

Chapter 3: Pontryagin's Maximum Principle, Shooting & Multiple Shooting & LQR, Riccati, QP viewpoint (finite / infinite horizon)

1. Introduction

Optimal control theory provides a systematic framework for determining control inputs that minimize a cost functional while satisfying the system dynamics.

In this chapter, we focus primarily on **discrete-time optimal control**, while occasionally drawing parallels to the continuous case to build intuition.

We will develop the theoretical foundation using **Pontryagin's Maximum Principle (PMP)** in discrete form, and then move toward **numerical solution methods**—notably **shooting methods**, **Quadratic Programming (QP)** formulations, and the **Riccati recursion** that leads to the Linear Quadratic Regulator (LQR).

Our approach will make frequent connections to **optimization techniques** discussed in Lecture 2, particularly **Newton's method** and the **structure of quadratic cost functions**.

1.1 From Optimization to Control

Consider a finite-horizon optimization problem of the form:

$$\min_{u_0, \dots, u_{N-1}} J = \sum_{k=0}^{N-1} \ell(x_k, u_k) + \ell_f(x_N)$$

subject to the discrete dynamics

$$x_{k+1} = f(x_k, u_k), \quad x_0 = x_{\text{init}}.$$

This setup resembles nonlinear programming, where the **decision variables** are the control inputs u_0, \dots, u_{N-1} , and the **constraints** are imposed by the system dynamics.

Hence, many of the methods for constrained optimization—such as Lagrange multipliers, gradient-based updates, and Newton's method—can be directly applied.

1.2 The Discrete Pontryagin's Maximum Principle (PMP)

To derive necessary conditions for optimality, we introduce the **Hamiltonian function**

$$H(x_k, u_k, \lambda_{k+1}) = \ell(x_k, u_k) + \lambda_{k+1}^\top f(x_k, u_k),$$

where λ_k denotes the **adjoint variable** (or costate).

The discrete PMP yields the following set of necessary conditions:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k), \quad x_0 = x_{\text{init}}, \\ \lambda_k &= \frac{\partial H}{\partial x_k} = \frac{\partial \ell}{\partial x_k} + \left(\frac{\partial f}{\partial x_k} \right)^\top \lambda_{k+1}, \\ 0 &= \frac{\partial H}{\partial u_k} = \frac{\partial \ell}{\partial u_k} + \left(\frac{\partial f}{\partial u_k} \right)^\top \lambda_{k+1}. \end{aligned}$$

These conditions define a **two-point boundary value problem (BVP)**:

the state evolves forward in time, while the costate evolves backward.

For **linear dynamics** and **quadratic cost**, these equations simplify significantly, giving rise to the **Linear Quadratic Regulator (LQR)** formulation, which will serve as the primary example for this chapter.

1.3 Structure of the Chapter

The chapter is organized as follows:

1. Section 2 introduces shooting and multiple-shooting methods for solving the discrete PMP equations.
2. Section 3 reformulates the finite-horizon optimal control problem as a Quadratic Program (QP) and shows how it can be efficiently solved using **Newton's method** (see Lecture 2).
3. Section 4 develops the **Riccati recursion**, revealing how it corresponds to solving the KKT system of the QP in a structured and computationally efficient way.
4. Section 5 connects the **Riccati recursion** to **feedback control**, showing how the optimal control law emerges naturally.

Throughout the chapter, we will complement each theoretical section with code examples in Python/NumPy to illustrate convergence, stability, and computational efficiency.

2. Shooting and Multiple Shooting Methods

When solving discrete-time optimal control problems using Pontryagin's Maximum Principle (PMP), we obtain a **two-point boundary value problem (BVP)**.

In this formulation, the **state** evolves forward in time, while the **costate** (or adjoint variable) evolves backward.

Shooting and multiple shooting methods are numerical approaches for solving these coupled forward–backward equations efficiently.

2.1 Single Shooting

The **single shooting** method reformulates the optimal control problem as an initial value problem (IVP).

