = \mathbf{K}\mathbf{x} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \\ \alpha_5 & \alpha_6 & & \\ \vdots & & & \end{bmatrix} \mathbf{x}.$$

- open loop controller

$$\mathbf{u} = \alpha_n, \text{ where } n = \text{floor}(t/dt).$$

  Or α could be spline or fourier coefficients of the trajectory $\mathbf{u}(t)$.

- Neural networks and general function approximators.

12.2 NONLINEAR OPTIMIZATION

Having defined the policy space, our task is to search over α for the best policy. If the number of parameters is small, then brute force search may work. More generally, we must employ tools from nonlinear optimization.

12.2.1 Gradient Descent

First-order methods in general. Pros/cons of gradient descent. Suffer from:

- local minima
- have to chose a step-size (problems with anisotropy, or the golf-course problem)
- can require many iterations to converge

12.2.2 Sequential Quadratic Programming

Newton's method, etc. Intro chapter from Betts is a great guide. Powerful software packages (e.g., SNOPT).

12.3 SHOOTING METHODS

Perhaps the most obvious method for evaluating and optimizing the cost function from a single initial condition is by defining α as the decision variables and by evaluating $J^\alpha(\mathbf{x}_0)$ via forward simulation. Typically, the only way that these *shooting* methods take advantage of the additive cost structure of the optimal control problem is through efficient calculations of the policy gradient - $\frac{\partial J^\alpha(\mathbf{x}_0)}{\partial \alpha}$. Computing this gradient explicitly (rather than through numerical differentiation) greatly improves the performance of the optimization algorithms. We present two methods for computing this gradient here.

12.3.1 Computing the gradient with Backpropagation through time (BPTT)

Known for a long time in optimal control, but the name backprop-through-time came from the neural networks [66] community.

- Given the long-term cost function

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

- Starting from the initial condition $\mathbf{x}(0)$, integrate the equations of motion

$$\dot{\mathbf{x}} = f(\mathbf{x}, \pi_\alpha(\mathbf{x}, t)) \tag{12.1}$$

  forward in time to $t = T$. Keep track of $\mathbf{x}(t)$.

- Starting from the initial condition $\mathbf{y}(T) = \mathbf{0}$, integrate the *adjoint* equations

$$-\dot{\mathbf{y}} = \mathbf{F}_\mathbf{x}^T \mathbf{y} - \mathbf{G}_\mathbf{x}^T \tag{12.2}$$

  backward in time until $t = 0$, where

$$\mathbf{F}_\mathbf{x}(t) = \frac{\partial f}{\partial \mathbf{x}(t)} + \frac{\partial f}{\partial \mathbf{u}(t)} \frac{\partial \pi_\alpha}{\partial \mathbf{x}(t)}, \quad \mathbf{G}_\mathbf{x}(t) = \frac{\partial g}{\partial \mathbf{x}(t)} + \frac{\partial g}{\partial \mathbf{u}(t)} \frac{\partial \pi_\alpha}{\partial \mathbf{x}(t)},$$

  evaluated at $\mathbf{x}(t), \mathbf{u}(t)$.

- Finally, our gradients are given by

$$\frac{\partial J(\mathbf{x}_0)}{\partial \alpha} = \int_0^T dt \left[\mathbf{G}_\alpha^T - \mathbf{F}_\alpha^T \mathbf{y} \right], \tag{12.3}$$

  where

$$\mathbf{F}_\alpha(t) = \frac{\partial f}{\partial \mathbf{u}(t)} \frac{\partial \pi_\alpha}{\partial \alpha}, \quad \mathbf{G}_\alpha(t) = \frac{\partial g}{\partial \mathbf{u}(t)} \frac{\partial \pi_\alpha}{\partial \alpha},$$

  evaluated at $\mathbf{x}(t), \mathbf{u}(t)$.

Derivation w/ Lagrange Multipliers.
This algorithm minimizes the cost function

$$J^\alpha(\mathbf{x}_0) = \int_0^T g(\mathbf{x}, \pi_\alpha(\mathbf{x}, t)) dt$$

subject to the constraint

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

We will prove this using Lagrange multipliers and the calculus of variations, but it can also be shown using a finite-difference approximation of the unfolded network.

Consider the functional (a function of functions)

$$S[\mathbf{x}(t), \mathbf{y}(t)] = \int_0^T dt \left[g(\mathbf{x}, \pi_\alpha(\mathbf{x}, t)) + \mathbf{y}^T [\dot{\mathbf{x}} - f(\mathbf{x}, \pi_\alpha(\mathbf{x}, t))] \right].$$

If $\mathbf{x}(t)$ and $\mathbf{y}(t)$ are changed by small amounts $\delta\mathbf{x}(t)$ and $\delta\mathbf{y}(t)$, then the change in S is

$$\delta S = \int_0^T dt \left[\frac{\delta S}{\delta \mathbf{x}(t)} \delta \mathbf{x}(t) + \frac{\delta S}{\delta \mathbf{y}(t)} \delta \mathbf{y}(t) \right].$$

If $\frac{\delta S}{\delta \mathbf{x}(t)} = 0$ and $\frac{\delta S}{\delta \mathbf{y}(t)} = 0$ for all t, then $\delta S = 0$, and we say that S is at a stationary point with respect to variations in $\mathbf{x}$ and $\mathbf{y}$. To ensure that $\delta \mathbf{x}(0) = 0$, we will hold the initial conditions $\mathbf{x}(0)$ constant.

Now compute the functional derivatives:

$$\frac{\delta S}{\delta \mathbf{y}(t)} = [\dot{\mathbf{x}} - f(\mathbf{x}, \pi_\alpha(\mathbf{x}, t))]^T.$$

The forward dynamics of the algorithm gaurantee that this term is zero. To compute $\frac{\delta S}{\delta \mathbf{x}(t)}$, we first need to integrate by parts to handle the $\dot{\mathbf{x}}$ term:

$$\int_0^T \mathbf{y}^T \dot{\mathbf{x}} dt = \mathbf{y}^T \mathbf{x}|_0^T - \int_0^T \dot{\mathbf{y}}^T \mathbf{x} dt.$$

Rewrite S as

$$S = \int_0^T dt \left[g(\mathbf{x}, \pi_\alpha(\mathbf{x}, t)) - \mathbf{y}^T f(\mathbf{x}, \pi_\alpha(\mathbf{x}, t)) - \dot{\mathbf{y}}^T \mathbf{x} \right] + \mathbf{y}(T)^T \mathbf{x}(T) - \mathbf{y}(0)^T \mathbf{x}(0).$$

Therefore, the function derivative is

$$\frac{\delta S}{\delta \mathbf{x}(t)} = \mathbf{G_x} - \mathbf{y}^T\mathbf{F_x} - \dot{\mathbf{y}}^T + \mathbf{y}(T)^T\delta(t-T) - \mathbf{y}(0)^T\delta(t).$$

By choosing $\mathbf{y}(T) = \mathbf{0}$, the backward dynamics guarantee that this term is zero (except for at $t = 0$, but $\delta\mathbf{x}(0) = 0$).

The forward and backward passes put us at a stationary point of S with respect to variations in $\mathbf{x}(t)$ and $\mathbf{y}(t)$, therefore the only dependence of S on $\mathbf{w}$ is the explicit dependence:

$$\frac{\partial S}{\partial \alpha} = \int_0^T dt \left[\mathbf{G}_\alpha - \mathbf{y}^T\mathbf{F}_\alpha\right]$$

12.3.2 Computing the gradient w/ Real-Time Recurrent Learning (RTRL)

Backpropagating through time requires that the network maintains a trace of it's activity for the duration of the trajectory. This becomes very inefficient for long trajectories. Using Real-Time Recurrent Learning (RTRL), we solve the temporal credit assignment problem during the forward integration step by keeping a running estimate of the total effect that parameters α have on the state $\mathbf{x}$. The algorithm is