Given an initial condition $x_0 = x_{\text{init}}$ and a guessed control sequence u_0, u_1, \dots, u_{N-1} , the system evolves as:

$$x_{k+1} = f(x_k, u_k), \quad k = 0, \dots, N-1.$$

The total cost is:

$$J(x_{0:N}, u_{0:N-1}) = \sum_{k=0}^{N-1} \ell(x_k, u_k) + \ell_f(x_N).$$

The discrete-time Hamiltonian is defined as:

$$H(x_k, u_k, \lambda_{k+1}) = \ell(x_k, u_k) + \lambda_{k+1}^\top f(x_k, u_k),$$

and the necessary conditions for optimality (discrete PMP) are:

$$\begin{aligned} x_{k+1} &= f(x_k, u_k), \\ \lambda_k &= \frac{\partial \ell}{\partial x_k} + \left(\frac{\partial f}{\partial x_k} \right)^\top \lambda_{k+1}, \\ 0 &= \frac{\partial \ell}{\partial u_k} + \left(\frac{\partial f}{\partial u_k} \right)^\top \lambda_{k+1}. \end{aligned}$$

The boundary conditions are $x_0 = x_{\text{init}}$ and $\lambda_N = \frac{\partial \ell_f}{\partial x_N}$.

To solve for the unknown initial costate λ_0 , we define a **shooting function**:

$$\Phi(\lambda_0) = x_N(\lambda_0) - x_N^{\text{target}},$$

and apply **Newton's method**:

$$\lambda_0^{(i+1)} = \lambda_0^{(i)} - \left(\frac{\partial \Phi}{\partial \lambda_0} \right)^{-1} \Phi(\lambda_0^{(i)}).$$

Although conceptually simple, this approach becomes unstable or slow when N is large or f is nonlinear, because small errors in λ_0 amplify across the horizon.

2.2 Example: Linear-Quadratic Single Shooting in Julia

Consider the discrete-time linear system:

$$x_{k+1} = Ax_k + Bu_k,$$

and the quadratic cost function:

$$J = \sum_{k=0}^{N-1} \left(x_k^T Q x_k + u_k^T R u_k \right) + x_N^T Q_f x_N.$$

A simple single-shooting simulation and cost evaluation can be written in Julia as follows:

```
using LinearAlgebra

# System matrices
A = [1.0 0.1;
     0.0 1.0]
B = [0.0; 0.1]

# Cost matrices
Q = I(2)
R = 0.1 .* I(1)
Qf = 10.0 .* I(2)

# Horizon and initial condition
N = 30
x0 = [1.0, 0.0]
u = zeros(N) # initial guess

# Rollout function
function rollout(A, B, x0, u)
    x = Vector{Vector{Float64}}{undef, N+1}
    x[1] = x0
    for k in 1:N
        x[k+1] = A * x[k] + B * u[k]
    end
    return x
end

# Cost computation
function cost(x, u, Q, R, Qf)
    J = 0.0
    for k in 1:length(u)
        J += x[k]' * Q * x[k] + u[k]' * R * u[k]
    end
    J += x[end]' * Qf * x[end]
    return J
end

x = rollout(A, B, x0, u)
J = cost(x, u, Q, R, Qf)
println("Initial cost: ", J)
```

2.3 Multiple Shooting Method

The **multiple shooting method** enhances the numerical robustness of solving optimal control problems by dividing the horizon into smaller subintervals.

Unlike single shooting, which integrates the entire trajectory from a single initial condition, multiple shooting introduces **intermediate states** as additional decision variables.

This approach significantly improves numerical conditioning, especially for long or nonlinear trajectories.

2.3.1 Problem Setup

We divide the time horizon into M segments:

$$0 = k_0 < k_1 < k_2 < \dots < k_M = N.$$

For each segment $i \in \{0, \dots, M-1\}$, we integrate the dynamics locally as:

$$x_{k_{i+1}} = F_i(x_{k_i}, U_i),$$

where:

- F_i represents the numerical propagation function (the flow map) for the system within segment i ,
- $U_i = [u_{k_i}, u_{k_i+1}, \dots, u_{k_{i+1}-1}]$ is the control sequence for that segment.