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \pi_\alpha(\mathbf{x}, t), t) \tag{12.4}$$

$$\dot{\mathbf{P}} = \mathbf{F_x}\mathbf{P} + \mathbf{F}_\alpha \tag{12.5}$$

$$\frac{d}{dt}\frac{\partial J^\alpha(\mathbf{x}_0)}{\partial \alpha} = \mathbf{G_x}\mathbf{P} + \mathbf{G}_\alpha \tag{12.6}$$

These equations are integrated forward in time with initial conditions $\mathbf{x}(0) = \mathbf{x}_0$, $\mathbf{P}(0) = \mathbf{0}$, $\frac{\partial J}{\partial \alpha} = \mathbf{0}$.

This algorithm can be computationally expensive, because we have to store $O(NM)$ variables in memory and process as many differential equations at every time step. On the other hand, we do not have to keep a trace of the trajectory, so the memory footprint does not depend on the duration of the trajectory. The major advantage, however, is that we do not have to execute the backward dynamics - the temporal credit assignment problem is solved during the forward pass.

Derivation.

Once again, we can show that this algorithm performs gradient descent on the cost function

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

Define

$$P_{ij} = \frac{\partial x_i}{\partial \alpha_j}.$$

The gradient calculations are straight-forward:

$$\frac{\partial J}{\partial \alpha} = \int_0^T dt \left[\frac{\partial g}{\partial \mathbf{x}(t)} \frac{\partial \mathbf{x}(t)}{\partial \alpha} + \frac{\partial g}{\partial \mathbf{u}(t)} \frac{\partial \pi_\alpha}{\partial \alpha} + \frac{\partial g}{\partial \mathbf{u}(t)} \frac{\partial \pi}{\partial \mathbf{x}(t)} \frac{\partial \mathbf{x}(t)}{\partial \alpha} \right]$$
$$= \int_0^T dt \left[\mathbf{G_x P} + \mathbf{G}_\alpha \right]$$

To calculate the dynamics of $\mathbf{p}$ and $\mathbf{q}$, differentiate the equation

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \pi_\alpha(\mathbf{x}, t), t)$$

to obtain

$$\frac{d}{dt} \frac{\partial \mathbf{x}}{\partial \alpha} = \left[\frac{\partial \mathbf{f}}{\partial \mathbf{x}} + \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \frac{\partial \pi_\alpha}{\partial \mathbf{x}} \right] \frac{\partial \mathbf{x}}{\partial \alpha} + \frac{\partial \mathbf{f}}{\partial \mathbf{u}} \frac{\partial \pi_\alpha}{\partial \alpha}$$
$$\dot{\mathbf{P}} = \mathbf{F_x P} + \mathbf{F}_\alpha \quad \mathbf{P}(0) = \mathbf{0}$$

By initializing $\mathbf{P}(0) = \mathbf{0}$, we are simply assuming that the $\mathbf{x}(0)$ does not depend on α.

12.3.3 BPTT vs. RTRL

Both methods compute the gradient. So which should we prefer? The answer may be problem specific. The memory cost of the BPTT algorithm is that you must remember $\mathbf{x}(t)$ on the forward simulation, which is $dim(\mathbf{x}) \times length(T)$. The memory cost of RTRL is storing $\mathbf{P}$ which has size $dim(\mathbf{x}) \times dim(\alpha)$. Moreover, RTRL involves integrating $dim(\mathbf{x}) + dim(\mathbf{x}) \times dim(\alpha)$ ODEs, where as BPTT only integrates $2 \times dim(\mathbf{x})$ ODEs. For long trajectories and a small number of policy parameters, RTRL might more efficient that BPTT, and is probably easier to implement. For shorter trajectories with a lot of parameters, BPTT would probably be the choice.

Another factor which might impact the decision is the constraints. Gradients of constraint functions can be computed with either method, but if one seeks to constrain $\mathbf{x}(t)$ for all t, then RTRL might have an advantage since it explicitly stores $\frac{\partial \mathbf{x}(t)}{\partial \alpha}$.

12.4 DIRECT COLLOCATION

The shooting methods require forward simulation of the dynamics to compute the cost function (and its gradients), and therefore can be fairly expensive to evaluation. Additionally, they do not make very effective use of the capabilities of modern nonlinear optimization routines (like SNOPT) to enforce, and even take advantage of, constraints.

In the direct collocation approach, we formulate the decision parameters as *both* α, which in the open-loop case reduces to $\mathbf{u}[n]$, and additionally $\mathbf{x}[n]$. By over-parameterizing the system with both $\mathbf{x}$ and $\mathbf{u}$ are explicit decision variables, the cost function and its gradients can be evaluated without explicit simulation of the dynamics. Instead, the dynamics are imposed as constraints on the decision variables. The resulting formuation is:

$$\min_{\forall n, \mathbf{x}[n], \mathbf{u}[n]} J = \sum_{n=1}^{N} g(\mathbf{x}[n], \mathbf{u}[n]), \quad \text{s.t. } \mathbf{x}[n+1] = \mathbf{f}(\mathbf{x}[n], \mathbf{u}[n]).$$

These methods are very popular today. The updates are fast, and it is very easy to implement constraints. The most commonly stated limitation of these methods is their accuracy, since they use a fixed step-size integration (rather than a variable step solver). However, methods like BPTT or RTRL can also be used to implement the state-action constraints if accuracy is a premium.

A less obvious attribute of these methods is that they may be easier to initialize with trajectories (eg, specifying $\mathbf{x}_0(t)$ directly) that are in the vicinity of the desired minima.

12.5 LQR TRAJECTORY STABILIZATION

When the policy parameterization is explicitly the open-loop trajectory (u_{tape}), then trajectory optimization by shooting methods and/or direct collocation both have the property that the solution upon convergence satisfies the Pontryagin minimum principle. When the policy is parameterized with feedback, the story is a little more complicated (but similar). But there is no reason to believe that the system is stable along these trajectories - executing the locally optimal open-loop trajectories on the system with small changes in initial conditions, small disturbances or modeling errors, or even with a different integration step can cause the simulated trajectories to diverge from the planned trajectory.

In this section we will develop one set of tools for trajectory stabilization. There are many candidates for trajectory stabilization in fully-actuated sytems (many based on feedback linearization), but trajectory stabilization for underactuated systems can be easily implemented using a version of the Linear Quadratic Regulator (LQR) results from chapter 10.

12.5.1 Linearizing along trajectories

In order to apply the linear quadratic regulator, we must first linearize the dynamics. So far we have linearized around fixed points of the dynamics... linearizing around a non-fixed point is just slightly more subtle. Considering the system

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

if we perform a Taylor expansion around a random point $(\mathbf{x}_0, \mathbf{u}_0)$, the result is

$$\dot{\mathbf{x}} \approx \mathbf{f}(\mathbf{x}_0, \mathbf{u}_0) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x} - \mathbf{x}_0) + \frac{\partial \mathbf{f}}{\partial \mathbf{u}}(\mathbf{u} - \mathbf{u}_0) = \mathbf{c} + \mathbf{A}(\mathbf{x} - \mathbf{x}_0) + \mathbf{B}(\mathbf{u} - \mathbf{u}_0).$$

In other words, the resulting dynamics are not linear (but affine) in the coordinates of $\mathbf{x}$. The simple trick is to change coordinates to

$$\bar{\mathbf{x}}(t) = \mathbf{x}(t) - \mathbf{x}_0(t), \quad \bar{\mathbf{u}}(t) = \mathbf{u}(t) - \mathbf{u}_0(t),$$

where $[\mathbf{x}_0(t), \mathbf{u}_0(t)]$ is a solution to the dynamics - a feasible trajectory. Then we have

$$\dot{\bar{\mathbf{x}}}(t) = \dot{\mathbf{x}}(t) - \dot{\mathbf{x}}_0(t) = \dot{\mathbf{x}}(t) - \mathbf{f}(\mathbf{x}_0(t), \mathbf{u}_0(t)),$$

and therefore

$$\begin{aligned}\dot{\bar{\mathbf{x}}}(t) =& \frac{\partial \mathbf{f}(\mathbf{x}_0(t), \mathbf{u}_0(t))}{\partial \mathbf{x}}(\mathbf{x}(t) - \mathbf{x}_0(t)) + \frac{\partial \mathbf{f}(\mathbf{x}_0(t), \mathbf{u}_0(t))}{\partial \mathbf{u}}(\mathbf{u} - \mathbf{u}_0(t)) \\ =& \mathbf{A}(t)\bar{\mathbf{x}}(t) + \mathbf{B}(t)\bar{\mathbf{u}}(t).\end{aligned}$$