To ensure the global trajectory is continuous, we impose **matching constraints** between segments:

$$c_i = x_{k_{i+1}}^{(i)} - x_{k_{i+1}}^{(i+1)} = 0, \quad i = 0, \dots, M-1.$$

Hence, the **multiple-shooting optimal control problem** is formulated as:

$$\begin{aligned} \min_{\{x_{k_i}, U_i\}} \quad & \sum_{i=0}^{M-1} J_i(x_{k_i}, U_i) + \ell_f(x_N), \\ \text{s.t.} \quad & x_{k_{i+1}}^{(i)} = F_i(x_{k_i}, U_i), \\ & x_{k_{i+1}}^{(i)} - x_{k_{i+1}}^{(i+1)} = 0. \end{aligned}$$

Compared with single shooting, this formulation has more variables but produces a **better-conditioned** system for numerical optimization (e.g., SQP or Newton-based methods).

2.3.2 Advantages

- **Improved numerical conditioning:**
Since each segment is shorter, local sensitivities are smaller and the Jacobian matrices are better conditioned.
- **Parallel computation:**
Each segment can be integrated independently, enabling parallelization across processors.
- **Better convergence:**
The method is more robust to poor initial guesses of control inputs or states, particularly in nonlinear or stiff systems.

2.3.3 Linear-Quadratic Case

For a linear discrete-time system:

$$x_{k+1} = Ax_k + Bu_k,$$

and a quadratic cost:

$$J = \sum_{k=0}^{N-1} (x_k^\top Q x_k + u_k^\top R u_k) + x_N^\top Q_f x_N,$$

the multiple-shooting formulation remains convex.

We can solve it efficiently as a structured **Quadratic Program (QP)** with equality constraints that enforce continuity between subintervals.

2.3.4 Julia Example: Two-Segment Multiple Shooting

Below is an example implementation in **Julia** using the JuMP package and Ipopt solver.

```
using JuMP, Ipopt, LinearAlgebra

# System dynamics
A = [1.0 0.1;
     0.0 1.0]
B = [0.0; 0.1]

# Cost matrices
Q = I(2)
R = 0.1 .* I(1)
Qf = 10.0 .* I(2)

# Problem setup
N = 30
ΔN = 15
x0 = [1.0, 0.0]

model = Model(Ipopt.Optimizer)
set_silent(model)

# Decision variables
@variable(model, u1[1:ΔN])
@variable(model, u2[1:ΔN])
@variable(model, x1[1:2, 1:ΔN+1])
@variable(model, x2[1:2, 1:ΔN+1])

# Initial condition
@constraint(model, x1[:, 1] .== x0)

# Segment 1 dynamics
for k in 1:ΔN
    @constraint(model, x1[:, k+1] .== A * x1[:, k] + B * u1[k])
end

# Continuity between segments
@constraint(model, x2[:, 1] .== x1[:, ΔN+1])

# Segment 2 dynamics
for k in 1:ΔN
    @constraint(model, x2[:, k+1] .== A * x2[:, k] + B * u2[k])
end

# Objective
@objective(model, Min,
    sum(x1[:,k]'*Q*x1[:,k] + u1[k]'*R*u1[k] for k in 1:ΔN) +
    sum(x2[:,k]'*Q*x2[:,k] + u2[k]'*R*u2[k] for k in 1:ΔN) +
    x2[:,ΔN+1]'*Qf*x2[:,ΔN+1]
)

optimize!(model)
println("Optimal cost (multiple shooting): ", objective_value(model))
```

3. From Shooting to Quadratic Programming (QP)

3.1 Motivation

Both **single shooting** and **multiple shooting** reformulate the optimal control problem by treating control inputs and, optionally, intermediate states as optimization variables.

However, when the dynamics are *linear* and the cost is *quadratic*, the entire optimal control problem can be transformed into a **Quadratic Program (QP)** — a form that can be solved efficiently using modern optimization algorithms.

This section shows how to rewrite the discrete-time Linear Quadratic Regulator (LQR) problem as a structured QP, forming the foundation for the **Riccati recursion**.