In other words, if we are willing to change to a coordinate system that moves along a feasible trajectory, than the Taylor expansion of the dynamics results in a time-varying linear system. Linear time-varying (LTV) systems are a very rich class of systems which are still amenable to many of the linear systems analysis[21].

The big idea here is that we are using a particular solution trajectory to reparameterize the path through state-space as purely a function of time. [Add cartoon images here].

12.5.2 Linear Time-Varying (LTV) LQR

Given a linear time-varying approximation of the model dynamics along the trajectory,

$$\dot{\bar{\mathbf{x}}} = \mathbf{A}(t)\bar{\mathbf{x}} + \mathbf{B}(t)\bar{\mathbf{u}},$$

we can formulate a trajectory stabilization as minimizing the cost function

$$J(\mathbf{x}_0, 0) = \bar{\mathbf{x}}(t_f)^T\mathbf{Q}_f\bar{\mathbf{x}}(t_f) + \int_0^T dt \left[\bar{\mathbf{x}}(t)^T\mathbf{Q}\bar{\mathbf{x}}(t) + \bar{\mathbf{u}}(t)^T\mathbf{R}\bar{\mathbf{u}}(t)\right].$$

This cost function penalizes the system (quadratically) at time t for being away from $\mathbf{x}_0(t)$. Even with the time-varying components of the dynamics and the cost function, it is still quite simple to use the finite-horizon LQR solution from 2. If, as before, we guess

$$J(\bar{\mathbf{x}}, t) = \bar{\mathbf{x}}^T\mathbf{S}(t)\bar{\mathbf{x}},$$

we can satisfy the HJB with:

$$-\dot{\mathbf{S}}(t) = \mathbf{Q} - \mathbf{S}(t)\mathbf{B}(t)\mathbf{R}^{-1}\mathbf{B}^T\mathbf{S}(t) + \mathbf{S}(t)\mathbf{A}(t) + \mathbf{A}^T(t)\mathbf{S}(t), \quad \mathbf{S}(T) = \mathbf{Q}_f.$$
$$\mathbf{u}^*(t) = \mathbf{u}_0(t) - \mathbf{R}^{-1}\mathbf{B}^T(t)\mathbf{S}(t)\bar{\mathbf{x}}(t)$$

In general, it is also trivial to make $\mathbf{Q}$ and $\mathbf{R}$ functions of time. It is also nice to observe that if one aims to stabilize an infinite-horizon trajectory, for which $\forall t \geq t_f, \mathbf{x}_0(t) = \mathbf{x}_0(t_f), \mathbf{u}_0(t) = \mathbf{u}_0(t_f)$, and $\mathbf{f}(\mathbf{x}_0(t_f), \mathbf{u}_0(t_f) = \mathbf{0}$, then we can use the boundary conditions $\mathbf{S}(T) = \mathbf{S}_\infty$, where $\mathbf{S}_\infty$ is the steady-state solution of the LTI LQR at the fixed point.

[Add simulation results from the pendulum, with and without feedback, and cart-pole, using both dircol and shooting?]

[Add image of value function estimates on top of pendulum trajectory]

Notice that, like the LTI LQR, this control derivation should scale quite nicely to high dimensional systems (simply involves integrating a $n \times n$ matrix backwards). Although dynamic programming, or some other nonlinear feedback design tool, could be used to design trajectory stabilizers for low-dimensional systems, for systems where open-loop trajectory optimization is the tool of choice, the LTV LQR stabilizers are a nice match.