3.2 Discrete Dynamics and Constraints

We start from the linear time-invariant system:

$$x_{k+1} = Ax_k + Bu_k, \quad x_0 = x_{\text{init}}.$$

We can recursively express all future states as functions of the control sequence $\{u_k\}_{k=0}^{N-1}$:

$$\begin{aligned} x_1 &= Ax_0 + Bu_0, \\ x_2 &= A^2x_0 + ABu_0 + Bu_1, \\ x_3 &= A^3x_0 + A^2Bu_0 + ABu_1 + Bu_2, \\ &\vdots \\ x_N &= A^Nx_0 + \sum_{i=0}^{N-1} A^{N-1-i}Bu_i. \end{aligned}$$

From these relations, we can stack all states into a single vector form:

$$X = \text{col}(x_1, \dots, x_N).$$

$$X = \mathcal{S}U + \mathcal{T}x_0.$$

where $X = [x_1^\top, \dots, x_N^\top]^\top$ and $U = [u_0^\top, \dots, u_{N-1}^\top]^\top$.

Here:

- \mathcal{S} collects all coefficients of the controls u_k ;
- \mathcal{T} stacks the propagated powers of A applied to the initial state x_0 .

Explicitly:

$$\mathcal{S} = \begin{bmatrix} B & 0 & \cdots & 0 \\ AB & B & \cdots & 0 \\ A^2B & AB & \ddots & \vdots \\ \vdots & \vdots & \ddots & B \end{bmatrix}, \quad \mathcal{T} = \begin{bmatrix} A \\ A^2 \\ \vdots \\ A^N \end{bmatrix}.$$

Therefore, the compact matrix form of the system dynamics is:

$$X = \mathcal{S}U + \mathcal{T}x_0.$$

3.3 Cost Function as a Quadratic Form

The total cost is defined as:

$$J = \sum_{k=0}^{N-1} (x_k^\top Q x_k + u_k^\top R u_k) + x_N^\top Q_f x_N.$$

Substitute the dynamics expression $X = \mathcal{S}U + \mathcal{T}x_0$ into the cost function:

$$J(U) = \frac{1}{2}U^\top H U + U^\top g + c,$$

where:

$$H = 2 (\mathcal{S}^\top \bar{Q} \mathcal{S} + \bar{R}), \quad g = 2 \mathcal{S}^\top \bar{Q} \mathcal{T} x_0.$$

and:

$$\bar{Q} = \text{diag}(Q, \dots, Q, Q_f), \quad \bar{R} = \text{diag}(R, \dots, R).$$

Thus, the LQR problem is transformed into a **convex Quadratic Program**:

$$\begin{aligned} \min_U \quad & \frac{1}{2}U^\top H U + U^\top g, \\ \text{s.t.} \quad & U \in \mathcal{U}, \quad X \in \mathcal{X}. \end{aligned}$$

If there are no state or control constraints, the QP can be solved analytically — leading directly to the Riccati equations.

3.4 QP Formulation and KKT Conditions

The Karush–Kuhn–Tucker (KKT) conditions for the QP correspond to the **first-order optimality conditions** of PMP. In the unconstrained case, solving the KKT system is equivalent to solving the backward–forward costate equations.

Specifically, the linearized system of optimality equations has a **block-tridiagonal structure**:

$$\mathbf{H} \text{col}(\Delta x_0, \Delta u_0, \dots, \Delta u_{N-1}, \Delta x_N) = \mathbf{r}.$$

where the block matrix \mathbf{H} has the tridiagonal structure

$$\mathbf{H} = \begin{bmatrix} H_0 & A_1^\top & 0 & \cdots & 0 \\ A_1 & H_1 & A_2^\top & \cdots & 0 \\ 0 & A_2 & H_2 & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & A_N^\top \\ 0 & 0 & 0 & A_N & H_N \end{bmatrix}, \quad \mathbf{r} = \begin{bmatrix} r_0 \\ r_1 \\ \vdots \\ r_{N-1} \\ r_N \end{bmatrix}.$$

Solving this efficiently is the purpose of the **Riccati recursion**.