12.6 ITERATIVE LQR

The LTV LQR solutions used to stabilize trajectories in the last section can also be modified to create an algorithm for unconstrained optimization of open-loop trajectories. Iterative LQR (iLQR), also known as Sequential LQR (SLQ)[74], and closely related to Differential Dynamic Programming (DDP)[42], can be thought of as almost a drop-in replacement for a shooting or direct collocation method.

The instantaneous cost function for trajectory stabilization took the form: $\bar{\mathbf{x}}^T\mathbf{Q}\bar{\mathbf{x}}$, with the result being that states off the desired trajectory are regulated back to the desired trajectory. But what happens if we use the LQR derivation to optimize a more arbitrary cost function? Given an instantaneous cost function, $g(\mathbf{x},\mathbf{u})$, we can form a quadratic approximation of this cost function with a Taylor expansion about $\mathbf{x}_0(t), \mathbf{u}_o(t)$:

$$g(\mathbf{x},\mathbf{u}) \approx g(\mathbf{x}_0,\mathbf{u}_0) + \frac{\partial g}{\partial \mathbf{x}}\bar{\mathbf{x}} + \frac{\partial g}{\partial \mathbf{u}}\bar{\mathbf{u}} + \frac{1}{2}\bar{\mathbf{x}}^T\frac{\partial^2 g}{\partial \mathbf{x}^2}\bar{\mathbf{x}} + \bar{\mathbf{x}}\frac{\partial^2 g}{\partial \mathbf{x}\partial \mathbf{u}}\bar{\mathbf{u}} + \frac{1}{2}\bar{\mathbf{u}}^T\frac{\partial^2 g}{\partial \mathbf{u}^2}\bar{\mathbf{u}}.$$

By inserting this (time-varying) cost function into the LQR solution, with the time-varying linearization around a nominal trajectory, we can once again derive an optimal control policy by integrating a Riccati equation backwards (almost identical to the derivation in example 3). [insert official derivation here]

Given an initial trajectory $\mathbf{x}_0(t), \mathbf{u}_0(t)$ (generated, for instance, by choosing $\mathbf{u}_0(t)$ to be some random initial trajectory, then simulating to get $\mathbf{x}_0(t)$), the cost function no longer rewards the system for stabilizing the desired trajectory, but rather for driving towards the minimum of the cost function (which has an interpretation as a different desired trajectory, $\mathbf{x}_d(t) \neq \mathbf{x}_0(t)$). The LQR solution will use the linearization along $\mathbf{x}_0(t)$ to try to stabilize the system on $\mathbf{x}_d(t)$. Because the LTV model is only valid near $\mathbf{x}_0(t)$, the resulting controller may do a poor job of getting to $\mathbf{x}_d(t)$, but will be better than the previous controller. The algorithm proceeds by replacing $\mathbf{u}_0(t)$ with the control actions executed by the LQR feedback policy from the initial conditions, computes the corresponding new $\mathbf{x}_0(t)$, and iterates.

[insert implementation snapshots here for intuition.]

Iterative LQR can be thought of as a second-order solution method (like SQP), which should converge on a trajectory which satisfies the Pontryagin minimum principle in a relatively small number of interations. It will likely require fewer trajectories to be considered that using BPTT, RTRL, or DIRCOL, because the LQR solution directly computes a second-order update, whereas the other methods as presented compute only $\frac{\partial J}{\partial \alpha}$, and rely on SNOPT to approximate the Hessian.

The very popular DDP method[42] is very similar to this solution, although it also uses a second-order expansion of the equations of motion. Many authors say that they are using DDP when they are actually using iLQR[1].

12.7 REAL-TIME PLANNING (AKA RECEDING HORIZON CONTROL)

LQR trajectory stabilization is one approach to making the optimized open-loop trajectories robust to disturbances. But if you can plan fast enough (say for steering a ship), then computing a short-term finite-horizon open-loop policy at every dt using the current state as initial condtions can be another reasonable approach to creating what is effectively a feedback policy. The technique is known as receding-horizon control.