3.5 Julia Example: QP Formulation for LQR

The following Julia example constructs and solves the QP equivalent of the LQR problem using **JuMP** and **OSQP**.

```

using JuMP, OSQP, LinearAlgebra, SparseArrays

# System definition
A = [1.0 0.1;
      0.0 1.0]
B = [0.0; 0.1]
Q = I(2)
R = 0.1 * I(1)
Qf = 10.0 * I(2)
N = 30
x0 = [1.0, 0.0]

nx, nu = size(B)

# Build lifted dynamics matrices
A_powers = [A^k for k in 1:N]
T = vcat(A_powers...)
S = zeros(nx*N, nu*N)
for i in 1:N
    for j in 1:i
        S[(nx*(i-1)+1):(nx*i), (nu*(j-1)+1):(nu*j)] = A^(i-j) * B
    end
end

# Quadratic cost matrices
Qbar = kron(Diagonal(ones(N-1)), Q)
Qbar = blockdiag(Qbar, Qf)
Rbar = kron(Diagonal(ones(N)), R)

H = 2 * (S' * Qbar * S + Rbar)
g = 2 * S' * Qbar * T * x0

# Solve QP
model = Model(OSQP.Optimizer)
set_silent(model)
@variable(model, U[1:nu*N])
@objective(model, Min, 0.5 * U' * H * U + g' * U)
optimize!(model)

println("Optimal QP cost: ", objective_value(model))

```

4. Riccati Recursion and the Linear Quadratic Regulator (LQR)

4.1 From QP to Riccati Recursion

The QP formulation of the discrete LQR problem leads to a linear system with block-tridiagonal structure.

Instead of solving the full system directly, we can exploit this structure to perform **backward recursion** — the Riccati recursion — which computes the optimal feedback law efficiently.

4.2 Backward Recursion Derivation

Consider the discrete-time dynamics:

$$x_{k+1} = Ax_k + Bu_k,$$

and the quadratic cost function:

$$J = \sum_{k=0}^{N-1} (x_k^\top Q x_k + u_k^\top R u_k) + x_N^\top Q_f x_N.$$

We minimize J subject to the system dynamics.

At the final step ($k = N$):

$$V_N(x_N) = x_N^\top Q_f x_N.$$

Assume that at step $k + 1$ we know the optimal cost-to-go function $V_{k+1}(x_{k+1}) = x_{k+1}^\top P_{k+1} x_{k+1}$.

Then, at step k :

$$V_k(x_k) = \min_{u_k} [x_k^\top Q x_k + u_k^\top R u_k + (Ax_k + Bu_k)^\top P_{k+1} (Ax_k + Bu_k)].$$

Expanding and grouping terms in u_k :

$$V_k(x_k) = x_k^\top (Q + A^\top P_{k+1} A) x_k + 2 u_k^\top (B^\top P_{k+1} A) x_k + u_k^\top (R + B^\top P_{k+1} B) u_k.$$

Minimizing with respect to u_k gives:

$$u_k^* = -K_k x_k, \quad K_k = (R + B^\top P_{k+1} B)^{-1} B^\top P_{k+1} A.$$

Substitute u_k^* back to obtain the Riccati recursion:

$$P_k = Q + A^\top P_{k+1} A - A^\top P_{k+1} B (R + B^\top P_{k+1} B)^{-1} B^\top P_{k+1} A.$$

4.3 Algorithmic Form

For a finite-horizon LQR problem:

1. Initialize
 $P_N = Q_f$.
2. For $k = N - 1, \dots, 0$:
compute K_k and P_k from the equations above.
3. Apply the control law $u_k = -K_k x_k$.

This recursion propagates information backward in time and yields the optimal feedback gain sequence.

4.4 Julia Implementation

```
using LinearAlgebra

function lqr_finite_horizon(A, B, Q, R, Qf, N)
    n, m = size(B)
    P = Vector{Matrix{Float64}}(undef, N+1)
    K = Vector{Matrix{Float64}}(undef, N)
    P[N+1] = Qf
    for k in N:-1:1
        S = R + B' * P[k+1] * B
        F = B' * P[k+1] * A
        K[k] = inv(S) * F
        P[k] = Q + A' * P[k+1] * (A - B * K[k])
    end
    return K, P
end

# Example
A = [1.0 0.1; 0.0 1.0]
B = [0.0; 0.1]
Q = I(2)
R = 0.1 * I(1)
Qf = 10.0 * I(2)
K, P = lqr_finite_horizon(A, B, Q, R, Qf, 30)
println("Optimal K0 = ", K[1])
```